Jennifer Programming Language

A small, batteries-included interpreted programming language, written in Go and shipping as one self-contained static binary (plus a constrained, embeddable TinyGo build).

Made with <3 by mplx <jennifer@mplx.dev>

2026-08-16

Jennifer Programming Language

Jennifer is an interpreted programming language written in (Tiny)Go.

It is **batteries-included** , and not just in name: **41 built-in libraries and 72 distributable modules - over 1,800 functions, constants, and types** cover what real programs actually need, so you build genuine tools, not toys. Talk to a real database with the SQL library (MySQL/MariaDB/Galera and PostgreSQL, parameterized and injection-safe) or map rows to structs with the ORM. Read and write JSON, YAML, TOML, and XML; match text with full regular expressions. The web works both ways: an HTTP/S client and ergonomic REST layer to call out, a built-in HTTP server and web framework to serve. Email is a complete stack - SMTP to send, POP3 / IMAP to receive; caches and stores come through Redis and memcached clients. Security is the real thing, not a stub: AES-256-GCM, Ed25519, HKDF / PBKDF2, and JWT. Wire your program into AI agents with the Model Context Protocol: expose your own functions as tools / resources / prompts to a host (Claude, an IDE, ...), or call another MCP server as a client - validated end to end against the official MCP SDK. And lightweight concurrency is built into the language via `spawn` and the task library. Something more niche - Gotify push, strict SemVer, S3 object storage, vCard / iCalendar, MQTT and AMQP messaging, ACME certificate issuance, TOTP two-factor codes - and lots of others? No need to ask: it is all there already. The interpreter and its built-in libraries ship as **one self-contained static binary** - about 23 MB with everything, ~10 MB for the embeddable build - with no runtime, package manager, or dependency tree to install; the modules ride along as plain `.j` source files (installed beside the binary and resolved through the module path), readable and hackable with no compile step.

It is also a natural fit for **teaching and learning** : an interactive REPL, an easy-to-read grammar, and token and AST dumps that make it ideal for mastering language design, plus a built-in linter and profiler and full test support. Its strict, explicit design - conditions must be `bool` , conversions are spelled out, names never shadow, and errors are positioned - surfaces a mistake as a clear message instead of a silent surprise, so a learner sees exactly what went wrong.

Source files use the `.j` extension. Whitespace is not significant anywhere; statements end with `;` .

```
use io;
use time;

def start as time.Time init time.now();
io.printf("hello, world\n");
def gap as time.Duration init time.sub(time.now(), $start);
io.printf("ran for %d ms\n", time.milliseconds($gap));
```

Two binaries: jennifer and jennifer-tiny

Jennifer ships as two binaries built from the same source:

- **jennifer** - standard Go build, full host-feature surface. This is the default binary you install and reach for.
- **jennifer-tiny** - TinyGo build, smaller and embeddable. Missing `os/exec` and the network stack (TinyGo runtime gaps); calls into those surfaces return a friendly runtime error pointing back at `jennifer` . Use this variant when binary size or embeddability matters (embedded systems, minimal containers, small-footprint scripting hosts).

Write Jennifer with your editor and an AI assistant

Syntax highlighting ships in editors/ (https://github.com/jennifer-language/jennifer/tree/main/editors) (a Vim / Neovim drop-in, a TextMate grammar for VS Code / Sublime / Zed, a highlight.js definition). And because Jennifer is new, we ship `JENNIFER.md` : drop it into your project and point an AI coding assistant at it ("we code in Jennifer, see JENNIFER.md, let's go") so it writes correct `.j` from the start. See Editor & AI support.

What's in this site

- **Getting started** - install the interpreter and run your first program.

- **Editor & AI support** - highlighting and the drop-in `JENNIFER.md` for AI-assisted coding.
- **Language reference** - syntax, types, methods, control flow, imports, style.
- **Libraries** - per-library reference plus an alphabetical cheatsheet of every builtin.
- **Modules** - the Jennifer-coded, importable module ecosystem (`import "name.j";`): formats, mail, stores, web, and protocol clients, each with its own reference page.
- **Technical reference** - implementation details for the lexer, parser, interpreter, and CLI.
- **Project** - milestones, design stances, glossary.

Status

Pre-1.0. The major version stays at `0.x.y` while the language is still finding its shape; breaking changes can happen at milestone boundaries and are called out in docs/milestones.md. Once Jennifer reaches `1.0.0` , semver applies and breaking changes need a major bump.

Source

Source, issues, and pull requests live at https://github.com/jennifer-language/jennifer .

License: LGPL-3.0-only.

Manual

> **Download the whole manual as a single PDF (jennifer-manual.pdf).** It ships with a clickable outline (the H1/H2/H3 headings) in your PDF reader's bookmark panel, so you can jump straight to any section. The entire documentation in one file, regenerated from these pages on every build - handy for reading or searching offline.

Getting started

Installing & running

Which binary?

On **Linux** (the supported platform) Jennifer ships as two binaries. Same source, same language; only the compiler differs. Pick by use case:

Binary	Build	Pick when
jennifer	standard Go (default)	What most users want. Full host-feature surface; competitive on compute-heavy work (the two builds are now within ~1.5x either way per workload on the serial benchmark, and the Go binary wins the end-to-end wall clock once spawn parallelism is involved; see technical/benchmark.md) and the reliable choice for multi-core parallel spawn. Required for os.run / os.spawn / os.wait / os.poll / os.kill and the whole net library.
jennifer-tiny	TinyGo	Constrained variant. Smaller binary, embeddable in minimal-footprint deployments (embedded systems, minimal containers, small-footprint scripting hosts). Missing os/exec (TinyGo runtime gap) and the network stack (no netdev driver). Also run-only: the tokens / ast / fmt / lint / profile / test development subcommands live only in the default binary. Calls into any of these surfaces return a friendly error pointing back at jennifer.

Both binaries install side by side and never overlap. The packaged distributions below install both; for tarball or from-source builds you get both binaries in one go too. (The best-effort **macOS / Windows** builds ship the standard `jennifer` only - see macOS / Windows below.)

Install

Debian / Ubuntu (.deb)

Pick the right `.deb` for your architecture from the latest Releases page (https://github.com/jennifer-language/jennifer/releases), verify the checksum, and install:

```
# Replace X.Y.Z with the release version, e.g. 0.14.0
ARCH=$(dpkg --print-architecture)   # amd64 or arm64
curl -LO "https://github.com/jennifer-language/jennifer/releases/download/X.Y.Z/
jennifer_X.Y.Z_${ARCH}.deb"
curl -LO "https://github.com/jennifer-language/jennifer/releases/download/X.Y.Z/
jennifer_X.Y.Z_${ARCH}.deb.sha256"
sha256sum -c "jennifer_X.Y.Z_${ARCH}.deb.sha256"
sudo dpkg -i "jennifer_X.Y.Z_${ARCH}.deb"
```

Installs `/usr/bin/jennifer` + `/usr/bin/jennifer-tiny` , man pages under `/usr/share/man/man1/` , bash completion, the XDG MIME definition that registers `.j` as `text/x-jennifer` with file managers and editors, and Vim + Neovim syntax highlighting (dropped in both `/usr/share/vim/vimfiles` and `/usr/share/nvim/site` so `.j` files highlight with no per-user setup in either editor). A Sublime Text / `bat` (https://github.com/sharkdp/bat) syntax also ships at `/usr/share/jennifer/syntaxes/jennifer.sublime-syntax` ; `bat` needs a one-time activation (copy it into `$(bat --config-dir)/syntaxes/` and run `bat cache --build`) since it compiles syntaxes into a per-user cache.

Arch Linux (AUR)

Two packages, take whichever fits:

```
# Prebuilt binary, downloads the release tarball (fast install):
yay -S jennifer-bin
# or paru -S jennifer-bin, or any other AUR helper.

# Builds from source on each install, tracks main:
yay -S jennifer-git
```

Both install the same set of files as the `.deb` . The `jennifer-bin` package is on par with each release; `jennifer-git` rebuilds against the latest commit on `main` whenever you ask your AUR helper to upgrade.

Linux (tarball)

For distros without a native package, grab the per-arch tarball from the Releases page (https://github.com/jennifer-language/jennifer/releases):

```
# Replace X.Y.Z and ARCH with the release version + your arch
curl -LO "https://github.com/jennifer-language/jennifer/releases/download/X.Y.Z/
jennifer-X.Y.Z-linux-${ARCH}.tar.gz"
tar -xzf "jennifer-X.Y.Z-linux-${ARCH}.tar.gz"
cd "jennifer-X.Y.Z-linux-${ARCH}"
./jennifer version
./jennifer-tiny version
```

The tarball lays out as:

```
jennifer-X.Y.Z-linux-ARCH/
+-- jennifer               # standard-Go binary (default)
+-- jennifer-tiny          # TinyGo binary (constrained)
+-- README.md
+-- JENNIFER.md
+-- share/
    +-- man/man1/          # jennifer.1, jennifer-tiny.1
    +-- mime/packages/     # jennifer.xml (XDG MIME)
    +-- bash-completion/   # completions/jennifer (+ jennifer-tiny symlink)
    +-- vim/vimfiles/      # syntax/ + ftdetect/ (Vim highlighting)
    +-- nvim/site/         # syntax/ + ftdetect/ (Neovim highlighting)
    +-- jennifer/
        +-- modules/       # importable .j modules (import "name.j";)
        +-- syntaxes/      # jennifer.sublime-syntax (Sublime Text / bat)
```

To install system-wide:

```
sudo install -m 0755 jennifer      /usr/local/bin/
sudo install -m 0755 jennifer-tiny /usr/local/bin/
sudo install -m 0644 share/mime/packages/jennifer.xml /usr/local/share/mime/
packages/
sudo update-mime-database /usr/local/share/mime || true
# System modules, so a bare `import "name.j";` resolves. /usr/share/jennifer/
modules
# is the built-in default; install elsewhere and point JENNIFER_SYSMODDIR (or
# --sysmoddir) at it instead. To print the path in effect:
#   echo 'use io; use meta; io.printf("%s\n", meta.SYSMODDIR);' | jennifer run -
sudo mkdir -p /usr/share/jennifer/modules
sudo install -m 0644 share/jennifer/modules/*.j /usr/share/jennifer/modules/
```

Container image

Each release publishes multi-arch (`linux/amd64` + `linux/arm64`) images to GHCR. The image bundles

both binaries and the system modules, so a bare `import "name.j";` resolves with no setup. The container sets `WORKDIR /work` and `ENTRYPOINT ["jennifer"]` , so you mount your code at `/work` and pass the `jennifer` subcommand as the container command.

Two variants: `:latest` / `:<version>` is the Debian-slim default (full host features - `os.run` , TLS); `:static` / `:<version>-static` is a minimal distroless build (~15-25MB, no `/bin/sh` , so `os.run` of external programs is unavailable - use it for pure-interpreter or web-serving workloads). `:latest` and the version tags track releases; a **:dev** (also **:main**) slim image is rebuilt from the latest commit on `main` on every push - a moving tag for trying the bleeding edge, always overwriting the previous one. Build details and local-build recipes are in `packaging/docker/README.md` .

Docker

```
# Run a script from the current directory (mounted at /work).
docker run --rm -v "$PWD:/work" ghcr.io/jennifer-language/jennifer run app.j

# Pipe a program on stdin.
echo 'use io; io.printf("hi\n");' | docker run --rm -i ghcr.io/jennifer-language/
jennifer run -

# Interactive REPL (the default command).
docker run --rm -it ghcr.io/jennifer-language/jennifer

# Serve a web app (port is whatever the program passes to web.run / httpd).
docker run --rm -p 8080:8080 -v "$PWD:/work" ghcr.io/jennifer-language/jennifer
run server.j
```

For a long-running service, a Compose file keeps the mount, port, and restart policy in one place:

```
# compose.yaml  (or docker-compose.yml)
services:
  app:
    image: ghcr.io/jennifer-language/jennifer:latest
    command: ["run", "server.j"]   # args to the `jennifer` entrypoint
    volumes:
      - ./:/work                   # your code, resolved from the image's /work
workdir
    ports:
      - "8080:8080"                # match the port your program serves on
    restart: unless-stopped
```

```
docker compose up        # start (add -d to run detached)
```

Podman

Podman is a daemonless, rootless-capable drop-in: the same image and flags work, just swap `docker` for `podman` .

```
# Run a script (add :Z to the bind mount on SELinux hosts so the container can
read it).
podman run --rm -v "$PWD:/work:Z" ghcr.io/jennifer-language/jennifer run app.j

# Interactive REPL.
podman run --rm -it ghcr.io/jennifer-language/jennifer

# Serve a web app.
podman run --rm -p 8080:8080 -v "$PWD:/work:Z" ghcr.io/jennifer-language/jennifer
run server.j
```

`podman compose up` reads the same Compose file shown above. To run it as a managed background service, generate a systemd/Quadlet unit with `podman generate systemd` (or a `.container` Quadlet file). Rootless Podman can't bind host ports below 1024 without extra configuration - map to a high port (`-p 8080:8080`) and reverse-proxy if you need `:80` / `:443` .

macOS / Windows (unsupported)

Linux is the only supported platform. As a convenience, best-effort **unsupported** binaries for macOS (Intel + Apple Silicon) and Windows (64- and 32-bit) are attached to each release (https://github.com/jennifer-language/jennifer/releases), named `...-UNSUPPORTED` . Read the caveats before relying on them:

- **Best-effort, may be absent.** They come from a pipeline step that is allowed to fail; if a build breaks, that release simply won't have them, and it does not hold up the Linux release.
- **Standard jennifer only.** No `jennifer-tiny` - TinyGo's macOS / Windows host support is too limited to ship. This is the full-featured build, so `os.run` / `os.spawn` , the `net` library, and the rest of the surface all work.
- **Unsigned.** On macOS, Gatekeeper quarantines the download - clear it with `xattr -d com.apple.quarantine ./jennifer` (or right-click -> Open). On Windows, SmartScreen warns about an unknown publisher - choose "More info" -> "Run anyway".
- **No support.** Bugs specific to macOS / Windows may not be fixed; supported development and testing happen on Linux. Fully supported builds for these platforms are separate future work (see milestones.md).
- **Just the binary (macOS).** The `-UNSUPPORTED` archive holds only the executable plus `JENNIFER.md` , `README.md` , and the licence - no installer, man pages, MIME registration, or shell completion. For a nicer macOS install, use the Homebrew tap (below); Windows gets an installer (further below).

macOS: Homebrew (recommended)

The lowest-friction way onto macOS is the Homebrew tap, which builds `jennifer` **from source** - so it runs on both Intel and Apple Silicon with **no Gatekeeper prompt** (Homebrew does not quarantine what it builds), puts `jennifer` on your `PATH` , and bundles the system modules so a bare `import "name.j";` resolves with no setup:

```
brew install jennifer-language/tap/jennifer
```

`brew install --HEAD jennifer-language/tap/jennifer` builds from `main` . It is still a best-effort **unsupported** build (Linux is the only supported platform), and installs the standard `jennifer` only (no `jennifer-tiny`). The plain `-UNSUPPORTED` tarball above stays available for anyone who wants just the binary. See `packaging/homebrew/` .

Windows installer

Windows releases also ship a `jennifer-<version>-setup.exe` - the same best-effort **unsupported** build wrapped in an Inno Setup (https://jrsoftware.org/isinfo.php) installer. It is still unsigned (SmartScreen: "More info" -> "Run anyway") and still unsupported, but it saves the manual setup:

- Offers a choice at startup: **Install for all users** (elevates, installs to `C:\Program Files\Jennifer` , system-wide `PATH` / env) or **Install for me only** (no admin, `%LOCALAPPDATA%\Programs\Jennifer` ,

per-user). Running the setup as administrator gets the all-users / Program Files install. Either way it adds `jennifer.exe` to `PATH`, so `jennifer` works in a fresh terminal.
- Bundles the Jennifer-coded system modules and sets `JENNIFER_SYSMODDIR`, so a bare `import "name.j";` resolves (on Windows the built-in module path is a Unix path that does not exist, so the plain `.zip` cannot import modules without setting this yourself).
- Optionally associates `.j` files (opt-in): double-click opens the source in Notepad; a "Run with Jennifer" right-click action runs it.
- Uninstall from **Apps & Features**, which reverses the `PATH`, the environment variable, and the association.

Prefer the plain `-UNSUPPORTED.zip` if you want a portable, no-registry copy (or you are on 32-bit Windows - the installer is 64-bit only); set `JENNIFER_SYSMODDIR` yourself to use modules from the zip.

Windows 8.1 and earlier are not possible: this project's Go toolchain (Go 1.21+) produces binaries that require **Windows 10 or newer** (or Windows Server 2016+). Go discontinued support for older releases in Go 1.21, so Windows 7, 8, and 8.1 - as well as Vista and XP - are all excluded, not just XP. The 32-bit build targets 32-bit Windows 10 / 11.

Build from source

For development, or any platform without a prebuilt artifact. You need a working TinyGo (https://tinygo.org/) toolchain plus standard Go. From the repository root:

```
# Build both binaries:
make build

# Or just one:
make build-go      # produces ./jennifer      (standard Go, default)
make build-tinygo  # produces ./jennifer-tiny (TinyGo, constrained)

# Quick iteration without rebuilding:
go run ./cmd/jennifer run examples/hello.j
```

The `make` targets regenerate `internal/version/version_gen.go` from git state before invoking the toolchain, so `./jennifer version` always reflects the current commit. See ../libraries/meta.md for the `meta.VERSION` string format.

Running

```
# Run a Jennifer source file (.j extension required):
jennifer run examples/hello.j

# Print the build version:
jennifer version
```

You can also pipe source in on stdin by passing `-` as the filename:

```
echo 'use io; io.printf("hi\n");' | jennifer run -
jennifer run - < program.j
cat program.j | jennifer run -
```

When reading from stdin, error messages identify the source as `<stdin>` and file imports (`include "name.j";`) resolve relative to the current working directory.

Interactive REPL

For experimenting with the language, start an interactive session with `jennifer repl`:

```
$ jennifer repl
jennifer - Jennifer programming language interpreter
type :quit (or Ctrl-D) to exit; :help for help
>>> use io;
>>> def x as int init 21;
>>> $x + $x;
42
>>> io.printf("hi\n");
hi
>>> func dbl(n as int) {
...   return $n * 2;
... }
>>> dbl(7);
14
>>> :quit
```

A few notes:

- Statements still end with `;` . If a line ends with an unclosed `{` or `(` , the prompt switches to `...` and waits for you to finish the block.
- A bare expression at the end of an input (like `$x + $x;`) prints its value. `null` results (including the return value of `printf`) are suppressed.
- String results are printed with surrounding double quotes so they're distinguishable from numbers (`"hello"` , not `hello`).
- Variables, constants, methods, and library imports persist for the whole session. Methods can be redefined freely as you iterate.
- File splices (`include "lib.j";`) work in the REPL and resolve relative to the directory you launched `jennifer repl` from.
- `:quit` , `:exit` , or Ctrl-D end the session; `:help` shows a reminder.

The prompt supports the standard line-editing keys you'd expect from a modern shell:

Key	Action
Left / Right	Move cursor
Home / End	Jump to line start / end
Ctrl+A / Ctrl+E	Same as Home / End
Ctrl+Left / Ctrl+Right	Move by word
Backspace, Delete	Delete character
Ctrl+W, Ctrl+Backspace	Delete word backward
Ctrl+U / Ctrl+K	Kill to line start / end
Up / Down	Browse history
Ctrl+C	Cancel the current line

History is in-memory only (no on-disk persistence yet) and holds up to 100 entries. When stdin is piped (e.g. `echo ... | jennifer repl` in a test harness) the editor is bypassed and the REPL reads lines normally, so non-interactive uses keep working.

Inspection and formatting

Three commands help you see what Jennifer is doing under the hood and keep your source in canonical shape:

```
# Print the lexer's token stream, one per line
jennifer tokens examples/hello.j

# Print the parsed (and preprocessed) AST as JSON
jennifer ast examples/hello.j

# Reformat the source to canonical style (see style-guide.md)
jennifer fmt examples/hello.j
```

By default `fmt` writes the formatted source to stdout. Pass `-w` (or `--write`) to rewrite files in place instead - one or more named files (`jennifer fmt -w foo.j` , `jennifer fmt -w *.j`); an already-canonical file is left untouched. `fmt` formats the files you name and does no globbing itself - let the shell select files, recursively via `**` (see the formatter reference). The formatter is idempotent (`fmt` of `fmt` output equals `fmt` output) and preserves runtime behavior - every example in this repo is checked both ways by the test suite. See style-guide.md for the full style rules.

Your first program

Save the following as `hello.j` :

```
# hello.j
use io;

def x as int init 21;
io.printf($x + $x);
```

Run it:

```
./jennifer run hello.j
```

You should see `42` .

What just happened

1. `use io;` makes Jennifer's standard library functions available.
2. `def x as int init 21;` declares an integer variable named `x` and initializes it to `21` . Notice that **using** a variable requires the `$` prefix.
3. `io.printf($x + $x)` calls the standard library function with the result of `21` + `21` .

Top-level statements run in source order - there is no required entry-point method. You can still group reusable code into methods (`def`) and call them explicitly.

Editor support and AI-assisted coding

Two things make writing Jennifer outside this repo comfortable: syntax highlighting in your editor, and a drop-in language reference so an AI coding assistant can write correct Jennifer for you. Both ship in the repository.

Editor syntax highlighting

Highlighting definitions live in `editors/` (https://github.com/jennifer-language/jennifer/tree/main/editors). Jennifer's lexical rules are regular enough that highlighting is genuinely accurate - `$x` is always a variable, `UPPER_CASE` a constant, `NS.name` a namespaced call, `#` and `/* */` comments.

- **Vim / Neovim** - a true drop-in. Copy `editors/vim/syntax/jennifer.vim` (https://github.com/jennifer-language/jennifer/blob/main/editors/vim/syntax/jennifer.vim) and `editors/vim/ftdetect/jennifer.vim` (https://github.com/jennifer-language/jennifer/blob/main/editors/vim/ftdetect/jennifer.vim) into `~/.vim/` (or `~/.config/nvim/`); `.j` files are detected automatically.
- **VS Code / Sublime Text / Zed** - use the TextMate grammar `editors/textmate/jennifer.tmLanguage.json` (https://github.com/jennifer-language/jennifer/blob/main/editors/textmate/jennifer.tmLanguage.json) (scope `source.jennifer`) from a thin language extension.
- **bat / Sublime Text** - the native `editors/sublime/jennifer.sublime-syntax` (https://github.com/jennifer-language/jennifer/blob/main/editors/sublime/jennifer.sublime-syntax). For `bat` , copy it into `$(bat --config-dir)/syntaxes/` and run `bat cache --build` (it caches syntaxes per user, so a system path can't auto-activate it).
- **Static sites / blogs** - the highlight.js definition (https://github.com/jennifer-language/jennifer/blob/main/editors/highlightjs/jennifer.js) registers a `jennifer` language.

Per-editor install steps are in `editors/README.md` .

One caveat: GitHub's Linguist assigns the `.j` extension to Objective-J, so GitHub's web UI will not highlight Jennifer source as Jennifer. That is a GitHub-side limitation; local editors and self-hosted sites are unaffected.

Jennifer as a shell filter

`jennifer run -` reads a program from stdin, so Jennifer slots into a pipe like any other filter. A handy one is a `json-pretty` that reformats JSON flowing through it. Save the program to a file (say `~/.local/share/jennifer/json-pretty.j`):

```
use json;
use io;

def src as string init "";
while (not io.eof()) {
    $src = $src + io.readLine() + "\n";
}
io.printf("%s\n", json.encodePretty(json.decode($src)));
```

then alias it:

```
alias json-pretty='jennifer run ~/.local/share/jennifer/json-pretty.j'

echo '{"b":2,"a":1}' | json-pretty
curl -s https://api.example.com/thing | json-pretty
```

Swap `json` for any other decode / re-encode pair to get, for example, a `pretty-xml` . A no-file variant that pipes the program itself through `jennifer run -` is in the CLI reference.

Run profiles (--env)

`jennifer run --env=<profile> app.j` selects a **run profile** for a program - `prod` , `dev` , `staging` , or whatever names you use. It is pure sugar: the flag sets the `JENNIFER_ENV` environment variable before the program runs, so

```
jennifer run --env=prod app.j
```

is identical to

```
JENNIFER_ENV=prod jennifer run app.j
```

The program reads the profile from that one environment variable; nothing about the language changes. The main consumer is the `dotenv` module, whose `autoload` / cascade loaders pick the `.env.<profile>` file to layer on top of `.env` - so `--env=prod` loads `.env.prod` , `--env=dev` loads `.env.dev` , and no profile loads only the base files.

The profile label is validated (`[A-Za-z0-9_-]` , 1 to 64 characters), which is also what stops a hostile value from steering `.env.<profile>` at another path. The flag goes **before** the script file (like `-I` and `--sysmoddir`); anything after the file is passed to the program as its own arguments. An explicit `--env=X` overrides any `JENNIFER_ENV` already in the environment.

AI-assisted coding with JENNIFER.md

Jennifer is new and small, so a general-purpose AI assistant has no built-in knowledge of it and will otherwise guess (usually Python-with-dollar-signs). `JENNIFER.md` is a single, self-contained language reference written for exactly this: drop it into your project and point your assistant at it.

```
We're coding in Jennifer, a batteries-included interpreted language. Read
JENNIFER.md for the syntax and standard library, then let's build ...
```

It covers the lexical rules (the $ sigil, letters-only identifiers, `UPPER_CASE` constants), types, operators (including `/` being float division), control flow, methods, concurrency, imports, the namespaced standard library, and a checklist of the mistakes an assistant most often makes. It describes the *language* , not the interpreter internals, and stays in sync with this spec.

It doubles as a human quick-reference. For the exhaustive per-function detail behind it, see the library reference and cheatsheet.

Language reference

Jennifer - User Guide

Jennifer is an interpreted programming language with a rich type system (structs, enums, value semantics), lightweight concurrency, a broad topic-based standard library, and a large ecosystem of Jennifer-coded modules.

This guide covers everything you can do in Jennifer. Run `jennifer version` to see which build you're on; the language history lives in docs/milestones.md.

Design stances

A handful of decisions shape every feature in Jennifer. Read them before the topical chapters - they explain why the language looks the way it does and rule out the "but why don't you just..." reflex.

See docs/design-stances.md for the full seven-stance table.

Contents

- Installing & running - building the binary, running a program, the interactive REPL, the inspection and formatting commands.
- Your first program - a one-screen `hello.j` and a walkthrough of what each line does.
- Editor & AI support - syntax highlighting for your editor, and the drop-in `JENNIFER.md` that lets an AI assistant write Jennifer.
- Syntax - tokens, comments, identifier rules.
- Types and values - the primitive types, variables and constants, scoping rules, and the compound types `list` and `map` .
- Methods - declaring methods, parameters, return values, recursion, hoisting, the no-shadowing rule for builtins.
- Control flow - operators, precedence, `if` / `while` / `for` / for-each.
- Imports - `use LIB;` for libraries, `import "file.j";` for source files, the library catalog.
- Examples - small programs that exercise the language end-to-end.
- Style guide - the canonical source style that `jennifer fmt` produces; the spacing/indentation rules, the names convention, the `[]` and `{}` literal layout.
- Best practices - stylistic guidance and code-smell heuristics for writing Jennifer that ages well. Not enforced by the language; click through if you want the "why".
- Performance - the few patterns that turn a quadratic hot path into a linear one (append-and-join over + in a loop, hoisting per-element calls out of hot loops), and what the interpreter already optimizes for you.

Related

- docs/libraries/ - one reference page per standard library (`io` , `convert` , `math` , `strings` , ...).
- docs/modules/ - the Jennifer-coded modules that ship with the interpreter, brought in with `import` (`ansi` , `csv` , `mime` , `redis` , ...).
- docs/technical/ - interpreter internals for contributors.
- docs/glossary.md - canonical project terminology. When two words could plausibly name the same concept (function vs method, library vs module, list vs array), this page picks the one the project uses everywhere.
- docs/milestones.md - what's shipped, what's planned.

Syntax

Tokens and whitespace

Whitespace (spaces, tabs, newlines) is not significant anywhere in Jennifer. The lexer skips it between tokens and never reads it from inside one. Statements are terminated by `;` , blocks by matched `{` / `}` - never by indentation or line breaks. The same program can be written across many lines or jammed onto one, and it parses the same:

```
# canonical form
use io;
def x as int init 21;
io.printf("%d\n", $x + $x);
```

```
# all three statements on one line - same program
use io; def x as int init 21; io.printf("%d\n", $x + $x);
```

```
# split across many lines - also the same program
use   io
;
def
    x
        as
            int
                init
                    21 ;
printf (
    "%d\n"
    ,
    $x
        +
            $x
)
;
```

The same flexibility applies to qualified (namespaced) calls. All three of these print the host OS and are accepted by the parser:

```
use io;
use os;

io.printf(os.JENNIFER_OS);            # canonical - tight everywhere
io.printf( os.JENNIFER_OS );          # ugly - spaces inside the call parens
printf   (   os     . JENNIFER_OS  );   # uglier - spaces around `.` too
```

The third form parses fine because the `.` between `os` and `JENNIFER_OS` is just another token boundary - the lexer doesn't care that there are spaces around it. **jennifer fmt rewrites all three into the canonical form** , so you'll only ever see the first one after a format pass. The style guide makes "no space around `.` " explicit, but it's a style rule, not a syntax rule.

A few practical consequences worth knowing up front:

- `jennifer fmt` is the enforcement layer for style. The style guide describes the canonical shape (one space around binary operators, no space inside `(` / `[` / `{` , 1TBS braces, tight `.` in qualified calls, ...) and `fmt` re-emits any well-formed source in exactly that shape. You're never *required* to write canonical form; you just won't see anything else after `fmt` .
- The first line may be a `#!` shebang because # starts a line comment; everything from # to the next `\n` is whitespace as far as the parser is concerned.

- **Inside a string literal, whitespace is literal content.** A space or tab between the quotes is part of the string value; an actual newline between the quotes is a literal newline in the value (though the conventional spelling is the `\n` escape - multi-line literals work but aren't the canonical form).

Indentation and blank lines never change the meaning of a program; they only change how it reads.

Comments

```
# line comment - runs to end of line

/* block comment -
   can span multiple lines */
```

Block comments don't nest. Because # starts a line comment, the first line of a script may be a Unix shebang and the interpreter will skip it:

```
#!/usr/bin/env -S jennifer run
use io;
io.printf("hi\n");
```

(`env -S` splits the rest of the line into arguments, which is how `jennifer run` reaches the interpreter on Linux.)

Source files normally must end in `.j` , but a file with a `#!` shebang first line is exempt: `chmod +x` it and install it under a bare command name (`/usr/local/bin/mytool` , no extension) and it runs as a normal command. Only the entry point is exempt - files it `import` s or `include` s still need `.j` .

Number literals

Decimal:

```
42
1_000_000           # `_` is a visual digit separator
3.14
1_000.000_5         # the mantissa side of a float accepts `_` too
```

Non-decimal integer prefixes:

```
0xff                # hex
0xDEAD_BEEF
0o755               # octal
0b1010_0110         # binary
```

All four bases produce ordinary `int` values - same kind, same operators. The _ separator is allowed between digits but never adjacent to the prefix, adjacent to another _ , or at the start / end of the digit run (`0x_ff` , `1__000` , `100_` are all lex errors).

Identifiers

- Variable, method, parameter and library names are letters only: `[A-Za-z]` , up to 64 characters. No digits or underscores.
- **Constants** are uppercase chunks joined by single _ separators: `[A-Z]+(_[A-Z]+)*` , up to 64 characters. Every _ must be immediately followed by another uppercase letter. `MAX` , `MAX_RETRIES` , `HTTP_OK` , and `A_B_C` are all legal; `_MAX` , `MAX_` , and `MAX__INT` are not.
- **Variable references** use a leading $: define `name` , refer to it as `$name` .
- **Constant references** are bare (no $).
- **Method calls** are bare and followed by `(...)` .

Types and values

Types

Type	Example literals	Default	Notes
int	42, 0xff, 0o755, 0b1010_0110, 1_000	0	64-bit signed; _ may separate digits
float	3.14, 0.5, 6.022e23, 1_000.000_5	0.0	64-bit; scientific notation with an e/E exponent (1e10, 1.6e-19); promoted from int in mixed math
string	"cooked \n", 'raw \d+'	""	"..." processes escapes + interpolates {expr}; '...' is raw (verbatim, multi-line, no interpolation)
bool	true, false	false	Produced by comparison operators
null	null	null	A type with a single value (the unit)
bytes	(no literal)	empty	Mutable byte sequence; element = int in [0, 255]; built via convert.bytesFromString or grown with $b[] = byte;
list of T	[1, 2, 3]	[]	Ordered sequence; 0-indexed; mutable
map of K to V	{"a": 1, "b": 2}	{}	Key->value; insertion-ordered; mutable
user struct	Point{x: 1, y: 2} (after def struct Point ...;)	every field zero	Named fixed set of typed fields; see Structs
user enum	Shape.Circle{r: 2.0} (after def enum Shape ...;)	first variant, payload zeroed	One of a fixed set of variants (a sum type); consumed with match; see Enums
task of T	(no literal - produced by spawn { ... })	(cannot be defaulted; must be initialised)	Handle to a concurrent computation; observed via the task library. See Concurrency
func	a bare method name: greet (after func greet() {...})	null function (calling it errors)	A first-class function value; call via $f(args). See Function values
channel of T	(no literal - produced by channel.make(capacity))	(must be initialised)	A CSP channel between goroutines; a shared conduit that copies the values sent through it. See channel

The **Default** column is the value an uninitialized variable receives (`def x as int;` produces `0`). For compound types the default is an empty container of the declared element / key / value type, not `null` .

Lists and maps are compound types - they hold other Jennifer values. Nesting works: `list of list of int` , `map of string to list of int` , etc. Both are **value-typed** : `$ys = $xs;` makes an independent copy, function parameters bind by copy, and `const` is deep (you cannot mutate the contents of a `const` list or map).

Note: Jennifer's `list` is an array-backed sequence (Go slice underneath), not a Lisp linked list. You get O(1) random access via `$xs[i]` , but no O(1) prepend.

Numeric literals

`int` literals come in four bases - decimal `42` , hex `0xff` , octal `0o755` , and binary `0b1010_0110` - and a `_` may separate digits in any of them for readability (`1_000_000` , `0xDEAD_BEEF`).

`float` literals are written with a decimal point (`3.14` , `0.5`) or in **scientific notation** : a mantissa, an `e` or `E` , and an optional signed exponent (`6.022e23` , `1.6e-19` , `2.5E8`). The exponent makes a literal a `float` even with no fractional part, so `1e10` is the float `10000000000.0` , not an `int` . The mantissa accepts `_` separators (`1_000.5e3`); the exponent does not, and is always decimal - in `0xe5` the `e` stays a hex digit, so that is an `int` , not a float.

Numeric literals are strict, matching the rest of the language: an exponent that overflows to infinity (`1e400`) is a positioned error rather than a silent `Infinity` , while one that underflows below the smallest representable value (`1e-400`) rounds to a finite `0.0` .

Arithmetic is strict the same way. Integer arithmetic that overflows `int64` is a positioned runtime error, not a silent wraparound (`9223372036854775807 + 1` errors, and so does negating the most negative int). Float arithmetic that overflows to a non-finite value is an error too: `1e308 * 10.0` raises `float overflow` , never yielding `+Inf` or `NaN` . So no computation puts an `Infinity` or `NaN` into a value - the strict boundary is the operator itself, not just the literal.

String literals: cooked ("...") and raw ('...')

The two delimiters do different jobs.

"..." is cooked - escape sequences are processed:

Escape	Meaning
\n	newline
\r	carriage return
\t	tab
\\	backslash
\"	double quote
\'	single quote
\0	null character
\uXXXX	A Unicode code point, exactly 4 hex digits (the Basic Multilingual Plane, U+0000-U+FFFF).
\UXXXXXXXX	Unicode code point, exactly 8 hex digits (any plane), e.g. "\U0001F600" is ?

The `\u` / `\U` forms are strict: a surrogate (`\uD800` - `\uDFFF`), a value above `U+10FFFF` , or the wrong number of hex digits is a positioned lex error, not a silent replacement character - the same canonical contract as `convert.fromCodepoint` . Use `\u` for the common case and `\U` for astral code points (emoji, rarer scripts). The brace-free forms deliberately leave `{...}` free for a future interpolation syntax.

A Jennifer `string` is a byte string, so a literal may contain control characters: `\0` (as above) and any other control byte via `\uXXXX` in the cooked form, or written **verbatim** in a raw literal (a raw `'...'` copies every byte, including a literal tab, newline, or NUL). These are legal and preserved in memory; the boundary layers reject them where they'd be unsafe - `fs` / `os` refuse a NUL in a path or argument, and the network modules reject control characters at the wire boundary.

'...' is raw - there is **no** escape processing. Every byte from the opening `'` to the next `'` is content verbatim, backslashes and newlines included. So `'\d+\.\d+'` is the eight-character string `\d+\.\d+` (not a broken escape), and a multi-line block is just a single-quoted literal that spans newlines - a free heredoc, no `<<<EOF` needed:

```
def sql as string init 'SELECT *
FROM users
WHERE name = ?';                # three lines, verbatim

def pattern as string init '\d{4}-\d{2}-\d{2}';   # a regex, no double-escaping
```

Raw literals shine for regexes, Windows-style paths, and embedded JSON, where the cooked form would need `\\` everywhere. To put a **single quote inside** a string, use the cooked form (`"it's"` or `"it\'s"`) - a raw literal ends at the first `'` , so it cannot contain one. There is deliberately no `r"..."` prefix; the delimiter *is* the mode.

String interpolation ({expr} in cooked strings)

A **cooked** `"..."` string interpolates: each unescaped `{expr}` is a slot - one Jennifer **expression** evaluated in the current scope and stringified in place, so a value sits beside its label without `sprintf` verb / argument matching:

```
def name as string init "Ada";
def n as int init 41;
def xs as list of int init [1, 2, 3];

io.printf("hello {$name}, next is {$n + 1}\n");   # hello Ada, next is 42
io.printf("list {$xs}, first {$xs[0]}\n");        # list [1, 2, 3], first 1
```

A slot may be any single expression - a variable, a constant (`{MAX}`), a field or index access (`{$p.x}` , `{$xs[0]}`), arithmetic (`{$a + $b}`), or a call (`{strings.upper($s)}`). Stringification is the `convert.toString` form, so every value kind interpolates and **no use convert is required** . The rules:

- A slot is **one expression** , not a statement: a `;` , `def` , `if` , `return` , or assignment in a slot is a parse error, and an empty `{}` is an error.
- A **literal brace** is written `\{` / `\}` (backslash, matching `\n` / `\"`); a bare unescaped `}` is a lex error that points you at `\}` .
- A **raw** `'...'` string never interpolates - it is the literal form. `'{$x}'` is the four-character string `{$x}` , and `'{"port": 8080}'` is literal JSON.
- There is no `f"..."` prefix - the cooked / raw split already *is* the opt-in.

A slot is real code, so an undefined variable inside one (`"{$typo}"`) is caught at parse time, just like any reference. **Keep slots simple** - variables, field / index access, and arithmetic; a side-effecting or expensive call reads better pulled out to a variable first (the `lint L204` check flags a call in a slot). To build a string with a literal brace *and* no interpolation, prefer a raw string.

Variables and constants

```
def name as int init 5;           # declare and initialize
def count as int;                 # declare with the zero value of int (0)
def const MAX as int init 100;    # constant: uppercase name, init required
```

Uninitialized variables get the **default value** of their declared type (see the Types table).

init accepts any expression of the declared type , not just literals. Arithmetic, comparisons, function calls, and index reads all work as long as the result kind matches:

```
def half as float init 5 / 2;               # 2.5 (arithmetic)
def isZero as bool init 1 == 0;             # false (comparison)
def winner as string init decide($a, $b);   # whatever decide() returns
def first as int init $xs[0];               # element read
```

The same goes for `def const NAME` - the `init` expression is evaluated once at declaration time and the result is frozen.

At the def site, names are bare identifiers (no $). The $ sigil is reserved for use-site references that read or assign a variable. So:

```
def x as int init 5;     # def site - bare name
io.printf($x);              # use site - $ prefix
$x = 42;                 # assignment - $ prefix

def $x as int init 5;    # ERROR: drop the $ here
```

Constants don't use $ anywhere (they're not mutable, so the sigil would have no meaning):

```
def const MAX as int init 100;
io.printf(MAX);            # use site - bare name
MAX = 200;              # ERROR: cannot assign to constant
```

Constant names must be UPPERCASE. The full rule is `[A-Z]+(_[A-Z]+)*` : one or more uppercase chunks joined by single underscores. `MAX` , `MAX_RETRIES` , `HTTP_OK` , and `A_B_C` are all legal; `max` , `Max` , `_MAX` , `MAX_` , and `MAX__INT` are not. The uppercase-only rule is what tells the parser at use sites that a bare identifier is a constant reference, not a variable that forgot its `$` . Constants also **require an init expression** - there is no "declare-then-set" form (`def const X as int;` is rejected).

Assignment uses = :

```
def x as int init 0;
$x = 42;          # ok
$x = "string";    # error: cannot assign string to int variable
```

Function values

A **function value** (type `func`) is a first-class handle to a top-level method. The rule is the one the parser already uses to tell a call from a name: **a bare method name in expression position is the function value; a name followed by (is a call.**

```
func greet(name as string) { return "hi " + $name; }

def f as func init greet;   # `greet` (no parens) is the function value
io.printf("%s\n", $f("ada"));   # call it through the variable -> "hi ada"
```

Function values are ordinary values: pass them to methods, store them in a `list of func` or a struct field, and return them.

```
func applyTo(fn as func, x as int) { return $fn($x); }
func double(n as int) { return $n * 2; }
io.printf("%d\n", applyTo(double, 21));   # 42
```

Call one through any function-valued expression, not just a variable: `$fns0(x)` (a `func` in a list), `makeAdder(1)(2)` (a returned `func`).

Key properties:

- **Immutable.** Holding a function in a value aliases nothing and can never become a write-through handle, so the value-semantics rule is untouched - copying a `func` shares the underlying method, and there is nothing to mutate.
- **Checked at the call site.** The `func` type carries no parameter or return signature; arity and argument kinds are checked when you *call* the value, exactly as for a named method call. Wrong count or wrong kind is a positioned runtime error.
- **The zero is a null function.** `def f as func;` with no `init` is a null handle; calling it is a positioned error until you assign a method.
- **No &name and no closures (yet).** There is no reference sigil (it would read as a pointer the language does not have), and no anonymous-function / closure literal - that is a later, additive step. For now a function value always names a top-level `func` .

The payoff is the higher-order `lists` layer, which takes a `func` per element: `lists.map` / `filter` / `reduce` / `find` / `any` / `all` / `sortBy` .

```
use lists;
func isEven(n as int) { return $n % 2 == 0; }
def evens as list of int init lists.filter([1, 2, 3, 4, 5], isEven);   # [2, 4]
```

Scoping

- Each `{...}` block introduces a new scope.

- A binding is visible from its `def` to the end of the enclosing block, and is inherited by inner blocks.
- Inner scopes can **read** outer bindings but **cannot redefine** a name already in scope (no shadowing). The interpreter rejects shadowing at runtime.
- A `for` loop opens a private scope wrapping `init` / `cond` / `step` /body, so the loop variable does not leak out.
- Constants follow the same scoping rules and reject any later assignment.

Lists and maps

Two compound types let you hold collections of values.

```
use io;

# A list is an ordered, 0-indexed, mutable sequence.
def xs as list of int init [10, 20, 30];
io.printf("%d\n", $xs[0]);          # 10
$xs[1] = 99;                      # index write
io.printf("%d\n", len($xs));        # 3

# A map is a key->value lookup. Iteration is in insertion order.
def m as map of string to int init {"a": 1, "b": 2};
io.printf("%d\n", $m["a"]);         # 1
$m["c"] = 3;                      # adds new key
$m["a"] = 99;                     # updates existing

# Iterate a list's elements, or a map's keys.
for (def x in $xs) { io.printf("%d ", $x); }      io.printf("\n");
for (def k in $m) { io.printf("%s ", $k); }       io.printf("\n");
```

A few rules worth knowing up front:

- **Out-of-bounds list reads and writes are errors** , not silent no-ops. Same for reads of missing map keys - use `maps.has($m, key)` to test for presence first.
- **Lists and maps copy on assignment and on function-call binding.** `$ys = $xs;` makes an independent copy; mutating `$ys[0]` doesn't change `$xs` .
- **const is deep.** `def const NUMS as list of int init [1, 2, 3];` rejects both `$NUMS = ...` and `$NUMS[0] = ...` . Nested const lists/maps follow the same rule transitively.
- **Nesting works** : `list of list of int` , `map of string to list of int` , and so on. See Nested lists and maps below for the shape rules; best practices has guidance on when nesting gets too deep.
- **Empty literals require a declared type** : `[]` and `{}` are valid literals but the surrounding `def x as list of T` decides what they hold.

The $xs[] append sugar

For the common "build a list by appending" pattern, `$xs[] = item;` writes to the position just past the end of the list:

```
def xs as list of int init [];
$xs[] = 10;
$xs[] = 20;
$xs[] = 30;
# $xs is now [10, 20, 30]
```

It's equivalent to `$xs = lists.push($xs, item);` and produces the same result, but the two are not the same performance-wise (see below); use `$xs[]` for building a list, `lists.push` when you want a fresh list and keep the original.

Rules:

- **Write-only.** `$xs[]` is only meaningful as a write target. Any read context (`io.printf($xs[])` , `def y init $xs[] + 1`) is a parse error.
- **Lists and bytes only.** `$m[] = ...;` on a map errors at runtime; maps have no "end-of" position.
- **Type-checked.** The value is checked against the list's declared element type, same as `$xs[i] = item;` .
- **const is still deep.** `$NUMS[] = ...;` on a `def const` list errors with the usual "cannot mutate contents of constant" message.
- **Prefer it in hot loops.** `$xs[]` mutates the list in place through the copy-on-write protocol, so appending N items is amortized O(N). `lists.push` instead returns a *new* list (values are copy-on-assign), so `$xs = lists.push($xs, item)` in a loop copies the whole list each pass - O(N^2) overall. For a few appends the difference is invisible; for a per-element build (a raster, a large buffer, a big result set), reach for `$xs[]` . Reserve `lists.push` for the "give me a new list, leave the original alone" case.

Slicing with a..b

`$xs[a..b]` takes a **half-open** slice `[a, b)` of a list, and returns a **fresh copy** - it includes index a and excludes index b :

```
def xs as list of int init [10, 20, 30, 40, 50];
def mid as list of int init $xs[1..4];   # [20, 30, 40]
$mid[0] = 99;                            # mutating the slice...
io.printf("%d\n", $xs[1]);               # ...leaves the source at 20
```

Either endpoint may be omitted to run to the edge:

```
$xs[2..];    # from index 2 to the end   -> [30, 40, 50]
$xs[..3];    # from the start to index 3 -> [10, 20, 30]
$xs[..];     # a full copy               -> [10, 20, 30, 40, 50]
```

The same `..` slices **bytes** and **strings** too (strings are rune-indexed, so `$s[0..5]` is the first five *characters* , not bytes):

```
def s as string init "hello world";
io.printf("%s\n", $s[0..5]);            # hello
io.printf("%s\n", $s[6..]);             # world
```

Rules:

- **Half-open and int-bounded.** `a..b` is `[a, b)` ; both bounds are int.
- **A copy, never a view.** A slice is value-semantic like any other assignment, so mutating the slice never touches the source (and vice versa).
- **Read-only.** `$xs[a..b] = ...;` is a parse error - because a slice is a copy, a write through it could not reach the original, so the syntax is rejected rather than silently doing nothing.
- **Strict bounds.** `0` <= a <= b <= `len` or it's a positioned runtime error (an out-of-range or inverted slice never clamps silently).
- The same `..` builds a list on its own (`1..5` is `[1, 2, 3, 4]`) and drives a `for` -each loop - see control-flow. For a stepped or reversed range, use `lists.range` .

Nested lists and maps

Compound types nest by repeating the keyword. `list of list of int` is a list whose elements are themselves lists of ints; `map of string to list of int` is a map whose values are lists of ints. There's no depth cap - the parser will recurse as far as you nest.

The "different dimensions, same type" gotcha

Coming from C or Java, you might expect `int[3][3]` to mean "a 3×3 grid - exactly nine ints, fixed shape". **Jennifer does not work that way.**

The declared type only fixes *what each level holds* , not *how many elements are at each level* . So all of these are the same `list of list of int` type:

```
# 2×2 grid - two rows of two columns
def gridA as list of list of int init [[1, 2], [3, 4]];

# 3×3 grid - three rows of three columns
def gridB as list of list of int init [[0, 0, 0], [0, 0, 0], [0, 0, 0]];

# Jagged - rows have different lengths
def gridC as list of list of int init [[1], [2, 3], [4, 5, 6]];

# Empty - zero rows
def gridD as list of list of int init [];
```

Same declared type, four very different shapes. At runtime each list just knows its own length; reading `$gridA[2]` is an out-of-bounds error (only indices 0 and 1 exist), reading `$gridC[2][2]` works (the third row has three elements), but `$gridC[0][2]` is out of bounds (the first row has only one element). **len($gridC[i]) is the only way to ask "how wide is this particular row?"**

If you need a strict shape, enforce it in code:

```
func makeGrid(size as int) {
    def out as list of list of int init [];
    for (def i as int init 0; $i < $size; $i = $i + 1) {
        def row as list of int init [];
        for (def j as int init 0; $j < $size; $j = $j + 1) {
            $row[] = 0;
        }
        $out[] = $row;
    }
    return $out;
}
```

When nesting gets deep enough that you're counting brackets, it's usually time to reach for a struct or another abstraction - see best practices for the heuristics.

Bytes

`bytes` is a **mutable byte sequence** . It looks and acts a lot like a `list of int` , with two important specialisations:

- Each element is constrained to `int` in `[0, 255]` . A write outside that range is a positioned runtime error.
- Indexing returns the byte as an `int` (you can't get a one-byte `bytes` slice via `$b[i]` - it's the integer value of the byte).

```
use io;
use convert;

# Constructing - bytes has no literal form. Either decode a string,
# or start empty and append.
def from_string as bytes init convert.bytesFromString("Hello", "utf-8");
def grown as bytes;
$grown[] = 0x48;
$grown[] = 0x69;

io.printf("from_string: %v\n", $from_string);  # bytes[48 65 6c 6c 6f]
io.printf("grown:       %v\n", $grown);        # bytes[48 69]
io.printf("len:         %d\n", len($from_string));  # 5

# Reading - $b[i] is the byte's value as int.
io.printf("first byte:  %d (= 0x%d|base=16)\n", $from_string[0], $from_string[0]);

# Writing - same int-in-range rule.
$from_string[0] = 0x68;       # lowercase h
io.printf("after edit:  %v\n", $from_string);

# Round-trip back to string.
def s as string init convert.stringFromBytes($from_string, "utf-8");
io.printf("string back: %s\n", $s);
```

Why bytes is its own type (not just list of int)

The range constraint is the point. A `list of int` can hold any 64-bit signed integer; `bytes` can only hold a byte. The runtime enforces this on every write so I/O, hashing, encoding, and crypto code can rely on it. Trying to write `$b[i] = 256;` is a positioned runtime error, not a silent truncation.

Value semantics, just like lists and maps

```
def src as bytes init convert.bytesFromString("Hi", "utf-8");
def dst as bytes init $src;
$dst[0] = 0x78;            # mutates only dst
# $src is still bytes[48 69]
```

Function parameters bind by copy too, so a `func mutate(b as bytes)` that writes into `$b` doesn't leak back to its caller. `const` is deep: `def const B as bytes init ...;` rejects both `$B = ...` and `$B[i] = ...`.

The $b[] = byte; append form

Bytes share the append sugar with lists:

```
def buf as bytes;
$buf[] = 0x48;
$buf[] = 0x69;
# buf is now bytes[48 69]
```

The byte you append must be an `int` in `[0, 255]`.

Codecs and rune vs byte counts

- `convert.bytesFromString(s, codec)` and `convert.stringFromBytes(b, codec)` are the canonical bridges. These two handle `"utf-8"` only; every other character encoding lives in the `encoding` library.
- `stringFromBytes` is **strict at boundaries** : invalid UTF-8 input is a runtime error, not a silent replacement

character.
- `len($b)` returns the **byte count** ; `len($s)` on a string returns the **rune count** . They will disagree for any non-ASCII input.
- `io.readBytes(n) -> bytes` reads n bytes from stdin; `io.readChars(n) -> string` reads n Unicode code points (1-4 bytes each, decoded from UTF-8). See libraries/io.md for details.

Structs

A **struct** names a fixed set of typed fields. Use a struct whenever a multi-value bundle would otherwise be a map keyed by string literals - the fields are checked at construction time, the field names are part of the type, and reading `$p.x` is faster and clearer than indexing a map by `"x"` .

A struct is defined once at the top level and reused everywhere:

```
def struct Point { x as int, y as int };
def struct Line { from as Point, to as Point };
```

The shape is `def struct Name { field as type, field as type, ... };` . The struct name follows the identifier rule (letters only, up to 64 characters); field names follow the same rule. The trailing `;` is required (every statement ends in one).

Constructing, reading, writing

```
# Construct - every field must be named at the literal.
def p as Point init Point{ x: 3, y: 4 };

# Read.
io.printf("%d %d\n", $p.x, $p.y);    # 3 4

# Write.
$p.x = 30;
```

The struct literal `Point{ x: 3, y: 4 }` requires **every** field; a missing field is a positioned error, and so is an unknown one (`z: 5` on a `Point`). Field order in the literal is free - the runtime stores each field at its declaration position regardless.

`def p as Point;` (no `init`) gives every field its declared zero, recursing through nested struct fields. So `def L as Line;` produces `Line{from: Point{x: 0, y: 0}, to: Point{x: 0, y: 0}}` without any extra ceremony.

Nested structs and chained access

A struct's field can itself be a struct, a list, a map, or any combination. Reads and writes chain through `.field` and `[index]` in whatever order makes sense:

```
def L as Line init Line{ from: Point{ x: 0, y: 0 }, to: Point{ x: 10, y: 20 } };

io.printf("%d\n", $L.to.x);    # 10  - field after field

$L.from.x = 5;                  # write through the chain
```

A struct field that's a list works the same way: `$bag.items[0] = 99;` descends through the `.items` field and writes into the list at index 0.

Value semantics

Like lists, maps, and bytes, structs are **value-typed** :

```
def p as Point init Point{ x: 1, y: 2 };
def q as Point init $p;     # independent copy
$q.y = 99;
# $p is still Point{x: 1, y: 2}; $q is Point{x: 1, y: 99}.
```

Function parameter binding copies too, so `func translate(pt as Point, dx as int)` that writes into `$pt` doesn't leak back to the caller.

const is deep

`def const ORIGIN as Point init Point{ x: 0, y: 0 };` rejects both `$ORIGIN = ...` (rebinding) and `$ORIGIN.x = ...` (content mutation), including writes that descend through nested struct fields. Same rule as lists and maps - the value behind a `const` is frozen at every depth.

Strict at boundaries

- **Unknown struct type at declaration** : `def x as Widget;` when no `def struct Widget` exists is a positioned runtime error ("unknown struct type").
- **Missing or unknown field at the literal** : positioned errors that point at the offending position.
- **Field type mismatch on write** : `$p.x = "hi";` on `x as int` errors with the declared type and the actual value's kind.
- **Field access on a non-struct value** : `$xs.foo` where `$xs` is a list errors with "field access `.foo` requires a struct, got list".

Enums (sum types)

A **struct** bundles several values that all exist at once. An **enum** models the opposite: a value that is *exactly one* of a fixed set of **variants** . Each variant is either a payload-less tag or carries its own named fields (a mini-struct).

```
def enum Shape {
    Circle { r as float },
    Rect { w as float, h as float },
    Empty
};
```

Construct a value by naming the variant:

```
def a as Shape init Shape.Circle{ r: 2.0 };
def b as Shape init Shape.Rect{ w: 3.0, h: 4.0 };
def e as Shape init Shape.Empty;              # payload-less: no braces
```

Consuming an enum with match

You do not read an enum's payload with `.field` - the whole point is that only *one* variant is present, so `$s.r` would be meaningless when `$s` is a `Rect` . Instead, `match` dispatches on the variant and hands you the payload:

```
func area(s as Shape) {
    match ($s) {
        when Circle(c) { return 3.14159 * $c.r * $c.r; }   # $c holds { r }
        when Rect(rc)  { return $rc.w * $rc.h; }            # $rc holds { w, h }
        when Empty     { return 0.0; }                      # no binder
    }
    return -1.0;
}
```

`when Circle(c)` binds the payload into a fresh `$c` for that arm only. `when Empty` matches the payload-less variant with no binder. The bound `$c` is a **read-only** snapshot - you read its fields (`$c.r`); to transform the data, build a new value.

A `match` over an enum must be **exhaustive** : every variant must be covered, or the `match` must carry an `else` . A forgotten variant is a **compile-time error** , not a silent no-op - so when you add a variant later, the compiler points you at every `match` that needs updating.

Value semantics, equality, zero

Enums copy by value like structs (`def b as Shape init $a;` is an independent copy), and compare equal when they are the same variant with equal payload (`Shape.Circle{r: 2.0}` == `Shape.Circle{r: 2.0}` is `true` ; `...` == `Shape.Empty` is `false`). `const` is deep.

`def s as Shape;` with no initializer gives the **first declared variant** , with its payload zeroed. So a recursive enum should declare a payload-less base case first:

```
def enum Lst { Nil, Cons { head as int, tail as Lst } };
def empty as Lst;            # -> Lst.Nil (a finite, well-defined zero)
```

An enum whose *first* variant refers to itself by value has no finite zero and is rejected when the program loads.

Naming

Enum and variant names can be spelled any way you like - `Shape` / `Circle` , `shape` / `circle` , `RGB` / `Red` , whatever reads well. The interpreter figures out what `Shape.Circle` means by looking it up, not from how it is capitalized.

One thing to know if you give an enum an all- UPPERCASE name: that is also the spelling of a **constant** , so if a constant or variable of the same name is in scope, it wins and `RGB.Red` reads as a field access on it. Rename either one to clear the ambiguity.

Encoding an enum as JSON

`json.encode` writes an enum in the conventional *externally tagged* form: a payload-less variant becomes its name as a string, and a payloaded one a single-key object naming the variant.

```
json.encode(Shape.Circle{ r: 2.5 })   # {"Circle":{"r":2.5}}
json.encode(Shape.Empty)              # "Empty"
```

Decoding does not rebuild an enum - the wire form carries no enum type, so `json.decode` gives you the plain string or map and your program decides what it means.

Methods

```
func greet(name as string) {
    io.printf("hello, %s\n", $name);
}

greet("Jennifer");   # call it from top level
```

Two keywords, two jobs:

- `def [const] NAME ...` introduces a **binding** (variable or constant).
- `func NAME(p as TYPE, q as TYPE) { ... }` introduces a **method** .

Parameters use bare identifiers (same rule as `def`) and each has a declared type. Inside the body, parameters are referenced as `$p` like any other variable. At the call site, the interpreter checks the number of arguments and the kind of each one - mismatches produce a positioned error.

Return values use `return EXPR;` to return a value or `return;` to return `null` . A body that runs to the end without `return` also yields `null` . Methods don't declare a return type; the caller's type check (e.g. `def x as int init mymethod();`) is what enforces the value's kind at the use site.

Recursion works out of the box - methods are hoisted, so any method can call any other (or itself) by name.

```
func fact(n as int) {
    if ($n == 0) { return 1; }
    return $n * fact($n - 1);
}

io.printf("%d\n", fact(5));    # 120
```

Methods are **hoisted** : all `func NAME() { ... }` declarations are collected before any top-level statement runs, so a method can be called from anywhere in the file regardless of where it's defined. There is no required entry point - top-level statements execute in source order.

Methods can only be defined at the top level (not inside another method's body). Method bodies inherit the global scope, so top-level variables are visible inside methods (subject to the no-shadowing rule).

Methods cannot shadow imported builtins. If you write `use io;` and then `func io.printf() { ... }` , the program is rejected:

```
runtime error at 2:1: method "printf" shadows a builtin from `io`;
rename it or remove `use io;`
```

Without the `use io;` , the name is yours to define. This is the same no-shadowing discipline Jennifer applies to variables.

Methods as values

A method name is also a **function value** : written bare (no `( )`, it evaluates to a `func` you can store, pass, and call later - `def f as func init greet;` then `$f(args)` . This is what powers the higher-order `lists` helpers (`lists.map` / `filter` / `reduce` / ...). See Function values for the full rules.

Control flow

Operators

Operator	Meaning
+	addition (int/float); also concatenation on string
-, *	subtraction, multiplication (int/float)
/	true division - always returns float
//	floor (integer) division; int // int -> int
%	modulo (int only); floored, matching //
unary -	numeric negation (int/float)
<, >, <=, >=	order two numbers (exact across int/float) or two strings (lexicographic); bool
==, !=	equality / inequality; same-kind plus exact int/float; bool
and, or	logical; both operands bool; short-circuit
not	unary logical negation; operand bool (there is no !)
&, `	, ^`
<<, >>	left / arithmetic right shift on int
unary ~	bitwise NOT on int (~x == -x - 1)

Division has two operators. `/` always returns a `float` (Python 3 style). `//` returns the floor, keeping the type when both operands are ints:

```
5 / 2          # 2.5 (float)
5 // 2         # 2   (int)
5.0 / 2.0      # 2.5 (float)
5.7 // 2.0     # 2.0 (float - floor of a float division)
```

So `def x as int init 5 / 2;` is rejected (right side is float). Use `5 // 2` for an int result, or `def x as float init 5 / 2;` .

% is floored , consistent with `//` , so the identity `(a // b) * b + (a % b) == a` holds for negative operands: `-7 // 3 == -3` and `-7 % 3 == 2` ; `7 % -3 == -2` . (This is Python's convention, not C/Go truncation toward zero.)

Integer overflow errors. Integer arithmetic whose result does not fit in a 64-bit int (`9223372036854775807 + 1`) is a positioned runtime error rather than a silent wrap. A mixed `int` / `float` comparison is **exact** - the int is not promoted to a lossy float - so a 64-bit int never spuriously compares equal to a nearby float.

(Line comments are `#` , freeing `//` for the Python-3 floor-division operator. The `#` choice also lets Jennifer files start with a shebang: `#!/usr/bin/env -S jennifer run` .)

Precedence (low to high): `or` , `and` , `not` , comparison, bitwise `|` , bitwise `^` , bitwise `&` , shifts `<< >>` , additive (`+` , `-`), multiplicative (`*` , `/` , `//` , `%`), unary `-` / `~` . Use parentheses to override: `(1 + 2) * 3` . The bit-op rungs sit between comparison and additive following Python's precedence, so `$x & 0xff == 0` parses as `($x & 0xff) == 0` (the intuitive interpretation), not the C/Go shape `$x & (0xff == 0)` . Examples that follow the rules:

```
not 1 == 2                 # not (1 == 2) -> true
1 > 0 and 2 > 1            # true
true or false and false    # true or (false and false) -> true
-3 + 10                    # (-3) + 10 -> 7
-3 * 2                     # (-3) * 2 -> -6
```

and and or short-circuit. The right operand is only evaluated when the left doesn't already decide the result. That matters when the right side has side effects:

```
def gate as bool init false;
def result as bool init $gate and expensive();   # expensive() not called
```

Mixed `int` / `float` arithmetic promotes the int to float and the result is a float (`3 + 0.5` -> `3.5`). **`/` always returns float** , even with two int operands (`5 / 2` is `2.5` , not `2`). Use `//` when you want an integer quotient: `5 // 2` is `2` . This is Python-3 division, not C/Java division.

Bitwise operators

The bit operators take `int` operands only - float is rejected with a positioned error. The shifts are arithmetic (sign-extending >>); a negative shift count is rejected, and a count >= 64 saturates to 0 or -1 the way hardware does. Non-decimal literals (`0xff` , `0o755` , `0b1010_0110`) and the _ digit separator (`1_000_000` , `0xDEAD_BEEF`) make bit-twiddling code much easier to read.

```
def mask as int init 0xff;
def x as int init 0xDEAD_BEEF;

io.printf("low byte:  %d|base=16\n", $x & $mask);   # ef
io.printf("flip last: %d|base=16\n", $x ^ 1);       # dead_beee
io.printf("shift 4:   %d|base=16\n", $x >> 4);      # dead_beef >> 4
```

Conditionals and loops

```
if ($n == 0) {
    io.printf("zero");
} elseif ($n < 10) {
    io.printf("small");
} else {
    io.printf("large");
}

while ($i < 5) {
    $i = $i + 1;
}

for (def i as int init 0; $i < 10; $i = $i + 1) {
    io.printf($i);
}

# for-each over a list or map.
for (def x in $xs) {
    io.printf("%d ", $x);
}
for (def k in $m) {
    io.printf("%s=%d ", $k, $m[$k]);
}

# for-each over a half-open range: 0..n yields 0, 1, ..., n-1.
# No list is built - the range is iterated lazily.
for (def i in 0..10) {
    io.printf("%d ", $i);
}
```

Conditions in `if` , `elseif` , `while` , and `for` **must be bool** - there is no implicit truthiness. Use a comparison (`$x == 0`) to get a bool. For-each (`for (def x in $coll)`) doesn't take a condition - it walks the whole collection (or the whole range).

A condition short-circuits. `and` / `or` only evaluate their right operand when the result is not already decided, so a call on the right side may never run:

```
use io;
func printer() { io.printf("ran "); return true; }
if (true or printer()) { io.printf("a\n"); }    # prints "a" - printer() is
skipped
if (false and printer()) { io.printf("b\n"); }  # prints nothing - printer() is
skipped
if (true and printer()) { io.printf("c\n"); }   # prints "ran c" - printer() runs
```

Put the cheap or decisive test on the left, and don't rely on a side effect in a short-circuited operand.

A range `lo..hi` is **half-open** (`[lo, hi)`): it includes `lo` and excludes `hi` , so `0..n` runs exactly n times. Bounds are int; `lo` > `hi` is an error and `lo` == `hi` is empty. The same `..` builds a list (`def r as list of int init 1..5;` is `[1, 2, 3, 4]`) and slices a collection (`$xs[a..b]` , covered in types-and-values).

Loop variable scope

C-style `for` opens its own scope. **Where you def the iterator variable decides whether you can still see it after the loop.**

```
# Loop-local: declare inside the for-init. The iterator lives only for
# the duration of the loop.
for (def i as int init 0; $i < 10; $i = $i + 1) {
    io.printf("%d\n", $i);
}
io.printf("%d\n", $i);   # ERROR: `i` not in scope here
```

```
# Outer-scope: declare in the surrounding scope, assign in the for-init.
# The variable survives past the loop and holds the value that made the
# condition false (10 here).
def i as int;
for ($i = 0; $i < 10; $i = $i + 1) {
    io.printf("%d\n", $i);
}
io.printf("%d\n", $i);   # ok - prints 10
```

The loop-local form is the recommended style; reach for the outer-scope form only when you actually need to inspect the iterator after the loop ends. For-each (`for (def x in $coll)`) is always loop-local - the iteration variable lives in a fresh scope each pass through the loop and is gone once the loop exits.

repeat ... until (post-test loop)

For loops that should run **at least once** , then keep going until a condition becomes true:

```
def n as int init 0;
repeat {
    io.printf("n=%d\n", $n);
    $n = $n + 1;
} until ($n >= 3);
# prints n=0, n=1, n=2 - the body runs three times before until is true.
```

The body runs unconditionally on entry, then `until (cond)` is checked **after** each iteration. The loop stops when `cond` evaluates true.

This is the post-test counterpart to `while` . The keyword pair `repeat` / `until` was chosen over `do { } while ...` so the condition inversion ("loop until done") reads as English and matches the rest of Jennifer's word-operator style (`and` , `or` , `not`). Like every other condition slot, `cond` must be `bool` .

match (multi-way value dispatch)

When a chain of `if` / `elseif` compares **one subject** against several values, `match` says it once:

```
match ($cmd) {
    when "start" {
        start();
    }
    when "stop", "halt" {      # several values in one arm
        stop();
    }
    else {                     # optional default, must be last
        io.printf("unknown: %s\n", $cmd);
    }
}
```

- The **subject** is evaluated **once** , then compared to each `when` value by the strict == operator (the same exact, type-strict rules - an `int` subject never matches a `float` value). The **first** arm with a matching value runs; the rest are skipped.
- An arm lists **one or more values** (`when 2, 3, 4`), which is an *OR of equality* - the arm runs if the subject equals any of them. The values are evaluated left-to-right and **stop at the first match** , so a side-effecting value in a later position may not run.
- Values are **any expression** , not just literals - `when MAX` , `when $limit` , `when lo(), hi()` . A bare `when Name { ... }` reads `Name` as the value and the `{` as the arm block, so **parenthesize a composite-literal value** : a struct literal `when (Point{x: 1, y: 2}) { ... }` (required), and a map literal `when ({"a": 1}) { ... }` (optional, but the parens let `jennifer fmt` lay the arm out cleanly).
- **No fall-through.** Each arm is an independent block; there is no `break` to end an arm and nothing falls into the next. Because of that, **match is not a break target** : a `break` or `continue` inside an arm acts on the enclosing **loop** , never the match - which sidesteps C's "break breaks the switch" and "forgot the break" traps at once.

```
for (def n in [1, 2, 3, 4]) {
    match ($n) {
        when 3 {
            break;     # breaks the for loop, not the match
        }
    }
    io.printf("%d ", $n);   # 1 2
}
```

- `else` is the **optional** default and must come last. **No matching arm and no else is a well-defined no-op** (nothing runs), not an error.
- `match` is a **statement, not an expression** - arms act, assign, or `return` , the same as `if` . Each arm's block is its own scope.

`jennifer fmt` lays a `match` out as a flat list of arms - like a `switch` / `case` in other languages: the subject on the opening line, then each `when` (and the `else`) starting its **own** line at the arm indent with its body on its own indented lines. Arms do **not** cuddle the previous arm's `}` (unlike an `if` 's `} else {`), so you can scan the `when` column top to bottom. A long `when` value list wraps with each continuation value aligned under the first.

`break;` exits the **innermost** enclosing loop:

```
for (def i as int init 0; $i < 10; $i = $i + 1) {
    if ($i == 5) { break; }
    io.printf("%d ", $i);
}
# prints "0 1 2 3 4 "
```

`continue;` skips the rest of the current iteration and starts the next one. In a C-style `for` loop, the step expression (`$i = $i + 1`) still runs before the condition is re-checked - matching the behaviour in C, Go, Java, and Python:

```
for (def i as int init 0; $i < 5; $i = $i + 1) {
    if ($i % 2 == 0) { continue; }
    io.printf("%d ", $i);
}
# prints "1 3 "
```

Both work in `while` , C-style `for` , for-each (`for (def x in $coll)`), and `repeat ... until` . In `repeat` , `continue` jumps to the `until` check (skipping the rest of the body); the loop still terminates normally when `until` becomes true.

Misuse:

- `break` and `continue` **only exist inside a loop** . Using one at the top level or as a stray statement in a method body that has no enclosing loop is a positioned runtime error.
- They do **not** cross the method-call boundary. A `break` inside a method body looks for a loop in **that method** , not in the caller. If the called method has no loop, the `break` errors.
- They only catch the **innermost** loop. To exit several levels at once, use a flag variable that the outer loop checks, or refactor the inner work into a method that `return` s when done.

Matching an enum (variant patterns)

When the subject of a `match` is a variable or parameter whose type is an enum, the arms are **variant patterns** instead of values. Each arm names a variant and, optionally, binds its payload:

```
def enum Shape { Circle { r as float }, Rect { w as float, h as float }, Empty };

func describe(s as Shape) {
    match ($s) {
        when Circle(c) { return "circle radius " + convert.toString($c.r); }
        when Rect(rc)  { return "rectangle"; }   # binder optional
        when Empty     { return "nothing"; }     # payload-less variant
    }
    return "?";
}
```

`when Circle(c)` binds the variant's payload into a fresh `$c` (a mini-struct with that variant's fields) for that arm only. A pattern `match` must be **exhaustive** - cover every variant or add an `else` - and a missing variant is a **compile-time error** , so adding a variant later flags every `match` you need to revisit. (A `match` over an ordinary value like an `int` or `string` stays non-exhaustive, as above.) `break` / `continue` in an arm still act on the enclosing loop, never the `match` .

The check applies wherever the subject's enum type is known: in the same file, inside a `spawn` body, and for an enum imported from a module. A same-file match is checked when the file is parsed (so `jennifer lint` reports it); one over an imported enum is checked when the program loads, since the module has to be read first.

exit

`exit;` terminates the whole program immediately - it skips the rest of the current method, every caller frame,

and every remaining top-level statement. The bare form yields exit code 0:

```
use io;
io.printf("ok\n");
exit;                   # process ends with code 0
io.printf("never\n");   # not reached
```

`exit EXPR;` sets the exit code; EXPR must evaluate to `int` :

```
use io;
io.printf("error: input missing\n");
exit 2;                 # process ends with code 2
```

On Unix a process exit status is only 8 bits, so the code a shell or parent process observes is EXPR & `0xff` : `exit 256` is seen as `0` and `exit 300` as `44` . Stick to the `0..255` range for an exit code a caller can actually read back.

`exit` is **distinct from return** . `return` ends the current method's body and yields a value to the caller; `exit` ends the program. Use `return` when a method has done its job; use `exit` when the whole run is over.

try, catch, throw

Catchable errors. `throw EXPR;` signals an error from any reachable point; `try { body } catch (NAME) { handler }` runs the body and, if anything inside it throws (user code or a runtime failure like out-of-bounds), runs the handler with `NAME` bound to the thrown value:

```
use io;

try {
    def n as int init convert.toInt($input);
    process($n);
} catch (err) {
    io.printf("not a number: %s\n", $err.message);
}
```

What can be thrown

Any value. The **convention** is an `Error` struct - the runtime auto-defines that struct shape so user code can rely on it without a `def struct Error { ... };` of its own:

```
def struct Error {
    kind    as string,    # short symbolic tag
    message as string,    # human-readable
    file    as string,
    line    as int,
    col     as int,
};
```

User code throws an `Error{...}` to signal expected failure modes; catch sites dispatch on `$err.kind` :

```
func parseConfig(src as string) {
    if (not strings.contains($src, "=")) {
        throw Error{
            kind: "parse_error",
            message: "missing `=`",
            file: "", line: 0, col: 0
        };
    }
    # ... happy path ...
}

try {
    parseConfig($cfg);
} catch (err) {
    if ($err.kind == "parse_error") {
        io.printf("config invalid: %s\n", $err.message);
    } else {
        throw $err;     # not our concern; let it propagate
    }
}
```

A bare `throw "boom";` still works (any value); the catch handler just won't be able to read `.kind` / `.message` off it. Use `convert.typeOf($err)` if you need to branch on the kind.

What can be caught

- **User-issued throw EXPR;** - whatever the user passed, copied into the catch binding (value semantics, like every other binding boundary).
- **Runtime errors** - out-of-bounds reads / writes, missing map keys, type mismatches, division by zero, undefined names, bytes-element range violations, and the rest of the positioned runtime errors. The runtime wraps them into the canonical `Error` struct with `kind = "runtime"` (more specific tags will land per site over time) and the original file / line / col preserved.

What can NOT be caught

- **exit / exit EXPR;** - the program-level escape hatch stays escape. `try { exit 1; } catch (e) { ... }` lets the exit through; the catch block does not run.
- **return / break / continue** - they're control flow, not errors. `try { break; } catch (e) { ... }` breaks the enclosing loop; the handler does not run.

Re-throwing

`throw $err;` inside a catch re-raises - the value propagates past the current `try` / `catch` to the next enclosing `try` . Same value unless replaced.

No finally

Jennifer has no `finally` clause, and won't - `defer` (below) covers the same need without `finally` 's footguns.

defer (deterministic cleanup)

`defer CALL(args);` schedules a single call to run when the **enclosing block** exits - on **every** exit path: normal fall-through, `return` , `break` , `continue` , a `throw` that unwinds through it, and `exit` . It is the clean way to pair "acquire a resource" with "release it", right next to each other:

```
use fs;
use io;

func dumpFirst(path as string) {
    def f as fs.File init fs.open($path, "read");
    defer fs.close($f);                # runs however this function exits

    if (fs.eof($f)) { return; }        # <- fs.close($f) runs here
    io.printf("%s\n", fs.readLine($f)); # <- and here, on normal exit
}
```

Rules:

- **The deferred thing must be a call** - a method call (`defer cleanup();`) or a namespaced / module call (`defer fs.close($f);`). A non-call (`defer 1 + 2;`) is a parse error. Because it is a single call, not a block, there is no way to smuggle a `return` or `throw` into the cleanup.
- **Arguments are evaluated at the defer line; the call runs at block exit.** `defer io.printf("done %s\n", $path);` captures `$path` 's value now.
- **LIFO.** Several defers in one block run last-registered-first, so resources release in reverse acquisition order:

```
def conn as net.Conn init net.connect("db.local:6379");
defer net.close($conn);              # released last
def f as fs.File init fs.open("query.log", "write");
defer fs.close($f);                  # released first
```

- **Block-scoped.** A `defer` runs at the end of its enclosing `{ }` , so one inside a loop body runs at the end of **each iteration** (no pile-up):

```
for (def name in $files) {
    def f as fs.File init fs.open($name, "read");
    defer fs.close($f);              # closes at the end of this iteration
    process($f);
}
```

A top-level `defer` runs at program end.

- **Does not cross the method or spawn boundary** (like `break` / `return`): a method's defers run when that method returns, not in its caller.
- A deferred call that **throws** propagates and is catchable; if the block was already unwinding an error, the deferred error supersedes it. An `exit` is never superseded - defers still run, but the exit code stands.

For durability specifically (flushing to disk), `defer` the `close` but call `fs.sync` explicitly and check it - see fs; a durability failure should surface before you tell the user "done", not from a deferred call running on the way out.

errdefer (undo on error only)

`errdefer CALL(args);` is `defer` 's error-path sibling: same single-call form, same argument snapshot at the `errdefer` line, same LIFO stack - but the call runs **only when the enclosing block exits with a propagating error** (a `throw` or a runtime error). On fall-through, `return` , `break` , `continue` , and `exit` it is skipped.

Reach for it when success must **keep** the resource and only failure should release it - the classic connect-then-handshake, where a plain `defer` would close the very connection you mean to hand to the caller:

```
func connect(addr as string) {
    def c as net.Conn init net.connect($addr);
    errdefer net.close($c);        # a failed handshake must not leak the socket
    handshake($c);                 # may throw partway through
    return Session{conn: $c};      # success: the caller owns the open conn
}
```

Rules beyond `defer` 's:

- **One teardown stack.** `defer` and `errdefer` in the same block run in one last-registered-first pass; on an error exit both kinds fire, on a normal exit the `errdefer` entries are skipped.
- **A failing defer arms the errdefers.** If a plain deferred call throws during teardown, the block is now exiting with an error - `errdefer` entries later in the teardown (registered earlier) do run.
- **exit is not an error.** A deliberate `exit` runs plain defers but skips errdefers, and the exit code stands.
- **Keep the undo call unlikely to throw.** An errdefer only ever runs while an error is already propagating, and if the undo call *itself* throws, its error supersedes the original (the same rule as `defer`) - the catch handler then sees the cleanup failure, not the root cause. A close on a handle you own is fine; anything that can plausibly fail belongs in explicit error handling.
- **Block-scoped, like defer - and unlike Zig.** An `errdefer` registered inside an `if` (or any inner block) is resolved when *that block* exits: leave the `if` normally and the errdefer is gone, even if the function throws two lines later. Register the errdefer in the same block that should own the undo - for the connect pattern, the function body, right after the acquire:

```
def c as net.Conn init net.connect($addr);
errdefer net.close($c);           # function-body scope: armed until return
if ($opts.tls) {
    $c = net.startTLS($c);        # NOT here - this block exits right away
}
handshake($c);
```

In the REPL, each input is its own frame (as with `defer`): an input that errors runs its errdefers, an input that succeeds discards them - they do not carry over to later inputs.

If cleanup must happen on *every* path (a temp file, a one-shot request's socket), use `defer` ; `errdefer` is only for the acquire-or-undo shape.

Concurrency

Jennifer's concurrency surface is small on purpose. A program can launch a computation in the background with `spawn` , get back a handle of type `task of T` , and read the result later with the `task` library.

```
use io;
use task;

def t as task of int init spawn { return slowComputation(); };
# ... other work happens here, in parallel with the spawn body ...
def n as int init task.wait($t);
io.printf("answer: %d\n", $n);
```

That's the whole story. No channels, no shared memory, no locks, no cancellation tokens, no goroutine-leak gotchas. The four pieces

- the `spawn` keyword, the `task of T` type, the `task` library,

and the exit-time loud-fail contract - are what concurrency ships.

The model

A `spawn { ... }` block evaluates its body **concurrently with the rest of the program** . It returns immediately with a `task of T` , where `T` is the body's declared return type at the use site. The body runs to its own `return EXPR;` ; that becomes the task's result. When the body raises a runtime error instead, that becomes the task's error.

Reading the result is a separate, explicit step:

Want	Use
The value (block until ready)	task.wait($t)
Check whether it's done yet	task.poll($t)
Fire and forget	task.discard($t)
Wait for a list of tasks	task.waitAll($ts)
First-to-finish from a list	task.waitAny($ts)
Stop a task	task.cancel($t) + task.discard($t)
Wait with a deadline	task.waitTimeout($t, ms)

See the `task` library reference for the full surface and worked examples per builtin.

Cancellation is cooperative. `task.cancel($t)` sets a flag the spawned body observes at its next loop checkpoint, where the runtime raises a catchable "task cancelled" - so a runaway `spawn` can be stopped (retiring the exit-time hang below), and a body that catches it inside a `try` / `catch` exits cleanly with a partial result. `task.waitTimeout` / `task.waitAnyTimeout` bound a wait, throwing a catchable "timed out" error. Full detail in the `task` reference's Cancellation and Timeouts sections.

Channels stream values. Where a `task` observes one result, a `channel of T` carries many values between goroutines: `channel.make(capacity)` , `channel.send` (which copies the value in, so the receiver gets its own copy), `channel.recv` , `channel.close` , and a fan-in `channel.select` . A channel is a shared conduit, but the data through it is copied, so the value-semantics guarantee below still holds. Producer / consumer, worker pools, and pipelines are the usual shapes.

Why value-semantics capture matters

The biggest single design choice is that **a spawn block captures its surrounding scope by deep copy, not by reference** . Variables visible at the `spawn` site are copied into the spawn frame at the moment of launch; mutations on either side afterwards are independent.

```
use io;
use task;

def xs as list of int init [1, 2, 3];
def t as task of int init spawn {
    return $xs[0] + $xs[1] + $xs[2];   # sees [1, 2, 3]
};

$xs[] = 99;                            # parent mutates after spawn

def total as int init task.wait($t);   # still 6
io.printf("%d\n", $total);
```

This is the same value-semantics discipline Jennifer applies everywhere - lists, maps, structs, bytes all copy on assignment and on function-parameter binding. Concurrency reuses the rule, so **data races are impossible by construction** . There is no shared mutable state to race on.

The cost is the obvious one: deep-copying a large structure into a spawn frame is O(N). For a 10 000-element list that's measurable. The benefit is that you never need a lock, a mutex, an atomic, or a channel to coordinate; concurrent code reads like sequential code with `spawn` markers.

Patterns

Parallel compute

```
use io;
use task;

func work(n as int) {
    # ... CPU-bound computation ...
    return $n * $n;
}

def tasks as list of task of int init [];
def i as int init 1;
while ($i <= 4) {
    $tasks[] = spawn { return work($i); };
    $i = $i + 1;
}

def results as list of int init task.waitAll($tasks);
# results are in list order, regardless of completion order
io.printf("%a\n", $results);
```

`task.waitAll` returns results in the same order as the input tasks, not the order in which they completed. That's the property most parallel-compute code wants.

Fire and forget

For background work whose result you genuinely don't care about, launch with `spawn` and mark the handle with `task.discard` :

```
use task;

def alive as task of null init spawn {
    # ... log a metric, send a notification, prefetch a cache ...
    return null;
};
task.discard($alive);
```

`task.discard` is required even for happy-path background work because of the loud-fail contract (next section). It declares to the runtime that no result is expected, so a later failure won't crash the program.

First-to-finish

`task.waitAny` returns the index of whichever task finished first; the caller follows up with `task.wait($ts[$idx])` to read its value. The other tasks keep running and need explicit observation:

```
use io;
use task;

def fast as task of int init spawn { return 1; };
def slow as task of int init spawn { return 2; };
def candidates as list of task of int init [$fast, $slow];

def winner as int init task.waitAny($candidates);
def value as int init task.wait($candidates[$winner]);
task.discard($candidates[1 - $winner]);

io.printf("winner=%d val=%d\n", $winner, $value);
```

The loud-fail contract

If a spawn body raises a runtime error and the program never observes the resulting task (never `task.wait` s it, never `task.waitAll` s it, never `task.discard` s it), Jennifer prints the error to stderr at program exit and exits non-zero.

This is deliberate. The default of every other concurrency model is some flavour of "errors in spawned work get silently dropped unless you go out of your way to handle them." That's a footgun

- bugs hide inside unobserved tasks until something else breaks

much later. Jennifer's contract is the inverse: an unobserved spawn error is **always** loud. The only way to silence it is to say so explicitly with `task.discard` .

```
use task;

def bad as task of int init spawn {
    def xs as list of int init [];
    return $xs[5];                          # error inside the body
};
# the program ends here without touching $bad
```

```
$ jennifer run example.j
spawn error (unwaited): list index 5 out of bounds (len 0)
$ echo $?
1
```

Three ways to make a spawn quiet:

1. `task.wait($t)` - read the result. The error rethrows at the wait site; an enclosing `try` / `catch` can suppress it.

2. `task.waitAll($ts)` / `task.waitAny($ts)` - same idea, with `waitAll` observing every survivor on the way out.
3. `task.discard($t)` - the explicit fire-and-forget marker.

Doing nothing is not a fourth option.

> **Beware infinite spawns.** The loud-fail scan blocks on every unobserved task to determine whether it produced an error. `spawn { while (true) { ... } }` without `task.discard` will hang the program at exit, since the goroutine never finishes. Long-running or potentially-non-terminating spawns should be paired with an explicit `task.discard` at the top of the scope.

What's deliberately not in v1

The current surface stops short of several features common to other languages' concurrency stories. Each was considered and deferred, not rejected:

- **Channels.** Inter-task communication beyond "wait for a return value" is not in v1. The chosen surface (`task of T` + `wait/waitAll/waitAny`) handles the most common cases without the bookkeeping channels require. A channel primitive would arrive in a later milestone.
- **Cancellation tokens.** No way to signal a running task to stop. The spawn body runs to completion or to an unhandled error. Cancellation is an open design question (cooperative flag? hard abort? structured-concurrency tree?) and stays deferred.
- **Timeouts.** No built-in `task.waitWithTimeout` . Build one with a sentinel spawn that returns after `time.sleep` and a `task.waitAny` over the pair. A higher-level helper may ship later.
- **Structured concurrency.** No automatic scope-bounded termination of unwaited tasks. The loud-fail contract is Jennifer's lighter-weight answer: not as strong a guarantee as Trio / async-await scopes, but enough to keep spawned errors visible.
- **Parallel for / map.** No `for parallel (...)` syntax. If you need it, write the explicit `spawn` / `waitAll` pair; that's what a parallel-map combinator would compile to anyway.
- **Atomics, mutexes, shared mutable state.** Value-semantics capture removes the need.

The single most likely follow-on in this area is a `time` -aware helper for timeouts; channels are second; structured-concurrency scopes a distant third (and might never land, since the loud-fail contract already addresses the main pain point).

See also

- `task` library reference - the five builtins, error propagation, worked examples.
- Control flow - `try` / `catch` works the same way around `task.wait` as around any other runtime error.
- Methods - calling methods from inside a spawn body works normally; the body inherits the global env the same way any other call frame does.
- `docs/milestones.md` - the concurrency entry has the design rationale (data-race-freedom by construction, the loud-fail decision, what's deferred to later).

Imports

Three keywords, three mechanisms:

```
use io;                  # library import - enables `io.printf`, `io.sprintf`, ..
.
include "helpers.j";     # textual file splice - pastes helpers.j here
import "./config.j";     # module import - loads config.j as its own module
```

`use` and `include` operate at the textual level; `import` is the module system - a real module boundary with run-once initialisation. A module loads and initialises before the code that imports it (see Module imports below).

Library imports

`use NAME;` enables a built-in library. Every library is **namespaced** - every name lives behind the library's prefix (`io.printf` , `math.sqrt` , `convert.toInt`). Nothing auto-loads; every program states its imports. Each library has its own reference doc; the table below is the index.

`len(EXPR)` is a language built-in (not a library) - polymorphic over string / list / map / bytes; no import needed.

Library	Enable with	Contents	Reference
io	use io;	io.printf, io.sprintf, io.readLine, io.eof, and the format-verb mini-language	libraries/io.md
kv	use kv;	in-process key/value store with per-key TTL (no-server local counterpart to memcache/redis): kv.open / openFile (persisted) -> kv.Store, then set/add/get/has/delete/touch/incr (signed delta)/close. Shared handle (safe for spawn)	libraries/kv.md
convert	use convert;	convert.toInt, convert.toFloat, convert.toString, convert.toBool, convert.typeOf - explicit casts	libraries/convert.md
math	use math;	math.abs, math.min, math.max, math.sqrt, math.pow, math.floor, math.ceil, math.round, math.rand, math.randInt, math.randSeed; constants math.PI, math.E	libraries/math.md
strings	use strings;	strings.upper, lower, contains, startsWith, endsWith, indexOf, trim, trimLeft, trimRight, replace, repeat, substring, split, chars, join	libraries/strings.md
binary	use binary;	binary.concat, slice, find, split, startsWith, endsWith - bulk operations on bytes (the byte-data counterpart to strings/lists)	libraries/binary.md
json	use json;	RFC 8259 json.encode/encodePretty/decode; structs and map of string to V map to objects, bytes to base64, integral numbers decode to int else float; decode yields generic values (no map-to-struct coercion)	libraries/json.md
toml	use toml;	RFC-conformant TOML 1.0 toml.encode/encodePretty/decode; same opaque toml.Value + read / walk / write surface as json (JSON Pointer), plus toml.asDatetime (backed by time.Time) for date-times; the config format Jennifer ships, not INI	libraries/toml.md
lists	use lists;	lists.push, pop, first, last, head, tail, reverse, sort, contains, concat, slice, shuffle, range - non-mutating helpers	libraries/lists.md
maps	use maps;	maps.keys, values, has, delete, merge - non-mutating helpers	libraries/maps.md
os	use os;	os.getEnv/os.setEnv, os.hasFlag, os.flag, os.run/spawn/wait/poll/kill; constants os.PLATFORM, os.ARCH, os.EOL, os.DIRSEP, os.PATHSEP, os.ARGS	libraries/os.md
path	use path;	path.base, dir, ext, stem, join, clean, isAbs, split - OS-aware path manipulation (the string layer paired with fs), no I/O	libraries/path.md

meta	use meta;	meta.VERSION, meta.BUILD - interpreter-self-identity constants	libraries/meta.md
time	use time;	instants, durations, calendar accessors, fixed-offset zones, strftime format/parse, ISO; structs time.Time, time.Duration, time.Zone; constant time.UTC	libraries/time.md
hash	use hash;	hash.compute(b, algo) for "md5"/"sha1"/"sha256"; streaming via hash.stream/update/finalize; struct hash.Stream	libraries/hash.md
archive	use archive;	tar / zip containers over bytes (no fs): archive.pack/unpack with format "tar"/"zip"/"tar.gz"; bundle is a list of archive.Entry {name, data, mode, mtime}	libraries/archive.md
compress	use compress;	byte-stream compression: pack/unpack for "gzip"/"zlib"/"deflate" (optional "fast"/"default"/"best" level) + streaming (compress.stream/update/finalize); struct compress.Stream	libraries/compress.md
crc	use crc;	crc.compute(b, algo) for "crc32"/"crc64" (big-endian bytes); streaming via crc.stream/update/finalize; struct crc.Stream	libraries/crc.md
encoding	use encoding;	isAscii/lenBytes/lenRunes introspection; toText/fromText for "hex"/"base64"/"base64-url"; encode/decode for character codecs "ascii"/"iso-8859-1"/"windows-1252"/"ebcdic"	libraries/encoding.md
task	use task;	observe / join task of T handles from spawn { ... }. task.wait, task.poll, task.discard, task.waitAll, task.waitAny	libraries/task.md
fs	use fs;	filesystem I/O. Whole-file read/write/append (readString/readBytes/writeString/writeBytes/appendString/appendBytes), metadata (exists/isFile/isDir/stat), directory ops (mkdir/mkdirAll/remove/removeAll/rename/list/walk), buffered handles (open/readLine/readChars/readBytes/writeString/writeBytes/eof/close); structs fs.Stat, fs.File	libraries/fs.md
net	use net;	TCP + UDP sockets and DNS. TCP connect/listen/accept/readBytes/writeBytes/eof, UDP listenUDP/sendTo/recvFrom, DNS lookup/reverseLookup, polymorphic close/address; structs net.Conn, net.Listener, net.UDPSocket, net.Datagram. Supported on the default jennifer binary; the constrained jennifer-tiny returns friendly errors from every entry point	libraries/net.md
httpd	use httpd;	HTTP/1.1 server engine over net/http (TLS, HTTP/2, graceful shutdown). Pull loop httpd.listen/accept/respond; request accessors method/path/query/header/body/remoteAddr; setHeader, serveFile/serveDir, shutdown; structs httpd.Server, httpd.Request. Default jennifer only; jennifer-tiny returns friendly errors	libraries/httpd.md
regex	use regex;	regular expressions over string (RE2 syntax). matches/find/findAll/replace/split/escape + regex.Match with positional and named captures. Implicit LRU cache for compiled patterns; rune-index offsets in matches	libraries/regex.md
testing	use testing;	test-runner primitives. run/results/reset/report + testing.Result. Catches runtime errors, throws, and exit inside test bodies. Three report formats: "text", "tap", "junit". Foundation for the .j-side testing framework	libraries/testing.md
uuid	use uuid;	RFC 9562 UUIDs. uuid.v4() / uuid.v7() + parse/isValid/version + constant NIL. The version is a method; RNG is the crypto library's crypto-grade source (unguessable, safe as a security token)	libraries/uuid.md

serial	use serial;	serial-port I/O with termios config. open(path, baud) / openWith(path, Options) -> serial.Port, read/write/flush/close. Linux-only; jennifer-tiny and non-Linux return a friendly stub	libraries/serial.md
spi	use spi;	SPI devices. open(path) -> spi.Device, configure(dev, mode, speedHz), full-duplex transfer(dev, bytes), close. Linux-only; stub elsewhere	libraries/spi.md
i2c	use i2c;	the I2C bus. open(path, addr) -> i2c.Bus, read/write/readReg/writeReg/close. Linux-only; stub elsewhere	libraries/i2c.md
gpio	use gpio;	GPIO over /dev/gpiochipN (v2 ioctls). Pin-keyed setup/read/write/release + IN/OUT, chip via gpio.chip; mirrors the sysfs gpio module. Linux-only; stub elsewhere	libraries/gpio.md
sql	use sql;	relational client over database/sql: MySQL / MariaDB + PostgreSQL. open/query/exec/next + typed accessors, transactions, prepared statements; placeholder-only binding (injection safety). Default binary only; jennifer-tiny stubs it	libraries/sql.md

See libraries/index.md for a fuller catalog and the library-organization principles, or libraries/cheatsheet.md for an alphabetical lookup of every builtin function and constant.

Quick orientation - if you're reading top to bottom and just want a flavor:

```
use io;
use convert;
use math;

io.printf("%s\n", convert.typeOf(5 / 2));     # "float"   [convert]
io.printf("%d\n", math.floor(math.PI * 2.0)); # 6         [math + io]
io.printf("%s\n", convert.toString(true));      # "true"    [convert]
```

The per-library docs cover every function in detail along with error cases.

Namespaced calls and aliasing

A qualified call is always `prefix.name(...)` ; a qualified constant is `prefix.NAME` . No spaces around the dot. The prefix is the library name by default; an `as ALIAS` clause renames it at the use site:

```
use os;
use math as m;

io.printf("on %s\n", os.platform());      # canonical prefix
io.printf("pi=%f\n", m.PI);               # aliased prefix
```

Aliasing rules

- **Rename, not addition.** After `use os as o;` only `o.` resolves; `os.platform()` errors with a "did you mean o ?" hint. Matches Python's `import foo as bar` shadowing of `foo` .
- **Canonical name freed.** The aliased canonical name (`os` above) is freed for ordinary identifier use - you *could* write `func os() { ... }` after `use os as o;` . The style guide recommends against it - it reads as a library call at first glance and surprises the reader.
- **Without aliasing, the prefix is reserved.** Bare `use os;` reserves `os` as a namespace prefix for the rest of the program; `func os() {}` then errors with `shadows imported namespace 'os'` .
- **Repeating a use is a silent no-op in the REPL.** In a batch program a duplicate `use` is accepted as a no-op too. Pick one form per program.
- **Module import works in the REPL.** `import "./mod.j";` at the prompt loads the module and binds its namespace (local `./` / `../` paths resolve against the current directory; a bare name walks the system module path), so a later `mod.member` resolves across inputs. Re-typing the same import is a no-op. A module is loaded once and cached by path, so editing the file and re-importing in the same session keeps the cached version - restart the REPL to pick up edits.

len is a language built-in

`len(EXPR)` is not a library function - it's a reserved keyword and a language primary expression. Polymorphic over string / list / map / bytes; no `use` statement needed:

```
def n as int init len("hello");        # 5 (rune count)
def m as int init len([1, 2, 3]);      # 3 (element count)
```

`len` used to live in an auto-loaded `core` library; `use core;` now errors with a migration hint. Build-version constants moved from `core` to `meta` (`use meta;` then `meta.VERSION` , `meta.BUILD`).

File splices (include)

`include "PATH.j";` textually splices another `.j` source file at the point of include. The path is a **string literal** that must end in `.j` . Relative paths resolve from the directory of the file containing the include; absolute paths and subdirectories work:

```
include "helpers.j";          # sibling file
include "subdir/utils.j";     # subdirectory
include "../shared/util.j";   # parent dir
include "/abs/path/lib.j";    # absolute path
```

File splices may appear anywhere a statement is allowed, including inside a block:

```
use io;
include "helpers.j";          # <- spliced here; whatever helpers.j contains
lands here
io.printf($helperValue);
```

Circular includes (file A includes file B, B includes A) are detected and rejected with an error.

Module imports

`import "PATH.j" [as NAME];` loads another `.j` file as a **module** - a real boundary, not a textual splice. Where an `include` pastes tokens into the current scope, an `import` runs the referenced file as its own program: its top level executes once, in its own scope, before the importing file's body runs.

```
use io;
import "./config.j";   # config.j initialises before this line returns
import "./db.j";       # db.j (which may itself import config.j) initialises next

io.printf("app: running\n");
```

The path decides where the module is found by its leading token:

- **Local** - `import "./util.j";` or `import "../shared.j";` resolves relative to the importing file's directory.
- **Absolute** - `import "/opt/pkg/m.j";` (leading `/`) is an absolute filesystem path.
- **Module** - `import "util.j";` (no `./` , no `/`) is looked up on the module search path: the system module directory first, then each `-I DIR` passed on the command line. The importing file's own directory is **not** consulted for this form. A `/` anywhere but the front is an ordinary subdirectory (`import "sub/util.j";` works in every form).

Guarantees:

- **Run-once.** A module's top level runs exactly once per program, cached by its resolved absolute path. Importing it again returns the same initialised module without re-running it.
- **Post-order init.** Imports initialise depth-first: a module is fully initialised before any module that imports it. If `main` imports `db` and `db` imports `config` , the order is `config` , then `db` , then `main` 's body.
- **Acyclic.** An import cycle (a imports b imports a) is a load-time error that names every edge in the loop.
- **Load errors are not catchable.** A parse error or a `throw` during a module's initialisation fails the program

at load. An `import` is a declaration, not an expression, so it cannot appear inside a `try` / `catch` - wrapping one is itself a parse error.

A module publishes with export; unmarked names are private. Put `export` in front of a top-level `def const` , `def struct` , or `func` to make it reachable through the importer's alias. A forgotten `export` stays internal (the safe default) - reaching a private name from outside is a positioned "not exported from module" error. (`export` is meaningful only in a module: it is a parse error in a `jennifer run` script.)

```
# config.j
use convert;
export def const MAXCONN as int init 16;   # public
def const NAME as string init "jdb";        # private (no export)
export func describe() {                     # public
    return NAME + " x" + convert.toString(MAXCONN);
}
```

Reaching a module's surface:

- **alias.fn(args)** calls an exported function - the arguments are evaluated in your code, but the body runs in the module's own scope (its constants and other functions, plus whatever *it* imported - `use` is not transitive, so the module's `use net;` does not give you `net.*`).
- **alias.CONST** reads an exported constant.
- **alias.Struct** names an exported struct type, and **alias.Struct{...}** builds one. A module struct keeps its identity (`module, name`) across the boundary, so you can hold one (`def p as alias.Struct init alias.make();`), read its fields, and pass it back - while `a.Point` and `b.Point` from two modules stay distinct types. An exported struct or function signature may not expose a *private* struct (a referential-closure error at the `export`).
- A module top level is **declarations-only** : `def const` , `def struct` , `func` , `use` , `import` - no mutable `def` and no free-standing statements. A `def const` initializer may still call a private `func` .

```
use io;
import "./config.j" as config;

io.printf("%s\n", config.describe());   # calls an exported function
io.printf("%d\n", config.MAXCONN);      # reads an exported constant
```

Testing: a co-located `MODULE_test.j` is a white-box overlay - `jennifer test MODULE_test.j` splices `MODULE.j` in first, so its `test*` methods reach the module's private names by bare identifier. For black-box tests, `import` the module and exercise its exported surface. See the runnable `examples/modules/` (https://github.com/jennifer-language/jennifer/tree/main/examples/modules) chain.

include vs import

Both read another `.j` file, but they differ at the boundary:

- `include` does a **textual splice** with no module boundary - the spliced file's top-level names land directly in the enclosing program's scope, and the same file spliced twice contributes its definitions twice. Use it to share snippets within one program.
- `import` loads a **module** - separate scope, run-once, post-order init. Use it to compose independent files.

Mixing the keywords up produces a positioned, actionable error:

```
include io;     -> error: `include` is for files; use `use io;` for
                   system libraries
use foo.j;      -> error: `use` is for system libraries; for files use
                   `include "foo.j";`
include foo.j;  -> error: file splices take a string literal:
                   `include "foo.j";`
include "foo.go"; -> error: include path "foo.go" must end with `.j`
import foo;     -> error: `import` takes a quoted module path
                   (`import "foo.j";`); for a system library use `use foo;`
```

Notes:

- The included file's contents must be valid where the include appears. A file containing a top-level `def` cannot be included inside a block (since definitions are only allowed at the top level).

Requirement header

A file can declare the minimum interpreter version and the host capabilities it needs, so a mismatch is refused with a clear message at **read time** - before lex or parse - instead of failing cryptically deep in a stub or after a breaking change. Requirements are stated with typed `pragma` directives in the leading header block:

```
# pragma-jennifer-version: >=0.25.0
# pragma-jennifer-capability: net
```

- **version** is a *minimum* floor (`>=major.minor.patch` ; newer always satisfies). The >= is **mandatory** - it is the only comparator. There is no > (strictly-greater), < , ~ , or range grammar: a floor names the release that introduced what you need, and any later version satisfies it. A bare `0.25.0` (no >=) is a malformed directive. Any development build of the interpreter bypasses the check (a `-dev` build is ahead of the last release); only a clean release tag is compared. A file states **one** version floor - a duplicate is an error.
- **capability** names a host facility the build must provide: `net` (TCP / UDP / TLS / DNS and the network-backed libraries) or `exec` (`os/exec` external processes). Neither is present on `jennifer-tiny` . List several on one line (`capability: net, exec`) or on separate lines; they accumulate. Query the same set at runtime with `meta.hasCapability` / `meta.CAPABILITIES` .

The header is a comment (a module top level is declarations-only, and the check runs before parsing), tolerant of the shebang, the SPDX block, and the docblock. It applies uniformly to a program, a module, and each `include` d file: **every file self-checks its own header** against the running interpreter, so `include` needs no merging - the effective floor is the highest and the effective capability set the union. A directive that opens # `pragma-jennifer-*` but does not parse, a bad version grammar, an unknown key, or an unknown capability name is a hard error (a typo must not silently disable the guard); `jennifer lint` (`L303`) reports the same problems before you run.

Jennifer style guide

This is the recommended source style for Jennifer programs. `jennifer fmt` re-emits source in exactly this shape, so anything you write that matches the spec will survive a `fmt` round-trip unchanged.

The guide is short on purpose: there are only a handful of rules, but they are consistent. If you've used `gofmt`, `prettier`, `rustfmt`, or PSR-12, nothing here will surprise you.

Spacing

- **One space around every binary operator**: `$i = 1 + 2;`, not `$i=1+2;`. Applies to `+ - * / // % < > <= >= == != and or` and `=`.
- **Unary `-` hugs its operand**: `-5`, `-$x`, `-fact($n - 1)`. No space between the `-` and the value it negates.
- **Word-form unary operators take a space**: `not $ok`, never `not$ok`. Same goes for any other keyword operator the language grows.
- **One space after `,` and `;` inside `for (...; ...; ...)`**, never before. `for (def i as int init 0; $i < 10; $i = $i + 1)`.
- **No space inside parentheses**: `io.printf("hi")`, not `io.printf( "hi" )`.
- **One space between a keyword and its `(`**: `if (cond)`, `while (cond)`, `for (...)`. Function calls don't get this space: `io.printf(...)`.
- **No space inside `[ ]` or `{ }` literals**: `[1, 2, 3]`, `{"a": 1, "b": 2}`, `Point{x: 1, y: 2}`, not `[ 1, 2, 3 ]` / `{ "a" : 1 }` / `Point{ x: 1 }`. This is uniform - list, map, and struct / enum literals are all tight (a struct literal also binds the brace to its type name: `Point{`, not `Point {`, see Braces). Empty literals are `[]`, `{}`, `Name{}`. Same rule as `()`.
- **No space before `[`**: `$xs[0]`, not `$xs [0]`. Index expressions hug their target.
- **No space inside `[]` for the append form**: `$xs[] = item;`, never `$xs[ ] = item;` or `$xs [ ] = item;`. Same rule as `$xs[0]` hugs its target and its brackets.
- **One space after `:`** in map literals, never before: `{"a": 1}`, not `{"a" :1}` or `{"a":1}`.
- **No trailing whitespace** on any line.

Indentation

- **4 spaces per level**, no tabs. Re-indenting on `}` always lands you back on a multiple of 4.
- One level per block - method body, `if` / `elseif` / `else` body, `while` body, `for` body.

Braces

- **1TBS (one true brace style): opening brace on the same line** for *everything* - methods and control flow alike - separated by one space: `func fact(n as int) {`, `if ($x > 0) {`, `else {`. (Jennifer uses the uniformly-same-line variant that Java, Go, Rust, and the Linux kernel also use.)
- **`}` on its own line**, except for the `} else {` / `} elseif (...) {` cascade, where `else` / `elseif` continues on the same line as the preceding `}`.
- **fmt always expands blocks across multiple lines** - the canonical form has the opening brace at end-of-line, each body statement on its own indented line, and the closing brace on its own line. Applies uniformly to method bodies, control-flow blocks (`if` / `elseif` / `else`, `while`, `for`, `repeat`, `match`), `try { } catch (e) { }`, and `spawn { }` block expressions. Single-line blocks are still legal source (the parser accepts them), but `fmt` rewrites them to the expanded form for consistency.
- **Struct and enum declarations expand to one member per line.** `def struct Point { x as int, y as int };` reflows to

```
def struct Point {
    x as int,
    y as int
};
```

and `def enum Shape { Circle, Square };` reflows the same way, one variant per line. A payload-carrying variant keeps its payload tight on the variant's line (`Circle{r as int},` - the brace binds to the variant name,

like a struct literal):

```
def enum Shape {
    Circle{r as int},
    Square{side as int},
    Empty
};
```

- **Struct / enum literals bind the brace to the type name, tight** : `Point{x: 1, y: 2}` - no space before `{` (so the brace reads as bound to `Point` , not a block) and no space inside it. Bodies are tight just like map and list literals (`{k: v}` , `[1, 2]`) - one uniform rule, no exceptions. An empty literal is `Name{}` .
- **Long literals wrap to one element per line.** A struct / enum / map / list literal stays inline while it fits; once its single-line form would pass the 100-column limit (or a struct / map literal has more than a handful of members), `fmt` breaks it with the opening bracket at end-of-line, one element per indented line, and the closing bracket on its own line:

```
def rule as FirewallRule init FirewallRule{
    chain: "forward",
    action: "drop",
    comment: "block inbound"
};
```

Each container decides independently, so a wrapped list of short maps keeps each map inline on its own line. A list of scalars wraps on width only (a long row of short numbers reads fine on one line).

- **Tail keywords cuddle the preceding }.** `} else { ... }` , `} elseif (cond) { ... }` , `} catch (e) { ... }` , and `} until (cond);` all keep the trailing keyword on the same line as the closing brace. `};` (a struct-decl terminator) also cuddles.
- **match arms each start on their own line - they do not cuddle.** A `match` is a flat list of peer arms (a `switch` / `case` , not a nested conditional), so - like `switch` / `case` / `when` in C, Go, Rust, Swift, Kotlin, Ruby, and Python - each `when` (and the `else`) begins its own line at the arm indent. An arm holding a **single short statement stays inline** (`when "idle" { begin(); }`); an arm with two or more statements, or one that overflows the line, expands one statement per indented line with its `}` on its own line. This is the one place `else` starts a line rather than cuddling a `}` : the `} else {` cuddle rule is for an `if` 's conditional tail, and a `match` 's `else` is a case-list arm, not that. Reading down the `when` column is the point. A `when` value list that overflows the line wraps with each continuation value aligned under the first:

```
match ($state) {
    when "idle" { begin(); }
    when longEventName1(),
         longEventName2() {
        prepare();
        handle();
    }
    else { reject(); }
}
```

Line length

- **Soft limit: 100 columns.** `fmt` keeps lines under 100 columns where it can without changing meaning. It has two levers: it **wraps a literal** (struct / enum / map / list) to one element per line (see Braces), and it **breaks a binary-operator chain** (`+` , `and` , `or`) after the operator. The operator break fills the line first, then hangs the operator at end-of-line and indents the continuation one level deeper - so it breaks at the last joiner whose next operand would still fit, not the first one past the limit:

```
def body as string init "line one\r\n" +
    "line two\r\n" +
    "line three\r\n";
```

Source-level line breaks at these joiners are also preserved even when the line would fit under 100 - so the shape above survives a `fmt` round-trip byte-for-byte. What `fmt` will **not** guess at: a long argument list or a deeply nested call has no safe break point, so it stays on one line, and a single token longer than the limit (a long string literal, a URL) stays over it - break those by hand. Because `fmt` never *introduces* an over-long line, its output does not add `lint L203` (`line-too-long`) findings that the source did not already have.

Statements

- **Every statement ends with ;** - no exceptions, including the last statement in a block.
- **One statement per line** . Don't chain multiple statements with `;` on a single line.
- **Blank lines separate logical groups** - imports from method definitions, methods from top-level code, distinct steps within a long block. Never more than one consecutive blank line.

Loops

- **Declare the iterator variable inside the for init** , not in the surrounding scope. The variable's lifetime should match the loop's:

```
for (def i as int init 0; $i < 10; $i = $i + 1) {   # preferred
    io.printf("%d\n", $i);
}
```

not

```
def i as int;
for ($i = 0; $i < 10; $i = $i + 1) {               # avoid
    io.printf("%d\n", $i);
}
```

The loop-local form is self-contained (reading the `for` line tells you everything about `i`), keeps the iterator out of the surrounding scope, and matches the for-each shape (`for (def x in $coll)`) which is *always* loop-local. The outer-scope form is only justified when you genuinely need the iterator's value after the loop ends - for example, to report which iteration triggered a break in a future language version that adds `break` . Use it deliberately, not by habit.

- **One concern per loop.** If the body is more than a screen, consider whether the work belongs in a helper method called from inside the loop.

Names

- **Variables, methods, parameters** : lowercase or `camelCase` if the name has multiple words.
- **Structs, enums, and enum variants** : `PascalCase` - `Point` , `Shape` , `Circle` . The interpreter does **not** enforce this (a name resolves by what it refers to, not by how it is capitalized), so it is a convention, not a rule. Following it keeps types visually distinct from `camelCase` values and `UPPERCASE` constants, and sidesteps the one real collision: an all- `UPPERCASE` type name would be read as a *constant* in `Name.member` position, so never name a type in all caps.
- **Constants** : `UPPERCASE` , with `_` as a single word separator. The full rule is `[A-Z]+(_[A-Z]+)*` , up to 64 characters: one or more uppercase chunks joined by single `_` . Every `_` must be immediately followed by `[A-Z]` - no leading, trailing, or consecutive `_` . Examples: `MAX` , `MAX_RETRIES` , `HTTP_OK` , `A_B_C` . Digits and lowercase letters are not allowed.
- **Library names** : lowercase, single word where possible (`io` , `math` , `strings` , `meta`).

Namespaced calls

Domain libraries are addressed by `prefix.name(...)` . The dot binds tight on both sides, like a method call's

(:

- **No space around .** : `os.platform()` , never `os . platform()` .
- **The call parens still hug the callee** : `os.platform()` , not `os.platform ()` .
- **use lib as alias; is one space on each side of as** : `use bio as b;` , never `use bio as b;` .

When you alias a library, the canonical name is freed for ordinary identifier use (e.g. you *could* write `func os() { ... }` after `use os as o;`). Don't. Reusing a library's canonical name reads as "this is a call into the library" at first glance, then surprises the reader when it isn't - keep the canonical name out of the user-method pool even when aliasing has technically freed it.

Strings

- **Default to "..." (cooked)** for ordinary text: `"hello"` , `"line\nbreak"` . The two delimiters are **not** interchangeable - `'...'` is **raw** (no escape processing), so pick by intent, not taste.
- **Reach for '...' (raw) when escapes would be noise** : a regex (`'\d+\.\d+'` over `"\\d+\\.\\d+"`), a Windows path, embedded JSON, or a multi-line block. A raw literal cannot contain a `'` , so a string with an apostrophe uses the cooked form (`"it's"`).
- **fmt preserves a literal's exact source spelling** - its delimiter, its escapes, and any line breaks inside it - so a deliberately raw or multi-line string survives a format untouched (a formatter must not alter a literal's value or shape).
- A string literal of **either** form *may* span multiple lines (a newline between the quotes is part of the value), which is handy for an embedded template or sample; `fmt` keeps it as written rather than collapsing it.

Interpolation slots

- **A cooked "..." interpolates {expr}** ; a raw `'...'` never does. Prefer interpolation over `sprintf` concatenation when you are placing a value beside its label: `"user {$name} has {$count} items"` reads better than a `%s` / `%d` template whose arguments trail off the end.
- **Keep a slot trivial** - a variable (`{$x}`), a constant (`{MAX}`), a field or index access (`{$p.name}` , `{$xs[0]}`), or arithmetic (`{$a + $b}`). These read at a glance and have no hidden cost.
- **Do not hide a call in a slot.** `"total {computeTotal($cart)}"` buries a method call - with its cost and any side effect - inside what looks like a string literal. Compute it first (`def total ...; "total {$total}"`). `lint` flags a call in a slot as **L204** .
- **A literal brace is \{ / \}** in a cooked string; if a string is mostly literal braces (JSON, a `{{ }}` template), reach for a **raw** `'...'` string instead of escaping every brace.

Comments

- `# line comment` for short notes that belong on or just above the thing they describe. The very first line may be a shebang (`#!/usr/bin/env -S jennifer run`); the lexer treats it as a comment.
- `/* block comment */` for longer commentary that doesn't fit one line. Block comments nest, so wrapping a chunk of code that already contains a block comment in another `/* ... */` works.
- **Inline block comments inside (, [, or after . get a space on the operand side.** `printf(/* note */ $x)` , not `printf(/* note */$x)` . This is a deliberate exception to the "no space inside `()` " rule above: `*/$x` runs together visually and is harder to read than the spaced form. The comment hugs the opening delimiter (no space between `(` and `/*`) so only the operand side picks up the space. Same rule for `[/* note */ 0]` and `$obj./* doc */ field` .
- Comments explain *why* , not *what* . The code already says what.

Doc comments

Document every public func, def struct, and def const with a doc comment , and open a file with a module preamble. A doc comment is a block comment that opens with exactly `/**` (a plain `/*` stays an ordinary comment), sits immediately above the construct it documents, and holds a summary line, an optional description, and `@` -tags. This is the format the `docblock` module parses, so your docs are machine-readable, not just prose.

```
/**
 * Distance between two points.
 * A longer description can follow the summary line.
 * @param ax {float} first x coordinate
 * @param ay {float} first y coordinate
 * @return {float} the Euclidean distance
 * @since 0.9
 */
export func distance(ax as float, ay as float, bx as float, by as float) { ... }
```

- **Types go in { }, in Jennifer's own syntax** : `{int}` , `{list of int}` , `{map of string to int}` , `{json.Value}` . There is no any - an opaque value documents as `json.Value` or a named struct.
- **export is read from the keyword** , never a tag - don't write `@public` .
- **Tags** : `@param name {type} desc` (functions) and `@field name {type} desc` (structs), one per parameter / field; `@return {type} desc` ; `@throws {type} desc` ; and the universals `@since` , `@deprecated [reason]` , `@see` , `@example` , `@internal` .
- **A file preamble** is a doc comment carrying `@module name` , plus optional `@author` , `@version` , `@license` . It goes at the top, after the SPDX header.
- **Keep doc names in step with the code.** `docblock` cross-checks `@param` / `@field` names against the real declaration and reports drift, so a stale doc is a caught bug, not a silent one.

`jennifer fmt` preserves doc comments and keeps each on its own line above its construct, so a formatted file is exactly what `docblock` expects to parse.

SPDX header

Every committed `.j` file opens with a two-line SPDX (https://spdx.dev/) header - a machine-readable license tag and a copyright line - so license-scanning and REUSE (https://reuse.software/) tooling can attribute each file without parsing prose:

```
# SPDX-License-Identifier: LGPL-3.0-only
# SPDX-FileCopyrightText: Copyright (C) 2026 mplx <jennifer@mplx.dev>
```

- **SPDX-License-Identifier** is the SPDX license expression for the file. The Jennifer project itself uses `LGPL-3.0-only` ; your own program carries whatever license you ship under.
- **SPDX-FileCopyrightText** is the copyright notice, kept behind its SPDX tag (`SPDX-FileCopyrightText: Copyright (C) <year> <holder> <email>`) and placed **directly under** the license line - one blank line then separates the header from the first code line.
- Both are # line comments. If the file is executable it may open with a `#!/usr/bin/env -S jennifer run` shebang on **line 1** , with the two SPDX lines immediately below it.
- The identical two-line header (written with `//`) tops every `.go` file in the interpreter, so the license/copyright form is uniform across the whole repo. Markdown (`.md`) files carry **no** header.

`jennifer fmt` treats the header as an ordinary leading comment block and re-emits it verbatim - it never rewrites, reorders, or drops your SPDX lines.

Source file conventions

- **.j extension** for all Jennifer source. The interpreter rejects anything else.
- **The two-line SPDX header** above tops every committed `.j` file.
- **use and import statements come first** , before any methods or top-level statements. Group `use` lines together, then `import` lines, then a blank line, then the rest of the program.
- **Blank line after a leading comment block.** If the file opens with a header comment (the SPDX license + copyright lines, an optional file description, a shebang), leave one blank line between the comment block and the first code line. Files that start directly with code (no header) start on line 1 - no leading blank.
- **Trailing newline** at end of file.

Editor configuration

Drop the following into a `.editorconfig` file at your project root and any editor with EditorConfig support (https://editorconfig.org/#download) will enforce the spacing and file-encoding rules above automatically:

```
# .editorconfig
root = true

[*.j]
indent_style = space
indent_size = 4
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true
```

That covers the indentation rule (4 spaces, no tabs), trailing-whitespace and final-newline conventions, and pins UTF-8 + LF line endings so collaborators on different OSes don't accidentally introduce CRLF diffs. `jennifer fmt` re-emits source in the same shape, so the EditorConfig settings and the formatter never disagree.

If you keep `.j` files alongside other languages in one repository, add a generic fallback as the first block so plain text files don't drift either:

```
[*]
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true
```

A complete example

```
# SPDX-License-Identifier: LGPL-3.0-only
# SPDX-FileCopyrightText: Copyright (C) 2026 mplx <jennifer@mplx.dev>

use io;

/**
 * Factorial of a non-negative integer.
 * @param n {int} the operand (assumed >= 0)
 * @return {int} n!
 */
func fact(n as int) {
    if ($n == 0) {
        return 1;
    }
    return $n * fact($n - 1);
}

for (def i as int init 0; $i <= 8; $i = $i + 1) {
    io.printf("%d! = %d\n", $i, fact($i));
}
```

Everything in this example follows the rules above: the two-line SPDX header, 1TBS braces, 4-space indent, spaces around binary operators, double-quoted strings, expanded blocks, and a doc comment on the `func` . `jennifer fmt` will produce this output byte-for-byte from any equivalent input.

Comments, blank lines, and a shebang on line 1 all survive a `fmt` round-trip. The two SPDX header lines (#

`SPDX-License-Identifier: ...` and `# SPDX-FileCopyrightText: ...`) and any inline `# why` notes you keep alongside the code are re-emitted at their original positions: leading comments stay on the line above their attached statement, trailing same-line comments stay on the same line, and runs of blank lines collapse to one (matching the "never more than one consecutive blank line" rule).

Best practices

Stylistic guidance for writing Jennifer that reads well, ages well, and fits the way the language is shaped. Each entry is a heuristic, not a hard rule - the language won't stop you, but the rule of thumb is there because the alternative tends to bite later.

Follow the style guide

The single biggest readability win is uniform source style across a codebase. When every file uses the same spacing, brace placement, and naming, the eye learns the shape of well-formed Jennifer and starts spotting bugs from rhythm alone - a one-off indent or a stray space becomes a signal. The reverse is also true: every codebase that tolerates "personal style" eventually pays for it in review friction, merge conflicts over whitespace, and reader-time spent on the wrong question ("is this code unusual because it does something unusual, or just because the author indents differently?"). Pick the agreed style once, then stop thinking about it.

Jennifer ships its style as both a written spec and an enforcement tool: read Style guide for the canonical rules (spacing, braces, naming, literal layout), then run `jennifer fmt` to make any file conform. Running `fmt` on save - or at minimum before every commit - is the cheapest habit you can adopt; it removes style from the list of things you and your reviewers have to think about.

Lint for suspect patterns with jennifer lint

`fmt` fixes how code *looks* ; `jennifer lint` flags what it *does* that is legal but probably wrong. It sits between the formatter and the parser: the code parses and runs, but something is still worth a second look. Each check has a stable ID, grouped by concern - the leading digit is the group: **L0nn** source errors, **L1nn** correctness, **L2nn** complexity and style, **L3nn** API lifecycle:

ID	Flags
L001	the source could not be tokenized (lex error)
L002	the source could not be parsed (parse error)
L003	an include could not be spliced (preprocess error)
L004	a malformed or unknown-ID # lint-disable comment
L101	a local variable declared but never used
L102	code after a return / throw / exit / break / continue
L103	an empty catch block (an error caught and silently thrown away)
L104	a throw of something that isn't an Error struct
L105	a condition that is always true or always false (if ($x == $x), ...)
L106	a use / import whose namespace is never used
L107	an unqualified call to a name that is not a defined method (a typo'd method)
L201	a method with too many statements (default over 60)
L202	block nesting deeper than the limit (default over 4 - see below)
L203	a source line longer than the 100-column limit
L301	use of a deprecated API (reserved, empty until an API is deprecated)
L302	use of a removed API (e.g. an old use core;)

The **L0nn source errors are always on** - you can't disable "the file doesn't parse". They report in whatever `--format` you ask for, so a `--format=json` pipeline gets valid output even when a file is broken. Everything else (`L1nn` / `L2nn` / `L3nn`) is selectable:

```
jennifer lint myprogram.j                # human-readable, with source carets
jennifer lint --format=json app.j        # JSON array of findings, for editors/CI
jennifer lint --checks=!L201,!L202 app.j  # run everything except the two style
checks
```

The exit code follows `gofmt -l` / `shellcheck` : `0` clean, `1` when there are findings (a source error counts), `2` only when the *invocation* itself is broken (bad flags, unreadable file, bad `--checks`). That makes `jennifer lint` a natural pre-commit or CI gate.

When a flag is a deliberate choice, silence it in place rather than disabling the check everywhere - the ID keeps the intent greppable. The directive sits on the line the finding anchors to: usually the offending statement, but the `func` line for `L201` and the block-introducer line for `L202` :

```
try { risky(); } catch (e) { }   # lint-disable: L103
```

A `# lint-disable-file: L201, L202` at the top of a file silences those IDs file-wide, and a `.jennifer-lint` file at your project root sets defaults for every run (same `IDS` / `!IDS` format, one direction). There is no blanket "disable everything" - a directive always names the IDs it turns off, so a reviewer can see exactly what was waved through. A doubled marker (`## lint-disable: ...`) is just a comment, not a directive - handy for writing *about* a directive. `examples/linting.j` demonstrates every check and its suppression.

Document your public surface with doc comments

A `/** ... */` doc comment on every exported construct - each `func` , `def struct` , `def enum` , `def const` , plus the file's `@module` preamble - is what turns a module from "source someone can read" into "an API someone can use". These comments are not just prose: the `docblock` module parses them into structured data, so they are the single source your API reference is generated from, and `docblock` reports **drift** - a `@param` naming no real parameter, a parameter with no `@param` , a `@field` on the wrong construct - which is checked in CI, so the docs cannot silently rot out of sync with the code.

The shape is a **summary** (the first paragraph), an optional **description** (the paragraphs after a blank line), then @ -tags:

```
/**
 * Parse a duration like "1h30m" into seconds.
 *
 * Accepts `h` / `m` / `s` units in any order; a bare number is seconds. A
malformed
 * string is a catchable Error, never a silent zero.
 * @param text {string} the duration to parse
 * @return {int} the total number of seconds
 * @throws {Error} kind "duration" on a malformed string
 */
export func parseDuration(text as string) { ... }
```

Two habits pay off:

- **Put a blank line between the summary and the description.** The summary is the first *paragraph* , not the first line - so an opening sentence that wraps across source lines stays one sentence - but a blank line is what ends the summary and begins the description. Leave it out and the whole opening block becomes the summary (which then reads as one long teaser in the generated catalog).
- **Write types in Jennifer syntax, in braces** - `{int}` , `{list of string}` , `{map of string to Point}` , `{json.Value}` . There is no `any` pseudo-type; an opaque value documents as its named type. The braces make the type unambiguous to a reader and to the parser.

See the `docblock` reference for the full tag set (`@since` / `@deprecated` / `@example` / `@see` / `@internal` / ...) and every diagnostic it reports.

Why 4+ levels of nesting is a code smell

The flexibility that lets `list of list of int` hold any shape gets unreadable fast as you nest deeper. Here's a four-level type holding "per game, per player, per character, per inventory slot, the item name":

```
def saves as list of list of list of list of string init [
    [["sword", "shield"], ["bow"]], [["dagger"]]],
    [["staff", "amulet"]], [[], ["potion", "rope", "torch"]]]
];

# What does this even mean?
$saves[0][1][0][0] = "axe";
```

Three problems:

1. **No semantic names for the dimensions.** Is index 2 "the character" or "the inventory slot"? You can't tell without going back to read the declaration and counting brackets.
2. **Bug-prone access.** `$saves[0][1][0][0]` is four indices that all look the same. Off-by-one or off-by-level errors are silent until the program either panics or, worse, modifies the wrong slot.
3. **Inflexible.** Adding a fifth dimension (per save slot, per timestamp, ...) means rewriting every access site in the program.

The standard fix is a struct or named record (see Structs). Other options that work without introducing a new type:

- **Wrap access in methods** : `getItem(save, player, character, slot)` reads better than four bare brackets and gives you one place to fix a bug. Internally the function still walks the nested lists, but call sites are self-documenting.
- **Flatten with composite keys** : `map of string to string` keyed on `"save:0/player:1/char:0/slot:0"` trades index speed for name clarity. Better when the structure is sparse anyway.
- **Decompose into parallel simpler structures** : one list of save metadata, one map from save-id to inventory, etc.

As a rule of thumb: **one level is normal, two is fine, three is uncommon, four is almost always a sign there's a missing abstraction.**

Write for clarity first; tune the hot path separately

Factor code into small, well-named helpers and build data structures the obvious way - the interpreter already elides the copies that would otherwise make that expensive (a read-only parameter is bound by alias, not deep-copied). The handful of places where the *shape* of the code decides whether a loop is linear or quadratic - growing a string with + in a loop, calling a helper once per element in a hot path - are collected in Performance, together with how to find them with `jennifer profile` . Reach for those rewrites only where a profile points; everywhere else, the readable version is the right version.

Performance

Most Jennifer code never needs performance tuning, and the interpreter already handles the parts that would otherwise be traps. Values are copied eagerly at every store site - assignment, parameter binding, a `spawn` snapshot - which is what makes aliasing bugs impossible. But the interpreter elides the copy wherever it can prove doing so is invisible, and it does this for you automatically:

- **Read-only parameters are borrowed, not copied.** A parameter a method never writes is bound by alias, so passing a large `list` or `map` into a helper that only reads it does not deep-copy it. You get this with no annotation and no special syntax - just write the read-only helper.
- **Call frames are pooled and slot-bound.** A deeply recursive program allocates a near-constant handful of objects rather than one allocation per call, so recursion and tight call loops do not generate garbage-collector pressure on their own.

So this page is not "hand-optimize everything." It is the short list of patterns where the *shape* of your code, not the interpreter, decides whether a hot path is linear or quadratic - plus how to find the hot path in the first place.

Measure before you change anything. `jennifer profile` reports hits and wall-clock time per source position; `jennifer profile --allocs` reports value-semantics copies. Optimize the line the profiler names, not the line you suspect. A hot path in a Jennifer program is almost always a single loop; the rest of the program does not repay the readability you would spend tuning it.

Build strings, lists, and bytes by appending, not by re-concatenating

Because values are copied rather than shared, `$s = $s + piece` copies the *entire* string accumulated so far on every step. Growing a string with + inside a loop is therefore **quadratic** - the total work grows with the square of the number of pieces, and a few thousand appends spend most of their time copying intermediate strings they immediately throw away:

```
# O(N^2): each `+` copies the whole accumulated string
def out as string init "";
for (def row in $rows) {
    $out = $out + $row + "\n";
}
```

Collect the pieces in a `list of string` with the `$xs[] = item;` append form - which is amortised **O(N)** , because a list grows its own backing in place with no whole-value copy - and join once at the end:

```
# O(N): append is amortised constant, a single join builds the result
def lines as list of string init [];
for (def row in $rows) {
    $lines[] = $row;
}
def out as string init strings.join($lines, "\n");
```

Building up `bytes` works the same way: append with `$b[] = byte;` and use the buffer directly (there is no + for `bytes`). And it is not only strings - any accumulator you rebuild with a whole-value operation each iteration has this shape. Prefer the in-place append (`$xs[] = item;`) over `$xs = lists.push($xs, item);` inside a tight loop for the same reason: the append grows the binding's own backing, the `push` builds a fresh copy.

Reach for + only to join a small, fixed number of pieces (`$name + ": " + $value`). The moment a concatenation lives inside a loop, switch to append-and-join.

When you must concatenate in a loop, group the small piece. A long left-associative chain re-copies the growing accumulator at every + : `$s + a + b + c + d` copies `$s` four times. Parenthesising the small part builds it once and touches the large accumulator a single time:

```
$out = $out + $field + ": " + $value + "\n";      # copies the growing $out at
every +
$out = $out + ($field + ": " + $value + "\n");    # builds the chunk once, one
copy of $out
```

Both produce the same string (concatenation is associative), so this is a free rewrite - do not touch the large accumulator more often than you must.

Push byte-buffer work into the binary library

Everything above is about not re-copying an accumulator. `bytes` has a second trap on top of that: every individual byte operation from `.j` - reading a byte, comparing it, appending one - pays the tree-walker's per-operation cost. On a kilobyte that is invisible; on a megabyte buffer it dominates. The `binary` library exists for exactly this - each function runs its inner loop once in Go, at native speed - so the two byte-buffer hot shapes both have a one-call answer.

Assembling many chunks - binary.join, never concat in a loop. `bytes` has no `+` , but `binary.concat` is the same quadratic trap: it copies both inputs into a fresh result, so growing a buffer with it re-copies the whole accumulator every iteration - **O(N^2)** .

```
# O(N^2): each concat re-copies the whole accumulated buffer
def out as bytes;
for (def chunk in $chunks) {
    $out = binary.concat($out, $chunk);
}
```

`binary.join` is the byte-data counterpart of `strings.join` : append the pieces to a `list of bytes` , then join once into a single allocation sized to the total - **O(N)** .

```
# O(N): append the pieces, one join builds the result
def parts as list of bytes init [];
for (def chunk in $chunks) {
    $parts[] = $chunk;
}
def out as bytes init binary.join($parts);   # optional separator: binary.
join($parts, $sep)
```

This is the same distinction as the string case, split by what you hold: appending *single bytes* one at a time (`$b[] = byte;`) is already the amortised-O(N) in-place form; joining *chunks* you already have is `binary.join` . Only `binary.concat` in a loop is quadratic. (When the chunks arrive off a *stream* rather than a list in hand, `net.readN` / `net.readAll` grow one Go slice for you - reach for those over a read-and-concat loop.)

Searching and slicing - one native call, not a hand-rolled per-byte scan. Walking a buffer byte by byte to find a delimiter, or copying out a range with an index loop, pays the interpreter on every byte. `binary.indexOf` / `binary.slice` / `binary.split` / `binary.contains` carry that loop into Go (`bytes.Index` , a `copy`), so locating a CRLF or a MIME boundary in a large body is one call:

```
# Slow: a per-byte comparison loop pays the interpreter's cost on every byte
def idx as int init -1;
def i as int init 0;
while ($i + len($needle) <= len($buf) and $idx < 0) {
    # ... byte-by-byte comparison ...
    $i = $i + 1;
}

# Fast: one native-speed scan
def idx as int init binary.indexOf($buf, $needle);
```

The rule mirrors the string one: build a `bytes` result by appending single bytes or by joining chunks, never by

concatenating a growing buffer in a loop - and when you search or slice a whole buffer, let one `binary` call do the walking.

Caution: byte offsets are not character offsets - do not cut a rune

Dropping from rune-indexed `strings` operations to byte-indexed `binary` ones buys speed, but hands you one responsibility the string layer handled for you: **a multi-byte UTF-8 character occupies several bytes, and a byte-position cut can land in the middle of one.** Decoding such a fragment back with `convert.stringFromBytes($b, "utf-8")` is a **runtime error** - the decoder is strict and rejects a truncated sequence - so a buffer that merely happens to carry an accented letter or an emoji across your cut point turns a working program into a throwing one.

What makes byte-scanning safe in the first place is that ASCII is transparent to UTF-8: every byte of a multi-byte character is `>= 0x80` , so an ASCII byte - a `\n` , a `,` , a `|` , the `*` in a `/**` - can never appear inside one. Cutting *at an ASCII delimiter* (the usual reason to scan bytes at all) is therefore always rune-safe: the boundaries fall between characters by construction. `binary.indexOf` / `binary.split` on an ASCII needle, and a slice that ends at an ASCII marker, are all fine.

The trap is a cut at an *arbitrary* byte position - a fixed `offset + 4096` look-ahead window, a midpoint derived from a byte length. If you must cut there, back the end off any partial trailing rune before decoding; a UTF-8 continuation byte is in `0x80..0xBF` :

```
# Back the cut to a character boundary so the slice never ends mid-rune
while ($end > $start and $end < len($buf) and $buf[$end] >= 128 and $buf[$end] <
192) {
    $end = $end - 1;
}
def s as string init convert.stringFromBytes(binary.slice($buf, $start, $end),
"utf-8");
```

(Counting bytes, searching for an ASCII delimiter, or moving raw `bytes` around untouched never has this problem - it bites only when you slice at a computed position and then decode the result back to a `string` .)

Get hot calls out of the innermost loop

A method call costs two separable things: the **copy** of each argument into the callee, and the **call itself** - borrowing a frame, binding the parameters, the call-depth check, and walking the method body. Read-only-parameter borrow (above) makes the copy free when the callee only reads its argument, but it cannot make the call itself free. So a helper invoked *once per character or per element* in a hot loop still pays the call overhead once per element, even after the argument copy is gone:

```
# classify() is invoked once per element - N call frames
func scan(cs as list of string) {
    def n as int init 0;
    def i as int init 0;
    while ($i < len($cs)) {
        if (classify($cs[$i])) { $n = $n + 1; }
        $i = $i + 1;
    }
    return $n;
}
```

In the hottest loops, get the call off the per-element path. In rough order of preference:

- **Inline** the helper's body into the loop when it is small. No call at all.
- **Batch** it: call the helper once over the whole collection (a `classifyAll(cs)` that returns a `list`), so the per-element work runs inside a single frame rather than one frame per element.
- **Lift loop-invariant calls** out of the loop entirely: anything the helper recomputes each iteration that does not

depend on the current element (a config read, a constant) belongs above the loop.

The win is real, and it is the last one available at the source level: one real workload - a syntax highlighter that called a predicate once per character - ran **1.37x** faster after this rewrite, with byte-identical output and no interpreter change.

Do this surgically. This is a hot-inner-loop transformation, the kind a profiler flags with a five- or six-digit hit count on one line. Everywhere else, keep factoring code into small helpers: the call overhead is invisible outside the hottest loops, and readable structure is worth far more than a call you make a few hundred times. Inlining is a readability cost you pay only where the profile earns it.

See also

- Editor & AI support / tooling - running `jennifer profile` .
- Types and values - value semantics, the copy-on-store model these patterns work around.
- `binary` library - the bulk- `bytes` functions (`join` / `concat` / `slice` / `indexOf` / `split`) and their throughput rationale.
- `strings` library - `join` , the string counterpart of the append-and-join pattern.
- Best practices - the stylistic companion to this page.

Examples

The repository's `examples/` directory (https://github.com/jennifer-language/jennifer/tree/main/examples) holds these plus many more (`showcase.j` , `wordcount.j` , `encoding.j` , `net.j` , `archive.j` , ...) - all golden-tested by `cmd/jennifer/examples_test.go` .

Strings

```
# greeting.j
use io;

def name as string init "Jennifer";
io.printf("hello, " + $name + "!\n");
```

Output:

```
hello, Jennifer!
```

FizzBuzz

```
# fizzbuzz.j
use io;

for (def i as int init 1; $i <= 15; $i = $i + 1) {
    if ($i % 15 == 0) {
        io.printf("FizzBuzz\n");
    } elseif ($i % 3 == 0) {
        io.printf("Fizz\n");
    } elseif ($i % 5 == 0) {
        io.printf("Buzz\n");
    } else {
        io.printf("%d\n", $i);
    }
}
```

Factorial (recursion + parameters)

```
# factorial.j
use io;

func fact(n as int) {
    if ($n == 0) { return 1; }
    return $n * fact($n - 1);
}

for (def i as int init 0; $i <= 8; $i = $i + 1) {
    io.printf("%d! = %d\n", $i, fact($i));
}
```

More substantive examples

For programs that exercise the full feature surface - lists, maps, iteration, the `core` and `strings` libraries - see `examples/showcase.j` (every feature in one file) and `examples/wordcount.j` (word-frequency analyzer with histogram, nested aggregation, and a 3x3 grid demo). Both are part of the golden test suite.

Libraries

Jennifer libraries

Jennifer's standard library is split into topic-based libraries. Each is enabled explicitly with `use NAME;` ; nothing is auto-loaded. This page catalogs every library that ships with the interpreter and links to the reference doc for each.

> **Looking for one specific function?** See the cheatsheet - alphabetical list of every builtin with its library and a one-line description.

> **len** is a language built-in primary, not a library function. Use it from any program with no `use` statement; it's polymorphic over string / list / map / bytes.

The **TinyGo** column reports whether the library runs in full on the constrained `jennifer-tiny` binary (TinyGo-built). A `partial` entry links to ../technical/tinygo.md for the restriction list; the default `jennifer` binary (standard Go) always supports the full surface.

`net` 's "stubs only" is about the **stock** `jennifer-tiny` build, which ships no network driver - not a TinyGo limitation. A tiny build rebuilt with a network stack runs `net` too; see the note on `net` and TinyGo.

Library	Enable with	Tiny Go	Contents
convert	use convert;	full	convert.toInt, convert.toFloat, convert.toString, convert.toBool, convert.typeOf - explicit casts; canonical-only toBool conversion
archive	use archive;	full	tar / zip containers over bytes (no fs). archive.pack/unpack with format "tar"/"zip"/"tar.gz"; bundle is a list of archive.Entry {name, data, mode, mtime}
asn1	use asn1;	full	ASN.1 BER decode / DER encode over an opaque asn1.Value (designed like json), the byte layer under LDAP / SNMP / PKI. Walk with decode/typeOf/tagClass/tagNumber/get/asInt/asString/asOid/...; build with integer/octetString/oid/sequence/tagged/retag/encode. Hand-rolled, dependency-free.
binary	use binary;	full	bulk operations on bytes (the byte-data counterpart to strings/lists). binary.concat, slice, find, split, startsWith, endsWith - non-mutating, value-semantic; pushes per-byte loops into Go for throughput. Named binary because bytes is a reserved type keyword.
compress	use compress;	full	byte-stream compression. compress.pack/unpack for "gzip"/"zlib"/"deflate" (bytes in/out, optional "fast"/"default"/"best" level) + streaming (compress.stream/update/finalize); struct compress.Stream
crc	use crc;	full	crc.compute(b, algo) + streaming (crc.stream/update/finalize) for "crc32", "crc64"; output is big-endian bytes; struct crc.Stream
crypto	use crypto;	full	security primitives. Crypto-grade random crypto.randBytes/randInt; constant-time crypto.hmacEqual; key derivation crypto.hkdf (HKDF) / crypto.pbkdf2 (PBKDF2), algo "sha1"/"sha256"/"sha512". Digests and HMAC live in hash.
encoding	use encoding;	full	introspection (isAscii, lenBytes, lenRunes); binary-to-text toText/fromText for "hex", "base64", "base64-url"; character codecs encode/decode for "ascii", "iso-8859-1", "windows-1252", "ebcdic"
fs	use fs;	full	filesystem I/O. Whole-file readString/readBytes/writeString/writeBytes/appendString/appendBytes; metadata exists/isFile/isDir/stat; dir ops mkdir/mkdirAll/remove/removeAll/rename/list/walk; handles open/readLine/readChars/readBytes/writeString/writeBytes/eof/close; structs fs.Stat, fs.File
hash	use hash;	full	hash.compute(b, algo) + streaming (hash.stream/update/finalize) for "md5", "sha1", "sha256"; struct hash.Stream

httpd	use httpd;	default only	HTTP/1.1 server engine over net/http (TLS, HTTP/2, graceful shutdown). Pull loop: httpd.listen/accept/respond; request accessors method/path/query/header/body; serveFile/serveDir/shutdown. jennifer-tiny returns a friendly stub.
intl	use intl;	full	internationalization: intl.load(lang, catalog) a map of string to string, intl.setLocale/locale, intl.tr(key[, params]) with %name% interpolation and a locale -> base-language -> default -> key fallback. Named intl (letters-only, like JS Intl), not i18n.
io	use io;	full	io.printf, io.sprintf, io.readLine, io.eof, plus the format-verb mini-language
kv	use kv;	full	in-process key/value store with per-key TTL (the no-server local counterpart to memcache / redis): kv.open / openFile (persisted) -> kv.Store, then set / add / get / has / delete / touch / incr / close. Shared handle (safe for spawn). Backs kvstore's in-process option.
json	use json;	full	RFC 8259 json.encode/encodePretty/decode. Structs and map of string to V map to objects, bytes to base64, integral numbers decode to int else float. Decode yields generic values (no map-to-struct coercion).
linalg	use linalg;	full	linear algebra over list of float (vectors) and list of list of float (matrices), the companion to stats: linalg.dot/distance/cross/normalize + norm/scale/add/sub (polymorphic over vector or matrix) + matmul/transpose/trace/determinant/inverse/solve/identity/zeros/shape. Direct algorithms (no gonum), pure-value; a singular / non-finite result is a catchable error.
lists	use lists;	full	lists.push, pop, first, last, head, tail, reverse, sort, contains, concat, slice, shuffle, range - all return a new list
maps	use maps;	full	maps.keys, values, has, delete, merge - all return a new map / list / bool
math	use math;	full	math.abs, min, max, sqrt, pow, floor, ceil, round, rand, randInt, randSeed; constants math.PI, math.E
meta	use meta;	full	meta.VERSION, meta.BUILD - interpreter-self-identity constants
net	use net;	stubs only	TCP connect/listen/accept/readBytes/writeBytes/eof/address, UDP listenUDP/sendTo/recvFrom, DNS lookup/reverseLookup, polymorphic close/address; structs net.Conn, net.Listener, net.UDPSocket, net.Datagram. jennifer-tiny returns friendly errors; use the default jennifer binary for real net I/O.
os	use os;	partial	os.getEnv, os.hasFlag, os.flag, os.run, os.spawn, os.wait, os.poll, os.kill; constants os.PLATFORM, os.ARCH, os.EOL, os.DIRSEP, os.PATHSEP, os.ARGS
path	use path;	full	OS-aware filesystem path manipulation (the string layer paired with fs): path.base, dir, ext, stem, join, clean, isAbs, split. Pure string transforms, no I/O.
regex	use regex;	full	regular expressions over string (RE2 syntax). regex.matches/find/findAll/replace/split/escape + regex.Match struct with positional and named captures. Implicit LRU cache for compiled patterns.
stats	use stats;	full	descriptive statistics over list of int / list of float: stats.mean/median/mode/variance/stddev/percentile/min/max/sum/correlation. Pure-value, dependency-free; variance/stddev are population.
ml	use ml;	full	classical / predictive machine learning on tabular data (scikit-learn-lite): fit / predict models ml.linearRegression/ridge/kNN/naiveBayes/logisticRegression/decisionTree/ randomForest/kMeans/pca + scalers, applied with ml.predict/transform; trainTestSplit/kFold; metrics accuracy/precision/recall/f1/confusionMatrix/rocAuc/rmse/mae/r2. Over stats/linalg; not a deep-learning framework.
strings	use strings;	full	strings.upper, lower, contains, startsWith, endsWith, indexOf, trim, trimLeft, trimRight, replace, repeat, substring, split, chars, join

task	use task;	full	observe and join task of T handles produced by spawn { ... }. task.wait, task.poll, task.discard, task.waitAll, task.waitAny, cancellation (cancel/cancelled), bounded waits (waitTimeout/waitAnyTimeout); pairs with the user-guide concurrency tour
channel	use channel;	full	CSP channels between goroutines: channel.make(capacity), send (copies the value in), recv, close, select (fan-in), len, capacity. The streaming counterpart to task; a shared conduit that copies the data through it. channel is a contextual keyword
term	use term;	default only	terminal control for interactive TUIs: raw mode (term.makeRaw/restore -> term.State), term.size -> term.Size{rows, cols}, raw single-byte reads (term.readByte -> int, -1 at EOF). Over golang.org/x/term; jennifer-tiny returns a friendly stub. Output-only TUIs need only ansi + os.isTerminal.
testing	use testing;	full	test-runner primitives. testing.run/results/reset/report + testing.Result struct. Catches runtime errors, throws, and (uniquely) exit inside test bodies. Three report formats: "text", "tap", "junit". Foundation for the .j-side test framework.
serial	use serial;	default only (Linux)	blocking serial-port I/O with termios line config: serial.open(path, baud) / openWith(path, Options) -> serial.Port, read/write/flush/close. Linux /dev + ioctl; jennifer-tiny and non-Linux return a friendly stub.
spi	use spi;	default only (Linux)	SPI devices: spi.open(path) -> spi.Device, configure(dev, mode, speedHz), full-duplex transfer(dev, bytes) -> bytes, close. Linux-only; stub elsewhere.
i2c	use i2c;	default only (Linux)	the I2C bus: i2c.open(path, addr) -> i2c.Bus, read/write/readReg/writeReg/close. Linux-only; stub elsewhere.
gpio	use gpio;	default only (Linux)	GPIO over /dev/gpiochipN (GPIO v2 ioctls), pin-keyed setup/read/write/release + IN/OUT (mirrors the sysfs gpio module). Linux-only; stub elsewhere.
sql	use sql;	default only	relational-database client over database/sql: MySQL / MariaDB + PostgreSQL (pure-Go drivers). open/query/exec/next + typed row accessors, transactions, prepared statements; placeholder-only binding. First heavyweight library dependency; jennifer-tiny stubs it.
time	use time;	full	instant/duration arithmetic, calendar + Unix accessors, fixed-offset zones (time.zone, time.inZone, time.UTC, time.local), strftime format/parse, ISO round-trip; structs time.Time, time.Duration, time.Zone
toml	use toml;	full	RFC-conformant TOML 1.0 toml.encode/encodePretty/decode. Same opaque toml.Value + read / walk / write surface as json, name for name, addressed by JSON Pointer; adds toml.asDatetime (backed by time.Time) for TOML's native date-times.
uuid	use uuid;	full	RFC 9562 UUIDs. uuid.v4() (random) / uuid.v7() (time-ordered) + parse/isValid/version + constant NIL. The version is a method; draws from the crypto library's crypto-grade random source (unguessable, safe as a security token).
xml	use xml;	full	XML xml.decode/encode/encodePretty over an opaque xml.Value, designed like json/toml. Element tree with ordered attributes + children + mixed text; accessors tag/text/attr/attrs/children, XPath-style get/findAll/has, and a build surface element/setAttr/setText/append.
yaml	use yaml;	full	YAML 1.2 yaml.decode/decodeAll/encode/encodePretty over an opaque yaml.Value, same read / walk / write surface as json/toml, addressed by JSON Pointer. Anchors / aliases / << merge keys, multi-document streams, yaml.asDatetime for timestamps; flow (compact) vs block (pretty) encode. Backed by gopkg.in/yaml.v3 (TinyGo-clean).

A quick taste:

```
use io;
use math;
use meta;
use strings;

io.printf("Jennifer %s\n", meta.VERSION);
io.printf("pi is roughly %f\n", math.PI);
io.printf("math.sqrt(2) = %f\n", math.sqrt(2));
io.printf("upper: %s\n", strings.upper("hello"));
io.printf("len: %d\n", len("hello"));           # language built-in, no import
```

Namespace-first registration

Every library is **namespaced** : each name is reachable as `lib.name(...)` (call) or `lib.NAME` (constant). The library's name doubles as the namespace prefix at the use site.

Aliasing (`use lib as alias;`) is a **rename** , not an addition: after the alias the canonical name no longer resolves at call sites (it errors with a "did you mean *alias* ?" hint). The canonical name is also freed for use as an ordinary identifier, just like Python's `import foo as bar` .

(`core` used to be auto-loaded and exposed `len` / `JENNIFER_VERSION` as bare globals via `RegisterGlobal` / `RegisterGlobalConst` . `len` was promoted to a language built-in keyword, version constants moved to `meta` , and `core` was deleted. The `RegisterGlobal*` API surface remains on `Interpreter` but is unused by any shipping library; it gets removed in a later cleanup pass.)

How libraries are organized

The standard library favors many small, focused libraries over a few large ones. The organizing principle, captured for future extensions:

- Touches I/O (stdin/stdout/files/network/clock) -> `io` (which can later split into `fs` , `net` , `time` as it grows).
- Pure value transformation across kinds -> `convert` .
- Pure numeric -> `math` (includes the non-crypto random helpers).
- String manipulation -> `strings` .
- List manipulation -> `lists` .
- Map manipulation -> `maps` .
- Operating-system glue (env, args, host info) -> `os` .
- Interpreter-self-identity constants (version, build, future build-time / git-sha / GC stats) -> `meta` .
- Time / instants / durations -> `time` (formatting, parsing, and fixed-offset zones).
- Cryptographic-style digests (MD5, SHA-1, SHA-256) -> `hash` . Non-cryptographic checksums (CRC-32, CRC-64) -> `crc` . The split keeps "transport integrity" and "content addressing" visible at the import line.
- Crypto-grade randomness, constant-time comparison, and key derivation (HKDF, PBKDF2) -> `crypto` . The security primitives that need a secure source or a timing-safe operation, kept distinct from the digests in `hash` .
- Byte / string introspection and character-set codecs (ASCII, ISO-8859-1, Windows-1252, EBCDIC IBM-1047) plus hex / base64 binary-to-text -> `encoding` (long-tail codecs parked for later).
- Observing / joining background computations launched with `spawn` -> `task` . Concurrency itself is a language feature (the `spawn` keyword + `task of T` type); the library is just the observation surface.
- Filesystem I/O (whole-file reads/writes, metadata, directory operations, buffered file handles) -> `fs` . Blocking on purpose; non-blocking use composes with `spawn` .
- Network I/O (TCP + UDP sockets, DNS lookups) -> `net` . Blocking calls, same spawn-composition story as `fs` . The **stock** `jennifer-tiny` binary returns friendly "use the default `jennifer` " errors because it ships no network driver - a build choice, not a TinyGo limitation; a tiny build with a network stack runs `net` too.
- Regular expressions over `string` -> `regex` . RE2 syntax (Go's `regexp` engine); implicit LRU cache. Pure string processing, no other library dependencies.

- Test-runner primitives (name-based method dispatch, per-process result accumulator, format dispatcher for text/TAP/JUnit) -> `testing` . Lives here because Jennifer has no function references yet; the `.j` -side assertion vocabulary and CLI harness ship on top of these primitives.
- A genuinely new topic with **five or more** functions / constants -> a new library. Fewer than five names fold into the most-related existing library (the non-crypto random helpers were the first case the rule caught - they live under `math.rand*` rather than getting their own library).
- A single function with no clear topic -> the most-related existing library.
- Genuinely polymorphic structural primitives that every program needs (`len`) -> language built-in keyword, not a library. The bar is intentionally high; `len` is the only one.

Naming convention

Library names look mixed at first glance - `strings` is plural but `math` is singular. The rule:

- **Plural for count nouns** : when the library operates on instances of something you can have multiples of. `strings` , `lists` , `maps` , `bytes` , `files` .
- **Singular for mass nouns and conceptual wholes** : `math` , `meta` , `time` , `regex` (planned).
- **Bare verb when the library is named for what it does** , not what it touches: `convert` .
- **Idiomatic abbreviations are fine** : `os` , `fs` , `net` , `regex` .

Three practical constraints reinforce the count/mass rule:

1. Type keywords are reserved. `string` , `int` , `float` , `bool` , `list` , `map` , `null` cannot be library names because they tokenize as type tokens, not IDENTs. The plural form (`strings` , `lists` , `maps`) sidesteps this naturally.
2. The rule matches Go's stdlib: `strings` and `bytes` are plural; `math` , `io` , `os` are singular. Since the interpreter is written in Go, the convention transfers cleanly to library author intuition.
3. Within a library, function names are lowercase / camelCase (`upper` , `startsWith` , `typeOf`). Constants are uppercase (`PI` , `E` , `VERSION` , `PLATFORM`).

For implementation notes on how libraries register themselves with the interpreter (`RegisterNamespaced` , the `use` -gated lookup), see ../technical/interpreter.md > Builtins and libraries.

For canonical terminology (library vs module, function vs method, list vs array, ...), see ../glossary.md. This page uses the terms in that table.

Cheatsheet - all builtins at a glance

Alphabetical index of every standard-library function and constant. Use it when you know the *name* and want to know *which library* and *how to call it* ; use each library's own page when you want to read about a topic. Each row's library prefix links to the per-library doc.

The table covers what ships with the interpreter. New entries land here at the same time as the per-library doc - it's a flat lookup view, not authoritative.

Functions

Call	What it does
archive.pack(entries, format)	Bundle a list of archive.Entry into bytes; format "tar"/"zip"/"tar.gz".
archive.unpack(b, format)	Read a bundle back into a list of archive.Entry.
asn1.decode(b) / .encode(v)	Parse BER bytes into an opaque asn1.Value / serialise one back to DER bytes.
asn1.typeOf/tagClass/tagNumber/isConstructed(v[, ptr])	Element type name / class / numeric tag / constructed-ness; pointer tokens are child indices ("/0/2").
asn1.get(v[, ptr]) / .has(v, ptr) / .length(v[, ptr])	Sub-element as asn1.Value / whether a pointer resolves / child count of a constructed node.
asn1.asInt/asBool/asString/asBytes/asOid/isNull(v[, ptr])	Read a leaf: integer/enum, boolean, string type or octets, raw octets, dotted OID, or whether NULL.
asn1.integer/enumerated/boolean/null/octetString(...)	Build a primitive element.
asn1.utf8String/printableString/ia5String/oid(s)	Build a string / OID element.
asn1.sequence(items) / .set(items)	Build a constructed element from a list of asn1.Value.
asn1.tagged(class, n, v) / .retag(class, n, v)	EXPLICIT (wrap) / IMPLICIT (overwrite tag) context/application tagging; class "context"/"application"/...
binary.concat(a, b)	Join two bytes into a fresh bytes (O(len a + len b); avoid in an accumulation loop - use net.readAll/readN).
binary.contains(haystack, needle)	Whether needle occurs in bytes haystack (boolean sibling of indexOf).
binary.endsWith(b, suffix)	True iff bytes b ends with suffix.
binary.indexOf(haystack, needle)	Byte index of the first needle in haystack; -1 if absent, 0 for an empty needle. Native-speed scan.
binary.join(parts [, sep])	Concatenate a list of bytes into one bytes in O(n) (optional bytes sep between pieces). The byte-data strings.join.
binary.slice(b, start [, end])	Half-open byte range [start, end); end defaults to len(b). Out-of-range / start>end errors.
binary.split(b, sep)	Split bytes on a non-empty sep -> list of bytes (e.g. a MIME body on its boundary, one Go pass).
binary.startsWith(b, prefix)	True iff bytes b begins with prefix.
channel.make(capacity)	New channel of T; capacity 0 = unbuffered, n = buffered. Type from the binding.
channel.send(ch, value)	Deep-copy value in and send (blocks per capacity); send on a closed channel is a catchable error.

channel.recv(ch)	Block and return the next value; throws (catchable) on a closed and drained channel.
channel.close(ch)	Close the channel; double-close is a catchable error. Buffered values still drain.
channel.select(chs)	Fan-in: next value from any open channel; throws when all are closed and drained.
channel.len(ch) / .capacity(ch)	Buffered value count / buffer capacity.
compress.discard(stream)	Drop a streaming compressor without returning output; releases its state.
compress.finalize(stream)	Close a streaming compressor; returns all compressed bytes.
compress.pack(b, algo [, level])	Compress bytes; algo "gzip"/"zlib"/"deflate", optional level "fast"/"default"/"best".
compress.stream(algo [, level])	Start a streaming compressor -> compress.Stream.
compress.unpack(b, algo)	Decompress bytes with algo.
compress.update(stream, b)	Feed one chunk into a streaming compressor.
convert.fromCodepoint(n)	One-rune string for Unicode code point n (whole range, 1-4 UTF-8 bytes); errors on out-of-range / surrogate.
convert.toBool(v)	Canonical conversion to bool (0/1, 0.0/1.0, "true"/"false").
convert.toCodepoint(char)	Unicode code point (int) of a one-rune string; errors unless exactly one code point (not a grapheme cluster).
convert.toFloat(v)	Convert to float (int->float, float identity, string parses, bool->1.0/0.0).
convert.toInt(v)	Convert to int (float truncates toward zero, string parses, bool->1/0).
convert.toString(v)	Convert to string (always succeeds; uses the value's display form).
convert.typeOf(v)	Runtime kind as string ("int", "float", "string", "bool", "null", "list", "map", "object").
convert.objectType(v)	Specific registered name of an opaque object (e.g. "json.Value"); errors on a non-object.
crc.compute(b, algo)	One-shot checksum. algo is "crc32" or "crc64". Returns big-endian bytes (4 or 8).
crc.discard($s)	Drop a crc.Stream without computing its checksum; releases its state.
crc.finalize($s)	Final checksum as big-endian bytes; consumes the handle.
crc.stream(algo)	Allocate a crc.Stream for algo; feed chunks via crc.update then close with crc.finalize.
crc.update($s, $bytes)	Feed one chunk into a crc.Stream (mutates by side effect).
crypto.encrypt(key, plaintext) / .decrypt(key, box)	AES-256-GCM authenticated encryption. 32-byte key; a fresh nonce is prepended (nonce||ct||tag). decrypt of a tampered / wrong-key box is a catchable auth error.

crypto.hkdf(secret, salt, info, length, algo)	Derive length bytes from a high-entropy secret (HKDF, RFC 5869); algo "sha1"/"sha256"/"sha512"; salt/info may be empty.
crypto.hmacEqual(a, b)	Constant-time equality of two bytes (MAC comparison); unequal lengths return false.
crypto.signKeypair() / .sign($priv, msg) / .verify($pub, msg, sig)	Ed25519. signKeypair -> crypto.Keypair{public, private}; sign -> 64-byte signature; verify -> bool (error only on a wrong-length key/sig).
crypto.rsaSign($privPem, msg, algo) / .rsaVerify($pubPem, msg, sig, algo)	RSASSA-PKCS#1 v1.5 over PEM keys (for JWT RS*); algo "sha256"/"sha384"/"sha512". Default binary only.
crypto.ecdsaSign($privPem, msg, algo) / .ecdsaVerify($pubPem, msg, sig, algo)	ECDSA over PEM keys, JOSE R||S signature (for JWT ES*); curve from the key. Default binary only.
crypto.rsaGenerateKey(bits) / .ecGenerateKey(curve)	Generate an RSA (2048/3072/4096) / EC ("p256"/"p384"/"p521") private key as PEM. Default binary only.
crypto.jwkPublic($privPem) / .jwkToPem($jwkJson) / .csr($privPem, domains)	Canonical public JWK JSON (RFC 7638; SHA-256 it for the thumbprint); the inverse jwkToPem (public JWK -> PUBLIC KEY PEM for rsaVerify / ecdsaVerify, so a JWKS kid resolves to a key); DER PKCS#10 CSR over a list of string of domains. For ACME / JWT. Default binary only.
crypto.pbkdf2(password, salt, iterations, keyLen, algo)	Stretch a password into a keyLen-byte key (PBKDF2, RFC 8018); algo "sha1"/"sha256"/"sha512".
crypto.randBytes(n)	n crypto-grade random bytes (n >= 0).
crypto.randInt(lo, hi)	Uniform crypto-grade int in the inclusive range [lo, hi] (rejection-sampled, unbiased; unseedable).
encoding.codecs()	Canonical character-codec names in registration order.
encoding.decode(b, codec)	Decode bytes from a character codec to a Jennifer string.
encoding.encode(s, codec)	Encode a Jennifer string into a character codec's bytes.
encoding.fromText(s, format)	Decode a binary-to-text format. format: "hex", "base32", "base32-hex", "base64", "base64-url", "ascii85", "z85", "quoted-printable", "uri-percent", "uri-form".
encoding.isAscii(b)	True iff every byte in b is < 0x80.
encoding.lenBytes(s)	UTF-8 byte length of s (pair with len(s) for rune count).
encoding.lenRunes(b)	Rune count of valid UTF-8 bytes; errors on invalid UTF-8.
encoding.toText(b, format)	Encode bytes as printable text. format: "hex", "base32", "base32-hex", "base64", "base64-url", "ascii85", "z85", "quoted-printable", "uri-percent", "uri-form".
fs.appendBytes(path, content)	Append bytes to path; creates the file if missing.
fs.appendString(path, content)	Append UTF-8 string to path; creates the file if missing.

fs.chmod(path, mode)	Set permission bits (e.g. 0o600); Unix/Linux. Rejected outside [0, 0o7777].
fs.chown(path, uid, gid)	Set owner / group (-1 leaves unchanged); Unix/Linux, usually needs privilege.
fs.close($f)	Close an fs.File handle; removes it from the registry.
fs.eof($f)	True iff the next read on $f would error or return partial. Sticky.
fs.exists(path)	True if path resolves; permission errors still surface.
fs.hasEvent($w)	Whether an fs.Watcher has a queued change event (non-blocking peek).
fs.isDir(path)	True iff path exists and is a directory.
fs.isFile(path)	True iff path exists and is a regular file.
fs.list(path)	Sorted entry names in path. Non-recursive; returns list of string.
fs.mkdir(path)	Create a single directory; errors if any parent is missing.
fs.mkdirAll(path)	Create path and every missing parent (like mkdir -p).
fs.next($w)	Block for the next fs.Event (path, kind, isDir) from an fs.Watcher; errors when the watcher is closed.
fs.open(path, mode)	Open path and return an fs.File. mode: "read", "write", "append".
fs.readBytes(path) / .readBytes($f, n)	Whole-file read (1 arg) or up to n bytes from handle (2 args). Partial + sticky-EOF on short handle reads.
fs.readChars($f, n)	Up to n runes from handle, UTF-8 decoded. Partial + sticky-EOF on short reads.
fs.readLine($f)	One line from handle, \r\n / \n stripped. Errors on EOF - check fs.eof first.
fs.readString(path)	Whole file as UTF-8; invalid UTF-8 is a positioned runtime error.
fs.realpath(path)	Absolute, symlink-resolved canonical path (resolve os.ARGS[0] to find files beside the real script). Missing / unresolvable errors.
fs.readlink(path)	A symlink's target, verbatim (not resolved; use fs.realpath to resolve). Non-symlink / missing errors.
fs.symlink(target, linkPath)	Create a symlink at linkPath pointing to target (like ln -s target linkPath). Existing linkPath errors.
fs.remove(path)	Delete one file or empty directory. Non-empty dir errors.
fs.removeAll(path)	Recursive delete. Explicit second verb (no-footguns stance).
fs.rename(old, new)	Same-filesystem rename; cross-fs is a boundary error.
fs.stat(path)	Returns fs.Stat (path, size, isDir, mtimeNanos, mode). Missing path errors.

fs.sync($f)	Flush a write/append handle's data to the storage device (fsync); handle stays open. The "safe to remove the stick" step.
fs.walk(path)	Depth-first, sorted, includes path. Returns list of fs.Stat. Skips symlinks.
fs.watch(path[, intervalMs])	Watch a file / dir (recursive) for changes -> fs.Watcher; pull with fs.next / fs.hasEvent, stop with fs.close. mtime polling.
fs.writeBytes(path, content) / .writeBytes($f, b)	Whole-file overwrite (path form) or write via handle (fs.File form).
fs.writeString(path, content) / .writeString($f, s)	Whole-file overwrite (path form) or write via handle (fs.File form).
fs.makeTempFile([dir[, prefix[, suffix]]])	Create a unique empty file (atomic, 0600); returns its path. dir="" = system temp; parent must exist.
fs.makeTempDir([dir[, prefix]])	Create a unique directory (atomic, 0700); returns its path. Only the leaf is created (not mkdir -p).
gpio.setup(pin, direction)	Request pin (0..63) with gpio.IN / gpio.OUT on the current chip. Linux only.
gpio.read(pin) / .write(pin, value)	Read a line (0/1) / drive an output line (0 or 1).
gpio.release(pin) / .chip(path)	Free a requested line / select the gpiochip device (default /dev/gpiochip0).
hash.compute(b, algo)	One-shot digest. algo is "md5", "sha1", "sha256", "sha384", or "sha512". Returns raw bytes.
hash.hmac(key, message, algo)	Keyed-hash MAC (RFC 2104) over the same algorithms; raw bytes out. For JWT / TOTP / SigV4 / webhook signatures.
hash.equal(a, b)	Constant-time equality of two bytes (for MAC / token / digest checks); false on a length or content difference.
hash.discard($s)	Drop a hash.Stream without computing its digest; releases its state.
hash.finalize($s)	Final digest as bytes; consumes the handle (later calls error).
hash.stream(algo)	Allocate a hash.Stream for algo; feed chunks via hash.update then close with hash.finalize.
hash.update($s, $bytes)	Feed one chunk into a hash.Stream (mutates by side effect).
httpd.listen(addr) / .listenTLS(addr, cert, key)	Start an HTTP / HTTPS server -> httpd.Server (":0" = ephemeral port). Default binary only.
httpd.address($srv) / .shutdown($srv)	Bound address of a server / graceful drain (unblocks parked accept).
httpd.accept($srv)	Block for the next request -> httpd.Request (the pull loop). Errors once the server is shut down.
httpd.method($req) / .path($req) / .query($req, name) / .header($req, name) / .body($req) / .remoteAddr($req)	Read the accepted request (query / header -> "" if absent; body -> bytes).
httpd.setHeader($req, name, value) / .respond($req, status, body)	Set a response header / send the response once (body is string or bytes).
httpd.serveFile($req, path) / .serveDir($req, root)	Answer with a file / the file under root for the request path (.. cannot escape root).

i2c.open(path, addr)	Open an I2C bus and select 7-bit slave addr -> i2c.Bus. Linux only.
i2c.read($bus, n) / .write($bus, data)	Read n raw bytes / write raw bytes to the selected slave.
i2c.readReg($bus, reg, n) / .writeReg($bus, reg, data)	Register read (set pointer, read back) / register write.
i2c.close($bus)	Close the bus.
intl.load(lang, catalog)	Merge a map of string to string into the catalog for lang; the first language loaded is the default.
intl.setLocale(lang) / .locale()	Set / read the current locale (e.g. "de-AT"; locale() is "" until set).
intl.tr(key[, params])	Translate key (fallback: locale -> base language -> default -> the key); params (a map) fills %name% placeholders (%% escapes a literal %).
io.eof()	True if and only if the next io.readLine() would error. Pair with while (not io.eof()) {...}.
io.printf(format, args...)	Format-string write to stdout. Verbs: %d %f %s %t %v %%; per-verb |key=value modifiers (pad, prec, base, null=*, ...).
io.printf(value)	Write a value's display form to stdout.
io.eprintf(format, args...)	Like printf, but writes to stderr (diagnostics / logs that must not mix into stdout).
io.readLine()	Read one line from stdin (trailing newline stripped). Errors at EOF - check io.eof() first.
io.readLine(prompt)	Same as io.readLine() but writes prompt to stdout first.
io.readLines()	Slurp all remaining stdin lines into a list of string (newlines stripped, LF/CRLF, OS-independent). Use the readLine loop for large streams.
io.sprintf(format, args...)	Format-string version of sprintf. Same verbs and |key=value modifiers as printf.
io.sprintf(value)	Display-form of a value, returned as a string (doesn't write).
len(v) (language built-in)	Structural length: rune count (string), element count (list), entry count (map), byte count (bytes).
json.decode(s)	Parse JSON text into an opaque json.Value handle (walk it with the accessors below).
json.encode(v)	Compact JSON string for an encodable value (struct/map -> object, bytes -> base64, json.Value round-trips; task / non-string keys error).
json.encodePretty(v)	Like encode, 2-space indented.
json.typeOf(v[, ptr])	JSON type at an optional JSON Pointer: null bool int float string list map.
json.get(v[, ptr])	Sub-node at a JSON Pointer, as a json.Value (walk stays opaque; no pointer = the node itself).
json.has(v, ptr)	Whether the JSON Pointer resolves to an existing node.
json.keys(v[, ptr])	list of string keys of the addressed map, in document order.

json.length(v[, ptr])	Element count of a list / entry count of a map at the pointer.
json.asInt(v[, ptr]) / asFloat / asString / asBool	Extract the addressed leaf as a typed value (strict; asFloat promotes an integral number).
json.isNull(v[, ptr])	Whether the addressed node is JSON null.
json.map() / .list()	A fresh empty JSON map / list json.Value - the explicit start of a document (writes never auto-vivify).
json.set(v, ptr, val)	Non-mutating: upsert a map key or replace an in-range list index; returns a new json.Value. Strict (no missing intermediates).
json.insert(v, ptr, val)	Insert into a list before index ptr (or - = at end); returns a new handle.
json.append(v, ptr, val)	Push onto the list addressed by ptr (sugar for insert at /.../-).
json.remove(v, ptr)	Drop the map key or list element at ptr; returns a new handle.
json.move(v, from, to)	Relocate the subtree at from to to (read, remove, then set).
kv.open()	A fresh in-memory kv.Store (reset each run).
kv.openFile(path)	A kv.Store persisted to path (survives across runs; flush per mutation).
kv.set(store, key, value, ttl)	Store value, expiring in ttl seconds (0 = never).
kv.add(store, key, value, ttl)	Store only if absent; whether it stored (bool).
kv.get(store, key)	The value, or "" when absent / expired.
kv.has(store, key)	Whether the key is present and unexpired (bool).
kv.delete(store, key)	Remove the key; whether it existed (bool).
kv.touch(store, key, ttl)	Re-arm the expiry; whether it existed (bool).
kv.incr(store, key, delta)	Add signed delta to the numeric value; new value, or -1 when absent (not created).
kv.close(store)	Drop the store and free its handle.
linalg.dot(a, b) / .distance(a, b)	Dot product / Euclidean distance of vectors (list of float) -> float. Length-checked.
linalg.cross(a, b) / .normalize(v)	3-D cross product of length-3 vectors / the unit vector v / norm(v) (zero vector errors) -> list of float.
linalg.norm(x)	L2 norm of a vector, or the Frobenius norm of a matrix -> float (polymorphic).
linalg.scale(x, s) / .add(a, b) / .sub(a, b)	Scalar multiply / element-wise sum / difference of a vector or a matrix (same shape).
linalg.matmul(a, b)	General product: matrixmatrix -> matrix; matrixvector or vectormatrix -> vector; vectorvector errors (use dot).
linalg.transpose(m) / .trace(m)	Transpose of a list of list of float / trace (diagonal sum, square) -> float.
linalg.determinant(m)	Determinant of a square matrix -> float (singular is 0).
linalg.inverse(m) / .solve(a, b)	Matrix inverse / solve a x = b; a singular matrix is a catchable error.

linalg.identity(n) / .zeros(r, c) / .shape(m)	The n x n identity / an r x c zero matrix / [rows, cols] of a matrix (list of int).
lists.all(xs, fn)	True if the func fn(x) is true for every element (true for empty). Short-circuits.
lists.any(xs, fn)	True if the func fn(x) is true for any element. Short-circuits.
lists.concat(a, b)	New list with a's elements followed by b's.
lists.contains(xs, item)	True if item appears in xs (haystack, needle).
lists.filter(xs, fn)	New list of the elements where the func fn(x) returns true.
lists.find(xs, fn)	First element where the func fn(x) is true; errors (catchable) if none.
lists.first(xs)	Element at index 0. Empty input errors.
lists.head(xs, n)	New list of the first n elements.
lists.last(xs)	Element at the last index. Empty input errors.
lists.map(xs, fn)	New list of fn(x) for each element (fn a func); result type checked at the binding.
lists.pop(xs)	New list without the last element. Empty input errors.
lists.push(xs, item)	New list with item appended.
lists.range(start, end[, step])	Half-open list of consecutive ints; end excluded; step must match direction.
lists.reduce(xs, fn, init)	Fold left with the func fn(acc, x), starting from init.
lists.reverse(xs)	New list with elements reversed.
lists.shuffle(xs)	Fisher-Yates; respects math.randSeed. Non-mutating.
lists.slice(xs, start[, end])	New sublist [start, end); end defaults to len(xs).
lists.sort(xs)	New ascending-sorted list. Numeric / string / bool elements; mixed errors.
lists.sortBy(xs, keyFn)	New list sorted ascending by the key the func keyFn(x) returns (a comparable scalar).
lists.tail(xs, n)	New list of the last n elements.
maps.delete(m, key)	New map without key. Missing key errors (strict at boundaries).
maps.has(m, key)	True if map m contains key. The non-erroring companion to $m[key].
maps.keys(m)	List of keys in insertion order.
maps.merge(a, b)	New map; b's entries layered on top of a.
maps.values(m)	List of values in insertion order.
ml.linearRegression(X, y) / .ridge(X, y, alpha)	Fit OLS / L2-ridge regression -> ml.Model (regressor).
ml.kNN(X, y, k) / .naiveBayes(X, y)	Fit a k-NN / Gaussian naive-Bayes classifier -> ml.Model.
ml.logisticRegression(X, y [, lr [, epochs]])	Fit a binary (0/1) logistic classifier (gradient descent) -> ml.Model.
ml.decisionTree(X, y [, maxDepth]) / .randomForest(X, y [, nTrees [, maxDepth]])	Fit a CART tree / bagged forest classifier -> ml.Model.

ml.kMeans(X, k [, maxIter])	Fit k-means clustering (k-means++ seed) -> ml.Model. Uses math.randSeed.
ml.pca(X, nComponents)	Fit PCA (covariance eigendecomposition) -> ml.Model (transform).
ml.standardScaler(X) / .minMaxScaler(X)	Fit a z-score / [0,1] feature scaler -> ml.Model (transform).
ml.predict(model, X)	Predicted labels (int, classifier/cluster) or values (float, regressor) per row.
ml.transform(model, X) / .predictProba(model, X)	Transform features (scaler/PCA) / positive-class probability (logistic).
ml.free(model)	Drop a fitted model from the registry to free memory.
ml.trainTestSplit(X, y, testFraction)	Shuffle + split -> ml.Split{trainX,trainY,testX,testY}. Uses math.randSeed.
ml.kFold(nSamples, k)	k contiguous CV folds -> list of ml.Fold{trainIdx, testIdx}.
ml.accuracy(yTrue, yPred)	Fraction of exact matches.
ml.precision/recall/f1(yTrue, yPred [, positive])	Binary classification metrics for the positive label (default 1).
ml.confusionMatrix(yTrue, yPred)	ml.Confusion{labels, matrix} (rows=true, cols=predicted).
ml.rocAuc(yTrue, scores)	Binary ROC-AUC from 0/1 labels and predicted scores (tie-aware).
ml.rmse/mae/r2(yTrue, yPred)	Regression error metrics (r2 = coefficient of determination).
ml.lasso(X, y, alpha)	Fit L1-regularized regression (coordinate descent; sparse coefficients) -> ml.Model.
ml.kNNRegressor(X, y, k) / .decisionTreeRegressor(X, y [, maxDepth])	Regression variants of k-NN / CART (predict a continuous value).
ml.randomForestRegressor(X, y [, nTrees [, maxDepth]])	Bagged regression trees -> ml.Model (regressor).
ml.coefficients(model) / .intercept(model)	Read a linear/logistic model's learned weights / bias.
ml.centroids(model) / .components(model) / .explainedVariance(model)	k-means centres / PCA axes / PCA per-component variance ratios.
ml.featureImportances(model)	Gini feature importances of a tree / forest (sum to 1).
ml.polynomialFeatures(X, degree)	Expand X to all monomials up to degree (+ bias); stateless. Degree in [1,8].
ml.mse(yTrue, yPred) / .logLoss(yTrue, probas)	Mean-squared error / binary cross-entropy.
math.abs(x)	Absolute value of x (int->int, float->float).
net.accept($listener)	Block until a client connects to $listener; return the new net.Conn.
net.address($h)	Polymorphic. Conn -> peer address; Listener / UDPSocket -> local bound address.
net.close($h)	Polymorphic. Closes a net.Conn, net.Listener, or net.UDPSocket.
net.connect(address[, timeoutMs])	TCP client: dial "host:port" and return a net.Conn. Optional timeoutMs bounds connection establishment.

net.connectTLS(address[, net.TLSOptions][, timeoutMs])	TLS client: dial "host:port" + handshake, verifying the cert against the host. net.TLSOptions for caCert / skipVerify; optional timeoutMs bounds the dial + handshake.
net.startTLS($conn[, net.TLSOptions][, timeoutMs])	Upgrade an open plaintext net.Conn to TLS in place (STARTTLS); host reused from connect; same handle. Optional timeoutMs bounds the handshake.
net.eof($conn)	True iff the next read on $conn would return partial or fail. Sticky.
net.listen(address)	Bind TCP "host:port" (use ":0" for ephemeral). Returns a net.Listener.
net.listenUDP(address)	Bind a UDP socket. Returns a net.UDPSocket; usable as both client and server.
net.lookup(host)	DNS: resolve host to a list of string IPs.
net.readAll($conn [, maxBytes [, idleTimeoutMs]])	Read to EOF, returning the whole stream as one bytes in a single Go loop (whole-body / object download). maxBytes>0 caps (catchable), idleTimeoutMs>0 re-arms a per-chunk read deadline.
net.readBytes($conn, n)	Read up to n bytes; blocks for at least one byte. Sticky-EOF on close.
net.readN($conn, n [, idleTimeoutMs])	Read exactly n bytes for a length-prefixed frame; a close before n bytes is a catchable error, not a truncated return.
net.recvFrom($sock, n)	Block for one UDP datagram, up to n bytes. Returns net.Datagram{data, peer}.
net.reverseLookup(ip)	Reverse DNS: IP address to a list of string of hostnames.
net.sendTo($sock, peer, bytes)	Send one UDP datagram to peer ("host:port").
net.setDeadline($conn, ms)	Arm a read/write deadline ms ms out (0 clears). A read past it fails with a catchable read timed out.
net.writeBytes($conn, bytes)	Blocking write of every byte to a net.Conn.
regex.escape(s)	Escape RE2 metacharacters so s matches literally when used as a pattern.
regex.find(pattern, s)	First match as regex.Match; sentinel with start=-1 if no match.
regex.findAll(pattern, s)	Every non-overlapping match; returns list of regex.Match.
regex.matches(pattern, s)	True iff pattern matches somewhere in s.
regex.replace(pattern, s, replacement)	Replace every match. $1, ${name} expand to captured groups; $$ is a literal $.
regex.split(pattern, s)	Split s at every match; returns list of string.
serial.open(path, baud) / .openWith(path, opts)	Open a serial port (raw 8N1, or full serial.Options) -> serial.Port. Linux only.
serial.read($port, n) / .write($port, data)	Read up to n bytes (blocks for >=1) / write bytes (-> count).
serial.flush($port) / .close($port)	Discard buffered I/O / close the port.
spi.open(path) / .configure($dev, mode, speedHz)	Open an SPI device -> spi.Device / set clock mode (0..3) and speed. Linux only.

spi.transfer($dev, data) / .close($dev)	Full-duplex exchange (out and in together) / close the device.
sql.open(driver, dsn) / .close($c)	Open a MySQL / Postgres connection -> sql.Connection / close it. Default binary only.
sql.query(target, sql, params...) / .exec(target, sql, params...)	Query -> sql.Rows / statement -> sql.Result{affected, lastId}. target = Connection or Tx; params bind through placeholders.
sql.next($rows) / .columns($rows) / .closeRows($rows)	Advance the cursor (false at end) / column names / close early.
sql.asInt / .asFloat / .asString / .asBool / .asBytes($rows, col) / .isNull($rows, col)	Read the current row's column (name or index), typed; a NULL column is an error (check isNull).
sql.begin($c) / .commit($tx) / .rollback($tx)	Transaction: begin -> sql.Tx (a query/exec target), then commit / rollback.
sql.setQueryTimeout(ms)	Set the client query/read deadline (ms); 0 disables it for a long cursor read. Default 30 000, process-wide.
sql.prepare($c, sql) / .queryStmt($s, ...) / .execStmt($s, ...) / .closeStmt($s)	Prepared-statement lifecycle.
math.acos(x)	Arccosine (radians); errors outside [-1, 1].
math.acosh(x)	Inverse hyperbolic cosine; errors for x < 1.
math.asin(x)	Arcsine (radians); errors outside [-1, 1].
math.asinh(x)	Inverse hyperbolic sine.
math.atan(x)	Arctangent (radians).
math.atan2(y, x)	Two-argument arctangent, quadrant-aware (radians).
math.atanh(x)	Inverse hyperbolic tangent; errors outside (-1, 1).
math.beta(a, b)	Beta function B(a,b)=gamma(a)gamma(b)/gamma(a+b); a, b > 0.
math.cbrt(x)	Real cube root (handles negatives).
math.ceil(x)	Smallest int >= x. Accepts int (identity) or float.
math.comb(n, k)	Binomial coefficient nCr (exact int); k>n is 0; errors on overflow.
math.cos(x)	Cosine (radians).
math.cosh(x)	Hyperbolic cosine; errors on overflow.
math.erf(x)	Error function.
math.erfc(x)	Complementary error function (1 - erf).
math.exp(x)	e^x; errors on overflow.
math.expm1(x)	e^x - 1, accurate for small x.
math.factorial(n)	n! (exact int); errors on negative n or overflow (n > 20).
math.floor(x)	Largest int <= x. Accepts int (identity) or float.
math.gamma(x)	Gamma function; errors at the poles (0, negative ints) and on overflow.
math.gcd(a, b)	Greatest common divisor (non-negative; gcd(0,0)=0).
math.hypot(x, y)	sqrt(x^2 + y^2) without intermediate overflow.

math.lbeta(a, b)	Natural log of the beta function, ln B(a,b); a, b > 0.
math.lcm(a, b)	Least common multiple (lcm(x,0)=0); errors on overflow.
math.lgamma(x)	ln
math.ln(x)	Natural logarithm; errors for x <= 0.
math.log(x, base)	Logarithm of x to an arbitrary base; errors for non-positive x/base or base 1.
math.log10(x)	Base-10 logarithm; errors for x <= 0.
math.log1p(x)	ln(1 + x), accurate for small x; errors for x <= -1.
math.log2(x)	Base-2 logarithm; errors for x <= 0.
math.max(a, b)	Larger of two numbers; mixed int/float promotes to float.
math.min(a, b)	Smaller of two numbers; mixed int/float promotes to float.
math.perm(n, k)	k-permutations of n (nPr, exact int); k>n is 0; errors on overflow.
math.pow(x, y)	x raised to y; always float. Errors on NaN/Inf-producing inputs.
math.rand()	Float in [0, 1) from the shared seedable (non-crypto) source.
math.randInt(lo, hi)	Int in [lo, hi] inclusive; errors if lo > hi.
math.randSeed(n)	Reseed the shared source for reproducible runs (also drives lists.shuffle; uuid / password use crypto instead).
math.regBetaI(x, a, b)	Regularized incomplete beta I_x(a,b) in [0,1] (CDF engine for beta / t / F / binomial); 0<=x<=1, a,b>0.
math.regGammaP(a, x)	Regularized lower incomplete gamma P(a,x) in [0,1] (gamma / chi-square CDF); a>0, x>=0.
math.regGammaQ(a, x)	Regularized upper incomplete gamma Q(a,x) = 1 - P(a,x).
math.round(x)	Round to nearest int (half away from zero).
math.sign(x)	-1 / 0 / 1 with the sign of x, preserving its type (int -> int, float -> float).
math.sin(x)	Sine (radians).
math.sinh(x)	Hyperbolic sine; errors on overflow.
math.sqrt(x)	Square root; always float. Errors on negative input.
math.tan(x)	Tangent (radians).
math.tanh(x)	Hyperbolic tangent.
math.trunc(x)	Round toward zero -> int; int argument is the identity.
os.flag(name)	Value following name in os.ARGS, or "" if absent / at end. Exact-match (no --foo=bar parsing).
os.getEnv(name)	Read environment variable name. Unset -> empty string, no error.
os.setEnv(name, value)	Set environment variable name for this process (and children it spawns). Invalid name errors.

os.hasFlag(name)	True if name appears as an exact element of os.ARGS.
os.isTerminal(stream)	Is stream ("stdout"/"stderr"/"stdin") an interactive terminal? Pipe/file -> false.
os.cwd()	Absolute path of the current working directory.
os.homeDir()	Current user's home directory ($HOME / %USERPROFILE%).
os.tempDir()	Temp-file directory ($TMPDIR//tmp; %TMP% on Windows). Never errors.
os.catchSignal(name) / .gotSignal(name)	Opt into trapping a Unix signal ("int"/"term"/"hup"/"usr2") / poll-and-clear whether it arrived. Cooperative; "usr1" reserved for kill -USR1 diagnostics.
os.kill(p)	Send SIGTERM to spawned process $p.
os.poll(p)	True if spawned process $p has exited (a following os.wait returns immediately).
os.release(p)	Drop a finished process handle from the registry (frees captured output); errors if $p still runs.
os.run(argv)	Blocking: run argv to completion, return os.Result{exitCode, stdout, stderr}.
os.spawn(argv)	Non-blocking: start argv, return os.Process{pid} handle.
os.wait(p)	Block until spawned process $p exits; return os.Result. Idempotent.
path.base(p)	Last element of p. "" -> ".", "/" -> "/". OS-aware; not a filename sanitizer.
path.dir(p)	All but the last element of p, cleaned. "c.txt" -> ".".
path.ext(p)	File extension incl. the leading dot (".txt"), or "".
path.stem(p)	Base name of p without its extension.
path.join(a, b, ...)	Join >= 1 elements with the separator and clean; empty elements dropped. Portable alternative to hardcoding "/".
path.clean(p)	Shortest path equivalent to p (collapses ., .., repeated separators). "" -> ".".
path.isAbs(p)	Whether p is absolute (bool).
path.split(p)	[dir, file] where dir keeps its trailing separator, so dir + file == p.
stats.mean(xs) / .median(xs)	Arithmetic mean / median of a list of int|float -> float. Empty errors.
stats.variance(xs) / .stddev(xs)	Population variance / standard deviation -> float.
stats.sampleVariance(xs) / .sampleStddev(xs)	Sample (Bessel n-1) variance / stddev -> float. Needs >= 2 elements.
stats.mad(xs)	Median absolute deviation - robust spread -> float.
stats.skewness(xs) / .kurtosis(xs)	Shape: population skewness / excess kurtosis (normal -> 0) -> float. Non-constant, >= 2.
stats.percentile(xs, p)	The p-th percentile (p in [0,100], linear interpolation) -> float.

stats.quartiles(xs) / .iqr(xs)	[Q1,Q2,Q3] (list of float) / interquartile range p75-p25 (float).
stats.min(xs) / .max(xs) / .range(xs)	Smallest / largest element / spread max-min; all preserve int/float kind. Empty errors.
stats.sum(xs)	Sum: int for an all-int list (overflow-checked), float otherwise; sum([]) is 0.
stats.geometricMean(xs) / .harmonicMean(xs)	Geometric / harmonic mean -> float. Every element must be > 0.
stats.weightedMean(xs, ws)	Weighted mean sum(x*w)/sum(w) -> float. Equal length; weights not summing to 0.
stats.zscore(xs)	Each element standardized (x-mean)/stddev -> list of float. Constant list errors.
stats.mode(xs) (elem) / .modes(xs) (list)	Most frequent element / every tied modal value; both preserve kind.
stats.correlation(xs, ys) / .covariance(xs, ys)	Pearson correlation ([-1,1]) / population covariance of two equal-length lists -> float.
stats.sampleCovariance(xs, ys)	Sample (n-1) covariance -> float. Needs >= 2 elements.
stats.describe(xs)	One-call stats.Summary{count, min, q1, median, mean, q3, max, stddev}.
stats.normalPdf/normalCdf/normalQuantile(p,mean,sd)	Normal density / CDF / inverse-CDF; sd>0, quantile p in (0,1).
stats.normalSample(mean, sd)	One Normal(mean, sd) draw (uses math.randSeed).
stats.tCdf(x, df) / .tQuantile(p, df) / .tPdf(x, df)	Student's t CDF / inverse / density; df>0.
stats.chiSquareCdf(x, df) / .chiSquareQuantile(p, df)	Chi-square CDF / inverse; x>=0, df>0.
stats.fCdf(x, df1, df2) / .fQuantile(p, df1, df2)	F CDF / inverse; x>=0, dfs >0.
stats.binomialPmf(k, n, p) / .binomialCdf(k, n, p)	Binomial mass / CDF; k,n ints, 0<=p<=1.
stats.poissonPmf(k, lambda) / .poissonCdf(k, lambda)	Poisson mass / CDF; k int, lambda>0.
stats.linearRegression(xs, ys)	Simple OLS -> stats.Regression{slope, intercept, r, r2, stdErr, pValue, n}.
stats.multipleRegression(X, ys)	Multiple OLS coefficients [intercept, b1, ...] (list of float).
stats.confidenceInterval(data, level)	t-based CI for the mean -> stats.Interval{lower, upper}; level in (0,1).
stats.proportionCi(successes, n, level, method)	Binomial-proportion CI; method "wald"/"wilson"/"clopper-pearson" -> stats.Interval.
stats.tTest(data, mu) / .tTest2(a, b)	One-sample / two-sample (Welch) t-test -> stats.Test{statistic, df1, df2, pValue}.
stats.chiSquareTest(observed, expected)	Chi-square goodness of fit -> stats.Test; expected counts >0.
stats.fTest(a, b) / .anova(groups)	Two-sided F-test of variances / one-way ANOVA -> stats.Test.
stats.histogram(data, binEdges)	Bin counts (Excel FREQUENCY); k+1 ascending edges -> k counts (list of int).
strings.chars(s)	Split s into a list of string, one entry per Unicode code point.

term.makeRaw(stream) / .restore(state)	Enter / leave raw mode (unbuffered, no-echo) on a terminal ("stdin"); makeRaw returns a single-use term.State. Non-terminal errors.
term.size(stream)	The terminal's term.Size{rows, cols} (query "stdout").
term.readByte()	Next raw byte from stdin (0-255), or -1 at end of input. Bytes, not decoded keys. Refused in the REPL.
testing.assertContains(hay, needle)	Throw Error{kind:"assertion"} unless hay contains needle: substring / list element / map key.
testing.assertEqual(actual, expected)	Throw unless deeply equal (lists / maps / structs compare by value).
testing.assertFalse(cond)	Throw unless cond (a bool) is false.
testing.assertNotEqual(actual, expected)	Throw unless not deeply equal.
testing.assertThrows(name, kind)	Throw unless the named zero-arg method throws an Error of that kind.
testing.assertTrue(cond)	Throw unless cond (a bool) is true.
testing.report(results, format)	Render results to "text", "tap", or "junit" (returns string).
testing.reset()	Clear the process-wide result accumulator.
testing.results()	Snapshot of the accumulator as list of testing.Result.
testing.run(name)	Invoke a zero-arg user method by name; catch every failure mode into a testing.Result.
testing.runWith(name, args)	Like run, binding the args list to the method's parameters (arity + type checked).
strings.contains(s, sub)	True if s contains the substring sub.
strings.endsWith(s, suffix)	True if s ends with suffix.
strings.fold(s)	Remove common Latin diacritics for a sort / search key (Österreich -> Osterreich); case preserved. Not full Unicode collation.
strings.indexOf(s, sub)	Rune index of first sub in s, or -1 if absent.
strings.join(parts, sep)	Concatenate list of string parts separated by sep. Inverse of strings.split.
strings.lower(s)	Lowercase s (Unicode-aware).
strings.repeat(s, n)	n non-negative copies of s concatenated.
strings.replace(s, old, new)	Replace all occurrences of old in s with new.
strings.split(s, sep)	Split s on non-empty sep; returns list of string.
strings.startsWith(s, prefix)	True if s starts with prefix.
strings.substring(s, start)	Rune-indexed slice of s from start to end.
strings.substring(s, start, end)	Rune-indexed slice; exclusive end.
strings.trim(s)	Strip leading and trailing Unicode whitespace.
strings.trimLeft(s)	Strip leading whitespace.
strings.trimRight(s)	Strip trailing whitespace.
strings.upper(s)	Uppercase s (Unicode-aware).

task.cancel($t)	Request cooperative cancellation of $t; the body stops at its next loop checkpoint. Non-blocking.
task.cancelled()	Non-raising poll: true if the current spawn body has been cancelled (false on main).
task.discard($t)	Mark a task of T fire-and-forget; suppresses exit-time loud-fail. Returns null.
task.poll($t)	True if $t has finished (non-blocking).
task.wait($t)	Block until $t finishes; return its value or re-raise its error.
task.waitAll($ts)	Block for all tasks in $ts; results in list order; re-raises the first error if any.
task.waitAny($ts)	Block until any task in $ts is done; return its index.
task.waitAnyTimeout($ts, ms)	Like waitAny, but throws a catchable "timed out" error if none finishes within ms.
task.waitTimeout($t, ms)	Like wait, but throws a catchable "timed out" error if $t doesn't finish within ms.
time.add($t, $d)	time.Time shifted by duration $d.
time.after($a, $b)	True if $a is strictly later than $b.
time.before($a, $b)	True if $a is strictly earlier than $b.
time.day($t)	Day of month, 1-31.
time.equal($a, $b)	True if $a and $b are the same UTC instant.
time.format($t, layout)	Strftime-style format. Codes: %Y %m %d %H %M %S %z %a %A %b %B %j %u %%.
time.fromHours(n)	time.Duration of n hours.
time.fromIso(s)	Parse RFC 3339; accepts Z or +HH:MM; optional fractional seconds.
time.fromMilliseconds(n)	time.Duration of n milliseconds.
time.fromMinutes(n)	time.Duration of n minutes.
time.fromSeconds(n)	time.Duration of n seconds.
time.fromUnix(seconds)	time.Time at the given Unix second.
time.fromUnixMillis(ms)	time.Time at the given Unix millisecond.
time.fromUnixNanos(ns)	time.Time at the given Unix nanosecond.
time.hour($t)	Hour 0-23.
time.hours($d)	Span as whole hours (int).
time.inZone($t, $z)	Re-render $t in $z's wall-clock; UTC instant is preserved.
time.iso($t)	RFC 3339 string: Z for UTC, +HH:MM otherwise; fractional seconds when non-zero.
time.local()	Host's current time.Zone (name + offset).
time.milliseconds($d)	Span as whole milliseconds (int).
time.minute($t)	Minute 0-59.
time.minutes($d)	Span as whole minutes (int).
time.month($t)	Calendar month, January = 1.
time.nanosecond($t)	Fractional second, 0-999_999_999.
time.now()	Current instant in the host's local zone (time.Time).

time.parse(s, layout)	Strict strftime-style parse. Same code set as format (%j / %u are format-only).
time.second($t)	Second 0-59.
time.seconds($d)	Span as whole seconds (int).
time.sleep($d)	Block the running task for $d. Negative / zero returns immediately. Returns null.
time.sub($a, $b)	Signed time.Duration between two time.Time values.
time.unix($t)	Unix-second instant of $t (int).
time.unixMillis($t)	Unix-millisecond instant of $t (int).
time.unixNanos($t)	Unix-nanosecond instant of $t (int).
time.utc()	Current instant in UTC (time.Time).
time.weekday($t)	ISO 8601 weekday: Monday = 1 ... Sunday = 7.
time.year($t)	Calendar year (int).
time.zone(offset, name)	Build a time.Zone from an integer offset (seconds east of UTC) and a display name.
toml.decode(s)	Parse TOML text into an opaque toml.Value handle (walk it with the accessors below).
toml.encode(v) / .encodePretty(v)	TOML string for a toml.Value (or native map / list / scalar); encodePretty blank-lines sections. Null value / non-table root errors.
toml.typeOf(v[, ptr])	Node type at an optional JSON Pointer: null bool int float string list map datetime.
toml.get(v[, ptr])	Sub-node at a JSON Pointer, as a toml.Value (walk stays opaque; no pointer = the node itself).
toml.has(v, ptr)	Whether the JSON Pointer resolves to an existing node.
toml.keys(v[, ptr]) / .length(v[, ptr])	list of string table keys in document order / element count of a list or table.
toml.asInt(v[, ptr]) / asFloat / asString / asBool	Extract the addressed leaf as a typed value (strict; asFloat promotes an int).
toml.asDatetime(v[, ptr]) / .isDatetime(v[, ptr])	A date-time node as a time.Time (needs use time;) / whether the node is a date-time.
toml.map() / .list()	A fresh empty table / array toml.Value - the explicit start of a document (writes never auto-vivify).
toml.set(v, ptr, val) / .insert / .append / .remove / .move	Non-mutating edits by JSON Pointer; each returns a new toml.Value (strict / no missing intermediates).
uuid.generate(v)	New UUID string; v is "v4" (random) or "v7" (time-ordered).
uuid.isValid(s)	Whether s is a well-formed UUID string.
uuid.parse(s)	The 16 bytes of a UUID string; errors on malformed input.
uuid.version(s)	Version digit (4, 7, ...; 0 for NIL); errors on malformed input.
xml.decode(s)	Parse XML into an opaque xml.Value (root element); errors with line/column on malformed input.

xml.encode(v) / .encodePretty(v)	Serialize an xml.Value (compact / 2-space indented).
xml.typeOf(node)	"element" or "text".
xml.tag(node)	The element's tag name.
xml.text(node)	The element's concatenated direct character data (or a text node's string).
xml.attr(node, name)	An attribute value; errors if absent. xml.hasAttr(node, name) tests presence.
xml.attrs(node)	The attribute names (list of string, document order).
xml.children(node)	The element children (list of xml.Value; text excluded).
xml.get(node, path)	First element matching an XPath-style path (name/name[k]/*, /-separated); errors if none.
xml.findAll(node, path)	Every element matching the path (list of xml.Value). xml.has(node, path) -> bool.
xml.element(name)	A fresh empty element xml.Value.
xml.setAttr(node, name, value)	The element with the attribute added/updated (fresh handle).
xml.setText(node, s)	The element with its children replaced by one text node (fresh handle).
xml.append(parent, child)	The parent element with child appended (fresh handle).
yaml.decode(s) / .decodeAll(s)	Parse one YAML document (multi-doc errors) / every document of a --- stream (list of yaml.Value), into opaque handles.
yaml.encode(v) / .encodePretty(v)	YAML string for a yaml.Value (or native map / list / scalar): flow (compact) vs block (readable) style.
yaml.typeOf(v[, ptr])	Node type at an optional JSON Pointer: null bool int float string bytes list map datetime.
yaml.get(v[, ptr])	Sub-node at a JSON Pointer, as a yaml.Value (walk stays opaque; no pointer = the node itself).
yaml.has(v, ptr)	Whether the JSON Pointer resolves to an existing node.
yaml.keys(v[, ptr]) / .length(v[, ptr])	list of string map keys in document order / element count of a list or map.
yaml.asInt(v[, ptr]) / asFloat / asString / asBool / isNull	Extract the addressed leaf as a typed value (strict; asFloat promotes an int) / test for null.
yaml.asDatetime(v[, ptr]) / .isDatetime(v[, ptr])	A timestamp node as a time.Time (needs use time;) / whether the node is a timestamp.
yaml.map() / .list()	A fresh empty mapping / sequence yaml.Value - the explicit start of a document (writes never auto-vivify).
yaml.set(v, ptr, val) / .insert / .append / .remove / .move	Non-mutating edits by JSON Pointer; each returns a new yaml.Value (strict / no missing intermediates).

Constants

Name	Type	Value
math.E	float	Euler's number, 2.718281828459045.

math.PI	float	pi, 3.141592653589793.
math.TAU	float	? = 2pi, the full-turn constant, 6.283185307179586.
meta.BUILD	string	Which Go toolchain compiled the interpreter: "go" / "tinygo".
meta.VERSION	string	The interpreter's build version (e.g. "0.14.0").
meta.SYSMODDIR	string	Resolved system module directory (--sysmoddir > JENNIFER_SYSMODDIR > compile default).
meta.CAPABILITIES	list of string	Host capabilities this build has, sorted: ["exec", "net", "sql"] (standard) / [] (tiny).
meta.hasCapability(name)	bool	Whether the build includes capability name ("net" / "exec"); the # pragma-jennifer-capability: set.
meta.call(name, args...)	value	Invoke a top-level method by runtime name (arity + types checked); errors / exit propagate.
meta.defined(name)	bool	Whether a top-level method name exists.
meta.callMain(name, args...) / .definedMain(name)	value / bool	Like call / defined but against the entry program's methods (a module reaching its host's handlers).
os.ARCH	string	CPU architecture: "amd64", "arm64", "wasm", ...
os.ARGS	list of string	Argv. Index 0 is the script path, the rest are user args.
os.DIRSEP	string	Path-component separator: "/" Unix, "\\" Windows.
os.EOL	string	Platform line ending. "\n" Unix-likes, "\r\n" Windows.
os.NCPU	int	Logical CPUs usable by the process (runtime.NumCPU). 1 on jennifer-tiny (single-thread scheduler).
os.PATHSEP	string	PATH-list separator: ":" Unix, ";" Windows.
os.PLATFORM	string	OS tag: "linux", "darwin", "windows", ...
time.PROGRAM_START	time.Time	Captured the moment the time library installed; "since program launched" anchor.
time.UTC	time.Zone	Canonical UTC: Zone{offset: 0, name: "UTC"}.
uuid.NIL	string	The all-zero UUID 00000000-0000-0000-0000-000000000000.

Type-conversion calls

`int` , `float` , `string` , `bool` are also type keywords (used in `def x as int`). The parser allows them in expression position **only** when immediately followed by `(` , so `def x as int init convert.toInt("42");` works but `def x as int init int;` errors. See convert.md for the parser detail.

See also

- index.md - library catalog with code samples and the organizing principles.
- Per-library reference pages: io.md, convert.md, math.md, strings.md, lists.md, maps.md, os.md, meta.md, time.md, hash.md, crc.md, encoding.md, task.md, fs.md, net.md, regex.md, testing.md.
- ../user-guide/imports.md - how to import a library in a Jennifer source file.

archive - tar / zip containers

Enable with `use archive;` . Bundle files into a tar or zip container and read them back, entirely over `bytes` - no `fs` dependency, value-semantic. Shares the `pack` / `unpack` verbs with `compress` (byte streams there, file bundles here); the container format is a string argument. Backed by Go's archive/tar + archive/zip; works on both binaries.

Surface

Call	Returns	Notes
archive.pack(entries, format)	bytes	Bundle a list of archive.Entry.
archive.unpack(b, format)	list of archive.Entry	Read a bundle back.

`format` is `"tar"` , `"zip"` , or the gzip combo `"tar.gz"` (alias `"tgz"`). An unknown format, or corrupt input to `unpack` , is a positioned runtime error (catchable with `try` / `catch`).

`unpack` bounds untrusted input: the total decompressed payload of one call (summed across every entry) is capped at 256 MiB and the member count at 65536; past either cap it raises a normal catchable error instead of expanding a small "zip bomb" into gigabytes of memory. It also rejects any member whose name is an absolute path or escapes with `..` (a "zip-slip" name), so a naive extraction loop (`fs.writeBytes($dir + "/" + $e.name, $e.data)`) can't be tricked into writing outside the target directory.

archive.Entry

A struct `{ name as string, data as bytes, mode as int, mtime as int }` :

- **name** - the path within the archive (subdirectories with `/`).
- **data** - the file contents.
- **mode** - unix permission bits (e.g. `0o644`); `0` means the default `0o644` .
- **mtime** - modification time, unix seconds.

Only regular files map to an `Entry` ; directory members are skipped on `unpack` .

Example

```
use io;
use archive;
use convert;

def readme as archive.Entry init archive.Entry{
    name: "README", data: convert.bytesFromString("hello", "utf-8"), mode: 0o644,
mtime: 0
};
def blob as bytes init archive.pack([$readme], "tar.gz");   # one call: tar, then
gzip
def back as list of archive.Entry init archive.unpack($blob, "tar.gz");
io.printf("%s = %s\n", $back[0].name, convert.stringFromBytes($back[0].data, "utf-
8"));
```

`"tar.gz"` layers `compress` 's gzip over a tar internally, so the everyday `.tar.gz` case is a single call rather than a `tar` -then-gzip pair.

See also

- compress.md - `pack` / `unpack` for byte streams (`"gzip"` / `"zlib"` / `"deflate"`).
- fs.md - write the packed `bytes` to disk, or read an archive off disk to unpack.

asn1 - ASN.1 BER/DER

The `asn1` library encodes and decodes ASN.1 (https://en.wikipedia.org/wiki/ASN.1) (Abstract Syntax Notation One) in its BER / DER transfer encodings - the byte-level foundation for LDAP, SNMP, and PKI formats. It is designed like the other opaque-value libraries (`json` / `toml` / `xml` / `yaml`): `decode` yields an opaque `asn1.Value` you walk with `(node, pointer)` accessors, and there is no coercion into a typed struct. Values are built with typed constructors and serialised with `encode` .

It is hand-rolled (Go's `encoding/asn1` is DER-only and reflect-bound), so it is dependency-free and both binaries build it. `decode` accepts **BER** (indefinite lengths, the alternative encodings LDAP / SNMP permit); `encode` always emits canonical **DER** (definite lengths).

```
use asn1;
use convert;

# Build  SEQUENCE { INTEGER 42, OCTET STRING "hi", OID 1.2.840.113549 }
def msg as asn1.Value init asn1.sequence([
    asn1.integer(42),
    asn1.octetString(convert.bytesFromString("hi", "utf-8")),
    asn1.oid("1.2.840.113549")
]);
def wire as bytes init asn1.encode($msg);          # DER bytes

def back as asn1.Value init asn1.decode($wire);
asn1.asInt($back, "/0");                            # 42
asn1.asString($back, "/1");                         # "hi"
asn1.asOid($back, "/2");                            # "1.2.840.113549"
```

The tree and the pointer

A decoded `asn1.Value` is a tree of elements. Each element has a **tag** (a *class* - `universal` , `application` , `context` , or `private` - and a numeric *tagNumber*), is either **primitive** (carrying content octets) or **constructed** (carrying ordered child elements), and a `SEQUENCE` / `SET` is the common constructed shape.

Accessors take the node and an optional **pointer** whose tokens are **child indices** : `""` (or omitted) is the node itself, `"/0"` its first child, `"/2/1"` the second child of its third child. (This is the child-index analogue of the JSON Pointer the `json` family uses over map keys.)

Reading

Call	Returns	Notes
asn1.decode(bytes)	asn1.Value	Parse one BER element; trailing bytes are an error.
asn1.typeOf(v[, ptr])	string	A friendly type: the universal type name (integer, octetString, sequence, ...) or the class (context / application / private) for a non-universal tag.
asn1.tagClass(v[, ptr])	string	universal / application / context / private.
asn1.tagNumber(v[, ptr])	int	The numeric tag.
asn1.isConstructed(v[, ptr])	bool	Whether the element holds children (vs. content).
asn1.length(v[, ptr])	int	Child count of a constructed element (a primitive is an error).
asn1.get(v[, ptr])	asn1.Value	The addressed sub-element, as an asn1.Value (the walk stays opaque).
asn1.has(v, ptr)	bool	Whether the pointer resolves.
asn1.asInt(v[, ptr])	int	An INTEGER / ENUMERATED as int (must fit int64).

asn1.asBool(v[, ptr])	bool	A BOOLEAN.
asn1.asString(v[, ptr])	string	A string type (UTF8String / PrintableString / IA5String) or an OCTET STRING, as UTF-8 (invalid UTF-8 is an error - use asBytes).
asn1.asBytes(v[, ptr])	bytes	The raw content octets of any primitive element.
asn1.asOid(v[, ptr])	string	An OBJECT IDENTIFIER as a dotted string.
asn1.isNull(v[, ptr])	bool	Whether the element is NULL.

Building and encoding

Call	Builds
asn1.integer(n) / asn1.enumerated(n)	INTEGER / ENUMERATED from an int.
asn1.boolean(b)	BOOLEAN.
asn1.null()	NULL.
asn1.octetString(bytes)	OCTET STRING.
asn1.utf8String(s) / asn1.printableString(s) / asn1.ia5String(s)	the named string type.
asn1.oid(dotted)	OBJECT IDENTIFIER from a dotted string ("1.3.6.1").
asn1.sequence(items) / asn1.set(items)	SEQUENCE / SET from a list of asn1.Value.
asn1.tagged(class, tagNumber, value)	EXPLICIT tag: a constructed [class tagNumber] wrapping value as its single child (its own tag intact).
asn1.retag(class, tagNumber, value)	IMPLICIT tag: value with its outer tag replaced by [class tagNumber], content and constructed-ness kept.
asn1.encode(v)	The DER bytes of an asn1.Value.

`class` is `"universal"` , `"application"` , `"context"` , or `"private"` . EXPLICIT vs. IMPLICIT tagging is the ASN.1 distinction the two builders capture: `tagged` nests the original element inside the new tag; `retag` overwrites the tag in place (as SNMP PDUs and many LDAP fields are defined).

Strictness and limits

Like the rest of the standard library, malformed input is a catchable error, not a panic or a wrong answer: a truncated element, a length that runs past the buffer, an indefinite length on a primitive, an unterminated indefinite element, trailing bytes after the top-level value, and a leaf extractor called on the wrong element type all raise a positioned error. Two resource guards convert a crafted input into a catchable error rather than a crash: nesting is bounded (a deeply nested value cannot overflow the Go stack), and the number of elements one `decode` will materialise is capped (a flat "decode bomb" of millions of tiny elements is rejected).

Scope

BER decode / DER encode over the tree above is the whole of v1 - the enabler for the `ldap` (../modules/) and SNMP clients. It does not interpret schema (no ASN.1-module compiler), and `asn1.asInt` is `int64` -bounded (read a larger `INTEGER` , such as an SNMP `Counter64` near `2^64` , with `asBytes`).

binary - bulk operations on bytes

Enable with `use binary;` . Namespaced under `binary.` ; every function is called as `binary.NAME(...)` . It is to `bytes` what `strings` is to `string` and `lists` is to `list` : concatenate, slice, search, and split byte buffers.

The library is named `binary` because `bytes` is a reserved type keyword (it appears after `as` , as in `def b as bytes;`) and so cannot be a library namespace.

All indices and lengths are byte offsets , 0-relative. `binary.slice(b, 0, 2)` returns the first two bytes; `len(b)` is the byte count. (Contrast `strings` , which counts Unicode code points.)

Every operation is non-mutating and value-semantic. The inputs are never aliased or written; each result is a freshly-allocated `bytes` (or `list of bytes` / `int` / `bool`). Mutating a result never touches the source.

```
use io;
use convert;
use binary;

def a as bytes init convert.bytesFromString("Hello, ", "utf-8");
def b as bytes init convert.bytesFromString("World", "utf-8");
def c as bytes init binary.concat($a, $b);

io.printf("%s\n", convert.stringFromBytes($c, "utf-8"));         # "Hello, World"
io.printf("%d\n", binary.indexOf($c, $b));                       # 7
io.printf("%s\n", convert.stringFromBytes(binary.slice($c, 7, 12), "utf-8")); #
"World"
io.printf("%t\n", binary.startsWith($c, $a));                    # true
```

Functions

Call	Returns	Notes
binary.concat(a, b)	bytes	Join two byte sequences. See Building buffers for the loop caveat.
binary.join(parts)	bytes	Concatenate every bytes in a list of bytes into one, in O(n). The byte-data strings.join.
binary.join(parts, sep)	bytes	As above, with sep (a bytes) placed between adjacent pieces.
binary.slice(b, start)	bytes	From start to the end of b.
binary.slice(b, start, end)	bytes	The half-open range [start, end); exclusive end.
binary.indexOf(haystack, needle)	int	Byte index of the first occurrence of needle; -1 if absent. An empty needle returns 0. Same shape as strings.indexOf.
binary.contains(haystack, needle)	bool	Whether needle occurs in haystack (the boolean sibling of indexOf, like strings.contains).
binary.split(b, sep)	list of bytes	Split on every occurrence of a non-empty sep; preserves empty segments.
binary.startsWith(b, prefix)	bool	True iff b begins with prefix.
binary.endsWith(b, suffix)	bool	True iff b ends with suffix.

`binary.split` is the natural way to cut a container at a delimiter in one pass - for example splitting a MIME multipart body on its boundary, or a length-unprefixed record stream on its separator:

```
def parts as list of bytes init binary.split($body, $boundary);
```

Errors

- `binary.slice(b, -1, 3)` - negative start.
- `binary.slice(b, 0, 99)` - end past the byte length.
- `binary.slice(b, 4, 2)` - end before start.
- `binary.split(b, sep)` with an empty `sep` .
- Non-bytes arguments where `bytes` are required, or non-int where int is required (`binary.slice(b, "3")`).
- Arity errors (too many or too few arguments).

All are positioned, catchable runtime errors.

Why this library exists: throughput

The reason to reach for `binary` over a hand-written loop is speed. A `.j` program that concatenates, slices, searches, or splits a byte buffer *one byte at a time* pays the tree-walking interpreter's per-operation cost on every byte (an index read, a comparison, an append, a loop step). For kilobyte-sized data that is invisible; for a megabyte-scale buffer it dominates. Each `binary` function pushes the inner loop into Go and runs it once at native speed (`binary.indexOf` / `binary.split` are Go's `bytes.Index` / `bytes.Split`), so scanning a large buffer for a delimiter is one call, not a per-byte scan.

```
# Slow: a per-byte scan pays the interpreter's cost on every byte.
def idx as int init -1;
def i as int init 0;
while ($i + len($needle) <= len($buf) and $idx < 0) {
    # ... byte-by-byte comparison ...
    $i = $i + 1;
}

# Fast: one Go-speed scan.
def idx as int init binary.indexOf($buf, $needle);
```

Building buffers

`binary.concat` copies both inputs into a fresh result, so it is `O(len(a) + len(b))` per call. Using it to accumulate a growing buffer in a loop (`$buf = binary.concat($buf, $chunk)` each iteration) re-copies the whole buffer every time, which is `O(n^2)` overall. To assemble a buffer from a stream, read it in one call instead: `net.readAll` reads a whole response body to EOF as one `bytes` , and `net.readN` reads an exact length-prefixed frame - both grow a single Go slice rather than concatenating in a `.j` loop. Reserve `binary.concat` for joining a small, fixed number of pieces.

When you already hold the pieces - you built them in a loop rather than read them off a stream - collect them in a `list of bytes` and make one `binary.join` call. That is `O(n)` (a single allocation sized to the total), where the concat-in-a-loop it replaces is `O(n^2)` . It is the exact byte-data analogue of building a string with list-append plus `strings.join` :

```
# Slow: each concat re-copies the whole accumulator - O(n^2).
def out as bytes;
for (def chunk in $chunks) {
    $out = binary.concat($out, $chunk);
}

# Fast: append the pieces, join once - O(n).
def parts as list of bytes init [];
for (def chunk in $chunks) {
    $parts[] = $chunk;
}
def out as bytes init binary.join($parts);
```

TinyGo

`binary` is pure byte manipulation with no host dependencies, so it runs in full on both binaries - the default `jennifer` and the constrained `jennifer-tiny` .

See also: `strings` , `net` (`readAll` / `readN`), index.md.

channel - CSP channels between goroutines

The `channel` library provides CSP-style channels: typed conduits that carry values between a `spawn` ed body and the rest of the program (or between several spawns). Channels are the coordination primitive that pairs with `spawn` / `task` - where `task` observes a single result, a channel streams many values and lets producers and consumers rendez-vous.

```
use channel;
use task;

def ch as channel of int init channel.make(0);   # unbuffered
def producer as task of int init spawn {
    def i as int init 0;
    while ($i < 5) {
        channel.send($ch, $i * 10);
        $i = $i + 1;
    }
    channel.close($ch);
    return 0;
};
# consumer: drain until the producer closes the channel
def sum as int init 0;
try {
    while (true) {
        $sum = $sum + channel.recv($ch);
    }
} catch (e) {
}
task.wait($producer);
```

Why channels (not locks)

A channel is a **shared handle** : copies - including the deep copy a `spawn` makes of its captured scope - refer to the one underlying conduit, which is what lets a spawn body and its parent talk. But the **values sent through it are copied** at the send site, so channels carry copies and Jennifer's no-shared-mutable-state guarantee still holds. That is precisely why channels, not mutexes or atomics, are the right primitive: a shared lock would expose shared mutable state and stays rejected. A channel shares the *conduit* , never the *data* .

`channel` is a **contextual keyword** : `channel of T` is a type, but `channel` stays an ordinary identifier everywhere else (a field or parameter named `channel` keeps working).

Surface

Call	Returns	Notes
channel.make(capacity)	channel of T	A new channel. capacity 0 is unbuffered (send blocks for a receiver); n buffers up to n values (a capacity beyond a generous ceiling is a catchable error, not an OOM crash). T comes from the binding.
channel.send(ch, value)	null	Checks value against the channel's T, deep-copies it in, then sends (blocking per capacity). Wrong type / send on a closed channel are catchable errors.
channel.recv(ch)	T	Blocks and returns the next value. On a closed and drained channel it throws a catchable "receive on a closed channel".
channel.close(ch)	null	Closes the channel so a draining receiver sees the end. Double-close is a catchable error.

channel.select(chs)	T	Fan-in: the next value from any of the channels; drops closed ones and throws when all are closed and drained.
channel.len(ch)	int	Number of values currently buffered.
channel.capacity(ch)	int	The channel's buffer capacity (0 = unbuffered).

Value semantics

`channel.send` copies the value in, so a later mutation by the sender never reaches the receiver:

```
def ch as channel of list of int init channel.make(1);
def xs as list of int init [1, 2, 3];
channel.send($ch, $xs);
$xs[0] = 999;                                  # mutate after send
def got as list of int init channel.recv($ch); # still [1, 2, 3]
```

The element type is enforced at **both** ends. `channel.send` validates the value against the channel's declared `T` at the send site (`channel.send($intCh, "x")` is a catchable "value must be int, got string"), and the receiving binding checks it again (`def n as int init channel.recv($ch);` rejects a non-int) - so a mismatch is caught where it happens, and a `channel of int` still cannot be assigned to a `channel of string` . A channel not yet bound to a `channel of T` (a bare `channel.make` result) carries no recorded type, so its first send is unchecked - like a fresh generic list literal - until a typed binding stamps `T` .

Closing and draining

`channel.close` signals "no more values". A receiver draining with `channel.recv` sees a catchable "receive on a closed channel" once the buffer is empty, so the idiomatic consumer loop is a `try` / `catch` :

```
try {
    while (true) {
        process(channel.recv($ch));
    }
} catch (e) {
    # channel drained and closed - done
}
```

Only the producer should close, and only once. After close, buffered values still receive until drained; further sends are catchable errors.

select (fan-in)

`channel.select(chs)` blocks until any channel in the list has a value and returns it - a merge of several producers into one consumer. A closed channel drops out of the wait set; when every channel is closed and drained, `select` throws a catchable "all channels are closed".

```
try {
    while (true) {
        handle(channel.select([$a, $b, $c]));
    }
} catch (e) {
    # every input closed
}
```

`select` returns the received **value** , not an index of which channel fired (a channel receive is destructive and Jennifer has no multiple-return, so an index-plus-value form would need a different shape). It is the merge / fan-in primitive; per-channel dispatch is a possible later addition.

Blocking and cancellation

Channel operations block like Go's: an unbuffered `send` waits for a receiver, a `recv` waits for a value. A channel

with no counterpart (an unbuffered `send` nothing will receive, a `recv` on a channel nothing will send to or close) **hangs** the program - the same sharp edge as a non-terminating `spawn` . Note this does *not* trigger Go's "all goroutines asleep" deadlock abort: the interpreter keeps a signal-handler goroutine alive (for `SIGUSR1` diagnostics), so the runtime never sees every goroutine blocked, and the program waits until it is killed rather than aborting. A spawn body blocked in `channel.send` / `recv` is also **not** at a loop checkpoint, so `task.cancel` cannot interrupt it. Unblock a waiting body by `channel.close` -ing the channel it waits on (a drained `recv` then raises), or size a buffered channel so a send does not block. Cancellable channel operations are a possible later addition.

Patterns

- **Producer / consumer** : one spawn sends, the main goroutine drains until close (the opening example).
- **Fan-out (worker pool)** : several spawned workers `recv` from one job channel; close it to signal shutdown, each worker's `recv` then ends.
- **Fan-in** : several spawned producers `send` into one channel (or use `channel.select` over several channels) that one consumer drains.
- **Pipeline** : stage N reads its input channel and sends into stage N+1's; each stage closes its output when its input drains.

See also

- `task` - observe a single spawned result; cancellation and timeouts.
- Concurrency - the `spawn` model and value-semantics capture.

compress - byte-stream compression

Enable with `use compress;` . gzip, zlib, and raw DEFLATE - `bytes` in, `bytes` out - plus a streaming compressor for large data. The algorithm is a string argument, the same shape as `hash .compute(b, "sha-256")` and `crc .compute(b, "crc32")` ; the `pack` / `unpack` verbs pair with `archive` 's (byte streams here, file bundles there). Distinct from `encoding` , which is *reversible representation* (hex / base64, which don't reduce information); this is entropy-based *size reduction* . Backed by Go's `compress/*` ; works on both binaries.

Surface

Call	Returns	Notes
compress.pack(b, algo [, level])	bytes	Compress; algo is "gzip" / "zlib" / "deflate".
compress.unpack(b, algo)	bytes	Decompress; same algo.
compress.stream(algo [, level])	compress.Stream	Start a streaming compressor.
compress.update(stream, b)	null	Feed one chunk of input.
compress.finalize(stream)	bytes	Close and return the full compressed output.
compress.discard(stream)	null	Drop a stream without flushing/returning output, releasing its state. Subsequent calls error.

`level` is optional: `"fast"` , `"default"` (when omitted), or `"best"` . Decompressing malformed input, or an unknown `algo` , is a positioned runtime error (catchable with `try` / `catch`).

The three algorithms:

- **"gzip"** (RFC 1952) - magic + CRC-32 + size. `gzip(1)` -compatible; the reliable HTTP `Content-Encoding: gzip` . Use for standalone files.
- **"zlib"** (RFC 1950) - a compact DEFLATE + Adler-32 wrapper. This is what HTTP `Content-Encoding: deflate` *officially* means (and it's the wrapper inside PNG, zip entries, etc.) - but that coding is inconsistently implemented in the wild, so prefer `"gzip"` for HTTP.
- **"deflate"** (RFC 1951) - raw DEFLATE, no framing. Smallest, when you supply your own framing. **Not** the HTTP `deflate` content-coding despite the name - that one is `"zlib"` .

One-shot

```
use io;
use compress;
use convert;

def raw as bytes init convert.bytesFromString("hello hello hello world", "utf-8");
def packed as bytes init compress.pack($raw, "gzip", "best");
def back as bytes init compress.unpack($packed, "gzip");
io.printf("%t\n", convert.stringFromBytes($back, "utf-8") == "hello hello hello
world");
```

Streaming

Feed input in chunks (so you never hold it all at once); the full compressed result comes back at `finalize` :

```
def s as compress.Stream init compress.stream("gzip");
compress.update($s, chunkOne);
compress.update($s, chunkTwo);
def packed as bytes init compress.finalize($s);   # equals compress.pack of the
concatenation
```

`compress.Stream` is a handle (like `hash.Stream`): copies share the underlying state, and `finalize`

consumes it - a second `finalize` or `update` on the same handle errors.

See also

- encoding.md - hex / base64 (representation, not compression).
- hash.md - the same algo-string + streaming-handle shape.
- archive.md - `pack` / `unpack` for file bundles (`"tar"` / `"zip"`).

convert - explicit type conversion

Enable with `use convert;` . Provides one-argument conversion functions plus `typeOf` for runtime kind introspection. Every function either returns the converted value or produces a positioned runtime error.

```
use io;
use convert;

def n as int init convert.toInt("42");        # parse string -> 42
def f as float init convert.toFloat(5);       # int -> 5.0
def s as string init convert.toString(3.14);  # any -> "3.14"
def b as bool init convert.toBool(0);         # 0 -> false

io.printf("%s\n", convert.typeOf(5 / 2));      # "float" (after Python 3 / change)
io.printf("%s\n", convert.typeOf(5 // 2));     # "int"
```

Behavior summary

Call	Source kinds	Behavior
convert.toInt(v)	int / float / string / bool	identity / truncate / parse / true=1, false=0
convert.toFloat(v)	int / float / string / bool	convert / identity / parse / true=1.0, false=0.0
convert.toString(v)	any	always succeeds; uses the value's display form
convert.toBool(v)	bool / int / float / string	identity / canonical only (0/1, 0.0/1.0, "true"/"false")
convert.typeOf(v)	any	returns the kind as a string: "int", "float", ..., "list", "map", "object"
convert.objectType(v)	object	specific registered name of an opaque object, e.g. "json.Value"; errors on a non-object
convert.bytesFromString(s, codec)	(string, string)	string to bytes; "utf-8" only (other codecs live in encoding)
convert.stringFromBytes(b, codec)	(bytes, string)	bytes to string; "utf-8" only; invalid UTF-8 is an error
convert.toCodepoint(char)	string	Unicode code point (int) of a one-rune string; errors unless the argument is exactly one code point
convert.fromCodepoint(n)	int	the one-rune string for code point n; errors on a negative / out-of-range / surrogate value

`bytesFromString` / `stringFromBytes` are the **UTF-8** cross-kind pair (`string` to `bytes` and back) - the one codec every program needs. Every other character encoding (ISO-8859- *, Windows-* , EBCDIC) and the binary-to-text codecs (hex, base64, quoted-printable) live in `encoding` : `encoding.encode` / `decode` and `encoding.toText` / `fromText` .

toCodepoint / fromCodepoint: code point, not "character"

These convert between a single **code point** (a Unicode scalar value, an `int`) and the **one-rune string** that holds it - the primitive a Unicode algorithm needs (Punycode digits, a `\x01` control byte, an escape). Two things worth being precise about:

- **The whole range works, not just ASCII or the BMP.** `fromCodepoint` accepts any scalar value `0` .. `0x10FFFF` (hex literals are fine: `fromCodepoint(0x20AC)` is `€` , `fromCodepoint(0x1F602)` is `?`). The result is *one rune* whose UTF-8 encoding is **1 to 4 bytes** - a code point is not a byte. Only a negative value, one above `0x10FFFF` , or a UTF-16 surrogate (`0xD800` .. `0xDFFF`) errors.

- **"One rune" is not "one character a reader sees."** A user-perceived character - a *grapheme cluster* - can be several code points: a base plus combining marks (e + U+0301 = é), an NFD-decomposed accent, an emoji ZWJ sequence (???). Each of those code points round-trips individually, but `toCodepoint` takes **exactly one** code point, so it rejects a multi-rune cluster the same way it rejects `"ab"` . `len` , `strings.chars` , and `strings.substring` are all **rune-indexed** too (see strings.md); Jennifer has no grapheme-cluster API. The tell: precomposed é (U+00E9) is `len` 1 and `toCodepoint` gives 233; decomposed é (e + U+0301) is `len` 2 and `toCodepoint` errors.

Errors

- `convert.toInt("abc")` - parse failure (string doesn't represent a valid integer).
- `convert.toInt(null)` - no conversion defined.
- `convert.toInt(f)` for a NaN, +/-Infinity, or out-of-int64-range float - the value has no representable integer (truncation would be garbage).
- `convert.toBool("maybe")` - strings: only `"true"` and `"false"` accepted.
- `convert.toBool(123)` , `convert.toBool(-1)` - ints: only `0` and `1` accepted.
- `convert.toBool(1.5)` - floats: only `0.0` and `1.0` accepted.
- `convert.stringFromBytes(b, "utf-8")` on bytes that aren't valid UTF-8 - strict at boundaries; no silent replacement characters.
- `convert.bytesFromString(s, "iso-8859-1")` or any non- `"utf-8"` codec name - rejected as unsupported; use `encoding .encode` / `decode` for those.
- Arity errors (too many or too few arguments).

For "any nonzero counts as true" semantics, write the comparison explicitly:

```
def b as bool init $count != 0;     // not convert.toBool($count)
```

This matches the strict-conditions rule everywhere else in Jennifer - if you want to project an arbitrary value into a bool, state the criterion.

Notes on the verb naming

The convert library's four conversion callees are named `toInt` , `toFloat` , `toString` , `toBool` (not `int` / `float` / `string` / `bool`) so they don't collide with the type keywords - the parser keeps those reserved for declarations (`def x as int ...`). The `to` -prefixed verb also reads as English at the call site: `convert.toInt("42")` says "convert to int." `typeOf` is a normal identifier and is not a type keyword.

Writing the bare form (`int("42")` , `string(42)` , ...) is a parse error directing you at the `convert.to*` form.

Objects: typeOf vs objectType

Some libraries hand back an **opaque object** - a value that carries data but exposes it only through that library's own accessors, not through operators or `[index]` / `.field` . The first is `json.Value` (from `json.decode`). For any such value `convert.typeOf` returns the generic `"object"` , and `convert.objectType` returns the specific registered name so you can tell one object family from another:

```
def doc as json.Value init json.decode("{}");
io.printf("%s\n", convert.typeOf($doc));       # object
io.printf("%s\n", convert.objectType($doc));   # json.Value
```

`convert.objectType` errors on a non-object, so guard with `convert.typeOf(v) == "object"` first if the kind is unknown.

See also: encoding.md (all other character and binary-to-text codecs), ../user-guide/index.md, ../technical/interpreter.md, index.md.

crc - cyclic redundancy checks

Enable with `use crc;` . Computes CRC-32 (IEEE polynomial) and CRC-64 (Go's `crc64.ECMA` polynomial) checksums over `bytes` , matching the codec-table shape used by `hash` . Output is the natural-width digest in big-endian byte order.

CRCs are designed to catch transport / storage corruption, not to resist deliberate tampering. For content-addressing or signature work, use `hash` (MD5, SHA-1, SHA-256). The library split keeps the difference visible at the import line.

```
use io;
use convert;
use crc;

def input as bytes init convert.bytesFromString("abc", "utf-8");
def sum as bytes init crc.compute($input, "crc32");
io.printf("crc32 checksum is %d bytes\n", len($sum));
```

Algorithm selection

Algo string	Output width	Polynomial
"crc32"	4 bytes	IEEE 802.3 (Go crc32.IEEE).
"crc64"	8 bytes	Go crc64.ECMA (0xC96C5795D7870F42).

Note: the "CRC-64/XZ" vector you may see elsewhere (`6c40df5f0b497347` for `"123456789"`) uses a different polynomial. We ship Go's stdlib choice (`995dc9bbdf1939fa` for the same input).

Output bytes are big-endian (network byte order). The convention matches the natural display of `Sum()` from Go's CRC types and removes any ambiguity at the call site.

Passing an unknown algorithm is a positioned runtime error: `crc.compute: unknown algorithm "adler32"; known: "crc32", "crc64"` .

One-shot

Call	Returns	Notes
crc.compute(b, algo)	bytes	Full checksum of the entire input. Big-endian.

Streaming

Feed chunks into a stream handle and finalize. Same shape as `hash` :

```
use crc;
def s as crc.Stream init crc.stream("crc32");
crc.update($s, $chunkOne);
crc.update($s, $chunkTwo);
def sum as bytes init crc.finalize($s);
```

Call	Returns	Notes
crc.stream(algo)	crc.Stream	Allocate a fresh handle for the named algorithm.
crc.update($s, $bytes)	null	Feed one chunk. Mutates the handle's state by side effect.
crc.finalize($s)	bytes	Compute the checksum and consume the handle. Subsequent calls error.
crc.discard($s)	null	Drop a stream without computing its checksum, releasing its state. Subsequent calls error.

Errors

- Wrong arity: `crc.compute expects 2 arguments (bytes, algo), got 1` .
- Wrong scalar type: `crc.compute: first argument must be bytes, got string` .
- Unknown algorithm: `crc.compute: unknown algorithm "adler32"; known: "crc32", "crc64"` .
- Reuse of a finalized stream: `crc.update: stream 3 has already been finalized or never existed` .

All errors are positioned at the call site.

See also

- hash.md - cryptographic-style digests (MD5, SHA-1, SHA-256).
- milestones.md - ships hash/crc.

crypto - security primitives

Enable with `use crypto;` . The home for the security-sensitive primitives that need a cryptographically secure random source or a timing-safe operation: crypto-grade randomness, constant-time comparison, and the two standard key-derivation functions.

Digests (MD5 / SHA-*) and the keyed-hash MAC (`hash.hmac`) live in `hash` ; non-cryptographic checksums live in `crc` . `crypto` is the layer above them - the operations where predictability or a timing side channel is a vulnerability.

Everything here is Go standard library only (`crypto/rand` , `crypto/subtle` , `crypto/hkdf` , `crypto/pbkdf2` , `crypto/aes` , `crypto/cipher` , `crypto/ed25519`), so the library adds no dependency and works on **both binaries** (`jennifer` and `jennifer-tiny`). The one exception is the RSA / ECDSA signature surface below (`crypto/rsa` , `crypto/ecdsa` , `crypto/x509`), which is default- `jennifer` only - everything else runs on both.

```
use io;
use crypto;
use encoding;

def token as bytes init crypto.randBytes(32);
io.printf("session token: %s\n", encoding.toText($token, "base64-url"));
```

Crypto-grade randomness

Unlike `math` 's `rand` family (fast, seedable, and therefore **predictable** - a PRNG for simulations and sampling), these draw from the operating system's cryptographically secure source. They are **not** seedable: there is no reproducible sequence, by design.

Call	Returns	Notes
crypto.randBytes(n)	bytes	n secure random bytes (0 <= n <= 64 MiB; 0 yields empty bytes).
crypto.randInt(lo, hi)	int	A uniform int in the inclusive range [lo, hi].

`crypto.randInt` has the same shape as `math.randInt` but is unpredictable and unseedable - the drop-in when the value guards something (a token, a nonce, a shuffle of secret material). It uses rejection sampling, so the distribution is exactly uniform with no modulo bias, across the full `int` range.

```
use crypto;

def dieRoll as int init crypto.randInt(1, 6);        # secure, unbiased
def key as bytes init crypto.randBytes(16);          # a 128-bit key
```

Since this library landed, `uuid` draws its v4 / v7 randomness here, so `uuid.v4()` is unguessable and safe to hand a client as a session id or bearer token.

Constant-time comparison

`crypto.hmacEqual(a, b)` compares two `bytes` values for equality **without an early-out** , so an attacker cannot recover a secret byte by byte from response-timing differences. Use it to check a computed MAC against a supplied one, rather than == (which short-circuits on the first differing byte). Unequal-length inputs return `false` .

```
use crypto;
use hash;
use convert;

def key as bytes init convert.bytesFromString("secret", "utf-8");
def body as bytes init convert.bytesFromString("payload", "utf-8");
def expected as bytes init hash.hmac($key, $body, "sha256");
def supplied as bytes init requestSignature();      # from the caller

if (crypto.hmacEqual($expected, $supplied)) {
    # signature verified
}
```

Key derivation

Two functions turn a secret into keying material. Both output `bytes` .

Call	Returns	Notes
crypto.hkdf(secret, salt, info, length, algo)	bytes	HKDF (RFC 5869): expand a high-entropy secret into length bytes. salt / info may be empty bytes.
crypto.pbkdf2(password, salt, iterations, keyLen, algo)	bytes	PBKDF2 (RFC 8018): stretch a low-entropy password over iterations rounds against salt.

`algo` names the PRF hash: `"sha1"` , `"sha256"` , or `"sha512"` (the `hash` library's names, minus `md5` - too weak to derive a key with). An unknown algorithm is a positioned error. Use `"sha256"` unless you are matching an external protocol: SCRAM-SHA-1 (MongoDB, XMPP) requires `"sha1"` .

- **HKDF** derives one or more subkeys from material that is already strong (a shared secret, a master key). It is fast; do not use it to hash passwords.
- **PBKDF2** is for passwords: the `iterations` count makes each guess expensive, so pick the largest value the deployment can afford. The total work - `ceil(keyLen / hashLen) × iterations` - is capped so an untrusted parameter (e.g. a hostile SCRAM server's `i=`) cannot pin a core for days; a single-block key allows up to ~10? iterations (worst case ~20 s), and a larger `keyLen` proportionally fewer. `keyLen` is also capped at 1 MiB, and `hkdf` 's `length` likewise - real key material is tens of bytes. It unblocks SASL SCRAM (see the `sasl` module) and password-based key wrapping.

```
use crypto;
use convert;
use encoding;

def password as bytes init convert.bytesFromString("correct horse", "utf-8");
def salt as bytes init crypto.randBytes(16);
def derived as bytes init crypto.pbkdf2($password, $salt, 600000, 32, "sha256");
io.printf("derived key: %s\n", encoding.toText($derived, "hex"));
```

Password hashing is not here

Storing passwords for later verification wants a memory-hard function (Argon2id, bcrypt, scrypt), which needs a dependency outside the standard library. That is deliberately out of scope for `crypto` ; use PBKDF2 only where an interoperable KDF is required (e.g. SCRAM), not as a general password store.

Authenticated encryption

`crypto.encrypt` / `crypto.decrypt` are **AES-256-GCM** , an authenticated (AEAD) cipher: the ciphertext carries a tag, so tampering is *detected* , not silently decrypted into garbage. There is one algorithm and no mode / nonce knobs - the footguns (ECB, IV reuse) are not expressible.

Call	Returns	Notes
crypto.encrypt(key, plaintext)	bytes	Seal. key must be exactly 32 bytes (AES-256). A fresh 12-byte nonce is generated and prepended: the result is nonce || ciphertext || tag, so you never handle (or reuse) a nonce.
crypto.decrypt(key, box)	bytes	Open. Splits the nonce, verifies the tag, returns the plaintext. A wrong key or a tampered box is a catchable authentication error.

```
use crypto;
use convert;

def key as bytes init crypto.randBytes(32);          # keep this secret
def box as bytes init crypto.encrypt($key, convert.bytesFromString("secret", "utf-
8"));
def back as bytes init crypto.decrypt($key, $box);   # throws if $box was tampered
```

The key is caller-managed - store it, or derive it from a password with `crypto.pbkdf2(..., 32, "sha256")`. Random 96-bit nonces can collide by the birthday bound, so NIST's guidance (SP 800-38D) caps one key at **2^32 messages** (~4 billion) to keep that chance below 2^-32 - rotate the key before then, which no ordinary workload approaches. Encrypting the same plaintext twice yields different boxes (fresh nonce each time).

Signatures

`crypto.sign` / `crypto.verify` are **Ed25519** - a modern signature scheme with no parameters to choose. A keypair is a namespaced struct `crypto.Keypair { public as bytes, private as bytes }`.

Call	Returns	Notes
crypto.signKeypair()	crypto.Keypair	A fresh keypair (32-byte public, 64-byte private).
crypto.sign(private, message)	bytes	A 64-byte signature over message.
crypto.verify(public, message, signature)	bool	true iff signature is public's signature over message; false on any mismatch. A malformed key / signature length is a positioned error.

```
use crypto;
use convert;

def kp as crypto.Keypair init crypto.signKeypair();
def msg as bytes init convert.bytesFromString("ship it", "utf-8");
def sig as bytes init crypto.sign($kp.private, $msg);
def ok as bool init crypto.verify($kp.public, $msg, $sig);   # true
```

Publish the `public` key; keep the `private` key secret. `verify` returning `false` means the message, key, or signature does not match - it never throws for a genuine mismatch, only for a wrong-length key or signature.

The random source draws from the same crypto-grade generator as `randBytes`. All Go stdlib (`crypto/aes` , `crypto/cipher` , `crypto/ed25519`), so both binaries carry it. Out of scope by design: password *hashing* (see above) and raw block modes.

Asymmetric signatures (RSA / ECDSA)

For interop with formats that mandate RSA or ECDSA - **JWT** RS256 / ES256 foremost - `crypto` signs and verifies with **PEM-encoded** keys. These are the one part of `crypto` that is **default-binary only** : they pull in `crypto/rsa` , `crypto/ecdsa` , and `crypto/x509` (for PEM parsing), which are off the TinyGo build, so on `jennifer-tiny` they raise a friendly "not available" error (the same build-tag split as `net`). For a modern signature with no key files, prefer Ed25519 (`crypto.sign` , above), which runs on both binaries.

Call	Returns	Notes

crypto.rsaSign(privatePem, message, algo)	bytes	RSASSA-PKCS#1 v1.5 signature. privatePem is a PKCS#1 or PKCS#8 PEM key (as bytes).
crypto.rsaVerify(publicPem, message, signature, algo)	bool	Verify against a PKIX / PKCS#1 public-key PEM. false on mismatch.
crypto.ecdsaSign(privatePem, message, algo)	bytes	ECDSA signature in the JOSE R||S form (fixed-width, what JWT / WebCrypto use), not ASN.1 DER. privatePem is a SEC1 or PKCS#8 EC key.
crypto.ecdsaVerify(publicPem, message, signature, algo)	bool	Verify against a PKIX EC public-key PEM. A wrong-length signature is false, not an error.

`algo` is the digest: `"sha256"` / `"sha384"` / `"sha512"` (JWT's 256 / 384 / 512 variants). The curve of an ECDSA key is taken from the key itself (P-256 for `ES256` , and so on). A key that is not valid PEM, or not the key type the call expects, is a positioned error; a genuine signature mismatch is `false` .

```
use crypto;
use fs;
use convert;

def priv as bytes init fs.readBytes("rsa_private.pem");
def pub as bytes init fs.readBytes("rsa_public.pem");
def msg as bytes init convert.bytesFromString("payload", "utf-8");
def sig as bytes init crypto.rsaSign($priv, $msg, "sha256");        # RS256
def ok as bool init crypto.rsaVerify($pub, $msg, $sig, "sha256");   # true
```

These are the primitives the `jwt` module's `RS*` / `ES*` algorithms build on. Still out of scope: x509 *certificate* handling (chains, SANs, expiry) - key parsing is all that is exposed.

Key generation, CSR, and JWK

The rest of the asymmetric surface - what an ACME client needs to mint keys and request certificates. Same default-binary-only split.

Call	Returns	Notes
crypto.rsaGenerateKey(bits)	bytes	A fresh RSA private key as PKCS#8 PEM. bits is 2048, 3072, or 4096.
crypto.ecGenerateKey(curve)	bytes	A fresh EC private key as SEC1 PEM. curve is "p256", "p384", or "p521".
crypto.jwkPublic(privatePem)	string	The RFC 7638 canonical public JWK JSON of the key (members sorted, no whitespace).
crypto.jwkToPem(jwkJson)	string	The inverse of jwkPublic: parse a public JWK (an RSA {kty, n, e} or EC {kty, crv, x, y} key) and return a PKIX / SubjectPublicKeyInfo PUBLIC KEY PEM - exactly the shape rsaVerify / ecdsaVerify accept. Resolves a JWKS entry (by kid) to a verification key. Curves P-256 / P-384 / P-521; an off-curve point or malformed JWK is an error.
crypto.csr(privatePem, domains)	bytes	A DER PKCS#10 certificate-signing request over a list of string of domains (subject-alt DNS names; the first is the common name), signed with the key.

`jwkPublic` is canonical, so its SHA-256 is the **JWK thumbprint** (RFC 7638) - the value ACME challenges are keyed on:

```
use crypto;
use hash;
use encoding;

def key as bytes init crypto.ecGenerateKey("p256");
def jwk as string init crypto.jwkPublic($key);               # {"crv":"P-
256","kty":"EC",...}
def thumbprint as bytes init hash.compute(convert.bytesFromString($jwk, "utf-8"),
"sha256");
def csr as bytes init crypto.csr($key, ["example.com", "www.example.com"]);
```

Errors

Every function validates argument kinds and counts and raises a positioned runtime error on misuse (wrong type, negative `randBytes` length, `lo` > `hi` , non-positive `iterations` / `keyLen` / `length` , a key that is not 32 bytes for `encrypt` / `decrypt` , a wrong-length Ed25519 key or signature). The secure random source is assumed always available on the supported platform; an impossible failure aborts rather than returning weak bytes.

encoding - introspection + text and character codecs

Enable with `use encoding;` . Three groups of functions:

1. **Introspection** - rune count vs byte count, and an ASCII test.
2. **Binary-to-text** (`toText` / `fromText`) - hex, base64, quoted-printable, and URI percent / form encoding, `bytes` to a printable `string` and back.
3. **Character codecs** (`encode` / `decode`) - a Jennifer `string` to and from a single-byte legacy encoding (ISO-8859- , *Windows*- , EBCDIC).

The cross-kind UTF-8 codec ships with `convert` (`convert.bytesFromString` / `convert.stringFromBytes`); this library is where the rest of the codec proliferation lives.

```
use io;
use convert;
use encoding;

def src as bytes init convert.bytesFromString("café", "utf-8");
io.printf("%s\n", encoding.toText($src, "hex"));           # 636166c3a9
io.printf("%t\n", encoding.isAscii($src));                 # false
io.printf("%x\n", encoding.encode("café", "iso-8859-1"));  # 63 61 66 e9
```

Introspection

Call	Returns	Notes
encoding.isAscii(b)	bool	True iff every byte is < 0x80. Empty bytes is true.
encoding.lenBytes(s)	int	Byte length of s's UTF-8 encoding. Pair with len(s) (runes).
encoding.lenRunes(b)	int	Rune count of valid UTF-8 bytes; errors on invalid UTF-8.

Binary-to-text: toText / fromText

One verb pair, the encoding named by a string. Reversible representation - these grow or reshape the bytes, they don't reduce information (that's `compress`).

Call	Returns	Notes
encoding.toText(b, format)	string	Encode bytes as printable text.
encoding.fromText(s, format)	bytes	Decode back to bytes.

Formats

format	Standard	Notes
"hex"	base-16	Lowercase on encode; decode accepts upper and lower case. Two chars per byte.
"base32"	RFC 4648 §6	Standard alphabet (A-Z 2-7), = padding.
"base32-hex"	RFC 4648 §7	Extended-hex alphabet (0-9 A-V), = padding.
"base64"	RFC 4648 §4	Standard alphabet (+ /, = padding).
"base64-url"	RFC 4648 §5	URL / filename-safe alphabet (- _).
"ascii85"	Adobe / btoa	Base-85, the !..u alphabet, with z for an all-zero group.
"z85"	ZeroMQ RFC 32	Base-85, a source-safe alphabet, no padding. Input must be a multiple of 4 bytes (decode input a multiple of 5 chars).
"quoted-printable"	RFC 2045	MIME transfer encoding (see below).

"uri-percent"	RFC 3986	RFC 3986 percent-encoding (see below). Space is %20.
"uri-form"	WHATWG URL	application/x-www-form-urlencoded (see below). Space is +.

Quoted-printable keeps printable ASCII literal, turns = , control, and 8-bit bytes into =XX , and soft-wraps lines to 76 columns with a trailing = . Decode reverses it, tolerant of both CRLF and LF soft breaks, and round-trips `bytes` .

URI percent / form. `"uri-percent"` is RFC 3986 percent-encoding: every byte outside the unreserved set (`A - Z a - z 0 - 9 - . _ ~`) becomes %XX (uppercase hex), and a space becomes `%20` . `"uri-form"` is the `application/x-www-form-urlencoded` variant used for query strings and HTML form bodies - identical, except a space becomes + (and a literal + therefore encodes as `%2B`). Decode reverses each; a malformed % escape (`%` , `%2` , `%zz`) is a positioned error. These are the byte-level codecs; the `uri` module builds on them for URL parsing, building, and query strings (its `encode` / `decode` are `uri-percent` , `encodeForm` / `decodeForm` and `buildQuery` / `parseQuery` are `uri-form`).

Format names are **exact** (case-sensitive, no `-` / _ normalisation) - they're the library's own fixed set, not external standards with variant spellings. `"base64"` works; `"BASE64"` errors with the supported set listed.

> **Why a format string instead of encoding.hex() / encoding.base64()?** Jennifer's letters-only identifier rule rejects digits in method names, so `encoding.base64` won't parse; and the codec-table shape already matches the rest of this library plus `convert` , `hash` , and `crc` .

Character codecs: encode / decode

Convert a Jennifer `string` to and from a named single-byte encoding.

Call	Returns	Notes
encoding.encode(s, codec)	bytes	Errors when a rune has no byte in the codec (e.g. € in ASCII).
encoding.decode(b, codec)	string	Errors when a byte is undefined in the codec.
encoding.codecs()	list of string	Every registered codec name, in registration order.

Codec set

Every codec below is single-byte: one byte maps to one rune. Bytes with no assignment in a codec (some Windows code pages have a few) are a positioned error on decode; runes with no byte are a positioned error on encode.

ASCII and EBCDIC

Codec	Notes
"ascii"	7-bit US-ASCII. Rejects any byte or rune >= 0x80.
"ebcdic"	IBM Code Page 1047 (the Open Systems Latin-1 EBCDIC variant).

ISO/IEC 8859 (single-byte Latin and script families)

Codec	Part	Coverage
"iso-8859-1"	Latin-1	Western European (identity U+0000..U+00FF).
"iso-8859-2"	Latin-2	Central / Eastern European.
"iso-8859-3"	Latin-3	South European (Maltese, Esperanto).
"iso-8859-4"	Latin-4	North European (Baltic).
"iso-8859-5"	Cyrillic	Latin / Cyrillic.
"iso-8859-6"	Arabic	Latin / Arabic.
"iso-8859-7"	Greek	Latin / Greek.

"iso-8859-8"	Hebrew	Latin / Hebrew.
"iso-8859-9"	Latin-5	Turkish.
"iso-8859-10"	Latin-6	Nordic.
"iso-8859-11"	Thai	Latin / Thai.
"iso-8859-13"	Latin-7	Baltic Rim.
"iso-8859-14"	Latin-8	Celtic.
"iso-8859-15"	Latin-9	Western European, Latin-1 with the euro sign at 0xA4 and Š š Ž ž Œ œ Ÿ.
"iso-8859-16"	Latin-10	South-Eastern European.

(There is no `iso-8859-12` ; the draft was abandoned.)

Windows code pages

Codec	Coverage
"windows-1250"	Central European.
"windows-1251"	Cyrillic.
"windows-1252"	Western European - Latin-1 plus the 0x80..0x9F "smart quotes" set (incl. the euro sign). Five positions (0x81 0x8D 0x8F 0x90 0x9D) are undefined.
"windows-1253"	Greek.
"windows-1254"	Turkish.
"windows-1255"	Hebrew.
"windows-1256"	Arabic.
"windows-1257"	Baltic.
"windows-1258"	Vietnamese.

Codec names are exact

Codec names are **exact-match** - the one canonical spelling only, no case-folding, separator-stripping, or IANA aliases. `"iso-8859-1"` works; `"latin-1"` , `"ISO-8859-1"` , `"iso88591"` , and `"cp1252"` all error with the known set listed. This is deliberate strictness (stance #2, explicit over implicit): a codec is named one way, and no lenient spelling hides a typo. Map an external name (an HTTP `charset=ISO-8859-1` , say) to the canonical form yourself before calling.

How the tables are built

Only `ascii` (7-bit, with its own out-of-range errors) and `ebcdic` (IBM-1047, not in the standard Unicode mapping path) are hand-written. Every ISO-8859 and Windows single-byte codec - including `iso-8859-1` and `windows-1252` - is **generated from the Unicode Consortium mapping files** , so every table comes from one authoritative source rather than being hand-transcribed:

```
go generate ./internal/lib/encoding/   # writes codecs_gen.go
```

Converting a text file

Jennifer strings are UTF-8 internally, so converting a legacy-encoded file to UTF-8 is just decode-then-write - there's no separate "encode to UTF-8" step:

```
use fs;
use encoding;

def raw as bytes init fs.readBytes("legacy.txt");              # Windows-1252
bytes
def text as string init encoding.decode($raw, "windows-1252"); # -> a UTF-8 string
fs.writeString("utf8.txt", $text);                             # written as UTF-8
```

The reverse (UTF-8 to a single-byte encoding) is `encoding.encode(text, codec)`, which errors if a character has no byte in the target codec.

Length and memory

`encode` / `decode` / `toText` / `fromText` are **one-shot**: each takes the whole input and builds the whole output in memory, so both are held at once. There is no fixed cap - a Jennifer `string` / `bytes` is a Go `string` / `[]byte`, bounded only by available memory. For a file too large to hold twice over, note that the single-byte codecs decode each byte independently, so decoding can be split at any byte boundary: read the file in fixed-size byte chunks, `decode` each, and append to the output. (Encoding the other way must instead split on rune boundaries, since a UTF-8 character can span several bytes.)

Errors

- Wrong arity: `encoding.encode expects 2 arguments (string, codec), got 1`.
- Wrong scalar type: `encoding.encode: first argument must be string, got bytes`.
- Unknown codec (the message lists every registered name): `encoding.encode: unknown codec "klingon"; known: "ascii", "iso-8859-1", ...`.
- Unrepresentable rune on encode: `encoding.encode (ascii): rune U+00E9 at byte position 3 is outside ASCII (0x00..0x7F)`.
- Undefined byte on decode: `encoding.decode (windows-1252): byte 0x81 at position 0 has no mapping in windows-1252`.
- Invalid UTF-8 to `lenRunes`: `encoding.lenRunes: input is not valid UTF-8`.

All errors are positioned at the call site.

See also

- convert.md - the UTF-8 pair (`bytesFromString` / `stringFromBytes`).
- compress.md - size reduction (distinct from representation).
- hash.md, crc.md - digests whose output you'll often `encoding.toText($digest, "hex")`.

fs - filesystem I/O

Enable with `use fs;` . Blocking whole-file reads and writes, filesystem metadata, directory operations, and buffered file handles for line-oriented reads. Non-blocking use composes with `spawn` rather than duplicating each call as a `*Async` variant.

```
use io;
use fs;

fs.writeString("hello.txt", "hi, world\n");
def content as string init fs.readString("hello.txt");
io.printf("%s", $content);
```

One-shot operations

Whole-file reads and writes. Cheap to write, cheap to read.

Call	Returns	Notes
fs.readString(path)	string	Whole file as UTF-8. Invalid UTF-8 is a positioned runtime error.
fs.readBytes(path)	bytes	Whole file as raw bytes. See below for the handle form.
fs.writeString(path, content)	null	Overwrites. Creates the file if missing. See below for the handle form.
fs.writeBytes(path, content)	null	Overwrites. Creates the file if missing. See below for the handle form.
fs.appendString(path, content)	null	Appends. Creates the file if missing.
fs.appendBytes(path, content)	null	Appends. Creates the file if missing.

Reading a file that doesn't exist is a positioned runtime error; use `fs.exists(path)` first when the "missing file is normal" case matters.

Metadata

Boundary-friendly predicates plus a full `fs.stat` .

Call	Returns	Notes
fs.exists(path)	bool	True if the path resolves. Permission errors still surface.
fs.isFile(path)	bool	True iff the path exists and is a regular file. False for missing.
fs.isDir(path)	bool	True iff the path exists and is a directory. False for missing.
fs.stat(path)	fs.Stat	Missing path errors. Pair with fs.exists when tolerating absent files.
fs.realpath(path)	string	Absolute, symlink-resolved canonical path. A missing / unresolvable path errors.
fs.readlink(path)	string	The target a symlink points to, verbatim (the stored link text, not resolved; use fs.realpath to fully resolve). A non-symlink or missing path errors.
fs.symlink(target, linkPath)	null	Create a symlink at linkPath pointing to target (argument order mirrors ln -s TARGET LINK). An already-existing linkPath errors - fs.remove it first to re-point.

`fs.realpath` follows every symlink in the chain and makes the result absolute - the userland companion to the entry-script resolution the interpreter already does for module imports. A tool installed by symlinking its entry point onto `PATH` gets the *invocation* path in `os.ARGS[0]` (the symlink), so to load data files that sit beside its real source rather than beside the symlink, resolve it:

```
use os; use path; use fs;
def self as string init fs.realpath(os.ARGS[0]);   # the real file, symlinks
followed
def assets as string init path.join(path.dir($self), "assets");
```

fs.Stat

```
def struct fs.Stat {
    path as string,        # the path as passed to fs.stat / fs.walk
    size as int,           # bytes; -1 for directories
    isDir as bool,
    mtimeNanos as int,     # Unix nanoseconds
    mode as int            # POSIX permission bits (e.g. 0o644, 0o755)
};
```

`mtimeNanos` is deliberately an `int` , not a `time.Time` - that keeps `fs` decoupled from the `time` library at the Go-package level. Composition is one line:

```
use fs;
use time;

def stat as fs.Stat init fs.stat("hello.txt");
def modified as time.Time init time.fromUnixNanos($stat.mtimeNanos);
io.printf("modified: %s\n", time.iso($modified));
```

`size` is `-1` for directories so callers don't accidentally interpret it as "empty directory."

Permissions

The write verbs create files at a fixed `0644` ; `fs.chmod` tightens (or loosens) a file after the fact - e.g. to `0600` for a secret / token store that must not be world-readable. `fs.chown` sets owner and group.

Call	Returns	Notes
fs.chmod(path, mode)	null	Sets the permission bits; mode is the low 12 bits (e.g. 0o600, 0o755), rejected outside [0, 0o7777].
fs.chown(path, uid, gid)	null	Sets owner / group; -1 leaves that id unchanged. Usually needs privilege.

```
use fs;
fs.writeString("token.json", $json);
fs.chmod("token.json", 0o600);   # owner-only, like a gh / aws token store
```

Both are **Linux/Unix semantics** and match the platform target. On Windows `fs.chmod` honours only the read-only bit and `fs.chown` always fails; the error is catchable like any other `fs` failure.

Directory operations

The library ships **two verbs each** for create and delete (`mkdir` / `mkdirAll` , `remove` / `removeAll`). The safe non-recursive form keeps the short name; the recursive form gets an explicit second name so a code review can grep for the risky sites. This is Jennifer's "no footguns" stance applied at the API level - a `bool` parameter or a mode string would obscure the recursive intent at the call site.

Call	Returns	Notes
fs.mkdir(path)	null	Errors if any parent is missing (matches POSIX mkdir).
fs.mkdirAll(path)	null	Creates every missing parent (matches POSIX mkdir -p).
fs.remove(path)	null	Removes one file or one empty directory. Non-empty dir errors.

fs.removeAll(path)	null	Recursive delete. Explicit second verb.
fs.rename(old, new)	null	Same-filesystem rename; cross-fs is a boundary error.
fs.list(path)	list of string	Sorted entry names. Non-recursive.
fs.walk(path)	list of fs.Stat	Depth-first, sorted, includes path itself as the first entry. Skips symlinks.

```
use fs;

# Safe: mkdir refuses to create with missing parents.
fs.mkdirAll("build/output/cache");        # explicit intent

# Safe: remove refuses to delete non-empty directories.
fs.removeAll("build/output/cache");       # explicit intent
```

Temporary files and directories

`fs.makeTempFile` and `fs.makeTempDir` create a fresh, uniquely-named entry and return its **path** . Creation is atomic - the OS reserves the name, so two concurrent callers never collide - and restrictive (`0600` for a file, `0700` for a directory), so scratch data holding secrets is not world-readable. They return a path (not an open handle) so the result drops straight into the ordinary path-based verbs.

Call	Returns	Notes
fs.makeTempFile([dir[, prefix[, suffix]]])	string	Create an empty unique file; returns its path.
fs.makeTempDir([dir[, prefix]])	string	Create a unique directory; returns its path.

Arguments are positional and trailing-optional, `""` meaning the default:

- `dir` - where to create it. `""` (or omitted) uses the system temp directory (`os.tempDir()` - `$TMPDIR` else `/tmp` on Linux). An explicit `dir` **must already exist** : only the final unique component is created (like `fs.mkdir` , not `fs.mkdirAll`), so run `fs.mkdirAll` first if the parent is missing.
- `prefix` / `suffix` - name the entry around the random component; `suffix` gives a file a real extension. Neither may contain a path separator or NUL (that would escape the target directory), which is a positioned error.

```
use fs;

# A scratch file in the system temp dir, cleaned up when done.
def report as string init fs.makeTempFile("", "report-", ".json");
defer fs.remove($report);              # cleanup runs however the block exits
fs.writeString($report, $payload);
# ... use it ...

# A scratch directory under an existing build tree (create the parent first).
fs.mkdirAll("build");
def work as string init fs.makeTempDir("build", "job-");
defer fs.removeAll($work);
# ... write files under $work ...
```

Both are **strict** : any OS failure - a read-only filesystem, no permission, a missing parent directory, or a name the filesystem cannot hold - is a catchable error, never a fabricated directory tree or a name mangled to fit. Nothing is cleaned up automatically: pair every `makeTemp*` with an `fs.remove` / `fs.removeAll` , ideally as a `defer` right after the create (as above) so the cleanup also runs when an error unwinds through the block (control-flow > `defer` ; see the concurrency and error notes below).

File handles

For line-oriented reads and files that don't fit comfortably in memory, `fs.open` returns an `fs.File` handle backed by the integer-registry pattern also used by `hash.Stream` and `crc.Stream` .

```
def struct fs.File { id as int };
```

Mode strings use the codec-table shape:

Mode	Semantics
"read"	Read-only. fs.readLine, fs.readChars, fs.readBytes allowed.
"write"	Write, create+truncate. fs.writeString, fs.writeBytes allowed.
"append"	Write, create+append. Same write ops as "write".

Unknown mode strings error with the known set listed.

Handle surface

Call	Returns	Notes
fs.open(path, mode)	fs.File	Opens per the mode string.
fs.close($f)	null	Removes the handle from the registry; later ops on the id error.
fs.readLine($f)	string	One line; \r\n / \n stripped. Errors on EOF - check fs.eof first.
fs.readChars($f, n)	string	Up to n runes, UTF-8 decoded. Partial result on EOF, sticky-EOF flip.
fs.readBytes($f, n)	bytes	Up to n bytes. Partial result on EOF, sticky-EOF flip.
fs.writeString($f, s)	null	Read-mode handle errors.
fs.writeBytes($f, b)	null	Read-mode handle errors.
fs.sync($f)	null	Flush written data to the storage device (fsync); handle stays open. Read-mode handle errors.
fs.eof($f)	bool	Looks ahead: true iff the next read would error / return partial.

The canonical read-loop:

```
use io;
use fs;

def f as fs.File init fs.open("input.txt", "read");
while (not fs.eof($f)) {
    def line as string init fs.readLine($f);
    io.printf("[%s]\n", $line);
}
fs.close($f);
```

`fs.readLine` on a handle whose next read is EOF errors with `end of input` ; the `fs.eof($f)` guard is what keeps the loop tight. `fs.eof` looks one byte ahead (through the buffered reader) so a file ending cleanly with `\n` still trips the guard after the last line comes out.

Durability: sync vs close. `fs.close` guarantees the bytes reach the *operating system* , not the *disk* - the OS may still hold them in its cache after close returns. `fs.sync($f)` forces the file's written data all the way to the storage device (an `fsync`) and leaves the handle open. This is the "safe to remove the USB stick" step: sync explicitly and let a failure surface *before* you tell the user it is safe, rather than discovering the write never landed:

```
use fs;
use io;

def f as fs.File init fs.open("/media/usb/report.csv", "write");
fs.writeString($f, $payload);
fs.sync($f);                          # push it to the device; errors if the
write failed
io.printf("safe to remove the stick\n");
fs.close($f);
```

`fs.sync` requires a write- or append-mode handle (a read handle has nothing to flush). Note `fsync` covers the OS side; a removable device's own internal cache is the job of the OS eject / unmount.

Polymorphic verbs (path vs handle)

Three verbs are polymorphic on the first argument's kind:

Verb	1-arg form	2-arg form
fs.readBytes	(path) -> whole	($f, n) -> partial handle read
fs.writeString	-	(path, content) or ($f, s)
fs.writeBytes	-	(path, content) or ($f, b)

The dispatcher picks based on whether the first argument is a `string` (path form) or an `fs.File` (handle form). Any other kind is a positioned boundary error.

Handles share state between copies

An `fs.File{id}` value is small; copies share the underlying Go `*os.File` state via the integer id. This mirrors the `task of T` carve-out to the "value semantics everywhere" rule:

```
def a as fs.File init fs.open("x.txt", "read");
def b as fs.File init $a;             # `b` and `a` reference the same file
fs.close($a);                         # closes for both
def s as string init fs.readLine($b);  # errors: id is not open
```

Handles are the second "handles wrap shared state" carve-out in the language, sitting alongside `task of T`. Every other type keeps whole-value semantics.

Watching for changes

`fs.watch` starts a background watch over a file or directory and returns an `fs.Watcher` handle; a pull loop drains change events - the same shape as an `httpd` server, so a `spawn` ed loop reacts to changes while the program does other work (a dev server rebuilding on save, a live reloader).

Call	Returns	Notes
fs.watch(path)	fs.Watcher	Watch a file, or a directory recursively. Default poll ~300 ms.
fs.watch(path, intervalMs)	fs.Watcher	Same, with an explicit poll interval (>= 20 ms).
fs.next($w)	fs.Event	Block until the next change; errors if the watcher is closed (which is how a blocked loop is released).
fs.hasEvent($w)	bool	Whether an event is queued (non-blocking peek).
fs.close($w)	null	Stop the watcher. fs.close is polymorphic over fs.File and fs.Watcher.

An `fs.Event` is `{ path as string, kind as string, isDir as bool }`, where `kind` is `"created"`, `"modified"`, or `"deleted"`.

```
use fs; use io;
def w as fs.Watcher init fs.watch("content", 200);
def worker as task of null init spawn {
    try {
        while (true) {
            def e as fs.Event init fs.next($w);   # blocks until something changes
            if (not $e.isDir) {                    # ignore bare directory-mtime
bumps
                io.printf("rebuild: {$e.kind} {$e.path}\n");
            }
        }
    } catch (stop) {
        # fs.next errors when fs.close releases the watcher - a clean shutdown.
    }
};
# ... serve here; `fs.close($w)` ends the loop, then `task.wait($worker)`.
```

How it works, and the trade-offs. The watch is **mtime polling** (a background goroutine re-scans the tree each interval and diffs a snapshot), not inotify - so it is pure-stdlib, dependency-free, and works on both binaries, matching `jennifer serve --watch` . Two consequences worth knowing:

- **Bursts coalesce.** An editor's save (write / rename / chmod) within one poll interval collapses into one event per path - a save is one `"modified"` , not three. Very short-lived transients (a file created and deleted between two polls) may not be reported; the watch reflects the *net* change.
- **Directory mtimes fire.** Adding or removing a file bumps its directory's mtime, so you get a `"modified"` event for the directory alongside the child's `"created"` / `"deleted"` . Filter with `$e.isDir` when you only care about files.

For a huge tree (100k+ files) polling gets expensive; that is not the target (dev tooling watches hundreds to low-thousands of files), and an inotify-backed variant could be added later behind a build tag if it is ever needed.

Concurrency composition

`fs` is blocking on purpose. Non-blocking use is a one-line composition with `spawn` :

```
use fs;
use task;

def t as task of string init spawn {
    return fs.readString("/etc/hosts");
};
def content as string init task.wait($t);
```

Multiple files in parallel:

```
use fs;
use task;

func loadOne(path as string) {
    return fs.readString($path);
}

def paths as list of string init ["a.txt", "b.txt", "c.txt"];
def tasks as list of task of string init [];
for (def p in $paths) {
    $tasks[] = spawn { return loadOne($p); };
}
def contents as list of string init task.waitAll($tasks);
```

Under the default `jennifer` binary this actually parallelises across cores; under `jennifer-tiny` (TinyGo, cooperative single-threaded 0.41) the composition is correct but sequential. See ../technical/tinygo.md.

Errors

Every error is positioned at the Jennifer call site with the path or handle id in the message.

- **Missing files** on `fs.readString` / `fs.readBytes` / `fs.stat` : `fs.readString: PATH: open PATH: no such file or directory` .
- **Non-empty directory** given to `fs.remove` : `fs.remove: PATH: directory not empty` . Use `fs.removeAll` for recursive delete.
- **Missing parent** given to `fs.mkdir` : `fs.mkdir: PATH: no such file or directory` . Use `fs.mkdirAll` for `mkdir -p` .
- **Wrong mode for op** : `fs.readLine: fs.File "x.txt" was opened in mode "write"; open with mode "read" to read` . Same message shape in reverse for read-mode handles given a write op.
- **Unknown mode string** : `fs.open: unknown mode "rw"; known: "read", "write", "append"` .
- **Use after close** : `fs.readLine: fs.File id 3 is not open (already closed, or never opened)` .
- **Non-negative int required** : `fs.readChars: n must be non-negative, got -1` .

Every error is catchable with `try` / `catch` :

```
try {
    def s as string init fs.readString("optional-config.toml");
} catch (e) {
    io.printf("no config, using defaults: %s\n", $e.message);
}
```

What's not in v1

Recorded so the design decisions stay visible; ships if a concrete workload forces it.

- **Streaming line iterator** (`for (def line in fs.lines(path))`). Compose with `fs.open` + `while (not fs.eof)` .
- **fs.copy(src, dst)** and **fs.chmod(path, mode)** .
- **fs.stat($f) on an open handle** . Only path-based `fs.stat` in v1.
- **Watch / notify** (inotify / kqueue / FSEvents).
- **Temp file / dir creation helpers** (`fs.tempFile` , `fs.tempDir`). Use `os.getEnv("TMPDIR")` plus your own naming.
- **Follow symlinks in fs.walk** . Symlink-loop protection is the reason for the deferral; a `follow=true` flag or a distinct `fs.walkFollowing` verb ships once the story is clear.

See also

- ../user-guide/concurrency.md - the `spawn` -and-compose story `fs` builds on.

- `task` - observe handles produced by `spawn` .
- `time` - `time.fromUnixNanos` converts `fs.Stat.mtimeNanos` into a `time.Time` .
- `os` - process-level operations that pair with `fs` : env vars, argv, external commands.
- ../milestones.md - design spec.

hash - cryptographic-style digests

Enable with `use hash;` . Computes fixed-size digests over `bytes` using common cryptographic-style algorithms (MD5, SHA-1, SHA-256, SHA-384, SHA-512). Output is raw `bytes` ; users hex- or base64-encode through the `encoding` library when they need a string representation.

For non-cryptographic checksums (transport integrity rather than content addressing), use `crc` instead. The split makes the semantic difference visible at the import line.

```
use io;
use convert;
use hash;

def input as bytes init convert.bytesFromString("abc", "utf-8");
def digest as bytes init hash.compute($input, "sha256");
io.printf("sha256 digest is %d bytes\n", len($digest));
```

Algorithm selection

The library uses the codec-table shape - one verb per category, with the algorithm passed as a string. The shape mirrors `convert.bytesFromString(s, "utf-8")` and `encoding.encode(s, codec)` . Algorithm names are lowercase.

Algo string	Output width	Notes
"md5"	16 bytes	Non-cryptographic (broken for collision resistance) - use for integrity / caching / legacy interop, never security.
"sha1"	20 bytes	Non-cryptographic (broken for collision resistance) - legacy formats (e.g. SCRAM-SHA-1) only, never security.
"sha256"	32 bytes	The default choice for new code.
"sha384"	48 bytes	Truncated SHA-512; used by some TLS / JOSE suites.
"sha512"	64 bytes	Wider SHA-2 digest; used by some HMAC / TOTP variants.

Passing an unknown algorithm is a positioned runtime error that lists the supported set: `hash.compute: unknown digest algorithm "md4"; known: "md5" (non-cryptographic), "sha1" (non-cryptographic), "sha256", "sha384", "sha512"` .

One-shot

Call	Returns	Notes
hash.compute(b, algo)	bytes	Full digest of the entire input.
hash.hmac(key, message, algo)	bytes	Keyed-hash MAC (RFC 2104) over the same algorithms.
hash.equal(a, b)	bool	Constant-time equality of two bytes (see below).

The algorithm is always a value (`compute(b, "sha256")`), the same shape as `hmac` and the `crypto` KDFs - one convention across the whole hash / crypto family, so a runtime-negotiated algorithm (SCRAM, JWT) is just a variable, not a dispatch ladder. There are deliberately no `hash.sha256(b)` shortcuts (see rejected.md).

HMAC

`hash.hmac(key, message, algo)` computes the keyed-hash message authentication code (RFC 2104) - the primitive behind JWT (HS256), TOTP, AWS SigV4, and webhook signatures. `key` and `message` are `bytes` ; the result is the raw MAC as `bytes` (hex / base64 via `encoding` , matching `compute`).

```
use hash;
use convert;
use encoding;

def key as bytes init convert.bytesFromString("secret", "utf-8");
def msg as bytes init convert.bytesFromString("payload", "utf-8");
def mac as bytes init hash.hmac($key, $msg, "sha256");
io.printf("%s\n", encoding.toText($mac, "hex"));
```

To **verify** , recompute the MAC over the same message and compare it to the received one (comparing the full digests, not a prefix) with `hash.equal` .

Constant-time comparison

`hash.equal(a, b)` reports whether two `bytes` are identical in **constant time** - the comparison takes the same time whether the values differ in the first byte or the last (`crypto/subtle.ConstantTimeCompare`). Use it to check a received MAC, digest, or token against the expected one: a plain == on the raw bytes (or a byte-by-byte loop that returns early) leaks *where* the first mismatch is through its timing, which over many attempts lets an attacker forge a valid MAC one byte at a time. Values of different lengths compare unequal.

```
use hash;

# expected and received are both `bytes` (e.g. two MACs)
if (hash.equal($expected, $received)) {
    # authentic
}
```

Reach for `hash.equal` (or `crypto.hmacEqual` , the same primitive) whenever the comparison is a security decision; ordinary == on non-secret data is fine.

Streaming

For inputs that don't fit comfortably in memory (files, large network reads), feed chunks into a stream handle and finalize:

```
use hash;
def s as hash.Stream init hash.stream("sha256");
hash.update($s, $chunkOne);
hash.update($s, $chunkTwo);
def digest as bytes init hash.finalize($s);
```

Call	Returns	Notes
hash.stream(algo)	hash.Stream	Allocate a fresh handle for the named algorithm.
hash.update($s, $bytes)	null	Feed one chunk. Mutates the handle's state by side effect.
hash.finalize($s)	bytes	Compute the digest and consume the handle. Subsequent calls error.
hash.discard($s)	null	Drop a stream without computing its digest, releasing its state. The abort path for a handle opened but abandoned. Subsequent calls error.

`hash.Stream` carries an integer `id` field that indexes into a Go-side map of live state. Users should not read or mutate the field; pass the struct around as an opaque token.

Composing through convert and (future) encoding

No convenience wrappers like `hash.md5String(s)` ship. Stance #1 "one way per thing": strings become bytes through `convert.bytesFromString` , the digest stays as bytes, and the user hex-encodes through `encoding.hex` . The example `examples/hash.j`

(https://github.com/jennifer-language/jennifer/blob/main/examples/hash.j) carries a tiny inline `bytesToHex` helper for printing until `encoding` ships.

Errors

- Wrong arity: `hash.compute expects 2 arguments (bytes, algo), got 1` .
- Wrong scalar type: `hash.compute: first argument must be bytes, got string` .
- Unknown algorithm: `hash.compute: unknown digest algorithm "md4"; known: "md5" (non-cryptographic), "sha1" (non-cryptographic), "sha256"` .
- Reuse of a finalized stream: `hash.update: stream 3 has already been finalized or never existed` .

All errors are positioned at the call site.

See also

- crc.md - non-cryptographic checksums (CRC-32, CRC-64).
- milestones.md - hash/crc, `encoding` for hex/base64 round-trips, and key-based crypto.

httpd - HTTP server engine

Enable with `use httpd;` . An HTTP/1.1 server engine wrapping Go's `net/http` , so keep-alive, chunked transfer, TLS (and HTTP/2 over TLS), request timeouts, and graceful shutdown come from the battle-tested Go stack rather than being re-implemented in the interpreter. It is the server counterpart to the `net` client primitives and the `http` client module.

Default binary only. Like `net` , `httpd` needs a network stack, so it runs on the standard `jennifer` build; on `jennifer-tiny` every call returns a friendly error (TinyGo ships no netdev driver). See technical/tinygo.md.

The pull loop

You cannot hand Go's `net/http` a Jennifer request-handler callback: the interpreter is not re-entered from Go's handler goroutines. Instead the engine accepts and parses requests concurrently on Go's side and hands them to your program **one at a time** : `httpd.accept` blocks for the next request, and `httpd.respond` answers it.

```
use httpd;

def srv as httpd.Server init httpd.listen("127.0.0.1:8080");
while (true) {
    def req as httpd.Request init httpd.accept($srv);
    httpd.respond($req, 200, "hello\n");
}
```

The two concurrency worlds stay cleanly separate: **Go owns the I/O concurrency** (accepting, parsing, keep-alive), and your program stays a simple serial loop. When you *want* per-request parallelism, opt into it with your own `spawn` - several `spawn` ed workers can each call `httpd.accept` on the same server handle to form a worker pool, since the handle's state is shared:

```
use httpd;
use task;

def srv as httpd.Server init httpd.listen("127.0.0.1:8080");
def workers as list of task of null init [];
for (def i in lists.range(0, 4)) {
    $workers[] = spawn {
        while (true) {
            def req as httpd.Request init httpd.accept($srv);
            httpd.respond($req, 200, "handled by a worker\n");
        }
    };
}
```

Surface

Call	Returns	Notes
httpd.listen(addr)	httpd.Server	Start listening. "127.0.0.1:8080" (TCP), ":0" (ephemeral TCP port), or "unix:/run/app.sock" (a Unix domain socket).
httpd.listenTLS(addr, cert, key)	httpd.Server	HTTPS; cert / key are PEM bytes. HTTP/2 negotiated automatically.
httpd.address(srv)	string	The actual bound address (resolve ":0" to the chosen port).
httpd.accept(srv)	httpd.Request	Block for the next request. Errors once the server is shut down.
httpd.method(req)	string	"GET", "POST", ...

httpd.path(req)	string	URL path, e.g. /users/42.
httpd.query(req, name)	string	Query parameter ("" if absent).
httpd.header(req, name)	string	Request header ("" if absent; case-insensitive name).
httpd.body(req)	bytes	The request body (buffered; a body over the 10 MiB cap is answered 413 by the engine).
httpd.remoteAddr(req)	string	Client host:port.
httpd.setHeader(req, name, value)	null	Set a response header (before respond).
httpd.respond(req, status, body)	null	Send the response; body is a string or bytes.
httpd.serveFile(req, path)	null	Answer with a file (content type, range requests handled by net/http).
httpd.serveDir(req, root)	null	Answer with the file under root matching the request path (.. cannot escape root).
httpd.shutdown(srv)	null	Graceful drain: stop accepting, unblock parked accept calls, finish in-flight requests.

Each request must be answered **exactly once** - a second `respond` / `serveFile` / `serveDir` on the same request, or a `setHeader` after the answer, is an error.

Handles

`httpd.Server` and `httpd.Request` are `{id as int}` handles into a Go-side registry (the same pattern as `fs` , `net` , `os.Process`): value-semantic to copy, but every copy refers to the same underlying server / request. That is what lets a copied `Server` handle inside a `spawn` worker pull from the same accept queue.

A tiny JSON API

Everything the engine hands you is a value, so the rest of the standard library composes normally - here, `json` for the response body:

```
use httpd;
use json;

def srv as httpd.Server init httpd.listen(":8080");
while (true) {
    def req as httpd.Request init httpd.accept($srv);
    def out as json.Value init json.map();
    $out = json.set($out, "/method", httpd.method($req));
    $out = json.set($out, "/path", httpd.path($req));
    httpd.setHeader($req, "Content-Type", "application/json");
    httpd.respond($req, 200, json.encode($out));
}
```

Static files

```
use httpd;
def srv as httpd.Server init httpd.listen(":8080");
while (true) {
    def req as httpd.Request init httpd.accept($srv);
    httpd.serveDir($req, "./public");
}
```

`serveDir` cleans the request path so a `../` cannot climb above `root` , rejects a request path containing a backslash (`400`), and re-verifies the joined path is still under `root` before serving it (`404` otherwise); `serveFile` answers with one specific file regardless of the request path.

Symlinks are followed. Both verbs open the resolved path directly (Go's `http.ServeFile` behaviour), so a symbolic link *inside* `root` that points outside it exposes its target. If the served tree can contain links created by another user or an upload feature, resolve and containment-check the path yourself before serving, or serve from a directory you fully control.

Response headers and the Server line

The engine adds **no Server: header** . Go's `net/http` sends `Date` and the framing headers (`Content-Length` / `Transfer-Encoding`) but never advertises the server software, and `httpd` adds nothing on top - so a response carries no `Server: ...` fingerprint (no stack name, no version a scanner can match a CVE against). This is a deliberate default; you opt *in* to advertising a server, you do not opt out.

To send a `Server` header (or any custom header), set it **before** `respond` , on that one request:

```
func handle(req as httpd.Request) {
    httpd.setHeader($req, "Server", "jennifer");
    httpd.respond($req, 200, "hi\n");
}
```

The pull loop has no middleware of its own, so to stamp a header onto **every** response use the `web` framework's `web.before` middleware, which runs before each handler:

```
func stampServer(ctx as web.Context) {
    web.setHeader($ctx, "Server", "jennifer");
    return true;
}
$app = web.before($app, stampServer);
```

(`web.setHeader` is the framework wrapper over `httpd.setHeader` ; a bare `httpd` program repeats the per-request call, or factors it into a small helper the handlers call.)

Graceful shutdown

`httpd.shutdown` closes the listener, wakes any workers blocked in `httpd.accept` (they get an error so their loops can exit), and lets in-flight requests finish before returning. A typical server installs a signal handler (via `os`) that calls `shutdown` , or shuts down after a sentinel request.

Behind a reverse proxy (nginx)

In production an `httpd` / `web` app usually sits behind nginx, which terminates TLS, serves static assets, buffers slow clients, and can load balance. nginx speaks plain HTTP to the app over either a **TCP port** or a **Unix domain socket** - `httpd.listen` supports both.

TCP port. The app listens on a local port; nginx proxies to it:

```
def srv as httpd.Server init httpd.listen("127.0.0.1:8080");
```

```
server {
    listen 443 ssl;
    server_name app.example;
    location /static/ { root /srv/app; }        # nginx serves assets directly
    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Unix domain socket. No TCP port; nginx proxies over a socket file (cleaner permissions, a touch less overhead). The `unix:` prefix selects it, and a graceful `httpd.shutdown` unlinks the socket on the way out. If a

socket file lingers from a prior crash, `httpd.listen` clears it only after confirming it is stale (nothing is listening) - it never deletes a socket a live server is still using.

`httpd.listen`TLS floors the negotiated protocol at TLS 1.2 (TLS 1.0 / 1.1 are deprecated by RFC 8996).

```
def srv as httpd.Server init httpd.listen("unix:/run/app/app.sock");
```

```
upstream app { server unix:/run/app/app.sock; }

server {
    listen 443 ssl;
    server_name app.example;
    location / {
        proxy_pass http://app;
        proxy_set_header Host $host;
    }
}
```

Each process handles one request at a time (the pull loop is serial per accept loop - see Scope and limits), so for concurrency and multi-core use run several app processes on distinct ports or sockets behind one nginx `upstream {}` block.

Scope and limits

- **HTTP/1.1** over plaintext; **HTTP/2** is negotiated automatically over TLS by `net/http` .
- The request body is buffered with a **10 MiB cap** ; a body over the cap is rejected with **413 Request Entity Too Large** before it reaches the program (never silently truncated - a truncated body would defeat body-signature checks). A configurable limit is a planned follow-up.
- **Admission control.** At most 256 requests buffer a body / stay in flight at once; further connections wait for a slot, so buffered memory is bounded (~slots x 10 MiB) rather than growing with the connection count.
- **Must respond.** Every accepted request must be answered with `httpd.respond` (or `serveFile` / `serveDir`). A request left unanswered - e.g. the program threw between `accept` and `respond` - is answered **500** by the engine after a 60-second safety timeout, so the handler goroutine and client connection don't leak.
- **Routing, path parameters, middleware, cookies, and sessions** are not in the engine - they belong to the `web` framework module built on top of it, which does name-based handler dispatch itself (the engine never calls back into the interpreter). `web` owns the session **id cookie** ; the session **store** stays with the app, so the engine and `web` both stay storage-agnostic.

See also

- `http` - the HTTP/1.1 *client* module.
- `net` - the lower-level TCP / TLS / UDP primitives.
- `json` / `toml` - encode / decode request and response bodies.
- technical/tinygo.md - why `httpd` is default-binary-only.

intl - internationalization (message catalogs)

Enable with `use intl;` . Message catalogs and locale-aware translation: load a `map of string to string` per language, pick a locale, and translate keys with named-placeholder interpolation and an automatic fallback chain. The name is `intl` (letters only, matching JavaScript's `Intl`) because a Jennifer library namespace cannot contain a digit - so no `i18n` .

`intl` is a **system library** , not a `.j` module, for two independent reasons: it holds **global mutable state** (the loaded catalogs plus the current locale), which a declarations-only module cannot; and it keeps each catalog in a Go `map[string]string` for **O(1)** lookup, where a Jennifer `map` (a linear-scan `[]MapEntry`) would be O(n) on every translation. The `map of string to string` you pass to `intl.load` is the one-time ingest; the per-lookup scan is what the library avoids.

Surface

Call	Returns	Notes
intl.load(lang, catalog)	null	Merge a map of string to string into the catalog for lang. The first language loaded becomes the default.
intl.setLocale(lang)	null	Set the current locale (e.g. "de" or "de-AT").
intl.locale()	string	The current locale ("" until set).
intl.tr(key)	string	Translate key in the current locale, with fallback.
intl.tr(key, params)	string	Same, filling %name% placeholders from params (a map).

Loading catalogs

A catalog is a `map of string to string` - a key to a message template. It can be a literal, or built from a config file decoded with `json` , `toml` , or `yaml` . Load one per language; loading the same language again **merges** (later keys win):

```
use intl;

intl.load("en", {
    "greeting": "Hello, %name%!",
    "cart": "You have %n% items in your cart"
});
intl.load("de", { "greeting": "Hallo, %name%!" });
```

The **first** language loaded is the *default* (source) language - the last stop in the fallback chain before the key itself, so make it your most complete catalog.

From .yml files on disk

In practice each language lives in its own file. Keep them as flat `key: value` YAML and read each with `fs` + `yaml` , bridging the decoded `yaml.Value` to the `map of string to string` `intl.load` wants - `yaml.keys` walks the top-level keys, `yaml.asString` pulls each value:

```
# locales/en.yml
greeting: "Hello, %name%!"
cart: "You have %n% items in your cart"
bye: "Goodbye"
```

```
# locales/de.yml
greeting: "Hallo, %name%!"
bye: "Auf Wiedersehen"
```

```
use fs;
use yaml;
use intl;

# Read a flat "key: value" YAML catalog into a map of string to string.
func loadCatalog(path as string) {
    def doc as yaml.Value init yaml.decode(fs.readString($path));
    def out as map of string to string init {};
    for (def key in yaml.keys($doc)) {
        $out[$key] = yaml.asString($doc, "/" + $key);
    }
    return $out;
}

intl.load("en", loadCatalog("locales/en.yml"));   # the default language
intl.load("de", loadCatalog("locales/de.yml"));
intl.setLocale("de");

intl.tr("greeting", {"name": "Welt"})   # "Hallo, Welt!"
intl.tr("cart", {"n": 3})               # "You have 3 items ..." (falls back to
en)
```

The files are flat on purpose: `yaml.asString` reads a string leaf, so a nested YAML value would need flattening into dotted keys first (`menu.file`), and a non-string value (an unquoted number) would need `convert.toString` . The same shape works with `json` (`json.keys` / `json.asString`) or `toml` if you prefer those formats - `examples/intl_toml.j` is a runnable version that writes `en.toml` / `de.toml` , reads them back, and loads them with `toml.keys` / `toml.asString` :

```
func loadCatalog(path as string) {
    def doc as toml.Value init toml.decode(fs.readString($path));
    def out as map of string to string init {};
    for (def key in toml.keys($doc)) {
        $out[$key] = toml.asString($doc, "/" + $key);
    }
    return $out;
}
intl.load("en", loadCatalog("locales/en.toml"));
```

Translating

`intl.tr(key)` resolves the key through a fallback chain, so a missing translation degrades gracefully and is always **visible** rather than blank:

1. the **current locale** exactly (`"de-AT"`);
2. its **base language** (`"de-AT"` -> `"de"` , region stripped);
3. the **default** (first-loaded) language;
4. the **key itself** .

```
intl.setLocale("de-AT");
intl.tr("greeting", {"name": "Welt"})   # "Servus, Welt!" if de-AT has it,
                                        # else "Hallo, Welt!" from de,
                                        # else the en default, else "greeting"
```

Placeholders

With the two-argument form, each %name% in the template is replaced by `params["name"]` . A non-string value is rendered to its display form (so a number interpolates without a manual `convert.toString`). An unknown

placeholder is left **literal** (a missing value is visible), and %% is the escape for a literal % :

```
intl.tr("cart", {"n": 3})               # "You have 3 items in your cart"
intl.tr("greeting", {"name": "Welt"})   # "Hallo, Welt!"
# template "raw %missing% and %%brace%%" -> "raw %missing% and %brace%"
```

The marker is %name% , not {name} , so a template reads the same whether it lives in a cooked "..." or a raw '...' string: the language reserves {} for string interpolation and \ for escapes, while intl owns % . The two never collide.

Output cap (untrusted catalogs)

A catalog from an untrusted source could carry an amplification bomb - a template that repeats a placeholder many times ("%x%%x%%x%..."), so a small template plus a modest param blows up into a huge string. To keep that from driving the process toward OOM, one interpolated translation is **capped at 1 MiB of output** , checked incrementally so intl.tr **errors before** the oversized string is ever built (the error is catchable with try / catch). A genuine translation is a UI message far below the cap; a document that large is a templating layer's job, not a catalog entry. Substitution is also single-pass - a param whose value itself contains %other% is inserted verbatim, never re-expanded - so there is no recursive-expansion blow-up either.

Why intl.tr, not a global _()

There is no gettext-style ambient _("key") . _ is not a valid Jennifer method name (identifiers are letters-only; _ is reserved for constant-name separators), and a bare tr() builtin does not clear the bar for a language built-in (translation is not useful to nearly every program, the way len is). An ambient global is also exactly what the **explicit** and **namespaced, no globals** stances rule out - so translation is a namespaced call, intl.tr("key") (or use intl as t; t.tr("key")). Translation is *content substitution from stateful external data* , distinct from printf 's job of *presenting the value in hand* , which is why it is not a printf extension.

Not yet here

Pluralization (CLDR per-language plural rules), an intl.loadFile(path) convenience, and locale-aware **value** formatting (number / date grouping - a separate, open printf question) are follow-ons. intl.load is also **merge-only** today - there is no reset or replace, so a key removed from a reloaded catalog lingers; a reset / replace-semantics verb is a follow-on for the hot-reload use case.

See also

- json / toml / yaml - config formats a catalog can be decoded from.
- convert - convert.toString if you would rather stringify a placeholder value yourself.

io - input/output

Enable with `use io;` . It provides formatted output - `printf` (stdout), `eprintf` (stderr), and `sprintf` (to a string), which share a Go-style format-string mini-language - and stdin input: `readLine` / `readLines` / `eof` for lines, `readBytes` / `readChars` for binary.

printf

Writes formatted output to standard output. Three calling shapes:

```
io.printf("hi\n");                                # literal string (no verbs)
io.printf($x);                                    # single non-string value,
displayed
io.printf("you are %d years old!\n", $age);      # format string + arguments
io.printf("%s = %d, ok=%t\n", "answer", 42, true);
```

> **Never pass a dynamic string as the format argument.** A single **string** argument is always the format string, so `io.printf(s)` scans `s` for verbs. That is safe only for a string *you* wrote. For any value you did not author - a generated password, user input, file contents - use `io.printf("%s", s)` (or `"%s\n"`). A stray `%` in the data would otherwise be read as a verb: `%c` fails as an unknown verb, `%s` as a missing argument, so `io.printf(password)` is a latent bug. This is deliberate - there is no separate verbatim-print builtin (see technical/rejected.md); `printf("%s", s)` is the one canonical way to print a dynamic string.

eprintf

Exactly like `printf` - same argument forms and verbs - but writes to **standard error** instead of standard output. Use it for diagnostics, warnings, and logs that must not mix into a program's stdout data (so a pipeline consuming the stdout stays clean).

```
io.eprintf("warning: retrying (%d/%d)\n", $attempt, $max);   # goes to stderr
```

sprintf

Same arguments as `printf` but **returns** the formatted string instead of writing it.

```
def msg as string init io.sprintf("%d + %d = %d", 1, 2, 3);
io.printf("%s\n", $msg);   # "1 + 2 = 3"
```

Format verbs

Verb	Required kind	Notes
%d	int	decimal
%f	float	shortest round-trip
%s	string	raw
%t	bool	true / false
%v	any	uses the value's display form
%%	-	literal %

Mismatches (wrong verb for the value kind, too few or too many args, dangling `%` , unknown verb) all produce runtime errors.

Escaping the meta-characters:

- A literal `%` in any string passed to `printf` / `sprintf` must be doubled to `%%` .
- A literal `|` *immediately after a verb* must be doubled to `||` , because `|` otherwise starts a modifier list (see Format modifiers). The escape consumes one of the two `|` s; the other appears in the output. Pipes that don't touch a verb are normal characters and need no escaping: `io.printf("a|b %s||c|d\n", "X")`

prints a|b X|c|d - the || after %s is the escape, while the | s in a|b and c|d sit between non-verb characters and pass through unchanged.

Format modifiers

Each verb (except %v) accepts an optional pipe-separated modifier list:

```
%verb[|key=value]*
```

Modifiers are **order-independent flags** - %d|pad=5|fill=0|align=right and %d|align=right|fill=0|pad=5 produce the same output. The list runs left-to-right until it hits a byte that isn't part of a key or value. To put a literal | immediately after a verb, double it: || writes one | and ends the modifier list (same shape as %%). Unknown keys, bad values, missing companions (e.g. group= without sep=), and the same key set twice are all runtime errors.

Evaluation order within one verb:

1. **Null check.** If the value is null *and* the spec includes a null= modifier, the verb-specific render is skipped and the configured replacement is used.
2. **Verb-specific render.** mode , base , prec , sci , sign , group / sep , case apply here.
3. **Layout.** max truncates (rune-aware), then pad + fill + align extends. Layout still applies to the null replacement, so columns line up.

null= (shared by %s, %d, %f, %t)

Form	Output when value is null
null=empty	""
null=null	"null"
null=literal("X")	X - the quoted text, with Jennifer string escapes parsed

Without a null= modifier, a null value is still a type-mismatch error against any verb except %v . null= wins over every other modifier on its verb: %s|mode=quote|null=literal("X") on a null prints X , not "X" .

%s modifiers

Key	Values	Default	Effect
pad	non-negative integer	-	minimum rune width
max	non-negative integer	-	truncate to N runes
align	left, right, center	left	which side gets the pad spaces; center splits the pad evenly (odd leftover goes right)
mode	raw, quote, escape	raw	wrap in "..." (quote) / show escapes (escape)
null	see above	-	substitute when value is null

mode=quote wraps the string in double quotes and escapes interior \ , " , and control bytes. mode=escape does the same escaping without the wrapping - useful for showing a string's structure in debug output.

```
io.printf("[%s|pad=8|align=right]\n", "hi");        # [      hi]
io.printf("[%s|max=3]\n", "abcdef");                # [abc]
io.printf("%s|mode=quote\n", "a\nb");               # "a\nb"
```

%d modifiers

Key	Values	Default	Effect
pad	non-negative integer	-	minimum width
fill	0	space	zero-pad between sign and digits; requires align=right (the default)

align	left, right	right	which side gets the pad
base	2, 8, 10, 16	10	digit base; hex uses lowercase
sign	negative, always, space	negative	sign for non-negative values
group	positive integer	-	digit-group size, reading right-to-left
sep	one of _, ,, ., -, :	-	group separator; required with group= and vice versa
null	see above	-	substitute when value is null

```
io.printf("%d|base=2\n", 5);                                  # 101
io.printf("%d|base=16|group=4|sep=_\n", 3735928559);         # dead_beef
io.printf("%d|pad=5|fill=0|sign=always\n", 42);              # +0042
```

%f modifiers

Key	Values	Default	Effect
prec	non-negative integer	shortest	fraction digits (or mantissa fraction digits when sci=true)
trim	true, false	false	strip trailing fraction zeros and the . if all zero
sci	true, false	false	force scientific notation (1.23e+03)
pad	non-negative integer	-	minimum width
align	left, right	right	which side gets the pad
sign	negative, always, space	negative	sign for non-negative values
null	see above	-	substitute when value is null

```
io.printf("%f|prec=2\n", 3.14159);              # 3.14
io.printf("%f|prec=4|trim=true\n", 3.0);        # 3
io.printf("%f|sci=true|prec=2\n", 0.00123);     # 1.23e-03
```

%t modifiers

Key	Values	Default	Effect
case	lower, upper, title	lower	true/false, TRUE/FALSE, True/False
null	see above	-	substitute when value is null

%v modifiers

%v takes no modifiers - it is deliberately the "I don't care, just print it" verb. Use a typed verb plus modifiers when you want to shape the output.

%a modifiers

%a is the aggregate verb: it renders a list or map in literal-like shape, recursing into nested aggregates. Non-collection input is a runtime error.

Key	Values	Default	Effect
sep	quoted string	", "	element separator (between list items, between map entries)
kv	quoted string	": "	key/value separator for map entries
open	quoted string	[(list) / { (map)	opening bracket
close	quoted string	] (list) / } (map)	closing bracket
depth	non-negative integer	unlimited	max recursion depth; deeper levels collapse to [...] / {...} (depth=0 collapses at the top)
null	skip	-	omit null list elements and null map values

Modifier values can be **double-quoted** to include spaces, brackets, or other characters reserved by the modifier-list grammar. The escape set is the standard `\n \r \t \\ \"` :

```
def xs as list of int init [1, 2, 3];
io.printf("%a\n", $xs);                              # [1, 2, 3]
io.printf("%a|sep=\" | \"\n", $xs);                  # 1 | 2 | 3 (with brackets)
io.printf("%a|open=\"<\"|close=\">\"\n", $xs);       # <1, 2, 3>

def grid as list of list of int init [[1, 2], [3, 4]];
io.printf("%a\n", $grid);            # [[1, 2], [3, 4]]
io.printf("%a|depth=1\n", $grid);    # [[...], [...]]

def m as map of string to int init {"a": 1, "b": 2};
io.printf("%a\n", $m);                         # {a: 1, b: 2}
io.printf("%a|kv=\"=\"|sep=\" \"\n", $m);    # {a=1 b=2}
```

Per-element rendering uses the same display form as `%v` , so primitive values inside an aggregate look the way they would in a print statement. `null=skip` is only valid on `%a` - it omits null elements entirely; the other `null=` modes (`empty` , `null` , `literal`) are rejected for `%a` because they don't have a sensible per-element meaning.

Input from stdin

Three builtins for reading lines from standard input. They are **output-symmetric** with `printf` / `sprintf` and intentionally minimal - line at a time, with an explicit end-of-input predicate so nothing happens implicitly.

io.readLine() -> string

Read one line from stdin. The trailing `\r\n` or `\n` is stripped; the returned string never carries a newline. Calling at end-of-input is a positioned runtime error (`readLine: end of input`), so the caller must check `io.eof()` first.

A final line that has no trailing newline is returned normally on the call that reaches it; the subsequent call errors.

io.readLine(prompt) -> string

Same as `io.readLine()` but writes `prompt` to stdout (no newline added) before reading. The prompt is written **unconditionally** , even when stdin is piped - explicit beats silently skipping the prompt off a non-TTY.

```
def name as string init io.readLine("name: ");
io.printf("hi, %s\n", $name);
```

io.eof() -> bool

True if and only if the next `io.readLine()` (or `io.readBytes` / `io.readChars`) would return less than requested. Implemented by peeking one byte through a buffered reader; the byte stays in the buffer for the next read. Once true, `io.eof()` stays true for the rest of the run.

io.readBytes(n) -> bytes

Reads exactly n bytes from stdin and returns them as a `bytes` value. If EOF is hit before n bytes are available, returns the partial result and `io.eof()` becomes true on the next call. n must be a non-negative `int` .

```
use io;
def first as bytes init io.readBytes(8);   # exactly 8 bytes or less at EOF
io.printf("got %d bytes\n", len($first));
```

io.readLines() -> list of string

Reads every remaining line from stdin and returns them as a `list of string` , each with its trailing newline stripped - the slurp counterpart to the streaming `readLine()` + `eof()` loop, for tiny filter scripts over bounded

input. Line splitting is **OS-independent** : it breaks on `\n` and strips a `\r` only when it precedes that `\n` (a CRLF terminator), so LF and CRLF input read the same on any platform, while a bare `\r` that is not part of a CRLF is kept as data (exactly like `readLine`). Empty stdin yields an empty list; a blank line is preserved as `""` ; a final line with no trailing newline is included.

It reads the **whole input into memory with no cap** , at roughly **1 KB of heap per line** (one value per line), so a few hundred thousand lines is already hundreds of MB - a ~20 MB line-oriented file is closer to ~2 GB of heap. Budget `readLines()` for small inputs (low single-digit MB); for large or untrusted streams keep the streaming `readLine()` loop, which stays flat in memory. Shares one buffered stdin reader with `readLine` / `readBytes` / `readChars` , so it returns only the lines not already consumed - and, like them, must not be called from concurrent `spawn` tasks (the shared reader is unsynchronized). Refused inside the REPL.

```
use io;
def lines as list of string init io.readLines();
io.printf("read %d line(s)\n", len($lines));
```

io.readChars(n) -> string

Reads exactly n Unicode code points from stdin, decoded from UTF-8, and returns them as a string. Same EOF behavior as `readBytes` (partial result, sticky `io.eof()`). n is a rune count, not a byte count - one Unicode character can be 1-4 bytes wide.

```
use io;
def first as string init io.readChars(3);  # exactly 3 runes
io.printf("got %d runes (%d bytes)\n", len($first), 0);  # len = 3
```

Canonical loop

```
use io;
while (not io.eof()) {
    def line as string init io.readLine();
    io.printf("[%s]\n", $line);
}
```

This is the only pattern the language asks you to learn. There is no `for line in stdin` shortcut, no `lines()` that slurps the whole stream, and `io.readLine()` does not return a sentinel value at EOF - they were considered and rejected because the existing trio is already complete and adding parallels would violate Jennifer's "one way per thing" stance.

REPL limitation

The interactive REPL owns stdin via its line editor, so `readLine` and `eof` both refuse inside the REPL with a clear error:

```
readLine: stdin is owned by the REPL editor
```

A proper side-channel for REPL input is a future milestone. To play with the input functions today, put your program in a `.j` file and run it with stdin piped or redirected: `jennifer run prog.j < input.txt` .

Float display

Floats always display with a decimal point so the value's type stays visible: `5.0` prints as `"5.0"` , not `"5"` . That matters most after the Python 3 division change - `4 / 2` is the float `2.0` , and you can tell at a glance rather than wondering whether it's an `int` .

See also: ../user-guide/index.md, ../technical/interpreter.md, index.md.

json - JSON encode / decode

Enable with `use json;` . RFC 8259 JSON, mapped onto Jennifer's value model. Hand-rolled (no host `encoding/json`) so it works identically on both binaries.

Surface

Call	Returns	Notes
json.encode(v)	string	Compact JSON for any encodable value (including a json.Value).
json.encodePretty(v)	string	Same, 2-space indented; empty lists / maps stay []/{}.
json.decode(s)	json.Value	Parse JSON text into an opaque handle (see below).

`json.decode` rejects container nesting past a fixed depth with a normal (catchable) decode error, so hostile deeply-nested input can't exhaust the interpreter's stack. The limit is shared by every parser in the toolchain and is set per binary: 1000 levels on the default `jennifer` (Go's growable stack), 64 on `jennifer-tiny` (its fixed 4 MB stack). Both are far past any real document.

Accessors over a decoded `json.Value` . Every one takes an optional trailing **JSON Pointer** (RFC 6901) string, relative to the passed node (`""` or omitted = the node itself):

Call	Returns	Notes
json.typeOf(v[, ptr])	string	The node's type: null bool int float string list map.
json.get(v[, ptr])	json.Value	The addressed sub-node, as a json.Value (walk stays opaque).
json.has(v, ptr)	bool	Whether the pointer resolves to an existing node.
json.keys(v[, ptr])	list of string	Keys of the addressed map, in document order.
json.length(v[, ptr])	int	Element count of a list, or entry count of a map.
json.asInt(v[, ptr])	int	A number node with no fractional part. A float node errors.
json.asFloat(v[, ptr])	float	A number node; an integral one promotes (JSON has one number type).
json.asString(v[, ptr])	string	A string node.
json.asBool(v[, ptr])	bool	A true / false node.
json.isNull(v[, ptr])	bool	Whether the addressed node is JSON null.

Write surface - every verb is **non-mutating** , returning a fresh `json.Value` ; the idiom is `$v = json.set($v, ...)` :

Call	Returns	Notes
json.map()	json.Value	A fresh empty JSON map (object) - the start of a document.
json.list()	json.Value	A fresh empty JSON list (array).
json.set(v, ptr, val)	json.Value	Upsert a map key, or replace an in-range list index, at ptr.
json.insert(v, ptr, val)	json.Value	Insert val into a list before index ptr (or - = at end).
json.append(v, ptr, val)	json.Value	Push val onto the list addressed by ptr (sugar for insert at /.../-).
json.remove(v, ptr)	json.Value	Drop the map key or list element at ptr.
json.move(v, from, to)	json.Value	Relocate the subtree at from to to (read, remove, then set).

Encoding

`json.encode` walks the value and writes its JSON image:

Jennifer	JSON
null	null
bool	true / false
int	integer number
float	number (always with a . or exponent, so it decodes back as float)
string	string (escaped)
bytes	base64 string
list of T	array
map of string to V	object (in insertion order)
struct	object, keys in field-declaration order
json.Value	the document it wraps (a decode / encode round-trip)

Encode errors (positioned at the call): a `map` with a non-string key, a `task` value, or a non-finite `float` (`NaN` / infinity have no JSON image).

HTML-safe strings. In string values the characters < , > , and & , and the line / paragraph separators U+2028 / U+2029, are escaped as `\u003c` , `\u003e` , `\u0026` , `\u2028` , and `\u2029` (matching Go's default `SetEscapeHTML(true)`). This keeps encoded JSON safe to embed directly in an HTML `<script>` block or a JSONP response - a value of `</script>` cannot break out of the element, and the two separators cannot terminate a JavaScript string literal. The output is still valid JSON and decodes back to the exact same string, so a decode / encode round-trip is unaffected.

```
use io;
use json;

io.printf("%s\n", json.encode({"id": 1, "tags": ["a", "b"], "ok": true}));
# {"id":1,"tags":["a","b"],"ok":true}
```

Decoding

`json.decode` returns an opaque **json.Value** - a handle onto the parsed tree. It is deliberately opaque: operators, `[index]` , and `.field` all reject it (with a hint to the accessors), so you never mix a still-generic JSON node into typed code by accident. You reach inside with the accessors, addressing nodes by **JSON Pointer** - the same paths the (planned) write surface uses, so reads and writes are mirror images.

```
use io;
use json;

def doc as json.Value init json.decode("{\"x\": 7, \"y\": 8}");
io.printf("%d\n", json.asInt($doc, "/x") + json.asInt($doc, "/y"));   # 15
```

Under the hood a JSON object is a `map of string to V` , an array a `list of V` , and a number an `int` when it has no fractional or exponent part (else a `float`) - which is why `json.typeOf` reports `map` / `list` / `int` / `float` , Jennifer's own vocabulary, not "object" / "array" / "number". `convert.typeOf($doc)` is the generic `"object"` ; `convert.objectType($doc)` is the specific `"json.Value"` .

A `def r as json.Value;` with no initializer is a JSON `null` node (`json.isNull($r)` is `true`).

A `json.Value` **displays as its compact JSON** , so echoing `$doc` at the REPL, `io.printf("%v", $doc)` , and `convert.toString($doc)` all show the document (not an opaque `<json.Value>`); `json.encodePretty($doc)` is the multi-line form.

JSON Pointer (RFC 6901)

A pointer is a slash-separated list of keys and list indices, addressing **exactly one** node:

- `""` (or an omitted argument) is the node itself.
- `/user/name` walks map key `user` , then key `name` .
- `/user/roles/0` walks to a list and takes index `0` . Indices are `0` or `[1-9][0-9]*` - no leading zeros, no negative indices. The `-` end-marker is a write-only position (`insert` / `append`); reads reject it.
- Escapes: a literal `/` in a key is written `~1` , a literal `~` is `~0` (so key `a/b` is `/a~1b` , key `m~n` is `/m~0n`).

Pointers are relative to the node you pass, so `json.get` to a subtree and short relative pointers compose:

```
def user as json.Value init json.get($doc, "/user");
io.printf("%s\n", json.asString($user, "/name"));   # relative to $user
```

A missing key, an out-of-range or malformed index, or descending into a scalar is a positioned error (catchable with `try` / `catch`); use `json.has` to test a path without raising. There is no wildcard, recursive-descent, or predicate syntax - a pointer names one node, which is what lets the write verbs target it. (Query-style selection over many nodes is a separate, deliberately unbuilt concern.)

To read the **last** element of a list, compute the index from `json.length` - a plain pointer, no special syntax:

```
use convert;
def last as int init json.length($doc, "/qux") - 1;
io.printf("%d\n", json.asInt($doc, "/qux/" + convert.toString($last)));
```

Walking a nested document

```
use lists;

def doc as json.Value init json.decode(
    "{\"user\": {\"name\": \"ada\", \"roles\": [\"admin\", \"dev\"]}}");

io.printf("name = %s\n", json.asString($doc, "/user/name"));
for (def i in lists.range(0, json.length($doc, "/user/roles"))) {
    io.printf("role = %s\n", json.asString($doc, "/user/roles/" + convert.
toString($i)));
}
```

`json.typeOf` and `json.has` let you branch on shape before extracting - handy when a field may be absent or polymorphic:

```
if (json.has($doc, "/count") and json.typeOf($doc, "/count") == "int") {
    def n as int init json.asInt($doc, "/count");
}
```

Building and editing

The write verbs mirror the reads: same JSON Pointer addressing, but they **return a new json.Value** rather than mutating in place (like `lists` / `maps`), so you rebind the result:

```
def v as json.Value init json.map();          # {}
$v = json.set($v, "/name", "ada");            # {"name":"ada"}
$v = json.set($v, "/tags", json.list());      # {"name":"ada","tags":[]}
$v = json.append($v, "/tags", "admin");       # ...,"tags":["admin"]
$v = json.insert($v, "/tags/0", "root");      # ...,"tags":["root","admin"]
$v = json.set($v, "/tags/1", "dev");          # replace index 1
$v = json.remove($v, "/name");                # drop a key
$v = json.move($v, "/tags", "/roles");        # rename a branch
```

The value you store may be a scalar, a Jennifer `list` / `map` / struct (a struct normalizes to a map, `bytes` to a

base64 string), or another `json.Value` (its tree is spliced in). Since writes never mutate, an earlier handle keeps its old value:

```
def a as json.Value init json.decode("{\"n\":1}");
def b as json.Value init json.set($a, "/n", 2);
# json.asInt($a, "/n") is 1; json.asInt($b, "/n") is 2
```

Writes are **strict - no auto-vivification.** `set` creates only the final pointer segment; a missing intermediate is an error, as is a `set` / `insert` on a scalar or a bare `null` root:

```
def v as json.Value;                          # a null node
# json.set($v, "/a", 1)          -> error: cannot set a member of null
# json.set(json.map(), "/a/b", 1) -> error: no key "a" (build /a first)
```

So you build a nested document a level at a time, each parent created before its child - the same discipline XML forces (you make an element before you hang things on it), which is why the model ports cleanly. `set` grows a map (new key) but only *replaces* an in-range list slot; grow a list with `append` / `insert` . There is no deep-path creation and no negative indexing - both deliberately, to keep pointers exactly RFC 6901 and unambiguous about list-vs-map intent.

The `-` end-of-array marker (RFC 6901's "position after the last element") is honoured by **insert / append only** , where it means *at the end* - so `json.insert($v, "/xs/-", x)` and `json.append($v, "/xs", x)` are the same push. It is a write-position, not an element: `set` rejects `/xs/-` (it never grows a list), and reads reject it too (there is no node there - the last element is `length - 1`).

Heterogeneous data

A JSON array or object whose values are of mixed types has no single Jennifer type - a `list of T` / `map of string to V` is homogeneous. The opaque `json.Value` sidesteps that: each leaf is extracted one at a time with a type-checked `as*` , so a document like `["Vienna", 2026, true, null, {...}, [1,2,3]]` is walked freely, element by element. This is the main practical win of the handle over decoding straight into a typed collection.

No map-to-struct coercion

Jennifer does no coercion at binding boundaries (you write `5.0` , not `5` , for a float), and JSON decode is no exception: there is no automatic map-to-struct landing. To put JSON in a typed struct, rebuild it explicitly from the decoded handle - the schema is right there:

```
def struct Point { x as int, y as int };
def d as json.Value init json.decode("{\"x\": 7, \"y\": 8}");
def p as Point init Point{ x: json.asInt($d, "/x"), y: json.asInt($d, "/y") };
```

The encode direction has no such restriction - `json.encode($p)` serializes a struct to an object directly, and `json.encode($d)` re-serializes the decoded handle. See technical/rejected.md for why the decode coercion was declined.

See also

- convert.md - `convert.typeOf` (`"object"`) and `convert.objectType` (`"json.Value"`) for identifying a handle.
- encoding.md - hex / base64 and character-set codecs (`json` reuses base64 for `bytes`).
- cheatsheet.md - every builtin at a glance.

kv - in-process key/value store

Enable with `use kv;` . An in-process key/value store with per-key **TTL** - the local, no-server counterpart to the `memcache` / `redis` clients. Open a store, then set / get / delete / touch / increment; values expire on their own. A store is a handle (`kv.Store`) into a per-interpreter registry, so a `kv.Store` value **shares its backing map across copies and across spawned tasks** - the shared mutable state a pure `.j` module cannot hold itself. This is what backs the in-process option of the `kvstore` backend selector (and so `session` / `ratelimit`). Pure Go stdlib, so it is on **both** binaries.

```
use kv;

def store as kv.Store init kv.open();
kv.set($store, "greeting", "hello", 60);   # expires in 60 s
io.printf("%s\n", kv.get($store, "greeting"));
```

Surface

Call	Returns	
kv.open()	kv.Store	a fresh, empty in-memory store (reset each run)
kv.openFile(path)	kv.Store	a store persisted to path - loaded on open, rewritten after every mutation, so it survives across jennifer run invocations
kv.set(store, key, value, ttl)		store value, expiring in ttl seconds (0 = never)
kv.add(store, key, value, ttl)	bool	store only if the key is absent; whether it stored
kv.get(store, key)	string	the value, or "" when absent / expired
kv.has(store, key)	bool	whether the key is present and unexpired (tells "" from absent)
kv.delete(store, key)	bool	remove the key; whether it existed
kv.touch(store, key, ttl)	bool	re-arm the key's expiry; whether it existed
kv.incr(store, key, delta)	int	add delta (signed - negative decrements, no separate decr) to the numeric value; the new value, or -1 when the key is absent (it is not created)
kv.close(store)		drop the store and free its handle

Semantics

- **TTL: lazy + periodic sweep.** An expired entry is evicted on the next access to that key, **and** a full sweep of expired entries runs periodically (every few hundred mutations). Without the sweep, a flood of distinct short-lived keys that are never accessed again (a rate limiter keyed by an untrusted IP) would pile up expired entries forever; the sweep bounds memory to the *unexpired* working set.
- **No hard size cap on live data.** There is no ceiling on *unexpired* entries - storing unbounded live data OOMs the process, exactly as an unbounded `list` would. That is the program's own responsibility. For an **adversarial or unbounded working set** , use `memcache` / `redis` (a server-side `maxmemory` eviction policy) via the `kvstore` selector; `close` frees the whole store.
- **incr mirrors memcache.** It does not create the key (returns `-1` when absent) and errors on a non-numeric value, so it composes into the same increment-then-add window pattern the distributed backends use. `delta` is **signed** - `kv.incr($s, $k, -1)` decrements, so there is no separate `decr` (one verb, both directions). Unlike memcached's DECR it does **not** floor at 0; clamp yourself if you need a non-negative counter.
- **Handles are shared, values are copies.** The `kv.Store` handle shares one backing map (that is the point - it survives value-copies and `spawn`); the string values it stores are ordinary copies.
- **Single process.** State lives in this process only. `open` is in-memory (gone when the program exits); `openFile` persists to disk and survives across runs (a rewrite-the-whole-file flush after every mutation - simple and correct, not fast). Both are safe for `spawn` ed tasks within one process (a per-store mutex; `incr`

is atomic), but **not** for concurrent *separate* processes on one file. For state shared across processes (multiple workers, a web fleet), use `memcache` / `redis` - the `kvstore` selector switches backends behind one API.

See also

- kvstore.md - the backend selector that puts `kv` , `memcache` , and `redis` behind one interface.
- memcache.md / redis.md - the distributed stores with the same verb shape.
- libraries/index.md - the library catalog.

linalg - linear algebra

The `linalg` library does linear algebra over Jennifer's own value types - the companion to `stats` . **Vectors** are a `list of float` and **matrices** a `list of list of float` , so the data is idiomatic, value-semantic, and interchangeable with the rest of the language (no opaque handle to construct or unwrap). It is pure-value and dependency-free (Go stdlib only, algorithms implemented directly - no `gonum`), so both binaries build it.

```
use linalg;
def a as list of list of float init [[4.0, 7.0], [2.0, 6.0]];
linalg.determinant($a);              # 10.0
linalg.inverse($a);                  # [[0.6, -0.7], [-0.2, 0.4]]
linalg.solve([[1.0, 1.0], [1.0, -1.0]], [3.0, 1.0]);  # [2.0, 1.0]  (x+y=3, x-y=1)
linalg.dot([1.0, 2.0, 3.0], [4.0, 5.0, 6.0]);         # 32.0
```

Vectors (list of float)

Call	Returns	Notes
linalg.dot(a, b)	float	Dot product. Vectors must be the same length.
linalg.distance(a, b)	float	Euclidean distance norm(a - b). Same length.
linalg.cross(a, b)	list of float	3-D cross product. Both vectors must have length 3.
linalg.normalize(v)	list of float	The unit vector v / norm(v). The zero vector errors.

Matrices (list of list of float)

Call	Returns	Notes
linalg.transpose(m)	list of list of float	Rows become columns.
linalg.trace(m)	float	Sum of the main-diagonal entries. Square only.
linalg.determinant(m)	float	Determinant via Gaussian elimination. Square only; a singular matrix is 0.
linalg.inverse(m)	list of list of float	Inverse via Gauss-Jordan. Square, non-singular - a singular matrix errors.
linalg.solve(a, b)	list of float	Solve a x = b for x. a square n x n, b length n; singular errors.
linalg.identity(n)	list of list of float	The n x n identity matrix (n >= 1).
linalg.zeros(rows, cols)	list of list of float	A rows x cols matrix of zeros (rows, cols >= 1).
linalg.shape(m)	list of int	[rows, cols] of a matrix.

Vector-or-matrix operations

Four operations are **polymorphic** - each takes either a vector or a matrix and returns the matching shape:

Call	Returns	Notes
linalg.norm(x)	float	Euclidean (L2) norm of a vector, or the Frobenius norm of a matrix (sqrt of the sum of squared entries).
linalg.scale(x, s)	same shape as x	Every element of a vector or matrix times the scalar s.
linalg.add(a, b)	same shape as the inputs	Element-wise sum of two vectors or two matrices (same shape).
linalg.sub(a, b)	same shape as the inputs	Element-wise difference of two vectors or two matrices.

`linalg.matmul(a, b)` is the general product, dispatched on the operand shapes:

Operands	Result	Meaning

matrix x matrix	list of list of float	Matrix product. The inner dimensions must match (m x n by n x p).
matrix x vector	list of float	M v, the vector v treated as a column. Matrix columns must match the vector length.
vector x matrix	list of float	v M, the vector v treated as a row. The vector length must match the matrix rows.
vector x vector	error	Ambiguous - use linalg.dot for the dot product.

A vector or matrix element may be an `int` or a `float` (an `int` promotes to `float`). A matrix must be **rectangular** - every row the same length - and non-empty; anything else is a positioned error.

Matrices are a plain nested list

There is no dedicated matrix type: a matrix is exactly a `list of list of float`, built and read with the language's own list syntax and value semantics. That keeps the data first-class (pass it to `stats`, iterate it, slice it) at the cost of the per-element boxing a Go-backed dense matrix would avoid - a fine trade at the sizes a scripting language handles. A Go-backed opaque matrix handle is the noted future escape hatch if big-matrix throughput ever demands it.

Strictness

Following `math` / `stats`, `linalg` never returns a `NaN` or `Inf`: a shape or dimension problem and a numerically undefined result are both catchable errors.

- A **dimension mismatch** (`dot` / `add` / `sub` / `distance` on unequal lengths, `matmul` on non-conforming inner dimensions, `solve` on a right-hand side that does not match the matrix), a **non-square** matrix where one is required (`determinant` / `inverse` / `solve`), a **non-rectangular** matrix, a `cross` on a non-3 vector, and an empty vector or matrix all raise a positioned error.
- A **singular** matrix has no inverse and no unique solution, so `inverse` and `solve` raise an error (`determinant` returns `0`, its correct value).
- When the input *magnitudes* overflow the computation - values near the float64 ceiling (`~1.8e308`) can push an intermediate product or sum to `+/-Inf` (and thence `NaN`) - the non-finite result is rejected rather than returned, so a `NaN` never escapes into the type system.
- A vector or matrix is bounded to a fixed **element count** (large but far below the point where materialising it would exhaust memory): a constructor called with an oversized dimension (`identity(100000)`, `zeros(100000, 100000)`) and an operation handed an oversized input both raise a positioned, catchable error rather than a fatal out-of-memory crash. This ceiling also keeps the `O(n^3)` routines (`matmul` / `inverse` / `solve`) bounded to a small, finite amount of work.

```
use linalg;
try {
    linalg.inverse([[1.0, 2.0], [2.0, 4.0]]);   # rows are linearly dependent
} catch (e) {
    io.printf("%v\n", $e.message);               # linalg.inverse: matrix is
singular (not invertible)
}
```

See also

`stats` for descriptive statistics over a `list`; `linalg` is its linear-algebra companion. Further ML primitives atop the two are on the horizon.

lists - list manipulation

Enable with `use lists;` . Namespaced under `lists.` ; every function is called as `lists.NAME(...)` . Each function **returns a new list** - nothing mutates the input. Commit the result with the usual assignment:

```
use io;
use lists;

def xs as list of int init [3, 1, 4, 1, 5];
$xs = lists.push($xs, 9);          # append item
$xs = lists.pop($xs);              # drop last
$xs = lists.sort($xs);             # sort ascending
io.printf("first=%d last=%d\n", lists.first($xs), lists.last($xs));
```

For the common "append to a list as you build it" pattern, the language ships the `$xs[] = item;` sugar (see user-guide/types-and-values.md). It's shorthand for `$xs = lists.push($xs, item);` **and, in a loop, much faster** : `$xs[]` mutates in place through the copy-on-write protocol (amortized O(N) for N appends), whereas `$xs = lists.push($xs, item)` returns a new list each pass and copies the whole thing - O(N^2) overall. Prefer `$xs[]` when building a list element by element; use `lists.push` when you specifically want a fresh list and keep the original.

Functions

Call	Returns	Notes
lists.push(xs, item)	list	New list with item appended. In a build-a-list loop prefer the $xs[] sugar (O(N) vs this call's O(N^2)).
lists.pop(xs)	list	New list without the last element. Empty input errors.
lists.first(xs)	element kind	Element at index 0. Empty input errors.
lists.last(xs)	element kind	Element at the last index. Empty input errors.
lists.head(xs, n)	list	New list of the first n elements. n must be in [0, len(xs)].
lists.tail(xs, n)	list	New list of the last n elements. Same range constraint.
lists.reverse(xs)	list	New list, elements in reverse order.
lists.sort(xs)	list	New list sorted ascending. See "Sort" below.
lists.contains(xs, item)	bool	True iff item appears in xs under structural equality.
lists.concat(a, b)	list	a's elements followed by b's.
lists.slice(xs, start)	list	Elements from start to end (exclusive end = len(xs)).
lists.slice(xs, start, end)	list	Elements [start, end). Out-of-range bounds error.
lists.shuffle(xs)	list	New list, elements in uniformly random order. See "Shuffle" below.
lists.range(start, end)	list of int	Half-open: [start, start+1, ..., end-1]. See "Range" below.
lists.range(start, end, step)	list of int	Walks by step while staying strictly before end. See "Range".
lists.map(xs, fn)	list	New list of fn(x) for each x. See "Higher-order functions".
lists.filter(xs, fn)	list	The elements where fn(x) returns true.
lists.reduce(xs, fn, init)	element kind	Fold left: acc = fn(acc, x), starting from init.
lists.find(xs, fn)	element kind	The first x where fn(x) is true; errors (catchable) if none.
lists.any(xs, fn)	bool	True if fn(x) is true for any element (short-circuits).
lists.all(xs, fn)	bool	True if fn(x) is true for every element (true for empty).
lists.sortBy(xs, keyFn)	list	New list sorted ascending by the key keyFn(x) returns.

Higher-order functions

`map` / `filter` / `reduce` / `find` / `any` / `all` / `sortBy` each take a first-class **func** value (see the language guide) and call it back per element. A bare method name in expression position is the function value, so you pass a method by name:

```
use lists;
func dbl(n as int) { return $n * 2; }
func isEven(n as int) { return $n % 2 == 0; }

def xs as list of int init [1, 2, 3, 4, 5];
lists.map($xs, dbl);           # [2, 4, 6, 8, 10]
lists.filter($xs, isEven);     # [2, 4]
lists.reduce($xs, add, 0);     # sum, with func add(a as int, b as int) {...}
lists.any($xs, isEven);        # true
lists.all($xs, isEven);        # false
```

- **map** returns a **generic** list - its element type is whatever the callback returns, checked against the declared type at the binding site, so `def names as list of string init lists.map($people, nameOf);` type-checks only when every result is a string.
- **filter** / **find** / **any** / **all** : the callback must return a `bool` , else a positioned error. `find` errors (catchably) when nothing matches
  - guard with `lists.any` first, or wrap it in `try` / `catch` .
- **sortBy** takes a **key extractor** `keyFn(x) -> K` (a comparable scalar: `int` / `float` / `string` / `bool`) and sorts ascending by that key, stably. The key is extracted once per element, so `keyFn` runs O(n) times, not O(n log n). Sort descending by negating a numeric key (`func neg(n as int) { return 0 - $n; }`) or reversing the result. Sort structs by a field with a one-line accessor (`func ageOf(p as Person) { return $p.age; }`).

Sort

`lists.sort` works on lists whose elements share a comparable kind:

- All elements `int` or `float` (mixed allowed - the comparison promotes int to float, same rule as `+`).
- All elements `string` - lexicographic order on the underlying rune sequence.
- All elements `bool` - `false` < `true` .

A list mixing strings with numbers, or containing `null` , `list` , or `map` elements, errors at runtime rather than silently producing nonsense. For a custom order, `lists.sortBy(xs, keyFn)` sorts by the key a `func` value extracts (see "Higher-order functions" above).

first/last versus head/tail

`first` and `last` return *elements* (the value at index 0 or `len-1`). `head` and `tail` return *sublists* of length n , modelled on the Unix `head` / `tail` commands. They are not aliases; pick the one that matches what you actually want:

```
def xs as list of int init [10, 20, 30, 40, 50];
lists.first($xs);          # 10              (an int)
lists.last($xs);           # 50              (an int)
lists.head($xs, 2);        # [10, 20]        (a list of int)
lists.tail($xs, 2);        # [40, 50]        (a list of int)
```

For "everything except the first/last element", use `slice` :

```
lists.slice($xs, 1);               # [20, 30, 40, 50]
lists.slice($xs, 0, len($xs) - 1);  # [10, 20, 30, 40]
```

Argument order

`lists.contains` puts the *haystack* first and the *needle* second (`lists.contains($xs, item)`). Mirrors `strings.contains($s, $sub)` . PHP's `in_array($needle, $haystack)` order is deliberately not adopted - it's famously confusing.

Shuffle

`lists.shuffle(xs)` returns a uniformly-random permutation of `xs` - non-mutating, like every other helper in this library. The algorithm is Durstenfeld's variant of the Fisher-Yates shuffle (O(n), uniform across the n! permutations). The random source is the same one `math.rand` / `math.randInt` / `math.randSeed` use, so calling `math.randSeed(N)` before a shuffle makes the result deterministic across runs:

```
use lists;
use math;

math.randSeed(1);
def a as list of int init lists.shuffle([1, 2, 3, 4, 5]);
math.randSeed(1);
def b as list of int init lists.shuffle([1, 2, 3, 4, 5]);
# $a and $b are byte-identical permutations.
```

Empty and single-element inputs are returned (still copied per the non-mutating convention). The function does NOT require `use math;` in the calling program - the shared source is a Go-side implementation detail.

Range

`lists.range(start, end)` returns a **half-open** list of consecutive integers: `[start, start+1, ..., end-1]` for ascending, descending implied when `start` > `end` . **End is exclusive.** Same semantic as `lists.slice` and `strings.substring` , and the same shape every half-open range function in the wider ecosystem has (Python `range` , Go slice indexing, Rust `..` , etc.). The full design rationale (including why we deviated from Jennifer's English-reading default) is in design-decisions.md > Half-open ranges.

```
use lists;

lists.range(0, 5);          # [0, 1, 2, 3, 4]    - 5 elements
lists.range(1, 5);          # [1, 2, 3, 4]       - 4 elements (5 excluded)
lists.range(5, 5);          # []                 - coincident bounds: empty
lists.range(3, 0);          # [3, 2, 1]          - descending implied
```

Two properties fall out:

- **Index alignment.** `lists.range(0, len($xs))` gives exactly the valid indices for a list of `len($xs)` elements, matching how `$xs[i]` indexing works.
- **Composability.** `lists.concat(lists.range(a, b), lists.range(b, c))` is exactly `lists.range(a, c)` . Partitioning a range never duplicates or skips the boundary.

For the "count from 1 to N inclusive" idiom, write `lists.range(1, N + 1)` .

`lists.range(start, end, step)` walks by `step` , always stopping strictly before `end` . Positive step requires `start` <= `end` ; negative step requires `start` >= `end` ; step must be non-zero (positional error).

```
lists.range(0, 9, 3);       # [0, 3, 6]          - 9 excluded
lists.range(1, 9, 3);       # [1, 4, 7]          - 10 past 9, stop at 7
lists.range(10, 1, -3);     # [10, 7, 4]         - 1 excluded
lists.range(10, 0, -3);     # [10, 7, 4, 1]      - -2 past 0, stop at 1
```

There's no "did the step land?" question - the rule is uniformly "emit while inside the open end."

Value semantics

Every function copies its inputs; the original list is never modified. Callers always re-bind the result:

```
def xs as list of int init [1, 2, 3];
def ys as list of int init lists.push($xs, 4);
# $xs is still [1, 2, 3]; $ys is [1, 2, 3, 4]
```

This matches the rest of Jennifer's value-semantics design - the same rule that makes `$dst = $src;` a copy, not an alias.

See also: maps.md, index.md. `len(xs)` is a language built-in (no import needed).

maps - map manipulation

Enable with `use maps;` . Namespaced under `maps.` ; every function is called as `maps.NAME(...)` . Each function **returns a new map (or list)** - the input is never mutated:

```
use io;
use maps;

def scores as map of string to int init {"alice": 90, "bob": 80};
def names as list of string init maps.keys($scores);   # ["alice", "bob"]
def values as list of int init maps.values($scores);   # [90, 80]
def shrunk as map of string to int init maps.delete($scores, "bob");
def with_dave as map of string to int init maps.merge($scores, {"dave": 75});
```

Functions

Call	Returns	Notes
maps.keys(m)	list of <key type>	The map's keys, in insertion order.
maps.values(m)	list of <value type>	The map's values, in insertion order.
maps.has(m, key)	bool	Key membership test. The non-erroring companion to $m[key].
maps.delete(m, key)	map	New map without key. Missing key is an error - see below.
maps.merge(a, b)	map	New map: a's entries with b's layered on top (b wins on collision).

maps.has

Reports whether the map contains the given key. Pair it with the indexed-read to avoid the missing-key runtime error:

```
def m as map of string to int init {"a": 1};
if (maps.has($m, "a")) {
    io.printf("%d\n", $m["a"]);   # safe - we just checked
}
```

This lived in core as bare `has(...)` ; it moved here because map-only membership is domain-specific and didn't fit core's "universally needed structural primitives" charter (`len` , by contrast, is genuinely polymorphic across string / list / map and stays). The lists side of "does this contain that?" is `lists.contains` , which checks values rather than keys - the differing names keep the role distinction visible at every call site.

maps.delete is strict

Deleting a key that isn't in the map raises a positioned runtime error - matching the read-side rule that `$m[missing]` is an error rather than a silent default. Callers who want a "delete if present" shape can guard with `maps.has` :

```
if (maps.has($m, "stale")) {
    $m = maps.delete($m, "stale");
}
```

This is "strict at boundaries" applied to writes: silent no-ops on missing keys would let typos drift through unnoticed.

maps.merge ordering

`merge(a, b)` returns a 's entries in a 's insertion order, with b 's values overwriting where keys collide. New keys from b are appended in b 's insertion order. So merging `{"x": 1, "y": 2}` with `{"y": 99, "z": 3}`

yields `{"x": 1, "y": 99, "z": 3}` - same shape Python's `{**a, **b}` produces.

Value semantics

The library never modifies its inputs. `maps.delete` and `maps.merge` both copy, so the source maps stay intact for further use; the new map is independent of either.

See also: lists.md, index.md. `len(m)` is a language built-in (no import needed).

math - numeric functions and constants

Enable with `use math;` . The numeric toolbox: the everyday functions (arithmetic helpers, trigonometry, hyperbolics, exponentials and logarithms, combinatorics) plus the special functions a probability layer is built on (the error, gamma and beta functions and their regularized incomplete forms), and the constants `math.PI` , `math.E` , and `math.TAU` . The library is strict on undefined inputs - anything that would produce `NaN` or `Infinity` in IEEE arithmetic instead produces a positioned runtime error, so a domain slip (`math.ln(0)` , `math.asin(2)`) is a catchable error, never a silent NaN.

```
use io;
use math;

io.printf("%f\n", math.PI);                    # 3.141592653589793
io.printf("%d\n", math.abs(0 - 42));           # 42
io.printf("%d\n", math.min(3, 7));             # 3
io.printf("%f\n", math.sqrt(2));               # 1.4142135623730951
io.printf("%f\n", math.pow(2, 10));            # 1024.0
io.printf("%d\n", math.floor(3.7));            # 3
io.printf("%d\n", math.ceil(3.2));             # 4
io.printf("%d\n", math.round(2.5));            # 3 (half away from zero)
```

Functions

Call	Returns	Notes
math.abs(x)	same type as x	|x|; int -> int, float -> float; errors on math.abs(MinInt64) (no representable result)
math.min(a, b)	int or float	int+int -> int; mixed -> float
math.max(a, b)	int or float	same rule as min
math.sqrt(x)	float	errors on negative input
math.pow(x, y)	float	errors if the result would be NaN or Infinity
math.floor(x)	int	toward -inf; accepts int (identity); errors if the result does not fit in a 64-bit int (NaN / Inf / out of range)
math.ceil(x)	int	toward +inf; same int-range error as floor
math.round(x)	int	half-away-from-zero (math.round(2.5) = 3); same int-range error as floor

`min` / `max` follow the same numeric-promotion rule as `+` : same-type arguments return that type; any `float` involved produces a `float` .

Trigonometry and hyperbolics

Angles are in **radians** . Every function returns a `float` ; the inverse functions error outside their domain (via the strict-NaN rule), and `sinh` / `cosh` error when they overflow to infinity.

Call	Returns	Notes
math.sin(x) / math.cos(x) / math.tan(x)	float	Basic trig.
math.asin(x) / math.acos(x)	float	Inverse; error outside [-1, 1].
math.atan(x)	float	Inverse tangent.
math.atan2(y, x)	float	Quadrant-aware arctangent of y/x.
math.sinh(x) / math.cosh(x) / math.tanh(x)	float	Hyperbolic; sinh/cosh error on overflow.
math.asinh(x)	float	Inverse hyperbolic sine (all reals).
math.acosh(x)	float	Inverse; error for x < 1.

math.atanh(x)	float	Inverse; error outside (-1, 1).

Exponentials and logarithms

Call	Returns	Notes
math.exp(x)	float	e^x; errors on overflow.
math.expm1(x)	float	e^x - 1, accurate for small x.
math.ln(x)	float	Natural log; errors for x <= 0.
math.log10(x)	float	Base-10 log; errors for x <= 0.
math.log2(x)	float	Base-2 log; errors for x <= 0.
math.log1p(x)	float	ln(1 + x), accurate for small x; errors for x <= -1.
math.log(x, base)	float	Log of x to an arbitrary base; errors for non-positive x / base or base 1.

Roots, magnitude, and sign

Call	Returns	Notes
math.cbrt(x)	float	Real cube root (handles negative x).
math.hypot(x, y)	float	sqrt(x*x + y*y) without intermediate overflow.
math.sign(x)	same type as x	-1 / 0 / 1 with the sign of x (int -> int, float -> float).
math.trunc(x)	int	Round toward zero; int argument is the identity.

Combinatorics

Integer in, integer out; a result past `int64` is a catchable overflow error (not a silent wrap), consistent with the language's integer arithmetic.

Call	Returns	Notes
math.factorial(n)	int	n!; errors on negative n or overflow (n > 20).
math.comb(n, k)	int	Binomial coefficient nCr (exact); k > n is 0; errors negative n/k.
math.perm(n, k)	int	k-permutations of n (nPr, exact); k > n is 0.
math.gcd(a, b)	int	Greatest common divisor (non-negative; gcd(0, 0) = 0).
math.lcm(a, b)	int	Least common multiple (lcm(x, 0) = 0); errors on overflow.

Special functions

The functions a statistics / probability layer needs. All return a `float` . The error and gamma functions come from Go's standard library; the **regularized incomplete** gamma and beta - the engine every distribution CDF is built on - are computed by the standard series / continued-fraction algorithms, verified to machine precision on their convergent domain. Their results are pinned into `[0, 1]` (a CDF by definition), and an extreme argument that drives the series to a non-finite or non-converged result is a catchable error, not a silently-wrong value or a leaked `NaN` - the same strict contract the rest of the library keeps.

Call	Returns	Notes
math.erf(x) / math.erfc(x)	float	Error function and its complement (erf + erfc = 1).
math.gamma(x)	float	Gamma function ?(x); errors at the poles (0, negative integers) and on overflow.
math.lgamma(x)	float	ln|?(x)|, for range; errors at the poles.
math.beta(a, b)	float	Beta function B(a, b) = ?(a)?(b) / ?(a+b); a, b > 0.
math.lbeta(a, b)	float	ln B(a, b), the stable log form; a, b > 0.

math.regGammaP(a, x)	float	Regularized lower incomplete gamma P(a, x) in [0, 1] (the gamma / chi-square CDF); a > 0, x >= 0.
math.regGammaQ(a, x)	float	Upper complement Q(a, x) = 1 - P(a, x).
math.regBetaI(x, a, b)	float	Regularized incomplete beta I_x(a, b) in [0, 1] (the CDF engine for the beta, Student's t, F, and binomial); 0 <= x <= 1, a, b > 0.

Randomness

Call	Returns	Notes
math.rand()	float	uniform in [0, 1)
math.randInt(lo, hi)	int	uniform in [lo, hi] inclusive; errors if lo > hi
math.randSeed(n)	null	reseeds the shared source deterministically

The library keeps one shared pseudo-random source. It is seeded from OS entropy at startup, so every run produces a different stream; call `math.randSeed(n)` when you need reproducible output (tests, demos, simulations). The same source drives `lists.shuffle` , so one `randSeed` call makes both deterministic together. (`uuid` and `password` draw from the `crypto` library's crypto-grade source instead, so `randSeed` does not affect them.)

The source is **not cryptographically secure** - it is Go's `math/rand` , seedable and predictable once observed. Don't derive secrets, tokens, keys, or anything security-sensitive from it.

```
use io;
use math;

math.randSeed(42);                              # reproducible from here on
io.printf("%f\n", math.rand());                 # same value every run
io.printf("%d\n", math.randInt(1, 6));          # die roll, seeded
```

For secure random numbers, use crypto

When the value must be unpredictable - a password, a token, a nonce, a shuffle an adversary must not guess - use the `crypto` library's crypto-grade source instead of `math` . `crypto.randInt(lo, hi)` has the **same shape** as `math.randInt` (inclusive `[lo, hi]`), so it is a drop-in replacement; it is unseedable by design (there is no `crypto.randSeed` , since predictability is the thing you are avoiding):

```
use io;
use crypto;

io.printf("%d\n", crypto.randInt(1, 6));        # secure, unbiased die roll
def token as bytes init crypto.randBytes(32);   # 256 bits of secure entropy
```

If you specifically want `math` 's fast, `lists.shuffle` -sharing stream but seeded **unpredictably** (so each run differs and the seed can't be guessed), draw the seed from `crypto` once at startup:

```
use math;
use crypto;

# a full-width unpredictable seed from the crypto-grade source
math.randSeed(crypto.randInt(0 - 9223372036854775807, 9223372036854775807));

# math.rand / math.randInt / lists.shuffle now run off an unguessable seed -
# fast and non-reproducible, but still NOT cryptographically secure output.
```

Note the trade-off: seeding `math` from `crypto` only makes the *starting point* unpredictable. The stream itself is still `math/rand` , so once enough output is observed it remains predictable - for output that must stay secret, use `crypto.randInt` / `crypto.randBytes` directly.

Constants

Name	Kind	Value
math.PI	float	3.141592653589793...
math.E	float	2.718281828459045...
math.TAU	float	6.283185307179586... (2*PI, the full-turn constant)

Constants are referenced through the `math.` namespace prefix like every other library name; the bare identifiers `PI` and `E` are not in scope. With `use math as m;` the alias takes over (`m.PI` , `m.E`).

Strictness

The library refuses to produce floating-point edge values:

- `math.sqrt(-1)` - undefined for negative input.
- `math.pow(0, -1)` - division-by-zero territory; result would be Infinity.
- `math.pow(-1, 0.5)` - would be NaN.

If a future use case needs the NaN/Infinity values, a `math.NAN` / `math.INF` constant (or dedicated check functions) can be added later. For now Jennifer treats them as errors at the boundary - consistent with how the language already refuses to silently coerce types.

See also: ../user-guide/index.md, ../technical/interpreter.md, index.md.

meta - interpreter identity and reflection

Enable with `use meta;` . Holds facts about the running Jennifer interpreter itself - the build version, which Go toolchain compiled it, and similar information that programs typically log for bug reports, embed in error messages, or branch on for build-specific behaviour - plus a small **reflection** surface for invoking a top-level method by a runtime string name (`meta.call` / `meta.defined` and their entry-program siblings `meta.callMain` / `meta.definedMain`).

This is distinct from `os` (which is about the *host environment* - operating system, CPU architecture, environment variables) and from the rest of the standard library (which is about user data).

```
use io;
use meta;

io.printf("Jennifer %s (%s build)\n",
    meta.VERSION, meta.BUILD);
```

Constants

Name	Kind	Value
meta.VERSION	string	The interpreter's build version. See format below.
meta.BUILD	string	Which Go toolchain compiled the interpreter: "go" or "tinygo".
meta.SYSMODDIR	string	The resolved system module directory (where bare imports look). Resolved from --sysmoddir > JENNIFER_SYSMODDIR > the compile-time default; jennifer version -v shows the layers.
meta.CAPABILITIES	list of string	The host capabilities this build was compiled with, sorted: ["exec", "net", "sql"] on the standard jennifer, [] on jennifer-tiny. The same set the # pragma-jennifer-capability: header is checked against.

Capability query

Signature	Returns	Description
meta.hasCapability(name)	bool	Whether this build includes the named host capability ("net" for TCP/UDP/TLS/DNS and the network-backed libraries, "exec" for os/exec external processes). Lets a script branch instead of failing in a stub.

```
use meta;
use io;
if (meta.hasCapability("net")) {
    io.printf("networking available\n");
}
io.printf("built with: %v\n", meta.CAPABILITIES);
```

A module can *require* a capability declaratively with a header pragma (`# pragma-jennifer-capability: net`), which refuses the module at read time on a build that lacks it - see imports.

VERSION string format

The build pipeline derives `meta.VERSION` from `git describe --tags --long` :

Repository state	meta.VERSION value
HEAD is exactly on a semver tag	"0.4.0"
HEAD is N commits past the most recent tag	"0.4.0-dev+2.1023204"
No tags exist yet	"0.0.0-dev+<N>.<shortsha>"
Built without git (or outside a repo)	"dev"

The dev+ prefix is intentional: any non-tagged build is a development build, and the N.shortsha suffix lets you trace which commit produced it.

BUILD values

meta.BUILD distinguishes which Go variant compiled the interpreter binary. Useful because TinyGo has subtly different runtime behaviour from standard Go (different GC, different scheduler tuning, different stdlib subset) - a program that needs build-specific behaviour or just wants to log "which interpreter is this for the bug report" can branch on this value.

Value	Meaning
"go"	Built with the standard Go toolchain (gc) - the default jennifer binary
"tinygo"	Built with TinyGo - the constrained jennifer-tiny binary

make build produces both binaries: the default jennifer (standard Go, meta.BUILD == "go") and jennifer-tiny (TinyGo, meta.BUILD == "tinygo"). make build-go and make build-tinygo produce only one each. go run against the source also reports "go" . If a future alternative compiler shows up, its identifier passes through directly rather than being normalised - so the constant always reports honestly what built the binary.

Reflection - calling a method by name

A func value binds to a specific method at parse time, and a bare call greet(...) is resolved at compile time, so neither dispatches on a name computed at runtime. meta.call closes that gap - it invokes a top-level user method by a runtime string, the general form of what testing.run does for tests.

Call	Returns	
meta.call(name, args...)	the method's return value	Invoke the method name with the given arguments (arity + declared types checked, as at a normal call site).
meta.defined(name)	bool	Whether a method name exists - validate a name before calling it.
meta.callMain(name, args...)	the method's return value	Like call, but resolves against the entry program's methods.
meta.definedMain(name)	bool	Like defined, against the entry program.

```
use io;
use meta;

func greet(name as string) { return "hi " + $name; }

io.printf("%s\n", meta.call("greet", "ada"));   # hi ada
io.printf("%t\n", meta.defined("nope"));        # false
```

Unlike testing.run , meta.call is transparent: it does not catch exit , and every sentinel a normal call can raise - a runtime error, a thrown Error , exit - propagates to the caller, catchable with try / catch .

Security: never dispatch on untrusted input

meta.call / meta.callMain invoke **any** top-level method by name, with no built-in restriction. Passing a name that comes from request data, a URL, a message body, or any other untrusted source is a remote-code-execution hole - it reaches deleteAllUsers , runMigration , or any internal helper you never meant to expose. meta.defined answers "does it exist", which is **not** an authorization check.

Always match untrusted input against a **program-defined allowlist** first, and call only a name that is on it:

```
use meta;
use lists;

# The only names an outside caller may reach.
def const ACTIONS as list of string init ["listItems", "showItem"];

func handleAction(name as string, arg as string) {
    if (not lists.contains(ACTIONS, $name)) {
        throw Error{kind: "app", message: "unknown action: " + $name, file: "",
line: 0, col: 0};
    }
    return meta.call($name, $arg);      # safe: $name is proven to be on the
allowlist
}
```

A dispatcher that maps a runtime name to a method must be careful: never let untrusted input (a request path, a wire field) name an arbitrary method. Gate on a name you control - a registered allow-list, or at least a `meta.defined` check that the target exists - and never dispatch a name you did not intend. Follow that discipline whenever you build dispatch on `meta.call` .

callMain / definedMain - reaching the entry program

Modules run on isolated sub-interpreters, so a `meta.call` *inside a module* reaches that module's own methods, not the program that imported it. The `*Main` variants cross that boundary: they resolve against the **entry program's** top-level methods. This is what lets a module dispatch to methods the application defined - the `jsonrpc` module resolves each request's wire method name against the entry program with `meta.callMain` (a cross-boundary `func` value, as in `web` , uses the same host reference to run in its home context). Called from the entry program itself, `callMain` / `definedMain` coincide with the plain forms (the entry program is its own host). Struct arguments are re-tagged across the boundary automatically, so a module can pass one of its own struct values (e.g. a `web.Context`) to an entry-program handler that declares it.

Build flow

`meta.VERSION` is set at build time by a small codegen step.

The Makefile runs `scripts/gen-version.sh` before `tinygo build` / `go build` , writing a generated `internal/version/version_gen.go` whose `init()` assigns `version.Version` to the string from `scripts/version.sh` . The `meta` library then mirrors that into the interpreter as the `meta.VERSION` constant.

You don't need to run the codegen step manually if you build via `make build` (TinyGo) or `make build-go` (Go). A bare `go test ./...` skips codegen and uses the default `"dev"` baked into `version.go` .

Codegen rather than `go build -ldflags -X` is used because TinyGo 0.41 silently ignores `-X` . The generated file is `.gitignore` d.

Roadmap

`meta` is a new library and intentionally small. Future candidates if they earn their slot: build time, git SHA (separated from the version string), REPL-vs-script mode, runtime GC stats, scheduler diagnostics. The library exists in part to give those a natural home when they land.

See also: os.md, index.md.

net - TCP + UDP sockets and DNS lookups

Enable with `use net;` . Blocking TCP and UDP sockets plus two DNS lookup helpers. Non-blocking use composes with `spawn` rather than a duplicated `*Async` surface.

```
use io;
use net;
use convert;

def c as net.Conn init net.connect("example.com:80");
net.writeBytes($c, convert.bytesFromString(
    "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n", "utf-8"));
def reply as bytes init net.readBytes($c, 4096);
net.close($c);
io.printf("%s\n", convert.stringFromBytes($reply, "utf-8"));
```

TCP

TCP is stream-oriented: writes go into a byte stream on one side; reads pull bytes off the other side in the order they arrived.

Call	Returns	Notes
net.connect(address[, timeoutMs])	net.Conn	TCP client. address is "host:port" (Go's convention). An optional timeoutMs bounds connection establishment (a slow / unreachable peer otherwise blocks the dial forever); 0 / omitted means no connect timeout.
net.listen(address)	net.Listener	Bind and listen. ":0" selects an ephemeral port.
net.accept($listener)	net.Conn	Blocking accept. Non-blocking use pairs with spawn.
net.readBytes($conn, n)	bytes	Blocks for at least one byte; returns whatever's available, capped at n. Sticky-EOF on close.
net.readAll($conn[, maxBytes[, idleTimeoutMs]])	bytes	Read until EOF and return the whole remaining stream as one bytes. maxBytes omitted or 0 uses the default 256 MiB cap (so a hostile endless stream is a catchable error, not an OOM); > 0 sets an explicit cap; a negative value is the explicit "no limit" opt-in. idleTimeoutMs > 0 re-arms a read deadline before each chunk (a timeout is a distinct catchable error). See Whole-stream reads.
net.readN($conn, n[, idleTimeoutMs])	bytes	Read exactly n bytes (a length-prefixed frame). A peer that closes before n bytes is a catchable "closed mid-frame" error, never a truncated return. idleTimeoutMs as in readAll.
net.writeBytes($conn, b)	null	Blocking write of every byte.
net.setDeadline($conn, ms)	null	Arm a read/write deadline ms milliseconds out; 0 clears it. A read past the deadline fails with a distinguishable read timed out error. Accepts a net.Conn or a net.UDPSocket (so a recvFrom can time out).
net.eof($conn)	bool	Looks ahead: true iff the next read would return partial or fail.
net.address($conn)	string	Peer's "host:port" (for logs). Polymorphic - see Address helpers.

net.Conn

```
def struct net.Conn { id as int };
```

Handles share underlying state between copies via the integer id (same discipline as `task of T` and `fs.File`). `net.close($c)` closes the connection for every copy of the handle.

Canonical read loop

```
use io;
use net;
use convert;

def c as net.Conn init net.connect("some.host:1234");
while (not net.eof($c)) {
    def chunk as bytes init net.readBytes($c, 4096);
    io.printf("got %d bytes\n", len($chunk));
}
net.close($c);
```

`net.eof` peeks one byte through the buffered reader so the loop terminates on the exact byte after the last real read.

Whole-stream reads

The canonical loop above reads and accumulates *one byte at a time* in Jennifer, which is fine for a small message but pays the interpreter's per-byte cost on a large body. When you want the whole thing in one `bytes` , `net.readAll` and `net.readN` do the accumulation in a single Go loop instead:

```
# The whole response body to EOF, in one call (an HTTP / object download):
def body as bytes init net.readAll($c, 0, 30000);   # default 256 MiB cap, 30s
idle timeout

# Exactly n bytes of a length-prefixed frame (n came off the wire first):
def frame as bytes init net.readN($c, $n);
```

`net.readAll` reads until the peer closes (or `maxBytes` is hit); `net.readN` reads a known count and treats an early close as an error rather than a short read. Both grow one Go slice, so a megabyte-scale transfer runs at native speed rather than a per-byte `net.readBytes` accumulation. To search or split the result once it is in hand, use `binary` (`indexOf` / `split` / `slice`).

With `idleTimeoutMs` > `0` both calls re-arm the connection's read deadline before each chunk, and restore the `net.setDeadline` state (no deadline when none was set) on return - the idle timeout never leaks into a later read on the same connection.

Deadlines: single-threaded poll with timeout

`net.setDeadline($conn, ms)` arms a read/write deadline `ms` milliseconds in the future; a read that reaches the deadline before data arrives fails with a distinguishable `net.readBytes: read timed out` error you can `catch` , rather than blocking forever or crashing. `net.setDeadline($conn, 0)` clears the deadline again. This turns a blocking read into a poll-with-timeout, so a protocol client can wait for a packet and, on a timeout, do idle work (send a keepalive) - all on one flow, without dedicating a `spawn` ed reader. The same call accepts a `net.UDPSocket` , so a datagram `recvFrom` can be bounded by a timeout the same way (an SNTP client that must not hang on a lost reply).

```
use net;

# Wait up to 1s for a packet; on timeout, send a keepalive and retry.
def running as bool init true;
while ($running) {
    net.setDeadline($c, 1000);
    try {
        def head as bytes init net.readBytes($c, 1);
        net.setDeadline($c, 0);     # clear while we read the rest of the packet
        # ... read and dispatch the rest of the message ...
    } catch (err) {
        if (strings.contains($err.message, "timed out")) {
            # idle - send a keepalive and loop
        } else {
            $running = false;       # a real failure (closed conn, etc.)
        }
    }
}
```

The deadline is absolute and is **not** rearmed automatically: reset it (or clear it with `0`) before the next read. It applies to writes too.

Server pattern

```
use net;
use task;

func handle(conn as net.Conn) {
    # ... read/write on $conn ...
    net.close($conn);
    return null;
}

def listener as net.Listener init net.listen(":8080");
while (true) {
    def conn as net.Conn init net.accept($listener);
    def worker as task of null init spawn { return handle($conn); };
    task.discard($worker);   # fire-and-forget per-connection worker
}
```

Under the default `jennifer` binary each `spawn` runs on its own OS thread and scales across cores; under `jennifer-tiny` (TinyGo) the server compiles but every net call surfaces the "not available" message - see TinyGo compatibility below.

TLS

Encrypted transport. Both entry points yield the **same net.Conn handle** as plaintext TCP, so `readBytes` / `writeBytes` / `close` / `address` work unchanged and callers stay transport-agnostic.

Call	Returns	Notes
net.connectTLS(address[, net.TLSOptions][, timeoutMs])	net.Conn	Dial address ("host:port", same as net.connect) and complete a TLS handshake (implicit TLS: 465 / 993 / 995, HTTPS). Optional trailing timeoutMs bounds the dial + handshake.

net.startTLS($conn[, net.TLSOptions][, timeoutMs])	net.Conn	Upgrade an open plaintext connection to TLS in place (STARTTLS: 587 / 143 / 110); same handle. Optional trailing timeoutMs bounds the handshake.

Both take an optional trailing `net.TLSOptions` argument. The certificate is verified against the connection's host: for `connectTLS` , the host in `address` (a missing port errors); for `startTLS` , the host the connection was originally opened with via `net.connect` - so you don't repeat it. TLS is built only into the default `jennifer` binary; `jennifer-tiny` returns the friendly no-network stub error.

net.TLSOptions

`net.TLSOptions { skipVerify as bool, caCert as bytes }` . Certificate verification is **on by default** - the zero value verifies against the system roots.

- **caCert** - a PEM certificate to trust, for a private CA or a self-signed server the system doesn't know. This is the **secure** way to reach such a server. An invalid PEM is a positioned error.
- **skipVerify: true** - accept *any* certificate (self-signed, expired, wrong-host). Development and testing only; a deliberate, greppable opt-out, never a default.

A struct literal names every field, so set both - or default-construct and assign only what you need (the zero value is the secure default):

```
use net;
use fs;

# secure: trust a self-signed / private-CA certificate
def o as net.TLSOptions;                        # skipVerify false, caCert empty
$o.caCert = fs.readBytes("server-ca.pem");
def c as net.Conn init net.connectTLS("localhost:8443", $o);

# insecure (dev / testing): skip verification entirely
def x as net.TLSOptions;
$x.skipVerify = true;
def d as net.Conn init net.connectTLS("localhost:8443", $x);
```

STARTTLS example

```
use net;
def c as net.Conn init net.connect("smtp.example.com:587");   # plaintext
# ... read the greeting, send EHLO, send STARTTLS, read the "220 ready" line ...
$c = net.startTLS($c);                                        # upgraded in
place, same handle
# ... continue the session, now encrypted (host reused from connect) ...
```

UDP

UDP is datagram-oriented: each `sendTo` / `recvFrom` is one packet with an associated peer address. There's no connection to establish; the socket is a bound port.

Call	Returns	Notes
net.listenUDP(address)	net.UDPSocket	Bind a UDP port. Usable as both client and server (send from wherever you bound).
net.sendTo($sock, peer, bytes)	null	Send one datagram to peer ("host:port").
net.recvFrom($sock, n)	net.Datagram	Block for one datagram, up to n bytes.

Structs

```
def struct net.UDPSocket { id as int };

def struct net.Datagram {
    data as bytes,
    peer as string      # "host:port" of the sender
};
```

Example: minimal client

```
use net;
use convert;

def s as net.UDPSocket init net.listenUDP(":0");
net.sendTo($s, "1.2.3.4:53", convert.bytesFromString("query", "utf-8"));
def reply as net.Datagram init net.recvFrom($s, 4096);
# $reply.data is the payload; $reply.peer is who sent it
net.close($s);
```

The bind-and-use pattern doubles as the client pattern: bind to `:0` (kernel picks a port), send to the remote peer.

DNS

Two lookup helpers; specialised record-type variants (`net.lookupMX` , `net.lookupTXT` , ...) ship when needed.

Call	Returns	Notes
net.lookup(host)	list of string	Resolve host to a list of IP addresses (v4 and v6 mixed).
net.reverseLookup(ip)	list of string	Reverse DNS: IP address to a list of hostnames.

```
use io;
use net;

def ips as list of string init net.lookup("example.com");
for (def ip in $ips) {
    io.printf("%s\n", $ip);
}
```

DNS lookups are blocking. Compose with `spawn` to overlap resolution with other work.

Close

`net.close` is polymorphic - it accepts a `net.Conn` , a `net.Listener` , or a `net.UDPSocket` and dispatches based on the struct tag. One verb; three closable kinds; the boundary check errors cleanly if the caller passes anything else.

```
net.close($conn);         # closes a TCP connection
net.close($listener);     # closes a TCP listener
net.close($socket);       # closes a UDP socket
```

Use-after-close on any of the three surfaces: `net.readBytes: net.Conn id 3 is not open (already closed, or never opened)` .

Address helpers

`net.address` is polymorphic over the three handle kinds:

- `net.address($conn)` returns the peer's remote address ("who am I talking to?").
- `net.address($listener)` returns the local bound address ("what did the kernel pick for me?").

- `net.address($sock)` (UDP) returns the local bound address.

The listener form is the one you need after binding to `":0"` to discover which ephemeral port you actually got.

```
def listener as net.Listener init net.listen(":0");
io.printf("bound to %s\n", net.address($listener));
```

Address format

TCP and UDP addresses take the standard `"host:port"` form. For IPv6 you must bracket the host: `"[::1]:8080"` . `host` may be a hostname (resolved via the system's DNS at connect/bind time) or a literal IP address; `:port` alone binds all interfaces.

```
"example.com:80"         # v4 or v6, resolver decides
"127.0.0.1:8080"         # v4 loopback
"[::1]:8080"             # v6 loopback (brackets required)
":8080"                  # bind on all interfaces
":0"                     # bind to any free ephemeral port
```

Concurrency composition

Blocking calls compose with `spawn` for non-blocking use:

```
use net;
use task;

def slow as task of net.Conn init spawn {
    return net.connect("some.slow.host:80");
};
# ... other work while the connect is in flight ...
def c as net.Conn init task.wait($slow);
```

The parallel-server pattern in TCP > Server pattern above is the workhorse case.

Errors

Every error is positioned at the Jennifer call site with the address or handle id in the message.

- **Missing host / unreachable peer** : `net.connect: nonexistent.invalid:9999: dial tcp: lookup nonexistent.invalid: no such host` .
- **Bind failure** (port in use, permission): `net.listen: :80: listen tcp :80: bind: permission denied` .
- **Use after close** : `net.readBytes: net.Conn id 3 is not open (already closed, or never opened)` .
- **Wrong type on polymorphic close** : `net.close: argument must be a net.Conn, net.Listener, or net.UDPSocket; got net.Datagram` .
- **DNS misconfig** : `net.lookup: whatever: lookup whatever: no such host` .
- **Peer address parse** : `net.sendTo: bogus: address bogus: missing port in address` .

Every error is catchable with `try` / `catch` :

```
try {
    def c as net.Conn init net.connect("possibly-down.host:80");
} catch (e) {
    io.printf("connect failed: %s\n", $e.message);
}
```

TinyGo compatibility

The **stock** `jennifer-tiny` binary ships **without a network stack** , so every `net` entry point on it returns a friendly Jennifer-level error rather than failing cryptically deep inside Go's `net` package:

```
net.connect: `jennifer-tiny` (TinyGo build) does not include a
network stack; use the default `jennifer` binary for network I/O
```

This is a property of our **stock tiny build** , not a hard TinyGo limitation. TinyGo compiles most of `net.Dial` / `net.Listen` ; what a default target lacks is a **netdev driver** registered at runtime (the network device the `net` package dials through), and our stock build registers none - so we compile the `tinygo` -tagged stub. A `jennifer-tiny` **rebuilt with a network stack** - a registered netdev driver / a net-capable target, with the `tinygo` build tag dropped on `net` - restores `net` and every net-backed module. So read "use the default `jennifer` binary" as "use a build that includes a network stack"; the stock `jennifer` has one, the stock `jennifer-tiny` does not. (UDP is the thinner spot: `net.ListenPacket` isn't part of TinyGo's surface today, so a rebuild covers TCP / DNS more readily than UDP.)

Same pattern as `os.run` / `os.spawn` on TinyGo. See ../technical/tinygo.md.

What's not in v1

Recorded so the design decisions stay visible; ships if a concrete workload forces it.

- **TLS extras: ALPN and session tickets.** Core TLS shipped (`net.connectTLS` / `net.startTLS` with certificate verification and `net.TLSOptions` - see the TLS section above). ALPN protocol negotiation and session-ticket resumption are the remaining pieces, deferred until a workload needs them.
- **Unix domain sockets.**
- **Socket options** (SO_REUSEADDR, KEEPALIVE, NODELAY).
- **DNS record-type helpers** (`net.lookupMX` , `net.lookupTXT` , `net.lookupSRV`). The current pair covers 90% of use.
- **Explicit IPv6 control.** Auto-selected by the resolver; users force by writing `"[::1]:port"` or `"127.0.0.1:port"` .

See also

- ../user-guide/concurrency.md - the `spawn` -and-compose story `net` builds on.
- `task` - observe results from `spawn` ed network workers.
- `fs` - the parallel design for filesystem I/O.
- `convert` - `bytesFromString` / `stringFromBytes` bridge network payloads and Jennifer strings.
- ../technical/tinygo.md - the netdev-driver row in the restrictions table.
- ../milestones.md - design spec.

os - operating-system glue

Enable with `use os;` . Every name lives behind the `os.` prefix (`os.PLATFORM` , `os.getEnv`). Nothing here is reachable as a bare identifier.

```
use io;
use os;

io.printf("platform:     %s\n", os.PLATFORM);
io.printf("architecture: %s\n", os.ARCH);
io.printf("dir sep:      %s\n", os.DIRSEP);
io.printf("home:         %s\n", os.getEnv("HOME"));
io.printf("args:         %d arguments\n", len(os.ARGS));
```

The library splits cleanly: **immutable per-run host facts are uppercase constants** (no arguments to take, no reason to be a function); **operations that take arguments are functions** .

Process exit is the language statement `exit EXPR;` , not an `os` function - see rejected.md > `os.exit(n)` .

Functions

Call	Returns	Notes
os.getEnv(name)	string	Reads an environment variable. Unset variables return "", no error.
os.setEnv(name, value)	null	Sets an environment variable for this process (and any child it later spawns). An invalid name (empty, or containing = / NUL) is a positioned error.
os.hasFlag(name)	bool	True if name is an exact-match element of os.ARGS. See "Flag inspection" below.
os.flag(name)	string	The element immediately after name in os.ARGS, or "" if absent or at end.
os.run(argv[, stdin])	os.Result	Blocking. Run argv to completion; capture stdout/stderr. An optional stdin (string or bytes) is fed to the child then closed. See "External programs" below.
os.spawn(argv)	os.Process	Non-blocking. Start argv, return a handle.
os.wait(p)	os.Result	Block until $p terminates; return captured streams + exit code. Idempotent.
os.poll(p)	bool	Non-blocking: true once $p has exited (a following os.wait returns immediately).
os.kill(p)	null	Send SIGTERM to $p.
os.release(p)	bool	Drop a finished handle from the registry (frees its captured output); errors if $p is still running. Returns whether the handle existed.
os.isTerminal(stream)	bool	Is stream ("stdout" / "stderr" / "stdin") an interactive terminal? See "Terminal detection".
os.cwd()	string	Absolute path of the current working directory. Errors only if it can't be determined.
os.homeDir()	string	The current user's home directory ($HOME on Unix, %USERPROFILE% on Windows). Errors if unresolved.
os.tempDir()	string	Directory for temporary files ($TMPDIR or /tmp on Unix; the %TMP%/%TEMP% location on Windows). Never errors; the directory is not created.

os.catchSignal(name)	null	Start trapping a Unix signal so the program can react to it (see "Signals").
os.gotSignal(name)	bool	Whether name has arrived since the last poll, clearing the flag.

Signals

`os.catchSignal(name)` opts into trapping a signal, and `os.gotSignal(name)` polls whether it has arrived (clearing the flag). The model is **cooperative, never preemptive** : a delivered signal only sets a flag; your program reacts at a point of its own choosing, so no Jennifer code runs in a signal context and the value-semantics guarantees hold. `name` is one of `"int"` , `"term"` , `"hup"` , `"usr2"` (letters-only, lowercased, no `SIG` prefix). The main use is a **graceful shutdown** :

```
use os;

os.catchSignal("term");                  # opt in (also "int" / "hup" / "usr2")
def running as bool init true;
while ($running) {
    serveOneRequest();
    if (os.gotSignal("term")) { $running = false; }   # poll at a safe point
}
shutdownCleanly();                       # close connections, remove temp files, .
..
```

Trapping is opt-in per signal , so defaults survive: until you call `os.catchSignal("int")` , Ctrl-C still terminates the program as usual (a script that never traps SIGINT is stopped by Ctrl-C; one that traps it must poll). A signal never caught is never pending, so a stray `os.gotSignal` is harmless. Note the trade-off: a program that catches a signal and then never polls will not see it (polling latency is real and explicit).

"usr1" is reserved for interpreter diagnostics and cannot be caught: `kill -USR1 <pid>` makes the interpreter print a one-shot snapshot to stderr and keep running - the answer to "my program is stuck in a loop, *where* ?". Use `"usr2"` for a program-defined signal. The snapshot is a fixed, labeled block (delivered at the next loop / method-call boundary):

```
=== jennifer diagnostics (SIGUSR1) ===
time:       2026-01-02T15:04:05+01:00
executing:  path/to/program.j:42:5
tasks:      3 spawned, 2 live
goroutines: 6
memory:     heap 12.4 MiB (sys 68.0 MiB), 14 GCs
=======================================
```

- **time** - when the snapshot was taken (RFC 3339), to line it up with logs.
- **executing** - the `.j` source position at the current loop / call checkpoint.
- **tasks** - `spawn` ed tasks this run: total spawned and how many are still live.
- **goroutines** - the process goroutine count (tracks the spawn workers plus the interpreter).
- **memory** - runtime heap in use, memory obtained from the OS, and GC count - the "is it growing while stuck?" read.

A method-name **call stack** and loop depth are a planned addition (they need call-frame tracking in the interpreter). The dump rides the Unix `SIGUSR1` wiring, so it is Unix-only - but it works on **both** binaries there, `jennifer` and `jennifer-tiny` (subject to the scheduler note below); `kill -USR1` never terminates either binary.

Signals are **Unix-only** , and work on both binaries on a Unix host (`SIGUSR1` / `SIGUSR2` / `SIGHUP` do not exist on Windows, where `os.catchSignal` is a positioned error rather than a crash). Aborting a script is unaffected: Ctrl-C on Windows terminates through the OS, not through this library.

jennifer-tiny scheduler note. The TinyGo binary runs a **cooperative** single-thread scheduler (

`-scheduler=tasks`), so a delivered signal - both an `os.gotSignal` flag and the `SIGUSR1` dump - is observed only when the program reaches a scheduler yield point (a `time.sleep` , blocking I/O, a `spawn` handoff). The trap itself is installed at the OS level, so a signal never terminates the program; but a **pure CPU-bound loop** with no yield can defer the observation indefinitely. Programs that block or sleep (servers, poll loops with `time.sleep`) see signals normally. The default `jennifer` binary uses a preemptive scheduler and has no such latency.

Terminal detection

`os.isTerminal(stream)` answers "is this standard stream an interactive terminal?" - the usual gate for deciding whether to emit ANSI colour or a progress spinner. `stream` is `"stdout"` , `"stderr"` , or `"stdin"` ; any other string, or a non-string, is an error.

```
use os;
def coloured as bool init os.isTerminal("stdout");   # false when piped or
redirected
```

It reports `true` for a terminal and `false` for a pipe or a file redirect. Detection uses the character-device mode bit (no external dependency), so `/dev/null` - also a character device - reads `true` ; that is harmless, since escapes written there are discarded. A stream that can't be inspected reports `false` (the conservative answer: when in doubt, don't emit escapes). On `jennifer-tiny` the minimal runtime may not introspect terminals, in which case it reports `false` .

Flag inspection

`os.hasFlag` and `os.flag` are convenience helpers for the most common "did the user pass `--verbose` ?" and "what value follows `--port` ?" patterns. They are **exact-match only** :

- `os.hasFlag("--port")` is true if `"--port"` appears as a standalone element of `os.ARGS` . It is **false** if `"--port=8080"` appears (different element value).
- `os.flag("--port")` returns the element immediately after `"--port"` if there is one, else `""` . The `--foo=bar` form is not parsed.

This is deliberately minimal. Real CLI parsing (combined short flags like `-rf` , `--foo=bar` , repeated flags, subcommands) belongs to a future `cli` library; if you need any of it now, walk `os.ARGS` yourself.

```
use io;
use os;

if (os.hasFlag("--help")) {
    io.printf("Usage: %s [options]\n", os.ARGS[0]);
    exit 0;
}
def port as string init os.flag("--port");
if ($port == "") {
    $port = "8080";
}
io.printf("listening on %s\n", $port);
```

External programs

`os.run` and the `os.spawn` / `os.wait` / `os.poll` / `os.kill` quartet let Jennifer programs execute other programs. Two struct types are introduced for this:

```
def struct os.Result {                  # not actually written by users -
    exitCode as int,                    # the library registers it under
    stdout   as string,                 # the `os.` prefix.
    stderr   as string
};

def struct os.Process { pid as int };   # opaque handle for a spawned child
                                        # (a monotonic internal id, not the OS
pid).
```

`os.run(argv)` is the synchronous form. `argv` is a `list of string` in the conventional argv shape - program name first, arguments following. Stdout and stderr are captured into the returned `os.Result` :

```
use io;
use os;

def r as os.Result init os.run(["echo", "hello, world"]);
io.printf("%s", $r.stdout);
io.printf("exit: %d\n", $r.exitCode);
```

Feeding stdin. An optional second argument - a `string` or `bytes` - is written to the child's standard input, which is then closed (the child sees EOF). This drives a filter (`sort` , `jq` , `gzip`) or any program that reads stdin, in one call:

```
def up as os.Result init os.run(["tr", "a-z", "A-Z"], "shout");
io.printf("%s", $up.stdout);                # SHOUT

def sorted as os.Result init os.run(["sort"], "banana\napple\ncherry\n");
io.printf("%s", $sorted.stdout);            # apple / banana / cherry
```

It is **deadlock-free** : the whole input is buffered up front and the output is drained as the child runs, so neither side blocks on a full pipe. The captured output is still capped at 16 MiB per stream (a large input can produce a larger output, which truncates with the usual marker). This is the "feed all input, read all output" one-shot form; a persistent, interleaved stdin/stdout session on a spawned process is a separate future capability.

`os.spawn(argv)` is the asynchronous form. It returns immediately with an `os.Process` handle. Drain the streams with `os.wait` (blocking) or check completion with `os.poll` (non-blocking):

```
def p as os.Process init os.spawn(["my-long-task", "--input", "data"]);
while (not os.poll($p)) {
    # do other work
}
def r as os.Result init os.wait($p);
io.printf("done: exit=%d\n", $r.exitCode);
```

`os.wait` is **idempotent** - calling it again on the same handle returns the same `os.Result` immediately. If the child did not terminate cleanly (an I/O failure draining its streams, not a non-zero exit), `os.wait` raises a catchable error rather than reporting a false exit code 0. `os.kill($p)` sends SIGTERM (falling back to a hard kill on Windows, which has no SIGTERM); a subsequent `os.wait` returns whatever the OS reports for the terminated child.

Captured output is capped at 16 MiB per stream for both `os.run` and `os.spawn` : a child that writes more has the excess dropped and a `\n[output truncated at 16 MiB]` marker appended, so a runaway child can't grow the interpreter heap without bound.

Because a handle (and its captured output) stays live for idempotent `os.wait` , a long-running program that spawns many children should `os.release($p)` each handle once it has read the result - it drops the entry from the process registry so it does not grow without bound. Releasing a still-running process is an error (`os.wait` or `os.kill` it first); releasing an already-released handle returns `false` .

```
def r as os.Result init os.wait($p);
# ... use $r ...
os.release($p);   # free the handle in a per-job server loop
```

No shell parsing. `argv` is always a list - Jennifer never concatenates a command string and hands it to a shell. If you genuinely want shell parsing, pass `["sh", "-c", $cmd]` explicitly so the shell hop is visible at the call site. This avoids the shell-injection footguns that plague languages where the easy form is the unsafe form.

Non-zero exit codes are not errors. A failed exit (`exit 1` from the child) populates `$r.exitCode` with the value; the caller branches on it. Only *boundary* failures - program not found, not executable, permission denied, fork/exec failure - are positioned runtime errors at the call site.

Stream buffering. Both stdout and stderr are buffered in memory. A child that produces gigabytes of output will exhaust the interpreter's memory; for streaming workloads, redirect to a file via `["sh", "-c", "cmd > /tmp/out"]` or wait for a future streaming variant.

TinyGo limitation. The `jennifer-tiny` binary (TinyGo build) does not support `os.run` , `os.spawn` , `os.wait` , `os.poll` , or `os.kill` - TinyGo's runtime hasn't implemented the underlying `os/exec` syscalls. Calling these functions on `jennifer-tiny` returns a friendly runtime error pointing at the default `jennifer` binary. The default `jennifer` (`make build` produces both, or `make build-go` for just it) supports the full exec surface. This was the first user-visible gap in Jennifer's two-binary story; `net` hit the same boundary and adopted the same friendly-error pattern.

Constants

Host facts

Name	Kind	Value
os.PLATFORM	string	Operating-system name as reported by the runtime: "linux", "darwin", "windows", ...
os.ARCH	string	CPU architecture: "amd64", "arm64", "wasm", ...
os.NCPU	int	Number of logical CPUs usable by the running process (runtime.NumCPU). Portable stdlib, so it stays OS-independent - it reports usable parallelism, not a raw core count: on jennifer-tiny (cooperative single-thread scheduler) it is 1. For richer host details (CPU model, RAM), read the OS's own files from Jennifer, e.g. /proc/cpuinfo via fs on Linux, so the interpreter stays portable.
os.EOL	string	Platform line ending. "\n" on Unix-likes, "\r\n" on Windows.
os.DIRSEP	string	Path-component separator: "/" on Unix-likes, "\\" on Windows.
os.PATHSEP	string	PATH-list separator (between entries in $PATH): ":" on Unix-likes, ";" on Windows.

Process

Name	Kind	Value
os.ARGS	list of string	Command-line arguments passed to the running program. Index 0 is the script path, the rest follow.

Interpreter-self-identity constants (`VERSION` , `BUILD`) live in `meta` since they describe the interpreter binary itself rather than the host environment.

See also: meta.md, ../user-guide/index.md, ../user-guide/imports.md, ../user-guide/style-guide.md, index.md.

path - filesystem path manipulation

`use path;` enables `path.` -prefixed functions for taking paths apart and building them back up: base name, directory, extension, join, clean. Every function is a **pure string transform** - it never touches the disk (that is `fs` 's job) - so `path` is the string layer that pairs with `fs` 's I/O, the same way `strings` pairs with text.

Paths use the **host separator** (`/` on Linux, `\` on the best-effort Windows build), via Go's `path/filepath` . Build paths with `path.join` instead of hardcoding `"/"` and they stay portable across platforms; the separator itself is `os.DIRSEP` .

Functions

Call	Returns	Notes
path.base(p)	string	The last element of p. "" -> ".", "/" -> "/", a trailing slash is dropped.
path.dir(p)	string	Everything but the last element, cleaned. "c.txt" -> ".".
path.ext(p)	string	The extension including the leading dot (".txt"), or "" when there is none.
path.stem(p)	string	The base name without its extension ("/a/c.txt" -> "c").
path.join(a, b, ...)	string	Join any number (>= 1) of elements with the separator and clean the result; empty elements are dropped.
path.clean(p)	string	The shortest path equivalent to p (collapses ., .., and repeated separators). "" -> ".".
path.isAbs(p)	bool	Whether p is absolute.
path.split(p)	list of string	[dir, file], where dir keeps its trailing separator so dir + file == p.

Not a sanitizer

`path.base` is path **logic** , not a way to sanitize an untrusted filename. It is OS-aware, so on Linux `path.base("a\b")` is `"a\b"` unchanged (backslash is a legal Unix filename byte). To neutralize an attacker-controlled name - an email attachment's filename, an upload field - before writing it, strip **both** separators yourself and reject `.` / `..` , rather than reaching for `path.base` .

Examples

```
use path;
use io;

io.printf("%s\n", path.base("/var/log/app.txt"));      # app.txt
io.printf("%s\n", path.dir("/var/log/app.txt"));       # /var/log
io.printf("%s\n", path.ext("/var/log/app.txt"));       # .txt
io.printf("%s\n", path.stem("/var/log/app.txt"));      # app
io.printf("%s\n", path.join("var", "log", "app.txt")); # var/log/app.txt
io.printf("%s\n", path.clean("a//b/../c"));            # a/c
io.printf("%t\n", path.isAbs("/etc"));                 # true

def parts as list of string init path.split("/var/log/app.txt");
io.printf("[%s][%s]\n", $parts[0], $parts[1]);      # [/var/log/][app.txt]
```

Building a destination path portably (pairs with `fs`):

```
use path;
use os;
use fs;
def dest as string init path.join(os.homeDir(), "downloads", $name);
fs.mkdirAll(path.dir($dest));
fs.writeBytes($dest, $data);
```

TinyGo

The manipulation subset does no I/O, so `path` is TinyGo-clean and ships in **both** binaries (unlike `fs` / `net`, which need a build-tag split).

regex - regular expressions

Enable with `use regex;` . Six verbs over `string` , one match struct. Uses RE2 syntax (Go's `regexp` package) - a documented subset of PCRE: no backreferences, no lookahead/lookbehind.

```
use io;
use regex;

if (regex.matches("^[A-Z][a-z]+$", "Hello")) {
    io.printf("looks capitalised\n");
}
```

Surface

Call	Returns	Notes
regex.matches(pattern, s)	bool	Does pattern match anywhere in s?
regex.find(pattern, s)	regex.Match	First match, or a sentinel with start=-1 on no match.
regex.findAll(pattern, s)	list of regex.Match	Every non-overlapping match, left to right.
regex.replace(pattern, s, replacement)	string	Replace every match. $1, ${name} expand to captured groups.
regex.split(pattern, s)	list of string	Split s at every match of pattern.
regex.escape(s)	string	Escape metacharacters so s is treated as a literal pattern.

The regex.Match struct

```
def struct regex.Match {
    text as string,                          # the full matched substring
    start as int,                            # rune index where the match starts
    end as int,                              # rune index where the match ends
(exclusive)
    groups as list of string,                # positional captures (index 0 =
group 1)
    groupsNamed as map of string to string   # named captures (see below)
};
```

`start` and `end` are **rune indices** , consistent with `strings.substring` and friends. Multi-byte characters advance the count by one per rune, not per byte.

No-match sentinel

`regex.find` on no match returns a Match with `start=-1` , `end=-1` , `text=""` , empty `groups` , empty `groupsNamed` :

```
def m as regex.Match init regex.find("[0-9]+", "no digits here");
if ($m.start == -1) {
    io.printf("no match\n");
}
```

Worked examples

Write patterns as raw single-quoted literals (`'\d+'` , not `"\\d+"`). A single-quoted string is raw - backslashes are literal - so a pattern reads exactly as its RE2 form, with no double-escaping. The cooked double-quoted form works too (every `\` doubled), but the raw form is the idiom throughout these examples.

Match a whole string

```
if (regex.matches("^[A-Z][a-zA-Z]*$", $name)) {
    # ... $name starts with a capital and has no digits ...
}
```

Extract with positional groups

```
def m as regex.Match init regex.find('(\d+):(\d+)', "PORT=8080:9090");
if ($m.start >= 0) {
    io.printf("first=%s second=%s\n", $m.groups[0], $m.groups[1]);
}
```

Extract with named groups

Named groups are addressed by name in `groupsNamed` and also appear in positional `groups` (same order as they appear in the pattern):

```
use regex;
use maps;

def m as regex.Match init regex.find(
    "(?P<key>[a-z]+)=(?P<value>[0-9]+)", "port=8080");
if ($m.start >= 0) {
    io.printf("key=%s value=%s\n",
        $m.groupsNamed["key"], $m.groupsNamed["value"]);
}
```

`maps.has($m.groupsNamed, "some_name")` returns whether a named group is present without erroring on missing keys.

Iterate every match

```
def all as list of regex.Match init regex.findAll('\d+', "a1 b22 c333");
for (def m in $all) {
    io.printf("%s at %d..%d\n", $m.text, $m.start, $m.end);
}
```

Replace with group substitution

`$1` in the replacement string expands to positional group 1; `${name}` expands to a named group. Doubled `$$` produces a literal `$` .

```
def r as string init regex.replace('(\d+)', "port 8080", "[$1]");
# $r is "port [8080]"

def r2 as string init regex.replace(
    '(?P<host>\w+):(?P<port>\d+)', "example.com:80",
    "host=${host} port=${port}");
# $r2 is "host=example.com port=80"
```

Split on a pattern

```
def parts as list of string init regex.split('\s+', "one   two  three");
# $parts is ["one", "two", "three"]
```

Escape a literal

`regex.escape` returns a pattern string that matches its input verbatim. Use it to build patterns from user input or literal strings that would otherwise contain metacharacters:

```
def userInput as string init "1+2=(3)";
def pat as string init regex.escape($userInput);
# $pat is "1\\+2=\\(3\\)"
if (regex.matches($pat, $someHaystack)) { ... }
```

Syntax

Jennifer uses **RE2 syntax** exactly as Go's `regexp` package does. The full reference is at https://github.com/google/re2/wiki/Syntax .

A quick cheat sheet of the most-used pieces:

Pattern	Meaning
.	Any character except newline (add (?s) for dotall).
^ / $	Start / end of line (with (?m)) or of input (without).
\d \w \s	Digit / word char / whitespace.
\D \W \S	Their complements.
[abc] [a-z]	Character class.
[^abc]	Negated character class.
a? a* a+	0-or-1, 0-or-more, 1-or-more (greedy).
a?? a*?	Lazy variants.
a{n,m}	Bounded repetition.
`a	b`
(...)	Grouping and positional capture.
(?:...)	Grouping without capture.
(?P<name>...)	Named capture.
(?i)	Case-insensitive flag.
(?m)	Multiline (^ / $ match at line boundaries).
(?s)	Dotall (. matches newline).

Not supported by RE2 (compile errors):

- Backreferences: `\1` , `\k<name>` .
- Lookahead / lookbehind: `(?=...)` , `(?!...)` , `(?<=...)` , `(?<!...)` .
- Possessive quantifiers: `a++` .

RE2 avoids these by design so its worst-case runtime stays linear in the input; every RE2 pattern runs in bounded time, which is what makes the language usable for untrusted input.

Errors

- **Invalid pattern.** Positioned at the call site with the pattern quoted and the RE2 error message: `regex.find: invalid pattern "[unterminated": error parsing regexp: missing closing ]: [unterminated` .
- **Wrong argument type.** Boundary error: `regex.matches: pattern must be string, got int` .
- **Wrong argument count.** Boundary error: `regex.replace expects 3 arguments (pattern, s, replacement), got 2` .

Every error is catchable with `try` / `catch` .

Pattern caching

The library keeps an LRU cache of compiled patterns (128 entries). Hot loops that reuse a pattern string pay the RE2 compile cost once. Distinct patterns beyond 128 evict the oldest silently; correctness is unaffected.

You don't need to think about this. A future `regex.compile` verb would expose explicit control if a benchmark ever showed the implicit cache wasn't enough.

What's not in v1

Recorded so the design decisions stay visible.

- **regex.compile + regex.Pattern handle.** The implicit LRU cache handles the common case.
- **Non-string operations** (regex over `bytes`).
- **Streaming iterator** (`for (def m in regex.iter(pat, s))`).
- **Global-flag arguments.** Case-insensitive as a boolean parameter would leak an option that already lives in the pattern (`(?i)`).
- **regex.count.** Write `len(regex.findAll(pat, s))`.
- **Backreferences, lookahead, lookbehind.** RE2 doesn't support them; that's the price of guaranteed linear-time matching.

See also

- `strings` - non-regex string helpers (`contains`, `split`, `indexOf`).
- `convert` - `toString` for building patterns from mixed values.
- ../milestones.md - design spec.
- https://github.com/google/re2/wiki/Syntax - full RE2 syntax reference.

stats - statistics: descriptive, distributions, and inference

The `stats` library is the whole statistical toolkit: descriptive statistics over a `list of int` or `list of float`, the probability **distributions** those build on, and the classical **inference** layer (regression, confidence intervals, hypothesis tests). It is pure-value and dependency-free (Go stdlib only, the distribution CDFs reusing `math`'s special functions), so both binaries build it. There is deliberately no separate `prob` library - distributions live with the descriptive stats and inference they are used with, as in `scipy.stats` or R's base `dnorm` / `pnorm` / `qnorm`.

```
use stats;
def xs as list of int init [2, 4, 4, 4, 5, 5, 7, 9];
stats.mean($xs);              # 5.0
stats.median($xs);            # 4.5
stats.stddev($xs);            # 2.0
stats.percentile($xs, 90);    # 7.6
```

Functions

Call	Returns	Notes
stats.mean(xs)	float	Arithmetic mean. Empty list errors.
stats.median(xs)	float	Middle value (mean of the two middles for an even count). Empty errors.
stats.mode(xs)	element kind	Most frequent element (first-reached wins a tie). Empty errors.
stats.variance(xs)	float	Population variance sigma² (?(x - ?)² / n). Empty errors.
stats.stddev(xs)	float	Population standard deviation sigma = ?variance. Empty errors.
stats.sampleVariance(xs)	float	Sample variance s² (÷ n-1, Bessel-corrected). Needs >= 2 elements.
stats.sampleStddev(xs)	float	Sample standard deviation s = ?sampleVariance (the "empirical" SD). Needs >= 2.
stats.mad(xs)	float	Median absolute deviation - a robust (outlier-resistant) spread. Empty errors.
stats.skewness(xs)	float	Population skewness (3rd standardized moment); 0 symmetric, > 0 right tail.
stats.kurtosis(xs)	float	Excess kurtosis (4th moment - 3); 0 for a normal, > 0 heavy tails.
stats.percentile(xs, p)	float	The p-th percentile (p in [0, 100]), linear interpolation. See below.
stats.quartiles(xs)	list of float	[Q1, Q2, Q3] = [p25, p50, p75]. Empty errors.
stats.iqr(xs)	float	Interquartile range = p75 - p25. Empty errors.
stats.min(xs)	element kind	Smallest element. Empty errors.
stats.max(xs)	element kind	Largest element. Empty errors.
stats.range(xs)	element kind	The spread max - min (int for an all-int list, overflow-checked). Empty errors.
stats.sum(xs)	int / float	Sum: int for an all-int list (overflow errors), float otherwise. sum([]) is 0.
stats.geometricMean(xs)	float	Geometric mean (n-th root of the product). Every element must be > 0.

stats.harmonicMean(xs)	float	Harmonic mean (n / sum of reciprocals). Every element must be > 0.
stats.weightedMean(xs, ws)	float	Weighted mean sum(x*w) / sum(w). Equal-length; weights must not sum to 0.
stats.zscore(xs)	list of float	Each element standardized to (x - mean) / stddev (population). Constant list errors.
stats.modes(xs)	list	Every modal value (all tied for the top frequency), in first-seen order; keeps kind.
stats.correlation(xs, ys)	float	Pearson correlation of two equal-length lists (>= 2 elements). See below.
stats.covariance(xs, ys)	float	Population covariance sigma_xy (÷ n), the units-aware companion to correlation.
stats.sampleCovariance(xs, ys)	float	Sample covariance s_xy (÷ n-1), the companion to sampleVariance. Needs >= 2.
stats.describe(xs)	stats.Summary	One-call {count, min, q1, median, mean, q3, max, stddev}. See below.

Each function accepts a `list of int` or a `list of float` (an `int` element promotes to `float` for the real-valued reductions). A non-numeric element, or a non-list argument, is a positioned error.

Kind-preserving vs. real-valued

The reductions that are inherently real-valued - `mean` , `median` , `variance` , `stddev` , `percentile` , `correlation` - always return `float` . The selections that pick an element - `min` , `max` , `mode` - and `sum` preserve the input kind: over a `list of int` they return an `int` , over a `list of float` a `float` .

```
stats.min([2, 4, 9]);        # 2   (int)
stats.min([2.0, 4.0]);       # 2.0 (float)
stats.sum([1, 2, 3]);        # 6   (int, overflow-checked)
stats.sum([1.5, 2.5]);       # 4.0 (float)
```

Population vs. sample (sigma vs s)

`variance` / `stddev` are **population** statistics (sigma², sigma - divide by n) - the list is treated as the whole data set, matching NumPy's default. `sampleVariance` / `sampleStddev` are the **sample** (s², s - Bessel-corrected, `÷ (n-1)`) forms for inferential use, and need at least two elements:

```
population   sigma² = ?(x - ?)² / n           sigma = ?sigma²
sample       s² = ?(x - x?)² / (n - 1)     s = ?s²
```

where ? (population) / x? (sample) is the mean. The sample form `s` is what some sources call the **empirical** standard deviation; Jennifer names the two forms explicitly (`stddev` vs `sampleStddev`) rather than by that ambiguous label. `covariance` follows the population convention (sigma_xy, ÷ n) to match `variance` ; `sampleCovariance` is the sample form (s_xy, `÷ (n-1)`). The normalized `correlation` divides out the standard deviations, so its n factor cancels and the population/sample distinction does not matter there. The companion `linalg` library does linear algebra over the same value types; further ML primitives atop the two are on the horizon.

Percentile

`stats.percentile(xs, p)` sorts the data and interpolates linearly between the two nearest ranks (NumPy's default method): p = `0` is the minimum, p = `100` the maximum, and p = `50` equals `median` . p may be an `int` or `float` ; a value outside `[0, 100]` , or an empty list, is an error.

Correlation

`stats.correlation(xs, ys)` is the Pearson correlation coefficient in `[-1, 1]` : `1.0` for a perfect positive

linear relationship, `-1.0` for a perfect negative one, `0.0` for none. The two lists must be the same length with at least two elements, and neither may have zero variance (a constant list) - all of which are catchable errors. `stats.covariance(xs, ys)` is the raw, units-aware version (the numerator, ÷ n).

Quartiles, IQR, and z-scores

`stats.quartiles(xs)` returns `[Q1, Q2, Q3]` (`[p25, p50, p75]`) as a `list of float` ; `stats.iqr(xs)` is `p75 - p25` , useful for spread and outlier detection. `stats.range(xs)` is the full spread `max - min` (kind-preserving). `stats.zscore(xs)` standardizes every element to `(x - mean) / stddev` (population), returning a `list of float` - a constant list has no z-score and errors.

Geometric and harmonic means

`stats.geometricMean(xs)` (the n-th root of the product, computed via logs so a long list does not overflow) suits growth rates and ratios; `stats.harmonicMean(xs)` (n / sum of reciprocals) suits averaging rates. Both require every element to be strictly positive. `stats.weightedMean(xs, weights)` averages `xs` weighted by `weights` (same length; the weights must not sum to zero).

Shape

`stats.skewness(xs)` is the population skewness (the standardized third moment): `0` for a symmetric distribution, positive when the right tail is longer. `stats.kurtosis(xs)` is the **excess** kurtosis (the standardized fourth moment minus 3), so a normal distribution is `0` and heavier-than-normal tails are positive. Both need at least two elements and a non-constant list. `stats.mad(xs)` (median absolute deviation) and `stats.modes(xs)` (every modal value, versus `mode` 's single one) round out the robust / multimodal summaries.

describe

`stats.describe(xs)` computes a whole summary in one pass and returns a **stats.Summary** struct - `count` (`int`) plus `min` , `q1` , `median` , `mean` , `q3` , `max` , and `stddev` (population, all `float`):

```
def s as stats.Summary init stats.describe($data);
io.printf("median %v, IQR %v\n", $s.median, $s.q3 - $s.q1);
```

Distributions

The probability distributions inference is built on, with flat R-style names. Every function returns a `float` . The continuous CDFs reduce to `math` 's regularized incomplete gamma / beta; the quantiles (inverse CDFs) invert them numerically. Sampling draws on `math` 's shared random source (so `math.randSeed` makes it reproducible).

Call	Meaning
stats.normalPdf(x, mean, sd) / normalCdf / normalQuantile(p, mean, sd) / normalSample(mean, sd)	Normal (Gaussian) N(?, sigma²). sd > 0.
stats.tPdf(x, df) / tCdf(x, df) / tQuantile(p, df)	Student's t. df > 0.
stats.chiSquareCdf(x, df) / chiSquareQuantile(p, df)	Chi-square (?²). x >= 0, df > 0.
stats.fCdf(x, df1, df2) / fQuantile(p, df1, df2)	F distribution. x >= 0, dfs > 0.
stats.binomialPmf(k, n, p) / binomialCdf(k, n, p)	Binomial. k, n ints; 0 <= p <= 1.
stats.poissonPmf(k, lambda) / poissonCdf(k, lambda)	Poisson. k int; lambda > 0.

A quantile probability must be in the open interval `(0, 1)` (the boundaries map to infinity); an out-of-domain parameter is a catchable error, never a `NaN` .

```
use stats;
stats.normalCdf(1.96, 0, 1);        # 0.975...
stats.normalQuantile(0.975, 0, 1);  # 1.959...
stats.tQuantile(0.975, 10);         # 2.228... (t critical value)
stats.binomialPmf(5, 10, 0.5);      # 0.2461
```

Inference

Regression, confidence intervals, and hypothesis tests. Results come back as small structs whose fields are:

```
def struct Regression { n as int, slope as float, intercept as float, r as float,
r2 as float, stdErr as float, pValue as float };
def struct Interval { lower as float, upper as float };
def struct Test { statistic as float, df1 as float, df2 as float, pValue as float
};
```

`Regression` is a simple-OLS fit: `slope` / `intercept` , the correlation `r` and its square `r2` , the slope's standard error `stdErr` , its two-sided `pValue` , and the point count `n` . `Interval` is a `[lower, upper]` confidence interval. `Test` carries the test `statistic` , its degrees of freedom `df1` / `df2` (`df2` is `0` for a single-df test like the t or chi-square; both are set for an F / ANOVA), and the `pValue` .

Call	Returns	Meaning
stats.linearRegression(xs, ys)	Regression	Simple OLS: slope, intercept, r, r2, stdErr (of the slope), pValue, n.
stats.multipleRegression(X, ys)	list of float	Coefficients [intercept, b1, b2, ...] for the design rows X (normal-equations solve).
stats.confidenceInterval(data, level)	Interval	t-based CI for the mean at confidence level in (0, 1).
stats.proportionCi(successes, n, level, method)	Interval	Binomial-proportion interval; method is "wald", "wilson", or "clopper-pearson" (exact).
stats.tTest(data, mu)	Test	One-sample t-test of the mean against mu.
stats.tTest2(a, b)	Test	Two-sample Welch t-test (unequal variances).
stats.chiSquareTest(observed, expected)	Test	Chi-square goodness-of-fit; expected counts must be positive.
stats.fTest(a, b)	Test	Two-sided F-test for equal variances.
stats.anova(groups)	Test	One-way ANOVA over a list of groups.
stats.histogram(data, binEdges)	list of int	Bin counts (Excel FREQUENCY); k+1 ascending edges give k bins, the last closed on the right.

A degenerate input (zero variance, a singular design, a non-positive expected count, a negative observed count, or magnitudes that overflow the computation) is a catchable error.

A few boundary and numerical notes:

- **Proportion intervals at the extremes.** For `proportionCi` at `successes` == `0` or `successes` == `n` , the `"wald"` and `"wilson"` formulas collapse to a degenerate point interval (`[0, 0]` / `[1, 1]`) - the well-known boundary defect of those methods. Use `"clopper-pearson"` (the exact interval) when a boundary count matters; it yields a proper one-sided bound there.
- **Regression conditioning.** `linearRegression` / `multipleRegression` solve the normal equations by Gaussian elimination (the same approach as `linalg.solve`). That is fast and exact for well-conditioned data, but numerically weak for near-collinear predictors or extreme scales; the coefficients can lose precision before the design is flagged singular. Center / scale the predictors when they span very different magnitudes.
- **Cost.** `multipleRegression` is `O(n*k²)` to form the system and `O(k³)` to solve it, for `k` predictors - fine

for the modest tabular data a tree-walker handles, not a many-thousand-column design.

```
use stats;
use io;

def r as stats.Regression init stats.linearRegression([1, 2, 3, 4, 5], [2.1, 3.9,
6.2, 7.8, 10.1]);
io.printf("y = %.2fx + %.2f (r2 %.3f, p %.4f)\n", $r.slope, $r.intercept, $r.r2,
$r.pValue);

def ci as stats.Interval init stats.proportionCi(8, 10, 0.95, "clopper-pearson");
io.printf("95%% CI: [%.3f, %.3f]\n", $ci.lower, $ci.upper);

def t as stats.Test init stats.tTest([5.1, 4.9, 5.2, 4.8, 5.0], 4.5);
io.printf("t = %.3f, p = %.4f\n", $t.statistic, $t.pValue);
```

Strictness

Following `math` 's stance, a non-finite or mathematically undefined result is a catchable error rather than a `NaN` or `Inf` : an empty list (for every reduction except `sum`), a percentile p outside `[0, 100]` , mismatched or too-short `correlation` inputs, a zero-variance `correlation` , and an int `sum` that overflows `int64` all raise a positioned error you can `try` / `catch` . The same applies when an input's *magnitudes* overflow the computation - a `list of float` with values near the float64 ceiling (`~1.8e308`) can push an intermediate sum or power to `+/-Inf` (and thence `NaN`); every real-valued reduction rejects a non-finite result instead of returning it, so a `NaN` never escapes into the type system.

ml - classical machine learning

Enable with `use ml;` . Classical / predictive machine learning on tabular data - the scikit-learn-lite core companion to `stats` and `linalg` . Models follow a **fit / predict** shape: a fit function (`ml.kMeans` , `ml.linearRegression` , ...) trains and returns an opaque `ml.Model` handle, and `ml.predict` / `ml.transform` apply it. Pure Go stdlib, TinyGo-clean, both binaries.

It is **not** a deep-learning framework: tensors, autodiff, and training deep nets are native GPU / C++ territory a tree-walker cannot usefully replace, and "run a pre-trained model" is already `http` / `os.run` .

```
use ml;
use io;

# Fit a model, then apply it.
def X as list of list of float init [[1.0, 1.0], [2.0, 1.0], [1.0, 2.0], [3.0, 2.
0]];
def y as list of float init [6.0, 8.0, 9.0, 13.0];   # y = 2*x1 + 3*x2 + 1
def model as ml.Model init ml.linearRegression($X, $y);
io.printf("%v\n", ml.predict($model, [[5.0, 5.0]]));  # ~[26.0]
```

Data shape

A **feature matrix** `X` is a `list of list of float/int` (one inner list per sample, one entry per feature); **labels / targets** `y` are a `list of float/int` . Class labels are integers (`0` , `1` , `2` , ...); regression targets are any number. `ml.predict` returns a `list of int` for a classifier / cluster model and a `list of float` for a regressor.

Fitting models

Each returns an opaque `ml.Model` handle. A fitted model is immutable, so a handle is safe to share across value-copies and `spawn` ed tasks (read-only).

Fit call	Kind	Notes
ml.linearRegression(X, y)	regression	Ordinary least squares (normal equations).
ml.ridge(X, y, alpha)	regression	L2-regularized OLS; alpha >= 0 shrinks the coefficients (not the intercept).
ml.lasso(X, y, alpha)	regression	L1-regularized (coordinate descent); drives small coefficients to exactly 0.
ml.kNN(X, y, k)	classifier	k-nearest-neighbours majority vote (Euclidean).
ml.kNNRegressor(X, y, k)	regression	k-NN averaging the k nearest targets.
ml.naiveBayes(X, y)	classifier	Gaussian naive Bayes (multiclass).
ml.logisticRegression(X, y [, lr [, epochs]])	classifier	Binary or multiclass (one-vs-rest); gradient descent (lr default 0.1, epochs 1000). Only the binary case has predictProba.
ml.decisionTree(X, y [, maxDepth])	classifier	CART with Gini impurity (maxDepth default 8).
ml.decisionTreeRegressor(X, y [, maxDepth])	regression	CART with variance-reduction splits, leaf = mean target.
ml.randomForest(X, y [, nTrees [, maxDepth]])	classifier	Bagged trees with per-split feature subsampling (nTrees 10).
ml.randomForestRegressor(X, y [, nTrees [, maxDepth]])	regression	Bagged regression trees (mean of tree predictions).
ml.kMeans(X, k [, maxIter])	clustering	Lloyd's algorithm, k-means++ seeding.

ml.pca(X, nComponents)	transform	Principal component analysis (covariance eigendecomposition).
ml.standardScaler(X)	transform	Fit a z-score scaler (per-feature mean / stddev).
ml.minMaxScaler(X)	transform	Fit a [0, 1] min-max scaler.

Inspecting a fitted model

A model is opaque, but you can read the parameters it learned:

Call	Applies to	Returns
ml.coefficients(model)	linear / ridge / lasso / logistic	list of float (per feature), or list of list of float (per class) for multiclass logistic.
ml.intercept(model)	linear / ridge / lasso / logistic	float, or list of float per class for multiclass logistic.
ml.centroids(model)	kMeans	list of list of float (cluster centres).
ml.components(model)	pca	list of list of float (principal axes, one row per component).
ml.explainedVariance(model)	pca	list of float - the variance ratio each component captures (choose nComponents from these).
ml.featureImportances(model)	decisionTree / randomForest	list of float (Gini importances, summing to 1).

```
use ml;
use io;
def m as ml.Model init ml.linearRegression([[1.0, 1.0], [2.0, 1.0], [1.0, 2.0],
[3.0, 2.0]], [6.0, 8.0, 9.0, 13.0]);
io.printf("y = %v . x + %v\n", ml.coefficients($m), ml.intercept($m));   # [2,
3], 1
```

Applying a model

Call	Returns	
ml.predict(model, X)	list of int / list of float	Predicted labels (classifier / cluster) or values (regressor) for each row of X.
ml.transform(model, X)	list of list of float	Transformed features (scalers, PCA).
ml.predictProba(model, X)	list of float	Positive-class probability, for logisticRegression.
ml.free(model)	null	Drop the model to free its memory early (a handle otherwise lives for the run).

The random models (`kMeans` , `randomForest` , `trainTestSplit`) draw from `math` 's shared random source, so `math.randSeed(n)` makes a run reproducible. This is deliberate - reproducibility, not secrecy, is what training needs; a seed is not a secret, so `ml` uses `math` 's seedable source, never `crypto` 's unseedable one (which would make a split or clustering impossible to reproduce).

A fitted model lives in a per-run registry behind its handle until the program ends. When you fit **many** models in a loop (e.g. a large cross-validation), call `ml.free(model)` on the ones you are done with so the registry does not grow unbounded. The cost-driving hyper-parameters are bounded (tree `maxDepth` <= 64, forest `nTrees` <= 1000, `kMeans` `maxIter` <= 10000, logistic `epochs` <= 1e6, `kFold` `nSamples` * k <= 1e8, multiclass logistic <= 100 classes, `polynomialFeatures` degree <= 8, output width <= 1e5, and total cells (rows x columns) <= 2e7); a value above the ceiling is a catchable error, as is a `polynomialFeatures` product that overflows to a non-finite value.

`ml.lasso` , unlike ordinary least squares, is fine with more features than rows (p > n) - that is its

feature-selection use case, so it does **not** require `rows > features`. Feeding it a very wide `polynomialFeatures` design is allowed but does dense O(sweeps x features x rows) work; keep the expansion modest. `ml.logLoss` requires each probability in `[0, 1]` (an out-of-range value errors, rather than being silently clamped).

```
use ml;
use math;
use io;

math.randSeed(1);
def data as list of list of float init [[0.0, 0.0], [0.2, 0.1], [5.0, 5.0], [5.1,
4.8]];
def km as ml.Model init ml.kMeans($data, 2);
io.printf("clusters: %v\n", ml.predict($km, $data));   # e.g. [0, 0, 1, 1]

# Scale features, then reduce dimensionality.
def scaler as ml.Model init ml.standardScaler($data);
def scaled as list of list of float init ml.transform($scaler, $data);
```

Model selection

Call	Returns	
ml.trainTestSplit(X, y, testFraction)	ml.Split	Shuffle and split into {trainX, trainY, testX, testY}; testFraction in (0, 1).
ml.kFold(nSamples, k)	list of ml.Fold	k contiguous folds, each {trainIdx, testIdx} (index lists into your data).
ml.polynomialFeatures(X, degree)	list of list of float	Expand to all monomials up to degree (with a leading bias column); stateless, so the same call fits train and test.

```
export def struct Split { trainX as list of list of float, trainY as list of
float, testX as list of list of float, testY as list of float };
export def struct Fold  { trainIdx as list of int, testIdx as list of int };
```

Metrics

Pure functions over label / value lists (no model). Classification metrics take `(yTrue, yPred)`; `precision` / `recall` / `f1` take an optional third positive-label argument (default `1`).

Call	Meaning
ml.accuracy(yTrue, yPred)	Fraction of exact matches.
ml.precision(yTrue, yPred [, positive])	TP / (TP + FP) for the positive label.
ml.recall(yTrue, yPred [, positive])	TP / (TP + FN).
ml.f1(yTrue, yPred [, positive])	Harmonic mean of precision and recall.
ml.confusionMatrix(yTrue, yPred)	ml.Confusion{labels, matrix} (rows = true, columns = predicted).
ml.rocAuc(yTrue, scores)	Binary ROC-AUC from 0 / 1 labels and predicted scores (tie-aware).
ml.logLoss(yTrue, probas)	Binary cross-entropy of 0 / 1 labels against predicted probabilities.
ml.rmse(yTrue, yPred) / ml.mse / ml.mae	Root-mean-square / mean-square / mean-absolute error (regression).

ml.r2(yTrue, yPred)	Coefficient of determination R^2 (zero-variance targets error).

```
use ml;
use io;
def yt as list of float init [1.0, 0.0, 1.0, 1.0, 0.0, 1.0];
def yp as list of float init [1.0, 0.0, 0.0, 1.0, 0.0, 1.0];
io.printf("accuracy %v, F1 %v\n", ml.accuracy($yt, $yp), ml.f1($yt, $yp, 1));
```

Strictness and scope

Like the rest of the numeric stack, a degenerate input is a catchable error, not a silent NaN: an empty / ragged matrix, mismatched `X` / `y` lengths, a singular regression design (collinear features), a diverged logistic fit, or magnitudes that overflow the computation all raise a positioned error.

`ml` targets the modest tabular data a tree-walker handles - native loops over thousands of rows, not millions. Large-scale training stays a native-tool job. Support-vector machines, gradient-boosted trees, density-based clustering (DBSCAN), and model serialization are out of scope for this tier.

See also

`stats` (distributions + inference - the estimate / test companion to `ml` 's fit / predict), `linalg` (linear algebra), and `math` . The module index.

strings - text utilities

Enable with `use strings;` . Namespaced under `strings.` ; every function is called as `strings.NAME(...)` . Fourteen functions for the common things you do with strings: case conversion, search, trim, replace, repeat, substring extraction, and split / join.

> **Breaking change.** `strings` moved from flat to namespaced. Callers used to write `upper(s)` , `contains(s, sub)` , etc.; the namespaced form is `strings.upper(s)` , `strings.contains(s, sub)` . Same library, just the call-site prefix is mandatory now. The rationale matches the lists/maps move: collision-prone verbs (`contains` , `split` , `replace` , ...) belong in their domain library to keep the bare-name pool clean.

> **Looking for len(s)?** It lives in the auto-loaded `core` library, so it's available everywhere without any `use` statement. The same `len` covers strings, lists, and maps with one polymorphic dispatch.

String positions are 0-relative. The first character is at index `0` , not `1` . So `strings.indexOf("hello", "h")` returns `0` , `strings.substring("hello", 0, 1)` returns `"h"` , and `len("hello")` is the same as the index just past the last character (`5`). This matches Python, JavaScript, Go, Rust, Java, C, C++, C#, Swift, Ruby. Lua/MATLAB/Pascal are 1-relative; Jennifer is not.

All indices and lengths are rune-based (Unicode code points), not bytes. `len("héllo")` is `5` , not `6` . `strings.indexOf` and `strings.substring` agree.

The combination of "0-relative" plus "exclusive end" plus "rune-based" means `strings.substring(s, strings.indexOf(s, x), len(s))` always does what you'd guess - the same units come out of every function.

```
use io;
use strings;

io.printf("%d\n", len("hello"));                      # 5  (core, auto-loaded)
io.printf("%s\n", strings.upper("hello"));            # "HELLO"
io.printf("%t\n", strings.contains("hello", "ell"));  # true
io.printf("%t\n", strings.startsWith("hello", "he")); # true
io.printf("%d\n", strings.indexOf("hello", "l"));     # 2
io.printf("[%s]\n", strings.trim("  hi  "));          # "[hi]"
io.printf("%s\n", strings.replace("a-b-c", "-", "/")); # "a/b/c"
io.printf("%s\n", strings.repeat("ab", 3));           # "ababab"
io.printf("%s\n", strings.substring("hello", 1, 4));  # "ell"
```

Functions

Call	Returns	Notes
strings.upper(s), strings.lower(s)	string	Case conversion (Unicode-aware)
strings.fold(s)	string	Remove common Latin diacritics for a sort / search key (Österreich -> Osterreich, café -> cafe, ß -> ss); case preserved, NFD combining accents dropped. Not full Unicode collation.
strings.contains(s, sub)	bool	Substring search
strings.startsWith(s, prefix)	bool	
strings.endsWith(s, suffix)	bool	
strings.indexOf(s, sub)	int	Rune index of first occurrence; -1 if not found
strings.trim(s)	string	Strip leading and trailing whitespace

strings.trimLeft(s), strings.trimRight(s)	string	One-sided trim
strings.replace(s, old, new)	string	Replace all occurrences of old with new
strings.repeat(s, n)	string	n copies concatenated; n must be non-negative
strings.substring(s, start)	string	Rune-indexed; from start to the end of the string
strings.substring(s, start, end)	string	Rune-indexed; exclusive end
strings.split(s, sep)	list of string	Split on a non-empty separator; preserves empty segments
strings.chars(s)	list of string	One single-rune string per Unicode code point
strings.join(parts, sep)	string	Concatenate a list of string with sep between entries

`strings.split` and `strings.chars` complement each other: use `strings.chars(s)` to break a string into runes (one entry per code point), `strings.split(s, sep)` to break on a delimiter substring. `strings.join` is the inverse of `strings.split` for any non-empty separator: `strings.join(strings.split(s, sep), sep) == s` .

Indexing rules

`strings.substring` , `strings.indexOf` , and `len` all agree on rune indices. So given `s = "héllo"` :

- `len(s) = 5`
- `strings.indexOf(s, "l") = 2`
- `strings.substring(s, 0, 2) = "hé"`
- `strings.substring(s, 2, 5) = "llo"`
- `strings.substring(s, 2) = "llo"` (2-arg form, end defaults to `len(s)`)

The 2-arg `strings.substring(s, start)` is the same as `strings.substring(s, start, len(s))` - a common case worth shortening.

Errors

- `strings.substring(s, -1, 3)` - negative start.
- `strings.substring(s, 0, 99)` - end past the string length.
- `strings.substring(s, 4, 2)` - end before start.
- `strings.repeat(s, -1)` - negative count.
- Non-string arguments where strings are required (`len(42)`).
- Non-int arguments where ints are required (`strings.repeat("x", "3")`).
- Arity errors (too many or too few arguments).

Whitespace

`strings.trim` / `strings.trimLeft` / `strings.trimRight` use Unicode whitespace (Go's `unicode.IsSpace`): ASCII spaces, tabs, newlines, plus characters like non-breaking space (`U+00A0`) and Unicode line separators.

If you need to trim specific characters instead of whitespace, that's not in v1 - propose `strings.trimChars(s, chars)` as a follow-up if it comes up.

Performance

Most `strings` calls are cheap, and for short, human-sized text - a title, a filename, a line of input - you never need to think about speed. Two everyday patterns can get slow when strings grow to thousands of characters and you touch them in a loop. Both come down to how much work gets *repeated* ; the O(1) / O(n) / O(n²) shorthand below is explained in Big O notation (https://en.wikipedia.org/wiki/Big_O_notation). The gist: **O(n)** means the work grows in step with the input, and **O(n²)** means it grows with the *square* of the input - ten times

the text, a hundred times the work.

Build strings with join, not + in a loop

Every + on strings makes a new string and copies both sides into it. Once is fine. In a loop, each step re-copies everything built so far, so the total work is O(n²):

```
# Slow: O(n^2). Each step recopies the whole result so far.
def out as string init "";
for (def piece in $pieces) {
    $out = $out + $piece;
}
```

Instead, collect the pieces in a list and join them in one pass. `strings.join` visits each piece exactly once, so the whole thing is O(n):

```
# Fast: O(n). Collect the parts, then join once.
def parts as list of string init [];
for (def piece in $pieces) {
    $parts[] = $piece;
}
def out as string init strings.join($parts, "");
```

Same idea whenever you assemble something big from many small pieces - an HTML page, a CSV file, a log line built from several fields: gather the parts, then `strings.join` at the end.

Avoid re-copying a big value

Two things in Jennifer quietly make a *whole copy* of a value, which costs O(n) in the size of that value:

- **Passing a list or string to a function.** Jennifer uses value semantics: a function gets its own copy of each argument, so it can never change the caller's data by surprise. Convenient - but calling `helper($bigList)` inside a loop copies the entire list *every time* , turning an O(n) loop into O(n²).
- **strings.substring counts from the start.** Because indices are rune-based (see Indexing rules), `substring` walks the string from the beginning to reach `start` . One call is fine; slicing thousands of small pieces out of one long string adds up to O(n²).

The workaround for both is the same idea: don't hand the whole big value over and over. Convert the string to a rune list *once* with `strings.chars` , then read single positions with `$cs[i]` (that is O(1) each), and when you need a slice back as a string use `lists.slice` plus `strings.join` - which copies only the slice, not the whole string (needs `use lists;`):

```
# Slow: substring re-scans from the start on every call.
def piece as string init strings.substring($big, $start, $end);

# Fast: index into a rune list built once; copy only the slice.
def cs as list of string init strings.chars($big);              # once
# ...then, cheaply, as many times as you like:
def piece as string init strings.join(lists.slice($cs, $start, $end), "");
```

Keeping the big value in one place and passing only small things into helpers - an index, a short piece - follows the same rule.

Accent-folding for sorting and search

`strings.fold(s)` strips common Latin diacritics to build a locale-ish sort or search key, case preserved: `strings.fold("Österreich")` is `"Osterreich"` , `strings.fold("café")` is `"cafe"` , `strings.fold("Straße")` is `"Strasse"` . Ordering operators (< / > / <= / >=) and `lists.sort` compare strings by raw UTF-8 byte, so an accented word sorts *after* all of ASCII - `"Österreich"` lands after `"Zebra"` , not near `"Oslo"` . Fold a key to fix that:

```
use strings;
use lists;

func key(s as string) { return strings.lower(strings.fold($s)); }
def ordered as list of string init lists.sortBy($names, key);   # Österreich
sorts near O
```

It covers the common Western / Central European letters (Latin-1 Supplement plus common Latin Extended-A) and a few expansions (ß / ? -> ss / SS , Æ -> AE); runes outside that table pass through unchanged. A stray combining accent (an NFD-decomposed diaeresis, acute, etc. in the U+0300-U+036F range) is dropped, so the decomposed spelling of a word folds to the same key as its precomposed form. It is **not** full Unicode collation - no per-locale ordering (German phonebook ö = oe , Swedish ö after z), no non-Latin scripts. For those, sort by a fold key that suits your locale, or wait for a dedicated collation library (see the horizon notes).

See also: ../user-guide/index.md, ../technical/interpreter.md, index.md.

task - operations on spawned computations

Enable with `use task;` . Five builtins for observing and joining `task of T` values produced by `spawn { ... }` blocks. The library ships alongside the `spawn` keyword and the `task of T` type kind; together they form Jennifer's concurrency surface.

For the broader story (when to use `spawn` , what value-semantics capture buys you, the exit-time loud-fail contract for unwaited errors), see ../user-guide/concurrency.md.

```
use io;
use task;

def t as task of int init spawn { return 1 + 1; };
def n as int init task.wait($t);
io.printf("%d\n", $n);                      # 2
```

Surface

Call	Returns	Notes
task.wait($t)	T	Block until $t finishes; return its value, or re-raise its error.
task.poll($t)	bool	Non-blocking: true once $t has completed (value or error available).
task.discard($t)	null	Mark $t fire-and-forget so the exit-time loud-fail skips it. Does not block.
task.waitAll($ts)	list of T	Wait for every task in $ts; results in list order; re-raises the first error if any.
task.waitAny($ts)	int	Block until any task in $ts is done; return its index. Caller follows up with task.wait.
task.cancel($t)	null	Request cooperative cancellation of $t (see "Cancellation"). Does not block.
task.cancelled()	bool	Non-raising poll: true if the current spawn body has been cancelled (always false on main).
task.waitTimeout($t, ms)	T	Like task.wait, but throws a catchable "timed out" error if $t doesn't finish in ms (ms up to ~9.2e12; a larger value is a catchable error, not a silent instant timeout).
task.waitAnyTimeout($ts, ms)	int	Like task.waitAny, but throws a catchable "timed out" error if none finishes in ms.

`task.wait` is the workhorse - everything else is a convenience or a non-blocking variant. The full surface is intentionally small; patterns more complex than "wait for the result", "wait for them all", "wait for whichever is first" compose by hand on top of these five.

Error propagation

A `task of T` carries either a value or an error after its body finishes. `task.wait` returns the value when there is one and re-raises the error otherwise - the rethrow surfaces as a positioned runtime error at the wait site, so an enclosing `try` / `catch` catches it the same way it catches any runtime error:

```
use task;

def t as task of int init spawn {
    def xs as list of int init [];
    return $xs[5];                          # out-of-bounds inside the spawn
};

try {
    def n as int init task.wait($t);        # rethrown here
} catch (e) {
    io.printf("caught: %s\n", $e.message);  # "list index 5 out of bounds (len 0)"
}
```

A successful `task.wait` and a `task.wait` that re-raises *both* mark the task observed - the parent saw the outcome either way. `task.discard($t)` is the third way to mark a task observed; use it for fire-and-forget where you genuinely don't care about the result.

Exit-time loud-fail

The contract: a task that ends in an error and is never observed (never `task.wait` 'd, never `task.discard` 'd) has its error printed to stderr at program exit, and the process exits non-zero. No spawn error can silently disappear from the run.

Default to `task.wait` when you need the result; default to `task.discard($t);` when you genuinely don't. Both make the intent visible at the call site. Doing neither is the "no footguns" wake-up call - the loud-fail will surface it.

```
use task;

def alive as task of null init spawn {
    # ... long-running background work ...
    return null;
};
task.discard($alive);                       # explicit fire-and-forget
```

Note: the loud-fail scan blocks on every unobserved task to wait for completion before deciding. A `spawn { while (true) { ... } }` without `task.discard` will hang the program at exit since the goroutine never finishes. Spawned bodies that may not terminate should be paired with an explicit `task.discard` at the top of the scope.

Working with collections of tasks

task.waitAll($ts)

Common pattern: spawn N units of work, wait for all results in order.

```
use task;

func worker(n as int) {
    return $n * $n;
}

def tasks as list of task of int init [];
def i as int init 1;
while ($i <= 4) {
    $tasks[] = spawn { return worker($i); };
    $i = $i + 1;
}

def squares as list of int init task.waitAll($tasks);
# $squares is [1, 4, 9, 16]
```

If any task in the list ended in an error, `waitAll` drains every other task (so the loud-fail stays quiet) and then re-raises the first error in list order. The other errors are observed-but-not- surfaced - if you need them, wait on each task individually with your own try/catch.

task.waitAny($ts)

"First to finish wins" pattern. Returns the *index* of the first completed task; you follow up with `task.wait($ts[$idx])` to read the result (and mark that one observed). The losing tasks keep running; observe them or `task.discard` them so the loud-fail doesn't catch them.

```
use task;

def fast as task of int init spawn { return 1; };
def slow as task of int init spawn {
    # imagine more work here
    return 2;
};
def candidates as list of task of int init [$fast, $slow];

def winner as int init task.waitAny($candidates);
def value as int init task.wait($candidates[$winner]);
task.discard($candidates[1 - $winner]);     # release the loser
```

`task.waitAny([])` is a positioned runtime error - there's nothing to wait on.

Cancellation

`task.cancel($t)` requests **cooperative** cancellation: it sets a flag the spawned body observes at its next **loop checkpoint** (the start of any `while` / `for` / for-each / `repeat` iteration), where the runtime raises a **catchable "task cancelled"** error. That stops the loop - and the spawn - at a safe point. The tree-walker cannot preempt a goroutine, so a body makes progress toward a checkpoint by looping; a tight non-loop computation is instead bounded by the call-depth guard.

This directly retires the exit-time hang: a non-terminating `spawn` that would otherwise block the program at exit can be cancelled and it stops.

```
def t as task of int init spawn {
    def n as int init 0;
    while (true) { $n = $n + 1; }
    return $n;
};
task.cancel($t);
task.discard($t);          # cancel + discard: stop and forget
```

Clean partial results use `try` / `catch` : catch the raised "task cancelled" inside the body and return what you have.

```
def t as task of int init spawn {
    def n as int init 0;
    try {
        while (true) { $n = $n + 1; }
    } catch (e) {
        # cancelled - fall through with the partial count
    }
    return $n;
};
task.cancel($t);
def partial as int init task.wait($t);   # a clean value, not an error
```

`task.cancelled()` is a **non-raising** poll of the current body's cancel flag - useful at an arbitrary point (e.g. before starting an expensive step). It does **not** suppress the loop-checkpoint raise, so inside a loop the raise still fires; the `try` / `catch` idiom above is the way to exit a loop cleanly.

Notes:

- `task.cancel` does **not** wait and does **not** mark the task observed. The idiom to stop-and-forget is `task.cancel($t); task.discard($t);` . If you instead `task.wait` a cancelled task whose body didn't catch the error, the "task cancelled" error is re-raised at the wait site (catchable).
- Cancelling an already-finished task is a harmless no-op.

Timeouts

`task.waitTimeout($t, ms)` is `task.wait` bounded by a millisecond deadline: it returns the result (re-raising a body error, exactly like `wait`) if `$t` completes within `ms` , and otherwise throws a **catchable** "timed out" error. `task.waitAnyTimeout($ts, ms)` is the multi-task form, returning the first finisher's index or throwing on timeout. On a timeout the task(s) stay live and unobserved, so follow up with a retry, or `task.cancel` + `task.discard` .

```
try {
    def v as int init task.waitTimeout($t, 500);   # up to 500 ms
} catch (e) {
    task.cancel($t);
    task.discard($t);
}
```

Errors

The boundary checks are uniform across the library:

- Wrong argument count: `task.wait expects 1 argument (task), got 2` .
- Wrong scalar / structural type: `task.wait: argument must be a task, got int` , `task.waitAll: argument must be a list of task, got string` , `task.waitAll: element 2: argument must be a task, got int` .
- Empty list to `waitAny` : `task.waitAny: list is empty (no tasks to wait on)` .

All errors are positioned at the call site. Errors re-raised by `task.wait` carry the position from the spawn body, so debuggers and human readers see the actual fault location, not the wait site.

See also

- ../user-guide/concurrency.md - worked-example tour: when to spawn, what value-semantics capture buys you, the loud-fail contract, what's deliberately deferred to later milestones.
- ../technical/interpreter.md > Concurrency - internals: goroutine mapping, frame snapshot, error routing, registry, exit-time scan.
- ../milestones.md - ships `spawn` + `task of T` + the `task` library; later milestones use them to build `fs` , `net` , `httpd` .

term - terminal control

Enable with `use term;` . The terminal host capabilities an interactive TUI needs and pure `.j` cannot reach: **raw mode** (unbuffered, no-echo input), the **terminal size** , and **raw single-byte reads** from stdin. This is the low level - screen control, key decoding, and rendering belong in a `.j` layer on top; `term` is only the primitives that require the host.

An **output-only** TUI (a dashboard, a progress bar) needs none of this - just `ansi` escape codes and `os.isTerminal` . Reach for `term` when you need to read keypresses as they happen.

`term` is a **system library** built on `golang.org/x/term` (the package the REPL's own line editor uses). Build-tag split like `net` : the default `jennifer` binary has the real implementation; `jennifer-tiny` returns a friendly error (a minimal / embedded target may have no controlling terminal).

Surface

Call	Returns	Notes
term.makeRaw(stream)	term.State	Put the terminal into raw mode; stream must be "stdin" (raw mode governs input). Returns a handle for restore.
term.restore(state)	null	Undo makeRaw, restoring the terminal. The handle is single-use.
term.size(stream)	term.Size	The terminal's {rows, cols} (query "stdout").
term.readByte()	int	The next raw byte from stdin (0-255), or -1 at end of input.

`stream` is `"stdin"` / `"stdout"` / `"stderr"` (the same names as `os.isTerminal`). A stream that is not a terminal - a pipe or a redirect - is a catchable error for `makeRaw` / `size` .

Raw mode

In the terminal's default (cooked) mode the OS buffers a whole line, echoes it, and only hands it over on Enter. **Raw mode** turns that off: every keypress is delivered immediately, unechoed - which is what an editor, a pager, or a game loop needs. `term.makeRaw` enters it and returns a `term.State` handle; pass that handle to `term.restore` to put the terminal back:

```
use io;
use os;
use term;

if (not os.isTerminal("stdin")) {
    io.printf("needs an interactive terminal\n");
} else {
    def state as term.State init term.makeRaw("stdin");
    defer term.restore($state);                  # runs however this block exits
    def running as bool init true;
    while ($running) {
        def b as int init term.readByte();       # blocks until a key is pressed
        if ($b == -1 or $b == 113) {             # -1 = end of input, 113 = 'q'
            $running = false;
        } else {
            io.printf("byte %d\r\n", $b);         # raw mode: newline is not
cooked
        }
    }
}
```

Always restore. Raw mode is a property of the terminal device, not the process, so a program that exits still in

raw mode would leave the shell there. Put `defer term.restore($state);` right after the `makeRaw` (as above) so the restore runs on **every** exit path - including a `throw` unwinding through the block (control-flow > `defer`). Note that in raw mode the newline is no longer cooked, so prints need an explicit `\r\n` . The `term.State` handle is **single-use** : a second `term.restore` of the same handle is an error, so a live terminal is never clobbered by a stale handle.

As a **backstop** , the `jennifer` CLI cooks the terminal back on the way out even when your program doesn't - a normal exit, an `exit` , an uncaught error, a panic, or an uncaught terminating signal (`SIGINT` / `SIGTERM` / `SIGHUP` , re-raised so the process still dies as usual). This is a safety net for a crash, not a substitute for `restore` : it only runs as the process ends, so a long-running program that means to leave raw mode mid-run must still call `restore` itself, and a `kill -9` (`SIGKILL`) is uncatchable and bypasses it.

Reading keys

`term.readByte` returns one raw byte from stdin (`0` - `255`), or `-1` at end of input. In raw mode it returns the instant a key is pressed. It reads **bytes** , not decoded keys: a plain key is one byte, but an arrow key or a function key arrives as a multi-byte escape sequence (ESC `[` A for Up, ...). Decoding those into key events is the job of a TUI layer on top - `term` deliberately stops at the raw byte.

`term.readByte` is refused inside the REPL, which owns the terminal for its own line editor.

Terminal size

`term.size(stream)` returns a `term.Size{rows, cols}` - the character grid of the terminal behind `stream` (usually `"stdout"`):

```
def dim as term.Size init term.size("stdout");
io.printf("%d rows x %d cols\n", $dim.rows, $dim.cols);
```

The size is a snapshot; a terminal can be resized after the call, so re-query it when you need the current value.

See also

- `os` - `os.isTerminal(stream)` to gate on an interactive terminal before entering raw mode.
- `ansi` - escape codes for colour and cursor control, which pair with `term` for a full-screen TUI.
- milestones.md - the `term` design and the `screen` / `tui` module planned on top of it.

testing - assertions and test-runner primitives

Enable with `use testing;` . An assertion vocabulary plus the system-side runner surface: name-based method invocation, a per-process result accumulator, and a format dispatcher for human / TAP / JUnit output. The `jennifer test` subcommand (discovery, `setUp` / `tearDown` , `--format` , `--isolated`) orchestrates on top; see technical/cli_test.md.

```
use io;
use testing;

func addPasses() {
    testing.assertEqual(1 + 1, 2);
}

testing.run("addPasses");
io.printf("%s", testing.report(testing.results(), "text"));
```

Under `jennifer test` , the `testing.run` / `report` boilerplate is handled for you; a test file just defines `test*` methods with assertions in the body.

Quick start

The everyday path is the `jennifer test` subcommand: write `func test*` methods with assertions, and the runner discovers and runs them - no `testing.run` / `report` boilerplate.

```
use testing;

func testMath() {
    testing.assertEqual(2 + 3, 5);
    testing.assertTrue(len("abc") == 3);
}
```

```
$ jennifer test math_test.j
PASS testMath (0 ms)

1 passed, 0 failed, 1 total
```

`setUp` / `tearDown` methods (run before / after each test) and the flags (`--filter` , `--format=text|tap|junit` , `--isolated`) are documented in technical/cli_test.md. The rest of this page documents the `testing` **primitives** the runner is built from - reach for them directly only when building your own harness.

Why this is a system library

A pure `.j` test runner would have to take *the test body* as a value. Jennifer has no function references / first-class methods - you can't say `testing.run(myTest)` because `myTest` is a name, not a value. The interpreter already does name-based method lookup at every call site; `testing.run` exposes that as a builtin so a Jennifer-coded module can dispatch user methods by name without inventing its own indirection.

`testing.run` is also the one place in the language where `exit` is intercepted. Language-level `try` / `catch` deliberately does **not** catch `exit` (see control-flow.md); the testing runner catches it at the Go level so a runaway `exit` in a test body stays scoped to the runner. This carve-out is why the primitive can't live in `.j` .

Surface

Call	Returns	Notes
testing.run(name)	testing.Result	Look up a zero-arg method by name, call it, catch every failure mode, append.

testing.runWith(name, args)	testing.Result	Like run, but binds the args list to the method's parameters (arity + declared-type checked). For framework dispatchers, not the zero-arg tests the runner discovers.
testing.results()	list of testing.Result	Snapshot of the accumulator. Value semantics - safe to modify.
testing.reset()	null	Clear the accumulator between independent runs.
testing.report(results, format)	string	Render results to "text", "tap", or "junit".

Assertions

Six builtins for test bodies. Each reduces to a `Value.Equal` / Kind comparison in Go - native speed, no per-call interpreter overhead - and, on failure, throws `Error{kind: "assertion"}` positioned at the assertion call, which `testing.run` catches and records.

Call	Fails (throws) when
testing.assertEqual(actual, expected)	actual != expected (deep structural equality: lists / maps / structs compare by value).
testing.assertNotEqual(actual, expected)	actual == expected.
testing.assertTrue(cond)	cond is false (cond must be bool).
testing.assertFalse(cond)	cond is true (cond must be bool).
testing.assertContains(haystack, needle)	needle is absent: substring for a string, element for a list, key for a map (by haystack kind).
testing.assertThrows(name, kind)	the named zero-arg method doesn't throw, or throws an Error whose kind differs.

```
use testing;

func add(a as int, b as int) { return $a + $b; }

func testAdd() {
    testing.assertEqual(add(2, 3), 5);
    testing.assertContains([1, 2, 3], 2);
    testing.assertThrows("mustFail", "boom");
}
```

Table-driven tests

There is no `testing.subtest` primitive; drive a set of cases with a plain loop inside one test method - the idiomatic Jennifer shape:

```
use testing;

def struct Case { input as int, want as int };

func testDoubles() {
    def cases as list of Case init [
        Case{input: 0, want: 0},
        Case{input: 3, want: 6},
        Case{input: -2, want: -4}];
    for (def c in $cases) {
        testing.assertEqual($c.input * 2, $c.want);
    }
}
```

An assertion `throw`s, so the **first** failing case stops that test method (later iterations don't run) and the reported position points at the `assertEqual` line. Put a distinguishing value in the case (or a per-case `assertContains` message) when you need to tell which row failed.

The testing.Result struct

```
def struct testing.Result {
    name as string,
    ms as int,               # elapsed wall time in milliseconds
    passed as bool,
    errorKind as string,     # "" on pass; "runtime" / "error" / "exit" /
"unknown" on fail
    errorMessage as string,
    file as string,          # position where the failure originated (if known)
    line as int,
    col as int
};
```

`errorKind` mirrors the strings surfaced by `try` / `catch` plus a new `"exit"` value for the exit-intercept case:

Value	Meaning
""	The test passed.
"runtime"	An interpreter runtime error (out-of-bounds, missing key, type mismatch, ...).
"error"	A throw whose thrown value wasn't an Error struct.
"assertion" etc.	A throw Error{kind: "assertion", ...} - errorKind mirrors the struct's kind field.
"exit"	The test body called exit. errorMessage is "exit code N".
"unknown"	Anything else (method not found, wrong parameter count, ...).

How testing.run handles each failure mode

```
# Pass path
func passing() {
    return;                     # any normal return counts as a pass
}

# Failure via user throw
func failing() {
    throw Error{
        kind: "assertion",
        message: "expected 42, got 41",
        file: "", line: 0, col: 0
    };
}

# Failure via runtime error
func indexing() {
    def xs as list of int init [];
    def x as int init $xs[5];   # out-of-bounds; kind=runtime
}

# Exit inside a test - captured, doesn't propagate
func earlyExit() {
    exit 1;                     # kind=exit; program keeps running
}
```

Every call to `testing.run` appends exactly one `testing.Result` to the accumulator. The result is also returned, so the caller can inspect it immediately.

Reports

`testing.report(results, format)` takes any list of `testing.Result` and returns a rendered string. Three formats ship in v1; format strings are case-sensitive to match the codec-table shape used by `hash.compute` , `encoding.toText` , `fs.open` .

"text" - human-readable

```
PASS addPasses (0 ms)
FAIL addFails (1 ms)
     [assertion] 1 + 1 != 2
FAIL earlyExit (0 ms)
     [exit] exit code 1

1 passed, 2 failed, 3 total
```

"tap" - Test Anything Protocol v14

```
TAP version 14
1..3
ok 1 - addPasses
not ok 2 - addFails
  ---
  kind: assertion
  message: 1 + 1 != 2
  ...
not ok 3 - earlyExit
  ---
  kind: exit
  message: exit code 1
  ...
```

Works with the `prove` command and most CI harnesses.

"junit" - JUnit XML

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="jennifer" tests="3" failures="2">
  <testcase name="addPasses" time="0.000"></testcase>
  <testcase name="addFails" time="0.001">
    <failure type="assertion" message="1 + 1 != 2">1 + 1 != 2</failure>
  </testcase>
  <testcase name="earlyExit" time="0.000">
    <failure type="exit" message="exit code 1">exit code 1</failure>
  </testcase>
</testsuite>
```

The ubiquitous CI input format.

Unknown format strings error at the boundary: `testing.report: unknown format "html"; known: "text", "tap", "junit"` .

Errors

- **Wrong argument count.** Boundary error: `testing.run expects 1 argument (name), got 0` .
- **Wrong argument type.** Boundary error: `testing.run: name must be string, got int` .
- **Method with parameters.** `testing.run` in v1 only invokes zero-argument methods. Calling it with a method that takes parameters records a failing Result with `errorKind="unknown"` and the reason in `errorMessage` .
- **Unknown format.** `testing.report: unknown format "html"; known: "text", "tap", "junit"` .

Concurrency

The accumulator is guarded by a mutex, so `spawn { testing.run(...) }` from multiple tasks doesn't race. Ordering is by completion time, not spawn time. A test runner that wants stable ordering should run tests sequentially or sort the results before rendering.

Running with jennifer test

These builtins are the substrate; the `jennifer test` subcommand is the orchestration layer on top. It discovers `test*` methods (or `--filter=REGEX`), runs `setUp` / `tearDown` around each, renders `--format=text|tap|junit` , and with `--isolated` runs each test in a fresh interpreter subprocess so one test's crash, `exit` , or leaked global state can't affect the others. Its flags and exit codes are documented in technical/cli_test.md > Test runner. `testing.runWith` (and `Interpreter.CallByNameWith` beneath it) supplies the arg-taking dispatch that parameterised drivers use.

Still deferred:

- **Per-test timeouts.** A non-terminating test still hangs its subprocess; `--isolated` isolates state, not runtime.
- **Skip / xfail.** Runner-level policy, not a primitive.
- **First-class subtests.** A body loop (`for (def c in $cases) { testing.assertEqual(...); }`) covers the observed table-driven cases; a `testing.subtest(name)` primitive would need new language surface.

See also

- technical/cli_test.md > Test runner `--format` , `--isolated` , and the enabling interpreter change.
  - the `jennifer test` subcommand: discovery, `--filter` ,
- milestones.md - design spec and the follow-on.
- control-flow.md
  - the `try` / `catch` machinery `testing.run` builds on.
- concurrency.md - `spawn` under the mutex-guarded accumulator.

serial

Blocking serial-port I/O (`/dev/ttyUSB0` , `/dev/ttyAMA0` , ...) with termios line configuration that plain `fs` cannot reach. **Linux-only** (a `/dev` + ioctl feature); every entry point on a non-Linux host or the `jennifer-tiny` build returns a friendly error pointing at the default `jennifer` binary on Linux.

```
use serial;
use convert;

def port as serial.Port init serial.open("/dev/ttyUSB0", 115200);   # 8N1
serial.write($port, convert.bytesFromString("AT\r\n", "utf-8"));
def reply as bytes init serial.read($port, 64);                     # blocks for
>=1 byte
serial.close($port);
```

Call	Returns	Notes
serial.open(path, baud)	serial.Port	Opens raw 8N1 at baud (a standard rate: 9600 / 19200 / 115200 / ...). A non-standard rate errors.
serial.openWith(path, serial.Options)	serial.Port	Full configuration - see serial.Options below.
serial.read(port, n)	bytes	Blocks until at least one byte arrives, then returns up to n bytes.
serial.write(port, data)	int	Writes data; returns the number of bytes written.
serial.flush(port)	null	Discards buffered input and output.
serial.close(port)	null	Closes the port; later ops on the handle error.

serial.Options

`serial.Options { baud as int, dataBits as int, parity as string, stopBits as int }`

- `baud` - a standard rate.
- `dataBits` - 5, 6, 7, or 8.
- `parity` - `"none"` , `"even"` , or `"odd"` .
- `stopBits` - 1 or 2.

Blocking on purpose - run a read loop in a `spawn` for concurrency (same stance as `fs` / `net`). The handle is the integer-registry pattern: copies share the underlying port, so close it exactly once.

See also

fs, net, spi, i2c, gpio.

spi

SPI devices (`/dev/spidev0.0` , ...). Open a device, set mode / speed, then `transfer` bytes full-duplex (write and read happen together over one exchange). **Linux-only** ; a stub elsewhere and on `jennifer-tiny` .

```
use spi;
use convert;

def dev as spi.Device init spi.open("/dev/spidev0.0");
spi.configure($dev, 0, 1000000);                        # mode 0, 1 MHz
def rx as bytes init spi.transfer($dev, convert.bytesFromString("\x9f", "utf-8"));
spi.close($dev);
```

Call	Returns	Notes
spi.open(path)	spi.Device	Opens the device (default mode 0, 500 kHz, 8 bits/word).
spi.configure(dev, mode, speedHz)	null	Sets clock mode (0..3, CPOL/CPHA) and max speed in Hz.
spi.transfer(dev, data)	bytes	Clocks out len(data) bytes and returns the len(data) bytes clocked in at the same time. Empty in -> empty out.
spi.close(dev)	null	Closes the device.

`transfer` is the `SPI_IOC_MESSAGE` ioctl (one full-duplex message). Blocking; compose with `spawn` .

See also

serial, i2c, gpio, fs.

i2c

The I2C (Inter-IC) bus (`/dev/i2c-1` , ...). **Linux-only** ; a stub elsewhere and on `jennifer-tiny` .

```
use i2c;

def bus as i2c.Bus init i2c.open("/dev/i2c-1", 0x76);   # 7-bit slave address
def id as bytes init i2c.readReg($bus, 0xd0, 1);        # read register 0xd0
i2c.close($bus);
```

Call	Returns	Notes
i2c.open(path, addr)	i2c.Bus	Opens the bus and selects the 7-bit slave addr (0..127; the usable range is 0x08..0x77).
i2c.read(bus, n)	bytes	Reads n raw bytes from the selected slave.
i2c.write(bus, data)	int	Writes raw bytes; returns the count written.
i2c.readReg(bus, reg, n)	bytes	Writes the 1-byte register pointer reg, then reads n bytes (the common "set register, read back").
i2c.writeReg(bus, reg, data)	int	Writes reg followed by data in one transaction; returns the data-byte count.
i2c.close(bus)	null	Closes the bus.

Slave selection is the `I2C_SLAVE` ioctl - the reason a Go library is needed rather than plain `fs` . Blocking; compose with `spawn` .

See also

serial, spi, gpio, fs.

gpio

GPIO over the modern `/dev/gpiochipN` **character device** (the GPIO v2 line ioctls), the mainline-supported interface since sysfs `/sys/class/gpio` was deprecated. **Linux-only** ; a stub elsewhere and on `jennifer-tiny` .

This library reuses the pin-keyed shape of the sysfs-backed `gpio` module - `setup` / `read` / `write` / `release` plus the `IN` / `OUT` direction constants - so a script moves between the two unchanged. The chip device is selected once (default `/dev/gpiochip0`).

```
use gpio;

gpio.setup(17, gpio.OUT);
gpio.write(17, 1);
gpio.release(17);
```

Call	Returns	Notes
gpio.chip(path)	null	Selects the gpiochip device for later setups (default /dev/gpiochip0; env JENNIFER_GPIO_CHIP also honoured).
gpio.setup(pin, direction)	null	Requests pin (0..63) with gpio.IN or gpio.OUT. Errors if the pin is already set up.
gpio.read(pin)	int	0 or 1.
gpio.write(pin, value)	null	value is 0 or 1; the pin must be set up as gpio.OUT.
gpio.release(pin)	null	Releases the line back to the kernel.

Constants

- `gpio.IN` = `"in"` , `gpio.OUT` = `"out"` - the same values as the sysfs module.

The sysfs `gpio` module stays the portable default (it runs on the hobbyist Pi kernels it targets); this ioctl library is the successor for targets that ship without sysfs GPIO.

See also

gpio module, serial, spi, i2c.

sql

A relational-database client over Go's `database/sql` , shipping the two client-server engines: **MySQL/MariaDB/Galera** and **PostgreSQL** (both pure-Go drivers). SQLite (the one embedded engine) is deliberately not here. **Default jennifer binary only** - `jennifer-tiny` returns a friendly error (the drivers are not compiled there, and it has no network stack anyway).

Values bind **only through placeholders** . String interpolation (the SQL-injection vector) is never how a value reaches a query. The placeholder spelling is the engine's own - `?` for MySQL/MariaDB/Galera, `$1` .. `$n` for Postgres; the SQL text goes to the driver verbatim, so write the dialect you are connecting to.

```
use sql;
use io;

def db as sql.Connection init sql.open("postgres", "postgres://user:pw@host/app");
defer sql.close($db);

def result as sql.Result init sql.exec($db, "insert into t(v) values ($1)",
"hello");
io.printf("inserted, %d row(s)\n", $result.affected);

def rows as sql.Rows init sql.query($db, "select id, name from t where id > $1",
0);
while (sql.next($rows)) {
    io.printf("%d %s\n", sql.asInt($rows, "id"), sql.asString($rows, "name"));
}
```

Connections

Call	Returns	Notes
sql.open(driver, dsn)	sql.Connection	driver: "mysql" / "mariadb" or "postgres" / "postgresql". Pings, so a bad DSN / unreachable server errors here.
sql.close(conn)	null	Closes the connection pool.

The DSN format is the driver's:

- MySQL / MariaDB / Galera Cluster: `"user:pw@tcp(host:3306)/dbname"`
- Postgres: `"postgres://user:pw@host:5432/dbname"` (`?sslmode=disable` for a plaintext local server)

A failed `open` / ping error has its DSN password **redacted** before it is raised, so a credential does not leak into an `Error.message` , a log, or an HTTP 500 body.

Query and exec

`query` / `exec` take a **Connection or a Tx** as the first argument, the SQL next, then the placeholder values.

Call	Returns	Notes
sql.query(target, sql, params...)	sql.Rows	A row cursor.
sql.exec(target, sql, params...)	sql.Result	sql.Result{affected, lastId} - read the fields. A field the driver cannot report is -1 (lastId on Postgres, e.g.; affected on the rare driver without row counts).

Parameters bind by type: `int` / `float` / `string` / `bool` / `bytes` / `null` . A struct is a positioned error - build the value first.

Every `query` / `exec` (and the prepared-statement variants) runs under a **client-side deadline** (default **30 s**)

that also bounds the pool-connection acquisition it may block on, so a leaked cursor pinning a pool connection cannot make a later call block forever. Because that same deadline governs the returned cursor's `sql.next` reads, a long streaming loop over a big table would otherwise fail mid-iteration - looking like a database error when it is really the client timeout. So it is **caller-settable** : `sql.setQueryTimeout(ms)` raises it (or `sql.setQueryTimeout(0)` disables it entirely) for a long batch read. The timeout is process-wide; set it once at startup. When it does fire during a read, `sql.next` names the source and the knob, not a phantom database failure.

| `sql.setQueryTimeout(ms)` | `null` | Set the client query/read deadline in milliseconds; `0` (or negative) disables it. Default 30 000. |

Each handle registry (connections, cursors, statements, transactions) is bounded, so a handle leaked in a loop surfaces a catchable "too many open" error rather than growing without limit; always pair `open` / `query` / `prepare` / `begin` with their close verb (a `defer` is the idiom).

A **single list argument is spread** into the parameter sequence: `sql.exec($conn, $sql, $params)` binds each element of `$params` in turn, so a query whose parameter count is only known at runtime (the pattern the `orm` module builds on) works without a fixed call-site arity. The plain variadic form (`sql.exec($conn, $sql, a, b, c)`) is unchanged; a list mixed with other arguments, or a list element that is itself a list, is still a placeholder-binding error.

Reading rows

The cursor is pull-based; `next` advances and scans, then the typed accessors read a column of the current row by **name (string) or 0-based index (int)** .

Call	Returns	Notes
sql.next(rows)	bool	Advance to the next row; false at the end, which closes the cursor and releases the handle (a later sql.next on it is an error).
sql.columns(rows)	list of string	Column names.
sql.asInt(rows, col) / .asFloat / .asString / .asBool / .asBytes	typed	The current row's column, coerced. A NULL column is an error (check sql.isNull first); asString also stringifies a datetime (RFC 3339).
sql.isNull(rows, col)	bool	Whether the column is SQL NULL.
sql.closeRows(rows)	null	Close a cursor early (before exhaustion). A no-op on an already-released cursor, so defer sql.closeRows($rows); is always safe - whether or not the loop ran to the end.

Transactions

Call	Returns	Notes
sql.begin(conn)	sql.Tx	Pass the Tx as the target of query / exec.
sql.commit(tx) / sql.rollback(tx)	null	Ends the transaction; the handle is then unusable. errdefer sql.rollback($tx); pairs well with a commit on the success path.

Prepared statements

Call	Returns	Notes
sql.prepare(conn, sql)	sql.Statement	Prepare once, run many.
sql.queryStmt(stmt, params...)	sql.Rows	
sql.execStmt(stmt, params...)	sql.Result	
sql.closeStmt(stmt)	null	

Blocking; compose with `spawn` . Handles use the integer-registry pattern - close them (a `defer` right after acquisition is the idiom).

See also

net, fs, json, toml.

time - instants, durations, and arithmetic

Enable with `use time;` . Covers one type for absolute instants on the wall-clock timeline (`time.Time`) and one for signed spans (`time.Duration`), with constructors from Unix integers, calendar accessors, and arithmetic / comparison helpers.

This page covers the `time` library surface. A benchmark example uses `time.now()` to measure elapsed time. IANA zone names and daylight-saving transitions are not part of the library yet - it ships fixed-offset zones only. IANA / DST support is planned as a Go-backed extension to this library (delegating to the host tz database), not a hand-maintained data map.

```
use io;
use time;

def start as time.Time init time.now();
# ... do work ...
def gap as time.Duration init time.sub(time.now(), $start);
io.printf("took %d ms\n", time.milliseconds($gap));
```

Types

The library registers two namespaced structs at install time. Field names exist because Jennifer structs have no privacy mechanism, but the conventional API is the function set below - prefer `time.unix($t)` over `$t.nanos / 1000000000` , since the field shape can change between milestones.

```
def struct time.Time { nanos as int, offset as int };
def struct time.Duration { nanos as int };
def struct time.Zone { offset as int, name as string };
```

- `time.Time.nanos` - UTC nanoseconds since the Unix epoch (1970-01-01T00:00:00Z). Fits in `int` (Go `int64`) for any year between 1678 and 2262.
- `time.Time.offset` - seconds east of UTC. The calendar accessors use this to compute wall-clock parts.
- `time.Duration.nanos` - signed nanosecond span. Subtracting a later time from an earlier one produces a negative duration.
- `time.Zone.offset` - seconds east of UTC (`3600` for CET, `-28800` for PST, `0` for UTC). Capped at +/- 26 hours to catch obvious mistakes.
- `time.Zone.name` - display string (e.g. `"CET"` , `"UTC"`). Free-form; the empty string is allowed but `%z` and the ISO formatter use the numeric offset regardless of the name.

Constructors

Call	Returns	Notes
time.now()	time.Time	Current instant in the host's local zone.
time.utc()	time.Time	Current instant in UTC (offset = 0).
time.fromUnix(seconds)	time.Time	At UTC. seconds is int.
time.fromUnixMillis(ms)	time.Time	At UTC. ms is int.
time.fromUnixNanos(ns)	time.Time	At UTC. ns is int.
time.fromIso(s)	time.Time	Parse an RFC 3339 string. Accepts Z or +HH:MM, optional fractional seconds.
time.parse(s, layout)	time.Time	Parse using a strftime layout. See Format codes below.
time.fromSeconds(n)	time.Duration	Span of n seconds.
time.fromMilliseconds(n)	time.Duration	Span of n milliseconds.
time.fromMinutes(n)	time.Duration	Span of n minutes.

time.fromHours(n)	time.Duration	Span of n hours.
time.zone(offset, name)	time.Zone	Fixed-offset zone constructor. offset is seconds east of UTC.
time.local()	time.Zone	Host's current zone (name + offset). The only OS-zone read in the library.

All scalar arguments are strict: a non-int / non-string argument is a positioned runtime error. There is no float-seconds form; pass milliseconds or nanoseconds when sub-second precision is needed.

Two constants live alongside the constructors:

- **time.UTC** (= `Zone{offset: 0, name: "UTC"}`) - the canonical UTC zone. Coexists with the `time.utc()` function (which returns the *current* instant in UTC) because constants and functions live in separate namespace maps.
- **time.PROGRAM_START** (`time.Time`) - the moment the time library was installed, which for the `jennifer` and `jennifer-tiny` binaries is just before the user's source file is read. Use it to anchor "total elapsed since program start" timing without scattering a `def start = time.now();` line at the top of every script. See Measuring total runtime below.

Accessors

Unix integer accessors

Call	Returns	Notes
time.unix($t)	int	Unix seconds since 1970-01-01T00:00:00Z (truncates toward zero).
time.unixMillis($t)	int	Unix milliseconds.
time.unixNanos($t)	int	Unix nanoseconds (no loss of precision).

Calendar accessors

All take a `time.Time` and return `int` . The wall-clock parts honour the time's stored `offset` .

Call	Range	Notes
time.year($t)	full year	E.g. 2024. No two-digit year form.
time.month($t)	1-12	January = 1.
time.day($t)	1-31	Day of month.
time.hour($t)	0-23	24-hour clock.
time.minute($t)	0-59	
time.second($t)	0-59	Whole seconds; sub-second precision lives in nanosecond / unixNanos.
time.nanosecond($t)	0-999_999_999	Fractional second.
time.weekday($t)	1-7 (ISO 8601)	Monday = 1, Sunday = 7 (not Go's 0-based).

Duration accessors

All take a `time.Duration` and return `int` , truncating toward zero at the requested unit.

Call	Notes
time.seconds($d)	Whole seconds in the span.
time.milliseconds($d)	Whole milliseconds.
time.minutes($d)	Whole minutes.
time.hours($d)	Whole hours.

Arithmetic and comparison

Call	Returns	Notes
time.add($t, $d)	time.Time	$t shifted by $d. Preserves the time's offset.
time.sub($a, $b)	time.Duration	Signed: negative when $a is earlier than $b.
time.before($a, $b)	bool	Strictly earlier (UTC instant compare).
time.after($a, $b)	bool	Strictly later.
time.equal($a, $b)	bool	Same UTC instant (the stored offset is ignored).
time.sleep($d)	null	Block the running task for the requested duration. Negative or zero $d returns immediately.

Comparison is on the underlying UTC instant, so `13:00 CET` and `12:00 UTC` compare equal. The library has no operator overloading in v1: write `time.add($t, $d)` , not `$t + $d` .

Measuring total runtime

`time.PROGRAM_START` is captured before the interpreter reads the user source, so anchoring elapsed-time against it gives "total runtime since the program launched" with no per-script setup. Subtract the current instant from it and read the duration at whatever unit you want:

```
use io;
use time;

# ... script body ...

def elapsed as time.Duration init time.sub(time.now(), time.PROGRAM_START);
io.printf("ran for %d ms\n", time.milliseconds($elapsed));
```

For per-section timing inside the script, take a local snapshot of `time.now()` and subtract that instead:

```
def stepStart as time.Time init time.now();
# ... one step of the workload ...
def stepMs as int init time.milliseconds(time.sub(time.now(), $stepStart));
io.printf("step took %d ms\n", $stepMs);
```

`time.PROGRAM_START` is a constant - reading it multiple times always returns the same instant, so it's safe as a long-lived anchor without ever needing to "refresh" it.

Pausing execution

`time.sleep($d)` blocks the running task for the requested duration. Pair it with the duration constructors:

```
use time;
time.sleep(time.fromMilliseconds(250));   # 250 ms pause
time.sleep(time.fromSeconds(2));          # 2 second pause
```

Negative or zero durations return immediately - safe to use with a computed "remaining budget" without checking the sign:

```
def deadline as time.Time init time.add(time.now(), time.fromSeconds(5));
# ... some work that may already overshoot ...
def remaining as time.Duration init time.sub($deadline, time.now());
time.sleep($remaining);   # no-op if work already took longer than 5s
```

`time.sleep` returns `null` ; the caller who wants the wake-up instant calls `time.now()` after it returns. With concurrency, sleep blocks one task instead of the whole interpreter.

Zones

A `time.Zone` is a fixed offset plus a display name; the core library never resolves an IANA name like `"Europe/Vienna"` to an offset, because that resolution depends on date (DST) and on tzdata the interpreter doesn't ship. To shift a `time.Time` into a different wall-clock view, build a `Zone` with the offset you want, then call `time.inZone` :

```
def t as time.Time init time.now();
def vienna as time.Zone init time.zone(3600, "CET");
def tv as time.Time init time.inZone($t, $vienna);
# $tv represents the same UTC instant as $t, but calendar
# accessors and formatters now report wall-clock parts for CET.
```

Call	Returns	Notes
time.zone(offset, name)	time.Zone	offset in seconds east of UTC; capped at +/- 26h.
time.inZone($t, $z)	time.Time	Re-render $t in $z's wall-clock. UTC instant is preserved.
time.local()	time.Zone	Host's current zone. Reads time.Now().Zone() once.
time.UTC (constant)	time.Zone	Zone{offset: 0, name: "UTC"}. Canonical UTC.

DST-aware and IANA-named zones (`"Europe/Vienna"`) are not resolved today. They are planned as a **Go-backed extension** to this library - delegating to the host tz database (`time.LoadLocation`) for historically correct DST - rather than a hand-maintained data map. Until then, build a fixed-offset `Zone` for the offset you need.

Formatting and parsing

`time.format` and `time.parse` use a strftime-style layout. The codes below cover the v1 set; everything outside this list is a positioned error.

```
def t as time.Time init time.fromUnix(1718454896);
def s as string init time.format($t, "%Y-%m-%dT%H:%M:%S%z");
# s = "2024-06-15T12:34:56+0000"
def back as time.Time init time.parse($s, "%Y-%m-%dT%H:%M:%S%z");
```

Format codes

Code	Meaning	Format output	Parse expectation
%Y	4-digit year	2024	exactly 4 digits
%m	Month 01-12	06	exactly 2 digits, 1..12
%d	Day of month 01-31	15	exactly 2 digits, 1..31
%H	Hour 00-23	12	exactly 2 digits, 0..23
%M	Minute 00-59	34	exactly 2 digits, 0..59
%S	Second 00-59	56	exactly 2 digits, 0..59
%z	UTC offset	+0000, +0100	+HHMM, -HHMM, or Z (lenient)
%a	Short weekday (English)	Sat	3 letters, case-insensitive (informational, doesn't set the date)
%A	Long weekday (English)	Saturday	full name, case-insensitive (informational)
%b	Short month name (English)	Jun	3 letters, case-insensitive (sets the month)
%B	Long month name (English)	June	full name, case-insensitive (sets the month)
%j	Day of year 001-366	167	format-only in v1
%u	ISO weekday 1-7 (Mon=1, Sun=7)	6	format-only in v1
%%	Literal %	%	matches % in input

Codes not listed (e.g. `%I` , `%p` , `%y` , `%e`) are reserved and error if used in a layout - the v1 set deliberately stays small and adds only when a use case appears.

Missing parts default to year 1970, month 1, day 1, all the time-of-day at zero, offset 0 (UTC). Trailing input after the layout consumed errors with a positioned message.

time.iso / time.fromIso

ISO 8601 / RFC 3339 round-trip without needing a layout. The output uses `Z` for UTC and `+HH:MM` for any other offset; fractional seconds appear only when non-zero, with trailing zeros trimmed.

```
def t as time.Time init time.utc();
io.printf("%s\n", time.iso($t));         # 2024-06-15T12:34:56Z
def parsed as time.Time init time.fromIso("2024-06-15T13:34:56+01:00");
```

`time.fromIso` accepts both `Z` and `+HH:MM` (or `-HH:MM`), with optional fractional seconds of up to 9 digits.

Errors

The boundary checks are uniform:

- Wrong argument count: `time.now expects 0 arguments, got 1` , `time.fromUnix expects 1 argument (seconds), got 2` .
- Wrong scalar type: `time.fromUnix: seconds must be int, got float` .
- Wrong struct type: `time.seconds: argument must be a time.Duration, got struct` .
- Format / parse layout errors carry the offending verb or position: `time.format: unknown format verb %Q at position 4` , `time.parse: %m: month 13 out of range 1..12` .

All errors are positioned at the call site.

See also

- milestones.md - formatting, parsing, fixed-offset zones, `examples/benchmark.j` , and the planned Go-backed IANA / DST extension.
- imports.md - the library catalog.
- io.md - `io.printf` for displaying results.

toml - TOML encode / decode

Enable with `use toml;` . RFC-conformant TOML 1.0.0 (https://toml.io) decode / encode onto the same opaque, library-owned value that `json` uses - the **same read / walk / write surface, name for name** , so a program that reads, walks, and builds JSON does the same with TOML. The one thing TOML has that JSON does not, its four date-time forms, is surfaced through `toml.asDatetime` (backed by `time`).

`toml.decode(text)` returns an opaque `toml.Value` (a `KindObject` , the sibling of `json.Value`); operators, `[index]` , and `.field` all reject it, so the accessors below are the only way inside. `convert.typeOf` reports `"object"` ; `convert.objectType` reports `"toml.Value"` .

Surface

Call	Returns	Notes
toml.decode(text)	toml.Value	Parse a TOML document into an opaque handle.
toml.encode(v)	string	Render a toml.Value (or a native map / list / scalar) to TOML text.
toml.encodePretty(v)	string	Same, with a blank line separating [table] / [[array]] sections.
toml.typeOf(v[, ptr])	string	Node type: null / bool / int / float / string / list / map / datetime.
toml.get(v[, ptr])	toml.Value	The addressed sub-node, re-wrapped so a walk stays opaque.
toml.has(v, ptr)	bool	Whether the pointer resolves.
toml.keys(v[, ptr])	list of string	Keys of the addressed table, in document order.
toml.length(v[, ptr])	int	Element count of a list, entry count of a table.
toml.asInt(v[, ptr])	int	Strict: a float node errors.
toml.asFloat(v[, ptr])	float	An integer node promotes to float.
toml.asString(v[, ptr])	string	
toml.asBool(v[, ptr])	bool	
toml.asDatetime(v[, ptr])	time.Time	Any of the four date-time forms as a time.Time (needs use time;).
toml.isDatetime(v[, ptr])	bool	Whether the addressed node is a date-time.
toml.map()	toml.Value	A fresh empty table, to build a document from scratch.
toml.list()	toml.Value	A fresh empty array.
toml.set(v, ptr, val)	toml.Value	Upsert a table key / replace an in-range list index.
toml.insert(v, ptr, val)	toml.Value	Insert into a list at an index or - (end).
toml.append(v, ptr, val)	toml.Value	Push onto the list at ptr.
toml.remove(v, ptr)	toml.Value	Remove the addressed key / element.
toml.move(v, from, to)	toml.Value	Relocate a subtree.

The `[, ptr]` argument is a **JSON Pointer** (see below); omit it (or pass `""`) to address the node itself. Every write verb is **non-mutating** - it returns a fresh handle, so the idiom is `$v = toml.set($v, ...)` , the same shape lists and maps use.

`toml.decode` rejects container nesting (and dotted-key segment counts) past a fixed depth with a normal (catchable) decode error, so hostile deeply-nested input can't exhaust the interpreter's stack. The limit is shared by every parser in the toolchain and is set per binary: 1000 levels on the default `jennifer` (Go's growable stack), 64 on `jennifer-tiny` (its fixed 4 MB stack).

Decoding

```
use io;
use toml;

def src as string init "title = \"Jennifer\"
[server]
host = \"localhost\"
ports = [8000, 8001]

[[fruit]]
name = \"apple\"
[[fruit]]
name = \"banana\"
";

def doc as toml.Value init toml.decode($src);
io.printf("%s\n", toml.asString($doc, "/title"));          # Jennifer
io.printf("%d\n", toml.asInt($doc, "/server/ports/0"));    # 8000
io.printf("%s\n", toml.asString($doc, "/fruit/1/name"));   # banana
```

The full TOML 1.0 value grammar decodes: basic / literal / multiline strings, integers (decimal, `0x` hex, `0o` octal, `0b` binary, `_` digit separators), floats (including `inf` / `nan`), booleans, `[table]`, `[[array of tables]]`, dotted keys (`a.b.c = 1`), and inline tables (`{ x = 1, y = 2 }`). Tables become maps in document order; arrays become lists.

JSON Pointer (RFC 6901)

TOML has no document-pointer syntax of its own, and a dotted path would be ambiguous the moment a key itself contains a `.` (the quoted key `"a.b"`). So addressing reuses `json`'s **JSON Pointer** - identical `/`-separated syntax, so a program that walks JSON walks TOML unchanged:

```
toml.get($doc, "/server/ports/0")   # first port
toml.has($doc, "/server/tls")       # false if absent
```

A pointer is `""` (the whole document) or a `/`-led sequence of tokens; `~1` escapes a literal `/` inside a key and `~0` a literal `~` (apply `~1` first). List tokens are `0` or `[1-9][0-9]*`.

Date-times

The date-time forms are the one place TOML is richer than JSON. `toml.typeOf` reports `datetime`; `toml.asDatetime` returns a `time.Time`:

```
use time;
def doc init toml.decode("created = 1979-05-27T07:32:00Z");
def t as time.Time init toml.asDatetime($doc, "/created");
io.printf("%s\n", time.iso($t));    # 1979-05-27T07:32:00Z
```

All four forms parse - offset date-time (`...Z` / `...-07:00`), local date-time (no offset), local date (`1979-05-27`), and local time (`07:32:00`); a space separator (`1979-05-27 07:32:00Z`) is accepted and normalised to `T`. A local date is taken at midnight UTC and a local time on the zero date when converted to a `time.Time`; the original lexical form is preserved for round-trip encoding.

Encoding

`toml.encode` renders a document back to text; `toml.encodePretty` adds a blank line before each section header. The document **root must be a table** (TOML has no top-level array or scalar form), and TOML has **no null type** - encoding a `null` value is an error. A `bytes` value encodes as a base64 string, a `time.Time` as an

offset date-time.

```
def doc init toml.decode($src);
io.printf("%s", toml.encode($doc));
# leaf keys first, then [server], then the [[fruit]] sections -
# so keys always attach to the right table
```

Building and editing

Start from `toml.map()` / `toml.list()` and grow a level at a time. Writes are **strict** (no auto-vivification): `set` creates only the final pointer segment, so build intermediate tables explicitly.

```
use time;
def cfg as toml.Value init toml.map();
$cfg = toml.set($cfg, "/name", "demo");
$cfg = toml.set($cfg, "/server", toml.map());
$cfg = toml.set($cfg, "/server/host", "localhost");
$cfg = toml.set($cfg, "/ports", toml.list());
$cfg = toml.append($cfg, "/ports", 8000);
$cfg = toml.append($cfg, "/ports", 8001);
io.printf("%s", toml.encode($cfg));
# name = "demo"
# ports = [8000, 8001]
# [server]
# host = "localhost"
```

Why TOML and not INI

Jennifer ships **one** structured config format, and it is TOML. INI - the `[section]` / `key=value` shape people reach for first - is deliberately **not** supported, for three concrete reasons:

- **No real standard.** "INI" is a family of mutually-incompatible dialects, not a spec. Parsers disagree on comment characters (`;` vs `#`), quoting, escapes, whether `[a.b]` nests, duplicate keys, and case sensitivity. There is nothing to conform *to* , so "reads INI" would mean "reads our INI."
- **Flat.** INI has one level of `[section]` . It has no arrays of tables, no nested tables, no arrays at all in any agreed form - exactly the structure a real configuration needs.
- **Untyped.** Every INI value is a bare string. `port = 8080` , `debug = true` , and `ratio = 0.5` are all just text; the program re-parses each by hand and guesses the type. There is no `int` / `float` / `bool` / date-time distinction.

TOML fixes all three - a real (versioned) standard, nested tables and arrays of tables, and typed scalars including native date-times - which is why it, not INI, is the format Jennifer decodes into typed values. INI stays out on purpose (documented here rather than silently missing); a tiny `.ini` cousin is a candidate only if concrete demand appears.

See also

- `json` - the sibling library `toml` mirrors name for name.
- `time` - the `time.Time` `toml.asDatetime` returns.
- milestones.md - the `toml` system-library design and the `httpd` config dependency it was sequenced for.

uuid - generate and parse UUIDs

Enable with `use uuid;` . RFC 9562 UUIDs: version 4 (random) and version 7 (time-ordered), the `8-4-4-4-12` hex form, and parse / validate helpers. Self-contained and TinyGo-clean.

Surface

Call	Returns	Notes
uuid.v4()	string	A new random (version 4) UUID.
uuid.v7()	string	A new time-ordered (version 7) UUID.
uuid.parse(s)	bytes	The 16 bytes of a well-formed UUID string; errors on malformed input.
uuid.isValid(s)	bool	Whether s is a well-formed UUID string.
uuid.version(s)	int	The version digit (4, 7, ...; 0 for uuid.NIL); errors on malformed input.
uuid.NIL	string	The all-zero UUID "00000000-0000-0000-0000-000000000000".

```
use io;
use uuid;

def id as string init uuid.v4();
io.printf("%s (v%d)\n", $id, uuid.version($id));   # e.g. 524f1d03-...-
042736d40bd9 (v4)

if (uuid.isValid($id)) {
    def raw as bytes init uuid.parse($id);         # 16 bytes
}
```

v4 vs v7

- **uuid.v4()** - fully random. Use for opaque identifiers with no ordering.
- **uuid.v7()** - a 48-bit millisecond timestamp in the leading bytes, random after. Two v7s created later sort lexically after earlier ones, so they make good database keys (index locality) while staying globally unique.

```
def a as string init uuid.v7();
# ... later ...
def b as string init uuid.v7();
# $a < $b  (string comparison reflects creation order)
```

Randomness

v4/v7 randomness comes from the `crypto` library's crypto-grade source, so a generated UUID is **unguessable** - safe to use directly as a security token, session key, or bearer id. It is not seedable (crypto randomness has no reproducible sequence); for a fast, seedable, deliberately reproducible identifier stream, draw from `math` instead.

See also

- crypto.md - the crypto-grade random source behind v4 / v7.
- cheatsheet.md - every builtin at a glance.

xml - XML encode / decode

Enable with `use xml;` . Parses and serializes XML over an opaque `xml.Value` , designed exactly like `json` and `toml` : `decode` returns a handle you walk with accessors, and a small build surface constructs a tree to `encode` . Hand-rolled (no `encoding/xml` , which is reflect-bound), so it runs on **both binaries** .

Unlike JSON's map/list/scalar tree, XML is a tree of **elements** - each with a tag name, ordered **attributes** , and ordered **children** that mix elements and text. The accessors reflect that shape, and navigation uses a small **XPath-style path** dialect in place of JSON Pointer.

```
use io;
use xml;

def root as xml.Value init xml.decode(
    "<library><book id=\"1\"><title>Go</title></book></library>");
io.printf("%s\n", xml.tag($root));                          # library
io.printf("%s\n", xml.attr(xml.get($root, "book"), "id")); # 1
io.printf("%s\n", xml.text(xml.get($root, "book/title"))); # Go
```

Decode / encode

Call	Returns	Notes
xml.decode(s)	xml.Value	Parse a document into the root element. Errors (with line / column) on malformed XML.
xml.encode(v)	string	Serialize compactly.
xml.encodePretty(v)	string	Serialize with 2-space indentation.

`decode` handles elements, attributes, text, `<![CDATA[...]]>` (as text), comments, processing instructions, an XML declaration, and a `DOCTYPE` (the last four are skipped, not kept). The five predefined entities (`< > & ' "`) and numeric character references (`A` , `A`) are decoded; an unknown entity is an error. Namespace prefixes are kept verbatim in the name (`ns:tag`), and `xmlns` declarations are ordinary attributes.

`encodePretty` indents element-only content, but an element that contains any text child is emitted inline so its character data stays byte-exact.

Reading

Every accessor takes an `xml.Value` . `xml.get` / `xml.findAll` / `xml.has` take a path as a second argument (see below).

Call	Returns	Notes
xml.typeOf(node)	string	"element" or "text".
xml.tag(node)	string	The element's tag name (with prefix, if any).
xml.text(node)	string	The concatenated character data of the element's direct text children (or a text node's own string).
xml.attr(node, name)	string	An attribute's value; errors if the attribute is absent.
xml.hasAttr(node, name)	bool	Whether the attribute is present.
xml.attrs(node)	list of string	The attribute names, in document order.
xml.children(node)	list of xml.Value	The element children (text children are excluded).
xml.get(node, path)	xml.Value	The first element matching the path; errors if none.
xml.findAll(node, path)	list of xml.Value	Every element matching the path (empty if none).
xml.has(node, path)	bool	Whether the path matches at least one element.

Path dialect

A path is `/` -separated steps, evaluated against the element children of the node (a leading `/` is optional; `""` addresses the node itself):

Step	Matches
name	every element child with that tag.
name[k]	the 1-based k-th element child with that tag.
*	every element child, whatever its tag.

`xml.findAll($root, "book")` returns all `book` children; `xml.get($root, "book[2]/title")` drills to the second book's title. Attributes and text are read with `xml.attr` / `xml.text` after navigating, not addressed inside the path. A malformed path is an error; a well-formed path that matches nothing is an empty `findAll` (and a false `has` , and an error from `get`).

Building

The build surface is non-mutating - each call returns a fresh `xml.Value` , the same `$v = xml.set(...)` idiom as `json` / `toml` :

Call	Returns	Notes
xml.element(name)	xml.Value	A new empty element.
xml.setAttr(node, name, value)	xml.Value	The element with the attribute added or updated.
xml.setText(node, s)	xml.Value	The element with its children replaced by one text node.
xml.append(parent, child)	xml.Value	The parent with child (an xml.Value element) appended.

```
use xml;

def note as xml.Value init xml.element("note");
$note = xml.setAttr($note, "type", "info");
$note = xml.setText($note, "hi & bye");
def page as xml.Value init xml.append(xml.element("page"), $note);
# <page><note type="info">hi &amp; bye</note></page>
```

Notes and limits

- **convert.typeOf** reports `"object"` ; **convert.objectType** reports `"xml.Value"` . A bare `$v` at the REPL (or `%v`) shows the compact XML.
- **Comments, PIs, the prolog, and the DOCTYPE are dropped** on decode, so a decode / encode round-trip is faithful for the element tree but not for those. The one root element is required (XML has exactly one).
- **Namespaces are lexical** : prefixes are kept as written, not resolved to their `xmlns` URIs. Full namespace resolution and richer XPath (predicates, `@attr` / `text()` path steps) are future extensions.
- **Element nesting is capped** on both decode and encode: a deeper document (or a tree built that deep with `xml.append`) raises a catchable error rather than overflowing the stack. The limit is shared by every parser in the toolchain and is set per binary: 1000 levels on the default `jennifer` (Go's growable stack), 64 on `jennifer-tiny` (its fixed 4 MB stack). No entity is expanded beyond the five predefined names and numeric references, so there is no entity-expansion ("billion laughs") blow-up, and nothing is fetched externally - an unknown or external entity is simply an error.

See also

- json.md / toml.md - the sibling opaque-handle formats with the same read / write shape.
- cheatsheet.md - every builtin at a glance.

yaml - YAML decode / encode

Enable with `use yaml;` . YAML 1.2 (https://yaml.org) decode / encode onto the same opaque, library-owned value that `json` and `toml` use - the **same read / walk / write surface, name for name** , so a program that reads, walks, and builds JSON or TOML does the same with YAML. Like TOML, YAML has a timestamp scalar JSON lacks, surfaced through `yaml.asDatetime` (backed by `time`).

`yaml.decode(text)` returns an opaque `yaml.Value` (a `KindObject` , the sibling of `json.Value` / `toml.Value`); operators, `[index]` , and `.field` all reject it, so the accessors below are the only way inside. `convert.typeOf` reports `"object"` ; `convert.objectType` reports `"yaml.Value"` .

Unlike `json` / `toml` / `xml` , which are hand-rolled, `yaml` is backed by a Go dependency (`gopkg.in/yaml.v3` (https://pkg.go.dev/gopkg.in/yaml.v3)): full YAML - anchors and aliases, flow and block styles, implicit typing, and multi-document streams - is impractical to hand-roll and has no Go standard library. The dependency is verified TinyGo-clean, so `yaml` builds on **both binaries** (`jennifer` and `jennifer-tiny`).

Surface

Call	Returns	Notes
yaml.decode(text)	yaml.Value	Parse a single-document YAML string into an opaque handle. A multi-document stream errors (use decodeAll); an empty string decodes to null.
yaml.decodeAll(text)	list of yaml.Value	Parse every document of a ----separated stream.
yaml.encode(v)	string	Render to compact flow style ({a: 1, b: [x, y]}).
yaml.encodePretty(v)	string	Render to readable block style (indented, one entry per line).
yaml.typeOf(v[, ptr])	string	Node type: null / bool / int / float / string / bytes / list / map / datetime.
yaml.get(v[, ptr])	yaml.Value	The addressed sub-node, re-wrapped so a walk stays opaque.
yaml.has(v, ptr)	bool	Whether the pointer resolves.
yaml.keys(v[, ptr])	list of string	Keys of the addressed mapping, in document order.
yaml.length(v[, ptr])	int	Element count of a list, entry count of a map.
yaml.asInt(v[, ptr])	int	Strict: a float node errors.
yaml.asFloat(v[, ptr])	float	An integer node promotes to float.
yaml.asString(v[, ptr])	string	
yaml.asBool(v[, ptr])	bool	
yaml.isNull(v[, ptr])	bool	Whether the addressed node is null.
yaml.asDatetime(v[, ptr])	time.Time	A YAML timestamp scalar as a time.Time (needs use time;).
yaml.isDatetime(v[, ptr])	bool	Whether the addressed node is a timestamp.
yaml.map()	yaml.Value	A fresh empty mapping, to build a document from scratch.
yaml.list()	yaml.Value	A fresh empty sequence.
yaml.set(v, ptr, val)	yaml.Value	Upsert a map key / replace an in-range list index.
yaml.insert(v, ptr, val)	yaml.Value	Insert into a list at an index or - (end).
yaml.append(v, ptr, val)	yaml.Value	Push onto the list at ptr.
yaml.remove(v, ptr)	yaml.Value	Remove the addressed key / element.
yaml.move(v, from, to)	yaml.Value	Relocate a subtree.

The `[, ptr]` argument is a **JSON Pointer** (see below); omit it (or pass `""`) to address the node itself. Every write verb is **non-mutating** - it returns a fresh handle, so the idiom is `$v = yaml.set($v, ...)` , the same

shape lists and maps use.

Decoding

```
use io;
use yaml;

def src as string init "title: Jennifer\n"
    + "server:\n"
    + "  host: localhost\n"
    + "  ports: [8000, 8001]\n";

def doc as yaml.Value init yaml.decode($src);
io.printf("%s\n", yaml.asString($doc, "/title"));           # Jennifer
io.printf("%d\n", yaml.asInt($doc, "/server/ports/0"));     # 8000
io.printf("%s\n", yaml.asString($doc, "/server/host"));     # localhost
```

Scalars are typed by YAML's implicit resolution: `42` is an `int` (decimal, `0x` hex, and `0o` octal all resolve), `3.14` a `float` (`.inf` / `.nan` included), `true` / `false` a `bool` , `~` / `null` a `null` , a timestamp a `datetime` , and everything else a `string` . A quoted scalar keeps its string type (`"42"` stays a string), and `yaml.encode` round-trips it faithfully - a string that reads like another type is re-quoted on output. An integer too large for a 64-bit `int` is kept as its exact-digit `string` rather than a lossy `float` . Mappings become maps in document order; sequences become lists; a `!!binary` scalar decodes to `bytes` . A non-scalar mapping key (a `? [a, b]` complex key) and a **duplicate mapping key** are both rejected.

`yaml.decode` guards against resource-exhaustion input with normal (catchable) decode errors: a raw-text pre-scan rejects structural nesting deeper than 128 levels **before** the parse runs (the underlying parser is recursive, and a deeply-nested document would otherwise overflow the interpreter's stack - fatal on the fixed-stack `jennifer-tiny`), and a five-million-node budget stops an alias bomb (anchors that each reference the previous one twice) from exhausting memory.

Anchors, aliases, and merge keys

An anchor (`&name`) and its aliases (`*name`) are resolved **by value** during decode - each alias expands to an independent copy, so the resulting tree has no shared state. A `<<` merge key pulls another mapping's entries in; an explicit key always wins over a merged one, and an earlier merge source wins over a later one:

```
def src as string init "defaults: &def\n"
    + "  retries: 3\n"
    + "  timeout: 30\n"
    + "service:\n"
    + "  <<: *def\n"
    + "  timeout: 60\n";
def doc init yaml.decode($src);
yaml.asInt($doc, "/service/retries")   # 3  (merged in)
yaml.asInt($doc, "/service/timeout")   # 60 (own key wins)
```

Multi-document streams

`yaml.decode` handles a single document and errors on a `---` -separated stream, pointing at `yaml.decodeAll` , which returns one `yaml.Value` per document:

```
def docs as list of yaml.Value init yaml.decodeAll($stream);
for (def d in $docs) { io.printf("%v\n", yaml.keys($d)); }
```

JSON Pointer (RFC 6901)

YAML has no document-pointer syntax of its own, so addressing reuses `json` 's **JSON Pointer** - identical `/`

-separated syntax, so a program that walks JSON walks YAML unchanged:

```
yaml.get($doc, "/server/ports/0")   # first port
yaml.has($doc, "/server/tls")       # false if absent
```

A pointer is `""` (the whole document) or a `/` -led sequence of tokens; `~1` escapes a literal `/` inside a key and `~0` a literal ~ (apply `~1` first). List tokens are `0` or `[1-9][0-9]*` .

Timestamps

A YAML timestamp scalar is the one place YAML is richer than JSON. `yaml.typeOf` reports `datetime` ; `yaml.asDatetime` returns a `time.Time` :

```
use time;
def doc init yaml.decode("created: 2001-12-15T02:59:43Z");
def t as time.Time init yaml.asDatetime($doc, "/created");
io.printf("%s\n", time.iso($t));    # 2001-12-15T02:59:43Z
```

Encoding

`yaml.encode` renders **flow style** (compact, `{a: 1, b: [x, y]}`); `yaml.encodePretty` renders **block style** (the readable, indented form). Flow / block is YAML's own analogue of `json` 's compact / pretty pair. A native `map` / `list` / struct / scalar encodes directly, so a `yaml.Value` handle is not required; a `bytes` value encodes as a `!!binary` scalar and a `time.Time` as a timestamp.

```
def doc init yaml.decode($src);
io.printf("%s", yaml.encodePretty($doc));
```

Block style preserves a timestamp's implicit type on round-trip; flow style quotes a timestamp (so it round-trips as a string) - prefer `encodePretty` when a document carries timestamps.

Building and editing

Start from `yaml.map()` / `yaml.list()` and grow a level at a time. Writes are **strict** (no auto-vivification): `set` creates only the final pointer segment, so build intermediate maps explicitly.

```
def cfg as yaml.Value init yaml.map();
$cfg = yaml.set($cfg, "/name", "demo");
$cfg = yaml.set($cfg, "/ports", yaml.list());
$cfg = yaml.append($cfg, "/ports", 8000);
$cfg = yaml.append($cfg, "/ports", 8001);
io.printf("%s", yaml.encodePretty($cfg));
# name: demo
# ports:
#     - 8000
#     - 8001
```

See also

- `json` / `toml` - the sibling config libraries `yaml` mirrors name for name.
- `time` - the `time.Time` `yaml.asDatetime` returns.
- milestones.md - the `yaml` system-library design and the one place a config parser earns a Go dependency.

Modules

Jennifer modules

A **module** is distributable Jennifer source - a `.j` file whose `export` ed names you bring in with `import` , the same call shape as a system library:

```
import "ansi.j" as ansi;
io.printf("%s\n", ansi.bold(ansi.red("error")));
```

Modules are **not** the Go system libraries. A library (`use NAME;` - see ../libraries/index.md) is compiled into the interpreter binary; a module is ordinary Jennifer code that ships as a file, so you can read it, fork it, or write your own. The modules listed here are the reference set that ships with Jennifer under `modules/` ; the mechanism itself (`import` / `export` , resolution, run-once init) is documented in the Imports guide.

How a module resolves

`import` picks the resolution mode from the leading token of the path:

- `import "./util.j" as u;` (or `../`) - **local** , relative to the importing file's directory.
- `import "/opt/m.j" as m;` - **absolute** path.
- `import "ansi.j" as ansi;` (no `./` , no `/`) - **module** lookup through the search path: the system module directory first (see `jennifer version -v` or `meta.SYSMODDIR`), then any `-I DIR` passed to `jennifer run` . The importing file's own directory is never consulted in this mode.

Distribution packages install the shipped modules to the system module directory (`/usr/share/jennifer/modules/` by default), so `import "ansi.j";` resolves with no path. The `as ALIAS` clause is optional - without it the module is addressed by its file stem (`import "ansi.j";` then `ansi.red(...)`).

Available modules

The **TinyGo** column reports whether the module runs on the constrained `jennifer-tiny` binary. A module is only as portable as the libraries it `use` s: the pure-text modules run on either binary, while `smtp` `use` s `net` , so it needs a build with a network stack.

A **no (net)** entry means the module needs `net` , which the **stock** `jennifer-tiny` build ships without - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs the net-backed modules too; see the note on `net` and TinyGo. Read "needs the default `jennifer` binary" throughout these docs as "needs a build that includes a network stack" (the stock `jennifer` has one).

Module	Import with	TinyGo	Contents
acme	import "acme.j";	no (net)	ACME (RFC 8555) client: obtain / renew TLS certificates from Let's Encrypt and compatible CAs. connect / register an account, order domains, fetch each authorization + challenge, compute the HTTP-01 keyAuthorization or DNS-01 dnsRecord (pure), accept + pollAuthorization, then finalize with a crypto.csr and downloadCertificate. Every request a JWS (RS256 / ES256) over http + json; keys / CSR / JWK from crypto. Needs the default binary.

amqp	import "amqp.j";	no (net)	an AMQP 0-9-1 client for RabbitMQ over net: connect runs the connection / channel handshake (SASL PLAIN), then declareQueue -> QueueInfo, publish / publishText (method + content-header + body frames), get(c, queue, autoAck) -> Message (synchronous Basic.Get pull), ack, close. Server-pushed Basic.Consume via blocking receiveDelivery, declareExchange / bindQueue, message Properties on publish, nack / requeue, and publisher confirms (confirmSelect / waitConfirm). Binary frame / method encoding hand-built with the bitwise ops. The largest protocol module. Needs the default binary.
ansi	import "ansi.j";	full	terminal styling as string wrappers. color / bgColor / style / rgb / strip plus per-colour and per-style shortcuts; TTY-aware.
args	import "args.j";	full	a declarative CLI argument parser (argparse-style) over os.ARGS: a value-semantic Parser built with copy-returning builders - typed flags (long + short, default, required, choices), count / append actions, positionals with nargs (? / * / + / N), subcommands, and --version. parse -> a Result read with asString / asInt / asFloat / asBool / asList / count / has. Unknown / missing / bad-type / bad-choice throw a catchable Error{kind:"args"}; -h / --help set done + helpText. Both binaries.
barcode	import "barcode.j";	full	generate scannable codes as images. encode(data, symbology, opts) -> Symbol for qr (Reed-Solomon over GF(256), EC L/M/Q/H, versions 1-10, mask scoring, byte mode) and 1D code128 / code39 / ean13 / ean8 / itf; render with svg / png (hand-encoded over compress + crc) / terminal / matrix. The GF(256) math lives in a private barcode_ecc.inc.j. No image library. Both binaries.
bloom	import "bloom.j";	full	a Bloom filter (probabilistic set): new(size, hashes) (or optimal(n, fpr) for FPR-target sizing), add / addAll, mightContain - no false negatives, possible false positives; serialize / deserialize to bytes and union / merge. Bits packed in bytes; k positions from double-hashing one SHA-256 digest. Value-semantic. Over hash + strings + binary; both binaries.
cron	import "cron.j";	full	cron schedules: parse(expr) -> Schedule, matches(schedule, t), next(schedule, after) -> time.Time. Five fields with * / , / - / /n; the dom-OR-dow rule; named months (JAN-DEC) and weekdays (SUN-SAT); @-nickname macros (@daily / @hourly / @weekly / @monthly / @yearly / @midnight / @reboot, @reboot being startup-only, never a clock match). A pure calculator over time.
csv	import "csv.j";	full	RFC 4180 comma-separated values. parse / format (*With for any delimiter), toRecords / fromRecords for header-keyed maps; quoting-aware. formatSafe neutralises spreadsheet-formula injection (CWE-1236); a Dialect groups delimiter / quote / comment / trim; streaming reader / writer handles over fs.File.
discord	import "discord.j";	no (net)	post to a Discord channel Webhook on http: send(webhookUrl, content) for a plain message, or build a rich message with message / content / embed(m, title, description, color), full embeds (embedField / embedFooter / embedAuthor), and per-message username / avatar identity override, and post it with sendMessage. render gives the JSON payload; strings JSON-escaped. Sibling of slack / gotify. Needs the default binary.
docblock	import "docblock.j";	full	Jennifer doc-comment format + parser. docblock.parse(source) -> a typed FileDoc (module preamble, per-construct FuncDoc / StructDoc / ConstDoc, tags, and Diagnostics). Reports drift (a @param naming no real parameter, a parameter with no @param) and orphans; string-literal- and nesting-correct scanner. Data, not rendering.

dot	import "dot.j";	full	Graphviz DOT graph description: build a graph of nodes and edges with attributes (digraph / graph, node / nodeWith, edge / edgeWith, graphAttr / nodeAttr / edgeAttr) and render it to .dot text for an external Graphviz tool to lay out (dot -Tsvg). Value-semantic builders; DOT-escaped strings; emits the description only (graph layout is Graphviz's job). Pure .j over strings / lists, both binaries.
dotenv	import "dotenv.j";	full	.env config files: parse(text) / read(path) -> map of string to string, and load(path) (parse + os.setEnv each). Handles # comments, export, single / double quoting, inline comments. Over fs + strings + os.
feed	import "feed.j";	partial	RSS 2.0 and Atom 1.0 web syndication in one module - build and parse, format chosen on build ("rss" / "atom") and detected on parse. A value-semantic Feed of Entry with builders (feed / entry / add / feedUpdated / entryId / entryPublished / entryUpdated / entrySummary / entryContent), build / parse / kind, and fetch over http. Over xml + time (build / parse both binaries); fetch needs the default binary. Untrusted-feed-hardened (nesting cap, no billion-laughs, 64 MiB body cap).
flatdb	import "flatdb.j";	full	a file-backed JSON document store over json + fs: open a file into a value-semantic DB, query / edit by JSON Pointer (get / has / keys / length / set / append / remove), save with a crash-atomic temp+rename. Crash-atomic snapshotting of small data, not a database engine.
font	import "font.j";	full	a pure-Jennifer TrueType / SFNT font parser (no Go): parse(b) / open(path) -> Font, then unitsPerEm / name / advance(f, cp) and glyphPath(f, cp) -> SVG path d / glyph(f, cp) -> Glyph (contours of on / off-curve Points). Parses head / cmap (formats 4 + 12) / maxp / hhea / hmtx / loca / glyf (simple + composite glyphs, quadratic curves) / name. TrueType glyf backend; CFF later. Both binaries.
gotify	import "gotify.j";	no (net)	push notifications to a Gotify (https://gotify.net) server, on top of http: push(cfg, title, message, priority) POSTs the message form with the X-Gotify-Key header, plus pushMarkdown / pushWith (click action) / pushExtras for markdown + click-action notifications via extras; value-semantic Config (url + token).
gpio	import "gpio.j";	full	Raspberry-Pi (Linux SBC) GPIO over sysfs (fs is the whole backend): setup(pin, "in"/"out"), write(pin, 0/1), read(pin), release(pin). Stateless, pin-keyed; JENNIFER_GPIO_BASE overrides the sysfs root (tests / mounts). Absent-platform errors are clear, not crashes.
graphql	import "graphql.j";	no (net)	a thin GraphQL client over http / rest: client(endpoint) plus bearer / basic / header / withCA / insecure builders, then query(c, query, variables) -> json.Value (POSTs {query, variables}; result under /data). Handles the GraphQL error convention - a non-empty errors array is an HTTP 200, not a non-2xx - raising a graphql error with the joined messages; a non-2xx status also raises. tryQuery returns the envelope without raising (inspect via hasErrors / errorMessages + json accessors for extensions.code); queryNamed / tryQueryNamed add an operationName for multi-operation documents. Query is opaque; a mutation is just a query string.
html	import "html.j";	full	build an HTML element tree and render escaped HTML5. element / text / raw / attr / boolAttr (valueless boolean attribute, renders disabled not disabled="") constructors, render / renderAll, escape; void-element aware. Plus a tolerant parse -> the same Node tree (void / self-closing / unquoted-attr / mismatched-nesting / comment / script-raw tolerant), walked by get / findAll / has (XPath-ish selectors) + attrOf / hasAttr; build and parse round-trip through one model.

http	import "http.j";	no (net)	an HTTP/1.1 client over net (https:// via TLS): method-agnostic request plus get / post / put / patch / delete / head / options -> Response (status, headers, body); Content-Length + chunked framing, header case-insensitive lookup; requestRawBody / requestRawBodyTls send a raw bytes body byte-for-byte (multipart file upload). Redirects returned, not followed.
ical	import "ical.j";	full	iCalendar (RFC 5545) build and parse: a Calendar of Events encoded to a VCALENDAR of VEVENTs and parsed back. calendar / event / describe / locate / add value-semantic builders, encode / parse. DTSTAMP / DTSTART / DTEND go through time (UTC DATE-TIME); text values RFC 5545-escaped, long lines folded, so parse(encode(cal)) round-trips. Pure text over strings / lists + time; both binaries. VEVENT-only (no RRULE / VALARM / TZID).
idna	import "idna.j";	full	internationalized domain names: toAscii / toUnicode over a Punycode (RFC 3492) core (münchen.de <-> xn--mnchen-3ya.de), plus isAscii. Used by the mail clients for hosts and envelope domains.
imap	import "imap.j";	no (net)	IMAP4rev1 client over net - read and manage folders: connect / folders (LIST) / status / selectFolder / search (filtered) / fetch / append / flag / copy / expunge / logout, plus fetchAll; messages parsed by mime. RFC 2177 IDLE push (idle -> receiveNotification / pollNotification -> done, typed EXISTS / EXPUNGE / RECENT). A practical subset, not read-only.
influxdb	import "influxdb.j";	no (net)	an InfluxDB time-series client on http, both 1.x and 2.x / 3.x (a Version enum on the Client; the line protocol is shared): build line-protocol points with value-semantic builders (point / tag / field / intField / stringField / boolField / at, mixed field types via pre-rendered fragments), line / write them (/write for 1.x, /api/v2/write?org=&bucket= for 2.x). client / clientWith (1.x, Basic auth) and client2(url, org, bucket, token) (2.x, token auth, redacted from errors); query runs InfluxQL, queryFlux runs Flux over /api/v2/query. Nanosecond precision. Needs the default binary.
ipnet	import "ipnet.j";	full	IP addresses and CIDR networks, IPv4 and IPv6. parseAddress / toString (canonical, RFC 5952 for IPv6) / version / equal; parse(cidr) -> Network (host bits zeroed), contains(net, addr), netmask / broadcast / networkString. Addresses held as raw bytes (4 / 16); bitwise subnet math for allow-lists. Pure .j over strings + convert; both binaries.
jsonl	import "jsonl.j";	full	JSON Lines (JSONL / NDJSON): newline-delimited JSON, one json.Value per line. encode / decode (compact JSON split / joined on \n, blank lines skipped, so decode(encode(rows)) round-trips), whole-file readFile / writeFile / appendFile, and streaming Reader / Writer handles over an open fs.File (openReader / hasMore / readRecord / closeReader; writer / writeRecord / closeWriter) for files too large to hold in memory. A thin framing layer over json + fs; both binaries.
jsonrpc	import "jsonrpc.j";	no (net)	JSON-RPC 2.0, client and server, over http + json: a Client and call(client, method, params) -> json.Value / notify (params and result are json.Values; an error reply throws Error{kind: "jsonrpc"}), plus a transport-agnostic handle(requestBody) -> replyBody that dispatches to the entry program's func NAME(params) methods by name (via meta.callMain), handling notifications, batches, and the reserved error codes. Needs the default binary.

jwt	import "jwt.j";	partial	JSON Web Tokens (RFC 7519): sign(claims, key, alg) / verify(token, key, alg) / decode(token) / header(token) over a json.Value claims object. Ten algorithms - HMAC HS256/384/512, RSA RS256/384/512, ECDSA ES256/384/512, and EdDSA (Ed25519). verify pins the expected algorithm (rejects algorithm-confusion), enforces exp / nbf, and compares HMACs in constant time; verifyLeeway adds clock-skew tolerance, verifyWithKeys selects the key by the header's kid from a caller-supplied map, and verifyJwks resolves the kid against a JWKS (converting the JWK via crypto.jwkToPem). Over crypto + hash + encoding + json + time. HS* / EdDSA on both binaries; RS* / ES* need the default binary.
kvstore	import "kvstore.j";	full	selectable key/value backend with per-key TTL: one Store (a sum-type enum) is a memcache connection, a redis connection, or the in-process kv library (inProcessStore in-memory / fileStore persisted). Uniform set / get / delete / touch / incrWindow; the shared backend layer under session and ratelimit. The distributed backends need the default binary; the local backend runs on both.
label	import "label.j";	partial	industrial label printing in a build / render / emit pipeline. Build a device-independent Label in millimetres (new / text / barcode / box / image / quantity; barcodes code128 / ean13 / itf / code39 / gs1-128 / datamatrix / qr), render(label, device) to a selectable dialect ("zpl" Zebra, "cab" cab JScript), then emit - send(host, port, rendered) to a printer's raw :9100 port. Build / render run on both binaries; send needs the default binary.
ldap	import "ldap.j";	no (net)	an LDAP v3 client and directory server (RFC 4511) on the asn1 BER codec + net (LDAPS / StartTLS via transport.Security). Client: connect -> Conn, then bind / bindSasl (SCRAM), search (RFC 4515 parseFilter or equals / present / allOf / ... over SCOPE_BASE / ONE / SUB) / searchPaged (AD's paged control), and writes add / modify / delete / modifyDn / passwordModify (RFC 3062); binary values (AD objectGUID) come back base64. Server: a mutable in-memory (directory) or file-backed (openDirectory) directory of entry / group records answers simple bind (userPassword verify) + filtered search - read-only over LDAP but mutable from code (addEntry / modifyEntry / deleteEntry / setAttribute), enough to back an auth portal such as Authelia. Needs the default binary.
log	import "log.j";	partial	leveled, structured logging: a Logger carries a minimum level (debug < info < warn < error), a format (text / logfmt / json), and a sink; debug / info / warn / error / fatal (logs then exits 1) (at for a runtime level) render one record - timestamp, level, message, map of string to string fields - and write it, dropping records below the level. with(logger, fields) returns a child logger stamping persistent fields on every record. Sinks new (stdout) / toStderr / toFile / toSyslog (RFC 5424 over UDP). Console + file work on both binaries; the syslog sink needs the default binary.
markdown	import "markdown.j";	full	render a small CommonMark subset (headings, emphasis, links, lists, code, GFM tables) to HTML (through html) and styled terminal text (through ansi) with toHtml / toAnsi; parse(md) surfaces the document as a Node tree walked like xml / html (typeOf / children / text / attr / get / findAll), and render(doc, format) renders a parsed or hand-built tree; toPdf(md) / toPdfWith(md, opts) / renderPdf(doc, opts) lay the document out to a paginated PDF (through pdf); plus authoring helpers (header / style / link / bullets / numbered / codeBlock / table) and tablePretty to align table source.

mcp	import "mcp.j";	no (net)	Model Context Protocol (stateless JSON-RPC 2.0), server and HTTP client. Build a Server (server + addTool / addResource / addPrompt, each with a json.Value schema); handle(server, requestBody) -> replyBody is the transport-agnostic router (initialize / ping / tools/resources/prompts list + call, an allow-list so only a registered item dispatches), serveStdio(server) runs the stdio transport, and connect(endpoint) + initialize / listTools / callTool / readResource / getPrompt are the client over jsonrpc. Needs the default binary.
memcache	import "memcache.j";	no (net)	a memcached client (classic text protocol) over net: set / add / get / delete / incr / decr / touch, every store with a TTL. For caches, sessions, counters, and locks (a volatile store, not a system of record).
mikrotik	import "mikrotik.j";	no (net)	a MikroTik RouterOS API client over net (8728 / api-ssl 8729): connect (plaintext login, MD5 fallback), talk(s, command, attrs) -> list of map of string to string (each !re row), print read sugar, run for add / set / remove (returns the !done =ret=). .tag-correlated commands and /listen-style server-push streaming (listen / receiveReply / cancel); a bounded read no longer leaks a stale deadline. Sentence-based binary framing (variable-length word codec) hand-built with the bitwise ops; !trap throws. Needs the default binary.
mime	import "mime.j";	full	build and parse MIME messages (RFC 5322 headers, multipart, quoted-printable / base64 transfer encodings, RFC 2047 encoded-words for non-ASCII headers). text / attachment / multipart / encode / parse; the foundation the mail clients build on.
mqtt	import "mqtt.j";	no (net)	an MQTT 3.1.1 pub/sub client over net (mqtts via TLS): connect -> Client, then subscribe / publish / publishBytes (QoS 0), blocking receive and single-threaded poll(client, timeoutMs) (via net.setDeadline), ping, disconnect. QoS-1 publish/subscribe with the PUBACK handshake (publishQos1 / subscribeQos1), retained messages, a CONNECT Last-Will (connectWith), and reconnect session resumption. Binary packet framing built with bitwise ops + bytes.
multipart	import "multipart.j";	full	build and parse multipart/form-data (RFC 7578) - the file-upload counterpart to mime. field / file build Parts, build / buildWith -> Built{contentType, body} (fresh or fixed boundary), parse(contentType, body) -> list of Part; text / isFile read a part. Byte-level delimiter matching so binary bodies round-trip. Pairs with web / http. Pure .j; both binaries.
ntp	import "ntp.j";	no (net)	a simple SNTP network-time client (RFC 4330 / 5905) over UDP: query(host) / queryWith(address, timeoutMs) -> Result (serverTime + clock offset + round-trip delay). Packs / unpacks the 48-byte NTP packet with bytes + bitwise ops and converts the NTP epoch through time; a lost reply times out (not hangs). Query-only (no clock discipline / daemon). Needs the default binary.
oauth	import "oauth.j";	no (net)	a generic OAuth2 client (the get-a-token half) on http + json: Client Credentials, Refresh Token, and Device Authorization grants, google / microsoft presets, expiry + on-disk token store. Tokens feed sasl XOAUTH2 for mail.

orm	import "orm.j";	no (sql)	a relational mapper over the sql library - Data Mapper, not Active Record (structs have no methods). Declare an orm.Schema (schema / column + attribute setters notNull / unique / autoIncrement / withDefault, carrying the "mysql" / "postgres" dialect), then CRUD through an orm.Session (session auto-commit / transaction inside a sql.Tx): insert / find / update / delete and all over a functional query builder (from / where / orderBy / limit / join -> fresh Query -> toSql). Records are map of string to string; values bind only through placeholders (injection-safe). DDL builders (createTable / dropTable / addColumn / createIndex / addForeignKey / ...); relations (belongsTo / hasOne / hasMany / manyToMany + joinRelation) with eager loading (with / load -> Result, N+1-free in a fixed 1+R queries); a write path (upsert / insertMany / insertReturning / updateWhere / deleteWhere / save) and finders (first / exists / findBy / pluck / page). Migrations are the sqlmigrate module. Needs the default binary.
password	import "password.j";	full	generate / validate / score passwords against a policy Schema (classes, length range, per-class minimums, symbol set, exclude-ambiguous). schema + with* builders, generate -> string, validate -> Report{valid, reasons}, complexity -> Strength{length, classes, poolSize, entropy, label} (bits = length * log2(pool)). Crypto-grade RNG (via crypto), so generated passwords are safe as real credentials; pure .j, both binaries.
pdf	import "pdf.j";	full	generate simple PDF documents (text / lines / rectangles) the way html / label generate their formats: document / page / text / line / rect / color / addPage builders, info metadata (+ pdfDate), render() -> bytes writing the PDF object / xref structure by hand with FlateDecode content streams (via compress). Standard-14 fonts; points, 0-255 RGB. Byte-identical (no timestamps), qpdf-clean output - golden-test friendly; both binaries. A writer, not a reader (no embedded fonts / images yet).
plot	import "plot.j";	full	Data plotting to SVG: a unified chart(series, opts) renders one or more Series (line / points / both / filled area, solid or dashed, with error bars and marker shapes) on shared axes with a positioned legend; line / scatter / bar / bars (grouped or stacked multi-series, negative/diverging bars, optional data labels) / histogram are wrappers. Automatic "nice" ticks, a gridded frame, title, and labels, plus log scales, a date axis (x = Unix seconds via time), reference lines (hline / vline), fonts + margins, native <title> hover tooltips, and save(svg, path). The visual companion to the stats / ml numeric stack, pure .j over math / time / fs / strings / lists / convert, both binaries.
pop	import "pop.j";	no (net)	receive mail (POP3 client) over net: plaintext / STLS / implicit TLS, USER / PASS. connect / stat / sizes / retrieve / deleteMessage / quit, plus fetchAll; messages parsed by mime.
prometheus	import "prometheus.j";	partial	metrics in two halves. Exposition (counter / gauge / histogram / summary / observe / observeAt / render) builds a metric set and renders the Prometheus text format - histograms emit cumulative _bucket / _sum / _count, summaries emit quantile lines, an optional per-sample timestamp, plus pushgatewayPath for a Pushgateway grouping key. Pure text, runs on both binaries. Retrieval (query / queryRange -> Result) is a read client for the HTTP query API over http + json, so it needs the default binary. Strict name / label validation and escaping.
ratelimit	import "ratelimit.j";	no (net)	a fixed-window rate limiter on memcache (atomic incr + per-key TTL): allow(mc, key, limit, window) -> bool, remaining(mc, key, limit). The window resets on its own when it expires.

redis	import "redis.j";	no (net)	a Redis client speaking RESP2 over net: commands as RESP arrays, replies parsed into a Reply. Typed helpers get / set / del / exists / incr / keys / ping, plus a generic command for the rest. RESP2 pub/sub (subscribe / psubscribe / publish / blocking receiveMessage), one-round-trip pipeline, multi / exec / discard transactions, and a production-safe scan cursor (keys now flagged production-unsafe).
resque	import "resque.j";	no (net)	background jobs on Redis, wire-compatible with Resque: enqueue onto named queues, reserve from a worker in priority order (Job = queue / class / args), queueLength / queues / size / fail. Interops with Ruby-resque / php-resque workers. Built on redis + json.
rest	import "rest.j";	no (net)	an ergonomic REST layer over http + json: a value-semantic Client (base URL + headers) and get / post / put / patch / delete (+ getJson / postJson / ...). Base-URL joining, query strings, Bearer / Basic auth.
ringbuffer	import "ringbuffer.j";	full	a fixed-capacity ring buffer (bounded FIFO of strings, overwrite-oldest when full): new(capacity), push / pop, first / last peek, size / capacity / isEmpty / isFull / toList. A sliding window of recent items. Value-semantic. Over lists; both binaries.
s3	import "s3.j";	no (net)	S3-compatible object storage (AWS S3 / MinIO / R2 / B2), AWS Signature V4-signed: connect -> Client, then get / put / delete / listObjects (+ objectKeys). Path-style; configurable endpoint. Over hash.hmac + http + time.
sasl	import "sasl.j";	full	SASL auth encoders shared by the mail clients: plain / loginUser / loginPass / bearer (XOAUTH2, the "use a token" half of OAuth2). Pure base64, no networking.
transport	import "transport.j";	full	the shared connection-security mode for every socket client (smtp / pop / imap / redis / amqp / mqtt): one Security enum (None / Tls / Starttls) instead of a stringly-typed security field per module, plus encrypted(s). No networking; both binaries.
screen	import "screen.j";	partial	terminal user interfaces (an explicit screen, not a GUI). Output-only layer (both binaries): a cell Buffer you draw into (newScreen / set / text / textColor / box / fill / hline / vline), ANSI control strings (clear / moveTo / hideCursor / enterAlt / ...), and a flicker-free render / diff paint loop. Interactive layer (needs term, default binary): a pure decodeKey (arrows / nav / function / ctrl / alt keys) plus nextKey / begin / end / size over raw mode. 0-based coords, clipped drawing.
semver	import "semver.j";	full	strict Semantic Versioning 2.0.0, package-registry-grade. parse / isValid / toString, compare / lt / lte / eq / neq / gt / gte / diff, isStable / isPrerelease, inc*, sort / rsort; coerce / clean for loose tags; and npm/Composer range matching - satisfies (caret / tilde / comparators / || / hyphen / x-ranges, prerelease-aware), maxSatisfying / minSatisfying / minVersion / validRange, plus solver algebra intersects / subset / gtr / ltr / outside / simplifyRange (prerelease-precise). Struct Version.
session	import "session.j";	no (net)	server-side sessions on memcache: a map of string to string under sess:ID with a sliding TTL. create / load / save / touch / destroy; UUID v4 IDs, base64-wrapped JSON values. Volatile (a cache, not a store of record).
slack	import "slack.j";	no (net)	post to a Slack Incoming Webhook on http: send(webhookUrl, text) for a plain message, or build a Block Kit message with message / text / section / header / divider, plus contextBlock / fieldsSection / actionsBlock (button) blocks, and post it with sendMessage. render gives the JSON payload; strings JSON-escaped. Sibling of discord / gotify. Needs the default binary.
smtp	import "smtp.j";	no (net)	send mail (SMTP client) over net: plaintext / STARTTLS / implicit TLS, AUTH PLAIN, MAIL FROM / RCPT TO / DATA. smtp.send(opts, from, recipients, message); message built by mime.

snmp	import "snmp.j";	no (net)	an SNMP v1 / v2c client and agent over UDP, on asn1 + net. Client: client / clientWith -> Client, then get / getNext / set / subtree walk return a list of Varbind {oid, type, value, number} (typed by SNMP value type - integer / octetString / oid / counter32 / timeTicks / ipAddress / ...). Agent (server / hardware simulator): agent(community, version, bindings) + serve / serveOn answer GET / GETNEXT / SET for a MIB. Community-string auth; intVar / stringVar / oidVar / varbind build bindings. No SNMPv3 / traps. Needs the default binary.
sqlmigrate	import "sqlmigrate.j";	no (sql)	version-tracked schema migrations over the sql library, decoupled from orm: a Migration{version, description, up, down} whose up / down are plain DDL strings (from orm's DDL helpers or hand-written). migrate(conn, migrations) applies pending versions in lexical order (each in its own transaction, recorded in schema_migrations; idempotent), rollbackMigrations(conn, migrations, steps) reverses the newest N, migrationStatus reports applied / pending. Version allowlisted + description escaped (injection-safe). Needs the default binary.
statsd	import "statsd.j";	no (net)	a fire-and-forget StatsD metrics client over UDP: client / clientWith -> Client (agent address + optional name prefix), then count / increment / decrement (counter c), gauge (g), timing (ms), set (s) each emit one metric:value|type datagram. The push counterpart to a pull-based scrape; no reply, no error when no agent listens. Extensions: countRate / timingRate (|@rate), *Tagged verbs (DogStatsD |#k:v tags), countFloat / gaugeFloat, and a value-semantic Batch (batch / add* / flush) packing several metrics into one datagram; every line control-character validated. Needs the default binary.
telegram	import "telegram.j";	no (net)	a Telegram Bot API client on http + json: bot / botWith -> Bot, getMe, sendMessage / sendMessageWith (parse mode) / sendPhoto / sendChatAction -> Message / bool, and getUpdates(bot, offset, timeout) long-poll -> list of Update for a stateful receive loop. Inline keyboards (sendMessageWithKeyboard), callback_query handling (parseCallbackQuery / answerCallbackQuery), local-file upload (sendPhotoFile / sendDocumentFile), and bot-token redaction from errors. Form-encoded params, {"ok":false} throws. Needs the default binary.
tengine	import "tengine.j";	full	a lightweight-CMS text template engine (a subset of Go text/template) over a json.Value tree: newSet / add / render. .path / $ root / $var, if / else if with eq / and / or / not, range (with $i, $e) / with / block, {{ $x := }} variables, define / template layout inheritance, {{- -}} trim markers, and pipes upper / lower / title / trim / html / urlize / default / truncate / join / len / printf.
totp	import "totp.j";	full	time-based one-time passwords (RFC 6238 / 4226): generate / verify (+/-1-step skew), generateAt / verifyAt (explicit time), and verifyWindow (configurable skew), plus crypto-grade generateSecret / generateSecretN, an exported RFC 4226 hotp, and uri for the otpauth:// provisioning string. base32 secrets; SHA-1 / SHA-256 / SHA-512. Over hash.hmac + encoding + time.
uri	import "uri.j";	full	URL / URI parsing, building, and query-string handling (RFC 3986): parse -> Uri (scheme / user / host / port / path / query / fragment) and build back; encode / decode (RFC 3986 percent-encoding) and encodeForm / decodeForm (application/x-www-form-urlencoded, space as +); buildQuery / parseQuery between a map of string to string and a query string; and resolve for RFC 3986 relative-reference resolution. Pure .j over strings + encoding; both binaries. The shared URL layer the network modules build on.

validate	import "validate.j";	full	declarative data validation: check a map of string to string (a form body, query, or config) against a rule set and get a structured Failure list. Rules compose per field as value-semantic descriptors - required / isInt / isFloat / isBool / min / max / minLen / maxLen / pattern / email / url / datetime / oneOf / noneOf (blacklist) / password (a password.Schema policy) / custom (a func predicate) / withMessage; check -> list of Failure ({field, rule, param, message}), ok short-circuits, messages / byField render, localize(errs, templates) re-messages for i18n (rule-id -> template, {param} / {field}). An absent / blank field passes all but required. Over regex + uri + time + password + convert; both binaries.
vcard	import "vcard.j";	full	vCard (RFC 6350, vCard 4.0) contacts build and parse: a Card of contact fields encoded to a VCARD and parsed back. card / withName / withOrg / addEmail / addPhone / address / addAddress / withUrl / withNote value-semantic builders, encode / encodeAll / parse (one or many cards). Structured N / ADR / ORG, RFC 6350 text escaping and 75-char line folding - shares the content-line codec with ical. Pure text over strings / lists; both binaries. A contact subset (no BDAY / PHOTO / parameter round-trip).
web	import "web.j";	no (net)	a small HTTP framework over the httpd engine: register routes against handler func values (web.get / post / ...), :param capture, middleware, web.Context request / response helpers; web.run owns the accept loop. Handlers dispatch as func values called in their home context. Pairs with jennifer serve.
webapi	import "webapi.j";	no (net)	a JSON-API conventions layer over web: a uniform error envelope, request validation (reusing validate), versioned route mounting (mount / alias / deprecate), pluggable bearer auth + rate limiting (entry-program func values, like route handlers), content negotiation, and pagination. A value-semantic Api builder finished with install, enforced by one before guard shim; the Spec evaluation (evaluate) is a pure, testable core. Auth / Produces enums. Over web + validate + json; default binary only.
webhook	import "webhook.j";	full (send net)	HMAC-signed webhooks (GitHub X-Hub-Signature-256): sign(payload, secret) / verify(payload, signature, secret) are pure (both binaries); plus replay-protected timestamped signing schemes - Stripe (stripeSign / stripeVerify), Slack (slackSign / slackVerify), and a generic digest/encoding variant (timestampedSign / timestampedVerify); send(url, payload, secret) POSTs the signed body via http (default binary). Over hash.hmac + encoding (hex).
websocket	import "websocket.j";	no (net)	an RFC 6455 WebSocket client over net (ws:// / wss://): connect / connectWith do the HTTP Upgrade handshake (verifying the SHA-1 + base64 Sec-WebSocket-Accept), then send / sendBytes (masked frames) and receive -> Message (auto-pong, fragment reassembly), ping / close. Binary framing + masking with the bitwise ops over hash + encoding + math. Needs the default binary.

Writing your own

A module is a declarations-only file: its top level permits only `def const` , `def struct` , `func` , `use` , and `import` - no mutable module state and no free-standing statements. Prefix a top-level `func` / `def struct` / `def const` with `export` to publish it; unmarked names stay module-private. Each file states its own `use` imports (`use` is not transitive across a module boundary).

Every module that ships in this repository carries a co-located white-box test overlay (`NAME_test.j`) run with `jennifer test` , and a runnable demo under `examples/modules/` . See `modules/README.md` for the contributor checklist.

See also

- Imports guide - `use` vs `include` vs `import` , resolution rules, and the module boundary in depth.
- Libraries catalog - the Go system libraries a module builds on.

acme - ACME / Let's Encrypt TLS certificates

Import with `import "acme.j" as acme;` . An ACME (https://www.rfc-editor.org/rfc/rfc8555) (RFC 8555) client: obtain and renew TLS certificates from Let's Encrypt and compatible CAs. It drives the whole flow - account registration, a new order, an **HTTP-01** or **DNS-01** challenge, CSR finalize, and certificate download - over `http` + `json` , with every request a JWS signed by the account key (`RS256` for an RSA key; `ES256` / `ES384` / `ES512` for an EC key, by curve). Keys, the CSR, and the JWK thumbprint come from `crypto` , so `acme` needs the default `jennifer` binary.

> **Use a staging endpoint first.** Let's Encrypt's staging CA (`https://acme-staging-v02.api.letsencrypt.org/directory`) issues *untrusted* certificates with far higher rate limits - develop against it, then switch to production.

Runnable: `examples/modules/acme_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/acme_demo.j).

The flow

Proving you control a domain is **your** job, so the steps are separate calls, not one `obtain()` . The order is: register an account, create an order, and for each domain fetch its authorization, pick a challenge, **publish the response** , `accept` , and poll until valid; then finalize with a CSR and download the cert.

```
use crypto;
import "acme.j" as acme;

def key as bytes init crypto.ecGenerateKey("p256");                    # account
key
def client as acme.Client init acme.connect(DIRECTORY_URL, $key);
def client as acme.Client init acme.register($client, "you@example.com");

def order as acme.Order init acme.order($client, ["example.com"]);
def authz as acme.Authorization init acme.authorization($client, $order.
authorizations[0]);
def ch as acme.Challenge init acme.challenge($authz, "http-01");

# --- publish the challenge response (see below), then: ---
acme.accept($client, $ch.url);
acme.pollAuthorization($client, $order.authorizations[0], 2000, 30);

def certKey as bytes init crypto.ecGenerateKey("p256");                #
certificate key
def csr as bytes init crypto.csr($certKey, ["example.com"]);
def issued as acme.Order init acme.finalize($client, $order, $csr, 2000, 30);
def pem as string init acme.downloadCertificate($client, $issued);     # the cert
chain
```

Publishing the challenge response

For a challenge `ch` , compute the response and publish it, then `accept` :

- **HTTP-01** : serve `acme.keyAuthorization($client, $ch.token)` as the body at `http://<domain>/.well-known/acme-challenge/<ch.token>` (pair with `web` / `httpd`).
- **DNS-01** : publish `acme.dnsRecord($client, $ch.token)` as a TXT record at `_acme-challenge.<domain>` (via your DNS provider). DNS-01 is the only way to get a **wildcard** (`*.example.com`) certificate.

Both are pure (no network) - compute them, provision the response your own way, then tell the CA it is ready

with `accept` .

Surface

Call	Returns	Notes
acme.connect(directoryUrl, accountKey)	Client	Fetch the CA directory; build a client around the account key (PEM bytes).
acme.register(client, email)	Client	Register the account (agrees to the ToS); returns a client with its kid.
acme.order(client, domains)	Order	Create an order for a list of string of domains.
acme.authorization(client, authzUrl)	Authorization	Fetch an authorization and its challenges.
acme.challenge(authz, kind)	Challenge	The "http-01" / "dns-01" challenge within an authorization.
acme.keyAuthorization(client, token)	string	The HTTP-01 response body (token.thumbprint). Pure.
acme.dnsRecord(client, token)	string	The DNS-01 TXT value. Pure.
acme.accept(client, challengeUrl)	null	Tell the CA the response is published.
acme.pollAuthorization(client, authzUrl, intervalMs, maxTries)	Authorization	Poll until the authorization settles.
acme.finalize(client, order, csrDer, intervalMs, maxTries)	Order	Submit the CSR and poll until the certificate is issued.
acme.downloadCertificate(client, order)	string	The issued PEM certificate chain.
acme.fetchOrder(client, orderUrl)	Order	Re-fetch an order's state.

The structs (`Client` , `Order` , `Authorization` , `Challenge`) are value-semantic; `register` returns a client with the account `kid` filled in. `Client` holds the account key and the CA endpoints. Every signed request draws a fresh anti-replay nonce from the CA before it is sent.

```
def struct acme.Order { url as string, status as string, authorizations as list
of string, finalize as string, certificate as string };
def struct acme.Authorization { domain as string, status as string, challenges as
list of Challenge };
def struct acme.Challenge { kind as string, url as string, token as string,
status as string };
```

An `Order` 's `status` moves `pending` -> `ready` -> `processing` -> `valid` , its `authorizations` are the per-domain authorization URLs, `finalize` takes the CSR, and `certificate` is the download URL once issued. An `Authorization` carries the `domain` , its `status` , and the offered `challenges` ; each `Challenge` carries its `kind` (`http-01` / `dns-01` / `tls-alpn-01`), the `url` to `accept` , the `token` , and its `status` .

Keys and algorithms

- **Account key** : `crypto.ecGenerateKey("p256")` (-> `ES256`) or `crypto.rsaGenerateKey(2048)` (-> `RS256`). `connect` derives the JWS algorithm from the key, binding each EC curve to its JOSE algorithm - P-256 -> `ES256` , P-384 -> `ES384` , P-521 -> `ES512` (a curve must not be signed under another curve's algorithm). `ES256` (the P-256 default) is the most widely accepted; check your CA supports a curve before using P-384 / P-521. Reuse the *same* account key across renewals.
- **Certificate key** : a **separate** key, only used for the CSR (`crypto.csr(certKey, domains)`). Generate a fresh one, or reuse per domain.

Errors from the CA (an RFC 7807 problem document) surface as catchable `Error{kind: "acme"}` .

Renewal

Renewing is just running the flow again before expiry (Let's Encrypt certs last 90 days; renew at ~60). Keep the account key; a new order re-validates the domains and issues a fresh certificate.

Platform

`acme` needs `net` (via `http`) and the `crypto` RSA / ECDSA / CSR / JWK surface, so it runs on the **default jennifer** binary only; on `jennifer-tiny` the crypto and network calls raise friendly errors.

amqp - AMQP 0-9-1 client (RabbitMQ)

Import with `import "amqp.j" as amqp;` . A client for RabbitMQ and compatible AMQP 0-9-1 brokers over `net` : connect, declare a queue, publish messages, and pull them back. The binary frame and method encoding is built by hand from `bytes` and the bitwise operators - the largest protocol module in the library. Needs the default `jennifer` binary. A protocol error or dropped connection throws `Error{kind: "amqp"}` .

```
import "amqp.j" as amqp;

def c as amqp.Conn init amqp.connect(amqp.options("localhost", "guest", "guest"));
amqp.declareQueue($c, "jobs", true);
amqp.publishText($c, "", "jobs", "hello");

def m as amqp.Message init amqp.get($c, "jobs", false);
if (not $m.empty) {
    amqp.ack($c, $m.deliveryTag);
}
amqp.close($c);
```

Runnable: `examples/modules/amqp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/amqp_demo.j).

Connecting

`connect` runs the full handshake - protocol header, `Connection.Start` / `Start-Ok` (SASL **PLAIN** auth), `Tune` / `Tune-Ok` (heartbeats disabled), `Open` / `Open-Ok` , then `Channel.Open` - and returns a `Conn` on a single channel.

The dial is bounded by a connection-establishment timeout, so a slow or unreachable broker fails with a catchable error instead of blocking forever.

```
def struct amqp.Options { host as string, port as int, user as string, password
as string, vhost as string, security as transport.Security };
```

`security` is a `transport.Security` : `.None` (plaintext AMQP, the default) or `.Tls` (AMQPS - TLS on connect, verifying the broker certificate); `.Starttls` is rejected (AMQP has no in-band upgrade). `amqp.options(...)` defaults it to `.None` ; set it to `transport.Security.Tls` on the returned `Options` for a broker that requires TLS so credentials do not cross the wire in the clear.

Call	Returns	
amqp.options(host, user, password)	Options	defaults: port 5672, vhost "/", security "none"
amqp.withPort(o, port)	Options	copy with a different port
amqp.withVhost(o, vhost)	Options	copy with a different virtual host
amqp.connect(opts)	Conn	connect and open a channel
amqp.close(c)		Connection.Close and shut the socket

Queues and publishing

Call	Returns	
amqp.declareQueue(c, name, durable)	QueueInfo	declare a classic queue ("" name = server-generated); durable survives a restart
amqp.declareQuorumQueue(c, name)	QueueInfo	declare a quorum queue (x-queue-type=quorum); always durable, name required
amqp.publish(c, exchange, routingKey, body)		publish a bytes body

amqp.publishText(c, exchange, routingKey, text)		publish a UTF-8 string
amqp.publishWith(c, exchange, routingKey, body, props)		publish carrying message properties

`declareQueue` returns `QueueInfo{name, messageCount, consumerCount}` . `declareQuorumQueue` declares a **quorum queue** - a Raft-replicated, always-durable queue type for high availability - by sending the `x-queue-type=quorum` argument. Quorum queues cannot be server-named, so `name` must be non-empty, and there is no `durable` flag (they are always durable). Do not also declare the same name as a classic queue: a re-declare must use the same type. `publish` sends the method frame, a content-header frame (body size), and a body frame. Use exchange `""` (the default exchange) to route straight to a queue by name via `routingKey` .

Exchanges and bindings

Beyond the default exchange, declare a named exchange and bind a queue to it. The exchange type selects the routing rule: `"direct"` (exact routing-key match), `"topic"` (pattern match with `*` / `#`), or `"fanout"` (broadcast, routing key ignored). Declared exchanges are durable.

```
amqp.declareExchange($c, "events", "topic");
amqp.declareQueue($c, "audit", true);
amqp.bindQueue($c, "audit", "events", "order.#");
amqp.publishText($c, "events", "order.created", "{}");
```

Call	Returns	
amqp.declareExchange(c, name, exType)		declare a durable exchange ("direct" / "topic" / "fanout")
amqp.bindQueue(c, queue, exchange, routingKey)		bind a queue to an exchange with a routing key / pattern

Message properties

`publishWith` attaches message properties to a publish. Build a `Properties` and leave any field unset (`""` or `false`) to omit it from the wire:

```
def struct amqp.Properties {
    contentType as string,    # MIME type, e.g. "application/json"; "" omits it
    persistent as bool,       # true = delivery-mode 2 (survives a restart on a
durable queue)
    correlationId as string,  # request / reply matching; "" omits it
    replyTo as string         # reply-to queue / routing key; "" omits it
};
```

```
def p as amqp.Properties init amqp.Properties{
    contentType: "application/json", persistent: true, correlationId: "req-42",
replyTo: "replies"
};
amqp.publishWith($c, "", "jobs", convert.bytesFromString("{}", "utf-8"), $p);
```

`persistent` is the message-level durability flag (delivery-mode 2); combine it with a `durable` queue so a message survives a broker restart.

Consuming (pull)

`amqp.get(c, queue, autoAck)` pulls the next message with `Basic.Get` - a **synchronous pull** , not an async delivery loop. Call it in a loop until `Message.empty` is true; `ack` each message (unless `autoAck`).

```
def struct amqp.Message {
    empty as bool,        # true when the queue was empty (other fields zero)
    deliveryTag as int,   # pass to ack
    exchange as string,
    routingKey as string,
    body as bytes
};
```

```
def more as bool init true;
repeat {
    def m as amqp.Message init amqp.get($c, "jobs", false);
    if ($m.empty) {
        $more = false;
    } else {
        # handle $m.body
        amqp.ack($c, $m.deliveryTag);
    }
} until (not $more);
```

Call	Returns	
amqp.get(c, queue, autoAck)	Message	pull the next message (empty true when none)
amqp.ack(c, deliveryTag)		acknowledge a delivered message
amqp.nack(c, deliveryTag, requeue)		negatively acknowledge (requeue true re-queues, false drops / dead-letters)

`nack` (Basic.Nack) rejects a delivered message: with `requeue` true the broker puts it back for redelivery, with false it is dropped (or dead-lettered if the queue is configured for it). It works for messages from both `get` and `receiveDelivery` .

Consuming (push)

`amqp.consume(c, queue, autoAck)` starts a **server-pushed** subscription (`Basic.Consume`) and returns a broker-generated **consumer tag** . The broker then streams messages; `amqp.receiveDelivery(c)` **blocks** for the next one (`Basic.Deliver` + content-header + body frames) and returns a `Delivery` . There is no callback API - the app calls `receiveDelivery` in a loop.

```
def struct amqp.Delivery {
    consumerTag as string,
    deliveryTag as int,    # pass to ack / nack
    redelivered as bool,   # the broker has delivered this before
    exchange as string,
    routingKey as string,
    body as bytes
};
```

Because `receiveDelivery` blocks, run the loop inside its own `spawn` so pushed deliveries never stall the rest of the program (this is the cooperative receive-loop convention - no handler callbacks re-enter the interpreter):

```
def tag as string init amqp.consume($c, "jobs", false);

def worker as task of null init spawn {
    def running as bool init true;
    while ($running) {
        def d as amqp.Delivery init amqp.receiveDelivery($c);
        # handle $d.body ...
        amqp.ack($c, $d.deliveryTag);
    }
    return;
};

# ... main program does other work; cancel when done ...
amqp.cancelConsume($c, $tag);
```

Use one `Conn` per consuming `spawn` : a `Conn` holds a single socket + channel, and value semantics copy the handle but share the underlying `net.Conn` , so two tasks reading the same connection would interleave frames. Open a second connection for a concurrent consumer.

Call	Returns	
amqp.consume(c, queue, autoAck)	string	start a subscription; returns the consumer tag
amqp.receiveDelivery(c)	Delivery	block for the next pushed message
amqp.cancelConsume(c, consumerTag)		stop a subscription

Publisher confirms

`amqp.confirmSelect(c)` puts the channel into **publisher-confirm** mode (`Confirm.Select`). After that, the broker confirms each publish; `amqp.waitConfirm(c)` blocks for the next confirmation, returning `true` on `Basic.Ack` (the broker took responsibility for the message) and `false` on `Basic.Nack` (it could not). Call `waitConfirm` once per publish.

```
amqp.confirmSelect($c);
amqp.publishText($c, "", "jobs", "hello");
if (not amqp.waitConfirm($c)) {
    # broker could not accept the message - retry or fail
}
```

Call	Returns	
amqp.confirmSelect(c)		enable publisher confirms on the channel
amqp.waitConfirm(c)	bool	block for the next confirm (true ack / false nack)

Scope

- **Pull and push.** Receiving is either `Basic.Get` (one message per `get` call) or `Basic.Consume` + `receiveDelivery` (a server-pushed loop, run inside a `spawn`).
- **One channel.** A single channel (1) is opened per connection. For concurrent consumers open a second `Conn` .
- **Publisher confirms, no transactions.** `confirmSelect` + `waitConfirm` give per-message broker acknowledgement; AMQP transactions (`Tx`) are not implemented.
- **Message properties.** `publishWith` carries content-type, persistence (delivery-mode 2), correlation-id, and reply-to; other properties (headers, priority, timestamp, ...) are not exposed.
- **SASL PLAIN only.** TLS (`amqps`) is available via `Options.security` = `"tls"` ; without it, use a trusted network or a local broker.
- **The largest protocol module.** If the tree-walker ever becomes the bottleneck for high-throughput

messaging, this is a candidate to reimplement as a Go library.

See also

- net.md - the TCP transport this is built on.
- mqtt.md / redis.md - the other binary-protocol clients over `net`.
- modules/index.md - the module catalog and import rules.

ansi - terminal styling

Import with `import "ansi.j" as ansi;` . Wraps a string in ANSI SGR escape codes for colour, background colour, text style, and 24-bit truecolor - and strips them back off. Pure Jennifer (no Go), so it runs on either binary.

Styling is **TTY-aware** : it suppresses itself when stdout is not a terminal (redirected to a file or a pipe), so the wrapped text stays clean either way. The `NO_COLOR` (https://no-color.org) / `FORCE_COLOR` environment convention overrides the gate.

```
use io;
import "ansi.j" as ansi;

io.printf("%s\n", ansi.bold(ansi.red("error:")) + " something broke");
io.printf("%s\n", ansi.green("ok") + " / " + ansi.yellow("warn"));
io.printf("%s\n", ansi.rgb("truecolor orange", 255, 128, 0));
io.printf("%s\n", ansi.underline(ansi.cyan("nested + underlined")));
```

Runnable: `examples/modules/ansi_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/ansi_demo.j).

Surface

Call	Returns	Notes
ansi.color(s, name)	string	Wrap s in the named foreground colour. Unknown name throws.
ansi.bgColor(s, name)	string	Wrap s in the named background colour.
ansi.style(s, name)	string	Wrap s in a text style: bold / dim / italic / underline / reverse.
ansi.rgb(s, r, g, b)	string	24-bit truecolor foreground; each channel 0-255.
ansi.strip(s)	string	Remove every SGR escape - the inverse of the wrappers.

Wrapping composes and nests (`ansi.bold(ansi.red(s))`); each wrapper emits its own code and a reset, so an inner reset never truncates an outer style.

Colour and style names

- **Foreground / background** (`color` / `bgColor`): `black` , `red` , `green` , `yellow` , `blue` , `magenta` , `cyan` , `white` . `color` also accepts the bright `gray` (alias `grey`).
- **Styles** (`style`): `bold` , `dim` , `italic` , `underline` , `reverse` .

An unrecognized name is a thrown `Error` (`kind: "value"`), catchable with `try` / `catch` .

Shortcuts

Each common colour and style has a one-argument shortcut - `ansi.NAME(s)` is exactly `ansi.color(s, "NAME")` (or `ansi.style` for a style name):

- Colours: `black` , `red` , `green` , `yellow` , `blue` , `magenta` , `cyan` , `white` , `gray` .
- Styles: `bold` , `dim` , `italic` , `underline` , `reverse` .

When styling is emitted

`ansi` decides per call (it is stateless - there is no toggle to store):

1. `NO_COLOR` set (to anything) - **off** , always.
2. else `FORCE_COLOR` set - **on** , always.
3. else on when `os.isTerminal("stdout")` is true; off when it is false.
4. If the host cannot tell whether stdout is a terminal, defaults **on** .

`strip` ignores this gate - it removes escapes unconditionally, so it cleans up already-styled text (or a captured subprocess's output) regardless of the current terminal state.

```
def styled as string init ansi.bold(ansi.blue("styled"));
io.printf("stripped: [%s]\n", ansi.strip($styled));   # stripped: [styled]
```

See also

- os.md - `os.isTerminal` , the TTY gate, and `os.getEnv` for the `NO_COLOR` / `FORCE_COLOR` reads.
- regex.md - `strip` uses `regex.replace` to drop escape sequences.
- modules/index.md - the module catalog and import rules.

args - command-line argument parser

Import with `import "args.j" as args;` . A declarative CLI parser at the common surface of Python's `argparse` , plus the features an argparse user misses immediately - typed flags (long + short), defaults, `required` , `choices` , `count` / `append` actions, positionals with `nargs` , subcommands, and `--version` . It is the structured layer over `os.ARGS` (`os.hasFlag` / `os.flag` are primitive lookups). Pure Jennifer over `strings` + `convert` + `lists` + `maps` , so it runs on **both binaries** .

```
use io;
use os;
import "args.j" as args;

def p as args.Parser init args.parser("greet", "Say hello");
$p = args.flag($p, "name", "n", "world", "who to greet");
$p = args.countFlag($p, "verbose", "v", "verbosity (repeatable)");
$p = args.positionalList($p, "extra", "extra names");

def r as args.Result init args.parse($p, os.ARGS);
if ($r.done) { io.printf("%s\n", $r.helpText); exit 0; }   # --help / --version
io.printf("hello %s (v=%d)\n", args.asString($r, "name"), args.count($r,
"verbose"));
```

Runnable: `examples/modules/args_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/args_demo.j)

The parse model

A `Parser` is a **value-semantic** spec built with copy-returning builders (the same pattern as `rest.Client`): each builder returns an updated `Parser` , so you chain assignments (`$p = args.flag($p, ...)`). `args.parse($p, argv)` walks the argument vector and returns a `Result` .

- `argv` is the full `os.ARGS` ; its first element is the program name and is **skipped** (like `sys.argv[1:]` / `os.Args[1:]`).
- **Errors are catchable** , not a process `exit(2)` : an unknown flag, a missing required arg, a bad-type value, or a `choices` violation throws `Error{kind: "args"}` , so parsing composes with `try` / `catch` .
- **-h / --help (and --version) do not throw** : they set `$r.done = true` and put the text in `$r.helpText` . Print it and stop: `if ($r.done) { io.printf("%s\n", $r.helpText); exit 0; }` .

Building a parser

Builder	Adds
args.parser(prog, help)	a new empty parser
args.flag(p, long, short, deflt, help)	a string flag (--long / -short) with a default
args.intFlag(p, long, short, deflt, help)	an int flag
args.floatFlag(p, long, short, deflt, help)	a float flag
args.boolFlag(p, long, short, help)	a boolean flag (presence -> true; store_true)
args.countFlag(p, long, short, help)	a repeatable flag counting occurrences (-vvv -> 3)
args.listFlag(p, long, short, help)	a repeatable flag collecting values into a list (append)
args.positional(p, name, help)	one required positional
args.positionalOpt(p, name, deflt, help)	an optional positional (nargs "?")
args.positionalList(p, name, help)	a variadic positional, 0+ (nargs "*")
args.positionalList1(p, name, help)	a variadic positional, 1+ (nargs "+")

args.positionalN(p, name, n, help)	a positional taking exactly n values (nargs N)
args.required(p)	mark the most-recently-added argument required
args.choices(p, allowed)	constrain the most-recently-added argument to a set
args.command(p, name, help, sub)	add a subcommand with its own Parser
args.version(p, ver)	enable --version, printing ver

`required` and `choices` are **post-modifiers** on the last-added argument, so they read fluently right after the builder:

```
$p = args.choices(args.required(args.flag($p, "mode", "m", "", "build mode")),
["debug", "release"]);
```

Reading the result

Accessor	Returns	For
args.asString(r, name)	string	a string flag / positional (default if unset)
args.asInt(r, name)	int	an int flag
args.asFloat(r, name)	float	a float flag
args.asBool(r, name)	bool	a boolean flag
args.asList(r, name)	list of string	an append flag or variadic positional
args.count(r, name)	int	a count flag
args.has(r, name)	bool	whether the arg was supplied (vs defaulted)
args.usage(p)	string	the generated help text (what --help prints)

The `Result` also exposes fields directly: `$r.command` (the chosen subcommand, `""` if none), `$r.done` , and `$r.helpText` .

Dispatching subcommands

`args.dispatch(r, handlers)` routes the selected subcommand to its handler. The `handlers` map takes each subcommand name to a `func` value `func(r as Result)` ; the handler for `$r.command` is called with the whole `Result` (it reads its own args with the accessors above) and its return value is passed back, so a handler can return an exit code. A `Result` whose `done` is set (a handled `--help` / `--version`) dispatches nothing and returns `null` - check `$r.done` and print `$r.helpText` first, as usual. A selection of no subcommand, or one with no matching handler, is a catchable `Error{kind: "args"}` .

```
func cmdAdd(r as args.Result) { store.add(args.asString($r, "item")); return 0; }
func cmdList(r as args.Result) { io.printf("%s\n", store.all()); return 0; }

def r as args.Result init args.parse($p, os.ARGS);
if ($r.done) { io.printf("%s\n", $r.helpText); exit 0; }
exit args.dispatch($r, {"add": cmdAdd, "list": cmdList});
```

The handlers are ordinary `func` values rather than names: a func value called by `dispatch` (inside the `args` module) runs in the entry program's own context, so it resolves its own imports.

Argument syntax accepted

- `--flag value` , `--flag=value` , and (for a `store_true` / `count` flag) a bare `--flag` .
- Short flags `-x value` , `-xvalue` (glued), and `-abc` **bundling** (each character a flag; the first value-taking flag consumes the rest of the cluster or the next token).
- `--` ends flag parsing: everything after is positional.
- A `-` followed by a digit (`-5`) is a **positional** (a negative number), not a flag.

Subcommands

A subcommand is its own `Parser` . Global flags come **before** the subcommand word; everything after it is parsed by the subcommand's parser. The chosen name is read from `$r.command` , and the subcommand's values live in the same `Result` :

```
def add as args.Parser init args.positional(args.parser("add", "add a remote"),
"url", "the URL");
def git as args.Parser init args.parser("git", "a tiny vcs");
$git = args.boolFlag($git, "quiet", "q", "suppress output");
$git = args.command($git, "add", "add a remote", $add);

def r as args.Result init args.parse($git, os.ARGS);      # e.g. git -q add http:/
/x
# $r.command == "add"; args.asString($r, "url") == "http://x"; args.asBool($r,
"quiet") == true
```

Scope (and what's out)

This is **argparse Level B** - the ~95%. Out of scope (argparse Level C, diminishing returns for a `.j` module): argument groups, mutually-exclusive sets, parent parsers, prefix-abbreviation matching (`--verb` for `--verbose`), and `@argfile` .

See also

- os.md - `os.ARGS` (the input) and the primitive `os.hasFlag` / `os.flag` .
- modules/index.md - the module catalog and import rules.

barcode - barcode / QR code generation

Import with `import "barcode.j" as barcode;` . Generate scannable codes as **images** - the complement to `label` , which emits printer-native barcode *commands* . `encode(data, symbology, opts)` builds a device-independent `Symbol` (a module matrix for 2D, bar widths for 1D), and the renderers turn it into SVG, PNG, terminal art, or the raw matrix. Pure `.j` over `compress` (zlib)

- `crc` (CRC-32) + `encoding` + the bitwise operators - no image library; runs

on both binaries.

```
import "barcode.j" as barcode;

def opts as barcode.Options init barcode.defaults();
def qr as barcode.Symbol init barcode.encode("https://example.com", "qr", $opts);
def svg as string init barcode.svg($qr, $opts);   # embed in HTML / email
def png as bytes init barcode.png($qr, $opts);     # a monochrome PNG
```

Runnable: `examples/modules/barcode_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/barcode_demo.j).

Encoding

`barcode.encode(data, symbology, opts) ->` `Symbol` . Symbologies:

Symbology	Kind	Notes
qr	2D	Reed-Solomon over GF(256), EC levels L/M/Q/H (opts.ecLevel), automatic version selection 1-40, data-mask scoring; numeric / alphanumeric / byte mode chosen for compactness
datamatrix	2D	ECC200, square symbols 10x10 to 26x26, ASCII encodation (digit-pair packing), Reed-Solomon over GF(0x12d)
code128	1D	Code set B (ASCII 32-126), auto checksum
code93	1D	uppercase + digits + -. $/+%, two check characters (C, K)
code39	1D	uppercase + digits + -. $/+%, * start/stop
ean13	1D	12 or 13 digits (check digit computed if omitted)
ean8	1D	7 or 8 digits
upca	1D	11 or 12 digits (UPC-A; an EAN-13 with a leading 0)
upce	1D	6 / 7 / 8 digits (UPC-E, zero-compressed)
itf	1D	Interleaved 2 of 5, even digit count
gs1-128	1D	Code 128 with a leading FNC1 and parenthesised Application Identifiers, e.g. (01)09501101020917(10)ABC123

```
def enum barcode.SymbolKind { Matrix, Linear };
def struct barcode.Symbol {
    kind as SymbolKind,                # SymbolKind.Matrix (2D) or SymbolKind.
Linear (1D)
    size as int,                       # matrix dimension (2D; 0 for 1D)
    matrix as list of list of bool,    # the 2D module grid (true = dark)
    bars as list of int,               # 1D bar/space run widths, starting with a
bar
    text as string                     # the encoded data
};
```

Rendering

```
def struct barcode.Options {
    scale as int, height as int, quiet as int,
    ecLevel as string, foreground as string, background as string,
    humanReadable as bool   # render the data as a text line under a 1D SVG (on
by default)
};
```

`barcode.defaults()` gives scale 8, quiet 4, EC level M, black on white, and a human-readable text line under 1D SVGs (`opts.humanReadable = false` to omit it). Scale 8 (8 px per module) keeps a code comfortably scannable by a phone camera off a screen - a smaller render can defeat autofocus or pick up screen moire; lower `opts.scale` for a more compact image.

Call	Returns	
barcode.svg(symbol, opts)	string	resolution-independent SVG (embeds in HTML / email)
barcode.png(symbol, opts)	bytes	a monochrome (grayscale) PNG, hand-encoded over compress + crc
barcode.terminal(symbol)	string	Unicode half-block art for the CLI / REPL (2D only), wrapped in a 4-module quiet zone so it stays camera-scannable
barcode.matrix(symbol)	list of list of bool	the raw 2D cells (e.g. to feed label.image)

`opts.scale` is pixels (PNG) or units (SVG) per module / narrow bar; `opts.quiet` is the mandatory light border in modules; `opts.height` is the bar height for 1D codes; `opts.foreground` / `background` are the SVG / PNG colours.

Verification

Correctness is pinned two ways: the overlay (`modules/barcode_test.j`) checks the Reed-Solomon against the canonical QR vector, the format / version BCH against known values, mode codewords (including the QR spec's numeric worked example), the DataMatrix 10x10 matrix against `zint` , and 1D bar patterns against independent reference encoders; and the Go suite (`cmd/jennifer/barcode_test.go`) renders PNGs, decodes them with the standard library (proving the hand-rolled PNG is byte-correct), and - where `zbarimg` is available - **optically scans** them to confirm they read.

Scope

- **QR versions 1-40** , all four EC levels, with numeric / alphanumeric / byte mode chosen for compactness. A byte-mode payload with non-ASCII bytes gets an ECI(26) UTF-8 declaration, so it decodes as UTF-8 in a strict reader. Kanji mode, other ECI assignments, and structured-append are not covered. Large versions (v30+) are slow to render (the mask-penalty scoring runs in the tree-walking interpreter), not incorrect.
- **DataMatrix ECC200 square symbols 10x10 to 26x26** (single data region), ASCII encodation. Rectangular symbols, larger sizes (region interleaving), and the C40 / Text / X12 / EDIFACT / Base256 encodations are follow-ons; a letter-heavy payload is valid but larger than an optimal-encodation encoder would produce.
- **The GF(256) / Reed-Solomon math** lives in a private `barcode_ecc.inc.j` (`include` d, parameterised by primitive polynomial and generator base, so QR (0x11d) and DataMatrix (0x12d) share it).
- **No general image library** - the only raster need is a monochrome bitmap, which the PNG encoder covers directly.
- **Aztec / PDF417 / MaxiCode** are not included.

See also

- label.md - printer-native barcode *commands* (the other half).
- compress.md / crc.md - the PNG encoder's zlib and CRC-32.
- modules/index.md - the module catalog and import rules.

bloom - Bloom filter (probabilistic set)

Import with `import "bloom.j" as bloom;` . A compact, probabilistic set: `add` records a string and `mightContain` tests membership with **no false negatives** (a member always reports present) but possible **false positives** (a non-member may report present, with a probability that grows as the filter fills). Trades a little accuracy for a lot of space - ideal for "have I seen this before?" checks over large sets. Pure `.j` over `hash` + `strings` + `binary` ; runs on both binaries.

```
import "bloom.j" as bloom;

def f as bloom.Filter init bloom.new(1024, 4);
$f = bloom.add($f, "alice");
$f = bloom.add($f, "bob");
bloom.mightContain($f, "alice");   # true
bloom.mightContain($f, "carol");   # almost always false
```

Runnable: `examples/modules/bloom_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/bloom_demo.j).

Surface

```
def struct bloom.Filter { bits as bytes, size as int, hashes as int };
```

Call	Returns	
bloom.new(size, hashes)	Filter	an empty filter of size bits and hashes hash functions (both >= 1)
bloom.optimal(n, fpr)	Filter	an empty filter sized for n expected items at target false-positive rate fpr (n >= 1, 0 < fpr < 1)
bloom.add(f, item)	Filter	a fresh filter with item recorded
bloom.addAll(f, items)	Filter	a fresh filter with every item of a list of string recorded
bloom.mightContain(f, item)	bool	true if the item might be present, false if it is definitely absent
bloom.serialize(f)	bytes	encode a filter to bytes (4-byte size, 4-byte hashes, then the bit array)
bloom.deserialize(b)	Filter	reconstruct a filter from serialize output (identical filter, membership preserved)
bloom.union(a, b)	Filter	a fresh filter holding the members of both a and b (same size and hashes required)
bloom.merge(a, b)	Filter	alias for bloom.union

The `hashes` bit positions per item come from **double-hashing one SHA-256 digest** - `pos_i = (h1 + i*h2) mod size` , where `h1` / `h2` are the first two 32-bit words of the digest - so one hash yields all *k* positions.

Choosing size and hashes

Bigger `size` and a well-chosen `hashes` lower the false-positive rate. Rules of thumb for `n` expected items at a target false-positive probability `p` :

- **bits** `m ~= -n * ln(p) / (ln 2)^2` (about `9.6 * n` bits for `p = 1%` , `14.4 * n` for `p = 0.1%`).
- **hashes** `k ~= (m / n) * ln 2` (about 7 for `p = 1%`).

So for 10000 items at 1%: `size ~= 96000` , `hashes = 7` .

bloom.optimal(n, fpr) does this arithmetic for you - it computes `m = ceil(-(n * ln fpr) / (ln 2)^2)` and `k = round((m / n) * ln 2)` (clamped to `k >= 1`) and returns an empty filter of that shape. So `bloom.optimal(1000, 0.01)` yields a `size = 9586` , `hashes = 7` filter with no manual sizing.

Serialization and union

`bloom.serialize` encodes a filter to a self-describing `bytes` blob (a 4-byte big-endian `size` , a 4-byte big-endian `hashes` , then the raw bit array), and `bloom.deserialize` reverses it exactly - `serialize` then `deserialize` round-trips to a byte-identical filter with membership preserved, so a filter can be written to disk (`fs`) or sent over the wire and rebuilt later.

`bloom.union` (alias `bloom.merge`) bitwise-ORs two filters of the **same size and hashes** into a fresh filter: an item present in either input is present in the union. Filters that disagree on `size` or `hashes` cannot be combined and throw an `Error{kind: "bloom"}` .

```
def a as bloom.Filter init bloom.addAll(bloom.new(2048, 5), ["alice", "bob"]);
def b as bloom.Filter init bloom.addAll(bloom.new(2048, 5), ["carol"]);
def both as bloom.Filter init bloom.union($a, $b);   # holds alice, bob, carol

def blob as bytes init bloom.serialize($both);
def restored as bloom.Filter init bloom.deserialize($blob);   # identical filter
```

Scope

- **Add and test only** - a standard Bloom filter cannot remove an item or count occurrences (removal needs a counting Bloom filter; membership only, here).
- **Value-semantic add.** Each `add` copies the bit array and returns a fresh filter, so chain adds (`$f = bloom.add($f, x)`); it does not mutate in place. For many inserts this copies the array each time - fine for typical set sizes, not for millions of adds into a huge filter in a tight loop.
- **Strings only.** Hash other values through `convert.toString` or a `json` / `encoding` representation first.
- **Non-crypto use.** The filter is a set membership structure, not a security primitive.

See also

- hash.md - the SHA-256 the positions derive from.
- ringbuffer.md - the sibling data-structure module.
- modules/index.md - the module catalog and import rules.

cron - cron schedules

Import with `import "cron.j" as cron;` . Parse and evaluate **cron expressions** - the five-field `minute hour day-of-month month day-of-week` spec. `parse` builds a `Schedule` , `matches` tests whether a `time.Time` fires it, and `next` finds the next fire at or after a time. A **pure calculator** over `time` - no clock, no sleeping - so it runs on **both** binaries; a real scheduler is your own `spawn` + `time.sleep` loop over `cron.next` .

```
import "cron.j" as cron;

def s as cron.Schedule init cron.parse("30 9 * * 1-5");   # 09:30 on weekdays
def fire as time.Time init cron.next($s, time.now());
io.printf("next run: %s\n", time.iso($fire));
```

Runnable: `examples/modules/cron_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/cron_demo.j).

Functions

Call	Returns	Notes
cron.parse(expr)	Schedule	Parse a five-field expression.
cron.matches(schedule, t)	bool	Does the schedule fire at t? (minute granularity - seconds are ignored).
cron.next(schedule, after)	time.Time	The next fire at or after after, keeping its zone offset.

`parse` returns a `Schedule` :

```
def struct cron.Schedule {
    minutes as list of int, hours as list of int, daysOfMonth as list of int,
    months as list of int, weekdays as list of int,
    domStar as bool, dowStar as bool, reboot as bool
};
```

The five list fields hold the matching values for each cron field; `domStar` / `dowStar` record whether the day-of-month / day-of-week field was `*` (driving the either/both rule below), and `reboot` is `true` for an `@reboot` schedule.

Fields

Five whitespace-separated fields, each with the usual operators:

Field	Range	
minute	0-59	
hour	0-23	
day of month	1-31	
month	1-12	
day of week	0-7	0 and 7 are both Sunday

Each field accepts `*` (every value), a single number, an `a-b` range, an `a,b,c` list, and a `/n` step - on a wildcard (every `n` th value), a range (`a-b/n`), or a value (`a/n` , meaning `a` to the field maximum). Examples:

Expression	Fires
* * * * *	every minute
*/15 * * * *	every 15 minutes
0 9 * * 1-5	09:00 on weekdays (Mon-Fri)
0 0 1 * *	midnight on the 1st of each month

30 3 * * 0	03:30 on Sundays
0 0 13 * 5	midnight on Friday the 13th (see below)

Named months and weekdays

The month field also accepts the three-letter names `JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC` (as 1-12) and the day-of-week field accepts `SUN MON TUE WED THU FRI SAT` (as 0-6). Names are **case-insensitive** (`jan` , `Jan` , `JAN` all work) and can appear anywhere a number can - single values, ranges, and lists:

Expression	Same as
0 9 * * MON-FRI	0 9 * * 1-5 (09:00 on weekdays)
0 0 1 JAN,JUL *	0 0 1 1,7 * (1st of January and July)
30 3 * * SUN	30 3 * * 0 (03:30 on Sundays)

Like the numeric `0` , `SUN` is Sunday (there is no name for the alternate `7`).

Nickname macros

A whole expression may instead be one of the standard @ nicknames, which expand to a five-field expression before parsing:

Nickname	Expands to	Fires
@yearly, @annually	0 0 1 1 *	midnight on January 1st
@monthly	0 0 1 * *	midnight on the 1st of each month
@weekly	0 0 * * 0	midnight each Sunday
@daily, @midnight	0 0 * * *	midnight every day
@hourly	0 * * * *	the top of every hour

Nicknames are case-insensitive. An unknown `@name` throws a catchable `Error` (kind `"cron"`).

@reboot

`@reboot` means "at startup", so it has **no time-based schedule** . It parses to a `Schedule` whose `reboot` field is `true` , and that flag changes the two evaluators:

- `cron.matches($s, t)` is **always false** for a `@reboot` schedule (no clock time ever fires it).
- `cron.next($s, after)` **throws** a catchable `Error` (kind `"cron"`) - there is no next fire time.

Test for it yourself and run the job once at program start:

```
def s as cron.Schedule init cron.parse("@reboot");
if ($s.reboot) {
    runJob();   # fire once, at startup
}
```

The day-of-month / day-of-week rule

When **both** the day-of-month and day-of-week fields are restricted (neither is `*`), a day matching **either** one fires - the standard cron behavior. So `0 0 13 * 5` fires on the 13th *and* on every Friday. When one of the two is `*` , only the other constrains the day.

next

`cron.next(schedule, after)` returns the first matching minute at or after `after` (with its seconds zeroed), preserving the input's zone offset. If `after` already sits exactly on a matching minute, it is returned. The search skips non-matching days whole (so a yearly schedule is found quickly) and gives up after a five-year horizon - an impossible schedule (e.g. `0 0 31 2 *` , February 31st) throws a catchable `Error` (kind `"cron"`) rather than looping forever.

Zones are fixed-offset (as in the `time` library), so `next` does no DST arithmetic.

Scope

- **Standard five fields** , plus named months / weekdays and the `@` nickname macros (including `@reboot`) above. No seconds field and no non-standard extensions (`L` , `W` , `#` , `?`).
- **A calculator, not a runner.** It never touches the clock. Drive it yourself: `time.sleep(time.sub(cron.next($s, time.now()), time.now()))` , then run the job.

See also

- time.md - the `time.Time` cron computes over.
- concurrency.md - `spawn` for a background scheduler loop.
- modules/index.md - the module catalog and import rules.

csv - RFC 4180 comma-separated values

Import with `import "csv.j" as csv;` . Parses CSV text into rows of string fields and formats rows back into text, with a quoting-aware hand-written scanner. Pure Jennifer over `strings` and `maps` , so it runs on either binary. The delimiter is configurable, so the same code reads and writes TSV and other single-character-separated formats.

```
use io;
import "csv.j" as csv;

def rows as list of list of string init csv.parse("name,note\n\"Smith, J\",hi");
io.printf("%s | %s\n", $rows[1][0], $rows[1][1]);        # Smith, J | hi

def recs as list of map of string to string init csv.toRecords($rows);
io.printf("%s\n", $recs[0]["note"]);                      # hi
```

Runnable: `examples/modules/csv_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/csv_demo.j).

Surface

Call	Returns	Notes
csv.parse(s)	list of list of string	Parse standard comma-delimited CSV into rows of fields.
csv.parseWith(s, delim)	list of list of string	Same, with a single-character delimiter ("\t" for TSV).
csv.format(rows)	string	Encode rows as comma-delimited CSV; quotes fields that need it. Unsafe for spreadsheets - see below.
csv.formatWith(rows, delim)	string	Same, with a single-character delimiter.
csv.formatSafe(rows)	string	Like format, but neutralises spreadsheet formula injection. The export-to-spreadsheet path.
csv.formatSafeWith(rows, delim)	string	Like formatWith, injection-neutralised.
csv.dialect(delim)	csv.Dialect	A dialect with the given delimiter and defaults (quote ", no comment, no trim).
csv.parseDialect(text, d)	list of list of string	Parse under a csv.Dialect (custom quote, comment-line skip, unquoted-field trim).
csv.formatDialect(rows, d)	string	Format under a csv.Dialect (custom delimiter and quote).
csv.toRecords(rows)	list of map of string to string	Treat row 0 as a header; map each later row to a header-keyed record.
csv.fromRecords(header, records)	list of list of string	Inverse: a header row followed by one row per record, in header order.
csv.reader(file) / readerWith(file, delim)	csv.Reader	Wrap an open read-mode fs.File as a streaming reader.
csv.readRow(reader)	list of string	Read one complete record (spans physical lines inside a quoted field); [] at EOF.
csv.readerEof(reader)	bool	True once no further record remains; loop while (not readerEof($r)).
csv.reader/writer closeReader/closeWriter	null	Close the underlying file handle.

csv.writer(file) / writerWith(file, delim)	csv.Writer	Wrap an open write/append-mode fs.File as a streaming writer.
csv.writeRow(writer, fields)	null	Append one quoted-as-needed record terminated by LF.

Parsing (RFC 4180)

`parse` and `parseWith` implement the RFC 4180 (https://www.rfc-editor.org/rfc/rfc4180) rules:

- Fields are separated by the delimiter (a comma by default); records by `LF` or `CRLF` . A bare `CR` outside quotes also ends a record.
- A field wrapped in `"` may contain the delimiter, line breaks, and quotes; an embedded quote is written doubled (`""`) and decodes to one.
- An empty input yields no rows; a trailing record separator does **not** add an empty trailing row. A separator with nothing after it *within* a record is a real empty field (`a,` is two fields, the second empty).

```
# Embedded comma, doubled quote, and newline all survive.
def rows as list of list of string init csv.parse("\"Smith, J\",\"said
\"\"hi\"\"\",\"two\nlines\"");
# rows[0] == ["Smith, J", "said \"hi\"", "two\nlines"]
```

Formatting

`format` / `formatWith` are the inverse. A field is quoted only when it carries the delimiter, a quote, or a line break; embedded quotes double. Records are joined with `LF` and **no trailing newline** , so `parse(format(rows))` round-trips the data:

```
def rows as list of list of string init [];
$rows[] = ["plain", "has,comma", "q\"uote"];
io.printf("%s\n", csv.format($rows));
# plain,"has,comma","q""uote"
```

Only the **record separators** normalise: a `CRLF` - or `CR` -terminated input re-emits with `LF` between records. Line breaks *inside* a quoted field are field content and pass through verbatim, so no data is altered.

Spreadsheet formula injection (use formatSafe)

format / formatWith are UNSAFE for a CSV that a spreadsheet will open. RFC 4180 quoting does not stop CSV formula injection (CWE-1236 (https://cwe.mitre.org/data/definitions/1236.html)): a field whose first character is `=` , `+` , `-` , `@` , a tab (`0x09`), or a CR (`0x0D`) is interpreted by Excel and Google Sheets as a **formula** on import - so a value like `=SUM(...)` , or a payload like `=cmd|'/c calc'!A1` , executes in the victim's spreadsheet. A field is still just text in the CSV; the danger is the consuming spreadsheet.

`formatSafe` / `formatSafeWith` are the export-to-spreadsheet path: they behave exactly like `format` / `formatWith` but, before quoting, prefix any such field with a single apostrophe (`'`), the standard neutraliser that every major spreadsheet treats as "this cell is literal text":

```
def rows as list of list of string init [];
$rows[] = ["=SUM(A1:A9)", "normal"];
io.printf("%s\n", csv.formatSafe($rows));
# '=SUM(A1:A9),normal      <- the formula is now inert text on import
```

Only a field's **first** character is inspected (so `a=b` is untouched), and a normal field is left exactly as `format` would write it. Use plain `format` only when the target is another program that parses CSV as data, never a spreadsheet.

Dialects

`csv.Dialect` groups the non-standard knobs so a caller can read a `;` -delimited, # -commented, whitespace-padded file in one call:

```
def struct csv.Dialect {
    delimiter as string,   # the field delimiter
    quote as string,       # the quote character (default ")
    comment as string,     # a comment-line prefix, or "" for none
    trim as bool           # trim whitespace from *unquoted* fields on parse
};
```

`csv.dialect(delim)` builds one with the standard defaults (quote " , no comment, no trim); set the other fields to customise. `parseDialect` reads under it, `formatDialect` writes under it (the `comment` / `trim` fields are parse-only):

```
def d as csv.Dialect init csv.dialect(";");
$d.comment = "#";
$d.trim = true;
def rows as list of list of string init csv.parseDialect("# note\n a ; b ;c", $d);
# rows == [["a", "b", "c"]]
```

- **comment** skips a physical line only when the line's **record** starts with the prefix (outside any quoted field), so a # inside data is safe.
- **trim** trims **unquoted** fields only; a quoted field keeps its whitespace verbatim (`" keep "` stays `keep`).
- The plain `parse` / `parseWith` / `format` / `formatWith` are unchanged - a dialect is opt-in.

Streaming a large file

For a table too large to hold in memory, `csv.reader` / `csv.writer` wrap an open `fs.File` handle and process one record at a time. The wrapped file shares its read position across value copies (like every `fs` handle), so successive `readRow` calls advance the same stream. `readRow` reads a **complete** record even when a quoted field spans several physical lines:

```
use fs;
import "csv.j" as csv;

def rf as fs.File init fs.open("big.csv", "read");
def r as csv.Reader init csv.reader($rf);
while (not csv.readerEof($r)) {
    def row as list of string init csv.readRow($r);
    if (len($row) > 0) {
        # process one record without loading the whole file
    }
}
csv.closeReader($r);

def wf as fs.File init fs.open("out.csv", "write");
def w as csv.Writer init csv.writer($wf);
csv.writeRow($w, ["name", "note"]);
csv.writeRow($w, ["Smith, J", "hi"]);   # embedded comma auto-quoted
csv.closeWriter($w);
```

`readRow` returns `[]` at end-of-file (and for a blank line), so `readerEof` is the loop guard. `readerWith` / `writerWith` take a custom delimiter for streaming TSV. The streaming reader decodes an embedded newline (a quoted field split across lines) back to `\n` ; a `\r` inside a quoted field is normalised to `\n` on the streaming path (the whole-file `parse` preserves it).

Header-keyed records

Most CSV has a header row. `toRecords` pairs it with the data rows, giving one `map of string to string` per record keyed by column name; `fromRecords` rebuilds rows from records and an explicit header:

```
def rows as list of list of string init csv.parse("name,age\nAda,36\nGrace,45");
def recs as list of map of string to string init csv.toRecords($rows);
# recs[0] == {"name": "Ada", "age": "36"}

def back as list of list of string init csv.fromRecords(["name", "age"], $recs);
# back == [["name","age"], ["Ada","36"], ["Grace","45"]]
```

Details worth knowing:

- Every record carries **every** header key. A data row shorter than the header fills the missing fields with `""` ; fields past the header width are dropped (they have no name).
- Duplicate header names collapse - a later column overwrites an earlier one of the same name (map keys are unique).
- `fromRecords` takes the header **explicitly** rather than reading it off the records, because map iteration order is insertion order per record and would not give a stable column order across records. A key absent from a record writes `""` .
- `toRecords([])` is `[]` ; a header-only input yields an empty record list.

Out of scope

Type inference (numbers, booleans, dates) is not part of this module - every field is a `string` , and the caller converts what it needs with `convert.toInt` / `convert.toFloat` . For whole-file work, read with `fs.readString` (or slurp stdin) and hand the text to `parse` ; for a file too large to hold in memory, use the streaming `reader` / `writer` above.

See also

- strings.md - `split` / `join` / `replace` , which `csv` builds the scanner and encoder on.
- maps.md - `has` / `keys` , used by the record helpers.
- fs.md - `readString` to load a CSV file to hand to `parse` .
- modules/index.md - the module catalog and import rules.

discord - Discord Webhook client

Import with `import "discord.j" as discord;` . Post messages to a Discord channel through a channel Webhook (https://discord.com/developers/docs/resources/webhook), on top of the `http` module - a sibling of `gotify` and `slack` . Needs the default `jennifer` binary. The webhook URL is a secret: read it from the environment or a config file, never commit it.

```
import "discord.j" as discord;

discord.send("https://discord.com/api/webhooks/1/xxx", "deploy finished");

def m as discord.Message init discord.embed(
    discord.content(discord.message(), "heads up"),
    "Deploy", "build 1234 is live", 3066993);
discord.sendMessage("https://discord.com/api/webhooks/1/xxx", $m);
```

Runnable: `examples/modules/discord_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/discord_demo.j).

Plain messages

`discord.send(webhookUrl, content)` posts `{"content": content}` (the content is Discord markdown) and returns the `http.Response` - Discord answers `204 No Content` on success.

Rich messages (embeds)

Build a message from embeds with value-semantic builders - each returns a fresh `Message` , so they chain - then post it with `sendMessage` (or inspect the JSON with `render`).

```
def struct discord.Message {
    content as string,       # top-level content ("" to omit; embeds must then be
present)
    embeds as list of Embed, # the message embeds
    username as string,      # "" or a per-message username override
    avatar as string         # "" or a per-message avatar image URL override
};

def struct discord.Embed {
    title as string,
    description as string,
    color as int,
    fields as list of string, # pre-rendered field JSON fragments
    footer as string,         # "" or a pre-rendered footer JSON object
    author as string          # "" or a pre-rendered author JSON object
};
```

Call	Returns	
discord.message()	Message	start an empty message
discord.content(m, content)	Message	set the top-level content
discord.username(m, name)	Message	override the webhook's display name
discord.avatar(m, url)	Message	override the webhook's avatar image
discord.embed(m, title, description, color)	Message	append an embed

discord.embedField(m, name, value, inline)	Message	add a field to the latest embed
discord.embedFooter(m, text)	Message	set the footer of the latest embed
discord.embedAuthor(m, name)	Message	set the author of the latest embed
discord.render(m)	string	render the JSON payload
discord.sendMessage(webhookUrl, m)	http.Response	post the built message

`color` is the embed's left-bar colour as a **decimal** RGB integer (e.g. `3066993` for green, `0xFF0000` = `16711680` for red). At least one of `title` / `description` should be non-empty. All strings are JSON-escaped for you, so quotes and newlines are safe. `content` and `embeds` are each emitted only when present, so a plain `send` , an embeds-only message, and a content-plus-embeds message are all valid.

Embed fields, footer, and author

`embedField` / `embedFooter` / `embedAuthor` decorate the **most recent** embed (the one last appended by `embed`), so build the embed first, then add to it. Each throws an `Error{kind: "discord"}` if the message has no embed yet. `embedField` appends to the embed's `fields` array (`inline` lets Discord pack several fields side by side); `embedFooter` sets `footer.text` and `embedAuthor` sets `author.name` .

```
def m as discord.Message init discord.embed(discord.message(), "Deploy", "v1234",
3066993);
$m = discord.embedField($m, "Env", "production", true);
$m = discord.embedField($m, "Duration", "42s", true);
$m = discord.embedFooter($m, "ci bot");
$m = discord.embedAuthor($m, "release-pipeline");
# render($m) ->
# {"embeds":[{"title":"Deploy","description":"v1234","color":3066993,
#   "fields":[{"name":"Env","value":"production","inline":true},
#             {"name":"Duration","value":"42s","inline":true}],
#   "footer":{"text":"ci bot"},"author":{"name":"release-pipeline"}}]}
```

Webhook identity override

`username` and `avatar` set the top-level `username` / `avatar_url` on the payload, so a single webhook can post under different names and images.

```
def m as discord.Message init discord.content(discord.message(), "backup
complete");
$m = discord.username($m, "backup-bot");
$m = discord.avatar($m, "https://example.com/backup.png");
# render($m) -> {"username":"backup-bot","avatar_url":"https://example.com/backup.
png","content":"backup complete"}
```

Scope

- **Channel Webhooks** , not the bot API - no bot token, gateway, slash commands, threads, reactions, or attachments. The channel is fixed by the webhook.
- **A subset of embeds** - title, description, color, fields, footer, and author, plus the `username` / `avatar_url` identity overrides. Embed thumbnail, image, and timestamp are not built here (compose the JSON yourself and post via `http` if you need them).
- **No retry / rate-limit handling** - a non-2xx is returned as the response value for the caller to inspect, not thrown.

See also

- http.md - the HTTP client this module builds on.
- slack.md / gotify.md - the sibling notifiers.
- modules/index.md - the module catalog and import rules.

docblock - Jennifer doc comments and their parser

`import "docblock.j" as docblock;`

Two things in one: the **blessed doc-comment format** for Jennifer source, and `docblock.parse`, which reads source and returns that documentation as structured, typed values. It produces *data*; it does not render (turning docs into HTML is a separate consumer). Pure Jennifer over `regex` and `strings`; runs on either binary.

The format: Jennifer doc comments

A doc comment opens with **exactly `/**`** (a plain `/*` *block comment stays invisible) and closes with* `*/`. It **immediately precedes** the construct it documents - a func, a def struct, a def enum, or a def const - or, when it carries an @module tag, it is the **file preamble**. Whether a construct is exported is read from its export` keyword, never a tag.

```
/**
 * Distance between two points.
 *
 * A longer description may follow, separated from the summary by a blank line.
 * @param ax {float} first x coordinate
 * @param ay {float} first y coordinate
 * @return {float} the Euclidean distance
 * @since 0.9
 */
export func distance(ax as float, ay as float, bx as float, by as float) { ... }
```

The body is a **summary** - the first paragraph, with wrapped lines joined, so it can span several source lines - then an optional **description** (the remaining paragraphs, separated from the summary by a blank line), then `@`-tags. Because the split is by paragraph and not by line, a summary that wraps across source lines stays one sentence rather than breaking at the line break; put a blank line where you want the summary to end and the description to begin.

Types are written in Jennifer syntax, in braces

Every `{...}` type is written verbatim in Jennifer's own syntax - `{int}`, `{list of int}`, `{map of string to list of int}`, `{json.Value}`. There is no `any` / `mixed` pseudo-type (Jennifer has no top type); an opaque value documents as `json.Value` or a named struct. The braces make extraction unambiguous.

Tags

Tag	On	Meaning
@param name {type} desc	func	One per parameter.
@field name {type} desc	struct	One per field.
@return {type} desc	func	The return value.
@throws {type} desc	func	An error the function may throw (repeatable).
@since version	any	When it was introduced.
@deprecated [reason]	any	Marks it deprecated.
@see ref	any	A cross-reference (repeatable).
@example	func	A runnable example; its body is the following lines up to the next tag.
@internal	any	Not part of the public surface.
@module name	preamble	Marks the file preamble (module doc).
@author / @version / @license	preamble	Module metadata.

The reference layout is **jennifer fmt output** - fmt preserves doc comments and normalises layout deterministically, so associating a comment with the construct that follows it is reliable.

Parsing: docblock.parse(source)

`docblock.parse(source as string) -> FileDoc` returns the whole document:

```
import "docblock.j" as docblock;
def doc as docblock.FileDoc init docblock.parse(source);
io.printf("%s\n", $doc.module.summary);
for (def f in $doc.funcs) {
    io.printf("%s (%d params)\n", $f.name, len($f.params));
}
```

The result types

The result is **typed data, not a tag bag** - Jennifer has no sum types, so heterogeneous collections are modelled as parallel typed lists plus fixed-field structs. All are exported:

- `FileDoc { module, funcs, structs, enums, consts, diagnostics }`
- `ModuleDoc { summary, description, author, version, license, see }`
- `FuncDoc { name, exported, summary, description, params, returns, throws, examples, since, deprecated, see, internal }`
- `StructDoc { name, exported, summary, description, fields, since, deprecated, see, internal }`
- `EnumDoc { name, exported, summary, description, since, deprecated, see, internal }` (variants are described in prose, not tags, so there is no field list)
- `ConstDoc { name, exported, type, summary, description, since, deprecated, see, internal }`
- `ParamDoc { name, type, description }` (also used for a struct's `fields`)
- `ReturnDoc { type, description }`, `ThrowDoc { type, description }`
- `Diagnostic { severity, line, message }`

An absent value is its zero: a function with no `@return` has a `returns` whose `type` is `""`; a file with no preamble has an empty `module`.

Diagnostics: it reports, it never enforces

`docblock` never fails on a documentation error - problems come back as `Diagnostic` values and the caller decides what is fatal. Three are reported:

- A **@param / @field that names nothing real** - the documented name is not a parameter / field of the construct.
- A **real parameter / field with no @param / @field** - docs drifting behind the code, the commonest doc bug. Names and counts are cross-checked against the actual declaration.
- An **orphaned** doc comment - one that precedes neither a documentable construct nor a preamble.

```
[warning] line 36: @param "typo" is not a param of drifted
[warning] line 36: param "real" of drifted has no @param
```

Whole-body analysis (e.g. "a `@return` on a function that never returns a value") is **out of scope**: it needs the AST, and the module has only the text. What earns its keep is the signature cross-check above.

Scanner correctness

Comment boundaries are found by a character-level `/* */` depth scan that **skips string literals** and # line comments and **nests /* */ correctly** - not a fragile "delimiter alone on its line" rule. So a `/**` inside a string literal is not mistaken for a doc comment, and a nested `/* ... */` inside a doc body does not close it early.

Checking a file or tree

`scripts/docblock-check.sh` runs the parser over a `.j` file or a whole directory and reports each file's doc coverage plus any diagnostics, exiting non-zero when it finds a problem - a ready-made pre-commit / CI check:

```
scripts/docblock-check.sh modules/          # every .j under modules/
scripts/docblock-check.sh myapp.j           # a single file
# ok   modules/web.j  (1 module, 22 func, 3 struct, 0 enum, 0 const)
# WARN app.j  (1 diagnostic)
#        line 12: @param "nmae" is not a param of greet
```

It finds the interpreter via `JENNIFER=/path/to/jennifer`, else `./jennifer` at the repo root, else `jennifer` on `PATH`. It exits `0` when every file is clean and `1` when any file has diagnostics, so it drops straight into a pre-commit hook.

Enforced in CI

The project runs this check on every push and pull request (a "Check doc comments" step in `.github/workflows/test.yml`), over both trees:

```
scripts/docblock-check.sh examples/
scripts/docblock-check.sh modules/
```

A drift diagnostic fails the build, so a doc comment can never silently fall out of step with the code it documents - the same guarantee the linter and the module test overlays give. A missing doc comment is *not* an error (docblock reports drift, not absence); it flags a doc that is present but wrong, or one that documents nothing.

See also

- `jennifer fmt` - the canonical layout normaliser.
- `regex` / `strings` - the libraries `docblock` is built on.
- `html` - render parsed docs to HTML (a separate consumer, not part of `docblock`).

dot - Graphviz DOT graph description

Import with `import "dot.j" as dot;` . Builds a graph of nodes and edges with attributes and renders it to `.dot` text for an external Graphviz tool to lay out and draw. Pure Jennifer over `strings` and `lists` , so it runs on either binary. It emits the text description only - graph **layout** is Graphviz's job and is deliberately not reimplemented (drawing a graph is a large algorithm the `dot` tool exists to do). Every builder returns a fresh `Graph` , so value semantics hold: the graph you pass in is never mutated.

```
use io;
import "dot.j" as dot;

def g as dot.Graph init dot.digraph("pipeline");
$g = dot.graphAttr($g, "rankdir", "LR");
$g = dot.nodeAttr($g, "shape", "box");
$g = dot.nodeWith($g, "lexer", {"label": "Lexer"});
$g = dot.edgeWith($g, "lexer", "parser", {"label": "tokens"});
io.printf("%s", dot.render($g));
```

Pipe the rendered text to Graphviz to produce an image:

```
jennifer run graph.j | dot -Tsvg > graph.svg
```

Runnable: `examples/modules/dot_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/dot_demo.j).

Surface

Call	Returns	Notes
dot.digraph(name)	Graph	A new directed graph - renders digraph, edges a -> b.
dot.graph(name)	Graph	A new undirected graph - renders graph, edges a -- b.
dot.node(g, id)	Graph	Add a bare node (optional - an id first seen on an edge is created implicitly).
dot.nodeWith(g, id, attrs)	Graph	Add a node with a map of string to string of DOT attributes ({"shape": "box"}).
dot.edge(g, src, dst)	Graph	Add a bare edge between two node ids.
dot.edgeWith(g, src, dst, attrs)	Graph	Add an edge with its own attributes ({"label": "calls"}).
dot.graphAttr(g, key, value)	Graph	Set a graph-level attribute (rankdir = LR, a graph label, ...).
dot.nodeAttr(g, key, value)	Graph	Set a default node attribute, rendered once as node [...].
dot.edgeAttr(g, key, value)	Graph	Set a default edge attribute, rendered once as edge [...].
dot.render(g)	string	Render the graph to DOT text (with a trailing newline).

Attribute maps keep insertion order, so the rendered attribute list is stable. String values (labels, the graph name, attribute values) are DOT-escaped: a backslash and a double-quote are backslash-escaped, and a literal newline becomes the two-character `\n` DOT escape - so a label carrying quotes or newlines stays well-formed.

Types

- **Graph** { `directed` as `bool`, `name` as `string`, `nodes` as `list of Node`, `edges` as `list of Edge`, `graphAttrs` as `map of string to string`, `nodeAttrs` as `map of string to string`, `edgeAttrs` as `map of string to string` }
- **Node** { `id` as `string`, `attrs` as `map of string to string` }
- **Edge** { `from` as `string`, `to` as `string`, `attrs` as `map of string to string` }

The fields are public for inspection, but the builders (`digraph` / `graph` / `node*` / `edge*` / `*Attr`) are the

intended construction path.

dotenv - .env configuration files

Import with `import "dotenv.j" as dotenv;` . Read the KEY=VALUE files that keep configuration and secrets out of source, with layered environment profiles and `${VAR}` interpolation. Over `fs` + `strings` + `os` + `path` + `regex` + `maps` ; pure `.j` , runs on **both** binaries.

```
import "dotenv.j" as dotenv;
use os;

# Single file:
def cfg as map of string to string init dotenv.parse("PORT=8080\nURL=\"http://
h:${PORT}\"");
io.printf("%s\n", $cfg["URL"]);       # http://h:8080

# Layered, real-env-wins load (profile from JENNIFER_ENV):
dotenv.autoload(os.cwd());
```

Runnable: `examples/modules/dotenv_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/dotenv_demo.j).

Functions

Single-file primitives

Call	Returns	Notes
dotenv.parse(text)	map of string to string	Parse .env text; touches nothing else.
dotenv.read(path)	map of string to string	Read and parse one file.
dotenv.load(path)	map of string to string	Read, parse, and os.setEnv each variable (unconditional override); returns the map.

`load` overwrites whatever is already in the environment - it is the low-level primitive. For a layered, real-env-wins load, use the cascade loaders below.

Layered loaders

Merge, later overriding earlier and skipping absent files, from **one explicit base directory** (no search, no walk-up):

```
.env  ->  .env.local  ->  .env.<profile>  ->  .env.<profile>.local
```

The `<profile>` comes from the `profile` argument (or `JENNIFER_ENV` for `autoload`); an empty profile loads only the two base files (there is no `.env.default`). At the command line, `jennifer run --env=prod app.j` sets `JENNIFER_ENV=prod` before the program runs (identical to `JENNIFER_ENV=prod jennifer run app.j`), so `--env` is the ergonomic way to pick the profile `autoload` reads - see Run profiles.

Call	Returns	Notes
dotenv.readCascade(dir, profile)	map of string to string	The merged file layers. No env mutation.
dotenv.resolve(dir, profile)	map of string to string	The effective map: readCascade with the real OS env overlaid (real env wins). No env mutation.
dotenv.loadCascade(dir, profile)	map of string to string	readCascade, then os.setEnv only keys not already set in the real env. Returns the file map.
dotenv.autoload(dir)	map of string to string	loadCascade(dir, JENNIFER_ENV) - reads the JENNIFER_ENV variable for the profile.

Value precedence, highest first: a **pre-existing OS env var** (never overwritten)

> `.env.<profile>.local` > `.env.<profile>` > `.env.local` > `.env` .

Syntax

Each non-blank, non-comment line is a `KEY=VALUE` assignment:

```
# a comment line
PORT=8080
export NAME="ada lovelace"     # a leading `export` is ignored
GREETING='hi # not a comment'  # single quotes are literal
BANNER="line one
line two"                      # double quotes may span physical lines
URL="http://localhost:${PORT}" # ${VAR} interpolation
TOKEN=abc123        # trailing inline comment (unquoted values only)
EMPTY=
```

- **Comments** - a line starting with # is skipped; on an **unquoted** value a # (space then hash) starts an inline comment. Inside quotes, # is literal.
- **export** - a leading `export` prefix is stripped, so a file that doubles as a shell script parses the same.
- **Double quotes** expand the escapes `\n` , `\t` , `\r` (and `\"` for a literal quote) and **may span multiple physical lines** - the real newlines become part of the value. An unterminated double quote is a positioned `dotenv` error.
- **Single quotes** are fully literal - no escapes, no interpolation.
- **Unquoted** values are trimmed of surrounding whitespace.
- The value may contain = (only the first = splits the line); a line with no = or an empty key is ignored; a later duplicate key wins.

Interpolation

`${VAR}` in an **unquoted or double-quoted** value is replaced with `VAR` 's value. Resolution is **backward-reference only** - it looks up, in order:

1. a key already parsed (earlier in this file, or an earlier cascade layer),
2. the real OS environment,
3. the empty string.

Because a reference can only see keys defined *before* it, cycles are impossible by construction (a forward reference simply resolves to `""`).

- Interpolation runs in **unquoted** and **double-quoted** values; a **single-quoted** value is fully literal (`'${X}'` stays `${X}`).
- A `${...}` whose contents are not a valid variable name is kept **literal** .
- There is **no** `$(...)` / backtick command substitution - those are plain characters. A `.env` value can never execute a command. This is a hard security line, not a missing feature.

Security

`.env*` files are **trusted local input** - the developer controls the working directory; dotenv is not a sandbox. Within that model:

- **Strict profile validation.** A profile label must match `^[A-Za-z0-9_-]{1,64}$` before it is spliced into `.env.<profile>` , so a `JENNIFER_ENV` from an untrusted upstream cannot path-traverse (`../../etc/x` , `prod/../y`). An invalid label is a `dotenv` error.
- **Fixed base directory - no search, no walk-up.** Loading from one explicit `dir` closes the file-hijack class: there is no "nearest `.env` " walk that an attacker could poison by planting a file in a parent directory. `.env.local` overriding `.env` is safe because they share that same developer-controlled directory.
- **Real env wins.** The cascade loaders (`resolve` / `loadCascade` / `autoload`) never let a committed file

clobber a real environment variable - a deployment's real secret always beats a file value. (The single-file `load` still overrides unconditionally - use it only when that is what you want; reach for the cascade loaders otherwise.) To force-override from a cascade, use `readCascade` + your own `os.setEnv` loop.

> A variable set to the **empty string** in the real environment is treated as *unset* for "real env wins" (there is no `os.hasEnv`), so a file value can still fill it. Set a real value, not `""`, when you mean to override.

Scope

- **No $VAR (unbraced) interpolation** - only `${VAR}`. A bare `$FOO` is literal.
- **Multi-line applies to double quotes only** - a single-quoted value is one line.

See also

- fs.md - the file reads the loaders use.
- os.md - `os.setEnv` / `os.getEnv` for the environment.
- path.md - `path.join` builds the layer paths.
- toml.md - for richer, typed, nested configuration.
- modules/index.md - the module catalog and import rules.

feed - RSS 2.0 and Atom 1.0 syndication

Import with `import "feed.j" as feed;` . Build and parse web syndication feeds - **both RSS 2.0 and Atom 1.0 through one module** (design stance 1: not separate `rss` / `atom` modules). The format is chosen when you `build` and detected from the root element when you `parse` , so the same value-semantic `Feed` shape serves a feed reader, podcast client, news aggregator, or changelog-to-feed generator.

Parsing rides the `xml` library (entities, CDATA, Atom's namespaces); building emits escaped XML through it. Dates go through `time` . `fetch` pulls a feed over the `http` module.

```
import "feed.j" as feed;

def f as feed.Feed init feed.add(
    feed.feed("My Blog", "https://example.org"),
    feed.entry("Hello", "https://example.org/1"));
def rss as string init feed.build($f, "rss"); # or "atom"
def back as feed.Feed init feed.parse($rss);  # format auto-detected
```

Runnable: `examples/modules/feed_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/feed_demo.j).

The Feed, Entry and Enclosure structs

All three are value-semantic, so the builders return a fresh copy.

`feed.Feed` { `title` as `string`, `link` as `string`, `updated` as `time.Time`, `entries` as `list of Entry`, `author` as `string`, `categories` as `list of string` }

`feed.Entry` { `title` as `string`, `link` as `string`, `id` as `string`, `published` as `time.Time`, `updated` as `time.Time`, `summary` as `string`, `content` as `string`, `author` as `string`, `categories` as `list of string`, `enclosure` as `Enclosure` }

`feed.Enclosure` { `url` as `string`, `length` as `int`, `type` as `string` }

A field that maps to different names per format: `id` is RSS `guid` / Atom `id` ; `summary` is RSS `description` / Atom `summary` ; `content` is Atom-only (RSS carries the one description, kept in `summary`). `author` is RSS `<author>` / `<dc:creator>` (channel `<managingEditor>`) and Atom `<author><name>` ; `categories` are RSS `<category>` / Atom `<category term>` ; `enclosure` is the attached media file (RSS `<enclosure>` / Atom `<link rel="enclosure">`), the podcast episode's audio. An **unset date is the Unix epoch** and is omitted when building; an **empty enclosure url** means no enclosure and is likewise omitted.

Building

Call	Returns	Notes
feed.feed(title, link)	Feed	A new empty feed (no entries, unset date).
feed.feedUpdated(f, t)	Feed	A copy with the feed's updated instant set.
feed.add(f, e)	Feed	A copy with e appended to the entries.
feed.entry(title, link)	Entry	A new entry (empty id / summary / content, unset dates).
feed.entryId(e, id)	Entry	A copy with the stable id set.
feed.entryPublished(e, t)	Entry	A copy with the published instant set.
feed.entryUpdated(e, t)	Entry	A copy with the updated instant set.
feed.entrySummary(e, s)	Entry	A copy with the summary set.
feed.entryContent(e, c)	Entry	A copy with the full content set (Atom).
feed.entryAuthor(e, author)	Entry	A copy with the item author set.

feed.entryCategory(e, category)	Entry	A copy with category appended to the item's tags.
feed.entryEnclosure(e, url, length, type)	Entry	A copy with a media enclosure (podcast file) set.
feed.feedAuthor(f, author)	Feed	A copy with the feed author set.
feed.feedCategory(f, category)	Feed	A copy with category appended to the feed's tags.
feed.hasEnclosure(e)	bool	Whether the entry has a media enclosure (non-empty url).
feed.build(f, format)	string	Render to "rss" (RSS 2.0) or "atom" (Atom 1.0). Unknown format errors.

RSS uses RFC 822 dates (`pubDate` / `lastBuildDate`); Atom uses RFC 3339 (`published` / `updated`). All text is XML-escaped, so `&` , `<` , and `>` in titles or summaries round-trip.

Podcasts and metadata

`entryEnclosure` attaches the media file that makes a feed a **podcast** : RSS emits `<enclosure url length type/>` , Atom `<link rel="enclosure" href length type/>` , and parse reads either back into the `Enclosure` struct. The item's own page link (Atom's alternate `<link>`) stays separate from the enclosure, so `entry.link` is the episode page and `entry.enclosure.url` the audio. `author` and `categories` round-trip on both the feed and each entry; on parse an RSS author falls back to `<dc:creator>` when the native `<author>` is absent, and a blank / non-numeric enclosure `length` degrades to `0` rather than failing.

```
def ep as feed.Entry init feed.entryEnclosure(
    feed.entryAuthor(feed.entry("Episode 1", "https://show.example/1"), "Jane
Host"),
    "https://show.example/1.mp3", 12345678, "audio/mpeg");
def show as feed.Feed init feed.feedCategory(
    feed.feedAuthor(feed.feed("My Show", "https://show.example"), "Jane Host"),
    "Technology");
def rss as string init feed.build(feed.add($show, $ep), "rss");
```

Parsing

Call	Returns	Notes
feed.parse(text)	Feed	Parse feed XML, detecting RSS vs Atom from the root element.
feed.kind(text)	string	"rss" or "atom" - the detected format (errors if neither).
feed.fetch(url)	Feed	Fetch over HTTP (http.get) and parse. Non-2xx / transport errors are catchable. Needs the http module (default binary).

Parsing is **lenient about dates** : a malformed timestamp degrades to the epoch sentinel rather than failing the whole feed, and a malformed RSS document that omits `<channel>` is still read. A document whose root is neither `<rss>` nor `<feed>` is a catchable error.

Safety for untrusted feeds

Feeds come from URLs you do not control, so the untrusted-input paths are hardened:

- **Deeply-nested XML** is capped by the `xml` library's shared nesting limit (a catchable error, not a stack overflow) - see xml.
- **Entity-expansion ("billion laughs")** is impossible: `xml` decodes only the five predefined entities and numeric character references, never a DTD's custom entities.
- **Oversized bodies** are bounded: `feed.fetch` reads through `http` , which caps a response body (64 MiB) so a hostile server cannot drive the interpreter to OOM - the fetch fails with a catchable error instead.
- Parsing is **linear** in the document size; a large feed is slow, not quadratic.

For a fully-hostile source, prefer `feed.parse` on a body you have already size-bounded, and wrap the call in `try` / `catch` .

Platforms

Building and parsing are pure Jennifer over `xml` + `time` and run on **both binaries** . `feed.fetch` needs `http` (hence `net`), so it works on the default `jennifer` and returns a friendly network error on `jennifer-tiny` .

flatdb - a file-backed JSON store

`import "flatdb.j" as flatdb;`

A small JSON document store: load a file once into a value-semantic handle, query and edit it in memory through JSON Pointer, and write it back with a crash-atomic whole-file replace. Built from `json` (the data) and `fs` (the file); runs on either binary.

What it is - and isn't

`flatdb` is **not a database engine** . Honestly, it is *crash-atomic snapshotting of small data* :

- **Atomicity** - whole-file, via a temp file + rename. A reader ever sees the whole old file or the whole new one, never a torn write.
- **Consistency** - application-level (you decide what's valid).
- **Isolation** - none. One process, reload-the-whole-file, no concurrent transactions. Single-writer by construction.
- **Durability** - the rename is atomic, but flush-to-disk is OS-buffered.

For a real database, reach for a client over `net` (e.g. `redis`), not this. `flatdb` is the "embed a small store" need - config, a cache you can read, a benchmark history, a little app's saved state - where a single human-readable JSON file is exactly right.

Handle, not a connection

A module holds no mutable state and `spawn` deep-copies scope, so a store can't be a shared open connection - it's a **value** you hold:

```
export def struct DB { path as string, data as json.Value };
```

Reading verbs leave the `DB` untouched; writing verbs return a **fresh** `DB` (thread it through, the same shape lists / maps / `json` use); `save` is the only side effect.

Surface

Call	Returns	
flatdb.open(path)	DB	Load the file (an empty store if it's absent, so first run never fails).
flatdb.openString(text)	DB	Load from an in-memory JSON string - a read-only DB with no backing file (save throws). For a store fetched over the network or embedded in the program.
flatdb.get(db, pointer)	json.Value	The sub-document at a JSON Pointer ("" = the whole document).
flatdb.has(db, pointer)	bool	Whether the pointer resolves.
flatdb.keys(db, pointer)	list of string	Keys of the object at the pointer, in document order.
flatdb.length(db, pointer)	int	Element / entry count at the pointer.
flatdb.set(db, pointer, value)	DB	Upsert an object key / replace a list index (strict: no auto-vivify).
flatdb.append(db, pointer, value)	DB	Push onto the list at the pointer (create it first with set).
flatdb.remove(db, pointer)	DB	Drop the key / element at the pointer.
flatdb.save(db)	null	Write the document back to its own file, atomically (temp + rename). Throws for a read-only DB (no backing file).
flatdb.saveAs(db, path)	DB	Write the document to path and return a fresh DB bound to path (db unchanged). First dump for an openString DB, or a copy / new version of an on-disk one.

`value` is any JSON value - a `json.Value` . Build objects and lists with `json.map()` / `json.list()` (then `json.set` / `json.append` into them), and scalars with `json.decode` (`json.decode("42")` , `json.decode("\"hi\"")`). Addressing is JSON Pointer, identical to `json` 's.

Example

```
use io;
use json;
import "flatdb.j" as flatdb;

def db as flatdb.DB init flatdb.open("state.json");   # empty on first run
$db = flatdb.set($db, "/runs", json.list());

def rec as json.Value init json.map();
$rec = json.set($rec, "/cpu", "Ryzen 5 7600X3D");
$rec = json.set($rec, "/ms", 118);
$db = flatdb.append($db, "/runs", $rec);

flatdb.save($db);                                     # atomic replace

def store as flatdb.DB init flatdb.open("state.json");
io.printf("%d runs; first on %s\n",
    flatdb.length($store, "/runs"),
    json.asString(flatdb.get($store, "/runs/0/cpu")));
```

A runnable version is `examples/modules/flatdb_demo.j` (../../examples/modules/flatdb_demo.j).

Read-only from a URL or a string

`flatdb.openString(text)` loads a store from an in-memory JSON string instead of a file. flatdb stays transport-agnostic - it never imports `http` / `net` , so it remains `fs` -only and builds on both binaries - and you bring the bytes. To read a database published at a URL:

```
use io;
use json;
import "http.j" as http;
import "flatdb.j" as flatdb;

def resp as http.Response init http.get("https://example.com/config.json", {});
def db as flatdb.DB init flatdb.openString($resp.body);   # read-only
io.printf("theme: %s\n", json.asString(flatdb.get($db, "/ui/theme"), ""));
```

The result has no backing path, so every reader works and the mutating verbs still return fresh in-memory copies, but `save` throws (`kind: "flatdb"`) - there is nowhere local to write. To **persist** it (or fork any store to a new file), use `saveAs` :

```
def db as flatdb.DB init flatdb.openString($resp.body);
def edited as flatdb.DB init flatdb.set($db, "/ui/theme", json.
decode("\"dark\""));
def local as flatdb.DB init flatdb.saveAs($edited, "config.local.json");
flatdb.save($local);   # now writable - $local is bound to config.local.json
```

`saveAs(db, path)` writes to `path` and returns a **fresh DB bound to that path** ; the `db` you passed in is unchanged (value semantics, like `set` / `append` / `remove`). So after a `saveAs` you hold *both* - which one is "current" for the next `save` is simply the handle you keep: reassign (`$db = flatdb.saveAs($db, path);`) to make the new file current, or keep the old handle to keep writing the original file. There is no hidden "active file" state; a `DB` is a value.

Decoding untrusted input is safe against a nesting bomb: `json.decode` caps container depth (a deeply-nested payload is a **catchable** error, not a fatal stack overflow), and JSON has no entity/alias expansion to amplify size - so a malicious URL can't crash or blow up the process. Bound the *download* itself with `http` 's body cap (64 MiB by default; pass an explicit `maxBytes` to `http.requestWith` for more or less).

Atomic save, in detail

`save` writes the encoded document to a sibling `path` + `".tmp"` and then `fs.rename` s it over the target. On POSIX the rename is atomic, so a concurrent reader never sees a half-written file. If the process dies mid-save (temp written, rename not reached), the original file is untouched - only a stray `.tmp` remains, which the next `save` overwrites. Durability past the rename is the OS's call (there is no `fsync` today).

See also

- `json` - the value model and write surface `flatdb` layers over.
- `fs` - the file I/O (`readString` / `writeString` / `rename`) behind `open` / `save` .
- `redis` - a real store, over the network, when you outgrow a single file.

font - TrueType / OpenType font parsing

Import with `import "font.j" as font;` . A **pure-Jennifer** TrueType / OpenType parser: read a `.ttf` / `.otf` from `bytes` and expose its metrics, character map, and glyph outlines - no Go, no dependency, just the `bytes` type and the bitwise operators for the big-endian tables, so it runs on **both binaries** .

Both outline backends ship: the **TrueType glyf** backend (simple + composite glyphs, quadratic curves) and the **CFF / PostScript** backend for OpenType `OTTO` fonts (a Type2 charstring interpreter with global / local subroutines and CID-keyed FDArray / FDSelect, so CJK fonts outline too), detected on parse. It parses `head` , `cmap` (formats 4 and 12), `maxp` / `hhea` / `hmtx` , `OS/2` (vertical metrics), the legacy `kern` table, `loca` / `glyf` or `CFF` , and `name` .

```
use io;
import "font.j" as font;

def f as font.Font init font.open("/usr/share/fonts/TTF/DejaVuSans.ttf");
io.printf("%s, %d upem\n", font.name($f), font.unitsPerEm($f));
io.printf("<path d=\"%s\"/>\n", font.glyphPath($f, 65));   # outline of 'A'
```

Runnable: `examples/modules/font_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/font_demo.j) (a word rendered to an SVG).

Surface

Call	Returns	Notes
font.parse(b)	Font	Parse a font from bytes (.ttf or .otf).
font.open(path)	Font	Read a font file and parse it.
font.unitsPerEm(f)	int	The em-square size - the coordinate scale of every metric and outline.
font.name(f)	string	The font family name.
font.advance(f, codepoint)	int	The horizontal advance width of a codepoint's glyph, in font units.
font.kern(f, left, right)	int	The kern-table pair adjustment between two codepoints (0 when absent).
font.ascender(f)	int	Typographic ascender (OS/2 sTypoAscender, else hhea).
font.descender(f)	int	Typographic descender (negative).
font.lineGap(f)	int	Typographic line gap (line height = ascender - descender + lineGap).
font.capHeight(f)	int	Cap height (OS/2 sCapHeight; 0 when unavailable).
font.xHeight(f)	int	x-height (OS/2 sxHeight; 0 when unavailable).
font.glyphPath(f, codepoint)	string	The glyph outline as an SVG path d string.
font.glyph(f, codepoint)	Glyph	The raw outline: contours of on / off-curve points, advance, and bounding box.

A codepoint the font lacks maps to glyph 0 (`.notdef`).

Coordinates and outlines

Everything is in **font units** - divide by `unitsPerEm` (typically 1000 or 2048) to get em fractions, then multiply by your point size. Fonts store **y pointing up** , so flip y for screen rendering (an SVG `transform="scale(1,-1)"` around the whole word does it).

`font.glyphPath` emits an SVG path: `M` / `L` for straight segments, `Q` (quadratic Bezier) for TrueType curves,

and `C` (cubic Bezier) for CFF curves - each backend's native curve type. For the raw outline, `font.glyph` returns a `Glyph` (whose points are always quadratic - a CFF cubic is approximated as two quadratics there, while `glyphPath` keeps the exact cubic):

- `Glyph { advance, xMin, yMin, xMax, yMax, contours as list of Contour }`
- `Contour { points as list of Point }`
- `Point { x, y, onCurve }` - an off-curve point is a quadratic control point.

Composite glyphs (an accented letter built from a base plus a mark) are resolved into a single set of contours, with each component's translation and scale applied.

Laying out a string

Walk the string, outline each glyph, and advance the pen by its advance width:

```
def x as int init 0;
for (def i as int init 0; $i < len($chars); $i = $i + 1) {
    def cp as int init charCode($chars[$i]);
    render(font.glyphPath($f, $cp), $x);              # your renderer, offset by x
    $x = $x + font.advance($f, $cp);
}
```

Pair kerning from the legacy `kern` table is available via `font.kern(f, left, right)` (add it to the pen advance); vertical metrics come from `font.ascender` / `descender` / `lineGap` / `capHeight` / `xHeight` .

```
$x = $x + font.advance($f, $prev) + font.kern($f, $prev, $cp);   # kerned advance
```

Scope

In: TrueType `glyf` and OpenType `CFF` outlines (including CID-keyed CJK fonts, subroutines, and the flex operators), `cmap` formats 4 and 12, `hmtx` advances, OS/2 vertical metrics, and legacy `kern` -table pair kerning. **Out:** kerning / shaping from `GPOS` / `GSUB` (modern fonts put kerning there; the legacy `kern` table is read), hinting, colour / emoji tables, `CFF2` and variable-font axes.

A hostile font cannot hang the outliner: charstring subroutine recursion is bounded (a runaway self-call raises a catchable `font` error, not an infinite loop), the composite-glyph depth is capped, and every table loop is finite.

The tests parse tiny committed fixtures (`modules/testdata/font_fixture.ttf` and `font_fixture_cff.otf` , regenerable with `scripts/gen-font-fixture.py`); the CFF backend is cross-checked against `fontTools` (pixel-exact outlines, IoU 1.0) over ~170 glyphs on real system fonts, including a CID-keyed CJK font, and a hand-crafted recursive font pins the guard.

Performance

Parsing is cheap - even a multi-megabyte CJK CFF font opens in ~100 ms, because `parse` only reads the table directory and headers. Glyph access scales with the glyph, not the font: a single CharStrings / subr INDEX entry is read in O(1) (never walking the whole 65k-glyph INDEX), the format-12 cmap is binary-searched, and CFF subroutines are fetched on demand - so a CJK glyph outlines in a fraction of a second. There is no need to cache or serialise a parsed `Font` : it is just the raw file bytes plus a small table of offsets, so re- `open` ing the file is as fast as reloading any cache would be.

gotify - push notifications to a Gotify server

Import with `import "gotify.j" as gotify;` . A tiny module on top of the `http` client that pushes a notification to a Gotify (https://gotify.net) server. Hold a value-semantic `Config` (server URL

- application token) and call `push` . Because it builds on `http` (which uses

`net`), this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "gotify.j" as gotify;
import "http.j" as http;

def g as gotify.Config init gotify.Config{url: "https://push.example.com",
    token: "AqB3cD..."};
def r as http.Response init gotify.push($g, "Deploy", "build 1234 is live", 5);
io.printf("pushed -> %d\n", $r.status);      # 200 on success
```

Runnable: `examples/modules/gotify_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/gotify_demo.j).

Surface

Call / type	Notes
gotify.Config	url (server, no trailing slash) and token (application key).
gotify.Extras	markdown (bool) and clickUrl (string) - optional message extras.
gotify.push(cfg, title, message, priority)	POST the message form; returns the http.Response.
gotify.pushMarkdown(cfg, title, message, priority)	Render the body as markdown; returns the http.Response.
gotify.pushWith(cfg, title, message, priority, url)	Open url when the notification is tapped; returns the http.Response.
gotify.pushExtras(cfg, title, message, priority, ex)	Attach an arbitrary Extras (markdown and/or click URL).

`push` POSTs `title` / `message` / `priority` as `application/x-www-form-urlencoded` to `cfg.url` + `"/message"` with an `X-Gotify-Key: cfg.token` header - Gotify's push-message (https://gotify.net/docs/pushmsg) contract, where priority (https://gotify.net/docs/priority) is a plain int (0 lowest, higher is more urgent). It returns the raw `http.Response` , so the caller checks `.status` : a `200` on success, and a bad token comes back as a `4xx` **value** , not a crash.

Message extras (markdown, click action)

Gotify's extras (https://gotify.net/docs/msgextras) let a message carry client hints. Because extras are a nested object, the extras-carrying variants POST a **JSON** body (`application/json`) instead of the form - the plain `push` stays form-encoded and unchanged. Two extras are supported:

- **markdown** - `pushMarkdown` (or `Extras{markdown: true}`) sets `extras["client::display"]["contentType"]` = `"text/markdown"` , so the body renders as markdown.
- **click action** - `pushWith(..., clickUrl)` (or `Extras{clickUrl: url}`) sets `extras["client::notification"]["click"]["url"]` = `clickUrl` , so tapping the notification opens that URL.

Combine both with `pushExtras` and an `Extras` holding both fields. A zero `Extras` (markdown false, empty `clickUrl`) adds no `extras` object, so `pushExtras` with it matches a plain `push` (over JSON).

```
import "gotify.j" as gotify;

def g as gotify.Config init gotify.Config{url: "https://push.example.com", token:
"tok"};

# markdown body
def a as http.Response init gotify.pushMarkdown($g, "Report", "# Done\n- built\n-
shipped", 5);

# tap-to-open URL
def b as http.Response init gotify.pushWith($g, "Deploy", "build 1234 is live", 5,
    "https://ci.example.com/builds/1234");

# both at once
def ex as gotify.Extras init gotify.Extras{markdown: true, clickUrl: "https://ci.
example.com/builds/1234"};
def c as http.Response init gotify.pushExtras($g, "Deploy", "**build 1234** is
live", 5, $ex);
```

The JSON body `pushExtras` builds for the combined case is:

```
{"title":"Deploy","message":"done","priority":8,"extras":{"client::display":{"cont
entType":"text/markdown"},"client::notification":{"click":{"url":"https://x.
example/go"}}}}
```

Stateless by design

There is no `init()` that stashes the URL and token - a module has no mutable state. The caller holds the value-semantic `Config` and passes it to each `push` , the same shape as the `time` / `hash` structs. The URL and token are **yours to supply and never commit** : read them from the environment or a config file. The demo reads `GOTIFY_URL` / `GOTIFY_TOKEN` from the environment; the docs use placeholders.

See also

- http.md - the client `gotify` posts through.
- modules/index.md - the module catalog and import rules.

gpio - Raspberry-Pi GPIO over sysfs

`import "gpio.j" as gpio;`

General-purpose I/O pins on a Raspberry Pi (or any Linux single-board computer) through the sysfs interface - the physical-computing / IoT-teaching use case. `/sys/class/gpio` is plain files, so `fs` is the entire backend; there is no core change, no system library, and no build tag. Blink an LED from a five-line script:

```
import "gpio.j" as gpio;
gpio.setup(17, gpio.OUT);
gpio.write(17, 1);   # LED on
gpio.write(17, 0);   # LED off
gpio.release(17);
```

Surface

Stateless and pin-keyed - sysfs derives every path from the pin number, so no handle is needed:

Call	Returns	
gpio.setup(pin, direction)	null	Export pin and set its direction: gpio.IN or gpio.OUT.
gpio.write(pin, value)	null	Set an output pin's value: 0 or 1.
gpio.read(pin)	int	Read a pin's current value (0 / 1).
gpio.release(pin)	null	Unexport pin.

The direction is passed as one of two exported constants, `gpio.IN` / `gpio.OUT` (matching RPi.GPIO's `GPIO.IN` / `GPIO.OUT`). Their values are the sysfs strings `"in"` / `"out"` , so a raw string still works, but the constant is the readable form. A bad direction (not `gpio.IN` / `gpio.OUT`) or a value other than `0` / `1` throws `Error{kind: "gpio"}` . When the sysfs GPIO tree is absent - not a GPIO-capable host, or sysfs GPIO disabled - every call throws a clear positioned `Error{kind: "gpio"}` ("base directory not found: ...") rather than crashing.

The sysfs root

The root defaults to `/sys/class/gpio` . Set the **JENNIFER_GPIO_BASE** environment variable to point elsewhere - a differently mounted sysfs, or a mock tree under test:

```
use os;
os.setEnv("JENNIFER_GPIO_BASE", "/tmp/mock-gpio");   # e.g. in a test
```

That is how `gpio_test.j` (../../modules/gpio_test.j) drives the module against a temp-dir mock, and how `gpio_demo.j` (../../examples/modules/gpio_demo.j) runs on any machine (no hardware, no root) while making the same calls you'd run on a Pi.

Why a module, not a system library

`use` / `import` are static - they resolve before execution, are uncatchable, and can't be conditional - so there is no "check the platform, then import." The portability seam is instead **which module file is on the search path** (Go uses build tags; Jennifer uses the module file). A program writes one uniform line, `import "gpio.j" as gpio;` , and the deployment supplies the right `gpio.j` : this sysfs module on a Pi, or an emulator that blinks a console cell on a laptop. A genuinely *platform* -bound capability being **absent** off its platform (so `import` fails at the top with a clear message) is the right shape

- distinct from a *toolchain* -bound one like `net` , which stubs on

`jennifer-tiny` because the same source must load in both binaries.

sysfs, and the future

sysfs GPIO is deprecated in favour of the `/dev/gpiochip` character device and can be compiled out of a

kernel. The bet is that it stays available on the hobbyist Pi kernels this targets. The API is kept deliberately stable, so if sysfs is ever removed the backend can be swapped for a `/dev/gpiochip` ioctl system library with **no change to .j scripts** - the pure-module form is the default because it costs the language nothing; the system library is future-proofing, taken only when forced.

That system library now exists as the `gpio` library (the `/dev/gpiochipN` GPIO v2 ioctls). It reuses this module's pin-keyed shape - `setup` / `read` / `write` / `release` plus the `IN` / `OUT` constants - so a script moves between the two unchanged; this sysfs module stays the portable default, and the library is the successor for targets that ship without sysfs GPIO.

See also

- `gpio` library - the `/dev/gpiochip` character-device successor with the same pin-keyed surface (default `jennifer` binary, Linux).
- `fs` - the file I/O behind every call.
- `os` - `os.setEnv` to point `JENNIFER_GPIO_BASE` at a non-standard sysfs mount or a mock.

graphql - a thin GraphQL client

Import with `import "graphql.j" as graphql;` . A small GraphQL client over `http` / `rest` : point a `Client` at one endpoint, then `query` POSTs `{"query": ..., "variables": ...}` and returns the decoded JSON response as a `json.Value` . **Default jennifer binary only** (net-backed).

```
import "graphql.j" as graphql;
use json;
use io;

def gql as graphql.Client init graphql.bearer(
    graphql.client("https://api.github.com/graphql"), $token);

def vars as json.Value init json.set(json.map(), "/login", "octocat");
def resp as json.Value init graphql.query($gql,
    "query($login:String!){ user(login:$login){ name } }", $vars);

io.printf("%s\n", json.asString($resp, "/data/user/name"));
```

The query is an **opaque string** you supply - GraphQL syntax is the server's job, not this module's - and a mutation is just a query string, so there is no separate verb. `variables` is a `json.Value` object (use an empty `json.map()` when the operation takes none).

Runnable: `examples/modules/graphql_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/graphql_demo.j).

The client

`graphql.Client` wraps a `rest.Client` , so it carries the endpoint URL, per-request headers, and TLS options. It is value-semantic: each builder returns a **new** client, nothing is mutated.

Call	Returns	Notes
graphql.client(endpoint)	Client	A client for one endpoint URL. The URL is POSTed to verbatim - pass the complete GraphQL URL (e.g. https://host/graphql); no path is appended.
graphql.bearer(c, token)	Client	Add Authorization: Bearer <token>.
graphql.basic(c, user, pass)	Client	Add HTTP Basic Authorization.
graphql.header(c, name, value)	Client	Set an arbitrary request header.
graphql.withCA(c, pem)	Client	Trust a private-CA / self-signed certificate (PEM); verification stays on, against this CA.
graphql.insecure(c)	Client	Skip TLS verification - a trusted network or a local test server only.

Builders compose left to right - each takes a `Client` and returns one:

```
def gql as graphql.Client init graphql.header(
    graphql.bearer(graphql.client("https://host/graphql"), $token),
    "X-Api-Version", "2");
```

Running a query

Call	Returns	Notes
graphql.query(c, query, variables)	json.Value	POST the operation; raise on GraphQL / HTTP errors; return the full decoded response (result under /data).

graphql.queryNamed(c, query, variables, operationName)	json.Value	query, selecting one operation from a multi-operation document by name.
graphql.tryQuery(c, query, variables)	json.Value	Like query but does not raise on GraphQL errors - returns the envelope so you inspect /errors yourself. Still raises on a non-2xx HTTP status.
graphql.tryQueryNamed(c, query, variables, operationName)	json.Value	tryQuery with an operationName.
graphql.hasErrors(resp)	bool	Whether a returned envelope carries a non-empty errors array.
graphql.errorMessages(resp)	string	The envelope's error messages, joined with ; .

```
def resp as json.Value init graphql.query($gql, "{ viewer { login } }", json.
map());
def login as string init json.asString($resp, "/data/viewer/login");
```

The response is the whole GraphQL envelope, so read the payload under `/data` (and `/data/<field>/...`) with the `json` accessors.

`variables` is a `json.Value` object - pass an empty `json.map()` for an operation that takes none.

Named operations

When a document defines more than one named operation, pass the one to run as `operationName` (some servers require it in that case):

```
def q as string init "query Me { viewer { login } } query Repos { viewer {
repositories { totalCount } } }";
def resp as json.Value init graphql.queryNamed($gql, $q, json.map(), "Repos");
```

A single-operation document needs no name - use plain `query` / `tryQuery`.

Errors

GraphQL has an unusual error convention that this client handles for you: an **execution error is an HTTP 200 with a top-level errors array**, not a non-2xx status. `query` inspects the payload rather than trusting the status line and raises a positioned `Error` (kind `"graphql"`) in two cases:

- **GraphQL errors** - the response carries a non-empty `errors` array. The error's `message` is `graphql:` followed by every entry's `message`, joined with `;`. (A response with partial `data` *and* `errors` still raises - the errors are not silently dropped.)
- **HTTP failure** - a non-2xx status (a transport, auth, or server problem). The message carries the status code and the response body.

A successful return therefore has no `errors`, so `/data` is present and complete. Catch with `try` / `catch`:

```
try {
    def resp as json.Value init graphql.query($gql, $q, $vars);
    # ... use $resp ...
} catch (e) {
    io.eprintf("query failed: %s\n", $e.message);   # e.kind == "graphql"
}
```

Handling GraphQL errors yourself

The thrown `Error` carries only the joined message text. When you need the **structured** error - to branch on an error `code`, read partial `data`, or see `path` / `locations` - use `tryQuery`, which returns the raw envelope instead of raising on GraphQL errors (it still raises on a non-2xx HTTP status, where there is no envelope):

```
def resp as json.Value init graphql.tryQuery($gql, $q, $vars);
if (graphql.hasErrors($resp)) {
    def code as string init json.asString($resp, "/errors/0/extensions/code");
    if ($code == "RATE_LIMITED") {
        # ... back off and retry ...
    } else {
        io.eprintf("graphql: %s\n", graphql.errorMessages($resp));
    }
} else {
    def login as string init json.asString($resp, "/data/viewer/login");
    # ...
}
```

Everything under `/errors` is reachable with the ordinary `json` accessors, since the envelope is a plain `json.Value` .

TLS to a self-signed host

Because the client wraps `rest` , the TLS builders reach a homelab / private-CA endpoint (Hasura, a self-hosted API):

```
def pem as bytes init fs.readBytes("ca.pem");
def gql as graphql.Client init graphql.withCA(
    graphql.client("https://hasura.home.lan/v1/graphql"), $pem);
```

Use `graphql.insecure(c)` only on a trusted network or against a local test server - it accepts any certificate.

See also

- rest.md - the REST layer the client wraps (auth + TLS options).
- http.md - the underlying HTTP/1.1 client.
- json.md - reading the response and building `variables` .
- modules/index.md - the module catalog and import rules.

html - build and render an HTML tree

Import with `import "html.j" as html;` . Assembles an HTML element tree and renders it to a correctly-escaped HTML5 string. Pure Jennifer over `strings` and `lists` , so it runs on either binary. It is a **writer, not a parser** - serialization is a handful of string operations, so it has no dependency on an XML parser. It is the shared output layer an HTML-emitting consumer reuses (a Markdown renderer, a documentation generator, a view layer).

```
use io;
import "html.j" as html;

def kids as list of html.Node init [];
$kids[] = html.text("hi & bye");
def p as html.Node init html.element("p", [], $kids);
io.printf("%s\n", html.render($p));          # <p>hi &amp; bye</p>
```

Runnable: `examples/modules/html_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/html_demo.j).

The node model

An HTML tree is `Node` values. A node is one of three kinds, tagged by its `kind` field (a `NodeKind` enum), and is built with a constructor rather than a literal:

Kind	Built with	Renders as
NodeKind.Element	element(tag, attrs, children)	<tag ...>children</tag> (or a void tag)
NodeKind.Text	text(s)	s, HTML-escaped
NodeKind.Raw	raw(s)	s, verbatim (already-trusted markup only)

```
export def enum NodeKind { Element, Text, Raw };
export def struct Node {
    kind as NodeKind, tag as string,
    attrs as list of Attr, children as list of Node, text as string
};
export def struct Attr { name as string, value as string, boolean as bool };
```

`render` dispatches over `NodeKind` with an exhaustive `match` , so a new node kind cannot be added without handling it in the renderer.

Children are supplied to `element` as a `list of Node` you build first (the append sugar does not chain into a struct field, so build the list in a variable, then pass it). Attributes are a `list of Attr` built with `attr` (a normal `name="value"` attribute) or `boolAttr` (a valueless boolean attribute).

Surface

Call	Returns	Notes
html.element(tag, attrs, children)	Node	An element node. Pass [] for no attributes or no children.
html.text(s)	Node	A text node; s is HTML-escaped on render.
html.raw(s)	Node	A verbatim node; s is not escaped. Trusted markup only.
html.attr(name, value)	Attr	One attribute; value is escaped in attribute context on render.
html.boolAttr(name)	Attr	A boolean (valueless) attribute; renders as the bare name (disabled).
html.render(node)	string	Serialize a node and its subtree to HTML5.
html.renderAll(nodes)	string	Serialize a list of Node fragment in order.

html.escape(s)	string	HTML-escape a bare string for text context (the helper render uses).
html.safeUrl(url)	string	The URL if its scheme is http / https / mailto, else "#" (anti-XSS href/src gate).

Escaping

Escaping is automatic and context-aware, so a value is escaped exactly once:

- **Text nodes** escape `&` , `<` , `>` (with `&` first, so an existing entity is not double-escaped).
- **Attribute values** additionally escape `"` , since they render inside double quotes.
- **raw nodes** are emitted verbatim - the escape hatch for markup you have already produced (an SVG blob, a rendered sub-tree). Only pass trusted content.

```
def a as list of html.Attr init [];
$a[] = html.attr("title", "a \"b\" <c>");
io.printf("%s\n", html.render(html.element("span", $a, [])));
# <span title="a &quot;b&quot; &lt;c&gt;"></span>
```

`html.escape(s)` exposes the text-context escaper on its own, for when you need an escaped string without building a node.

Boolean attributes

HTML boolean attributes - `disabled` , `checked` , `selected` , `required` , `readonly` , `multiple` , `autofocus` , and the like - carry no value: their presence alone is the truth. Build one with `html.boolAttr(name)` and it renders as the bare name, not `name=""` . Normal and boolean attributes mix freely on one element, in the order you append them; a boolean-attribute name is validated exactly like an `attr` name.

```
def a as list of html.Attr init [];
$a[] = html.attr("type", "checkbox");
$a[] = html.boolAttr("checked");
$a[] = html.boolAttr("required");
io.printf("%s\n", html.render(html.element("input", $a, [])));
# <input type="checkbox" checked required>
```

Safe tag / attribute names

`html.element` and `html.attr` validate the tag / attribute name against `[A-Za-z][A-Za-z0-9-]*` and throw an `Error` of kind `"html"` on anything else. A name is markup structure, not data, so unlike a value it cannot be escaped: `html.attr("x onclick=alert(1)", "y")` or a tag built from an untrusted string would otherwise inject a live attribute or element. Legal names (`"data-id"` , `"h1"` , `"aria-label"`) build as before; a name carrying a space, `>` , `=` , `/` , or a quote is rejected at construction.

Safe URLs in links

`html.safeUrl(url)` returns the URL when its scheme is `http` , `https` , or `mailto` , and `"#"` otherwise, so a `javascript:` (or `data:`) URL built from untrusted input cannot become a live `href` / `src` . Whitespace and control characters are ignored while reading the scheme, so `"java\tscript:..."` is still caught; a relative reference (no scheme) is returned unchanged.

```
$a[] = html.attr("href", html.safeUrl($userUrl));   # "#" if $userUrl is
javascript:...
```

Void elements

The HTML5 void elements - `area` `base` `br` `col` `embed` `hr` `img` `input` `link` `meta` `param` `source` `track` `wbr` - render with no closing tag, and any children passed to them are dropped (they cannot have content). The tag is

matched case-insensitively.

```
def a as list of html.Attr init [];
$a[] = html.attr("src", "logo.png");
io.printf("%s\n", html.render(html.element("img", $a, [])));   # <img src="logo.
png">
```

Fragments

`render` serializes a single node; `renderAll` serializes a list of sibling nodes with no wrapping element - a document fragment:

```
def parts as list of html.Node init [];
$parts[] = html.element("h1", [], heading);
$parts[] = html.element("hr", [], []);
io.printf("%s\n", html.renderAll($parts));
```

Parsing

`html.parse(src)` reads an HTML string into the **same Node tree** the builders produce, so a document can be walked, edited, and re-rendered with `render` - build and parse round-trip through one model. The result is a synthetic `#root` element whose `children` are the document's top-level nodes:

```
def doc as html.Node init html.parse("<ul class=fruit><li>apples</li><li>figs</
li></ul>");
io.printf("%s\n", html.attrOf(html.get($doc, "ul"), "class"));   # fruit
for (def li in html.findAll($doc, "ul/li")) {
    io.printf("%s\n", $li.children[0].text);                      # apples, figs
}
```

Because `Node` is a plain struct, read `.tag` / `.attrs` / `.children` / `.text` directly; the query helpers add attribute lookup and tree search:

Call	Returns
html.attrOf(node, name)	an attribute's value, or ""
html.hasAttr(node, name)	whether the attribute is present
html.get(node, sel)	the first element matching sel (an empty node if none)
html.findAll(node, sel)	every element matching sel
html.has(node, sel)	whether any element matches sel

A selector is an XPath-ish `/` -path; each step is a tag name, `*` (any element), or `name[k]` (the k-th such child, 1-based). Steps match **direct element children** (text nodes are skipped), so `"article/ul/li[2]"` is the second `li` under the `ul` under the `article` .

The parser is **tolerant** of real-world HTML: void elements (`<br>` / `<img>`), self-closing tags, unquoted (`id=x`) and boolean (`disabled`) attributes, mismatched nesting (`<b>x</i>` closes the `<b>`), comments, `<!DOCTYPE>` , and `script` / `style` raw text are handled, not rejected; the common entities (`&` `<` `>` `"` `'`) decode. A nesting-depth and node budget make a malicious document a catchable `Error{kind: "html"}` rather than an unbounded parse.

Out of scope

This module **writes** HTML; it does not parse it (parsing arbitrary HTML is a separate, much larger job and would want the `xml` system library). There is no pretty-printing / indentation pass - output is compact, which round-trips and diffs predictably; wrap it in your own formatter if you need indented source. A `<!DOCTYPE html>` prologue is not emitted; prepend `html.raw("<!DOCTYPE html>")` (or a literal string) when you need a full document.

See also

- strings.md - `replace` / `lower` , which the escaping and void-element check build on.
- lists.md - `contains` , used for the void-element lookup.
- modules/index.md - the module catalog and import rules.

http - an HTTP/1.1 client

Import with `import "http.j" as http;` . An **HTTP/1.1 client** over the `net` system library: build a request (method, URL, headers, body), send it, and read the response back into a `Response` (status, headers, body). `http://` connects in the clear; `https://` connects with TLS (`net.connectTLS`). Because it uses `net` , this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "http.j" as http;

def r as http.Response init http.get("http://example.com/", {});
io.printf("status %d\n%s\n", $r.status, $r.body);

def sent as http.Response init http.post("https://api.example.com/items",
    "application/json", "{\"name\":\"ada\"}", {"Authorization": "Bearer xyz"});
```

Runnable: `examples/modules/http_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/http_demo.j).

Surface

`headers` is a `map of string to string` (pass `{}` for none); a `body` is a string (`""` for none).

Call / type	Notes
http.Response	status (int), statusText, headers (lowercased keys), body (string, UTF-8).
http.BytesResponse	Same, but body is raw bytes - the byte-safe response for binary downloads.
http.TlsOptions	skipVerify (bool), caCert (bytes, PEM). Zero value = full verification (what request uses).
http.request(method, url, headers, body)	The general request (default idle timeout); returns a Response.
http.requestWith(method, url, headers, body, timeoutMs, maxBytes)	As request, with an explicit per-read idle timeout (0 = none) and body cap (0 = 64 MiB default, negative = unlimited, positive = exact ceiling).
http.requestTls(method, url, headers, body, tls)	As request, with explicit TlsOptions for an https:// server (self-signed / private CA).
http.requestWithTls(method, url, headers, body, timeoutMs, maxBytes, tls)	As requestWith, with explicit TlsOptions.
http.requestBytes(method, url, headers, body)	Byte-safe request: returns a BytesResponse (raw bytes body) - for binary content.
http.requestWithBytes(method, url, headers, body, timeoutMs, maxBytes, tls)	As requestBytes, with explicit timeout / cap / TlsOptions (pass negative maxBytes for a large download).
http.requestRawBody(method, url, headers, body, timeoutMs, maxBytes)	Send a raw bytes body byte-for-byte (a multipart/form-data file upload, a protobuf), not UTF-8-encoded; returns a text Response. Set your own Content-Type.
http.requestRawBodyTls(method, url, headers, body, timeoutMs, maxBytes, tls)	As requestRawBody, with explicit TlsOptions.
http.getBytes(url, headers)	GET returning a raw-bytes body (the download shortcut).
http.get(url, headers)	GET.
http.post(url, contentType, body, headers)	POST; sets Content-Type.

http.put(url, contentType, body, headers)	PUT; sets Content-Type.
http.patch(url, contentType, body, headers)	PATCH (partial update); sets Content-Type.
http.delete(url, headers)	DELETE.
http.head(url, headers)	HEAD (status + headers, no body).
http.options(url, headers)	OPTIONS (capability probe; read the Allow header).
http.header(resp, name)	Read a response header case-insensitively, or "" if absent.
http.basic(user, pass)	Build a Basic <base64> Authorization value (mirrors rest.basic).
http.Options	Request policy: timeoutMs, maxBytes, maxRedirects, maxRetries, backoffMs, tls, allowCrossOriginRedirect. Zero value = defaults, no redirects, no retries, and credentials dropped on a cross-origin redirect.
http.defaultOptions()	The zero Options, for inline use.
http.send(method, url, headers, body, options)	One request with policy: follow up to maxRedirects 3xx, retry a 429 / 5xx up to maxRetries with backoff, carry cookies across the redirect chain. A redirect to a different origin drops Authorization / Cookie and the cookie jar (the browser "sensitive headers" rule) unless allowCrossOriginRedirect is set. Returns a Response.
http.Session	A persistent (keep-alive) connection to one origin, with a cookie jar.
http.Exchange	response (Response) + session (Session) to thread onward.
http.connect(url, options)	Open a persistent Session to the origin of url.
http.exchange(session, method, path, headers, body)	One request over the reused socket; returns an Exchange.
http.close(session)	Close a Session's connection.

The response body is bounded by default: `http` reads at most **64 MiB** (`MAX_BODY_BYTES`) and raises a catchable error beyond it. The per-read timeout only bounds a *stalled* server, so this size cap is what keeps a hostile server (e.g. behind an untrusted URL a feed reader follows) from streaming an unbounded body to OOM. A caller that needs a larger (or unbounded) body passes an explicit `maxBytes` to `http.requestWith` - `0` keeps the 64 MiB default, a negative value lifts the cap for a trusted large download, a positive value sets an exact ceiling. The `s3` module (object storage) already lifts it; the verb shortcuts (`get` / `post` / ...) and `request` keep the default.

The shortcuts are thin wrappers over `request` , which is **method-agnostic** - it sends whatever method string you pass. So a method without a shortcut still works: `http.request("TRACE", url, {}, "")` and the like. The one method that is **not** supported is `CONNECT` : it is the HTTP tunneling primitive (after a `200` the socket becomes a raw bidirectional tunnel), which needs a connection hand-off this request/response-then-close client does not do.

Request policy: redirects, retries, cookies

`http.send` is the one-shot verbs plus a policy, configured by an `http.Options` . The zero `Options` (`http.defaultOptions()`) behaves exactly like `request` ; set a field to opt in:

```
def o as http.Options;
$o.maxRedirects = 5;   # follow up to 5 3xx redirects
$o.maxRetries = 3;     # retry a 429 / 5xx up to 3 times
$o.backoffMs = 250;    # first backoff (doubles per attempt), honours Retry-After
def r as http.Response init http.send("GET", "https://example.com/", {}, "", $o);
```

- **Redirects.** A `301` / `302` / `303` / `307` / `308` with a `Location` is followed up to `maxRedirects` hops (`0` returns the 3xx as-is). A `303` (and a `301` / `302` on a `POST`) becomes a bodyless `GET` ; `307` / `308` preserve the method and body. Cookies set along the chain are replayed on later hops.

- **Retries.** A `429` or `5xx` is retried up to `maxRetries` with exponential backoff (`backoffMs * 2^attempt`), raised to a numeric `Retry-After` when the server sends a larger one, and clamped to 30s.

For **Basic auth** , `http.basic(user, pass)` builds the header value:

```
def r as http.Response init http.get(url, {"Authorization": http.basic("ada",
"s3cret")});
```

Persistent connections (keep-alive)

`http.connect` opens a `Session` to one origin (scheme + host + port) and reuses the socket across `exchange` calls, so a request loop pays a single handshake instead of one per request. The `Session` is value-semantic, but its underlying `net.Conn` is a shared handle - thread the returned session forward:

```
def s as http.Session init http.connect("https://api.example.com", http.
defaultOptions());
def x1 as http.Exchange init http.exchange($s, "GET", "/items?page=1", {}, "");
$s = $x1.session;                 # carry the reused socket + any cookies
def x2 as http.Exchange init http.exchange($s, "GET", "/items?page=2", {}, "");
$s = $x2.session;
http.close($s);
```

`exchange` maintains a small **cookie jar** on the session (name -> value; it preserves multiple `Set-Cookie` lines and replays them as one `Cookie` header) and reconnects transparently if the server closed the connection. It does **not** follow redirects (a redirect can cross origins and break the socket) - use `http.send` for that; `exchange` returns the 3xx.

The jar is deliberately small: no domain / path scoping and no expiry (a session-scoped `name -> value` store). A full RFC 6265 jar is a follow-up.

URLs and headers

A URL is parsed into scheme / host / port / path: `http://` defaults to port 80, `https://` to 443, an explicit `:port` overrides, and the path (with any query string) defaults to `/` . The `Host` header is set automatically (with the port when non-default), along with `Connection: close` and a default `User-Agent` (overridable by supplying your own).

Response header names are **lowercased** (HTTP header names are case-insensitive), so `$r.headers["content-type"]` works regardless of how the server cased it; `http.header($r, "Content-Type")` does the case-folding for you.

Response body and framing

The client reads the whole response (it sends `Connection: close` , so the server closes when done) and decodes the body, handling both framings:

- **Content-Length** - the body is taken as exactly that many bytes.
- **Transfer-Encoding: chunked** - the chunks are decoded and concatenated.

`request` and the verb shortcuts return the body as **text** (UTF-8): a JSON / HTML / XML body round-trips exactly, but a binary body (an image, a gzip stream) is not valid UTF-8 and raises an error. For binary content use the **byte verbs** below, which return the body as raw `bytes` - the framing above is byte-exact, so only the text decode differs.

Binary downloads

`http.getBytes(url, headers)` (and the general `requestBytes` / `requestWithBytes`) return an `http.BytesResponse` whose `body` is raw `bytes` - the byte-safe path for a `.tar.gz` , an image, or any non-text payload the string `Response` cannot hold:

```
use fs;
use compress;
use archive;
def r as http.BytesResponse init http.getBytes("https://example.com/app.tar.gz",
{});
fs.writeBytes("app.tar.gz", $r.body);                  # exact bytes, no
corruption
def entries as list of archive.Entry init archive.unpack(compress.unpack($r.body,
"gzip"), "tar");
```

`headers` keys are lowercased, so read metadata directly (`$r.headers["content-type"]`). A large download that would exceed the default 64 MiB cap uses `requestWithBytes(..., maxBytes, tls)` with a **negative** `maxBytes` (unbounded), and the same call carries `TlsOptions` for a self-signed host.

Timeouts

The initial connect (and, for `https` , the TLS handshake) is bounded by a connection-establishment timeout, so a slow or unreachable server fails the dial instead of blocking it forever. Every request then carries a **per-read idle timeout** (default 30 s): the deadline is re-armed before each read, so a server that accepts the connection and then stalls (or a hung endpoint) fails with a catchable `read timed out` error instead of blocking the caller forever. This is the difference between a slow dependency degrading one request and one exhausting your process on a pool of hung connections. Pass a different value (in milliseconds) with `http.requestWith` ; a `0` disables the timeout for that request (e.g. a long streaming download):

```
try {
    def r as http.Response init http.requestWith("GET", url, {}, "", 5000);   # 5
s
} catch (e) {
    io.printf("request timed out or failed\n");
}
```

The timeout bounds each read, not the whole transfer, so a large but steady download is fine while a stalled one is cut off.

TLS (self-signed / private CA)

An `https://` request full-verifies the server certificate against the URL host by default. To reach a server with a self-signed or private-CA certificate (a LAN appliance, an internal service), pass `http.TlsOptions` through `http.requestTls` (or `requestWithTls` for explicit timeout / cap). The zero `TlsOptions` verifies, so `request` and the verb shortcuts are unchanged; `http://` ignores the options.

```
use fs;
def opts as http.TlsOptions;
$opts.caCert = fs.readBytes("appliance-ca.pem");   # trust this cert (preferred)
def r as http.Response init http.requestTls("GET", "https://192.168.1.10/", {},
"", $opts);
```

- **caCert** (`bytes` , PEM) trusts a specific certificate in addition to the system roots. Preferred: the server stays authenticated.
- **skipVerify (true)** accepts *any* certificate. It disables authentication and exposes the connection to a man-in-the-middle - use only for a trusted LAN endpoint you cannot give a proper CA. Mirrors `net.TLSOptions` .

Errors

A malformed response, a body that is not valid UTF-8, or a network failure raises a positioned error (a `throw` n `Error` for a malformed response, kind `"http"` ; a `read timed out` error on an idle-timeout); wrap a request in `try` / `catch` to handle a down or slow server. A non-2xx **status is not an error** - a 404 or 500 comes back as a

normal `Response` with that `status` , for the caller to branch on.

Out of scope

- **Redirects.** The one-shot verbs return a 3xx as-is; `http.send` follows them (with a hop limit). No auto-follow on the persistent `exchange` path (it can cross origins).
- **Connection reuse is per-origin and single-request.** `connect` / `exchange` keep one socket alive to one origin; there is no cross-origin connection pool and no request pipelining.
- **Cookie jar is session-scoped.** `Session` keeps a `name -> value` jar with no domain / path scoping or expiry (a full RFC 6265 store is a follow-up); the one-shot verbs keep no cookies. No automatic decompression, no multipart builder - set the headers and body you need directly.
- **Binary bodies** use the byte verbs (`getBytes` / `requestBytes`); the text path decodes UTF-8.

See also

- net.md - the transport (and its TLS) `http` builds on.
- json.md - encode / decode JSON request and response bodies.
- modules/index.md - the module catalog and import rules.

ical - iCalendar (RFC 5545) build and parse

Import with `import "ical.j" as ical;` . Build a calendar of events and encode it to **iCalendar** text (a VCALENDAR of VEVENT s), and parse that text back into a `Calendar` . Pure Jennifer over `strings` / `lists` + `time` - no Go engine, so it runs on **both** binaries.

```
import "ical.j" as ical;
use time;

def ev as ical.Event init ical.event(
    "launch@team",
    time.fromIso("2024-06-20T14:00:00Z"),
    time.fromIso("2024-06-20T15:30:00Z"),
    "Product launch");
def cal as ical.Calendar init ical.add(ical.calendar(), $ev);
def text as string init ical.encode($cal);   # BEGIN:VCALENDAR ... END:VCALENDAR
```

Runnable: `examples/modules/ical_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/ical_demo.j).

Types

Both structs have public fields (read them directly - `$cal.events` , `$ev.summary`); the builder functions are the conventional way to construct them.

```
def struct ical.Calendar { prodid as string, events as list of Event, todos as
list of Todo };
def struct ical.Event {
    uid as string, stamp as time.Time,    # UID / DTSTAMP
    start as time.Time, end as time.Time, # DTSTART / DTEND
    summary as string, description as string, location as string,
    allDay as bool,                       # VALUE=DATE all-day event
    tzid as string,                       # TZID ("" = UTC)
    rrule as string,                      # RRULE ("" = non-recurring)
    rdates as list of time.Time,          # RDATE
    exdates as list of time.Time,         # EXDATE
    organizer as string,                  # ORGANIZER
    attendees as list of Attendee,        # ATTENDEE
    alarms as list of Alarm               # VALARM
};
def struct ical.Attendee { address as string, cn as string, role as string };
def struct ical.Alarm { action as string, trigger as string, description as
string };
def struct ical.Todo {
    uid as string, stamp as time.Time, summary as string,
    due as time.Time, hasDue as bool, description as string, status as string
};
```

Building

Call	Returns	
ical.calendar()	Calendar	an empty calendar with the default PRODID
ical.calendarWith(prodid)	Calendar	an empty calendar with a custom PRODID

ical.event(uid, start, end, summary)	Event	an event; DTSTAMP defaults to start
ical.describe(ev, description) / locate(ev, location)	Event	set the description / location
ical.withAllDay(ev, bool) / withZone(ev, tzid)	Event	mark all-day / set a TZID zone
ical.recur(ev, rrule)	Event	set the RRULE (build one with ical.rule(freq, interval, count))
ical.addRdate(ev, t) / addExdate(ev, t)	Event	add an extra / excluded instant
ical.withOrganizer(ev, addr) / addAttendee(ev, ical.attendee(addr, cn, role))	Event	set organizer / add an attendee
ical.addAlarm(ev, ical.alarm(action, trigger, description))	Event	add a VALARM
ical.occurrences(ev, max)	list of time.Time	expand the recurrence into up to max instants
ical.todo(uid, stamp, summary) + withDue / withStatus / describeTodo	Todo	build a to-do
ical.add(cal, ev) / addTodo(cal, td)	Calendar	a copy with the event / to-do appended

The builders are **value-semantic** - `describe` / `locate` / `add` return a fresh copy and never mutate their argument, so you thread them:

```
def ev as ical.Event init ical.event("id", $start, $end, "Meeting");
$ev = ical.describe($ev, "agenda...");
$ev = ical.locate($ev, "Room 5");
def cal as ical.Calendar init ical.add(ical.calendar(), $ev);
```

Encoding and parsing

Call	Returns	
ical.encode(cal)	string	the calendar as RFC 5545 text (CRLF-terminated)
ical.parse(text)	Calendar	parse iCalendar text back into a calendar

`parse(encode(cal))` round-trips the data. `encode` writes CRLF line endings, escapes text values, folds long lines, and emits `DESCRIPTION` / `LOCATION` only when set. `parse` unfolds folded lines, reads the relevant property parameters (`VALUE=DATE` , `TZID` , `CN` / `ROLE`), reads `VEVENT` / `VTODO` (and nested `VALARM`) components, skips a `VTIMEZONE` , unescapes text, skips a `VEVENT` with no `DTSTART` , and defaults a missing `DTEND` to the start.

Dates and times

`DTSTAMP` / `DTSTART` / `DTEND` go through `time` . `encode` writes each as a UTC `DATE-TIME` (`20240615T130000Z`), normalising a non-UTC `time.Time` to UTC first, so the output is always a correct `Z` value. `parse` accepts the UTC `...Z` form, a floating `DATE-TIME` (no `Z` , read as UTC), and a bare `DATE` (`20240615`).

Text escaping and folding

Text values (`SUMMARY` / `DESCRIPTION` / `LOCATION` / `UID` / `PRODID`) use RFC 5545 escaping: a backslash, `;` , `,` , and any newline become `\\` , `\;` , `\,` , and `\n` . Content lines longer than 75 characters are folded onto continuation lines (a CRLF followed by a space), and `parse` rejoins them - so a long description survives the round-trip intact.

Recurrence

An event carries a raw `RRULE` string plus `RDATE` / `EXDATE` instant lists. `ical.rule(freq, interval, count)` builds a simple rule (`"FREQ=WEEKLY;COUNT=10"`); `ical.occurrences(ev, max)` expands it into

concrete instants - the `RRULE` series from `DTSTART` (honouring `FREQ` / `INTERVAL` / `COUNT` / `UNTIL` , with `MONTHLY` / `YEARLY` day-clamping computed from `DTSTART`), plus `RDATE` s, minus `EXDATE` s. The `BY*` parts (`BYDAY` , `BYMONTH` , ...) are round-tripped in the string but not expanded (the base cadence is still produced).

Scope

- **Fixed-offset time zones.** A `TZID` event stores its wall clock as a floating value paired with the zone name and round-trips exactly, but the instant is not converted through the named IANA zone (the `time` library ships fixed-offset zones only). A `VTIMEZONE` is **parsed and skipped** , not interpreted; add one yourself for a strict consumer. All-day (`VALUE=DATE`) events are supported.
- **VEVENT + VTODO.** `VJOURNAL` / `VFREEBUSY` are not modelled. `VALARM` s are read into an event's `alarms` ; other nested components are skipped.
- **Fold width in characters.** Long lines fold on rune boundaries at 75 characters (never splitting a multi-byte character), rather than strictly on 75 octets - valid output that every reader unfolds.

See also

- time.md - the instant / duration types the dates use.
- strings.md - the text surface the codec is built on.
- modules/index.md - the module catalog and import rules.

idna - internationalized domain names

Import with `import "idna.j" as idna;` . Converts an internationalized domain name between its Unicode form and its ASCII-compatible (`xn--`) encoding, over a **Punycode** (RFC 3492) core - so `münchen.de` goes on the wire as `xn--mnchen-3ya.de` (DNS, SMTP envelopes, and URL hosts are ASCII-only). Pure Jennifer over `strings` , `convert` , and `encoding` ; no networking, TinyGo-clean.

```
use io;
import "idna.j" as idna;

io.printf("%s\n", idna.toAscii("münchen.de"));            # xn--mnchen-3ya.de
io.printf("%s\n", idna.toUnicode("xn--mnchen-3ya.de"));   # münchen.de
```

Runnable: `examples/modules/idna_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/idna_demo.j).

Surface

Call	Returns	Notes
idna.toAscii(domain)	string	Domain to its ASCII form; a Unicode label becomes xn--..., an ASCII label is lowercased.
idna.toUnicode(domain)	string	The inverse; an xn-- label is decoded, others pass through.
idna.isAscii(domain)	bool	Whether the domain is already all-ASCII (needs no conversion).

Both conversions work **label by label** (splitting on `.`), so a mixed domain like `sub.münchen.example` converts only the label that needs it. `toAscii` lowercases (IDNA case-folding); `toAscii(toUnicode(x))` round-trips a domain.

What it is (and isn't)

The `xn--` transformation is **Punycode** (RFC 3492): a bootstring encoding that packs the non-ASCII code points of a label into an ASCII string. This module is that transformation plus lowercasing - enough for the common cases (European accents, most scripts) - **not** full IDNA2008, which layers nameprep / mapping / validation tables on top. It does no length checks and no bidi / script-mixing validation.

The bootstring arithmetic works on rune **code-point integers** , which the `convert` library provides via `convert.toCodepoint(char)` / `convert.fromCodepoint(n)` (added for this module, useful for any Unicode algorithm).

Used by the mail suite

The mail clients call `idna.toAscii` on the connection host and on the domain part of each SMTP envelope address, so an internationalized recipient (`user@münchen.de`) is delivered correctly instead of throwing. A non-ASCII **local part** (before the `@`) still errors - it needs SMTPUTF8 (RFC 6531), which is a later step. Reusable beyond mail: URL hosts, DNS tooling, anywhere an IDN meets an ASCII-only protocol.

See also

- convert.md - `toCodepoint` / `fromCodepoint` , the rune / code-point pair the bootstring arithmetic uses.
- smtp.md / pop.md / imap.md - the mail clients that IDNA-encode their host and envelope domains.
- modules/index.md - the module catalog and import rules.

imap - receive mail (IMAP client)

Import with `import "imap.j" as imap;` . An **IMAP4rev1** receive client (RFC 3501): tagged commands and untagged `*` responses over the `net` system library, with plaintext / implicit-TLS / STARTTLS transport and auth by `LOGIN` , XOAUTH2, CRAM-MD5, or SCRAM-SHA-1 / SCRAM-SHA-256. A practical subset - not the full protocol, but covering both **reading and basic folder management** : select a folder, search it with criteria, fetch whole messages or named headers, set/clear flags, copy or atomically **move** messages (RFC 6851), create a folder, and delete (mark `\Deleted` + expunge). It is not read-only. Retrieved messages come back as strings for the `mime` module to parse. Because it uses `net` , this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "imap.j" as imap;
import "mime.j" as mime;

def opts as imap.Options init imap.Options{host: "mail.example.com", port: 993,
    security: "tls", user: "me", pass: "secret"};
for (def raw in imap.fetchAll($opts, "INBOX")) {
    def msg as mime.Part init mime.parse($raw);
    io.printf("subject: %s\n", mime.headerValue($msg, "Subject"));
}
```

Runnable: `examples/modules/imap_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/imap_demo.j).

Surface

A session is stateful: `connect` , `selectFolder` , `search` / `fetch` / `fetchHeaders` , optional `addFlags` / `expunge` , `logout` . `fetchAll` wraps the common "read every message in a folder" case. Messages are addressed by their stable **UID** throughout (see UIDs below) - `search` returns UIDs, and every message verb takes one.

Call / type	Notes
imap.Options	host, port, security, user, pass, auth.
imap.Session	A live session over one connection (from connect).
imap.connect(opts)	Open a session: greeting, optional STARTTLS, LOGIN / SASL.
imap.folders(session, pattern)	LIST the folders matching pattern ("*" = all) -> list of imap.Folder (name, delimiter, flags).
imap.status(session, folder)	STATUS counts for a folder without selecting it -> imap.Status (messages, recent, unseen, uidnext, uidvalidity).
imap.selectFolder(session, name)	SELECT a folder (e.g. "INBOX"); returns its message count.
imap.Criteria	A search filter (all fields optional; zero = match all). See Searching.
imap.criteria()	An empty Criteria (matches all). Set fields, then pass to search.
imap.search(session, criteria)	The UIDs matching criteria (list of int, via UID SEARCH). imap.criteria() = every message.
imap.fetch(session, uid)	UID FETCH uid BODY.PEEK[] - the message as a raw string, for mime.parse.
imap.fetchMessage(session, uid)	Fetch the message and parse it into a mime.Part tree (import mime too, then use mime.attachments / mime.textBodies).

imap.fetchHeaders(session, uid, flds)	UID FETCH uid BODY.PEEK[HEADER.FIELDS (flds)] - only the named headers (e.g. "SUBJECT DATE"), cheaper than the whole body.
imap.fetchPartial(session, uid, offset, length)	UID FETCH uid BODY.PEEK[]<offset.length> - a byte range of the body, to pull a large message in bounded chunks.
imap.flags(session, uid)	UID FETCH uid (FLAGS) - the flags set on the message as a space-separated string (confirm a STORE persisted).
imap.addFlags(session, uid, flags)	UID STORE uid +FLAGS.SILENT (flags) - add keywords / flags, e.g. "$cl_1" (Thunderbird tag colour) or "\Deleted". A server that disallows a keyword answers OK but drops it - verify with flags.
imap.removeFlags(session, uid, flags)	UID STORE uid -FLAGS.SILENT (flags) - clear keywords / flags (inverse of addFlags); removing an unset flag is a no-op.
imap.createFolder(session, name)	CREATE name - make a folder; errors if it already exists, so try/catch for a create-if-missing.
imap.copy(session, uid, folder)	UID COPY uid folder - copy the message into another (existing) folder. A "move" is copy + addFlags(..., "\Deleted") + expunge, or the atomic move.
imap.move(session, uid, folder)	UID MOVE uid folder (RFC 6851) - copy + delete + expunge in one atomic step (no manual \Deleted + EXPUNGE).
imap.append(session, folder, msg)	APPEND - upload a full RFC 5322 message into folder (e.g. save to Sent).
imap.appendWith(session, folder, flags, msg)	APPEND with initial flags, e.g. "\Seen" for Sent or "\Draft" for a draft.
imap.expunge(session)	EXPUNGE - permanently remove all \Deleted messages in the selected folder.
imap.logout(session)	LOGOUT and close.
imap.fetchAll(opts, folder)	Connect, select, retrieve every message, log out; list of string.
imap.Notification	One server push during IDLE: kind ("exists" / "expunge" / "recent" / ""), number.
imap.supportsIdle(session)	CAPABILITY gate - true when the server advertises IDLE (RFC 2177).
imap.idle(session)	Enter IDLE (IDLE -> + idling); the server now pushes mailbox changes.
imap.receiveNotification(session)	Block for the next push -> imap.Notification (empty sentinel when IDLE ends).
imap.pollNotification(session, timeoutMs)	Like receiveNotification, but wait at most timeoutMs ms (net.setDeadline), then the empty sentinel.
imap.done(session)	Leave IDLE (DONE + tagged completion), back to command mode.

`Options.security` is a `transport.Security` : `.None` (143), `.Tls` (implicit TLS on connect, 993), or `.Starttls` (needs `import "transport.j" as transport;`). `fetch` uses `BODY.PEEK[]` , so retrieving does **not** set the `\Seen` flag. An internationalized (IDN) host is IDNA-encoded to its `xn--` form automatically (via `idna`).

Authentication

`connect` authenticates according to `Options.auth` ; the non-LOGIN mechanisms go through the `sasl` module:

auth	Mechanism
"" (default)	Plain LOGIN (username + pass).

"xoauth2"	AUTHENTICATE XOAUTH2 - pass the OAuth2 access token as pass (Gmail / Microsoft 365).
"cram"	AUTHENTICATE CRAM-MD5 - challenge-response, no cleartext password.
"scram-sha-1" / "scram-sha-256"	AUTHENTICATE SCRAM-* - salted challenge-response (non-initial-response form); the server signature is verified.
"auto"	Probe CAPABILITY and pick the strongest mechanism offered, falling back to LOGIN.

Use `security: "tls"` / `"starttls"` so `LOGIN` / XOAUTH2 credentials never go over the wire in the clear.

Browsing folders

`folders(session, pattern)` runs `LIST` and returns the matching folders as `imap.Folder` values (`name` , `delimiter` , `flags`); `status(session, folder)` returns a folder's counts (`messages` / `unseen` / `recent` / `uidnext` / `uidvalidity`) **without** selecting it - ideal for a folder tree with unread badges. (IMAP calls these "mailboxes"; the module uses the everyday term *folder* .)

```
def s as imap.Session init imap.connect($opts);
for (def f in imap.folders($s, "*")) {          # every folder, at any depth
    def st as imap.Status init imap.status($s, $f.name);
    io.printf("%s (%d unread)\n", $f.name, $st.unseen);
}
```

Use `"%"` instead of `"*"` for the top level only, or a prefix like `Archive/*` to scope to a subtree. `flags` carries IMAP attributes such as `\HasChildren` (build a tree from the `delimiter`) and `\Noselect` (a container you cannot `selectFolder`). Non-ASCII folder names arrive in IMAP's modified-UTF-7 form.

Searching

`search(session, criteria)` returns the **UIDs** of the messages in the selected folder that match a `Criteria` (via `UID SEARCH`), so you fetch only the mail you want instead of pulling the whole folder. Build one from `imap.criteria()` (which matches everything) and set the fields you need - a zero-value `Criteria` is the old `SEARCH ALL` .

The fields split by **where they run** :

- **Server-side** - mapped straight to one IMAP `SEARCH` (a single round-trip, no message bodies downloaded), and all ANDed together:
  - `subject` / `from` / `to` / `text` - case-insensitive **substring** match (`text` covers headers + body).
  - `since` / `before` - an inclusive-since / exclusive-before range as standard `time.Time` values (a zero-value time, the default, ignores the bound). Set them straight from `time` - `$c.since = time.now()` , or a shifted time for "the last week". IMAP `SEARCH` filters by calendar **day** , so a bound at midnight is a pure server-side search; a bound that carries a **time-of-day** is transparently refined to the exact instant on the client (against each candidate's `INTERNALDATE` , the arrival clock `SEARCH` uses), so a sub-day range like "since 14:30 today" just works - no extra call.
  - `seen` / `unseen` / `flagged` / `answered` - flag state.
  - `largerThan` / `smallerThan` - size in bytes.
- **Client-side** - applied only to the messages the server-side search returns, by fetching just their headers or structure (never full bodies):
  - `subjectRegex` / `fromRegex` - an RE2 pattern matched against the decoded `Subject` / `From` header (what IMAP's substring search can't do).
  - `hasAttachments` - keep only messages whose `BODYSTRUCTURE` shows an attachment. This is a **heuristic** (it looks for an `attachment` content-disposition and downloads no body); an unusual message may fool it.

```
import "imap.j" as imap;
import "mime.j" as mime;
use time;

def s as imap.Session init imap.connect($opts);
imap.selectFolder($s, "INBOX");

# Unseen invoices from billing since the start of the year, with an attachment.
def c as imap.Criteria init imap.criteria();
$c.from = "billing@";                                    # server-side
substring
$c.since = time.fromIso("2026-01-01T00:00:00Z");         # server-side date
(a time.Time)
$c.unseen = true;                                        # server-side flag
$c.subjectRegex = "INV-2026-[0-9]+";                     # client-side regex
on Subject
$c.hasAttachments = true;                                # client-side
structure check

for (def uid in imap.search($s, $c)) {
    def msg as mime.Part init imap.fetchMessage($s, $uid);   # fetch only the
matches
    io.printf("%s\n", mime.headerValue($msg, "Subject"));
}
```

The server-side fields (`from` , `since` , `unseen`) become one SEARCH ; only the handful of messages it returns are then fetched to apply the `subjectRegex` and `hasAttachments` filters, so an inbox of thousands costs one search plus a few header/structure fetches - not a full download.

Search strings are safe to build from data: substrings are sent as quoted, control-checked IMAP strings, dates come from `time.Time` values, and sizes are integers, so no criteria field can inject an IMAP command. An inverted range (`since` set after `before`) is a catchable `Error` (kind `"imap"`) rather than a silent empty result; `since` == `before` is a valid (empty) range.

Not yet covered: non-ASCII search strings over SEARCH (use `subjectRegex` / `fromRegex` meanwhile).

UIDs: stable identifiers

Every message verb addresses by **UID** , never by a sequence number. A sequence number is only valid within the current selection - an EXPUNGE renumbers every message after the removed one, so a number captured earlier can silently point at the wrong message. A UID is stable: it keeps naming the same message across expunges and across sessions (paired with the folder's `UIDVALIDITY` from `status`). So there is one addressing scheme, and it is the correct one:

```
def uids as list of int init imap.search($s, imap.criteria());  # stable UIDs
def body as string init imap.fetch($s, $uids[0]);               # fetch by UID
```

This is the correct basis for "process only what is new since last run": record the UIDs (and `UIDVALIDITY`) you have handled, then next session `search` and skip the ones you have already seen - immune to the renumbering a sequence-number loop would trip over. Sequence numbers appear only where the *server* emits them as data - the count from `selectFolder` , and the numbers in IDLE EXISTS / EXPUNGE pushes - never as an addressing input.

Managing messages

Reading is only half of it - the module also **modifies** the folder. Given a UID (from `search`):

Flag a message (mark read, tag, star). Flags are a space-separated string; system flags start with `\` , keywords

don't. A server that disallows a keyword answers `OK` but silently drops it, so read it back with `flags` if it matters:

```
def uid as int init imap.search($s, imap.criteria())[0];
imap.addFlags($s, $uid, "\\Seen");            # mark read
imap.addFlags($s, $uid, "$cl_1");             # add a keyword/tag (Thunderbird
colour label)
imap.removeFlags($s, $uid, "\\Flagged");      # unstar it
io.printf("now: %s\n", imap.flags($s, $uid)); # e.g. "\Seen $cl_1"
```

Move a message. The RFC 6851 `MOVE` does copy + delete + expunge atomically:

```
imap.createFolder($s, "Archive");        # once; errors if it already exists (try/
catch)
imap.move($s, $uid, "Archive");          # atomic: no manual \Deleted + expunge
```

`MOVE` is widely but not universally supported; a server lacking it answers `BAD` , so fall back to the classic copy + `\Deleted` + `expunge` under a `try` / `catch` :

```
imap.copy($s, $uid, "Archive");           # copy into the target folder
imap.addFlags($s, $uid, "\\Deleted");     # then mark the original deleted
imap.expunge($s);                         # permanently remove every \Deleted
message
```

Large bodies can be pulled in ranges with `fetchPartial(session, uid, offset, length)` , which issues `UID FETCH ... BODY.PEEK[]<offset.length>` so a big message is retrieved in bounded chunks instead of one huge literal.

Save a message into a folder with `append` (e.g. keep a copy in Sent after sending, or store a draft). The message is a full RFC 5322 string - headers, a blank line, then the body - built however you like (the `mime` module helps). `appendWith` sets initial flags:

```
imap.append($s, "Sent", $rawMessage);              # arrives unflagged
imap.appendWith($s, "Drafts", "\\Draft", $draft);  # marked \Draft
```

Tagged responses and literals

Two IMAP mechanics the client handles for you:

- **Tags.** Each command carries a tag and completes with a tagged `OK` / `NO` / `BAD` line; a `NO` / `BAD` throws a catchable `Error` (kind `"imap"`). The client uses one fixed tag, which is safe here because it is synchronous (one command in flight at a time).
- **Literals.** A `FETCH` body arrives as a `{N}` literal - a byte count followed by exactly `N` bytes - which the client reads by count rather than by line, so a message body containing blank lines or its own `)` is returned intact.

Certificate verification for `"tls"` / `"starttls"` is the `net` default.

Testing

The pure protocol logic - tag detection, literal-length and literal extraction, `EXISTS` / `SEARCH` parsing, `LOGIN` argument quoting, and tagged `OK` / `NO` handling - is unit-tested in the overlay. The networked session (tagged responses **and** literal reading) is covered end to end by an in-process fake IMAP server in the Go test suite (`TestImapReceive`), so it runs in CI without an external server.

IDLE (server push)

Instead of re-polling `STATUS` on a timer, `IDLE` (RFC 2177) lets the server **push** mailbox changes as they happen - new mail, an expunge, a recent-count change. The idiom is a cooperative read loop: **idle -> receive / poll -> done** . There are no callbacks; you read pushes with a blocking `receiveNotification` (or a timeout-bounded `pollNotification`) and, when you want to stop, break out with `done` . It is the same M23.1 streaming / server-push shape the other read loops share, so wrapping the loop in a `spawn` runs push handling beside the rest of your program.

Each push arrives as an `imap.Notification` : `kind` is `"exists"` (the mailbox now holds this many messages - new mail), `"expunge"` (the message at this sequence number was removed), or `"recent"` (recent-count change); `number` is the count / sequence number it carried. A `kind` of `""` is the **idle-gap sentinel** - `pollNotification` timed out, or IDLE ended - so it never blocks a poll loop.

```
import "imap.j" as imap;

def s as imap.Session init imap.connect($opts);
imap.selectFolder($s, "INBOX");
if (not imap.supportsIdle($s)) {
    io.printf("server has no IDLE - fall back to a STATUS poll\n");
    exit 1;
}

imap.idle($s);                                  # enter IDLE; server now pushes
def n as imap.Notification init imap.receiveNotification($s);   # blocks for the
next push
if ($n.kind == "exists") {
    io.printf("new mail: the mailbox now has %d messages\n", $n.number);
}
imap.done($s);                                  # leave IDLE, back to command mode
# Ordinary commands work again. A push carries a sequence number, not a UID, so
# resolve the new mail's stable UID with a search before fetching it.
def uids as list of int init imap.search($s, imap.criteria());
def latest as string init imap.fetch($s, $uids[len($uids) - 1]);
```

`pollNotification($s, timeoutMs)` is the non-blocking variant - it returns the next push if one arrives within `timeoutMs` , otherwise the empty sentinel, so a loop can do other work between checks:

```
imap.idle($s);
while (running()) {
    def n as imap.Notification init imap.pollNotification($s, 1000);   # wait up
to 1 s
    if ($n.kind == "exists") {
        handleNewMail($s, $n.number);
    }
    doOtherWork();
}
imap.done($s);
```

A server **drops an idle session after about 29 minutes** (RFC 2177 advises re-issuing well before that), so a long-lived client must periodically `done` and `idle` again - break IDLE, then re-enter it - rather than assume the same IDLE stays live forever. Always pair an `idle` with a `done` before running any other command: while idling, the connection is dedicated to the push stream.

Out of scope

A practical subset of IMAP4rev1, not the whole protocol. What it does **not** cover:

- **Commands.** No `RENAME` / `DELETE` of *folders* (only `CREATE` / `LIST` / `STATUS`). `IDLE` is supported (see IDLE) for the `EXISTS` / `EXPUNGE` / `RECENT` pushes (whose numbers are sequence numbers - the one place they surface). Message operations address by **UID** ; `SEARCH` sends ASCII-only criteria (no `CHARSET UTF-8`).
- **Auth.** The supported mechanisms are in Authentication; *not* covered are GSSAPI / Kerberos, NTLM, client-certificate auth, and OAuth2 token **acquisition** / refresh - obtain the access token yourself (e.g. via the `oauth` module) and pass it as `pass` with `auth: "xoauth2"` .

- **Extensions.** Only the IMAP4rev1 core - no `SORT` / `THREAD` , `CONDSTORE` / `QRESYNC` , `COMPRESS` , `METADATA` , quota, or ACL support.

(Message literals are read by byte count, so a raw 8-bit / multi-byte body **is** returned byte-exact - see Tagged responses and literals.)

Timeouts and limits

Reads carry a 30 s idle timeout (a deadline re-armed before each read), so a hung server fails with a catchable error instead of blocking the caller forever. The initial connect (and a STARTTLS handshake) is bounded by its own connection-establishment timeout, so a slow or unreachable server fails the dial. A single accumulated response is capped at **64 MiB** : a literal's `{N}` byte count is attacker-declarable, and a server can also stream untagged lines that never reach the tagged completion, so either fails with a catchable error rather than an unbounded allocation.

See also

- mime.md - parse a fetched message (`imap.fetchMessage` / `mime.parse`) and pull out attachments (`mime.attachments` / `mime.data`) and text bodies (`mime.textBodies`).
- pop.md - the simpler POP3 receive client; smtp.md - send.
- net.md - the transport `imap` builds on.
- modules/index.md - the module catalog and import rules.

influxdb - InfluxDB time-series client

Import with `import "influxdb.j" as influxdb;` . Write measurements as line-protocol points and query them back as parsed results, against **both** major API generations: **1.x** (`/write?db=...` + InfluxQL) and **2.x / 3.x** (`/api/v2/write?org=...&bucket=...` + Flux). Built on the `http` module, so it needs the default `jennifer` binary. A failed request throws `Error{kind: "influxdb"}` (a 2.x client's token is redacted from it).

```
import "influxdb.j" as influxdb;

# 1.x: database + optional Basic auth, InfluxQL.
def db as influxdb.Client init influxdb.client("http://localhost:8086",
"metrics");
def p as influxdb.Point init influxdb.field(
    influxdb.tag(influxdb.point("cpu"), "host", "server01"), "value", 0.64);
influxdb.write($db, [$p]);
def r as influxdb.Result init influxdb.query($db, "SELECT last(\"value\") FROM
cpu");

# 2.x / 3.x: org + bucket + token, Flux.
def db2 as influxdb.Client init influxdb.client2("http://localhost:8086",
"myorg", "mybucket", "mytoken");
influxdb.write($db2, [$p]);   # same call - dispatches on the client version
def csv as string init influxdb.queryFlux($db2, "from(bucket:\"mybucket\") |>
range(start:-1h)");
```

Runnable: `examples/modules/influxdb_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/influxdb_demo.j).

Client

One `Client` struct serves both generations, with a `version` discriminator selecting which fields apply (stance 1: one module, one selectable backend). A 1.x client carries `db` and optional Basic-auth `user` / `password` ; a 2.x client carries `org` / `bucket` and a `token` . Unused fields stay `""` .

```
export def enum influxdb.Version { V1, V2 };

def struct influxdb.Client {
    url as string,          # base URL, e.g. "http://localhost:8086"
    version as Version,     # V1 (InfluxDB 1.x) or V2 (InfluxDB 2.x / 3.x)
    db as string,           # 1.x database name ("" for a 2.x client)
    user as string,         # 1.x Basic-auth username ("" for no auth)
    password as string,     # 1.x Basic-auth password
    org as string,          # 2.x organization ("" for a 1.x client)
    bucket as string,       # 2.x bucket ("" for a 1.x client)
    token as string         # 2.x API token ("" for a 1.x client)
};
```

Call	Returns	
influxdb.client(url, db)	Client	1.x: connect to a database, no authentication
influxdb.clientWith(url, db, user, password)	Client	1.x: with HTTP Basic-auth credentials
influxdb.client2(url, org, bucket, token)	Client	2.x / 3.x: token auth, addressed by org + bucket

Writing points

A `Point` is built with value-semantic builders - each returns a fresh `Point` , so they chain. Field types are carried as pre-rendered line-protocol fragments, so one point can mix float, integer, string, and boolean fields (Jennifer maps are homogeneous, so a single typed map could not).

Call	Returns	
influxdb.point(measurement)	Point	start a point (no tags/fields yet)
influxdb.tag(p, key, value)	Point	add an indexed string tag
influxdb.field(p, key, value)	Point	add a float field
influxdb.intField(p, key, value)	Point	add an int field (line-protocol i suffix)
influxdb.stringField(p, key, value)	Point	add a string field (quoted, escaped)
influxdb.boolField(p, key, value)	Point	add a bool field
influxdb.at(p, unixNanos)	Point	set an explicit timestamp (nanoseconds)
influxdb.atTime(p, t)	Point	set the timestamp from a time.Time
influxdb.line(p)	string	render one line-protocol line (throws if no fields)
influxdb.write(c, points)		write a list of Point to the database

Line-protocol escaping is automatic: measurement names escape space and comma; tag keys/values and field keys escape space, comma, and = ; string field values are double-quoted with `"` and `\` escaped. A point with **no fields** is invalid line protocol, so `line` / `write` throw for one. The line protocol is **identical across API generations** , so `line` is version-agnostic.

`write` dispatches on the client's `version` : a **1.x** client posts to `/write?db=...&precision=ns` with optional Basic auth; a **2.x** client posts to `/api/v2/write?org=...&bucket=...&precision=ns` with an `Authorization: Token

```
def p as influxdb.Point init influxdb.point("cpu");
$p = influxdb.tag($p, "host", "server01");
$p = influxdb.field($p, "value", 0.64);
$p = influxdb.intField($p, "cores", 8);
influxdb.line($p);   # cpu,host=server01 value=0.64,cores=8i
```

Querying

```
def struct influxdb.Series {
    name as string,                      # measurement name
    tags as map of string to string,     # GROUP BY tag set ({} if none)
    columns as list of string,           # column names, e.g. ["time", "value"]
    values as list of list of string     # rows, one stringified cell per column
};
def struct influxdb.Result {
    series as list of Series             # flattened across every statement
};
```

`influxdb.query(client, influxql)` runs an InfluxQL statement against `/query` and parses the tabular JSON into `Series` (the same read-a-parsed-result shape as `prometheus` 's retrieval half). Every **cell is stringified** - `time` comes back as its RFC 3339 string, numbers via their shortest form, booleans as `"true"` / `"false"` , and JSON `null` as `""` - so a homogeneous `list of list of string` can hold a row of otherwise mixed-type columns. Convert a cell you know is numeric with `convert.toFloat` . A per-statement `error` in the response throws `Error{kind: "influxdb"}` .

```
def r as influxdb.Result init influxdb.query($db, "SELECT value FROM cpu");
for (def s in $r.series) {
    for (def row in $s.values) {
        # row[0] = time (string), row[1] = value (stringified number)
    }
}
```

Flux (2.x / 3.x)

`influxdb.queryFlux(client2, flux)` runs a Flux script against a 2.x client's organization (`POST /api/v2/query?org=...` under `application/vnd.flux`). InfluxDB answers a Flux query with **annotated CSV** (RFC 4180 with `#datatype` / `#group` / `#default` header rows), not JSON, so `queryFlux` returns the response body verbatim as a `string` ; parse it with the `csv` module. (The JSON-parsing `query` / `Series` / `Result` shape is the 1.x InfluxQL path.)

```
def db2 as influxdb.Client init influxdb.client2("http://localhost:8086",
"myorg", "mybucket", "mytoken");
def csv as string init influxdb.queryFlux($db2,
    "from(bucket:\"mybucket\") |> range(start:-1h) |> filter(fn:(r) => r._
measurement == \"cpu\")");
```

Scope

- **Both API generations.** 1.x line protocol + InfluxQL (`client` / `clientWith` , `query`) and 2.x / 3.x org/bucket writes + Flux (`client2` , `queryFlux`), sharing the identical line protocol and the same value-semantic `Point` builders. `write` dispatches on the client's `version` .
- **Nanosecond write precision** (`precision=ns`); a point's timestamp is an integer nanosecond value (or a `time.Time` via `atTime`).
- **1.x query cells are stringified.** The `Result` keeps rows homogeneous (`list of list of string`) rather than exposing a typed cell union; you convert numeric columns yourself.
- **Flux returns raw annotated CSV.** `queryFlux` returns the response body as a `string` for the `csv` module to parse, rather than a bespoke Flux-CSV decoder.
- **Auth: 1.x Basic auth, 2.x token.** A 2.x token rides in the `Authorization` header and is redacted from any raised error. TLS client certs are not wired; use a reverse proxy for those.

See also

- http.md - the HTTP client this module builds on.
- prometheus.md - the pull-based metrics sibling; its retrieval half shares this parsed-result shape.
- modules/index.md - the module catalog and import rules.

ipnet - IP addresses and CIDR networks

Import with `import "ipnet.j" as ipnet;` . Parse and reason about **IPv4 and IPv6** addresses and CIDR blocks: canonical formatting, membership tests, subnet math (netmask, broadcast, split, aggregate, host iteration), and address classification (private / loopback / global / ...). Addresses are held as raw `bytes` (4 for IPv4, 16 for IPv6); the math is bitwise. Pure Jennifer over `strings` + `convert` ; runs on **both** binaries.

```
import "ipnet.j" as ipnet;

def net as ipnet.Network init ipnet.parse("192.168.1.0/24");
def ip as ipnet.Address init ipnet.parseAddress("192.168.1.42");
def inside as bool init ipnet.contains($net, $ip);   # true
```

Runnable: `examples/modules/ipnet_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/ipnet_demo.j).

Types

Both structs have public fields (read them directly), and the builder functions are the conventional way to construct them.

```
def struct ipnet.Address { version as int, octets as bytes };   # version 4 or 6
def struct ipnet.Network { addr as Address, prefix as int };
```

- `Address.version` is 4 or 6 ; `Address.octets` is the raw address bytes (4 or 16), in network byte order.
- `Network.addr` is the base address with host bits zeroed; `Network.prefix` is the prefix length (`0..32` for IPv4, `0..128` for IPv6).

Addresses

Call	Returns	
ipnet.parseAddress(s)	Address	parse an IPv4 dotted-quad or IPv6 address
ipnet.toString(addr)	string	canonical text (RFC 5952 for IPv6)
ipnet.version(addr)	int	4 or 6
ipnet.equal(a, b)	bool	same version and bytes
ipnet.unmap(addr)	Address	fold a v4-mapped IPv6 address down to v4
ipnet.next(addr)	Address	the next address (+1); throws at the last
ipnet.prev(addr)	Address	the previous address (-1); throws at the first
ipnet.compare(a, b)	int	-1 / 0 / 1 (v4 orders before v6)

`parseAddress` accepts IPv6 with `::` zero-compression and a trailing embedded IPv4 (`::ffff:192.168.1.1`). `toString` renders IPv6 canonically per RFC 5952: lowercase, no leading zeros, and the longest run of two-or-more zero groups compressed to `::` (leftmost on a tie). Because `equal` compares bytes, two different spellings of the same address compare equal. `next` / `prev` / `compare` give a total order, so you can walk or sort addresses.

CIDR networks

Call	Returns	
ipnet.parse(cidr)	Network	parse address/prefix (host bits zeroed)
ipnet.networkString(net)	string	render as address/prefix
ipnet.contains(net, addr)	bool	is the address in the network?
ipnet.netmask(net)	Address	the netmask (e.g. 255.255.255.0)
ipnet.broadcast(net)	Address	the last address (IPv4 broadcast)

`parse` zeroes the host bits, so `ipnet.parse("192.168.1.42/24")` has base `192.168.1.0` . `contains` returns `false` for a version mismatch (an IPv4 address is never inside an IPv6 network). `broadcast` sets every host bit: for IPv4 that is the broadcast address, for IPv6 the last address in the block.

```
def allowed as list of ipnet.Network init [ipnet.parse("10.0.0.0/8"), ipnet.
parse("192.168.0.0/16")];
def client as ipnet.Address init ipnet.parseAddress("10.4.5.6");
def ok as bool init false;
for (def net in $allowed) {
    if (ipnet.contains($net, $client)) { $ok = true; }
}
```

Subnet math

Call	Returns	
ipnet.hostCount(net)	int	total addresses in the block (2 ^ host-bits)
ipnet.firstUsable(net)	Address	first usable host (see below)
ipnet.lastUsable(net)	Address	last usable host (see below)
ipnet.hosts(net)	list of Address	every usable host, ascending (capped)
ipnet.split(net, newPrefix)	list of Network	divide into longer-prefix subnets
ipnet.aggregate(nets)	list of Network	collapse to the minimal covering set
ipnet.overlaps(a, b)	bool	do two networks intersect?
ipnet.subnetOf(child, parent)	bool	is child wholly inside parent?

- **hostCount** counts every address in the block. IPv4 always fits an `int` ; an IPv6 prefix must be >= `66` (a shorter one overflows `int` and throws).
- **firstUsable / lastUsable** apply the usual host conventions. IPv4 excludes the network and broadcast addresses, except a `/31` (RFC 3021 point-to-point, both usable) and a `/32` (single host). IPv6 has no broadcast, so the last address is usable; the first is the base + `1` (skipping the subnet-router anycast), except `/127` and `/128` .
- **hosts** materializes the usable range - capped at **65536** addresses; a larger block throws (walk it with `next` instead), so this cannot exhaust memory.
- **split** divides a network into the equal subnets of a longer prefix (a `/24` into four `/26` s), capped at **65536** subnets (`newPrefix` at most 16 bits longer).
- **aggregate** drops contained blocks and folds adjacent sibling pairs into their shorter parent (`192.168.0.0/25` + `192.168.0.128/25` -> `192.168.0.0/24`), cascading until stable. IPv4 and IPv6 entries aggregate independently; the result is sorted ascending.

```
def subnets as list of ipnet.Network init ipnet.split(ipnet.parse("192.168.1.0/
24"), 26);
# -> 192.168.1.0/26, .64/26, .128/26, .192/26

def merged as list of ipnet.Network init ipnet.aggregate([
    ipnet.parse("10.0.0.0/9"), ipnet.parse("10.128.0.0/9")
]);   # -> [10.0.0.0/8]
```

Classification

Call	Returns	
ipnet.scope(addr)	Scope	the address's category (see below)
ipnet.isGlobal(addr)	bool	a normal globally-routable unicast address
ipnet.isPrivate(addr)	bool	RFC 1918 or IPv6 ULA (fc00::/7)

ipnet.isLoopback(addr)	bool	127.0.0.0/8 or ::1
ipnet.isLinkLocal(addr)	bool	169.254.0.0/16 or fe80::/10
ipnet.isMulticast(addr)	bool	224.0.0.0/4 or ff00::/8
ipnet.isUnspecified(addr)	bool	0.0.0.0 or ::

`scope` is a **total, disjoint** classification: every address maps to exactly one variant of the enum

```
def enum ipnet.Scope { Global, Private, Loopback, LinkLocal, Multicast,
Unspecified, Reserved };
```

so the `is*` predicates are thin wrappers over it, and you can `match` the whole set in one place. `Reserved` covers the special-purpose ranges that are neither a normal global address nor one of the named categories (documentation `192.0.2.0/24` / `2001:db8::/32` , shared CGNAT `100.64.0.0/10` , benchmarking, `240.0.0.0/4` , and other reserved blocks). A v4-mapped IPv6 address is folded to its v4 self first, so `::ffff:10.0.0.1` classifies as `Private` .

```
match (ipnet.scope($client)) {
    when Global { web.serve($client); }
    when Private, Loopback, LinkLocal { web.serveInternal($client); }
    else { web.reject($client); }
}
```

The classification follows the well-known special-purpose registries and is **best-effort** : `isGlobal` means "not one of the categories above", not a routing-table check.

Errors

Malformed input throws `Error{kind: "ipnet"}` (a bad octet, too few / many groups, multiple `::` , a bad hex digit, a missing `/prefix` , an out-of-range prefix, an address overflow at `next` / `prev` , or a too-large `hosts` / `hostCount` / `split`) - catch it with `try` / `catch` .

Scope

- **Address and prefix math, not a resolver.** No DNS, no interface enumeration; hostname / interface lookups live in the `net` library.
- **Strict decimal octets.** `parseAddress` rejects a leading-zero IPv4 octet (`192.168.001.1`): many host stacks read a leading-zero octet as octal, so accepting it as decimal would let an allow-list disagree with the kernel (an SSRF / allow-list-bypass vector).
- **IPv4-mapped IPv6 folds to v4.** `parseAddress("::ffff:127.0.0.1")` returns a version-4 `Address` (via `unmap`), so it matches a v4 allow-list; `parse` likewise folds a v4-mapped CIDR (`::ffff:0:0/96` -> `0.0.0.0/0`), translating the prefix. The deprecated IPv4-compatible form (`::a.b.c.d`) is left as v6 (ambiguous with low addresses like `::1`).
- **Sequence-number-free.** Iteration and aggregation work on the address bytes directly; there is no scope-id / zone-id parsing.

See also

- net.md - sockets, TLS, and DNS lookups.
- strings.md - the text surface the parser builds on.
- modules/index.md - the module catalog and import rules.

jsonl - JSON Lines (JSONL / NDJSON)

Import with `import "jsonl.j" as jsonl;` . Read and write **newline-delimited JSON** : one independent JSON value per line. A thin framing layer over `json` - each record is a `json.Value` , so `encode` / `decode` compose `json.encode` / `json.decode` with a `\n` split / join, and the file helpers add `fs` . Pure Jennifer; runs on **both** binaries.

```
import "jsonl.j" as jsonl;
use json;

def rows as list of json.Value init [json.decode("{\"a\":1}"), json.
decode("[2,3]")];
def text as string init jsonl.encode($rows);   # {"a":1}\n[2,3]\n
def back as list of json.Value init jsonl.decode($text);
```

Runnable: `examples/modules/jsonl_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/jsonl_demo.j).

In-memory

Call	Returns	
jsonl.encode(records)	string	one compact JSON value per line, each newline-terminated
jsonl.decode(text)	list of json.Value	one record per non-blank line

`records` is a `list of json.Value` - build them with `json.decode` , `json.map()` / `json.set` , or by re-encoding a struct through `json` . Any top-level JSON type is a valid line (object, array, number, string, `true` / `false` / `null`). `decode` skips blank and whitespace-only lines and trims a trailing `\r` (CRLF input), so `decode(encode(records))` round-trips. An empty list encodes to `""` ; `decode("")` is the empty list.

Whole file

Call	Returns	
jsonl.readFile(path)	list of json.Value	read and decode a whole JSONL file
jsonl.writeFile(path, records)	null	encode and write (replacing existing content)
jsonl.appendFile(path, records)	null	encode and append (file created if missing)

`appendFile` is the common JSONL pattern - adding rows to a growing log or event stream without rewriting the file.

Streaming large files

For JSONL too large to hold in memory, a `Reader` yields one record at a time. The wrapped `fs.File` is a handle - it shares its read position across value copies, so successive `readRecord` calls advance the same stream.

Call	Returns	
jsonl.openReader(path)	Reader	open a file for streaming
jsonl.hasMore(reader)	bool	whether the file has unread bytes (coarse; see below)
jsonl.readRecord(reader)	Record	the next {value, done} (skips blank lines)
jsonl.closeReader(reader)	null	close the reader

`readRecord` returns a `Record {value, done}` : `done` is the reliable end-of-stream signal. Loop until `done`

rather than guarding with `hasMore` - `hasMore` is a coarse `not eof` check that still reports `true` when only trailing blank lines remain (which carry no record), so a `hasMore` -guarded loop would over-run the last record.

```
def r as jsonl.Reader init jsonl.openReader("events.jsonl");
while (true) {
    def rec as jsonl.Record init jsonl.readRecord($r);
    if ($rec.done) {
        break;
    }
    # process $rec.value without loading the whole file
}
jsonl.closeReader($r);
```

Streaming over a caller-owned file handle

When the caller already holds an open `fs.File` (or wants to stream both a reader and a writer), the file-handle surface wraps an open handle instead of a path. `reader` / `writer` take an `fs.File` ; `writeValue` appends one compact JSON value plus a newline; `readValue` returns the next record as a `json.Value` directly.

Call	Returns	
jsonl.reader(file)	Reader	wrap an open read-mode fs.File
jsonl.readValue(reader)	json.Value	the next record (skips blank lines); JSON null at end
jsonl.writer(file)	Writer	wrap an open write/append-mode fs.File
jsonl.writeValue(writer, value)	null	append one value terminated by a newline
jsonl.closeWriter(writer)	null	close the writer

```
use fs;
import "jsonl.j" as jsonl;

def wf as fs.File init fs.open("events.jsonl", "append");
def w as jsonl.Writer init jsonl.writer($wf);
jsonl.writeValue($w, json.decode("{\"event\":\"login\"}"));
jsonl.closeWriter($w);

def rf as fs.File init fs.open("events.jsonl", "read");
def r as jsonl.Reader init jsonl.reader($rf);
while (not fs.eof($rf)) {
    def v as json.Value init jsonl.readValue($r);
    # process $v
}
jsonl.closeReader($r);
```

`readValue` returns a JSON `null` once the stream is exhausted. JSONL written by `writeValue` has exactly one value per line and no trailing blanks, so `not fs.eof(...)` is a reliable loop guard; when you need to tell end-of-stream from a genuine `null` record, use `readRecord` and its explicit `done` flag instead.

Scope

- **Records are json.Value.** JSONL is a framing convention, not a new encoder - the actual JSON work stays in the `json` library, and rebuilding a typed target from a decoded record is the same explicit step it is there (no map-to-struct coercion).
- **\n-separated.** Records are separated by line feed; a trailing `\r` is tolerated on read. `encode` writes `\n` and terminates the last line too.

See also

- json.md - the encoder / decoder and the `json.Value` accessors each record is built from.
- fs.md - the file surface the read / write / stream helpers build on.
- modules/index.md - the module catalog and import rules.

jsonrpc - JSON-RPC 2.0 client and server

Import with `import "jsonrpc.j" as jsonrpc;` . A JSON-RPC 2.0 **client** that calls remote methods over HTTP, and a transport-agnostic **server** `handle` that dispatches an incoming request to your program's methods by name. Built on `json` for the wire format and `http` for the client transport, so it needs the default `jennifer` binary.

`params` and a call's result are `json.Value` s: you build params with the `json` write API and read the result with the `json` accessors. Every client-side failure - a JSON-RPC error reply, a transport error (connection refused / timeout / malformed HTTP), or a malformed reply - surfaces as a single catchable `Error{kind: "jsonrpc"}` , so one `catch` covers them all.

```
import "jsonrpc.j" as jsonrpc;
use json;

def c as jsonrpc.Client init jsonrpc.client("https://api.example.com/rpc");

# positional params: [2, 3]
def args as json.Value init json.list();
$args = json.append($args, "", 2);
$args = json.append($args, "", 3);

def sum as json.Value init jsonrpc.call($c, "add", $args);   # -> 5
```

Runnable: `examples/modules/jsonrpc_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/jsonrpc_demo.j).

Client

A `Client` is an endpoint plus optional headers; every call POSTs a request and reads the reply.

```
def struct jsonrpc.Client { endpoint as string, headers as map of string to
string };
```

Call	Returns	
jsonrpc.client(endpoint)	Client	bind to a JSON-RPC endpoint URL
jsonrpc.clientWith(endpoint, headers)	Client	with extra request headers (e.g. Authorization)
jsonrpc.call(client, method, params)	json.Value	call method and return its result; throws Error{kind: "jsonrpc"} on an error reply, a transport error, or a malformed reply (no result/error, or a mismatched id)
jsonrpc.notify(client, method, params)		send a notification (no id, no reply); returns nothing, but a transport error still throws Error{kind: "jsonrpc"} (the request never reached the server)

`params` is a `json.Value` : a `json.list` for positional arguments or a `json.map` for named ones (pass `json.list()` for a method that takes none). An `https://` endpoint uses TLS automatically.

```
# named params, and reading a structured result
def p as json.Value init json.set(json.map(), "/city", "berlin");
def w as json.Value init jsonrpc.call($c, "weather", $p);
io.printf("%d C\n", json.asInt($w, "/tempC"));
```

Server

`jsonrpc.handle(requestBody)` turns a raw request into a reply body - it does the whole JSON-RPC protocol (single request, notification, batch, and every reserved error code) and leaves the transport to you: read the body off `httpd` or `net` , pass it to `handle` , write the returned string back.

Each request's `method` names a **top-level method** `func NAME(params as json.Value)` in the program that imported the module (dispatched by name via `meta.callMain` - the method name arrives on the wire, so it is resolved as a runtime string, not a `func` value). The handler is called with the request's params and returns a `json.Value` **or a scalar** (int / float / string / bool / null) as its result.

Call	Returns	
jsonrpc.handle(requestBody)	string	the reply JSON, or "" when none is owed (a notification, or an all-notification batch)

```
# in the program that serves RPC:
func add(params as json.Value) {
    return json.asInt($params, "/0") + json.asInt($params, "/1");
}

# ... in your accept loop, given the request body:
def reply as string init jsonrpc.handle($body);
if (len($reply) > 0) {
    httpd.respond($req, 200, $reply);   # a notification returns "" - send 204
}
```

A missing method is a `-32601` (method not found) reply; a handler that **throws** becomes a generic `-32603` (internal error) reply - the thrown message stays server-side and is **not** put on the wire, so you can raise detailed errors without leaking internals. An unparseable body is `-32700` (parse error). A handler must take exactly one `json.Value` parameter.

> **Security - the whole top-level namespace is exposed.** `handle` resolves a request's `method` against *every* top-level `func` in the program (via `meta.callMain`); any one that takes a single `json.Value` argument is remotely callable. There is no route allow-list like `web` 's. So:
>
> - Name RPC handlers deliberately - a shared prefix, or a dedicated dispatch file - and do not co-locate a `handle` -served program with privileged one-argument helpers.
> - Authenticate at the transport: check a header / token **before** calling `handle` ; the module does no authentication of its own.
> - A batch is processed entry by entry with no size limit of its own (the transport's body cap bounds it); cap the request size upstream if untrusted.

Error codes

Constants for the reserved codes (JSON-RPC 2.0 section 5.1); application errors use the `-32000..-32099` server-error range or any code outside the reserved block.

Constant	Code	
jsonrpc.PARSE_ERROR	-32700	invalid JSON received
jsonrpc.INVALID_REQUEST	-32600	not a valid request object
jsonrpc.METHOD_NOT_FOUND	-32601	no such method
jsonrpc.INVALID_PARAMS	-32602	invalid method parameters
jsonrpc.INTERNAL_ERROR	-32603	internal JSON-RPC error

Scope

- **HTTP transport.** The client speaks JSON-RPC over HTTP POST (the common case); `handle` is transport-agnostic, so a raw-TCP / newline-framed server is a matter of wiring it to `net` .
- **Client is single-call.** `call` / `notify` send one request; the server `handle` accepts a batch (a JSON array) on the receiving side. A client-side batch builder is a possible later add.
- **Handler result shape.** A handler returns a `json.Value` or a scalar; a raw Jennifer `list` / `map` / struct is

not coerced - build a `json.Value` result explicitly. A custom per-handler error code (beyond the generic `-32603`) is a later add; today a thrown error maps to internal-error with a fixed message.

- **Client correlation.** `call` sends a fixed request `id` and rejects a reply whose `id` does not echo it. It is a synchronous single-call client, so there is no in-flight pipelining to correlate across.

See also

- json.md - the value type params and results are built from and read with.
- http.md - the client transport (`https://` via TLS).
- rest.md / web.md - the REST client and web framework, the other `http` -backed request/response modules.
- modules/index.md - the module catalog and import rules.

jwt - JSON Web Tokens (RFC 7519)

Import with `import "jwt.j" as jwt;` . Sign, verify, and decode compact JWTs. Claims are a `json.Value` object; a token is the usual `header.payload.signature` of base64url segments.

Ten algorithms across four families:

Family	Algorithms	Key (bytes)
HMAC	HS256 / HS384 / HS512	a shared secret
RSA	RS256 / RS384 / RS512	a PEM RSA key (PKCS#1 / PKCS#8 / PKIX)
ECDSA	ES256 / ES384 / ES512	a PEM EC key (SEC1 / PKCS#8 / PKIX)
EdDSA	EdDSA	a raw Ed25519 key from crypto.signKeypair

The key is **always bytes** - a secret, a PEM blob read from disk, or a raw Ed25519 key - and which one is decided by the algorithm family.

```
use convert;
import "jwt.j" as jwt;

def secret as bytes init convert.bytesFromString("topsecret", "utf-8");
def claims as json.Value init json.decode("{\"sub\":\"ada\",\"exp\":9999999999}");

def token as string init jwt.sign($claims, $secret, "HS256");
def back as json.Value init jwt.verify($token, $secret, "HS256");   # throws if
invalid
```

Runnable: `examples/modules/jwt_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/jwt_demo.j).

Surface

Call	Returns	Notes
jwt.sign(claims, key, alg)	string	Sign claims (a json.Value) into a compact JWT.
jwt.verify(token, key, alg)	json.Value	Verify the signature, the header algorithm, and exp / nbf; return the claims. Throws on any failure.
jwt.verifyLeeway(token, key, alg, leeway)	json.Value	Like verify, but widen the exp / nbf checks by leeway seconds each way for clock skew. verify is verifyLeeway(..., 0).
jwt.verifyWith(token, key, alg, policy)	json.Value	Like verify, then additionally enforce the jwt.Policy claim checks (issuer / audience). Throws on any failure.
jwt.verifyWithKeys(token, keysByKid, alg)	json.Value	Select the verification key by the header's kid from a map of string to string (secret or PEM text), then verify.
jwt.verifyJwks(token, jwksJson, alg)	json.Value	Verify against a JWKS ({"keys":[...]}): select the JWK by the header's kid, convert it to a key with crypto.jwkToPem, then verify. Asymmetric algs (RS* / ES*) only; needs the default binary.
jwt.decode(token)	json.Value	The payload claims without verifying - for inspection only.
jwt.header(token)	json.Value	The token header (its alg / kid), also without verifying.

Verifying safely

`jwt.verify` takes the algorithm you **expect** as its third argument, and this is deliberate - it is the difference between a safe verifier and a vulnerable one:

- **Algorithm confusion is rejected.** A token whose header `alg` does not equal the `alg` you pass is refused.

This closes the classic attack where an attacker re-signs a token as `HS256` using your RSA *public* key as the HMAC secret: you asked for `RS256` , so the `HS256` token never gets that far. Never derive the verification algorithm from the token's own header.

- **"none" is not an algorithm.** It is not in the supported set, so a token claiming `alg: none` cannot be verified (only `decode` d).
- **Time claims are enforced.** When present, `exp` (expiry) rejects an expired token and `nbf` (not-before) rejects one that is not yet valid, both against the current time. `NumericDate` claims are Unix seconds.
- **Clock skew is a leeway.** Issuer and verifier clocks rarely agree to the second, so `jwt.verifyLeeway($token, $key, $alg, $leeway)` widens each time check by `leeway` seconds: a token is accepted while `now` < `exp` + `leeway` and `now` >= `nbf` - `leeway` . `jwt.verify` is exactly `verifyLeeway(..., 0)` , so the strict, zero-tolerance behaviour stays the default; opt into a small leeway (30-60s is typical) only where clock drift is real. `leeway` must be non-negative.
- **HMAC comparison is constant-time** (`crypto.hmacEqual`), so a wrong MAC leaks nothing through timing.
- **Segments must be canonical base64url.** A segment with stray = padding or non-zero trailing bits decodes to the same bytes as the canonical spelling - a *second token string* that verifies as the same token, which breaks anything keyed on the token string (replay caches, denylists). Such spellings are rejected as malformed, matching strict JWS implementations.
- **Issuer / audience are application policy.** `jwt.verify` checks the signature, algorithm, and time claims; it does not check `iss` or `aud` , because the expected values are the application's, not the library's. Use `jwt.verifyWith($token, $key, $alg, jwt.Policy{iss: "https://issuer.example", aud: "my-api"})` to additionally require them. An empty `iss` (or `aud`) skips that check; a non-empty one must match. `aud` may be a JSON string or an array of strings (RFC 7519), and the check passes when the expected audience is present.

`jwt.decode` and `jwt.header` do **not** verify anything - use them only to read a token you have not trusted yet (for example, to read `kid` before fetching the matching key). Never authorize on their output.

Building claims

Claims are a `json.Value` , so build them however you build JSON - decode a literal, or assemble one with the `json` write surface:

```
use json;
use time;
import "jwt.j" as jwt;

def exp as int init time.unix(time.now()) + 3600;      # one hour
def claims as json.Value init json.decode(
    "{\"sub\":\"ada\",\"role\":\"admin\",\"exp\":" + convert.toString($exp) +
"}");
def token as string init jwt.sign($claims, $secret, "HS256");
```

Keys by family

- **HS*** : any `bytes` secret. Use a long, random secret (`crypto.randBytes(32)`).
- **RS* / ES*** : a PEM key as `bytes` - read it with `fs.readBytes("key.pem")` . Signing needs the private key, verifying needs the public key. Standard PEM encodings are accepted (RSA PKCS#1 or PKCS#8; EC SEC1 or PKCS#8; public keys in PKIX / `-----BEGIN PUBLIC KEY-----`).
- **EdDSA** : the `public` / `private` `bytes` from `crypto.signKeypair()` .

`jwt` does not generate RSA or EC keys - bring your own (from your identity provider, or `openssl`). It generates Ed25519 keys through `crypto.signKeypair` .

Selecting a key by kid

When an issuer rotates keys, each token's header carries a `kid` (key id) naming which key signed it.

`jwt.verifyWithKeys` reads that `kid` , looks it up in a caller-supplied `map of string to string` , and verifies with the matching value

- an HMAC secret for HS\ , *or PEM key text for RS*\ / ES* (each is decoded to

`bytes` for you). A token with no `kid` , or a `kid` absent from the map, is an error - never a silent skip.

```
def keys as map of string to string init {
    "2024-key": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----\n",
    "2025-key": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----\n"
};
def claims as json.Value init jwt.verifyWithKeys($token, $keys, "RS256");
```

`alg` is still required and enforced against the header: selecting the key by `kid` does **not** select the algorithm, so the algorithm-confusion protection above stays in force. Pin `alg` to what the issuer uses.

For a raw **JWKS** endpoint (`{"keys":[{"kty":..,"kid":..,"n":..,"e":..}, ...]}`), `jwt.verifyJwks` does the resolution for you: it finds the JWK whose `kid` matches the header, converts it to a PEM public key with `crypto.jwkToPem` , and verifies - no manual JWK-to-key step. It is for asymmetric algorithms (RS* / ES*) only, since a JWKS carries public keys; for HMAC use `verifyWithKeys` with the shared secret.

```
def jwks as string init http.get("https://issuer.example.com/.well-known/jwks.
json", {}).body;
def claims as json.Value init jwt.verifyJwks($token, $jwks, "RS256");
```

JWKS. A raw JWKS endpoint publishes keys as JSON Web Keys (`{"kty","kid","n","e"}` for RSA, `{"kty","kid","crv","x","y"}` for EC), not PEM. `jwt` does not consume a JWKS directly: turning a JWK's modulus/exponent (or EC point) into a verification key requires ASN.1 DER encoding that the `crypto` surface does not expose, and hand-rolled DER in `.j` would be fragile and a poor fit for security-sensitive code. Resolve the JWKS to a `kid -> PEM` (or `kid -> secret`) map out of band - for example with your identity provider's tooling or `openssl` - and pass that map to `verifyWithKeys` . A `crypto.jwkToPem` helper that would let `jwt` ingest a JWKS in-process is a noted follow-up.

JWT as web auth (jwt_auth)

There is no separate `jwt_auth` module - JWT authentication is this module used as a `web before` middleware. Register a guard that pulls the bearer token from the `Authorization` header, verifies it, and rejects on failure:

```
func requireJwt(ctx as web.Context) {
    def auth as string init web.header($ctx, "Authorization");
    if (not strings.startsWith($auth, "Bearer ")) {
        web.respond($ctx, 401, "missing bearer token");
        return false;                          # stop the chain
    }
    def token as string init strings.substring($auth, 7, len($auth));
    try {
        def claims as json.Value init jwt.verify($token, JWT_SECRET, "HS256");
        web.set($ctx, "user", json.asString($claims, "/sub"));
    } catch (e) {
        web.respond($ctx, 401, "invalid token");
        return false;
    }
    return true;                                # continue to the handler
}
```

Platforms

HS* (over `hash.hmac`) and `EdDSA` (over `crypto.sign` / `verify`) run on **both binaries** . RS* / ES* need

the `crypto` library's RSA / ECDSA surface, which is on the default `jennifer` binary; on `jennifer-tiny` they raise a friendly "not available" error (the same split as `net`).

kvstore - selectable key/value backend

Import with `import "kvstore.j" as kvstore;` . A **backend selector** for a key/value store with per-key TTL: one `Store` value is a `memcache` connection, a `redis` connection, or the in-process `kv` library, and a uniform `set` / `get` / `delete` / `touch` / `incrWindow` surface dispatches to whichever you chose. It is the shared backend layer under `session` and `ratelimit` , so each works over any backend without duplicating the plumbing.

```
import "kvstore.j" as kvstore;

# pick one backend:
def store as kvstore.Store init kvstore.redisStore($rc);         # distributed
(redis)
# def store as kvstore.Store init kvstore.memcacheStore($mc);    # distributed
(memcache)
# def store as kvstore.Store init kvstore.inProcessStore();      # local, in
memory
# def store as kvstore.Store init kvstore.fileStore("state.kv"); # local,
persisted

kvstore.set($store, "greeting", "hello", 60);   # expires in 60 s
io.printf("%s\n", kvstore.get($store, "greeting"));
```

Store is an enum, not a tagged struct

`Store` is a sum type - each variant carries only its own backend handle, and every op is a `match` the compiler checks for **exhaustiveness** :

```
export def enum Store {
    Memcache { mc as memcache.Session },
    Redis { rc as redis.Session },
    Local { local as kv.Store }
};
```

Surface

Call	Backend	
kvstore.memcacheStore(mc)	memcache	distributed; state on the server
kvstore.redisStore(rc)	redis	distributed; state on the server
kvstore.inProcessStore()	local	in memory, this process only (reset each run)
kvstore.fileStore(path)	local	persisted to path, survives across runs
kvstore.set(store, key, value, ttl)		store with a ttl-second expiry (0 = none)
kvstore.get(store, key)		the value, or "" when absent / expired
kvstore.delete(store, key)		remove the key; bool (existed)
kvstore.touch(store, key, ttl)		re-arm the expiry; bool (existed)
kvstore.incrWindow(store, key, ttl)		atomic increment, creating the key at 1 with a ttl TTL on the first hit; the new value. The fixed-window rate-limit primitive, portable across all backends.

Choosing a backend

- **Distributed** (`memcache` / `redis`) - state lives on the server, shared across every process and machine that connects. The right choice for a web fleet, a worker pool, or anything where more than one process must agree.

- **Local in-memory** (`inProcessStore`) - no server, fastest, but gone when the program exits and invisible to other processes. Good for a single-process cache or a test. Expired entries are swept periodically, so a rate limiter over it stays bounded; but there is no cap on *unexpired* data, so an **adversarial, unbounded working set** (millions of live sessions from an attacker) belongs on a distributed backend with a server-side eviction policy, not here.
- **Local persisted** (`fileStore`) - no server, and it survives across `jennifer run` invocations (handy for development and CLI tools that keep state). Single-process: safe for `spawn` within one process, but concurrent *separate* processes on one file can lose writes - use a distributed backend for cross-process coordination.

See also

- kv.md - the in-process store behind the local backend.
- memcache.md / redis.md - the distributed backends.
- session.md / ratelimit.md - the modules built on this selector.
- modules/index.md - the module catalog and import rules.

label - industrial label printing

Import with `import "label.j" as label;` . Describe and print labels for industrial label printers. **One** module, one way to describe a label, with the printer language as a **selectable backend** (a `Device` dialect) rather than a module per printer. A deliberate **three-stage pipeline** keeps the stages independent:

1. **build** a device-independent `Label` in millimetres,
2. **render** it to a chosen dialect string, and
3. **emit** that string anywhere.

Build and render are pure text and run on **both** binaries; only `send` (the `:9100` convenience) uses `net` , so it needs the default **jennifer** binary.

```
import "label.j" as label;

def l as label.Label init label.new(50.0, 30.0);          # 50 x 30 mm
def t as label.TextOptions; $t.height = 4.0;
$l = label.text($l, 5.0, 5.0, $t, "HELLO");
def o as label.BarcodeOptions;                            # zero-value = defaults
$l = label.barcode($l, 5.0, 15.0, "code128", $o, "12345678");
def zpl as string init label.render($l, label.zpl(203));
# label.send("192.168.1.50", 9100, $zpl);                 # to a printer's raw
port
```

Runnable: `examples/modules/label_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/label_demo.j).

Stage 1 - build (device-independent, millimetres)

Every builder is value-semantic and returns a new `Label` , so a label is assembled by reassignment. Coordinates and sizes are **millimetres** - never device dots.

Call / type	Notes
label.Label	width, height (mm), quantity, fields.
label.Field	one placed field (kind "text"/"barcode"/"box"/"image").
label.TextOptions	height (mm), points (pt, wins over height), rotation (0/90/180/270), bold.
label.new(width, height)	A new empty label of that size in mm (quantity 1).
label.text(label, x, y, opts, content)	Place text at (x, y); opts is a TextOptions.
label.barcode(label, x, y, type, opts, data)	Place a barcode (symbologies + opts below).
label.box(label, x, y, w, h, thickness)	Place a rectangular outline (all mm).
label.image(label, x, y, name)	Place a pre-stored image by name (native size).
label.quantity(label, n)	Set the number of copies.

`TextOptions` is zero-value-friendly (`def t as label.TextOptions;`): an unrotated, non-bold field sized by `height` mm. Set `points` for a point-sized font (it wins over `height`), `rotation` to turn it (degrees counter-clockwise), and `bold` for a bold face. Rotation and point size are portable; both dialects honour them (cab via the `r` / `ptN` fields, ZPL via the field-orientation letter and a dots conversion).

Barcode `type` is a **linear** symbology - `"code128"` , `"ean13"` , `"ean8"` , `"itf"` (Interleaved 2 of 5), `"code39"` , `"gs1-128"` - or a **2D** symbology - `"datamatrix"` , `"qr"` . GS1-128 data uses the parenthesised Application Identifier form (`(00)3006...`). `label.image` references an image already stored on the printer (cab: the `images/` folder; ZPL: a stored graphic); `name` is the stored name in that dialect's convention.

`opts` is a `label.BarcodeOptions` refining the barcode; a zero-value struct (`def o as`

`label.BarcodeOptions;`) means the defaults:

Field	Effect
height (float, mm)	Bar height (linear) or module size (2D); 0 uses the default (15 mm / 1 mm).
checkDigit (string)	Append an auto-computed check digit: "mod10", "mod11", "mod16", "mod36", "mod43" ("" = none). On cab this is +MODxx; on ZPL it toggles a symbology's native check (Code 39 / ITF) - Code 128 / EAN / GS1 carry the check digit in the data.
errorLevel (string)	2D error-correction level "L"/"M"/"Q"/"H" ("" = default).
hideText (bool)	true suppresses a linear code's human-readable line.
moduleWidth (float, mm)	A linear code's narrow-element width; 0 uses the dialect default (cab writes it as ne; ZPL uses its default ^BY module width).
ratio (float)	The wide:narrow bar ratio for a ratio-based code (Interleaved 2 of 5 / Code 39); 0 uses the default (3).

ITF (Interleaved 2 of 5, the standard shipping-carton symbology) is numeric-only and even-length because the encoding pairs digits: `label.barcode` rejects non-numeric ITF data (a catchable `Error` , kind `"label"`) and pads odd-length data with a leading zero (so a 13-digit body becomes ITF-14). An unknown barcode `type` also throws.

Stage 2 - render (to a dialect)

`label.render(label, device)` returns the command stream for the device's dialect as a plain string. Build the `device` with a constructor rather than a raw literal:

Constructor	Notes
label.zpl(dpi)	A ZPL target at the given printer resolution.
label.cab()	A cab target with default print-setup.
label.cabWith(setup)	A cab target carrying an explicit label.CabSetup.

The resolution converts millimetres to dots for raster dialects; millimetre- native dialects ignore it. An unknown dialect throws (kind `"label"`).

- **"zpl" - Zebra Programming Language.** The dominant, public label language; cab Squix printers accept it too, so this one dialect drives most hardware. Emits `^XA` / `^FO` / `^A0` / `^FD` / `^FS` , `^BY` / `^BC` (with `^BE` for EAN-13, `^B8` for EAN-8, `^B2` for ITF, `^BQ` for QR), `^GB` , `^PQ` , `^XZ` , converting millimetres to dots at the target `dpi` . The `^A0` orientation letter carries text rotation; a point size converts to dots. Text is escaped via `^FH` hex for the ZPL command characters (`^` , `~` , `_`) and non-ASCII bytes.
- **"cab" - cab JScript.** The native language of cab printers, millimetre-native (it ignores `dpi`). Emits `m` / `J` / `H` / `O` / `S` / `T` / `B` / `G` / `A` per the cab JScript Programming Manual (edition 05/2025): `T x,y,r,font,size;text` (`r` = rotation, font 3 = Swiss 721 / 5 = Bold, size = `ptN` or mm), `G x,y,r;R:w,h,hD,vD` for a box, and `B x,y,r,type,size;data` where an **uppercase** type name prints the human-readable line and a **lowercase** one suppresses it, with size `height,ne` for Code 128 / EAN-13 / EAN-8, `height,ne,ratio` for Interleaved 2 of 5, and a single module size for QR.

cab print-setup (CabSetup)

The `J` (job name), `H` (heat/speed), `O` (orientation), and `S` (label sensor + geometry) lines are printer/media setup with no ZPL equivalent, so they live in a cab-only `label.CabSetup` passed via `label.cabWith(setup)` . Every field is optional; the encoder emits a command only when the matching field is set, and a zero-value setup (`label.cab()`) emits a bare `J` , no `H` / `O` , and an `S` line derived from the label size. ZPL ignores the struct entirely.

Field	cab line
jobName (string)	J <name> (bare J when empty).

heat (int), speed (int), mode (string)	H <heat>,<speed>,<mode> (omitted when all are zero/empty).
orientation (string)	O <orientation> (e.g. "R"; omitted when empty).
sensor (string)	the S photocell/sensor type (e.g. "l1" = die-cut with gap -> S l1;...).
xOffset, yOffset (float, mm)	the S horizontal / vertical origin offset.
height (float, mm)	the S label height (transport direction). width 0 derives the whole S line from the label size.
pitch (float, mm)	the S label pitch = label height + the gap between labels.
width (float, mm)	the S label width.
columnPitch (float, mm), columns (int)	multi-up dies: the horizontal column pitch and label count; appended only when columns > 1.

The S line is `S [sensor;]xo,yo,ho,dy,wd[,dx,col]` . So a 4-up die of 17 x 12 mm die-cut labels with a 3 mm gap (pitch 15) at a 20 mm column pitch is:

```
def setup as label.CabSetup;
$setup.jobName = "Shipping";
$setup.heat = 100; $setup.speed = 5; $setup.mode = "T,R0";
$setup.orientation = "R";
$setup.sensor = "l1";
$setup.height = 12.0; $setup.pitch = 15.0; $setup.width = 17.0;
$setup.columnPitch = 20.0; $setup.columns = 4;   # -> S l1;0.0,0.0,12.0,15.0,17.
0,20.0,4
def job as string init label.render($l, label.cabWith($setup));
```

> **cab dialect note.** The encoder follows the cab JScript Programming Manual (edition 05/2025) and emits the manual's canonical forms - full barcode names rather than the deprecated one-letter short codes. The derived S label-size line uses gap 0 (`dy` = `height`); set the `CabSetup` `height` / `pitch` / `width` (and `columnPitch` / `columns` for a multi-up die) to match your media.

Stage 3 - emit (transport-agnostic)

The rendered string is yours to deliver: write it to a `*.prom` -style spool file or a USB device node with `fs` , store it, or send it over the network. The module ships one convenience for the common case:

Call	Notes
label.send(host, port, rendered)	Open a TCP connection and write the stream (raw :9100).

Keeping emit separate from render is what makes the same label printable, saveable, and testable without a printer attached.

Testing

The pure logic - the millimetre-to-dots conversion, ZPL hex escaping, ITF validation / padding, and both dialects' exact command output for a sample label - is unit-tested in the overlay (`modules/label_test.j`). The `send :9100` path is covered against an in-process fake printer in the Go test suite (`TestLabelSend`).

Out of scope

- **Two dialects** (`zpl` , `cab`). Adding another is a new encoder plus a dialect string, with no change to the build API.
- **Images are by reference only.** `label.image` recalls an image already stored on the printer; embedding a bitmap in the job (converting a PNG to the dialect raster) is a planned follow-on.
- **Text rotation, point sizes, and a bold face** are covered by `TextOptions` ; barcode size, narrow-element width, check digit, 2D error level, and the human-readable line by `BarcodeOptions` ; cab print-setup by

`CabSetup` . Full font selection beyond regular/bold is still a follow-on. The long-tail symbologies (Aztec, MaxiCode, PDF417, the GS1 DataBar family, ...) are added the same way when needed.

- **Brother ESC/P** is raster/bitmap, not a field command language, so it does not fit this vector-field model and is not a planned dialect.

See also

- net.md - the transport `send` uses.
- fs.md - for spooling a rendered label to a file / device.
- modules/index.md - the module catalog and import rules.

ldap - LDAP v3 client and directory server

Import with `import "ldap.j" as ldap;` . An LDAP v3 client and a lightweight, in-memory directory server (RFC 4511). The wire messages are ASN.1 BER, built and parsed with the `asn1` library; the transport is `net` TCP, with LDAPS / StartTLS through `net` 's TLS. SASL SCRAM binds reuse the `sasl` module; the mutable server directory is backed by the `kv` store. Needs the default `jennifer` binary (`net`).

```
import "ldap.j" as ldap;
import "transport.j" as transport;
use io;

def c as ldap.Conn init ldap.connect("localhost:389", transport.Security.None);
ldap.bind($c, "cn=admin,dc=example,dc=org", "secret");
def hits as list of ldap.Entry init ldap.search($c,
    "ou=people,dc=example,dc=org", ldap.SCOPE_SUB,
    ldap.parseFilter("(uid=alice)"), ["cn", "mail"]);
for (def e in $hits) {
    io.printf("%s -> %s\n", $e.dn, ldap.firstValue($e, "mail"));
}
ldap.unbind($c);
```

Runnable: `examples/modules/ldap_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/ldap_demo.j).

Values

Entries, attributes, and results come back as small structs:

```
export def struct Conn { handle as net.Conn, timeoutMs as int };
export def struct Attribute { name as string, values as list of string };
export def struct Entry { dn as string, attributes as list of Attribute };
export def struct Result { code as int, matchedDn as string, message as string };
```

An `Entry` is a DN plus multi-valued attributes. Read an attribute with `ldap.values(entry, name)` (a `list of string` , case-insensitive name) or `ldap.firstValue(entry, name)` (the first value, or `""`). A `Result` carries the LDAP result `code` - `ldap.SUCCESS` (0) on success, `ldap.INVALID_CREDENTIALS` (49) on a failed bind.

Client

Connecting

`ldap.connect(address, security)` opens a connection; `security` is a `transport.Security` value:

```
import "transport.j" as transport;

def a as ldap.Conn init ldap.connect("dir.example.org:389", transport.Security.
None);      # plaintext
def b as ldap.Conn init ldap.connect("dir.example.org:636", transport.Security.
Tls);       # LDAPS
def c as ldap.Conn init ldap.connect("dir.example.org:389", transport.Security.
Starttls);  # StartTLS
```

`transport.Security.Starttls` connects in plaintext and upgrades in-band via the StartTLS extended operation (or call `ldap.startTls(conn)` yourself). `ldap.close` closes without an unbind; `ldap.unbind` sends an unbind request and closes.

Binding

```
def r as ldap.Result init ldap.bind($c, "cn=admin,dc=example,dc=org", "secret");
if ($r.code == ldap.SUCCESS) { io.printf("bound\n"); }
```

`ldap.bind` performs a simple bind and returns the `Result` (it does not throw on a bad password - check `.code`). An empty DN and password is an anonymous bind. `ldap.bindSasl(conn, user, password, algo)` performs a SASL SCRAM bind (`algo` `"sha1"` or `"sha256"`), verifying the server signature.

Searching and filters

`ldap.search(conn, baseDn, scope, filter, attributes)` returns the matching entries; an empty `attributes` list requests all user attributes. `scope` is one of `ldap.SCOPE_BASE`, `ldap.SCOPE_ONE`, or `ldap.SCOPE_SUB`.

Build the filter from an RFC 4515 string with `ldap.parseFilter`, or from the constructors:

```
def f1 as asn1.Value init ldap.parseFilter("(&(objectClass=person)(uid=alice))");
def f2 as asn1.Value init ldap.allOf([ldap.equals("objectClass", "person"), ldap.
present("mail")]);
```

Constructor	Filter
ldap.equals(attr, value)	(attr=value)
ldap.present(attr)	(attr=*)
ldap.greaterOrEqual(attr, value)	(attr>=value)
ldap.lessOrEqual(attr, value)	(attr<=value)
ldap.approx(attr, value)	(attr~=value)
ldap.substrings(attr, initial, anyParts, final)	(attr=initial*any*final)
ldap.allOf(filters)	(&(...)(...))
ldap.anyOf(filters)	`(
ldap.negate(filter)	(!(...))

`ldap.searchPaged(conn, baseDn, scope, filter, attributes, pageSize)` pulls large result sets a page at a time via the simple-paged-results control.

Attribute values come back as strings. A **binary** value (one that is not valid UTF-8 - Active Directory `objectGUID` / `objectSid`, `userCertificate`, `jpegPhoto`) is returned **base64-encoded** (the LDIF convention) rather than throwing; decode it with `encoding.fromText(value, "base64")`.

Writing

The write operations return a `Result` (check `.code` against `ldap.SUCCESS`):

```
ldap.add($c, "uid=jdoe,ou=people,dc=example,dc=org", {
    "objectClass": ["inetOrgPerson"], "uid": ["jdoe"], "cn": ["J Doe"], "mail":
["jdoe@example.org"]
});
ldap.modify($c, "uid=jdoe,ou=people,dc=example,dc=org", [
    ldap.change(ldap.MOD_REPLACE, "mail", ["j.doe@example.org"]),
    ldap.change(ldap.MOD_ADD, "description", ["created via ldap"])
]);
ldap.modifyDn($c, "uid=jdoe,ou=people,dc=example,dc=org", "uid=jd", true, "");
# rename (newSuperior "" keeps the parent)
ldap.passwordModify($c, "uid=jd,ou=people,dc=example,dc=org", "oldpw", "newpw");
# RFC 3062
ldap.delete($c, "uid=jd,ou=people,dc=example,dc=org");
```

`ldap.change(operation, name, values)` builds one modify change; `operation` is `ldap.MOD_ADD`,

`ldap.MOD_DELETE` , or `ldap.MOD_REPLACE` (an empty value list on `MOD_DELETE` removes the whole attribute). `ldap.passwordModify` uses the RFC 3062 extended operation - pass `userDn ""` for the currently-bound user, `oldPassword ""` for an administrative reset, and `newPassword ""` to have the server generate one (do it over TLS). These operations require a directory server that accepts writes; the built-in server is read-only over LDAP.

Active Directory notes

The client works against Active Directory over LDAPS or StartTLS with a simple bind (`user@domain` , `DOMAIN\user` , or a full DN). AD caps a result set at 1000 rows, so use `ldap.searchPaged` for large searches (it sends AD's required paged control). Bind to a specific domain controller, or the Global Catalog on port `3268` , to avoid referrals. AD's native SASL is Kerberos (GSSAPI), which this module does not implement - use simple bind over TLS. Change a password with `ldap.passwordModify` over TLS (AD requires an encrypted connection for it).

Server

The server answers **simple bind** and **search** for an in-memory directory. It is read-only over the LDAP protocol (it rejects add / modify / delete on the wire), but the directory itself is **fully mutable from your own code** - so a web interface or a database sync can update the directory a running server is serving, and the change is visible immediately (the directory is a shared `kv` -backed store). There is no schema enforcement and no ACLs.

Building a directory

```
def dir as ldap.Directory init ldap.directory([
    ldap.entry("uid=alice,ou=people,dc=example,dc=org", {
        "objectClass": ["inetOrgPerson", "person"],
        "uid": ["alice"], "cn": ["Alice A"], "mail": ["alice@example.org"],
        "userPassword": [ldap.password("secret", "ssha")]
    }),
    ldap.group("cn=admins,ou=groups,dc=example,dc=org",
["uid=alice,ou=people,dc=example,dc=org"])
]);
```

`ldap.directory` is in-memory (ephemeral) - enough for a lightweight server. For a directory that **persists across restarts** , use `ldap.openDirectory(path)` instead: it is backed by a `kv.openFile` store, so every edit is written to disk. A new file starts empty; seed it on first run by checking `ldap.listEntries` and `ldap.addEntry` -ing when it is empty.

`ldap.entry(dn, attrs)` builds an entry from a `map of string to list of string` ; `ldap.group(dn, members)` builds a `groupOfNames` (its `cn` is taken from the DN's RDN). `ldap.password(plain, scheme)` hashes a password into a `userPassword` value. `scheme` is:

scheme	stored form	notes
"plain"	cleartext	testing only
"sha" / "ssha"	{SHA} / {SSHA}	SHA-1 (salted); fast hash, interop only
"sha256" / "ssha256"	{SHA256} / {SSHA256}	SHA-256 (salted); still a fast hash
"pbkdf2"	{PBKDF2-SHA256}$iterations$salt$hash	recommended - a slow KDF (crypto.pbkdf2, 100000 iterations)
"pbkdf2-sha512"	{PBKDF2-SHA512}$...	slow KDF over SHA-512

The `{SHA}` / `{SSHA}` names follow the OpenLDAP / RFC 2307 convention for interop, but they are fast hashes - weak for password storage even when salted. Prefer `"pbkdf2"` unless you need to interoperate with a directory

that expects `{SSHA}` . `ldap.verifyPassword` (used by the server on bind) accepts every scheme. The server withholds `userPassword` from search results unless it is explicitly requested.

Mutating a live directory

```
ldap.addEntry($dir, ldap.entry("uid=bob,ou=people,dc=example,dc=org", { ... }));
# insert / replace by DN
ldap.modifyEntry($dir, $entry);
# replace by DN
ldap.setAttribute($dir, "uid=bob,ou=people,dc=example,dc=org", "mail",
["bob@example.org"]);
ldap.deleteEntry($dir, "uid=bob,ou=people,dc=example,dc=org");
def e as ldap.Entry init ldap.getEntry($dir,
"uid=bob,ou=people,dc=example,dc=org");
def all as list of ldap.Entry init ldap.listEntries($dir);
def ok as bool init ldap.hasEntry($dir, "uid=bob,ou=people,dc=example,dc=org");
```

These edits are visible to a running server immediately, so the directory can be built from a database at startup and kept in sync by your own admin tooling while the server serves auth requests from it. (Serialize your writes: a mutation is a read-modify-write on the shared store, so a single admin path avoids a lost update.)

Serving

```
def dir as ldap.Directory init ldap.directory([ ... ]);
ldap.serve($dir, ":389");   # blocks, one spawn per connection
```

For a stoppable server, drive the listener yourself: `ldap.listen(address)` -> `net.Listener` , `ldap.serveOn(dir, listener)` runs the accept loop until the listener is closed (close it from another task to stop).

Configuring Authelia against this server

Authelia (https://www.authelia.com) can use the Jennifer LDAP server as its authentication backend. Build a directory with an admin bind account, a people subtree, and a groups subtree:

```
import "ldap.j" as ldap;

def dir as ldap.Directory init ldap.directory([
    ldap.entry("cn=admin,dc=example,dc=org", {
        "objectClass": ["person"], "cn": ["admin"],
        "userPassword": [ldap.password("adminpassword", "ssha")]
    }),
    ldap.entry("uid=alice,ou=people,dc=example,dc=org", {
        "objectClass": ["inetOrgPerson"], "uid": ["alice"],
        "cn": ["Alice A"], "displayName": ["Alice A"], "mail": ["alice@example.
org"],
        "userPassword": [ldap.password("alicepassword", "ssha")]
    }),
    ldap.group("cn=admins,ou=groups,dc=example,dc=org",
["uid=alice,ou=people,dc=example,dc=org"])
]);
ldap.serve($dir, ":389");
```

Then point Authelia's `authentication_backend.ldap` at it:

```
authentication_backend:
  ldap:
    address: 'ldap://127.0.0.1:389'
    implementation: 'custom'
    base_dn: 'dc=example,dc=org'
    additional_users_dn: 'ou=people'
    additional_groups_dn: 'ou=groups'
    users_filter: '(&({username_attribute}={input})(objectClass=inetOrgPerson))'
    username_attribute: 'uid'
    display_name_attribute: 'displayName'
    mail_attribute: 'mail'
    groups_filter: '(&(member={dn})(objectClass=groupOfNames))'
    group_name_attribute: 'cn'
    user: 'cn=admin,dc=example,dc=org'
    password: 'adminpassword'
```

Authelia binds as the `user` account, searches `users_filter` under `additional_users_dn` for the login name, binds again as the found user DN to check the password, then searches `groups_filter` for the user's groups - all of which this server answers. Put TLS in front (`transport.Security.Tls` on a 636 listener, or a reverse proxy) before exposing it beyond localhost. The same setup works for any LDAP-authenticating application; Authelia is one example.

PAM authentication

A Jennifer program can also be a system authentication helper for Linux PAM via `pam_exec.so` , which passes the username in the `PAM_USER` environment variable and (with `expose_authtok`) the password on stdin, then reads the helper's exit code (0 = allow). `examples/modules/pam_auth.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/pam_auth.j) is a ready-made helper that binds against this directory to authenticate; point a `/etc/pam.d/<service>` `auth` line at it (`auth sufficient pam_exec.so expose_authtok quiet /path/to/pam_auth.j`). It fails closed - a missing user, an empty password, or an unreachable server denies - and its `checkCredentials` is easy to repoint at a `kvstore` or a PBKDF2-hash file instead of LDAP.

Errors

Client and server failures throw an `Error` with kind `"ldap"` (a malformed message, a transport failure, a refused StartTLS, a failed SASL verification). LDAP-level result codes on a bind are returned in the `Result` , not thrown, so a bad password is a value to inspect rather than an exception.

log - leveled structured logging

Import with `import "log.j" as log;` . Leveled, **structured** logging: a `log.Logger` carries a minimum level, an output format, and a sink; `log.info(logger, message, fields)` (and the sibling levels) render one record - a timestamp, the level, the message, and the caller's key/value `fields` - and write it, dropping records below the logger's level.

```
import "log.j" as log;

def lg as log.Logger init log.new("info", "logfmt");
def f as map of string to string init {"user": "ada", "id": "42"};
log.info($lg, "user logged in", $f);
# time=2026-... level=info msg="user logged in" user=ada id=42
```

Runnable: `examples/modules/log_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/log_demo.j).

Loggers

A `log.Logger` is value-semantic; build one with a constructor that fixes the sink:

Constructor	Sink
log.new(level, format)	standard output
log.toStderr(level, format)	standard error
log.toFile(level, format, path)	append to a file (created if missing)
log.toSyslog(level, address, app)	an RFC 5424 syslog server over UDP

`level` is the **minimum** level to emit - one of `"debug"` < `"info"` < `"warn"` < `"error"` - so a logger at `"info"` silently drops `debug` records. `format` is `"text"` , `"logfmt"` , or `"json"` (the syslog sink uses RFC 5424 framing and ignores it).

Levels

Each level has a method taking the logger, a message, and a `map of string to string` of fields (`{}` for none):

Method	
log.debug(logger, message, fields)	verbose / development detail
log.info(logger, message, fields)	normal operation
log.warn(logger, message, fields)	something unexpected but handled
log.error(logger, message, fields)	a failure
log.fatal(logger, message, fields)	an unrecoverable failure - logs then exits
log.at(logger, level, message, fields)	emit at a level chosen at runtime

The level ordering is `"debug"` < `"info"` < `"warn"` < `"error"` < `"fatal"` . A record below the logger's level is dropped before rendering, so guarded `log.debug` calls in hot paths cost only the level comparison. `log.fatal` ranks above every other level, so it is **always** emitted regardless of the logger's minimum level, and then terminates the whole program with exit code `1` - the statement after a `log.fatal` call never runs.

Context loggers

`log.with(logger, fields)` returns a **new** logger that carries `fields` as persistent context: every record emitted through it includes those fields, merged with any per-call fields (a per-call key of the same name wins). The parent logger is unchanged (value semantics), and `with` **composes** - calling it again merges the new fields on top of the carried ones:

```
def base as log.Logger init log.new("info", "logfmt");
def req as log.Logger init log.with($base, {"svc": "api", "reqId": "abc"});
log.info($req, "handling", {"port": "8080"});
# time=... level=info msg=handling svc=api reqId=abc port=8080
```

Function	
log.with(logger, fields)	derive a child logger carrying persistent fields

Formats

The same record - timestamp, level, message, fields - in each format:

Format	Example
text	2026-01-02T03:04:05Z INFO user logged in user=ada id=42
logfmt	time=2026-01-02T03:04:05Z level=info msg="user logged in" user=ada id=42
json	{"time":"2026-01-02T03:04:05Z","level":"info","msg":"user logged in","user":"ada","id":"42"}

Timestamps are RFC 3339 in UTC. A field value containing a space, a quote, or an = is double-quoted (with inner quotes escaped) in the `text` / `logfmt` forms, so pairs stay parseable; the `json` form is a proper object with the field keys alongside `time` / `level` / `msg` (a field named `time` / `level` / `msg` overrides the built-in key).

Sinks and portability

- **stdout / stderr** (via `io.printf` / `io.eprintf`) and **file** (via `fs.appendString`) work on **both** binaries.
- **syslog** sends each record as an RFC 5424 datagram over **UDP** (`net`): `<PRI>1 TIMESTAMP HOST APP - - - MSG` , where the priority is facility `user` (1) times 8 plus the level's severity (`error` =3, `warn` =4, `info` =6, `debug` =7), the host is `$HOSTNAME` (or `-`), and the fields ride in the message as logfmt pairs. Because it uses `net` , the syslog sink needs the default **jennifer** binary - the module is therefore **partial** on `jennifer-tiny` (console and file logging work; the syslog sink returns the no-network error).

Scope

- **String fields.** `fields` is a `map of string to string` - convert numbers / bools to strings at the call site (`convert.toString`). This keeps the record model simple and the value quoting predictable.
- **UDP syslog.** No TCP / TLS syslog transport, and no local `/dev/log` socket.
- **No global default logger.** Loggers are values you pass explicitly - no hidden singleton, no package-level state.

See also

- io.md - `printf` / `eprintf` , the stdout / stderr sinks.
- fs.md - `appendString` , the file sink.
- net.md - the UDP transport the syslog sink uses.
- modules/index.md - the module catalog and import rules.

markdown - render a Markdown subset to HTML and ANSI

Import with `import "markdown.j" as markdown;` . Renders a small CommonMark subset to **HTML** (through the `html` module, so escaping is handled for you) and to **styled terminal text** (through the `ansi` module). Pure Jennifer: line-oriented block parsing with a small inline scanner. Runs on either binary.

```
use io;
import "markdown.j" as markdown;

io.printf("%s\n", markdown.toHtml("# Hi\n\nA **bold** word."));
# <h1>Hi</h1><p>A <strong>bold</strong> word.</p>

io.printf("%s\n", markdown.toAnsi("- one\n- two"));   # styled on a TTY
```

Runnable: `examples/modules/markdown_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/markdown_demo.j).

Surface

Rendering (Markdown in, HTML / terminal text / PDF out):

Call	Returns	Notes
markdown.toHtml(md)	string	Render to HTML: block elements concatenated, no indentation. Safe by default: raw HTML in the source is escaped.
markdown.toHtmlWith(md, opts)	string	toHtml with an HtmlOptions; set allowRawHtml: true to pass raw HTML blocks through (trusted input only).
markdown.toAnsi(md)	string	Render to terminal text with ansi styling (self-suppressing).
markdown.render(doc, format)	string	Render a parsed (or hand-built) tree; format is "html" / "ansi".
markdown.toPdf(md)	bytes	Lay the document out to a paginated PDF (through pdf).
markdown.toPdfWith(md, opts)	bytes	toPdf with a custom PdfOptions (page size, margins, fonts, document metadata, bookmarks).
markdown.renderPdf(doc, opts)	bytes	Lay a parsed (or transformed) tree out to a PDF.
markdown.renderPdfDoc(doc, opts)	pdf.Document	Lay a tree out to a pdf.Document (not yet serialized), so document-level work - a running header / footer, extra metadata - can happen before pdf.render.
markdown.pageBreak()	Node	A page-break node; renderPdf starts the next content on a fresh page.

`toHtml` / `toAnsi` are thin wrappers over `render(parse(md), ...)` , and `toPdf` over `renderPdf(parse(md), pdfDefaults())` , so build and parse share one document model.

Reading (surface the parse tree, walked like `xml` / `html`):

Call	Returns	Notes
markdown.parse(md)	Node	Parse to the document root (typeOf "document").
markdown.typeOf(node)	string	The node kind ("heading", "link", "text", ...).
markdown.children(node)	list of Node	The node's direct children.
markdown.text(node)	string	A leaf's text, else its descendants' text concatenated.

markdown.level(node)	int	A heading's level (1-6); 0 otherwise.
markdown.attr(node, name)	string	"href" / "title" / "lang" / "align" / "ordered" / "level", or "".
markdown.get(node, sel)	Node	First node matching a /-separated selector, or an empty node.
markdown.findAll(node, sel)	list of Node	Every node matching the selector.
markdown.has(node, sel)	bool	Whether any node matches.

Authoring (build Markdown text - the inverse):

Call	Returns	Notes
markdown.header(level, s)	string	ATX heading; level is "h1".."h6" (throws otherwise).
markdown.style(kind, s)	string	Inline emphasis; kind is "bold" / "italic" / "code".
markdown.link(text, url)	string	text (url).
markdown.bullets(items)	string	Unordered list, one - item per line.
markdown.numbered(items)	string	Ordered list, 1. item upward.
markdown.codeBlock(text)	string	Fenced code block around verbatim text.
markdown.table(headings, aligns, rows)	string	GFM table from column headings, per-column alignment, and rows.
markdown.tablePretty(md)	string	Reformat every table's source columns to line up; other lines untouched.

Supported Markdown

A deliberately small CommonMark (https://commonmark.org) subset:

Block	Syntax	HTML
Heading (levels 1-6)	# H ... ###### H	<h1> ... <h6>
Paragraph	consecutive text lines	<p> (lines joined by)
Unordered list	- x / * x / + x	
Ordered list	1. x	
Nested list	indent a sub-list under an item	a child / inside the parent
Blockquote	> x (recursive: > > y)	<blockquote> (inner text parsed as blocks)
Fenced code block	...	<pre><code>
Indented code block	four-space indent after a blank line	<pre><code>
Thematic break	--- / *** / ___ (3+, spaced ok)	<hr>
Raw HTML block	a line opening with <tag / </ / <!-- to the next blank line	passed through verbatim
Table (GFM)	| a | b | + | --- | --- | row	<table> (aligned terminal columns in ANSI)

Inline	Syntax	HTML	ANSI
Bold	**text**		bold
Italic	*text*		italic
Code	text	<code>	cyan
Link	text (url)	<a href="url">	underline + (url)

Autolink	<https://x> / <a@b.com>	<a href="..."> (email gets mailto:)	underline + (url)
Image	alt (url)	<img src alt>	[image] alt (url)

Inline spans nest. A link inside `**...**` , a `code` span or `*emphasis*` inside a link label, and so on all parse to a nested tree, so `**label (x)**` renders `<strong><a href="x">label</a></strong>` rather than dropping the link. A link label may contain a `]` inside a code span or a balanced nested `[...]` pair (`[the $xs[] sugar](x)` parses correctly).

A **lazy continuation** line - a soft-wrapped line indented to an item's content column - stays part of that item rather than splitting the list. A blank line still ends the item. A nested list is a more-indented list under a parent item; a blockquote's inner lines are parsed as blocks, so a quote can hold paragraphs, lists, or nested quotes. A link's `href` , an autolink's target, and an image's `src` all pass through the same scheme allowlist (`html.safeUrl`), so `javascript:` and other script schemes render as `#` . A raw HTML block is **escaped** by `toHtml` (rendered as literal text, not markup) and dropped by `toAnsi` / the PDF renderer, which have no HTML.

HTML output and safety

`toHtml` builds an `html` node tree and renders it, so all text and every link target are correctly escaped - `&` , `<` , `>` in text and code, and `&` / `"` in an `href` - and you cannot produce malformed markup:

```
markdown.toHtml("[t](http://x/?a=1&b=2) and <b> & `x<y`");
# <p><a href="http://x/?a=1&amp;b=2">t</a> and &lt;b&gt; &amp; <code>x&lt;y</
code></p>
```

Safe by default for untrusted Markdown. A raw HTML block in the source (a line opening with `<tag>`) is **escaped** , so rendering a README, a comment, or any publisher-supplied Markdown cannot inject a `<script>` or an event-handler attribute (`<img onerror=...>`) - it renders as visible text instead. This closes the stored-XSS vector that verbatim HTML passthrough would open on the rendering origin. Inline raw HTML was already escaped, and everything the renderer builds (text, link URLs, image attributes) is escaped too.

For **trusted** Markdown where raw HTML is intended (your own docs using `<details>` , `<div align>` , badges), opt back in to verbatim passthrough:

```
markdown.toHtmlWith($trustedMd, markdown.HtmlOptions{allowRawHtml: true});
```

`allowRawHtml` defaults to false, so `toHtml(md)` and `toHtmlWith(md, markdown.HtmlOptions{allowRawHtml: false})` are identical.

Output is compact (no newlines between block elements), which diffs and round-trips predictably; wrap it in your own formatter if you need indented source. A code block's content is escaped but never treated as Markdown.

ANSI output

`toAnsi` renders for a terminal: headings and `**bold**` in bold, `*italic*` in italic, inline code in cyan, links underlined with their URL in parentheses, list items with `-` / `N.` markers, and fenced code indented and dimmed. Styling comes from the `ansi` module, which **suppresses itself when stdout is not a terminal** (or `NO_COLOR` is set) and is forced on by `FORCE_COLOR` - so piping the output gives clean plain text, and `ansi.strip(markdown.toAnsi(md))` gives it unconditionally.

Reading the document tree

`markdown.parse(md)` returns the document as a tree of `Node` s, walked with the same accessor vocabulary as `xml` / `html` - so a document can be inspected or transformed (pull the headings for a table of contents, rewrite links, lint) and then rendered, rather than going straight to a string. A node's `typeOf` is a block kind (`"document"` , `"heading"` , `"paragraph"` , `"code"` , `"list"` , `"item"` , `"table"` , `"row"` , `"cell"` , `"quote"`) or an inline kind (`"text"` , `"codespan"` , `"strong"` , `"emphasis"` , `"link"` , `"image"`). `get` / `findAll` / `has` take a `/` -separated selector whose steps are kind names, `*` (any kind), or `name[k]` (the k-th such child, 1-based), matching direct children.

```
def doc as markdown.Node init markdown.parse("# Title\n\nSee [docs](https://
example.com).\n");

# Walk the headings for an outline.
for (def h in markdown.findAll($doc, "heading")) {
    io.printf("H{markdown.level($h)}: {markdown.text($h)}\n");   # H1: Title
}

# Pull a link's target.
def a as markdown.Node init markdown.get($doc, "paragraph/link");
io.printf("{markdown.text($a)} -> {markdown.attr($a, 'href')}\n");   # docs ->
https://example.com

# Render straight from the (possibly transformed) tree.
io.printf("{markdown.render($doc, 'html')}\n");
```

`attr` reads a node's string attributes: `"href"` / `"title"` (a link or image), `"lang"` (a fenced code block's language), `"align"` (a table cell), `"ordered"` (`"true"` / `"false"` , a list), and `"level"` (a heading). `render(doc, format)` renders a `"document"` node's children (or any single node as one block); it and the parse are hardened with a nesting-depth cap and a total-node budget, so a pathological document is a catchable `"markdown"` error rather than an unbounded parse.

Authoring Markdown

The authoring helpers are the inverse of the renderer: they build Markdown *text* , so a program can assemble a document (and, since it is Markdown, round-trip it through `toHtml` / `toAnsi`):

```
use io;
import "markdown.j" as markdown;

def items as list of string init ["fast", "small", "strict"];
def doc as string init markdown.header("h1", "Jennifer") + "\n\n";
$doc = $doc + "It is " + markdown.style("bold", "great") + ". Features:\n\n";
$doc = $doc + markdown.bullets($items) + "\n\n";
$doc = $doc + "See " + markdown.link("the docs", "https://example/docs") + ".";

io.printf("%s\n", $doc);                  # Markdown source
io.printf("%s\n", markdown.toHtml($doc)); # ... or rendered
```

The text is inserted literally: a caller passing Markdown metacharacters (a `*` or `inside a heading, say) is responsible for escaping them.` header `throws a catchable value error on a level outside` "h1" `..` "h6" `, and style on a kind other than` "bold" / "italic" / "code"`.

Tables

`table` turns tabular data into a GFM (https://github.github.com/gfm/) table in one call: column `headings` , per-column `aligns` (`"left"` / `"right"` / `"center"` / `"none"` , or `[]` for all-default), and `rows` (each a `list of string`):

```
def rows as list of list of string init [];
$rows[] = ["Ada", "95"];
$rows[] = ["Bo", "88"];
io.printf("%s\n", markdown.table(["Name", "Score"], ["left", "right"], $rows));
# | Name | Score |
# | :--- | ---: |
# | Ada | 95 |
# | Bo | 88 |
```

Columns follow `headings` : a short row is padded with empty cells and extra cells are dropped, so every row is the same width. A `|` in a cell is escaped to `\|` and a newline becomes a space, so cell content can't break the table. An `align` value outside the four names throws a catchable `value` error.

The reader understands GFM tables too, so an authored table round-trips: `toHtml(markdown.table(...))` renders a `<table>` (with per-column `align`), and `toAnsi` renders aligned terminal columns. A parsed table needs a header row, a delimiter row (`| --- | :--: |`), and its data rows; cell content is inline-parsed (emphasis / code / links work in cells), and a table interrupts an open paragraph.

`tablePretty` reformats the **source** of every table in a document so its columns line up - the handcraft-then-prettify workflow, in one call - and leaves every non-table line exactly as written:

```
def messy as string init "| Name | Score |\n|:-|-:|\n| Ada | 95 |";
io.printf("%s\n", markdown.tablePretty($messy));
# | Name | Score |
# | :--- | ----: |
# | Ada  |    95 |
```

Each column is padded to its widest cell (minimum three, so the delimiter keeps its dashes), data cells follow the column's alignment, and an escaped `\|` is preserved. It is idempotent: prettifying an already-pretty table is a no-op.

Not supported

This is a subset, chosen to stay small and TinyGo-clean:

- No reference links (`[text][ref]`), setext (underlined) headings, or hard-break-on-two-spaces.
- Autolinks cover `<scheme:...>` and `<user@host>` ; bare (unbracketed) URLs in prose are not linkified.
- The raw-HTML block rule is CommonMark's "type 6" shape (a tag at the line start, to the next blank line); inline raw HTML in a paragraph is still escaped.

For anything beyond this subset, render with an external tool. The module is sized for READMEs, help text, and comment / docblock bodies, not general-purpose CommonMark conformance.

Note

`toPdf` folds PDF rendering into `markdown` as a third output beside `toHtml` / `toAnsi` , so it imports `pdf` (which pulls in `font`). That makes every `markdown` import a bit heavier even for a `toHtml` -only program - a deliberate trade for one unified rendering surface; see design-decisions.md. Text outside the WinAnsi range the standard-14 fonts encode is replaced per the `unencodable` option (below) rather than aborting the render. `PdfOptions` comes from `markdown.pdfDefaults()` (US Letter, 54-pt margins, Helvetica / Courier); copy it and tweak the fields (`pageWidth` / `pageHeight` / `margin` / `bodyFont` / `boldFont` / `italicFont` / `monoFont` / `headingFont` / `bodySize` / `tablePad`). A level-1 heading starts a new page; table columns are sized to their content (a short column doesn't crowd a long one), and `tablePad` tunes cell density.

Backgrounds. `tableHeaderFill` (a `Fill`) shades a table's header row, and `headingStyles` (a `list of HeadingStyle` , index 0 = h1, 1 = h2, ...) shades each heading level. Build a colour with `markdown.gray(level)` / `markdown.rgb(r, g, b)` (or `markdown.noFill()` for none) and a heading style with `markdown.headingStyle(fill)` :

```
def o as markdown.PdfOptions init markdown.pdfDefaults();
$o.tableHeaderFill = markdown.gray(232);
$o.headingStyles = [markdown.headingStyle(markdown.gray(205)), markdown.
headingStyle(markdown.gray(225))];
def out as bytes init markdown.toPdfWith($md, $o);
```

Document metadata + bookmarks. Set `title` / `author` / `subject` / `keywords` on `PdfOptions` to write the PDF Info dictionary (shown in a viewer's Document Properties). Set `bookmarkLevel` to add a navigation outline: every heading up to that level becomes a bookmark, nested by heading level - `bookmarkLevel` = `2` bookmarks all # and ## headings, `1` only # , `0` (the default) none.

```
def o as markdown.PdfOptions init markdown.pdfDefaults();
$o.title = "My Report";
$o.author = "Ada Lovelace";
$o.bookmarkLevel = 2;
def out as bytes init markdown.toPdfWith($md, $o);
```

Block panels + un-encodable text. `codeFill` / `codeBorder` (each a `Fill` , off by default) draw a background and / or border behind every code block, and `quoteFill` / `quoteRule` do the same behind a blockquote (`quoteRule` is the vertical bar down its left edge) - the way `tableHeaderFill` shades a table header. A long code line or an unbreakable table token is folded to the column instead of running off the page. A character the standard-14 fonts cannot encode is replaced by `unencodable` (default `"?"` , `""` drops it) so one stray glyph never fails the whole document.

More document metadata. Alongside `title` / `author` / `subject` / `keywords` , set `creator` (the application that produced the document) and `producer` on `PdfOptions` ; each is written to the PDF Info dictionary when non-empty (an empty `producer` keeps `pdf` 's own `"Jennifer pdf"` default).

Page breaks. A level-1 heading starts a fresh page, but a document can force a break anywhere with `markdown.pageBreak()` - a node placed in a tree handed to `markdown.renderPdf` , or, in Markdown source, a lone `<!-- pagebreak -->` comment. `toHtml` / `toAnsi` ignore it.

```
def o as markdown.PdfOptions init markdown.pdfDefaults();
$o.codeFill = markdown.gray(240);
$o.codeBorder = markdown.gray(150);
$o.quoteFill = markdown.gray(244);
$o.quoteRule = markdown.gray(150);
$o.creator = "Grimoire";
def out as bytes init markdown.toPdfWith($md, $o);
```

Running headers / footers (page numbers). `toPdf` / `renderPdf` return `bytes` , but `pdf` 's running header / footer take a `pdf.Document` . Use `markdown.renderPdfDoc` to get the laid-out document, attach the footer with `pdf.setFooter` (its `%page%` / `%pages%` placeholders fill in per page, which needs the total - known only once the whole book is laid out), then serialize with `pdf.render` :

```
import "markdown.j" as markdown;
import "pdf.j" as pdf;
def doc as pdf.Document init markdown.renderPdfDoc(markdown.parse($md),
markdown.pdfDefaults());
def lbl as pdf.PageLabel init pdf.pageLabel();
$lbl.right = "%page%/%pages%";
def out as bytes init pdf.render(pdf.setFooter($doc, $lbl));
```

See also

- html.md - the HTML backend `toHtml` renders through.
- ansi.md - the terminal styling `toAnsi` renders through.
- modules/index.md - the module catalog and import rules.

mcp - Model Context Protocol server and client

Import with `import "mcp.j" as mcp;` . A Model Context Protocol implementation: a **server** that exposes tools / resources / prompts to an LLM host, and an HTTP **client** that calls a remote MCP server. MCP is JSON-RPC 2.0 on the wire, so the client is a thin layer over `jsonrpc` (which is over `http` + `json`) and the server is a purpose-built router. Because it uses `jsonrpc` / `http` , this module needs the default `jennifer` binary.

This module targets the **stateless** protocol only - no SSE, no sessions. The stdio server (the primary transport), the HTTP client, and the stdio subprocess client are all implemented (see Scope).

```
import "mcp.j" as mcp;
use json;

func echo(args as json.Value) {
    return json.asString($args, "/text");
}

def sch as json.Value init mcp.property(mcp.schema(), "text", "string", "text to
echo", true);
def srv as mcp.Server init mcp.addTool(mcp.server("demo", "1.0.0"), "echo", "echo
text", $sch, echo);

# answer one request (transport-agnostic):
def reply as string init mcp.handle($srv, "{\"jsonrpc\":\"2.
0\",\"method\":\"ping\",\"id\":1}");
```

Runnable: `examples/modules/mcp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/mcp_demo.j).

Building a server

A `Server` is an identity plus registries of tools, resources, and prompts. Build it value-semantically: each builder returns a fresh `Server` .

```
def struct mcp.Server {
    name as string, version as string,
    tools as list of mcp.Tool,
    resources as list of mcp.Resource,
    prompts as list of mcp.Prompt
};
```

Call	Returns	
mcp.server(name, version)	Server	an empty server with the given identity
mcp.addTool(server, name, description, inputSchema, handler)	Server	register a tool; inputSchema is a json.Value JSON Schema, handler a func value
mcp.addResource(server, uri, name, description, mimeType, handler)	Server	register a resource keyed by uri
mcp.addPrompt(server, name, description, arguments, handler)	Server	register a prompt template; arguments is a list of PromptArg (declared to the host, may be [])
mcp.promptArg(name, description, required)	PromptArg	one prompt-argument declaration for addPrompt

Each item's `handler` is a **func value** - a top-level method in the program that imported the module, passed by its bare name and called through its home interpreter, so it resolves its own imports and builds its `json.Value` result in the entry program's context (the same mechanism the `web` module uses):

- a **tool** handler is `func NAME(arguments as json.Value)` and returns a `json.Value` **or a scalar** (a string is used as the text content verbatim; anything else is JSON-encoded into the text content);
- a **resource** handler is `func NAME(uri as json.Value)` and returns the resource's text;
- a **prompt** handler is `func NAME(arguments as json.Value)` and returns the prompt messages as a `json.Value` array. Each message is `{role, content}` where **content is a content block** `{"type": "text", "text": ...}` - not a bare string (MCP requires the block form, and a host rejects a plain string).

A prompt's `arguments` (built with `promptArg`) are surfaced by `prompts/list` , so a host can collect them before calling `prompts/get` . Pass `[]` for a prompt that takes none.

Declaring a tool's input schema

A tool declares a JSON-Schema object describing its arguments. Build it with the two schema helpers (a small convenience over the `json` write API):

Call	Returns	
mcp.schema()	json.Value	a {"type":"object","properties":{},"required":[]} skeleton
mcp.property(schema, name, jsonType, description, required)	json.Value	add properties/<name> = {type, description}; append name to required when required is true

```
def sch as json.Value init mcp.property(
    mcp.property(mcp.schema(), "a", "integer", "first addend", true),
    "b", "integer", "second addend", true);
```

Serving

`mcp.handle(server, requestBody)` turns one raw JSON-RPC request into a reply body - it routes the whole stateless MCP surface and leaves the transport to you. A notification (a request with no `id` , or any `notifications/` method) owes no reply and returns `""` .

Call	Returns	
mcp.handle(server, requestBody)	string	the reply JSON, or "" when none is owed
mcp.serveStdio(server)		run the stdio transport: read newline-delimited JSON-RPC from the program's own stdin, dispatch each through handle, write each non-empty reply to stdout, until EOF

`serveStdio` is the primary MCP transport - an LLM host launches the program and speaks JSON-RPC over its stdin/stdout:

```
import "mcp.j" as mcp;
# ... build $srv ...
mcp.serveStdio($srv);   # blocks, reading stdin until EOF
```

For HTTP, wire `handle` to `httpd` yourself:

```
def req as httpd.Request init httpd.accept($hsrv);
def body as string init convert.stringFromBytes(httpd.body($req), "utf-8");
httpd.respond($req, 200, mcp.handle($srv, $body));
```

Routed methods

Method	Result
initialize	{protocolVersion, capabilities: {tools, resources, prompts}, serverInfo: {name, version}}
ping	{}
notifications/*	no reply ("")

tools/list	{tools: [{name, description, inputSchema}]}
tools/call	{content: [{type: "text", text}], isError} (params {name, arguments})
resources/list	{resources: [{uri, name, description, mimeType}]}
resources/read	{contents: [{uri, mimeType, text}]} (params {uri})
prompts/list	{prompts: [{name, description, arguments?}]} (arguments present when the prompt declares any)
prompts/get	{description, messages} (params {name, arguments})

The protocol version reported is `2025-06-18` . An unknown protocol method is a `-32601` (method not found) reply; an unparseable body is `-32700` (parse error); an unknown resource / prompt is `-32602` (invalid params).

> **Security - dispatch is an allow-list.** Unlike `jsonrpc` 's `handle` , which dispatches a request's `method` to any top-level `func` of that name, `mcp.handle` only ever invokes a handler **registered** in the server:
>
> - A `tools/call` looks the tool up by name in the registry; an unregistered name is a tool-result error (`isError: true`), never a dispatch to an arbitrary top-level method. The same holds for resources (by `uri`) and prompts (by `name`).
> - A tool handler that **throws** yields a tool-result error with a generic message; the thrown message stays server-side and is **not** put on the wire, so you can raise detailed errors without leaking internals.
> - Authenticate at the transport (check a header / token before calling `handle`); the module does no authentication of its own.

Client (HTTP)

`mcp.connect` binds a client to a remote MCP endpoint over HTTP (it wraps a `jsonrpc.Client`). Every call POSTs a JSON-RPC request; any failure

- a JSON-RPC error reply, a transport error, or a malformed reply - throws a

catchable `Error{kind: "jsonrpc"}` .

```
def struct mcp.Client { rpc as jsonrpc.Client };
```

Call	Returns	
mcp.connect(endpoint)	Client	bind to an MCP endpoint URL (HTTP)
mcp.connectWith(endpoint, headers)	Client	HTTP, with extra request headers (auth, ...)
mcp.connectStdio(argv)	Client	launch an MCP server subprocess and talk to it over stdio
mcp.initialize(client)	json.Value	run the initialize handshake (and send the required notifications/initialized on success); returns the server's result
mcp.listTools(client)	json.Value	the tools array
mcp.callTool(client, name, arguments)	json.Value	call a tool; returns the whole result (check isError)
mcp.listResources(client)	json.Value	the resources array
mcp.readResource(client, uri)	json.Value	read a resource; returns the whole resources/read result
mcp.listPrompts(client)	json.Value	the prompts array
mcp.getPrompt(client, name, arguments)	json.Value	get a prompt; returns the whole prompts/get result

```
def c as mcp.Client init mcp.connect("http://127.0.0.1:8080/");
def info as json.Value init mcp.initialize($c);

def args as json.Value init json.set(json.map(), "/text", "hello");
def out as json.Value init mcp.callTool($c, "echo", $args);
io.printf("isError=%t text=%s\n",
    json.asBool($out, "/isError"), json.asString($out, "/content/0/text"));
```

Scope

- **Stateless protocol only.** No SSE, no session lifecycle. `handle` treats each request independently; `notifications/initialized` (and any other `notifications/` message) is accepted and answered with no reply.
- **Transports.** The **stdio server** (`serveStdio`) and a **transport-agnostic handle** (wire to `httpd` / `net`) are implemented on the server side; the **HTTP client** (`connect` + the call verbs, over `jsonrpc`) and the **stdio subprocess client** (`connectStdio` , over `os.run`) are implemented on the client side.
- **The stdio client is one-shot.** Because `os.run` has no persistent child pipe, `connectStdio` relaunches the server per call: it writes the `initialize` handshake plus the one operation as newline-delimited JSON on the child's stdin, closes it, and reads the replies from stdout (correlating by `id`). A stateless server therefore sees a complete short session each call; a persistent, interleaved session (many calls over one launched process) would need the streaming-pipe primitive noted in `os` as a follow-on. `connectStdio` needs the default binary (`os.run`).
- **Handler result shape.** A tool handler returns a `json.Value` or a scalar; a raw Jennifer `list` / `map` / struct is not coerced - build a `json.Value` result explicitly. Structured (non-text) tool content and resource subscriptions are later adds.

See also

- jsonrpc.md - the JSON-RPC 2.0 client / server MCP is built on.
- json.md - the value type params and results are built from and read with.
- httpd.md / http.md - the server engine and client transport.
- web.md - the web framework, the other `func` -value-dispatched request/response module.
- modules/index.md - the module catalog and import rules.

memcache - a memcached client

Import with `import "memcache.j" as memcache;` . A client for a **memcached** server, speaking its classic **text protocol** over the `net` system library. Store with an expiration (`set` / `add`), read (`get`), remove (`delete`), count atomically (`incr` / `decr`), and re-arm a key's expiry (`touch`). memcached is a **volatile cache** - keys expire on their `exptime` and the server evicts under memory pressure - so it suits sessions, rate limits, and derived data, not a system of record. Because it uses `net` , this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "memcache.j" as memcache;

def mc as memcache.Session init memcache.connect(memcache.Options{
    host: "127.0.0.1", port: 11211});
memcache.set($mc, "greeting", "hello", 60);       # 60-second TTL
io.printf("%s\n", memcache.get($mc, "greeting")); # hello
memcache.quit($mc);
```

Runnable: `examples/modules/memcache_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/memcache_demo.j).

Surface

A session is stateful: `connect` , issue commands, `quit` . Every store carries an `exptime` in **seconds** (`0` = never expire, until evicted).

Call / type	Notes
memcache.Options	host, port (plaintext; the text protocol has no auth / TLS).
memcache.Session	A live session over one connection (from connect).
memcache.connect(opts)	Open a session.
memcache.set(session, key, value, exptime)	Store value (replacing any existing), TTL exptime seconds.
memcache.setBytes(session, key, value, exptime)	Store a raw bytes value byte-for-byte (binary counterpart to set).
memcache.add(session, key, value, exptime)	Store only if the key is absent; returns whether it stored.
memcache.get(session, key)	The string value, or "" when the key is absent / expired.
memcache.getBytes(session, key)	The raw bytes value (byte-exact; empty when absent) - for a binary value.
memcache.getMulti(session, keys)	Fetch several keys in one round-trip -> map of string to string (missing keys absent).
memcache.gets(session, key)	Read with a CAS token -> memcache.Item (value, cas, found), for a check-and-set.
memcache.cas(session, key, value, exptime, casId)	Store only if the CAS token still matches -> "stored" / "exists" / "not_found".
memcache.delete(session, key)	Remove the key; returns whether it existed.
memcache.incr(session, key, delta)	Atomically add delta; the new value, or -1 if the key is absent.
memcache.decr(session, key, delta)	Atomically subtract delta (not below 0); -1 if absent.
memcache.touch(session, key, exptime)	Re-arm the key's expiry; returns whether it existed.

memcache.quit(session)	End the session and close.

add, incr, and the primitives caches are built from

`add` stores **only if the key does not already exist** and reports which happened - the atomic building block for a lock ("did I win the key?") or a create-if-new. `incr` / `decr` are atomic server-side counters; memcached will not create a missing counter, so `incr` on an absent key returns `-1` and the caller decides whether to `add` an initial value:

```
def n as int init memcache.incr($mc, "hits", 1);
if ($n == -1) {                                  # first hit this window
    memcache.add($mc, "hits", "1", 60);
    $n = 1;
}
```

That `incr` -then- `add` shape is exactly what the planned `ratelimit` module builds on, and `add` + a TTL is what the planned `session` module uses to mint a session; both are small modules designed to sit on top of this client.

Errors

A protocol error reply (`ERROR` / `CLIENT_ERROR` / `SERVER_ERROR`) throws a catchable `Error` (kind `"memcache"`); `set` also throws if the server does not answer `STORED` . A network failure surfaces as the underlying `net` error.

Binary values and check-and-set

`get` / `set` speak **text** : a value is decoded as UTF-8, which is exact for JSON, numbers, and identifiers, and **throws** on a non-UTF-8 value (strict). For an arbitrary **binary** value, use `setBytes` (a `bytes` value stored byte-for-byte) and `getBytes` (`bytes` , never decoded); `getBytes` returns empty `bytes` for a missing key.

`gets` + `cas` are the optimistic-concurrency primitive: `gets` returns the value with a **CAS token** , and `cas` stores only if that token still matches (nobody else changed the value meanwhile). The loop is `gets` -> compute -> `cas` , retry on `"exists"` :

```
def it as memcache.Item init memcache.gets($mc, "counter");
def next as string init convert.toString(convert.toInt($it.value) + 1);
def r as string init memcache.cas($mc, "counter", $next, 0, $it.cas);
# r is "stored" (done), "exists" (someone else won - re-gets and retry), or
# "not_found" (the key expired).
```

`getMulti(session, keys)` fetches several keys in one round-trip as a `map of string to string` , with missing keys simply absent (test with `maps.has`).

Out of scope

- **A working subset** , not the full command set: `append` / `prepend` and `stats` are reachable later; the basics plus `gets` / `cas` and multi-key `get` cover caches, sessions, counters, and locks.
- **No binary protocol and no SASL auth.** Classic text protocol only.
- **No connection pool.** One `Session` is one connection.

Timeouts

Every read carries an idle timeout (default 30 s) so a hung server fails with a catchable error instead of blocking the caller forever. `connect` sets `Session.timeout` (milliseconds); lower it for a tighter bound, or set it to `0` to disable:

```
def s as memcache.Session init memcache.connect($opts);
$s.timeout = 5000;   # fail a read that stalls for 5 s
```

See also

- net.md - the transport `memcache` builds on.
- modules/index.md - the module catalog and import rules.

mikrotik - RouterOS API client

Import with `import "mikrotik.j" as mikrotik;` . Connect to a MikroTik RouterOS device over its **binary API** (not SSH) and run commands. The API is plain TCP (8728) or api-ssl (8729 over TLS); its wire protocol is sentence-based - a sentence is a run of length-prefixed words ending in a zero-length word. Built on `net` (+ TLS), with an MD5 fallback via `hash` . Needs the default `jennifer` binary. A `!trap` / `!fatal` reply throws `Error{kind: "mikrotik"}` .

> **Higher-level client.** This module is the thin protocol layer. For a thick-layer RouterOS client (typed resources / higher-level operations built on top of it), see the **routeros deck** at github.com/jennifer-language/deck-routeros (https://github.com/jennifer-language/deck-routeros)
>
> - a vendored `jvc` deck.

```
import "mikrotik.j" as mikrotik;

def s as mikrotik.Session init mikrotik.connect(mikrotik.options("192.168.88.1",
"admin", "secret"));
def ifaces as list of map of string to string init mikrotik.print($s, "/
interface");
def id as string init mikrotik.run($s, "/ip/address/add", {});   # (with attrs)
mikrotik.close($s);
```

Runnable: `examples/modules/mikrotik_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/mikrotik_demo.j).

Why the API, not SSH

A real SSH client needs key exchange, host-key verification, and cipher / MAC negotiation - the whole crypto surface plus a heavy dependency, against the dependency-free, TinyGo-clean stance. The RouterOS API is the purpose-built, crypto-optional door: plaintext auth over plain TCP, or confidentiality via api-ssl (TLS), exactly like the mail clients.

Connecting

Login is **plaintext** (`=name=` / `=password=` , RouterOS 6.43+ and all v7); for pre-6.43 routers the client automatically falls back to the MD5 challenge-response (which only needs `hash.compute(b, "md5")`).

```
def struct mikrotik.Options { host as string, port as int, user as string,
password as string, tls as bool };
def struct mikrotik.Session { socket as net.Conn };
```

Call	Returns	
mikrotik.options(host, user, password)	Options	plain TCP, port 8728
mikrotik.optionsTLS(host, user, password)	Options	api-ssl (TLS), port 8729
mikrotik.withPort(o, port)	Options	copy with a different port
mikrotik.connect(opts)	Session	connect and log in
mikrotik.close(s)		close the connection

Commands

A command is a menu path (`/interface/print`); attributes are a `map of string to string` sent as `=key=value` words. Each `!re` reply sentence folds into one row map.

Call	Returns	
mikrotik.talk(s, command, attrs)	list of map of string to string	the general call - the !re reply rows

mikrotik.talkQuery(s, command, attrs, queries)	list of map of string to string	like talk, plus raw ?... query words to filter rows on the router
mikrotik.print(s, path)	list of map of string to string	read sugar for path + "/print"
mikrotik.printWhere(s, path, queries)	list of map of string to string	read sugar: path + "/print" filtered by query words
mikrotik.run(s, command, attrs)	string	for add / set / remove - returns the !done =ret= (e.g. a new item id)

```
# read
for (def iface in mikrotik.print($s, "/interface")) {
    # $iface["name"], $iface["type"], $iface["running"]
}

# add (run returns the new item's id)
def attrs as map of string to string init {};
$attrs["address"] = "10.0.0.1/24";
$attrs["interface"] = "ether1";
def newId as string init mikrotik.run($s, "/ip/address/add", $attrs);

# filtered read: query words run on the router (each starts with "?")
def ethers as list of map of string to string init
    mikrotik.printWhere($s, "/interface", ["?type=ether", "?running=true"]);
# talkQuery is the general form - attributes plus queries (e.g. trim the columns):
def cols as map of string to string init {};
$cols[".proplist"] = "name,type";
def named as list of map of string to string init
    mikrotik.talkQuery($s, "/interface/print", $cols, ["?type=ether"]);
```

Query words filter server-side (the router returns only matching rows), so they are cheaper than fetching everything and filtering in Jennifer. Each is a raw RouterOS query word starting with `?` : `?type=ether` (equals), `?disabled` (property present), `?>mtu=1000` (greater), and the stack operators `?#!` / `?#&` / `?#|` for compound queries. A query word missing its leading `?` throws `Error{kind: "mikrotik"}` .

Tags and streaming

Some RouterOS commands do not run to a single `!done` - they **push !re sentences over time** (`/interface/monitor` , `/ping` , a `.../print` with `follow`). To read one you tag the command so its replies can be correlated, pull each pushed reply in a loop, and issue `/cancel` to stop it.

A **tag** is an API `.tag=<id>` word attached to a command sentence; the router echoes it on every reply to that command. `talkTagged` attaches one to an ordinary (one-shot) command; `listen` attaches an auto-generated tag to a streaming command and hands it back.

Call	Returns	
mikrotik.talkTagged(s, command, attrs, tag)	list of map of string to string	like talk, with a .tag=<id> word (pass "" to auto-generate); still one-shot
mikrotik.listen(s, command, params)	string	start a streaming command; returns its tag (does not read a reply)
mikrotik.receiveReply(s, tag)	map of string to string	block for the next pushed reply for tag; {} at stream end
mikrotik.cancel(s, tag)		stop the stream (issues /cancel naming the tag)

The idiom is **listen -> receive-loop -> cancel** . Jennifer has no callbacks, so `receiveReply` is a cooperative

blocking pull; wrap the loop in a `spawn` to run it alongside the rest of the program. `receiveReply` returns each reply's fields as a `=key=value` map, and an **empty map** (`len(...) == 0`) once the stream has ended - the loop's stop signal. Replies bearing a different tag (another stream on the same session) are skipped.

```
def tag as string init mikrotik.listen($s, "/interface/monitor", {"interface":
"ether1"});
def reader as task of null init spawn {
    repeat {
        def reply as map of string to string init mikrotik.receiveReply($s, $tag);
        if (len($reply) == 0) {
            return;   # stream ended (cancelled)
        }
        # $reply["rx-bits-per-second"], $reply["tx-bits-per-second"], ...
    } until (false);
};
# ... let it stream, then stop it:
mikrotik.cancel($s, $tag);
task.wait($reader);
```

`cancel` is fire-and-forget: the router answers the stream with a `!trap` (category `interrupted`) then `!done` , which `receiveReply` absorbs and turns into the `{}` stop signal, so the receive loop exits cleanly. A genuine `!trap` / `!fatal` (a bad command, a dropped connection) still throws `Error{kind: "mikrotik"}` out of `receiveReply` .

Read deadlines

Every bounded read arms a `net.setDeadline` window (`CONNECT_TIMEOUT_MS`) so a blackholed or mid-sentence router fails instead of hanging. That window is **cleared** (`net.setDeadline(conn, 0)`) on every exit path of the read - a `defer` in `readN` runs it on both the normal return and a throw - so a stale deadline never leaks to the next read or write on the socket (which would otherwise inherit it and spuriously time out, a hazard that matters once a long-lived streaming session interleaves reads and writes).

Scope

- **Binary API, v6 and v7.** The v7 REST API (HTTP + JSON) is a different, stateless shape and a possible second backend later; v1 ships the binary API.
- **Tagged + streaming, one connection.** `talkTagged` / `listen` / `receiveReply` / `cancel` add `.tag` -correlated commands and server-push read loops over the single session socket; a synchronous `talk` still runs to its `!done` before the next. Server-side **query words** are supported, via `talkQuery` / `printWhere` .
- **!trap throws.** A command error surfaces as `Error{kind: "mikrotik"}` (the trailing `!done` is consumed first, so the session stays usable); a `!fatal` (connection closing) throws immediately.
- **String values.** Attributes and reply fields are strings, exactly as the API carries them - parse numbers / booleans yourself.

See also

- net.md - the TCP / TLS transport (+ `connectTLS` for api-ssl).
- mqtt.md / amqp.md - the other hand-framed binary protocol clients.
- modules/index.md - the module catalog and import rules.

mime - build and parse MIME messages

Import with `import "mime.j" as mime;` . Builds and parses MIME messages (RFC 5322 headers plus RFC 2045/2046 bodies) - the header-and-boundary structure behind email. Pure Jennifer over `strings` , `convert` , and `encoding` ; it does **no networking** , so it runs on either binary and is the message-structure foundation the `mail` protocol clients (SMTP / POP3 / IMAP) build on.

```
use io;
import "mime.j" as mime;

def msg as mime.Part init mime.text("text/plain", "Hello, café.");
$msg = mime.withHeader($msg, "Subject", "Hi");
io.printf("%s", mime.encode($msg));

def back as mime.Part init mime.parse(mime.encode($msg));
io.printf("%s\n", mime.body($back));      # Hello, café.
```

Runnable: `examples/modules/mime_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/mime_demo.j).

The Part model

A message is a `Part` tree. A part is either a **leaf** (headers plus decoded content) or a **multipart container** (headers plus child `parts` under a `boundary`):

```
export def struct Part {
    headers as list of Header, body as string, encoding as string,
    parts as list of Part, boundary as string, data as bytes
};
export def struct Header { name as string, value as string };
```

Every leaf carries its raw decoded content as `data` (`bytes`), so **binary attachments round-trip intact** . `body` is the text view: `parse` fills it for text parts (`text/...`), decoding from the Content-Type `charset` (UTF-8 by default; `iso-8859-*` / `windows-*` codepages are honoured, with a UTF-8 fallback for a charset the `encoding` library does not know), and leaves it `""` for binary; `data` always holds the exact bytes. `encode` applies the transfer encoding and `parse` removes it, so you always read plain content.

Surface

Call	Returns	Notes
mime.text(contentType, body)	Part	Leaf text part; 7bit if ASCII, else quoted-printable. Adds charset=utf-8.
mime.attachment(filename, contentType, body)	Part	Base64 leaf with a Content-Disposition filename (text body); a non-ASCII name is written RFC 2231 filename*=.
mime.attachmentBytes(filename, contentType, data)	Part	Base64 attachment from raw bytes (images, PDFs, any binary); non-ASCII name as filename*=.
mime.multipart(subtype, boundary, parts)	Part	Container (multipart/subtype) over one boundary.
mime.withHeader(part, name, value)	Part	Copy with a header set (case-insensitive replace, else append).
mime.encode(part)	string	Serialize to a CRLF MIME message with transfer encodings applied.
mime.parse(text)	Part	Parse a message: unfold headers, split multipart, transfer-decode.
mime.headerValue(part, name)	string	Header value (case-insensitive) or "".

mime.body(part)	string	A leaf's decoded text body ("" for a binary part).
mime.data(part)	bytes	A leaf's raw decoded content (any type, incl. binary).
mime.parts(part)	list of Part	A container's direct child parts.
mime.contentType(part)	string	Media type without parameters (e.g. text/plain).
mime.disposition(part)	string	"attachment" / "inline" / "" (Content-Disposition).
mime.filename(part)	string	filename (else Content-Type name), RFC 2231 (filename*=, continued) + RFC 2047 decoded, or "".
mime.isAttachment(part)	bool	Disposition is attachment, or a filename is set.
mime.walk(part)	list of Part	Every leaf part in the subtree, depth-first.
mime.attachments(part)	list of Part	The attachment leaves (see isAttachment).
mime.textBodies(part)	list of Part	The text/... leaves that are not attachments (plain + html).
mime.findParts(part, mediaType)	list of Part	Leaves whose media type equals mediaType (e.g. text/html).
mime.address(name, email)	string	RFC 5322 mailbox: email, or Name <email> (name quoted, or RFC 2047-encoded when non-ASCII).
mime.encodeWord(text)	string	RFC 2047 UTF-8 base64 encoded-word(s), =?UTF-8?B?...?=, folded when long.
mime.decodeWord(value)	string	Decode every encoded-word in a header value back to text (B and Q).

Encoding

`encode` produces a canonical message with **CRLF** line endings and picks the transfer encoding automatically:

- **7bit** - `text` bodies that are pure ASCII pass through unencoded.
- **quoted-printable** - `text` bodies with non-ASCII are QP-encoded (e.g. `café` to `caf=C3=A9`).
- **base64** - `attachment` bodies are base64-encoded and folded at 76 columns.

A multipart container writes its child parts between `--boundary` delimiters and closes with `--boundary--` ; nesting works (a part can itself be a multipart).

```
def parts as list of mime.Part init [];
$parts[] = mime.text("text/plain", "plain");
$parts[] = mime.text("text/html", "<b>rich</b>");
def msg as mime.Part init mime.multipart("alternative", "b0", $parts);
io.printf("%s", mime.encode($msg));
```

The boundary is **yours to pass** (not auto-generated), so output is deterministic and testable; pick a string that cannot appear in the content.

Parsing

`parse` is the inverse: it splits the header block from the body at the first blank line, unfolds continuation lines (a header value wrapped onto an indented next line), reads `Content-Type` / `Content-Transfer-Encoding` , and either splits a multipart body on its boundary (recursively) or transfer-decodes a leaf. `encode` and `parse` round-trip:

```
def back as mime.Part init mime.parse(wire);
for (def part in mime.parts($back)) {
    io.printf("%s: %s\n", mime.contentType($part), mime.body($part));
}
```

Extracting attachments and bodies

A received message is usually a `multipart/mixed` (bodies + attachments) whose first child may itself be a `multipart/alternative` (the `text/plain` and `text/html` views). Rather than walk that tree by hand, use the accessors: they flatten it for you.

```
def msg as mime.Part init mime.parse(raw);   # or imap.fetchMessage(session, n)

# Every readable body (both the plain and html alternatives).
for (def part in mime.textBodies($msg)) {
    io.printf("[%s]\n%s\n", mime.contentType($part), mime.body($part));
}

# Save each attachment. data() is the raw bytes, filename() its name.
use fs;
for (def att in mime.attachments($msg)) {
    fs.write("/downloads/" + mime.filename($att), mime.data($att));
}
```

`walk` returns every leaf if you want to filter yourself; `findParts($msg, "text/html")` picks a specific media type. `data` works on any leaf (text or binary); `body` is the text convenience.

Non-ASCII headers (RFC 2047 encoded-words)

Header values must be ASCII on the wire, so a non-ASCII `Subject` or display name is carried as an RFC 2047 **encoded-word** (`=?UTF-8?B?...?=`). This is applied **automatically** and symmetrically:

- `encode` encodes a non-ASCII `Subject` / `Comments` value, and the display-name half of an address header (`From` / `To` / `Cc` / `Bcc` / `Reply-To` / `Sender`), leaving the `<addr>` untouched. Long values fold into several encoded-words split on rune boundaries.
- `parse` decodes those same headers back to plain text - so a fetched `Subject: =?UTF-8?B?QmVyaWNodCBhdXMgTcO8bmNoZW4=?=` reads back as `Bericht aus München` . A word that fails to decode is left verbatim, so a malformed header never crashes `parse` .
- `mime.address("Jörg Müller", "j@x.de")` encodes the name for you.

```
def m as mime.Part init mime.withHeader(
    mime.text("text/plain", "hi"), "Subject", "Grüße aus München");
io.printf("%s", mime.encode($m));   # Subject: =?UTF-
8?B?R3LDvMOfZSBhdXMgTcO8bmNoZW4=?=
```

The primitives `mime.encodeWord` / `mime.decodeWord` are exposed for the cases the auto-hooks don't cover (a custom header, a multi-address line). Both B (base64) and Q (quoted-printable) encoded-words decode; encoding always emits B. `us-ascii` / `iso-8859-*` / `windows-*` charsets decode through `encoding` ; UTF-8 is the default.

Out of scope

Deliberately a foundation, not a full mail stack:

- **Legacy multibyte charsets.** A text part is decoded to `body` using its Content-Type `charset` for UTF-8 and every single-byte codepage the `encoding` library ships (`us-ascii` / `iso-8859-*` / `windows-*`). A multibyte legacy charset (`shift_jis` , `gb2312` , `big5` , ...) has no `encoding` codec, so it falls back to a UTF-8 read of `body` ; the exact bytes always stay reachable via `data` .
- **Multi-address name encoding.** A comma-separated address list is left raw on `encode` ; encode each mailbox's name with `mime.address` when building it.
- **Networking.** This module only shapes messages; sending / fetching them is the `mail` SMTP / POP3 / IMAP clients (built on `mime`).

See also

- encoding.md - `toText` / `fromText` (base64, quoted-printable), the transfer codecs `mime` delegates to.
- strings.md - the text operations the header and boundary handling build on.
- modules/index.md - the module catalog and import rules.

mqtt - an MQTT 3.1.1 pub/sub client

Import with `import "mqtt.j" as mqtt;` . An **MQTT 3.1.1** publish/subscribe client over the `net` system library - the same "protocol clients are modules, `net` is the transport" line the other network clients follow. MQTT packets are a 1-byte fixed header, a variable remaining-length integer, then a length-prefixed payload; the module builds and parses them with Jennifer's bitwise operators (`& | ^ ~ << >>`) and `bytes` . It does QoS 0 and QoS 1 publish/subscribe (a synchronous PUBACK handshake), retained messages, a Last-Will registered in the CONNECT, and `reconnect` for session resumption. Because it uses `net` , this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "mqtt.j" as mqtt;

def c as mqtt.Client init mqtt.connect(mqtt.Options{host: "127.0.0.1",
    port: 1883, clientId: "demo", keepalive: 30, security: "none",
    username: "", password: ""});
$c = mqtt.subscribe($c, "sensors/temp");
mqtt.publish($c, "sensors/temp", "21.5");
def m as mqtt.Message init mqtt.receive($c);
io.printf("%s -> %s\n", $m.topic, convert.stringFromBytes($m.payload, "utf-8"));
mqtt.disconnect($c);
```

`subscribe` / `subscribeQos1` return an updated `Client` (they record the subscription so `reconnect` can restore it), so reassign the result: `$c = mqtt.subscribe($c, ...)` . A `Client` stays value-semantic - copies share the socket handle but each carries its own settings and subscription list.

Runnable: `examples/modules/mqtt_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/mqtt_demo.j).

Surface

A client is stateful: `connect` , subscribe / publish / receive, `disconnect` .

Call / type	Notes
mqtt.Options	host, port, clientId, keepalive (seconds), security, username, password.
mqtt.Will	A Last-Will: topic ("" disables it), payload (bytes), qos, retain.
mqtt.Subscription	A tracked subscription: topic, qos (the granted QoS).
mqtt.Client	A live connection (from connect / connectWith), carrying the settings and tracked subscriptions reconnect needs.
mqtt.Message	A received message: topic (string), payload (bytes).
mqtt.connect(opts)	Open a connection (clean session, no will), send CONNECT, check the CONNACK.
mqtt.connectWith(opts, will, cleanSession)	Connect with an explicit Last-Will and clean-session flag.
mqtt.reconnect(client)	Re-dial, re-CONNECT with the stored settings, re-subscribe the tracked routes; returns a fresh Client.
mqtt.subscribe(client, topic)	Subscribe at QoS 0, wait for the SUBACK, track it; returns an updated Client.
mqtt.subscribeQos1(client, topic)	Subscribe requesting QoS 1 (parses the granted QoS), track it; returns an updated Client.

mqtt.publish(client, topic, message)	Publish a UTF-8 text message at QoS 0 (fire and forget).
mqtt.publishBytes(client, topic, payload)	Publish a raw bytes payload at QoS 0.
mqtt.publishRetain(client, topic, message, retain)	Publish text at QoS 0 with an explicit retain flag.
mqtt.publishBytesRetain(client, topic, payload, retain)	Publish bytes at QoS 0 with a retain flag.
mqtt.publishQos1(client, topic, payload, retain)	Publish bytes at QoS 1 and block for the matching PUBACK.
mqtt.receive(client)	Block until the next application message arrives; returns a Message (PUBACKs a QoS-1 PUBLISH first).
mqtt.poll(client, timeoutMs)	Poll up to timeoutMs ms; returns a list of Message of length 0 or 1 (PUBACKs a QoS-1 PUBLISH first).
mqtt.ping(client)	Send a PINGREQ keepalive (fire and forget).
mqtt.disconnect(client)	Send DISCONNECT and close.

`Options.security` is a `transport.Security` : `.None` (plaintext, port 1883) or `.Tls` (implicit TLS, `mqtts` , port 8883); `.Starttls` is rejected (MQTT has no in-band upgrade). Needs `import "transport.j" as transport;` . `username` / `password ""` omit the CONNECT credentials. A non-empty `clientId` identifies the session to the broker.

QoS 1: the PUBACK handshake

`publishQos1(client, topic, payload, retain)` sends a PUBLISH with QoS 1 and a packet identifier, then blocks until the broker returns the matching **PUBACK** (at-least-once delivery). If no PUBACK arrives within the wait window the message is re-sent with the **DUP** (duplicate-delivery) flag set and the same packet id, up to a fixed number of attempts; if the budget is exhausted it throws an `Error` (kind `"mqtt"`). Because the call blocks for the PUBACK, only one message is ever in flight, so a fixed non-zero packet id is safe and the broker de-duplicates a re-sent PUBLISH by that id.

On the receive side, `subscribeQos1(client, topic)` asks the broker for QoS 1 and records the QoS it **granted** (parsed from the SUBACK - a broker may downgrade to QoS 0, or reject with `0x80`). When a QoS-1 PUBLISH then arrives, `receive` / `poll` send its PUBACK **before** returning the `Message` , so the broker considers it delivered. QoS 0 is unchanged: `publish` / `publishBytes` are fire-and-forget, and a QoS-0 PUBLISH is returned without any acknowledgement.

```
$c = mqtt.subscribeQos1($c, "commands/#");
mqtt.publishQos1($c, "commands/reboot",
    convert.bytesFromString("now", "utf-8"), false); # blocks for the PUBACK
def m as mqtt.Message init mqtt.receive($c);         # PUBACKs it, then returns
```

Retained messages and the Last-Will

Any publish takes a **retain** flag: `publishRetain` / `publishBytesRetain` at QoS 0 and `publishQos1` at QoS 1. A retained message is stored by the broker and delivered to every future subscriber of the topic at subscribe time; publishing an **empty** payload with `retain: true` clears the retained message.

A **Last-Will** is a message the broker publishes on the client's behalf if the connection drops without a clean `disconnect` . Register it in the CONNECT with `connectWith(opts, will, cleanSession)` ; a `Will` with an empty `topic` sets no will (what plain `connect` does).

```
def will as mqtt.Will init mqtt.Will{topic: "clients/demo/status",
    payload: convert.bytesFromString("offline", "utf-8"), qos: 1, retain: true};
def c as mqtt.Client init mqtt.connectWith($opts, $will, false); # persistent
session
mqtt.publishRetain($c, "clients/demo/status", "online", true);   # clear on clean
exit
```

Reconnect and session resumption

`reconnect(client)` re-dials the broker, re-sends CONNECT with the client's stored `opts` / `will` / `cleanSession` , and re-subscribes every subscription the client tracked (via `subscribe` / `subscribeQos1`); it returns a **fresh Client** the caller reassigns (`$c = mqtt.reconnect($c);`), best-effort closing the stale socket.

Session resumption is the broker's job, gated by the clean-session flag. Connect with `connectWith(opts, will, false)` (clean-session **false**) and the broker keeps the session keyed by `clientId` across a drop - the retained subscriptions and any queued QoS-1 messages - and reports it in the CONNACK's session-present bit. The re-subscribe `reconnect` performs is idempotent in that case, and also covers a clean-session (`true`) client, where the broker discarded the session and the routes must be re-established from scratch. Reconnect timing (when to give up and retry, backoff) is the caller's loop, the same division of labor as keepalive.

Single-threaded poll with timeout

Jennifer has no handler callbacks, so a subscriber drives its own loop. `poll` arms a read deadline (via `net.setDeadline`) so one flow can wait for a message and, when idle, do other work - send a keepalive, check a clock - without dedicating a `spawn` ed reader. It returns a list of zero or one message: empty when nothing arrived in the window, one `Message` when a PUBLISH was received. Non-PUBLISH control packets (a PINGRESP) are consumed and reported as an empty poll.

```
def running as bool init true;
def ticks as int init 0;
while ($running) {
    def msgs as list of mqtt.Message init mqtt.poll($c, 1000);
    if (len($msgs) > 0) {
        def m as mqtt.Message init $msgs[0];
        io.printf("%s -> %s\n", $m.topic,
            convert.stringFromBytes($m.payload, "utf-8"));
    } else {
        $ticks = $ticks + 1;
        if ($ticks == 20) {    # ~20s idle
            mqtt.ping($c);     # keepalive; the PINGRESP is consumed by poll
            $ticks = 0;
        }
    }
}
```

`receive` is the blocking counterpart: it waits for the next PUBLISH with no timeout, skipping any control packets in between.

Keepalive is the caller's job (call `ping` on your own cadence): the module holds no mutable timing state - a `Client` is value-semantic, sharing only the underlying socket handle across copies.

Errors

`connect` / `connectWith` throw a catchable `Error` (kind `"mqtt"`) when the broker refuses the connection (a non-zero CONNACK code) or does not answer with a CONNACK; `subscribe` / `subscribeQos1` throw when the SUBACK reports failure (`0x80`); `publishQos1` throws when no PUBACK arrives within the retry budget. A connection that closes mid-packet throws `mqtt: connection closed mid-packet` . A `poll` whose deadline elapses is **not** an error - it simply returns an empty list.

Testing

The pure packet logic - the remaining-length varint encode / decode, the length-prefixed string framing, the CONNECT builder (including the Last-Will and clean-session flags), the QoS-1 PUBLISH builder (packet id + DUP / retain flags), the PUBACK build / parse, the packet-id extraction, and the SUBACK granted-QoS parse - is

unit-tested in the overlay (`modules/mqtt_test.j`). The networked connect / subscribe / publish / receive / poll round-trip (QoS 0 and QoS 1, retained, will, reconnect) is covered end to end by an in-process MQTT-broker fake in the Go test suite (`TestMqttPubSub`), so it runs in CI without a broker install.

Out of scope

MQTT 3.1.1 with QoS 0 and QoS 1. Deferred until a workload needs them:

- **QoS 2** exactly-once (PUBREC / PUBREL / PUBCOMP with persistent packet-id state).
- **Multiple in-flight QoS-1 messages.** `publishQos1` is synchronous (one message in flight, a fixed packet id); a pipelined sender with a packet-id window is a larger design.
- **MQTT 5 properties.**

If full QoS with high-throughput processing ever makes the tree-walker the bottleneck, a Go-backed engine (build-tag split like `net`) is the fallback - but the pub/sub basics belong in a module.

Timeouts and limits

The initial connect is bounded by a connection-establishment timeout, so a slow or unreachable broker fails the dial instead of blocking it forever, and the CONNECT and SUBSCRIBE handshakes carry a 30 s timeout, so a broker that accepts the connection but never acknowledges fails instead of hanging. `publishQos1` waits a bounded window for each PUBACK and retries a fixed number of times before throwing. `poll(client, ms)` already bounds how long it waits for a message; `receive` blocks until one arrives. A single control packet is capped at **64 MiB** : the remaining-length varint is attacker-declarable, so a broker declaring an enormous packet fails with a catchable error rather than an unbounded allocation.

See also

- net.md - the transport `mqtt` builds on, including `net.setDeadline` for the poll loop.
- idna.md - the other module doing bit-level `bytes` work (Punycode).
- modules/index.md - the module catalog and import rules.

multipart - multipart/form-data build and parse

Import with `import "multipart.j" as multipart;` . Build and parse `multipart/form-data` bodies (RFC 7578) - the file-upload counterpart to `mime` 's email multipart. `build` turns a list of `Part` s (form fields and files) into a `(contentType, body)` pair ready to POST; `parse` turns a `Content-Type` header and a body back into parts. Bodies are `bytes` , so binary file content round-trips intact. Pure `.j` over `strings` + `bytes` ; runs on both binaries.

```
import "multipart.j" as multipart;
use convert;

def parts as list of multipart.Part init [
    multipart.field("title", "hello"),
    multipart.file("doc", "a.txt", "text/plain", convert.bytesFromString("hi",
"utf-8"))
];
def form as multipart.Built init multipart.build($parts);
# POST $form.body with header Content-Type: $form.contentType
def back as list of multipart.Part init multipart.parse($form.contentType, $form.
body);
```

Runnable: `examples/modules/multipart_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/multipart_demo.j).

Parts

```
def struct multipart.Part {
    name as string,         # the field name
    filename as string,     # the file name ("" for a plain field)
    contentType as string,  # the part Content-Type ("" for a plain field)
    data as bytes           # the part body
};
```

Call	Returns	
multipart.field(name, value)	Part	a plain text field
multipart.file(name, filename, contentType, data)	Part	a file part (body is bytes)
multipart.text(p)	string	decode a part's body as UTF-8 (for field values)
multipart.isFile(p)	bool	whether the part carries a filename

Building

Call	Returns	
multipart.build(parts)	Built	encode with a fresh random boundary
multipart.buildWith(parts, boundary)	Built	encode with an explicit boundary (deterministic)

`Built{ contentType, body }` pairs the ready-to-send `Content-Type` header (carrying the boundary) with the encoded `body` . Use `build` normally; `buildWith` when you need a fixed boundary (tests, reproducibility). The boundary must not occur inside any part body - the random one from `build` makes that effectively impossible.

Parsing

`multipart.parse(contentType, body)` reads the boundary from the `Content-Type` header and splits

the body into `Part` s. It matches the delimiter only at `CRLF--boundary` (normalising a leading boundary), so the boundary token appearing *inside* a file body does not split it. A missing boundary or a part without a header terminator throws `Error{kind: "multipart"}` .

```
def back as list of multipart.Part init multipart.parse($contentType, $body);
for (def p in $back) {
    if (multipart.isFile($p)) {
        # $p.filename, $p.contentType, $p.data
    } else {
        # multipart.text($p) is the field value
    }
}
```

Scope

- **form-data only.** The generic `multipart/*` (mixed, alternative, related) used in email is `mime` 's job; this is the browser/HTTP upload shape.
- **Content-Disposition / Content-Type headers only.** Other per-part headers and RFC 2231 extended (`name*=`) parameter encoding are not parsed; `name` / `filename` are read from the quoted `Content-Disposition` params.
- **No streaming.** The whole body is built / parsed in memory - fine for form posts, not for multi-gigabyte uploads.
- **Names / filenames are emitted verbatim** inside quotes; avoid `"` and CRLF in them (RFC 7578 percent-encoding is a follow-on).

See also

- mime.md - email-style multipart and header encoding.
- http.md / web.md - send an upload, or handle one.
- modules/index.md - the module catalog and import rules.

ntp - SNTP network-time client

Import with `import "ntp.j" as ntp;` . Query a time server over UDP and get back its time plus the local clock **offset** and round-trip **delay** . This is the one-shot query half of NTP - a simple **SNTP** client (RFC 4330 / 5905): it speaks the standard NTP wire protocol (the 48-byte packet on port 123) but does not discipline the clock or run as a daemon. It reports the measurement; acting on it is the caller's job. Needs the default `jennifer` binary (`net`).

```
import "ntp.j" as ntp;
use io; use time;

def r as ntp.Result init ntp.query("pool.ntp.org");
io.printf("server time: %s  offset: %d ms\n", time.iso($r.serverTime), time.
milliseconds($r.offset));
```

Runnable: `examples/modules/ntp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/ntp_demo.j).

Result

```
def struct ntp.Result {
    serverTime as time.Time,     # the server's transmit time
    offset as time.Duration,     # local clock offset (server minus local)
    delay as time.Duration       # measured round-trip delay
};
```

`offset` is computed from the four SNTP timestamps as `((T2 - T1) + (T3 - T4)) / 2` and `delay` as `(T4 - T1) - (T3 - T2)` , where T1/T4 are the local send/receive instants and T2/T3 are the server's receive/transmit timestamps. A positive `offset` means the local clock is behind the server.

Query

Call	Returns	
ntp.query(host)	Result	query host:123 with a 5-second timeout
ntp.queryWith(address, timeoutMs)	Result	query a full host:port with a custom timeout

`query` is the common form (`ntp.query("time.example.com")`); `queryWith` takes a full `host:port` address and an explicit receive timeout in milliseconds. A query that gets no reply within the timeout, or a short / malformed packet, throws `Error{kind: "ntp"}` - catch it with `try` / `catch` . The receive timeout is a real deadline on the UDP socket, so a lost reply fails fast instead of hanging.

Scope

- **Query, not discipline.** This measures the offset and hands it back; it does not step or slew the system clock (that is the OS / a daemon's job), and it does no server selection, filtering, or Kiss-o'-Death handling.
- **One server per call.** Poll several servers and combine the results yourself if you want robustness against a single bad source.
- **No authentication.** Plain SNTP; no symmetric-key or NTS-secured exchange.

See also

- net.md - the UDP surface (`listenUDP` / `sendTo` / `recvFrom` / `setDeadline`) the client is built on.
- time.md - the `time.Time` / `time.Duration` the result is expressed in.
- modules/index.md - the module catalog and import rules.

oauth - a generic OAuth2 client

Import with `import "oauth.j" as oauth;` . The **get-a-token** half of OAuth2 (the *use-a-token* half is `sasl` XOAUTH2). It acquires and refreshes access tokens against any OAuth2 token endpoint - not email-specific, any OAuth2-protected API - over `http` + `json` . Because it builds on `http` (which uses `net`), this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "oauth.j" as oauth;
import "sasl.j" as sasl;

def cfg as oauth.Config init oauth.google("client-id", "client-secret",
    "https://mail.google.com/");
def dev as oauth.DeviceAuth init oauth.deviceStart($cfg);
io.printf("visit %s and enter %s\n", $dev.verificationUri, $dev.userCode);
def tok as oauth.Token init oauth.deviceWait($cfg, $dev);   # blocks until
approved
# use the token, e.g. for IMAP: sasl.bearer("me@gmail.com", $tok.accessToken)
```

Runnable: `examples/modules/oauth_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/oauth_demo.j).

Surface

Call / type	Notes
oauth.Config	tokenUrl, deviceUrl, clientId, clientSecret, scope.
oauth.Token	accessToken, tokenType, refreshToken, scope, expiresAt (Unix seconds; 0 = unknown).
oauth.DeviceAuth	deviceCode, userCode, verificationUri, interval, expiresAt.
oauth.clientCredentials(config)	The Client Credentials grant (a service as itself) -> Token.
oauth.refresh(config, refreshToken)	Trade a refresh token for a new Token (keeps the refresh token when the server omits it).
oauth.deviceStart(config)	Begin the Device Authorization grant -> DeviceAuth (show the user the URL + code).
oauth.deviceWait(config, deviceAuth)	Poll until the user approves -> Token (backs off on slow_down).
oauth.isExpired(token)	Whether the token is past its expiry (30s skew buffer).
oauth.google(clientId, clientSecret, scope)	A Config with Google's endpoints.
oauth.microsoft(tenant, clientId, clientSecret, scope)	A Config with a Microsoft 365 / Entra tenant's endpoints.
oauth.save(path, token) / oauth.load(path)	Persist / reload a token as JSON (via fs). save tightens the file to owner-only 0600.

Flows

Three grants ship - the ones that need only `http` + `json` :

- **Client Credentials** - `clientCredentials(config)` : a service authenticates as itself (no user), for machine-to-machine APIs.

- **Refresh Token** - `refresh(config, refreshToken)` : exchange a long-lived refresh token for a fresh access token. The reply often omits the refresh token; the module carries the old one forward so the returned `Token` always has one.
- **Device Authorization Grant** - `deviceStart` then `deviceWait` : the CLI-friendly flow. There is no local redirect server - you show the user a URL and a short code, they approve in a browser, and `deviceWait` polls the token endpoint (honouring the server's `interval` and `slow_down`) until it returns a token.

Expiry and refresh

A `Token` carries `expiresAt` (a Unix timestamp, computed from the response's `expires_in`). `oauth.isExpired($token)` reports whether it is past due (with a 30-second skew buffer), so a caller can refresh proactively:

```
if (oauth.isExpired($tok)) {
    $tok = oauth.refresh($cfg, $tok.refreshToken);
}
```

`oauth.save` / `oauth.load` persist a token to disk (JSON via `fs`) so a long-running or restarted program keeps its refresh token. An access / refresh token is a bearer credential, so `save` tightens the file to owner-only `0600` after writing (via `fs.chmod`), the same file-permission model `gh` / `aws` use for their token stores.

Feeding mail auth

OAuth2's other half is presenting the token. For mail, that is SASL XOAUTH2: pass `token.accessToken` to `sasl.bearer` , and the result drives an SMTP / IMAP `AUTHENTICATE XOAUTH2` . So `oauth` (get the token) + `sasl` (use it)

- the mail clients compose into modern Google / Microsoft 365 mail access. The

provider presets (`google` , `microsoft`) exist to make that the headline case.

Errors

A token-endpoint error (`{"error":"...","error_description":"..."}`) throws a catchable `Error` (kind `"oauth"`) with the code and description. A non-terminal device-poll status (`authorization_pending` , `slow_down`) is handled internally by `deviceWait` , not surfaced.

Out of scope (later, dependency-gated)

- **Authorization Code + PKCE** - needs a local redirect server (`httpd`) to catch the callback and a crypto-grade random source for the PKCE verifier; lands with those.
- **Service-account JWT assertion** (Google) - an RSA-signed client assertion, so it waits on the `crypto` library.

See also

- sasl.md - XOAUTH2, the use-a-token half.
- http.md / json.md - what `oauth` composes over.
- modules/index.md - the module catalog and import rules.

orm - a relational mapper over sql

Import with `import "orm.j" as orm;` . A minimal relational mapper over the `sql` library. It is **Data Mapper, not Active Record** : Jennifer structs are value-semantic and carry no methods, and a module holds no state, so a row cannot `save()` itself. You pass a record and a `Schema` to repository functions - `orm.insert` / `find` / `update` / `delete` , and `orm.all` over a query.

There is no reflection, so you declare the table mapping once as an `orm.Schema` , which also carries the SQL **dialect** (`orm.Dialect.Mysql` or `orm.Dialect.Postgres`) - a backend selector on one module, not parallel modules. The dialect and each column's kind are closed **enums** (`orm.Dialect` , `orm.ColumnKind`), so an invalid value is a compile-time error, not a runtime one.

Every persistence / query-executing function takes an `orm.Session` (from `orm.session(conn)` or `orm.transaction(tx)`) as its first argument, so the same call runs standalone or inside a transaction.

```
import "orm.j" as orm;

def users as orm.Schema init orm.column(orm.column(
    orm.schema("users", "id", orm.Dialect.Postgres), "id", orm.ColumnKind.Int),
"name", orm.ColumnKind.String);

def sess as orm.Session init orm.session($conn);       # auto-committing session
def ada as map of string to string init {};
$ada["id"] = "1";
$ada["name"] = "ada";
orm.insert($sess, $users, $ada);                       # INSERT INTO users ...
def row as map of string to string init orm.find($sess, $users, "1");
```

Runnable: `examples/modules/orm_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/orm_demo.j). Schema migrations live in the separate `sqlmigrate` module (its `up` / `down` are DDL strings built by the DDL helpers below).

Records are map of string to string

Both the input to `insert` / `update` and the result of `find` / `all` are a `map of string to string` keyed by column name. This is the row form that needs no map-to-struct conversion; a typed-struct form waits on that language feature. The database coerces the string values to the column types, and only the columns **present** in the record are written (so an auto-generated key is simply omitted).

The schema

Declared once with `orm.schema` + `orm.column` (value-semantic - each `column` returns a fresh schema):

Call	Returns	Notes
orm.schema(table, primaryKey, dialect)	Schema	Start a schema; dialect is orm.Dialect.Mysql or orm.Dialect.Postgres.
orm.column(schema, name, kind)	Schema	Append a column; kind is orm.ColumnKind.Int / String / Float / Bool / Bytes.
orm.createTable(schema)	string	The CREATE TABLE DDL for the dialect, including each column's attributes and the primary key.

Column attributes

A column is **nullable by default** (SQL's own default). The fluent setters decorate the **most-recently-added** column and return a fresh `Schema` , so they chain onto `orm.column` :

Call	Effect on the last column

orm.notNull(schema)	NOT NULL
orm.unique(schema)	UNIQUE
orm.autoIncrement(schema)	SERIAL (Postgres) / AUTO_INCREMENT (MySQL)
orm.withDefault(schema, value)	DEFAULT value (numeric / bool bare, string quoted+escaped)

A `DEFAULT` value is rendered by the column kind and can never inject DDL: an `Int` / `Float` default is validated numeric; a `Bool` default accepts `true` / `false` / `1` / `0` ; a `String` / `Bytes` default is a quoted, escaped SQL literal (a backslash or control character is rejected).

```
def accounts as orm.Schema init orm.schema("accounts", "id", orm.Dialect.
Postgres);
$accounts = orm.autoIncrement(orm.column($accounts, "id", orm.ColumnKind.Int));
$accounts = orm.notNull(orm.unique(orm.column($accounts, "email", orm.ColumnKind.
String)));
$accounts = orm.withDefault(orm.column($accounts, "active", orm.ColumnKind.Bool),
"true");
# CREATE TABLE accounts (id SERIAL, email TEXT NOT NULL UNIQUE, active BOOLEAN
DEFAULT TRUE, PRIMARY KEY (id))
```

DDL builders

Standalone DDL emitters, each returning one statement string. Feed them to the `sqlmigrate` runner's `up` / `down` , or run them directly. Identifiers are allowlist-checked like the query builder.

Call	Statement
orm.dropTable(table)	DROP TABLE table
orm.addColumn(table, name, kind, dialect)	ALTER TABLE ... ADD COLUMN name type
orm.dropColumn(table, name)	ALTER TABLE ... DROP COLUMN name
orm.renameColumn(table, from, to)	ALTER TABLE ... RENAME COLUMN from TO to
orm.createIndex(name, table, columns, isUnique)	CREATE [UNIQUE] INDEX name ON table (...)
orm.dropIndex(name, table, dialect)	DROP INDEX name [ON table] (dialect-spelled)
orm.addForeignKey(table, name, column, refTable, refColumn)	ALTER TABLE ... ADD CONSTRAINT ... FOREIGN KEY ...
orm.dropForeignKey(table, name, dialect)	ALTER TABLE ... DROP CONSTRAINT/FOREIGN KEY name

The query builder (functional, pure)

Like the `json` write surface, each step returns a fresh `orm.Query` - no method chaining (values have no methods):

Call	Returns	Notes
orm.from(schema)	Query	Select all rows of the schema's table.
orm.select(query, cols)	Query	Project only cols (a list of string) instead of SELECT *.
orm.count(query, alias)	Query	Add COUNT(*) AS alias to the projection.
orm.aggregate(query, fn, col, alias)	Query	Add fn(col) AS alias; fn is COUNT / SUM / AVG / MIN / MAX.
orm.where(query, col, op, value)	Query	A col op value condition, AND-joined; value binds as a parameter.

orm.orWhere(query, col, op, value)	Query	Like where, but OR-joined to the previous condition.
orm.whereIn(query, col, values) / orm.orWhereIn / orm.whereNotIn	Query	col IN (...) / NOT IN, one placeholder bound per value (non-empty).
orm.whereNull(query, col) / orm.whereNotNull(query, col)	Query	col IS NULL / IS NOT NULL (no bound value).
orm.whereBetween(query, col, lo, hi)	Query	col BETWEEN lo AND hi (both bounds bound).
orm.distinct(query)	Query	Render SELECT DISTINCT.
orm.page(query, pageNum, pageSize)	Query	1-based LIMIT / OFFSET sugar (page 1 starts at offset 0).
orm.join / orm.leftJoin / orm.rightJoin (query, table, leftCol, rightCol)	Query	An INNER / LEFT / RIGHT JOIN.
orm.groupBy(query, cols)	Query	Add a GROUP BY.
orm.having(query, fn, col, op, value) / orm.orHaving	Query	A HAVING fn(col) op value condition over an aggregate.
orm.orderBy(query, col, dir)	Query	Add an ORDER BY term ("asc" / "desc").
orm.limit(query, n) / orm.offset(query, n)	Query	Add LIMIT / OFFSET.
orm.toSql(query)	Rendered	Render to parameterized SQL: Rendered{sql, params}.

`toSql` spells placeholders per dialect - `?` for MySQL, `$1` / `$2` … for Postgres - and **values only ever reach the query through those placeholders** , so the injection safety is inherited from `sql` . The whole builder is pure, so it is fully testable without a database.

Note the SQL precedence of `orWhere` : SQL binds `AND` tighter than `OR` , so `where(...)` then `orWhere(...)` reads a `AND` b `OR` c as `(a AND b) OR c` .

```
# projection + aggregate + GROUP BY + HAVING
def q as orm.Query init orm.having(
    orm.groupBy(orm.count(orm.select(orm.from($users), ["age"]), "n"), ["age"]),
    "COUNT", "*", ">", "5");
def r as orm.Rendered init orm.toSql($q);
# r.sql    = SELECT age, COUNT(*) AS n FROM users GROUP BY age HAVING COUNT(*) >
$1
# r.params = ["5"]

# OR + IN
def q2 as orm.Query init orm.whereIn(
    orm.orWhere(orm.where(orm.from($users), "age", ">=", "18"), "name", "=",
"ada"),
    "id", ["1", "2", "3"]);
# SELECT * FROM users WHERE age >= $1 OR name = $2 AND id IN ($3, $4, $5)
```

Render-time validation

Identifiers, operators, aggregate functions, join kinds, and sort directions are validated against fixed allowlists **twice** : once in the builder (an early, friendly error) and again inside `toSql` / `createTable` / the CRUD builders. So even a hand-built `orm.Query{...}` or `orm.Schema{...}` struct literal that skipped the builder is re-checked before it renders - an injected column name throws `Error{kind: "orm"}` at render time rather than reaching the database.

Relations

Declare associations once on the schema; they are metadata (a `list of Relation` on the `Schema`) that `joinRelation` turns into the right `JOIN` . Each builder returns a fresh schema, so they chain like `column` :

Call	Kind	Foreign key is on ...
orm.belongsTo(schema, name, target, foreignKey)	belongs-to	this table (references the target's id)
orm.hasOne(schema, name, target, foreignKey)	has-one	the target table (references this PK)
orm.hasMany(schema, name, target, foreignKey)	has-many	the target table (references this PK)
orm.manyToMany(schema, name, target, joinTable, localFk, targetFk)	many-to-many	a join table linking the two

`orm.joinRelation(query, schema, relationName)` emits the `INNER JOIN` a relation implies (two joins for a many-to-many, through its join table), built over the `join` primitive - so the key columns are always correct and identifier-checked:

```
def authors as orm.Schema init orm.hasMany(
    orm.schema("authors", "id", orm.Dialect.Postgres), "posts", "posts",
"authorId");
orm.toSql(orm.joinRelation(orm.from($authors), $authors, "posts")).sql;
# SELECT * FROM authors INNER JOIN posts ON posts.authorId = authors.id
```

A `belongsTo` and a `manyToMany` reference the target's primary key by the `id` convention.

Eager loading (N+1 elimination)

Mark a declared relation with `orm.with` , then `orm.load` fetches the base rows and their relations in a **fixed 1 + R queries** (R = relations requested): the base query once, then **one** batched `WHERE fk IN (...)` per relation - never one query per row.

Call	Returns	Notes
orm.with(query, relationName)	Query	Mark a relation to eager-load (ignored by toSql; consumed by load).
orm.load(session, schema, query)	Result	Run the base query + one batched query per marked relation.
orm.rows(result)	list of map of string to string	The base rows.
orm.related(result, row, name)	list of map of string to string	A row's children for a has-many / many-to-many (empty list if none).
orm.relatedOne(result, row, name)	map of string to string	A row's single related row for a belongs-to / has-one, or {} if none.

```
def r as orm.Result init orm.load($sess, $authors, orm.with(orm.from($authors),
"posts"));
for (def author in orm.rows($r)) {
    def posts as list of map of string to string init orm.related($r, $author,
"posts");
    # 2 queries total (authors + posts WHERE authorId IN (...)), regardless of
author count
}
```

`related` / `relatedOne` are pure lookups into the `Result` (no query). One level of nesting is supported; relations-of-relations is a follow-on. Eager loading needs the relation's key column present in the base projection (the default `SELECT *` includes it). A very large parent set is handled by splitting each relation's `WHERE fk IN (...)` batch so the placeholder count never exceeds the per-statement limit (so the query count is `1 + R` for the

usual case, growing only when a key set exceeds the limit - never an over-limit statement). For a many-to-many, a child row keeps the joined join-table columns (so pivot payload survives); if the target and join table share a column name, that name collapses in the row map (join tables conventionally use distinct FK names, so this is rare).

Sessions and CRUD

CRUD runs through an `orm.Session` , the unit of work that wraps either a connection (auto-committing) or a transaction:

Call	Returns	Notes
orm.session(conn)	Session	An auto-committing session over a sql.Connection.
orm.transaction(tx)	Session	A session over a sql.Tx, so CRUD runs inside a caller's transaction.

Each repository function takes a `Session` first, then the schema and a record or key:

Call	Returns	Notes
orm.insert(session, schema, record)	sql.Result	Insert the record's present columns.
orm.find(session, schema, id)	map of string to string	The row with that primary-key value (throws if none).
orm.update(session, schema, record)	sql.Result	Update the non-key columns, matched by the record's primary key.
orm.delete(session, schema, id)	sql.Result	Delete by primary-key value.
orm.all(session, query)	list of map of string to string	Every row matching a built query.

Transactions come straight from `sql` - `sql.begin` / `commit` / `rollback` - and `orm.transaction` bridges the `Tx` into a `Session` , so the same CRUD calls run inside the transaction:

```
def sess as orm.Session init orm.session($conn);   # auto-commit
orm.insert($sess, $users, $ada);

def tx as sql.Tx init sql.begin($conn);            # or a transaction
errdefer sql.rollback($tx);
def txn as orm.Session init orm.transaction($tx);
orm.insert($txn, $users, $bob);
sql.commit($tx);
```

Bulk and conditional writes

Beyond single-row CRUD, the write path covers upsert, batch insert, generated keys, and bulk conditional mutations - all through a `Session` :

Call	Returns	Notes
orm.upsert(session, schema, record, conflictCols)	sql.Result	Insert, or update the non-conflict columns on a unique/PK conflict (Postgres ON CONFLICT, MySQL ON DUPLICATE KEY).
orm.insertMany(session, schema, records)	int	One multi-row INSERT (records must set the same columns); returns the count inserted.
orm.insertReturning(session, schema, record)	string	Insert and return the generated primary-key value.
orm.updateWhere(session, schema, assignments, query)	sql.Result	Bulk-update the rows matching the query's WHERE.

orm.deleteWhere(session, schema, query)	sql.Result	Bulk-delete the rows matching the query's WHERE.
orm.save(session, schema, record)	sql.Result	Insert when the record has no primary key, else update.

`updateWhere` and `deleteWhere` **refuse a query with no WHERE** , so a missing condition can never update or delete every row (use a raw `sql.exec` to truncate deliberately). `insertMany` never silently caps: a batch whose placeholder count would exceed the per-statement limit (60000) is split across several `INSERT` s - wrap it in `orm.transaction` for all-or-nothing. Every value binds through a placeholder.

```
orm.upsert($sess, $users, $ada, ["email"]);          # insert, or update on an
email conflict
def id as string init orm.insertReturning($sess, $users, $bob);
def bump as map of string to string init {};
$bump["active"] = "false";
orm.updateWhere($sess, $users, $bump, orm.where(orm.from($users), "age", "<",
"13"));
```

Finders

Everyday read helpers over a `Session` and a query:

Call	Returns	Notes
orm.first(session, query)	map of string to string	The first matching row (adds LIMIT 1), or {} if none.
orm.exists(session, query)	bool	Whether any row matches.
orm.findBy(session, schema, col, value)	map of string to string	The first row where col = value, or {}.
orm.pluck(session, query, col)	list of string	One column's value for each matching row.

```
def ada as map of string to string init orm.findBy($sess, $users, "name", "ada");
def anyAdmins as bool init orm.exists($sess, orm.where(orm.from($users), "role",
"=", "admin"));
def names as list of string init orm.pluck($sess, orm.from($users), "name");
```

A raw-SQL `whereRaw` escape hatch was **rejected** - it is the one thing that could not keep orm's "no injection even from a hand-built query" guarantee (an arbitrary SQL fragment can't be allowlist-checked); genuinely bespoke SQL belongs in the `sql` library. See rejected.md.

Scope

The repository CRUD, the write path, finders, the query builder, column-attribute DDL, and the standalone DDL builders above. Schema migrations are a separate concern in the `sqlmigrate` module (which consumes the DDL builders' strings). **Out of scope here** : relations beyond a plain `join` (has-many / belongs-to eager loading wants object identity Jennifer does not have) and a typed-struct row form (gated on map-to-struct conversion). Needs `sql` , so the **default jennifer** binary.

password - password generation, validation, and scoring

Import with `import "password.j" as password;` . Generate passwords against a policy `Schema` , validate a candidate password against that policy, and estimate any password's strength in bits of entropy. Pure `.j` over `crypto` / `strings` / `convert` ; runs on both binaries.

> **Security note.** Randomness (character choice and the final shuffle) comes from the `crypto` library's crypto-grade source, so a generated password is unpredictable and **safe to mint as a real credential** . It is not seedable (crypto randomness has no reproducible sequence). Storing passwords for later verification is a separate concern - use a memory-hard hash for that, not this generator.

```
import "password.j" as password;
use io;

def policy as password.Schema init password.schema();          # 16 chars, all
classes
def pw as string init password.generate($policy);
io.printf("%s  valid=%t  %s\n", $pw,
          password.validate($policy, $pw).valid,
          password.complexity($pw).label);
```

Runnable: `examples/modules/password_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/password_demo.j).

The policy schema

```
def struct password.Schema {
    minLength as int,          # shortest length (also the generated-length floor)
    maxLength as int,          # longest length (generation picks in the range)
    lower as bool,             # include lowercase in the alphabet
    upper as bool,             # include uppercase
    digits as bool,            # include digits
    symbols as bool,           # include symbols (from symbolSet)
    symbolSet as string,       # the symbol characters to draw from
    minLower as int,           # minimum lowercase (generation guarantees,
validation requires)
    minUpper as int,
    minDigits as int,
    minSymbols as int,
    excludeAmbiguous as bool   # drop ambiguous glyphs (0 O o 1 l I |)
};
```

Build a schema with the constructor and copy-on-write modifiers (each returns a fresh `Schema` , so they chain):

Call	Returns	
password.schema()	Schema	the strong default: 16 chars, all four classes, min 1 of each
password.withLength(s, lo, hi)	Schema	set the length range (lo == hi for a fixed length)
password.withClasses(s, lo, up, dig, sym)	Schema	enable/disable each class (bools)
password.withMinimums(s, lo, up, dig, sym)	Schema	set the per-class minimum counts
password.withSymbolSet(s, chars)	Schema	replace the symbol pool
password.withoutAmbiguous(s)	Schema	exclude ambiguous glyphs from generation

A **disabled class is authoritative over a leftover minimum** : `withClasses(s, true, true, true,`

`false`) on the default (which sets `minSymbols` to 1) produces no symbols and requires none - the enable bool wins.

Generate

```
def pw as string init password.generate(schema);
```

Picks a length in `[minLength, maxLength]` , lays down the per-class minimums, fills the rest from the enabled alphabet, and shuffles. Throws `Error{kind: "password"}` for an infeasible schema - no classes enabled, an empty required pool, `minLength` > `maxLength` , or minimums that exceed the length.

Validate

```
def report as password.Report init password.validate(schema, pw);
# Report { valid as bool, reasons as list of string }
```

Checks the **length bounds** and each **per-class minimum** , returning `valid` plus a list of the failed rules (empty when valid). It checks minimums, not a whitelist: a password is not rejected for containing characters outside the schema's alphabet (which is how real password policies read). Disabled classes impose no minimum.

Complexity

```
def strength as password.Strength init password.complexity(pw);
# Strength { length, classes, poolSize, entropy as float, label }
```

Estimates strength independent of any schema. The **alphabet size** is the sum of the class sizes present (lowercase / uppercase 26 each, digits 10, symbols the default-set size of 28), and **entropy** is `length * log2(poolSize)` bits. The `label` bands the entropy:

Entropy (bits)	Label
< 28	very weak
28 - 35	weak
36 - 59	reasonable
60 - 127	strong
>= 128	very strong

Entropy is a **ceiling on guessing difficulty given the character set** , not a measure of memorability or dictionary resistance: `password` scores as "reasonable" by length yet is trivially guessed. Treat the score as "how big is the brute-force space," not "is this a good password."

Scope

- **Crypto-grade randomness** (see the security note) - generation draws from the `crypto` library, so a generated password is safe as a real credential.
- **Complexity is character-set entropy** , with no dictionary, keyboard-walk, or repeated-character analysis. It will happily call a common word "reasonable".
- **Rune-based length** : `len` counts runes, so multi-byte characters count as one, consistent with the rest of the language.

See also

- crypto.md - the crypto-grade random source behind password generation.
- modules/index.md - the module catalog and import rules.

pdf - generate simple PDF documents

Import with `import "pdf.j" as pdf;` . Build a `Document` of `Page` s with value-semantic builders - text, lines, rectangles - then `render()` writes the PDF object / xref structure by hand (no stdlib PDF) as `bytes` , the way `html` / `label` generate their formats. Content streams are FlateDecode-compressed via `compress` . Pure Jennifer; runs on **both** binaries.

```
import "pdf.j" as pdf;
use fs;

def p as pdf.Page init pdf.page(612, 792);
$p = pdf.text($p, 72, 720, "Helvetica", 24, "Hello, PDF");
def doc as pdf.Document init pdf.addPage(pdf.document(), $p);
fs.writeBytes("out.pdf", pdf.render($doc));
```

Runnable: `examples/modules/pdf_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/pdf_demo.j).

Coordinates and units

Coordinates are in **PDF points** (1/72 inch), with the origin at the **bottom-left** and y increasing **upward** . All coordinates and sizes are integers. Common page sizes: US Letter `612` x `792` , A4 `595` x `842` . Colours are `0-255` RGB integers.

Building

Every builder is **value-semantic** - it returns a fresh copy and never mutates its argument, so you thread them (`$p = pdf.text($p, ...)`).

Call	Returns	
pdf.document()	Document	an empty document
pdf.page(width, height)	Page	a blank page of the given size
pdf.text(pg, x, y, font, size, str)	Page	draw text in a standard-14 font at (x, y)
pdf.loadFont(name, ttfBytes)	LoadedFont	load an embeddable TrueType font under a resource name
pdf.addFont(doc, lf)	Document	register an embedded font in the document
pdf.textUnicode(pg, x, y, lf, size, str)	Page	draw Unicode text in an embedded font at (x, y)
pdf.loadImage(name, imgBytes)	Image	load a PNG / JPEG under a resource name
pdf.addImage(doc, img)	Document	register an embedded image in the document
pdf.drawImage(pg, img, x, y, width, height)	Page	draw an image into the box at (x, y)
pdf.measureText(font, size, str)	float	width of str in points (standard-14)
pdf.measureTextUnicode(lf, size, str)	float	width of str in points (embedded font)
pdf.wrapText(font, size, str, maxWidth)	list of string	word-wrap to maxWidth points (standard-14); a line with an unbreakable token is hard-folded so nothing exceeds maxWidth
pdf.foldLine(font, size, text, maxWidth)	list of string	hard-fold one line to maxWidth, breaking at a space / seam or mid-token (a code line, a bare URL)
pdf.toWinAnsi(s, replacement)	string	replace every character the standard-14 fonts cannot encode with replacement ("" drops it)
pdf.wrapTextUnicode(lf, size, str, maxWidth)	list of string	word-wrap to maxWidth points (embedded font)

pdf.textBlock(pg, x, y, width, font, size, leading, str, align)	Page	flow wrapped, aligned text into a column (standard-14)
pdf.textBlockUnicode(pg, x, y, width, lf, size, leading, str, align)	Page	flow wrapped, aligned text into a column (embedded font)
pdf.line(pg, fromX, fromY, toX, toY)	Page	draw a stroked line
pdf.rect(pg, x, y, width, height, filled)	Page	draw a rectangle (fill or stroke)
pdf.link(pg, x, y, width, height, uri)	Page	a clickable link annotation (/Link + /URI); an invisible rect over drawn text makes it a hyperlink
pdf.color(pg, red, green, blue)	Page	set fill + stroke colour for what follows
pdf.addPage(doc, pg)	Document	append a page
pdf.pageLabel()	PageLabel	a blank running header / footer spec (Helvetica 9pt, 36pt margin)
pdf.setHeader(doc, label)	Document	attach a running header, drawn on every page at render
pdf.setFooter(doc, label)	Document	attach a running footer, drawn on every page at render
pdf.getTotalPages(doc)	int	pages added so far (what %pages% expands to)
pdf.getCurrentPageNr(doc)	int	the page being built, getTotalPages(doc) + 1
pdf.render(doc)	bytes	the finished PDF

Running headers and footers

A `PageLabel` is drawn on every page at render time, once the total page count is known. It has three text slots - `left` / `center` / `right` , each aligned with the font's own metrics - and each may contain `%page%` (the 1-based page number) and `%pages%` (the total). The tokens are percent-delimited (not braces) so they do not collide with Jennifer's `{expr}` string interpolation. `border` draws a thin rule under a header / over a footer; `margin` is the inset from the page edge; `font` / `size` / `red` / `green` / `blue` style the text.

```
def f as pdf.PageLabel init pdf.pageLabel();
$f.left = "sample.pdf";
$f.center = "page %page%/%pages%";   # -> "page 13/108"
$f.right = "jennifer-lang.dev";
$f.rightUri = "https://jennifer-lang.dev";  # makes the right slot a clickable
backlink
$f.border = true;
$doc = pdf.setFooter($doc, $f);
def out as bytes init pdf.render($doc);
```

A slot becomes a clickable link when its `leftUri` / `centerUri` / `rightUri` is set - the running footer's backlink to a website, drawn on every page. (Under the hood that is a `pdf.link` annotation over the slot's measured rectangle; use `pdf.link` directly to make any drawn text a hyperlink.)

`getTotalPages` / `getCurrentPageNr` report the page count while building (the total is known only once every page is added, which is why the placeholders resolve at render). Because Jennifer has no closures, per-page footer text beyond the `%page%` / `%pages%` placeholders is not expressible; for fully custom per-page labels, build all pages, then loop them drawing with `pdf.text` and `getTotalPages(doc)` .

`color` sets the drawing colour for **subsequent** operations on that page (both fill and stroke), so order matters: set the colour, then draw. `rect` 's `filled` flag fills the rectangle when `true` , otherwise strokes its outline.

Fonts

`text` takes one of the **standard-14** base fonts every PDF viewer provides; any other name throws `Error{kind: "pdf"}` :

```
Helvetica  Helvetica-Bold  Helvetica-Oblique  Helvetica-BoldOblique
Times-Roman  Times-Bold  Times-Italic  Times-BoldItalic
Courier  Courier-Bold  Courier-Oblique  Courier-BoldOblique
Symbol  ZapfDingbats
```

Each distinct font used becomes one shared Type1 font object (`WinAnsiEncoding`). Text is escaped for the PDF literal-string syntax (`\` , `(` , `)` , and line breaks), so any ASCII / Latin-1 string is safe to pass.

Embedded TrueType fonts (Unicode)

For text beyond Latin-1 - accented text, Greek, Cyrillic, CJK, ... - embed a TrueType (`.ttf`) font and draw with `textUnicode` :

```
def body as pdf.LoadedFont init pdf.loadFont("Body", fs.readBytes("/path/NotoSans.
ttf"));
def doc as pdf.Document init pdf.addFont(pdf.document(), $body);
def p as pdf.Page init pdf.textUnicode(pdf.page(595, 842), 50, 780, $body, 18,
"Héllo, ???");
$doc = pdf.addPage($doc, $p);
```

The font is embedded as a **Type0 / CIDFontType2** composite (Identity-H encoding, `FontFile2` font program) over the `font` module: each character maps through the font's cmap to a glyph, so any script the font covers renders. A **ToUnicode** map is written too, so the text stays selectable and copyable in a viewer. The whole font file is embedded (glyph subsetting, for smaller files, is a follow-on). `loadFont` accepts a TrueType (`glyf`) font; a CFF / OpenType-PostScript (`.otf`) font throws (its `FontFile3` embedding is a follow-on).

Images

Embed a PNG or JPEG as an image XObject and draw it scaled into a box:

```
def logo as pdf.Image init pdf.loadImage("Logo", fs.readBytes("/path/logo.png"));
def doc as pdf.Document init pdf.addImage(pdf.document(), $logo);
def p as pdf.Page init pdf.drawImage(pdf.page(595, 842), $logo, 50, 700, 120, 90);
$doc = pdf.addPage($doc, $p);
```

`loadImage` detects the format from the file signature and reads the pixel geometry / colour space. `addImage` registers the image on the document (so the resource is written once and can be reused across pages); `drawImage` places it, **scaled to fill** the `width` x `height` box in points - pass a box in the image's own proportion to avoid stretching. The origin is the box's lower-left, like everything else.

Supported inputs:

Format	Handling
JPEG (baseline / progressive)	embedded as-is via DCTDecode; greyscale / RGB / CMYK (an Adobe CMYK file gets an inverting /Decode)
PNG greyscale / RGB / palette	embedded directly - the raw zlib image data with a FlateDecode PNG predictor, so no pixel decode happens (1 / 2 / 4 / 8 / 16-bit)
PNG greyscale+alpha / RGBA (8-bit)	decoded (inflate + de-filter) and split into a colour stream plus an 8-bit greyscale soft mask (/SMask), so transparency renders

The image and its soft mask round-trip pixel-exact (the JPEG is re-embedded losslessly). **Not** supported: interlaced (Adam7) PNG, 16-bit alpha, and palette `tRNS` transparency - each throws `Error{kind: "pdf"}` with a reason. `addImage` the image before you `drawImage` it, so the `/name` resolves (the same rule as `addFont` / `textUnicode`).

Text layout

`text` / `textUnicode` place a single string at exact coordinates. To flow a paragraph into a **column** - wrapped, aligned, multi-line - use the layout functions, which sit on top of accurate width measurement.

Measure. `measureText(font, size, str)` returns the rendered width of a string in points, using the Adobe Core-14 AFM metrics for the standard-14 fonts; `measureTextUnicode(lf, size, str)` uses an embedded font's own glyph advances.

Wrap. `wrapText(font, size, str, maxWidth)` greedily word-wraps to a maximum line width (points) and returns the lines. Existing newlines in `str` are honoured as hard breaks (so a `\n\n` leaves a blank line), and runs of spaces collapse. A single word wider than `maxWidth` lands alone on its line (it overflows rather than being broken mid-word). `wrapTextUnicode` is the embedded-font form.

Flow. `textBlock(pg, x, y, width, font, size, leading, str, align)` wraps `str` to `width` and draws each line - the first line's baseline at `(x, y)` , each next line `leading` points lower:

```
def body as string init "Jennifer flows wrapped, justified paragraphs into a "
    + "column using accurate standard-14 width metrics.";
$p = pdf.textBlock($p, 72, 720, 240, "Times-Roman", 11, 15, $body, "justify");
# the block is len(pdf.wrapText("Times-Roman", 11, $body, 240)) * 15 points tall
```

`align` is one of:

align	Effect
"left"	lines start at x (the default look)
"right"	lines end at x + width
"center"	lines centred in the column
"justify"	inter-word gaps padded so each line fills width exactly - except the last line of each paragraph, which stays left-aligned

`textBlockUnicode` is the same for an embedded font. The block does not clip or paginate; a column height is `len(wrapText(...)) * leading` , so you decide where it ends and when to start a new page. Measuring a Symbol / ZapfDingbats font, or a character outside WinAnsi, throws `Error{kind: "pdf"}` .

Metadata

Set document metadata - the PDF Info dictionary shown in a viewer's "Document Properties" - with `pdf.info(doc, key, value)` . `key` is a PDF Info key:

Key	
Title / Author / Subject / Keywords	the descriptive fields
Creator	the app that authored the source
Producer	the app that wrote the PDF (defaults to "Jennifer pdf")
CreationDate / ModDate	PDF date strings (see pdfDate below)

```
def doc as pdf.Document init pdf.document();
$doc = pdf.info($doc, "Title", "Q3 Report");
$doc = pdf.info($doc, "Author", "Ada Lovelace");
$doc = pdf.info($doc, "Keywords", "report, finance, q3");
```

`document()` presets `Producer` to `"Jennifer pdf"` ; every other field is unset until you set it. Any custom key works too. Dates use the PDF date syntax, which `pdf.pdfDate(t)` builds from a `time.Time` :

```
use time;
$doc = pdf.info($doc, "CreationDate", pdf.pdfDate(time.utc()));   #
D:20260714160000+00'00'
```

Bookmarks (outline)

`pdf.bookmark(doc, page, y, title, level)` appends an outline entry - a bookmark in the viewer's sidebar that jumps to page `page` (0-based) at `y` (PDF points, origin bottom-left). Entries **nest by level** in the

order added: a level-2 entry becomes a child of the most recent level-1, and so on, so a heading hierarchy becomes a bookmark tree. When any bookmark is present, `render` emits the `/Outlines` object tree and opens the document with the bookmark panel showing (`/PageMode /UseOutlines`).

```
$doc = pdf.bookmark($doc, 0, 720, "Introduction", 1);
$doc = pdf.bookmark($doc, 0, 600, "Background", 2);     # nests under Introduction
$doc = pdf.bookmark($doc, 1, 720, "Results", 1);
```

Rendering

`render(doc)` produces a complete PDF 1.7 file as `bytes` : a catalog, a page tree, one page dict + one FlateDecode-compressed content stream per page, the shared font objects, the image XObjects (each with a soft-mask XObject when it carries alpha), an Info dictionary when any metadata is set, a cross-reference table with correct byte offsets, and the trailer. Object numbers are assigned dynamically, so any mix of pages, fonts, and images cross-references correctly. Write it with `fs.writeBytes` , return it from an `httpd` handler, or attach it via `mime` . It validates clean under `qpdf --check` .

Byte-identical output. The same document always renders to the **exact same bytes** - on either binary, run to run. This is deliberate: `pdf` never auto-stamps a `CreationDate` or any other timestamp (you opt into one explicitly via `info` + `pdfDate`), so nothing varies with wall-clock time. That makes the output safe to assert against a golden file in an automated test, and reproducible for content-addressed builds.

Scope

- **Text, lines, rectangles.** The standard-14 fonts, solid fills / strokes, and RGB colour. No curves / paths beyond rectangles, no clipping, no transparency.
- **Embedded TrueType fonts** (Type0 / CIDFontType2, Unicode + ToUnicode) via `loadFont` / `textUnicode` . Glyph subsetting (smaller files) and CFF `.otf` embedding are follow-ons.
- **Raster images** (PNG greyscale / RGB / palette / alpha, JPEG) via `loadImage` / `drawImage` . Interlaced PNG, 16-bit alpha, and palette `tRNS` are follow-ons.
- **Text layout** - width measurement, word-wrap, and left / right / center / justify column flow via `measureText` / `wrapText` / `textBlock` . No automatic pagination or clipping (you place each block); tab stops, hyphenation, and mixed inline runs are out of scope.
- **A writer, not a reader.** It generates PDFs; it does not parse them.

See also

- compress.md - the FlateDecode (`zlib`) streams.
- html.md / label.md - the sibling format-generation modules.
- fs.md - `writeBytes` to save the rendered PDF.
- modules/index.md - the module catalog and import rules.

plot - data plotting to SVG

Import with `import "plot.j" as plot;` . Turns numbers into a self-contained <svg> chart string you can write to a `.svg` file or drop into HTML. A unified `chart` renders one or more **series** (line, points, both, or filled area, solid or dashed, with **error bars** and pickable **marker shapes**) on shared axes with a positioned **legend** ; `line` / `scatter` / `bar` / `bars` / `histogram` are focused wrappers, and `bars` **groups or stacks** multiple series (values may be negative - bars **diverge** from a zero baseline). Charts support **log scales** , a **date axis** , **fonts** , configurable **margins** , **reference lines** , **data labels** on bars, and native <title> **hover tooltips** . `save` writes the SVG to a file. Pure Jennifer over `math` and `time` / `fs` / `strings` / `lists` / `convert` ; both binaries; the visual companion to the `stats` / `ml` numeric stack.

```
import "plot.j" as plot;

def opts as plot.Options init plot.defaults();
$opts.title = "measured vs model";
$opts.refLines = [plot.hline(20.0, "target")];

def measured as plot.Series init plot.points("measured", [1.0, 2.0, 3.0], [1.1, 4.
2, 8.8]);
$measured.yErr = [0.4, 0.5, 0.6];            # symmetric error bars
$measured.shape = "square";                   # marker shape
def model as plot.Series init plot.series("model", [1.0, 2.0, 3.0], [1.0, 4.0, 9.
0]);
plot.save(plot.chart([$measured, $model], $opts), "fit.svg");
```

Runnable: `examples/modules/plot_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/plot_demo.j).

Surface

Call	Returns	Notes
plot.defaults()	Options	A 640x400 canvas, blue on white, sans-serif 11px, grid + top-right legend + tooltips on.
plot.series(name, xs, ys)	Series	A line series (set .mark / .color / .dash / .yErr / .shape).
plot.points(name, xs, ys)	Series	A points (scatter) series.
plot.chart(series, opts)	string	Render a list of Series on shared axes, with a legend for named series.
plot.line(xs, ys, opts)	string	One line series in opts.color.
plot.scatter(xs, ys, opts)	string	One points series in opts.color.
plot.bar(labels, values, opts)	string	A single-series bar chart, one bar per category (negatives diverge).
plot.bars(labels, series, opts)	string	A multi-series bar chart - grouped or stacked per opts.barMode.
plot.histogram(data, bins, opts)	string	A histogram of data into bins equal-width buckets.
plot.hline(value, label)	RefLine	A horizontal reference line (append to opts.refLines).
plot.vline(value, label)	RefLine	A vertical reference line.
plot.floats(xs)	list of float	Lift a list of int to list of float.
plot.save(svg, path)	string	Write an SVG string to a file with fs; returns the path.

Chart data is `list of float` (use `plot.floats` or float literals). Each render returns a complete SVG document. Empty / mismatched input, a ragged `bars` series, or a non-positive value on a log axis throw a catchable `Error{kind: "plot"}` ; `histogram` caps `bins` at 10000. Jennifer values have no methods, so

saving is `plot.save($svg, path)` , not `$svg.save(path)` .

The output is safe to build from untrusted data: text is XML-escaped, and every caller-supplied attribute value (`color` / `background` / `fontFamily` , series and ref-line colours) is attribute-escaped, so a hostile string cannot break out and inject markup. Per-mark tooltip names are truncated to keep output bounded, and data magnitudes past the int range (>= 2^63) render in scientific notation rather than failing.

Series

`{ name, xs, ys, color, mark, dash, yErr, shape }` - `name` the legend label (`""` = not listed); `xs` / `ys` the coordinates (`bars` uses only `ys`); `color` `""` takes a palette colour; `mark` is `"line"` , `"points"` , `"both"` , or `"area"` ; `dash` draws the line dashed; `yErr` , when the same length as `ys` , draws symmetric error bars (linear y axis only); `shape` picks the scatter marker

- `"circle"` , `"square"` , `"triangle"` , or `"diamond"` . Build with `plot.series`

(line) or `plot.points` , then set the fields.

Options

`{ width, height, title, xLabel, yLabel, color, background, fontFamily, fontSize, grid, legend, legendPos, xLog, yLog, xDate, dateFormat, marginLeft, marginRight, marginTop, marginBottom, barMode, barLabels, hover, refLines }`

- **size / captions** - `width` / `height` px; `title` / `xLabel` / `yLabel` .
- **colours / fonts** - `color` , `background` ; `fontFamily` and `fontSize` .
- **frame** - `grid` (gridlines vs. tick marks), the four `margin*` fields.
- **legend** - `legend` on/off and `legendPos` (`"top-right"` / `"top-left"` / `"bottom-right"` / `"bottom-left"`).
- **scales** - `xLog` / `yLog` (log10, decade ticks, positive only) and `xDate` / `dateFormat` (x = Unix seconds, calendar-boundary ticks via `time`).
- **bars** - `barMode` `"grouped"` / `"stacked"` ; `barLabels` draws each grouped bar's value; negative values diverge below a zero baseline.
- **tooltips** - `hover` attaches a `<title>` to every mark for native browser hover text.
- **reference lines** - `refLines` , a `list of RefLine` from `hline` / `vline` .

Scope

Data charts over `math` for the axis math. Possible follow-ons: a secondary y-axis, per-series point size, and non-10 log bases. For graph / network diagrams (nodes and edges) rather than data charts, see `dot` .

pop - receive mail (POP3 client)

Import with `import "pop.j" as pop;` . A **POP3** receive client (RFC 1939): the line-oriented status dialogue (`+OK` / `-ERR`) over the `net` system library, with plaintext / implicit-TLS / STLS transport and auth by USER / PASS , APOP, XOAUTH2, CRAM-MD5, or SCRAM-SHA-1 / SCRAM-SHA-256. Retrieved messages come back as strings, ready for the `mime` module to parse. Because it uses `net` , this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

> The module is named `pop` (not `pop3`): a Jennifer namespace is letters-only, so a digit in the name can't be a call prefix. It is POP version 3 - the only one in use - the same choice Ruby's `net/pop` makes.

```
import "pop.j" as pop;
import "mime.j" as mime;

def opts as pop.Options init pop.Options{host: "mail.example.com", port: 995,
    security: "tls", user: "me", pass: "secret"};
for (def raw in pop.fetchAll($opts)) {
    def msg as mime.Part init mime.parse($raw);
    io.printf("subject: %s\n", mime.headerValue($msg, "Subject"));
}
```

Runnable: `examples/modules/pop_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/pop_demo.j).

Surface

A session is stateful: `connect` , issue commands, `quit` . `fetchAll` wraps the common "get every message" case.

Call / type	Notes
pop.Options	host, port, security, user, pass.
pop.Session	A live session over one connection (from connect).
pop.Stat	count and total size, from stat.
pop.MessageId	number (this session) + id (persistent UIDL id), from uidl.
pop.connect(opts)	Open a session: greet, optional STLS, USER / PASS.
pop.stat(session)	Mailbox Stat (STAT).
pop.count(session)	Just the message count.
pop.sizes(session)	list of int - each message's octet size, in order (LIST).
pop.retrieve(session, n)	Message n as a raw string (RETR), for mime.parse.
pop.uidl(session)	UIDL - each message's stable id -> list of pop.MessageId (number, id).
pop.uidlOne(session, n)	UIDL n - one message's stable id (string).
pop.top(session, n, lines)	TOP n lines - the headers plus lines body lines (0 = headers only), for a cheap preview.
pop.deleteMessage(session, n)	Mark message n for deletion (DELE); removed at quit.
pop.reset(session)	RSET - unmark every message marked for deletion this session.

pop.noop(session)	NOOP - keep the connection alive (server resets its idle timer).
pop.quit(session)	End the session (commit deletions) and close.
pop.fetchAll(opts)	Connect, retrieve every message (no delete), quit; list of string.

`Options.security` is a `transport.Security` : `.None` (plaintext, port 110), `.Tls` (implicit TLS on connect, port 995), or `.Starttls` (STLS upgrade on 110). Build it with `import "transport.j" as transport;` .

Retrieval and dot-stuffing

`retrieve` and `sizes` read a **multi-line** response terminated by a `.` line, and undo the byte-stuffing POP3 applies (a body line that began with a `.` was sent doubled, e.g. `..sig` on the wire is `.sig` in the message), so the string you get back is the exact message:

```
def s as pop.Session init pop.connect($opts);
io.printf("%d messages\n", pop.count($s));
def raw as string init pop.retrieve($s, 1); # RFC 5322 message text
pop.deleteMessage($s, 1);                   # optional
pop.quit($s);                               # deletion commits here
```

A `-ERR` from the server throws a catchable `Error` (kind `"pop3"`).

Certificate verification for `"tls"` / `"starttls"` is the `net` default.

Stable ids: download only what is new

A message `number` is only meaningful within the current session; the `UIDL` `id` is **persistent** across sessions, so it is the key to leave-on-server / skip-seen (POP3's `RETR` does not delete, so a mailbox can be polled without consuming it). Keep the ids you have processed, and next time retrieve only the `number` s whose `id` is new:

```
def seen as map of string to bool init loadSeen();   # your persisted set
def s as pop.Session init pop.connect($opts);
for (def m in pop.uidl($s)) {
    if (not maps.has($seen, $m.id)) {
        def raw as string init pop.retrieve($s, $m.number);
        # ... process the new message ...
        $seen[$m.id] = true;
    }
}
pop.quit($s);   # no deleteMessage -> the mailbox is left intact
```

`top(session, n, 0)` fetches only the headers of message `n` - a cheap preview (Subject / From / Date via `mime.parse`) before deciding whether to `retrieve` the whole body. `reset` unmarks any pending `deleteMessage` s (abort a batch before `quit` commits it); `noop` keeps an idle connection from timing out. `UIDL` and `TOP` are widely but not universally supported - a server lacking one answers `-ERR` (a catchable `"pop3"` error).

Testing

The pure protocol logic - `+OK` detection, `STAT` parsing, `LIST` sizes, and the multi-line dot-terminator / un-stuffing - is unit-tested in the overlay. The networked session is covered end to end by an in-process fake POP3 server in the Go test suite (`TestPop3Receive`), so it runs in CI without an external server.

Out of scope

- **Receive only** (retrieve / delete). Sending is `smtp` .
- **Auth** : `USER` / `PASS` (default), APOP (`auth: "apop"` , RFC 1939 - proves the password with MD5 of the greeting's timestamp, never sending it), or SASL `AUTH` - XOAUTH2 (`auth: "xoauth2"` , for Google / Microsoft 365), CRAM-MD5 (`"cram"`), or SCRAM-SHA-1 / SCRAM-SHA-256 (`"scram-sha-1"` / `"scram-sha-256"`), via `sasl` . `auth: "auto"` picks the strongest the server offers (SASL from `CAPA` ,

then APOP when the greeting carries a timestamp, else USER / PASS); `auth: ""` is plain USER / PASS.
- **No TOP / UIDL.** Just `STAT` / `LIST` / `RETR` / `DELE` .
- An internationalized (IDN) host is IDNA-encoded to its `xn--` form automatically (via `idna`).

Timeouts and limits

Reads carry a 30 s idle timeout (a deadline re-armed before each read), so a hung server fails with a catchable error instead of blocking the caller forever. The initial connect (and a STARTTLS handshake) is bounded by its own connection-establishment timeout, so a slow or unreachable server fails the dial. A single accumulated response is capped at **64 MiB** , so a server that streams a status line or a dot-terminated body that never ends fails with a catchable error rather than growing the buffer without bound.

See also

- mime.md - parse a retrieved message (`mime.parse`).
- smtp.md - the send half of the mail suite.
- net.md - the transport `pop` builds on.
- modules/index.md - the module catalog and import rules.

prometheus - metrics exposition and query

Import with `import "prometheus.j" as prometheus;` . A **Prometheus** module in two halves. **Exposition** builds a metric set and renders the Prometheus text format - pure text over `strings` / `maps` / `lists` / `convert` , and transport-agnostic: write the string to a `*.prom` file for the node_exporter textfile collector, POST it to a Pushgateway, or serve it from a `/metrics` handler. **Retrieval** is a read client for Prometheus's HTTP query API, built on the `http` module + `json` .

The exposition half runs on **both** binaries. The query half uses `net` through `http` , so it needs the default **jennifer** binary; on `jennifer-tiny` the exposition functions still work, and only `query` / `queryRange` surface the no-network error.

```
import "prometheus.j" as prometheus;

def m as prometheus.Metric init prometheus.counter("http_requests_total",
    "Total HTTP requests");
$m = prometheus.observe($m, {"method": "get", "code": "200"}, 42.0);
io.printf("%s", prometheus.render([$m]));
# # HELP http_requests_total Total HTTP requests
# # TYPE http_requests_total counter
# http_requests_total{code="200",method="get"} 42.0
```

Runnable: `examples/modules/prometheus_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/prometheus_demo.j).

Exposition

Build metrics, record samples, render the text format. Every builder is value-semantic (returns a new `Metric`), so a metric set is assembled by reassignment.

Call / type	Notes
prometheus.Metric	name, help, type (Counter / Gauge / Histogram / Summary), samples, buckets, quantiles.
prometheus.Sample	labels (map), value (float), plus count / sum / buckets / observations / timestamp / hasTimestamp for the aggregate types.
prometheus.counter(name, help)	A new counter metric; throws on an invalid name.
prometheus.gauge(name, help)	A new gauge metric; throws on an invalid name.
prometheus.histogram(name, help, buckets)	A new histogram; buckets is a list of float of upper bounds (le), sorted ascending.
prometheus.summary(name, help, quantiles)	A new summary; quantiles is a list of float in [0, 1], sorted ascending.
prometheus.observe(metric, labels, value)	Record an observation; throws on an invalid label name.
prometheus.observeAt(metric, labels, value, timestampMs)	As observe, plus an explicit millisecond timestamp.
prometheus.render(metrics)	Render a list of Metric as the text exposition format.
prometheus.pushgatewayPath(job, grouping)	Build the /metrics/job/<job>/<k>/<v>/... Pushgateway path (pure string).

For a `Counter` / `Gauge` , `observe` **upserts** : a sample with an equal label set is replaced (last write wins), so re-observing the same series updates its value rather than duplicating the line. For a `Histogram` / `Summary` , `observe` **accumulates** (see below). `render` sorts label keys, so output is deterministic regardless of the order labels were inserted.

Histograms and summaries

A **histogram** buckets observations by an upper bound (`le`). Each `observe` increments the count, adds to the sum, and increments every cumulative bucket whose bound is >= the value; `render` emits, for a metric named `x` : `x_bucket{le="..."}` (cumulative counts, ascending, `le="+Inf"` last), `x_sum` , and `x_count` , under a `# TYPE x histogram` header.

```
def h as prometheus.Metric init prometheus.histogram(
    "http_request_duration_seconds", "Request latency", [0.1, 0.5, 1.0]);
$h = prometheus.observe($h, {"method": "get"}, 0.3);
$h = prometheus.observe($h, {"method": "get"}, 0.05);
$h = prometheus.observe($h, {"method": "get"}, 2.0);
io.printf("%s", prometheus.render([$h]));
# # TYPE http_request_duration_seconds histogram
# http_request_duration_seconds_bucket{le="0.1",method="get"} 1.0
# http_request_duration_seconds_bucket{le="0.5",method="get"} 2.0
# http_request_duration_seconds_bucket{le="1.0",method="get"} 2.0
# http_request_duration_seconds_bucket{le="+Inf",method="get"} 3.0
# http_request_duration_seconds_sum{method="get"} 2.35
# http_request_duration_seconds_count{method="get"} 3.0
```

A **summary** reports quantiles. Each `observe` increments the count, adds to the sum, and retains the observed value; `render` emits `x{quantile="..."}` (computed from the retained observations by the nearest-rank method, `rank = ceil(q * n)`), `x_sum` , and `x_count` , under `# TYPE x summary` . Each label set accumulates its own count / sum / buckets / observations independently.

Timestamps

`observeAt(metric, labels, value, timestampMs)` records a sample carrying an explicit millisecond Unix timestamp; `render` appends it after the value (`metric value timestamp`), the optional third field the exposition format allows. Plain `observe` records no timestamp.

Pushing to a Pushgateway

`pushgatewayPath(job, grouping)` is a pure string helper that builds the Pushgateway URL path from a job name and a `map of string to string` of grouping labels: `/metrics/job/<job>/<k1>/<v1>/...` , with the job and values percent-encoded and grouping keys sorted for a deterministic path. POST the `render` output to `base` + `pushgatewayPath(job, grouping)` via the `http` module. An invalid grouping label name throws.

Strictness

The format's rules are enforced:

- A metric name must match `[a-zA-Z_:][a-zA-Z0-9_:]*` ; a label name must match `[a-zA-Z_][a-zA-Z0-9_]*` . A violation throws a catchable `Error` (kind `"prometheus"`).
- Label values escape `\` , `"` , and newline; `# HELP` text escapes `\` and newline. An empty `help` omits the `# HELP` line.

Getting the text to Prometheus

`render` returns a plain string; delivery is your choice:

- **Textfile collector** - write it to a `*.prom` file (via `fs`) in node_exporter's textfile directory.
- **Pushgateway** - POST it with the `http` module.
- **Scrape endpoint** - serve it from a `/metrics` handler (e.g. the `web` framework over `httpd`).

Retrieval

A read client for the HTTP query API. Both return a `Result` .

Call / type	Notes

prometheus.Result	resultType + series (a list of Series).
prometheus.Series	metric (label map) + values (a list of Point).
prometheus.Point	timestamp (float, Unix seconds) + value (float).
prometheus.query(base, promql)	Instant query (/api/v1/query) -> Result.
prometheus.queryRange(base, promql, start, end, step)	Range query (/api/v1/query_range) -> Result.

`base` is the server URL (e.g. `"http://localhost:9090"`). `start` / `end` are RFC 3339 or Unix-timestamp strings; `step` is a duration (`"15s"`) or a seconds string. An instant query returns a `"vector"` (one `Point` per series); a range query returns a `"matrix"` (many `Point` s per series). A server-reported query error throws an `Error` (kind `"prometheus"`).

```
def r as prometheus.Result init prometheus.query("http://localhost:9090", "up");
for (def s in $r.series) {
    io.printf("%s = %f\n", $s.metric["instance"], $s.values[0].value);
}
```

Testing

The pure exposition logic - name / label validation, value and HELP escaping, label-key sorting, the counter/gauge upsert, histogram bucket accumulation (with pinned `_bucket` / `_sum` / `_count` output), summary quantiles, timestamps, and the Pushgateway path - is unit-tested in the overlay (`modules/prometheus_test.j`), alongside the result parser against canned vector / matrix / scalar / error responses. The networked `query` / `queryRange` path is covered end to end against an in-process fake Prometheus in the Go test suite (`TestPrometheusQuery`), which also proves the PromQL URL encoding round-trips.

Out of scope

- **No registry / auto-collection.** The caller holds and assembles the metric set; there is no global default registry or process/Go collectors.
- **Summary quantiles are exact over the retained observations** , not the streaming, bounded-error estimators the reference clients use - simpler and more accurate, at the cost of keeping every observed value per series.
- **Query results are read-only values.** No PromQL building or evaluation - the server does that.

See also

- http.md - the client transport the retrieval half builds on.
- json.md - the query-response decoder.
- modules/index.md - the module catalog and import rules.

ratelimit - rate limiter (fixed or sliding window)

Import with `import "ratelimit.j" as ratelimit;` . A rate limiter over a **selectable** key/value backend (`kvstore.Store` - `memcache` , `redis` , or the in-process `kv`). Each key (an IP, a user, an API token) is counted against a limit per time window; a `check` records a hit and returns a `Result` with the decision **plus the metadata for a compliant 429** - remaining budget, reset time, and Retry-After.

```
import "ratelimit.j" as ratelimit;
import "kvstore.j" as kvstore;

def lim as ratelimit.Limiter init ratelimit.slidingWindow(kvstore.
redisStore($rc), 100, 60);
def r as ratelimit.Result init ratelimit.check($lim, "ip:203.0.113.7");
if (not $r.allowed) {
    # respond 429; set Retry-After: $r.retryAfter, X-RateLimit-Remaining: $r.
remaining
}
```

Surface

Call / type	Returns	
ratelimit.fixedWindow(store, limit, window)	Limiter	a fixed-window limiter: limit hits per aligned window seconds
ratelimit.slidingWindow(store, limit, window)	Limiter	a sliding-window limiter (smooths the boundary burst)
ratelimit.check(limiter, key)	Result	record a hit and report the decision + metadata
ratelimit.peek(limiter, key)	Result	report the current state without recording a hit
ratelimit.Result		allowed (bool), remaining (int), retryAfter (seconds, 0 when allowed), resetSeconds (seconds to the window roll)

Fixed vs sliding window

Both are driven by the wall clock and **window-aligned keys** (a counter per aligned window), so they work identically on every backend, needing only atomic increment.

- **Fixed window** - one counter per aligned window; simplest. A burst can straddle a window boundary: up to 2 `* limit` requests across the seam (all of one window's budget at its end, then all of the next window's at its start).
- **Sliding window** - blends the current window's count with the previous window's, weighted by how far into the current window you are. At the very start of a new window the previous window still carries full weight, so a fresh burst is blocked; its influence decays linearly to zero by the window's end. This smooths the fixed-window boundary burst.

The counter is created and armed with the window TTL on the first hit, so it resets on its own - nothing to reap.

Building a 429

`check` returns everything a compliant response needs:

```
def r as ratelimit.Result init ratelimit.check($lim, $key);
if ($r.allowed) {
    # ... serve; optionally advertise X-RateLimit-Remaining: $r.remaining ...
} else {
    # HTTP 429 Too Many Requests
    #   Retry-After: $r.retryAfter
    #   X-RateLimit-Remaining: 0
    #   X-RateLimit-Reset: $r.resetSeconds
}
```

See also

- kvstore.md - the backend selector; kv.md - the in-process backend.
- session.md - the other module on the `kvstore` backend layer.
- modules/index.md - the module catalog and import rules.

redis - a Redis client (RESP2)

Import with `import "redis.j" as redis;` . A **Redis** client speaking **RESP2** (the REdis Serialization Protocol) over the `net` system library. Commands go out as RESP arrays of bulk strings; replies (`+OK` , `-ERR` , `:int` , `$bulk` , `*array`) parse back into a `Reply` . Typed per-command helpers (`get` / `set` / `incr` / `keys` / ...) keep the common path fully typed; `command` is the generic escape hatch for everything else. Because it uses `net` , this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "redis.j" as redis;

def db as redis.Session init redis.connect(redis.Options{host: "127.0.0.1",
    port: 6379, security: "none", user: "", password: "", db: 0});
redis.set($db, "greeting", "hello");
io.printf("%s\n", redis.get($db, "greeting"));     # hello
io.printf("visits: %d\n", redis.incr($db, "visits"));
redis.quit($db);
```

Runnable: `examples/modules/redis_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/redis_demo.j).

Surface

A session is stateful: `connect` , issue commands, `quit` .

Call / type	Notes
redis.Options	host, port, security, user, password, db.
redis.Session	A live session over one connection (from connect).
redis.Reply	A parsed reply: kind, str, num, items (see below).
redis.Message	A pushed pub/sub message: kind, channel, pattern, payload.
redis.ScanResult	One scan page: cursor (int), keys (list of string).
redis.connect(opts)	Open a session; AUTH when password set, SELECT when db > 0.
redis.command(session, args)	Send any command (list of string); returns the raw Reply.
redis.get(session, key)	GET - the string value, or "" when the key is missing.
redis.set(session, key, value)	SET.
redis.getBytes(session, key)	GET returning raw bytes (byte-exact; empty when missing) - for a binary value.
redis.setBytes(session, key, value)	SET a raw bytes value byte-for-byte (a blob / compressed payload).
redis.del(session, key)	DEL - number of keys removed (0 or 1).
redis.exists(session, key)	EXISTS - bool.
redis.incr(session, key)	INCR - the new value (int).
redis.decr(session, key)	DECR - the new value (int).
redis.keys(session, pattern)	KEYS - list of string matching a glob (production-unsafe; see below).
redis.scan(session, cursor, pattern, count)	SCAN - one ScanResult page (the production-safe walk).

redis.ping(session)	PING - the server's "PONG".
redis.quit(session)	QUIT and close.
redis.subscribe(session, channels)	SUBSCRIBE to a list of string of channels.
redis.psubscribe(session, patterns)	PSUBSCRIBE to a list of string of glob patterns.
redis.unsubscribe(session, channels)	UNSUBSCRIBE ([] leaves all channels).
redis.punsubscribe(session, patterns)	PUNSUBSCRIBE ([] leaves all patterns).
redis.publish(session, channel, message)	PUBLISH - the number of subscribers that received it (int).
redis.receiveMessage(session)	Block for the next Message push (pair with spawn).
redis.pipeline(session, commands)	Send list of list of string, read one Reply each (one round trip).
redis.multi(session)	MULTI - begin a transaction.
redis.exec(session)	EXEC - run the queued commands; list of Reply (empty if aborted).
redis.discard(session)	DISCARD - drop the queued transaction.
redis.hset / hget / hgetAll / hdel	Hash: HSET (new-field count), HGET (string), HGETALL (map of string to string), HDEL (removed count).
redis.lpush / rpush / lrange / llen / lpop	List: push (new length), LRANGE start stop (list of string, 0, -1 = all), LLEN, LPOP (string).
redis.sadd / srem / smembers / sismember / scard	Set: add / remove (count), SMEMBERS (list of string), SISMEMBER (bool), SCARD (int).

`Options.security` is a `transport.Security` : `.None` (plaintext, port 6379) or `.Tls` (implicit TLS, `rediss`); `.Starttls` is rejected (Redis has no in-band upgrade). Needs `import "transport.j" as transport;` . `password ""` skips `AUTH` ; `db 0` skips `SELECT` . When a `user` is set alongside `password` , `AUTH user password` (ACL) is sent; otherwise `AUTH password` .

The generic command and Reply

Every typed helper is a thin wrapper over `command` , which sends an arbitrary argument list and returns the raw `Reply` - use it for any command without a helper:

```
def r as redis.Reply init redis.command($db, ["LPUSH", "queue", "job-1"]);
io.printf("list length now %d\n", $r.num);
def range as redis.Reply init redis.command($db, ["LRANGE", "queue", "0", "-1"]);
for (def item in $range.items) {
    io.printf("  %s\n", $item.str);
}
```

A `Reply` is walked by its `kind` and the matching field, the same shape a `json.Value` is walked with accessors:

kind	RESP source	Read from
"string"	+simple / $bulk	.str
"error"	-ERR	.str (but see below)
"int"	:123	.num
"nil"	$-1 / *-1	(absent)
"array"	*N	.items (a list of Reply)

Errors

A `-ERR` reply throws a catchable `Error` (kind `"redis"`) at the call site, so a bad command surfaces like any

other runtime error:

```
try {
    redis.command($db, ["INCR", "greeting"]);   # greeting holds "hello"
} catch (e) {
    io.printf("redis said: %s\n", $e.message);  # ERR value is not an integer...
}
```

`command` only throws on an error *reply* ; a network failure surfaces as the underlying `net` error.

Pub/Sub

`publish` sends a message on an ordinary connection and returns the number of subscribers that received it. On the receiving side, `subscribe` / `psubscribe` put a connection into **subscribed mode** , and `receiveMessage` blocks for the next push:

```
# Publisher (an ordinary connection).
def n as int init redis.publish($db, "news", "hello");   # subscribers reached

# Subscriber (a dedicated connection).
def sub as redis.Session init redis.connect($opts);
redis.subscribe($sub, ["news", "weather"]);
def msg as redis.Message init redis.receiveMessage($sub);
io.printf("[%s] %s\n", $msg.channel, $msg.payload);
```

A `Message` has `kind` (`"message"` for a channel push, `"pmessage"` for a pattern push), `channel` , `pattern` (`""` for a plain message), and `payload` . `receiveMessage` drains the interleaved `subscribe` / `unsubscribe` confirmation frames itself and returns only real `message` / `pmessage` pushes; a server error reply or a closed connection throws a catchable `Error` (kind `"redis"`).

A subscribed connection is restricted. Once subscribed, the connection may only run `(P)SUBSCRIBE` / `(P)UNSUBSCRIBE` / `PING` / `QUIT` until it is fully unsubscribed - the server rejects any other command. Use a **separate** connection for `publish` and normal commands. `unsubscribe($sub, [])` (an empty list) leaves every channel; `punsubscribe($sub, [])` every pattern.

The receive loop (with spawn)

`receiveMessage` is a plain blocking call - there are no handler callbacks. The program opts into concurrency by wrapping the loop in its own `spawn` , so it can keep working while messages arrive:

```
def worker as task of null init spawn {
    while (true) {
        def m as redis.Message init redis.receiveMessage($sub);
        handle($m.channel, $m.payload);
    }
};
```

The per-read idle timeout (`Session.timeout` , milliseconds) bounds each wait: lower it for a tighter poll and catch the `Error` a stalled read raises, or set it to `0` to block indefinitely.

Pipelining

`pipeline` sends several commands in one write and reads exactly one reply per command, so N commands cost a single network round trip instead of N:

```
def replies as list of redis.Reply init redis.pipeline($db, [
    ["SET", "a", "1"],
    ["INCR", "a"],
    ["GET", "a"],
]);
io.printf("a is now %s\n", $replies[2].str);   # 2
```

Unlike `command` , `pipeline` does **not** throw on a `-ERR` reply - a pipeline can mix succeeding and failing commands, so each command's outcome is returned in order and the caller inspects each `Reply` (an error reply has `kind == "error"`).

Transactions

`multi` / `exec` / `discard` wrap `MULTI` / `EXEC` / `DISCARD` . Commands issued between `multi` and `exec` are **queued** (each replies `+QUEUED`); `exec` runs them atomically and returns one `Reply` per queued command, in order:

```
redis.multi($db);
redis.command($db, ["SET", "counter", "10"]); # +QUEUED
redis.command($db, ["INCR", "counter"]);      # +QUEUED
def results as list of redis.Reply init redis.exec($db);
io.printf("counter is %d\n", $results[1].num);   # 11
```

An aborted transaction (a `nil` `EXEC` reply) yields an empty list. `discard` drops the queued commands without running them.

scan vs keys

`keys` returns every match in one shot, but it **blocks the whole server** while it walks the keyspace - fine for a small dev database, unsafe in production. `scan` is the cursor-based iterator: it returns a page of keys and the next cursor, so the server stays responsive. Start at cursor `0` and repeat until the returned cursor is `0` again:

```
def cursor as int init 0;
repeat {
    def page as redis.ScanResult init redis.scan($db, $cursor, "user:*", 100);
    for (def k in $page.keys) {
        io.printf("%s\n", $k);
    }
    $cursor = $page.cursor;
} until ($cursor == 0);
```

Pass `pattern ""` to skip the `MATCH` filter and `count 0` to skip the `COUNT` per-page hint (`COUNT` is advisory - a page may hold more or fewer keys, and an empty page with a non-zero cursor is normal).

Binary values: setBytes / getBytes

`get` / `set` speak **text** : RESP bulk lengths are byte counts, but the string verbs decode a value as UTF-8, which is exact for keys, JSON payloads, and counters, and **throws** on a non-UTF-8 value (strict, by design). For an arbitrary **binary** value - a serialized blob, a compressed payload, an image - use `setBytes(session, key, value)` (a `bytes` value sent byte-for-byte) and `getBytes(session, key)` (`bytes` , never decoded):

```
redis.setBytes($s, "blob", $rawBytes);             # any bytes, exact
def back as bytes init redis.getBytes($s, "blob"); # round-trips byte-for-byte
```

`getBytes` returns empty `bytes` for a missing key (use `exists` to tell that from an empty value). The typed hash / list / set helpers are string-valued; for a binary field / element / member, build the command with `command` and the same byte-encoding the `setBytes` path uses, or store the blob at a plain key.

Testing

The pure protocol logic - the RESP command encoder (text and the byte-exact `encodeCommandBytes`) and the simple-string / error / integer / bulk / nil / array decoder (including the incomplete-buffer and leftover-buffer cases), the pub/sub command builders, the `message` / `pmessage` push classifier, the pipeline encoder, and the SCAN reply reader - is unit-tested in the overlay (`modules/redis_test.j`). The networked session, pub/sub delivery, pipelining, transactions, SCAN, the binary `setBytes` / `getBytes` round-trip, and the typed hash / list / set helpers are covered end to end by in-process RESP servers in the Go test suite (`TestRedisCommands` , `TestRedisBinaryAndTyped`), so they run in CI without a Redis install.

Testing

The pure protocol logic - the RESP command encoder and the simple-string / error / integer / bulk / nil / array decoder (including the incomplete-buffer and leftover-buffer cases), the pub/sub command builders, the `message` / `pmessage` push classifier, the pipeline encoder, and the SCAN reply reader - is unit-tested in the overlay (`modules/redis_test.j`). The networked session, pub/sub delivery, pipelining, transactions, and SCAN are covered end to end by an in-process RESP server in the Go test suite (`TestRedisCommands`), so it runs in CI without a Redis install.

Out of scope

- **A working subset** , not the full command set: strings, counters, keys, pub/sub, pipelining, transactions, and SCAN, plus the generic `command` for the rest. Lists / hashes / sets are reachable through `command` ; typed helpers for them can follow.
- **RESP2 only, no RESP3.** Pub/sub pushes are read from the same connection as replies (the RESP2 model), not over a RESP3 out-of-band push channel.
- **No connection pool.** One `Session` is one connection; a subscribed connection needs its own session, separate from the one issuing `publish` and ordinary commands.
- **rediss TLS** rides `net` 's default certificate verification.

Timeouts and limits

The initial connect is bounded by a connection-establishment timeout, so a slow or unreachable server fails the dial instead of blocking it forever. A single reply is also capped at **64 MiB** : a malicious or compromised server that declares an enormous bulk-string length (or never terminates a reply) fails with a catchable error rather than growing the read buffer without bound.

Every read carries an idle timeout (default 30 s) so a hung server fails with a catchable error instead of blocking the caller forever. `connect` sets `Session.timeout` (milliseconds); lower it for a tighter bound, or set it to `0` to disable:

```
def s as redis.Session init redis.connect($opts);
$s.timeout = 5000;   # fail a read that stalls for 5 s
```

See also

- json.md - the same accessor-walked-reply shape.
- net.md - the transport `redis` builds on.
- modules/index.md - the module catalog and import rules.

resque - background jobs on Redis

Import with `import "resque.j" as resque;` . Schedule background jobs onto named queues now and process them from a worker later, over the `redis` module. Deliberately **Resque wire-compatible** : queues are Redis lists at `resque:queue:NAME` , the queue registry is a set at `resque:queues` , and a job is the JSON envelope `{"class":"WorkerName","args":[...]}` . Because that layout is the de-facto Resque standard, a job Jennifer enqueues can be processed by a Ruby-resque / php-resque worker and vice versa. Built on `redis` (which uses `net`), so this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "resque.j" as resque;
import "redis.j" as redis;

def db as redis.Session init redis.connect(redis.Options{host: "127.0.0.1",
    port: 6379, security: "none", user: "", password: "", db: 0});

# producer: schedule a job
resque.enqueue($db, "email", "SendWelcome", ["user@example.com", "en"]);
```

Runnable: `examples/modules/resque_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/resque_demo.j).

Surface

The module works over an existing `redis.Session` - it adds no transport of its own.

Call / type	Notes
resque.Job	A reserved job: queue, class, args (list of string).
resque.enqueue(session, queue, class, args)	Register queue and push a job (class + string args) onto it, in one pipelined round trip (SADD + RPUSH, as Ruby-resque does).
resque.reserve(session, queues)	Pop the next job from the first non-empty queue (priority order), non-blocking; an empty Job when all are drained.
resque.reserveBlocking(session, queues, timeoutSec)	Like reserve, but blocks on the queues (Redis BLPOP) until one has work or timeoutSec elapses (0 = block forever); an empty Job on timeout.
resque.queueLength(session, queue)	Pending jobs on one queue.
resque.queues(session)	Registered queue names (list of string).
resque.size(session)	Total pending jobs across every queue (one pipelined batch of LLENs).
resque.fail(session, job, message)	Record a failed job on the failed list.

`args` is a `list of string` - it maps to Ruby-resque's *positional* arguments (`perform(a, b)`). A job enqueued elsewhere with a numeric or boolean arg still reserves cleanly (each arg is read back as its string form).

Producer and worker

The **producer** is one call. The **worker** is your loop: `reserve` , then dispatch on the `class` string. A Jennifer module can't call a method by a name computed at runtime, so the module hands you the decoded `Job` and you branch - the same class-lookup a Ruby worker does under the hood:

```
use io;
import "resque.j" as resque;

def db as redis.Session init resque... ;   # a redis.Session
while (true) {
    def job as resque.Job init resque.reserve($db, ["high", "email"]);
    if (len($job.class) == 0) {
        # every queue drained; sleep and poll again (blocking BLPOP is a later
add)
        break;
    }
    try {
        if ($job.class == "SendWelcome") {
            io.printf("welcome -> %s\n", $job.args[0]);
        } elseif ($job.class == "Ping") {
            io.printf("pong\n");
        } else {
            resque.fail($db, $job, "unknown class");
        }
    } catch (e) {
        resque.fail($db, $job, $e.message);
    }
}
```

`reserve` checks the queues in the order you pass, so put higher-priority queue names first. Within one queue jobs are FIFO. For a worker that should sleep rather than spin when the queues are empty, use `reserveBlocking` , which parks on the socket (Redis `BLPOP`) until a job arrives or `timeoutSec` elapses; it widens the session's read deadline for the duration of that one call so the client does not give up before the server does.

Compatibility notes

Two behaviours are inherent to Resque, not added here:

- **The class is resolved on the worker's side.** `enqueue` only ships the string; the runtime that pops the job must define a job by that name. So a Ruby worker runs a Jennifer-enqueued job only when its codebase has that class.
- **The namespace must match.** Keys use the `resque:` prefix (the Resque default); both ends must agree.

For the **php-resque** ecosystem, args follow a single-hash convention (`args: [{...}]` , plus `id` / `queue_time` envelope fields) rather than Ruby's positional array; this module emits the positional Ruby form.

Out of scope

Basics first - these are deferred to a later pass:

- **Full Resque failure records.** `fail` writes a simplified entry, not the complete `failed_at` / `exception` / `backtrace` / `worker` shape.
- **Scheduled / delayed jobs and retries.**
- **A configurable namespace** (fixed to `resque:`).

See also

- redis.md - the client this module runs on.
- json.md - the envelope encode / decode.
- modules/index.md - the module catalog and import rules.

rest - an ergonomic REST client

Import with `import "rest.j" as rest;` . A REST convenience layer over the `http` client and `json` : hold a value-semantic `Client` (base URL + default headers) and call JSON-aware verbs. It is **pure composition** - base-URL joining, query strings, `Content-Type` , and auth headers are string / map work; the transport (verbs, TLS, framing) is `http` 's and the bodies are `json` 's. Because it builds on `http` (which uses `net`), this module needs the default **jennifer** binary.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "rest.j" as rest;
use json;

def api as rest.Client init rest.withHeader(rest.client("https://api.example.
com"),
    "Authorization", rest.bearer("my-token"));

def user as json.Value init rest.getJson($api, "/users/1", {});
def created as rest.Response init rest.postJson($api, "/users",
    json.decode("{\"name\":\"ada\"}"));
io.printf("created -> %d\n", $created.status);
```

Runnable: `examples/modules/rest_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/rest_demo.j).

Surface

The `Client` is value-semantic: pass it to each call, auth lives in its `headers` . Every verb returns a `rest.Response` (`status` , lowercased `headers` , `body`), except the `*Json` reads which decode the body to a `json.Value` .

Call / type	Notes
rest.Client	baseUrl, default headers, and an http.Options request policy (options) applied to every request.
rest.Response	status, headers, body.
rest.client(baseUrl)	Build a client: no default headers, full TLS verification.
rest.get(c, path, query)	GET; query is a map of string to string ({} for none).
rest.delete(c, path, query)	DELETE.
rest.post(c, path, contentType, body)	POST with a raw body.
rest.put(c, path, contentType, body)	PUT with a raw body.
rest.patch(c, path, contentType, body)	PATCH with a raw body.
rest.getJson(c, path, query)	GET, decode the body -> json.Value.
rest.postJson(c, path, body)	POST a json.Value (encodes, sets Content-Type); -> Response.
rest.putJson(c, path, body)	PUT a json.Value.
rest.patchJson(c, path, body)	PATCH a json.Value.
rest.bearer(token)	An Authorization value: Bearer <token>.
rest.basic(user, pass)	An Authorization value: Basic <base64(user:pass)>.
rest.withHeader(c, name, value)	A copy of the client with one default header set.

rest.withCA(c, pem)	A copy trusting a private-CA / self-signed PEM cert (bytes) for https://. The safer TLS opt-out: the server is still authenticated.
rest.insecure(c)	A copy that skips TLS certificate verification (accepts any cert). Disables authentication - trusted-LAN endpoints only; prefer withCA.
rest.withTimeout(c, timeoutMs)	A copy with a per-request idle timeout (ms; 0 = 30s default).
rest.withRedirects(c, maxRedirects)	A copy that follows up to maxRedirects 3xx (0 = none). Inherits http.send's rules.
rest.withRetries(c, maxRetries)	A copy that retries a 429 / 5xx up to maxRetries with backoff.
rest.withBackoff(c, backoffMs)	A copy with a base retry backoff (ms; 0 = 250ms default).
rest.paginate(c, path, query, maxPages)	Walk a Link-header collection (rel="next"); returns list of json.Value (one per page).
rest.paginateCursor(c, path, query, cursorPointer, cursorParam, maxPages)	Walk a cursor collection: read the next cursor at cursorPointer, resend it as cursorParam; returns list of json.Value.

URLs, queries, and auth

- **Base-URL joining** puts exactly one slash between `baseUrl` and `path` , so `"https://api"` + `"/users"` and `"https://api/"` + `"users"` both give `https://api/users` - no double slashes.
- **Query strings** are built from a `map of string to string` and percent-encoded (`{"q": "a b"}` -> `?q=a%20b`).
- **Auth** is a header: set `Client.headers["Authorization"]` to `rest.bearer(token)` or `rest.basic(user, pass)` when building the client, or add it later with `rest.withHeader` . Basic base64-encodes `user:pass` through `encoding` .

TLS (self-signed / private CA)

Every `https://` request full-verifies the server certificate by default. A LAN appliance (Proxmox, Synology, an internal service) usually ships a self-signed cert, which that default rejects. Two per-client opt-outs, both returning a copy (value semantics, like `withHeader`):

- **rest.withCA(client, pem)** trusts a specific PEM certificate (`bytes` , e.g. from `fs.readBytes`) in addition to the system roots. **Preferred** - the server is still authenticated, just against the appliance's own CA.
- **rest.insecure(client)** accepts *any* certificate. This disables server authentication and exposes the connection to a man-in-the-middle; use it only for a trusted LAN endpoint you cannot give a proper CA.

```
use fs;
def ca as bytes init fs.readBytes("appliance-ca.pem");
def api as rest.Client init rest.withCA(rest.client("https://192.168.1.10"), $ca);
# or, last resort on a trusted LAN:
def loose as rest.Client init rest.insecure(rest.client("https://192.168.1.10"));
```

Request policy (redirects, retries, timeout)

A `Client` carries an `http.Options` request policy applied to every call, configured by copy-returning builders (like `withHeader`):

```
def api as rest.Client init rest.withRetries(
    rest.withRedirects(rest.withTimeout(rest.client("https://api.example.com"),
5000), 5), 3);
```

- `rest.withTimeout(c, ms)` - per-request idle timeout.
- `rest.withRedirects(c, n)` - follow up to n 3xx redirects (inherits `http.send` 's method rules; `0` returns the 3xx).
- `rest.withRetries(c, n)` / `rest.withBackoff(c, ms)` - retry a 429 / 5xx n times with exponential backoff (honouring a numeric `Retry-After`).

Use the builders rather than reaching into `$c.options.*` directly - the nested `http.Options` type is only in scope for a program that also imports `http` .

Pagination

Two walkers fetch every page of a collection and return the list of decoded page bodies (`list of json.Value`), each capped by `maxPages` :

```
# Link-header (GitHub / GitLab): follows Link: <url>; rel="next"
def pages as list of json.Value init rest.paginate($api, "/repos/x/y/issues",
{"state": "open"}, 20);

# cursor: reads the next cursor from the body, resends it as a query param
def all as list of json.Value init rest.paginateCursor($api, "/v2/records", {}, "/
meta/next_cursor", "cursor", 50);
```

`paginate` follows each response's `Link` header until there is no `rel="next"` ; `paginateCursor` reads the cursor at a JSON Pointer (string or integer) and stops when it is absent, null, or empty. Both throw `kind "rest"` on a non-2xx page.

Errors

A non-2xx **status is a value, not a crash** : a 404 or 500 comes back as a normal `Response` with that `status` for you to branch on. `getJson` (and the pagination walkers) will, however, `throw` `kind "rest"` on a non-2xx or a body that is not valid JSON (e.g. an HTML error page) - guard with `get` + a status check when a server may return non-JSON errors.

Out of scope

- **No stateful keep-alive session.** A `Client` is a value threaded per call; connection reuse across calls lives on the `http` side (`http.connect` / `exchange`). `rest` inherits per-request policy (redirects, retries, timeout) and per-redirect-chain cookies from `http.send` , but keeps no cookies across separate `rest` calls.

See also

- http.md - the client `rest` composes over.
- json.md - the `json.Value` request / response bodies.
- modules/index.md - the module catalog and import rules.

ringbuffer - fixed-capacity ring buffer

Import with `import "ringbuffer.j" as ringbuffer;` . A fixed-capacity FIFO of strings that **overwrites the oldest entry when full** - a sliding window of the most recent items (log lines, events, samples). Pure `.j` over `lists` ; runs on both binaries.

```
import "ringbuffer.j" as ringbuffer;

def rb as ringbuffer.RingBuffer init ringbuffer.new(3);
$rb = ringbuffer.push($rb, "a");
$rb = ringbuffer.push($rb, "b");
$rb = ringbuffer.push($rb, "c");
$rb = ringbuffer.push($rb, "d");   # overwrites "a"; buffer is now [b, c, d]
ringbuffer.first($rb);             # "b" (oldest)
$rb = ringbuffer.pop($rb);         # buffer is now [c, d]
```

Runnable: `examples/modules/ringbuffer_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/ringbuffer_demo.j).

Surface

```
def struct ringbuffer.RingBuffer { items as list of string, capacity as int };
```

Call	Returns	
ringbuffer.new(capacity)	RingBuffer	an empty buffer of the given capacity (>= 1)
ringbuffer.push(rb, item)	RingBuffer	append, dropping the oldest if already full
ringbuffer.pop(rb)	RingBuffer	remove the oldest (throws if empty)
ringbuffer.first(rb)	string	peek the oldest (throws if empty)
ringbuffer.last(rb)	string	peek the newest (throws if empty)
ringbuffer.size(rb)	int	the number of entries held
ringbuffer.capacity(rb)	int	the capacity
ringbuffer.isEmpty(rb)	bool	whether it holds no entries
ringbuffer.isFull(rb)	bool	whether it is at capacity
ringbuffer.toList(rb)	list of string	a copy of the entries, oldest to newest

Reading while removing

Because the buffer is **value-semantic** , a `pop` cannot return both the removed item and the new buffer. Read the oldest with `first` *before* you `pop` it:

```
repeat {
    if (ringbuffer.isEmpty($rb)) {
        # done
    } else {
        def oldest as string init ringbuffer.first($rb);
        # ... use $oldest ...
        $rb = ringbuffer.pop($rb);
    }
} until (ringbuffer.isEmpty($rb));
```

Scope

- **Strings only.** Store other values through `convert.toString` or a `json` / `encoding` representation.
- **Value-semantic.** `push` / `pop` return a fresh buffer; assign the result back (`$rb = ringbuffer.push($rb,`

`x`). The original is unchanged.

- **Overwrite on full is silent.** `push` drops the oldest when at capacity with no signal - check `isFull` first if you need to know.

See also

- lists.md - the list operations underneath.
- bloom.md - the sibling data-structure module.
- modules/index.md - the module catalog and import rules.

s3 - S3-compatible object storage

Import with `import "s3.j" as s3;` . An **object-storage client** for Amazon S3 and every S3-compatible store - the endpoint is configurable, so **one module** serves AWS S3, MinIO, Cloudflare R2, and Backblaze B2 (a selectable backend, not a module per vendor). Every request is signed with **AWS Signature Version 4** (HMAC-SHA256 key-chaining), built on `hash.hmac` + `hash.compute` + `encoding` (hex) + `time` (the request timestamp) + `http` . Needs the default **jennifer** binary (`net` via `http`).

```
import "s3.j" as s3;
import "http.j" as http;

def c as s3.Client init s3.connect(
    "https://s3.us-east-1.amazonaws.com", "us-east-1", accessKey, secretKey);

def put as http.Response init s3.put($c, "mybucket", "hello.txt", "hi there");
def obj as http.Response init s3.get($c, "mybucket", "hello.txt");
io.printf("%d\n%s\n", $obj.status, $obj.body);
```

Runnable: `examples/modules/s3_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/s3_demo.j).

Client

`s3.connect(endpoint, region, accessKey, secretKey)` returns a value-semantic `s3.Client` . The `endpoint` is any S3-compatible base URL (`scheme://host` , no trailing slash); addressing is **path-style** (`{endpoint}/{bucket}/{key}`), which works uniformly across AWS and self-hosted stores.

Every request carries a **timeout** so a hung S3 endpoint fails instead of blocking forever (the classic way a slow store exhausts a worker pool). `connect` defaults `Client.timeout` to 30 000 ms; set it to change the bound, or to `0` to disable it:

```
def c as s3.Client init s3.connect(endpoint, region, key, secret);
$c.timeout = 5000;   # fail a request that stalls for 5 s
```

Store	endpoint	region
AWS S3	https://s3.<region>.amazonaws.com	your bucket's region
MinIO	http://host:9000	us-east-1 (or as configured)
Cloudflare R2	https://<account>.r2.cloudflarestorage.com	auto
Backblaze B2	https://s3.<region>.backblazeb2.com	your bucket's region

Operations

Every call returns an `http.Response` (`status` / `headers` / `body`); reading it needs `import "http.j"` . A non-2xx status is a value to branch on, not an error.

Call	Method	Notes
s3.get(client, bucket, key)	GET	body is the object contents (string); a missing object is a 404.
s3.getBytes(client, bucket, key)	GET	Byte-safe download -> http.BytesResponse (body as bytes); use for binary objects.
s3.put(client, bucket, key, body)	PUT	Upload / overwrite from a string body; 200 on success.

s3.putBytes(client, bucket, key, data)	PUT	Upload from a raw bytes body (byte-for-byte).
s3.putWith(client, bucket, key, body, contentType, metadata)	PUT	Upload with a signed Content-Type + x-amz-meta-* metadata ({} = none).
s3.putBytesWith(client, bucket, key, data, contentType, metadata)	PUT	The bytes-body counterpart to putWith.
s3.head(client, bucket, key)	HEAD	Object metadata (status + headers, empty body); 404 when absent.
s3.copy(client, srcBucket, srcKey, dstBucket, dstKey)	PUT	Server-side copy via a signed x-amz-copy-source (no download / re-upload).
s3.delete(client, bucket, key)	DELETE	204 on success.
s3.listObjects(client, bucket)	GET ?list-type=2	body is the ListObjectsV2 XML (first page, up to 1000 keys).
s3.listObjectsFrom(client, bucket, token)	GET ?list-type=2&continuation-token=...	Fetch a further page from a continuation token.
s3.objectKeys(xml)	-	Pull the <Key> values out of a list body -> list of string.
s3.isTruncated(xml)	-	true when the listing has more pages.
s3.nextContinuationToken(xml)	-	The token for the next page, or "" when complete.

(The list op is `listObjects` , not `list` , because `list` is a reserved type keyword.)

```
# Page through every object (S3 caps a listing at 1000 keys per page).
def body as string init s3.listObjects($c, "mybucket").body;
repeat {
    for (def k in s3.objectKeys($body)) {
        io.printf("%s\n", $k);
    }
    if (s3.isTruncated($body)) {
        $body = s3.listObjectsFrom($c, "mybucket", s3.
nextContinuationToken($body)).body;
    }
} until (not s3.isTruncated($body));
```

Presigned URLs

`s3.presign(client, method, bucket, key, expiresSeconds)` builds a URL that grants time-limited access to an object without exposing the secret key, using SigV4 **query-signing** (the `X-Amz-*` query parameters, `UNSIGNED-PAYLOAD`). The returned URL works from any HTTP client (a browser, `curl`) until it expires. This is pure string work - no request is sent - so it runs on **both binaries** .

```
# A link that lets anyone GET the object for one hour.
def url as string init s3.presign($c, "GET", "mybucket", "report.pdf", 3600);
# A PUT upload link (hand it to a browser to upload directly to S3).
def up as string init s3.presign($c, "PUT", "mybucket", "upload.bin", 900);
```

The maximum validity is 604800 seconds (7 days).

Multipart upload

For objects beyond the ~5 GB single- PUT limit (or to stream parts as they are produced), upload in parts:

```
def uid as string init s3.createMultipartUpload($c, "mybucket", "big.iso",
"application/octet-stream");
def etags as list of string init [];
$etags[] = s3.uploadPart($c, "mybucket", "big.iso", $uid, 1, $part1); # each part
>= 5 MiB except the last
$etags[] = s3.uploadPart($c, "mybucket", "big.iso", $uid, 2, $part2);
def r as http.Response init s3.completeMultipartUpload($c, "mybucket", "big.iso",
$uid, $etags);
# On any failure, discard the parts so they do not accrue storage cost:
#   s3.abortMultipartUpload($c, "mybucket", "big.iso", $uid);
```

`uploadPart` returns each part's ETag; pass them to `completeMultipartUpload` in part-number order (part 1 first).

Signing

Requests are signed with SigV4 for service `s3` : the canonical request covers the method, the URI-encoded path (object keys keep their `/`), the canonical query (parameters sorted by key), the signed headers, and the SHA-256 of the body; the string-to-sign is HMAC-chained through `AWS4<secret>` -> `date` -> `region` -> `s3` -> `aws4_request` to the signature. The base signed set is `host` / `x-amz-content-sha256` / `x-amz-date` ; `putWith` / `putBytesWith` add `content-type` and each `x-amz-meta-*` header to the signed set, and `copy` adds `x-amz-copy-source` , all sorted in. The payload hash is a real SHA-256 of the body (not `UNSIGNED-PAYLOAD`) for the request-signing path, so the whole request is integrity-covered; presigned URLs use `UNSIGNED-PAYLOAD` (the standard for query-signing). The signature is pinned in the tests against independent SigV4 implementations - the header-signing over the wire (`TestS3Requests` , which re-derives the signature from the actual signed-header set) and the presigned URL against a Python reference.

Scope

- **Path-style, SigV4, s3 service.** Virtual-hosted addressing and the older SigV2 are out of scope.
- **String and byte bodies.** `put` / `get` carry text; `putBytes` / `getBytes` (and `putBytesWith`) carry raw `bytes` for binary objects. A single body is held in memory, so objects beyond ~5 GB use the multipart path.
- **Object ops + multipart + presign.** Get / put / delete / head / copy, content-type + metadata, multipart upload, and presigned URLs are covered. Bucket create / delete and ACL / policy management are not.
- **listObjects returns the raw XML** (plus `objectKeys`); pagination is supported through `isTruncated` / `nextContinuationToken` / `listObjectsFrom` , but full metadata parsing (size, etag, last-modified) is a follow-on.

See also

- hash.md - the `hmac` / `compute` primitives SigV4 builds on.
- http.md - the client transport requests go through.
- webhook.md / totp.md - the other `hash.hmac` -based modules.
- modules/index.md - the module catalog and import rules.

sasl - SASL authentication encoders

Import with `import "sasl.j" as sasl;` . The SASL authentication mechanisms shared by the mail clients (`smtp` / `pop` / `imap`). These format the client tokens; the protocol clients run the mechanism-specific wire dialogue around them. The simple encoders are pure base64; the challenge-response mechanisms (`cram` , SCRAM) key off the `hash` / `crypto` libraries. **No networking** , and every dependency is TinyGo-clean, so the module runs on either binary.

```
import "sasl.j" as sasl;

def p as string init sasl.plain("me@example.com", "secret");     # SASL PLAIN
def b as string init sasl.bearer("me@gmail.com", accessToken);   # SASL XOAUTH2
```

Runnable: `examples/modules/sasl_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/sasl_demo.j) (and it is exercised end to end by the mail clients' demos, e.g. `smtp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/smtp_demo.j)).

Surface

Call	Returns	Notes
sasl.plain(user, pass)	string	SASL PLAIN: base64 of "\0user\0pass".
sasl.loginUser(user)	string	SASL LOGIN step 1: base64 of the username.
sasl.loginPass(pass)	string	SASL LOGIN step 2: base64 of the password.
sasl.bearer(user, token)	string	SASL XOAUTH2: an OAuth2 bearer-token response.
sasl.cram(user, pass, challenge)	string	CRAM-MD5 (RFC 2195): one-step HMAC-MD5 response to a base64 challenge.
sasl.scramStart(user, algo)	Scram	Begin a SCRAM exchange; algo "sha1" or "sha256". Crypto-grade nonce.
sasl.scramClientFirst(s)	string	The base64 client-first message to send.
sasl.scramClientFinal(s, serverFirst, pass)	Scram	Process server-first; derive the proof + expected server signature.
sasl.scramFinalToken(s)	string	The base64 client-final message to send.
sasl.scramVerify(s, serverFinal)	bool	Verify the server's signature (constant-time); reject on false.
sasl.negotiate(advertised)	string	Pick the strongest password mechanism from a server's advertised list, as an auth token (see below).

Mechanism negotiation

`sasl.negotiate(advertised)` takes the mechanism names a server advertises (a `list of string` , any case) and returns the strongest one this module supports as a mail-client `auth` token: `"scram-sha-256"` > `"scram-sha-1"` > `"cram"` , or `""` when the server offers none (the caller then uses its protocol default - PLAIN / LOGIN / USER-PASS). XOAUTH2 is never auto-selected (it needs a bearer token, not a password). The mail clients call this when `Options.auth` is `"auto"` , feeding it the mechanisms parsed from the SMTP `EHLO` , POP3 `CAPA` , or IMAP `CAPABILITY` advertisement (`""` keeps their plain default).

SCRAM (RFC 5802 / RFC 7677)

SCRAM is a salted challenge-response: the password never crosses the wire, and both sides prove knowledge of it. The exchange threads a value-semantic `Scram` handle through four calls (each returns an updated copy), so a client drives it against the SASL `AUTHENTICATE` / `AUTH` loop:

```
def s as sasl.Scram init sasl.scramStart("user", "sha256");
# 1. send sasl.scramClientFirst($s); read the server-first reply
$s = sasl.scramClientFinal($s, serverFirst, "pencil");
# 2. send sasl.scramFinalToken($s); read the server-final reply
if (not sasl.scramVerify($s, serverFinal)) {
    # the server does not know the password - abort, do not trust the session
}
```

`scramClientFinal` runs PBKDF2 in the mechanism's hash (`crypto.pbkdf2` with `"sha1"` / `"sha256"`), so SCRAM-SHA-1 (MongoDB, XMPP) and SCRAM-SHA-256 (PostgreSQL, modern mail) both work. `scramVerify` compares the server signature with `crypto.hmacEqual` (constant-time). The wire strings are the base64 SASL tokens; the raw SCRAM messages are inside.

`sasl.cram(user, pass, challenge)` is the simpler one-step CRAM-MD5: the server sends a base64 challenge, the client answers `base64("user " + hex(HMAC-MD5))` . It is named `cram` , not `cramMd5` , for the letters-only method-name rule.

XOAUTH2 (the "use a token" half of OAuth2)

`sasl.bearer(user, token)` builds the SASL **XOAUTH2** initial response - `base64("user=" user 0x01 "auth=Bearer " token 0x01 0x01)` - which is how **Google** and **Microsoft 365** authenticate mail now that both have retired password auth. The mail clients accept it via `Options.auth` = `"xoauth2"` (with the access token in `pass`):

```
def opts as smtp.Options init smtp.Options{host: "smtp.gmail.com", port: 587,
    security: "starttls", clientName: "me.example", user: "me@gmail.com",
    pass: accessToken, auth: "xoauth2"};
smtp.send($opts, "me@gmail.com", ["you@example.com"], $message);
```

The function is named `bearer` , not `xoauth2` , because a Jennifer method name is letters-only (no digit); the wire mechanism name `"XOAUTH2"` is a string the client sends. Getting the **token itself** is the other half of OAuth2 - the job of the generic `oauth` client (planned, M18.7.3).

Out of scope

- **Encoders, not transport.** `sasl` formats tokens and drives the SCRAM state machine; the SMTP `AUTH` , IMAP `AUTHENTICATE` , and POP3 `AUTH` line dialogue lives in the respective clients.
- **No SASLprep.** Usernames / passwords are used as given (only SCRAM's = / `,` username escaping is applied); normalize non-ASCII credentials yourself.

See also

- smtp.md / pop.md / imap.md - the mail clients that consume these encoders.
- crypto.md - the KDF / constant-time compare SCRAM builds on; encoding.md - the base64 codec.
- modules/index.md - the module catalog and import rules.

screen - terminal user interfaces

Import with `import "screen.j" as screen;` . An *explicit* terminal UI - a cell buffer you draw into and paint to the terminal, plus a key-event decoder and event loop - not a GUI framework. Two layers:

- **Output-only** (no host capability, runs on both binaries): ANSI control strings, a `Buffer` of character cells, and a `render` / `diff` update loop for live dashboards, progress bars, and self-updating tables. Pure strings.
- **Interactive** (needs the `term` library, so the default `jennifer`): decode a raw byte stream into named `Key` events and read them in a loop over raw mode - menus, forms, key navigation. On `jennifer-tiny` these raise the `term` friendly-error stub.

Coordinates are **0-based** , origin at the top-left `(0, 0)` ; x is the column, y the row. Drawing that runs past an edge is clipped, never an error.

```
use io;
import "screen.j" as screen;

def buf as screen.Buffer init screen.newScreen(3, 20);
def out as screen.Buffer init screen.text(screen.box($buf, 0, 0, 20, 3), 2, 1,
"hello");
io.printf("%s%s", screen.clear(), screen.render($out));       # draw it
```

Runnable: `examples/modules/screen_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/screen_demo.j).

The cell buffer

`screen.Buffer { rows as int, cols as int, cells as list of string }` is a row-major grid; each cell is a one-column string (optionally wrapped in an SGR colour). It is value-semantic like any struct, so every drawing call returns a **fresh** buffer - assign it back (`$buf = screen.text($buf, ...);`).

Call	Returns	Notes
screen.newScreen(rows, cols)	screen.Buffer	A blank buffer, every cell a space. Non-positive dimensions error.
screen.set(buf, x, y, cell)	screen.Buffer	Set one cell (one column). Out-of-range is a no-op (clipped).
screen.get(buf, x, y)	string	The cell contents, or "" out of range.
screen.text(buf, x, y, s)	screen.Buffer	Write s one rune per cell from (x, y), clipped at the row's edge.
screen.textColor(buf, x, y, s, color)	screen.Buffer	Same, in a named foreground colour. Unknown colour errors.
screen.fill(buf, x, y, w, h, ch)	screen.Buffer	Fill a w-by-h rectangle with the glyph ch.
screen.hline(buf, x, y, n, ch)	screen.Buffer	A horizontal run of n cells of ch.
screen.vline(buf, x, y, n, ch)	screen.Buffer	A vertical run of n cells of ch.
screen.box(buf, x, y, w, h)	screen.Buffer	A single-line box border (Unicode box-drawing), interior untouched.
screen.clearBuffer(buf)	screen.Buffer	Reset every cell to a space (keeps the dimensions).

Colour names for `textColor` : `black` , `red` , `green` , `yellow` , `blue` , `magenta` , `cyan` , `white` , `gray` / `grey` . Each coloured cell carries its own SGR reset, so it still occupies exactly one column and the diff stays correct.

For efficiency, prefer `text` / `box` / `fill` (each copies the buffer once) over a per-cell `set` in a loop (each copies the whole buffer, so a loop is O(n^2)).

Painting: render and diff

Call	Returns	Notes
screen.render(buf)	string	The escape string that paints the whole buffer. Print it with io.printf.
screen.diff(old, new)	string	The minimal escape string that turns old into new - only changed cells, in row runs. Falls back to a full render(new) when the sizes differ.

The update loop keeps the previously-shown buffer and diffs the next one against it, so only the cells that changed are repositioned and rewritten - no flicker, no full repaint:

```
use io;
import "screen.j" as screen;

def prev as screen.Buffer init frame(0);
io.printf("%s%s", screen.clear(), screen.render($prev));
for (def i as int init 1; $i <= 100; $i = $i + 1) {
    def next as screen.Buffer init frame($i);
    io.printf("%s", screen.diff($prev, $next));       # only the deltas
    $prev = $next;
}
```

ANSI control sequences

Each returns the escape string; the caller prints it. All pure, so they work on both binaries.

Call	Effect
screen.clear()	Clear the whole screen and home the cursor.
screen.clearLine()	Clear the current line.
screen.moveTo(x, y)	Move the cursor to (x, y) (0-based).
screen.up(n) / down(n) / left(n) / right(n)	Move the cursor n cells.
screen.home()	Move the cursor to the top-left.
screen.hideCursor() / showCursor()	Hide / show the text cursor.
screen.enterAlt() / exitAlt()	Switch to / from the alternate screen buffer (so the app does not scroll the shell's scrollback).

Interactive input (needs term)

The key decoder is a pure function so it is testable without a terminal; the event loop reads through the `term` library.

Call	Returns	Notes
screen.decodeKey(seq)	screen.Key	Decode one raw byte sequence (list of int) into a key. Pure and total.
screen.nextKey()	screen.Key	Read and decode the next key from the terminal (blocking).
screen.begin()	term.State	Enter raw mode + alternate screen, hide the cursor, clear. Pair with end.
screen.end(state)	null	Show the cursor, leave the alternate screen, restore the terminal.
screen.size()	term.Size	The terminal dimensions ({rows, cols}), a passthrough to term.size.

`screen.Key { name as string, char as string }` . `name` is symbolic: a printable key is `"char"` (the

character in `char`); the rest are named - `"up"` / `"down"` / `"left"` / `"right"`, `"enter"` / `"tab"` / `"escape"` / `"backspace"` / `"delete"` / `"insert"`, `"home"` / `"end"` / `"pageup"` / `"pagedown"`, `"f1"` .. `"f12"`, `"ctrl-a"` .. `"ctrl-z"`, `"alt-<c>"`, `"eof"` at end of input, and `"unknown"`.

```
use io;
import "screen.j" as screen;

def state as term.State init screen.begin();
defer screen.end($state);                # restored on every exit path
repeat {
    def key as screen.Key init screen.nextKey();
    if ($key.name == "ctrl-c" or $key.name == "char" and $key.char == "q") {
        break;
    }
    # ... redraw based on $key ...
} until (false);
```

A lone **Escape** press is only reported once the next byte arrives: `term.readByte` has no timeout to tell a bare ESC from the start of an escape sequence. Prefer a named key (or `Ctrl-C`) to quit a loop. Multi-byte UTF-8 input is decoded byte-first; full rune assembly is a planned follow-on.

Layering and platforms

The output-only layer is pure Jennifer over ANSI strings and runs on either binary. The interactive layer's `nextKey` / `begin` / `end` / `size` call `term`, which is real on the default `jennifer` and a friendly-error stub on `jennifer-tiny`. Colour output should still gate on a real terminal - use `ansi`'s `enabled` check or `os.isTerminal` before entering full-screen mode.

semver - Semantic Versioning 2.0.0

Import with `import "semver.j" as semver;` . Parses, compares, sorts, increments, and **range-matches** strict SemVer 2.0.0 (https://semver.org) version strings - the full surface a package registry or dependency resolver needs. Pure Jennifer - parsing uses the canonical SemVer regex (`regex`); the precedence comparison, sort, and range matching are hand-written - so it runs on either binary.

```
use io;
import "semver.j" as semver;

def v as semver.Version init semver.parse("1.4.2-rc.1+build.9");
io.printf("%d.%d.%d pre=%s\n", $v.major, $v.minor, $v.patch, $v.prerelease);
io.printf("rc < release: %t\n", semver.lt($v, semver.parse("1.4.2"))); # true
io.printf("next minor: %s\n", semver.toString(semver.incMinor($v)));   # 1.5.0
```

Runnable: `examples/modules/semver_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/semver_demo.j).

The Version struct

```
export def struct Version {
    major as int,
    minor as int,
    patch as int,
    prerelease as string,   # "" when absent (the text after `-`, no dash)
    build as string         # "" when absent (the text after `+`, no plus)
};
```

A value-semantic struct you hold and pass around; `semver` stores nothing. Build it with `semver.parse` or a literal (`semver.Version{major: 1, minor: 0, patch: 0, prerelease: "", build: ""}`).

Surface

Call	Returns	Notes
semver.parse(s)	Version	Parse a strict version string; throws (kind: "value") on invalid.
semver.isValid(s)	bool	Whether s is a strict SemVer string (no throw).
semver.coerce(s)	string	Extract a version from loose text (a v-tag, a partial); "" if none.
semver.clean(s)	string	Strict-normalise (trim, drop v / =); "" if not a full version.
semver.toString(v)	string	Canonical form; round-trips parse.
semver.compare(a, b)	int	-1 / 0 / 1 by SemVer precedence. Build metadata ignored.
semver.lt(a, b)	bool	compare(a, b) < 0.
semver.eq(a, b)	bool	compare(a, b) == 0 (so 1.0.0+a equals 1.0.0+b).
semver.gt(a, b)	bool	compare(a, b) > 0.
semver.gte(a, b)	bool	compare(a, b) >= 0.
semver.lte(a, b)	bool	compare(a, b) <= 0.
semver.neq(a, b)	bool	compare(a, b) != 0.
semver.diff(a, b)	string	The change kind: "major" / "minor" / "patch" / "prerelease" / "".

semver.isStable(v)	bool	major >= 1 and no prerelease. 0.y.z is unstable by convention.
semver.isPrerelease(v)	bool	Whether a prerelease tag is present.
semver.incMajor(v)	Version	major+1; resets minor / patch and clears prerelease + build.
semver.incMinor(v)	Version	minor+1; resets patch and clears prerelease + build.
semver.incPatch(v)	Version	patch+1; clears prerelease + build.
semver.sort(vs)	list of Version	A new list ordered ascending by precedence.
semver.rsort(vs)	list of Version	A new list ordered descending (highest first).
semver.satisfies(ver, range)	bool	Whether the version string matches the range. See Ranges.
semver.maxSatisfying(vers, range)	string	Highest version string in vers matching range, or "".
semver.minSatisfying(vers, range)	string	Lowest version string in vers matching range, or "".
semver.validRange(range)	bool	Whether a range expression is well-formed.
semver.minVersion(range)	string	The lowest version that could satisfy range (its floor), or "".
semver.intersects(a, b)	bool	Whether two ranges share any satisfying version.
semver.subset(inner, outer)	bool	Whether every version in inner is also allowed by outer.
semver.gtr(v, range)	bool	Whether v is above the whole range.
semver.ltr(v, range)	bool	Whether v is below the whole range.
semver.outside(v, range)	bool	gtr(v, range) or ltr(v, range).
semver.simplifyRange(vers, range)	string	The shortest range matching the same subset of vers.

Strict, not a loose parser

`semver.parse` accepts exactly `MAJOR.MINOR.PATCH` with an optional `-prerelease` and `+build` , per the spec. It **rejects** everything the grammar disallows:

```
semver.isValid("1.2.3");        # true
semver.isValid("1.2.3-rc.1");   # true
semver.isValid("1.0.0+build");  # true
semver.isValid("1.2.3.4");      # false - four segments have no defined order
semver.isValid("1.2");          # false - too few
semver.isValid("01.0.0");       # false - leading zero in a numeric part
semver.isValid("1.0.0-01");     # false - leading zero in a numeric prerelease id
semver.isValid("1.0.0-");       # false - empty prerelease
```

A looser N-segment form (`1.2.3.4`) has no defined ordering (`1.2.3` vs `1.2.3.0` ?), which would quietly break sorting, so it is invalid rather than best-effort. Jennifer's own `meta.VERSION` is valid strict SemVer, so `semver.parse(meta.VERSION)` works out of the box.

Use `isValid` to test without a throw; `parse` for the value (wrap it in `try` / `catch` to handle untrusted input).

Precedence

`compare` implements SemVer clause 11 exactly:

1. Compare `major` , then `minor` , then `patch` numerically.
2. A version **with** a prerelease ranks **below** the same version without one: `1.0.0-alpha` < `1.0.0` .
3. Two prereleases compare field by field (split on `.`): numeric identifiers compare numerically and rank below alphanumeric ones, which compare in ASCII order; a longer run of otherwise-equal fields ranks higher.
4. **Build metadata is ignored** - `1.0.0+a` and `1.0.0+b` are equal.

```
# 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta
#   < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0
```

Note `beta.2` < `beta.11` (numeric, not lexical) - the classic loose-parser trap the field-by-field rule avoids.

Sorting a list

`lists.sort` is scalar-only, so `semver.sort` runs its own pass over `compare` and returns a new ascending list (the input is untouched - value semantics):

```
def vs as list of semver.Version init [];
$vs[] = semver.parse("2.0.0");
$vs[] = semver.parse("1.0.0-alpha");
$vs[] = semver.parse("1.0.0");
$vs[] = semver.parse("1.10.0");
$vs[] = semver.parse("1.2.0");

for (def s in semver.sort($vs)) {
    io.printf(" %s", semver.toString($s));
}
# sorted: 1.0.0-alpha 1.0.0 1.2.0 1.10.0 2.0.0
```

Ranges and constraints

`semver.satisfies(version, range)` matches a concrete version string against a range expression, following the npm / Composer grammar:

Form	Example	Means
exact	1.2.3 / =1.2.3	that version exactly
caret	^1.2.0	>=1.2.0 <2.0.0 (up to the next non-zero left component)
tilde	~1.2	>=1.2.0 <1.3.0
comparators	>=1.0.0 <2.0.0 >1.2 <=3	numeric bounds (a partial operand expands, npm-style)
AND	>=1.2.0 <2.0.0 (space or ,)	all comparators in the clause hold
OR	^1.0.0 || ^3.0.0	any clause holds
hyphen	1.2.3 - 2.3.4	>=1.2.3 <=2.3.4 (a partial upper bumps to <)
x-range	1.x / 1.2.*	>=1.0.0 <2.0.0 / >=1.2.0 <1.3.0
any	* / "" / "any"	any released version

```
semver.satisfies("1.4.0", "^1.2.0");                    # true
semver.satisfies("2.0.0", "^1.2.0");                    # false
semver.satisfies("1.9.0", ">=1.2.0 <2.0.0");            # true
semver.satisfies("3.4.0", "^1.0.0 || ^3.0.0");          # true
semver.satisfies("2.0.0", "1.2.3 - 2.3.4");             # true
```

Prereleases are excluded from ranges by default: a version like `2.0.0-rc.1` satisfies a range only when a comparator in the *same clause* pins a prerelease at the same `major.minor.patch` (the npm rule), e.g. `>=1.2.3-rc.1 <1.3.0` admits `1.2.3-rc.2` but not `1.4.0-rc.1` . An invalid version string never satisfies anything.

Selecting from a set

For a registry resolving "the best available version", `maxSatisfying` / `minSatisfying` pick the highest / lowest candidate that matches, skipping any non-SemVer entries and returning `""` when none match:

```
def tags as list of string init ["1.0.0", "1.2.0", "1.4.3", "2.0.0"];
semver.maxSatisfying($tags, "^1.2.0");   # "1.4.3"
semver.minSatisfying($tags, "^1.2.0");   # "1.2.0"
```

`semver.validRange(range)` reports whether a range expression is well-formed, without evaluating it. `semver.minVersion(range)` returns the lowest version that could satisfy a range (its floor), with no candidate list: `minVersion("^1.2.0")` is `"1.2.0"` , `minVersion(">1.2.3")` is `"1.2.4"` .

Ingesting loose versions

Real registries take messy tags. `semver.coerce(s)` extracts a version from a v -prefix, a partial, or surrounding noise (`coerce("v1.2.3")` -> `"1.2.3"` , `coerce("1.2")` -> `"1.2.0"` , `coerce("latest")` -> `""`), while `semver.clean(s)` strictly normalises a near-clean string (trim, drop a leading v / =) and returns `""` unless it is already a full version.

Range algebra

For a dependency **solver** (conflict detection, deduplication), the range-vs-range operators reason over interval sets - no candidate list needed:

Call	Question
semver.intersects(a, b)	do ranges a and b share any version? ^1.2.0 ? >=1.5.0 = true; ^1.2.0 ? ^2.0.0 = false
semver.subset(inner, outer)	is every version in inner also in outer? subset("^1.5.0", "^1.0.0") = true
semver.gtr(v, range)	is v above the whole range?
semver.ltr(v, range)	is v below the whole range?
semver.outside(v, range)	above or below (not in an interior gap)

The algebra is **prerelease-precise** : intervals carry full-version bounds with inclusivity flags plus the `major.minor.patch` tuples at which each clause admits prereleases, so `intersects(">=1.2.3-rc.1 <1.2.3", ">=1.5.0-rc.1 <1.5.0")` is `false` (prereleases at different tuples never meet) while `intersects(">=1.2.3-rc.1 <1.2.3", ">=1.2.3-rc.2 <1.2.3")` is `true` .

Simplifying against a version set

`semver.simplifyRange(versions, range)` returns the shortest range that matches the *same subset of the given versions* - runs of consecutive matches collapse to `>=lo <=hi` clauses joined by `||` , `*` when all match, `<0.0.0-0` when none do, and the original is kept when it is already at least as short:

```
def vers as list of string init ["1.0.0", "1.1.0", "1.2.0", "1.3.0", "2.0.0"];
semver.simplifyRange($vers, ">=1.0.0 <=1.0.0 || >=1.1.0 <=1.3.0");   # ">=1.0.0
<=1.3.0"
```

See also

- regex.md - `parse` matches against the canonical SemVer pattern with named groups.
- meta.md - `meta.VERSION` , itself valid SemVer.
- modules/index.md - the module catalog and import rules.

session - server-side sessions

Import with `import "session.j" as session;` . Server-side sessions over a **selectable** key/value backend: a session is a `json.Value` held under a `sess:ID` key with a sliding TTL, so it expires on its own when idle. The store is a `kvstore.Store` - back it with `memcache` or `redis` (distributed, shared across processes) or the in-process `kv` library (single process, no server), all behind one API. Session values are a `json.Value` , so a session holds **structured** data (nested objects, arrays, numbers), not just flat strings.

```
import "session.j" as session;
import "kvstore.j" as kvstore;
use json;

def store as kvstore.Store init kvstore.redisStore($rc);   # or memcacheStore /
inProcessStore / fileStore
def id as string init session.create($store, 1800);        # a 30-minute session
def data as json.Value init session.load($store, $id);
$data = json.set($data, "/user", "ada");
$data = json.set($data, "/prefs/theme", "dark");           # nested is fine
session.save($store, $id, $data, 1800);
```

Surface

Call	Returns	
session.create(store, ttl)	string	mint a new session ID; store an empty session with a ttl-second expiry
session.load(store, id)	json.Value	the session's data, or an empty object when absent / expired
session.save(store, id, data, ttl)		write the json.Value and re-arm the expiry
session.touch(store, id, ttl)	bool	re-arm the expiry without rewriting; whether it still existed
session.destroy(store, id)	bool	remove the session; whether it existed

Session IDs and storage

- **IDs are UUID v4** from the `crypto` crypto-grade random source (122 unguessable bits), so an ID is safe to hand a client as the session's bearer token. An ID arriving from a client cookie is **validated** (1 to 250 chars, `[A-Za-z0-9-]` only) before it reaches a store key, so it cannot inject a backend command.
- **Values are stored base64-wrapped JSON** , so any UTF-8 (or structured) value round-trips exactly, and the store only ever holds ASCII.

Caveats

- **Volatile.** Sessions are a cache of soft state, not a store of record: a memcache / in-process backend evicts, so treat expiry as expected (re-auth, do not lose money on a dropped session).
- **Pick the backend for your topology.** More than one process (a web fleet) must see the same session -> use `redis` / `memcache` . A single-process app can use `inProcessStore` (in memory) or `fileStore` (persisted across runs).

See also

- kvstore.md - the backend selector; kv.md - the in-process backend.
- json.md - the value type a session holds.
- modules/index.md - the module catalog and import rules.

slack - Slack Incoming Webhook client

Import with `import "slack.j" as slack;` . Post messages to a Slack channel through an Incoming Webhook (https://api.slack.com/messaging/webhooks), on top of the `http` module - a sibling of `gotify` and `discord` . Needs the default `jennifer` binary. The webhook URL is a secret: read it from the environment or a config file, never commit it.

```
import "slack.j" as slack;

slack.send("https://hooks.slack.com/services/T/B/xxx", "deploy finished");

def m as slack.Message init slack.section(
    slack.header(slack.message(), "Deploy"), "*build 1234* is live");
slack.sendMessage("https://hooks.slack.com/services/T/B/xxx", $m);
```

Runnable: `examples/modules/slack_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/slack_demo.j).

Plain messages

`slack.send(webhookUrl, text)` posts `{"text": text}` (the text is Slack mrkdwn) and returns the `http.Response` - Slack answers `200` with body `ok` on success. The webhook's configured channel receives the message.

Rich messages (Block Kit)

Build a message from Block Kit (https://api.slack.com/block-kit) blocks with value-semantic builders - each returns a fresh `Message` , so they chain - then post it with `sendMessage` (or inspect the JSON with `render`).

```
def struct slack.Message {
    text as string,           # top-level fallback / notification text ("" to
omit)
    blocks as list of string  # pre-rendered block JSON fragments
};
```

Call	Returns	
slack.message()	Message	start an empty message
slack.text(m, text)	Message	set the fallback / notification text
slack.section(m, markdown)	Message	append a section block (mrkdwn)
slack.header(m, heading)	Message	append a header block (plain text)
slack.contextBlock(m, text)	Message	append a context block (small muted mrkdwn)
slack.fieldsSection(m, fields)	Message	append a section laid out as two-column fields
slack.button(text, url)	string	build a URL button fragment for actionsBlock
slack.actionsBlock(m, buttons)	Message	append an actions block of buttons
slack.divider(m)	Message	append a divider block
slack.render(m)	string	render the JSON payload
slack.sendMessage(webhookUrl, m)	http.Response	post the built message

Text passed to any builder is JSON-escaped for you (via the `json` library), so quotes, newlines, and other meta-characters are safe. The fallback `text` is shown in notifications and by clients that do not render blocks - set it even when you use blocks.

`fieldsSection` takes a `list of string` of mrkdwn snippets (Slack packs them into a two-column grid - a common convention is `"*Key:*\nvalue"` per field). For an actions block, build each button with `slack.button(text, url)` and pass the list to `actionsBlock` :

```
def m as slack.Message init slack.message();
$m = slack.header($m, "Deploy");
$m = slack.fieldsSection($m, ["*Env:*\nprod", "*Build:*\n1234"]);
$m = slack.actionsBlock($m, [
    slack.button("View build", "https://ci.example.com/build/1234"),
    slack.button("Logs", "https://ci.example.com/build/1234/logs")
]);
$m = slack.contextBlock($m, "posted by ci-bot");
```

Scope

- **Incoming Webhooks** , not the Web API - no bot tokens, `chat.postMessage` , threads, reactions, or file uploads. The channel is fixed by the webhook.
- **A subset of Block Kit** - `section` / `header` / `context` / `fields` / `actions` (URL buttons) / `divider` . Accessories, images, and non-button interactive elements are not built here (compose the JSON yourself and post via `http` if you need them).
- **No retry / rate-limit handling** - a non-2xx is returned as the response value for the caller to inspect, not thrown.

See also

- http.md - the HTTP client this module builds on.
- discord.md / gotify.md - the sibling notifiers.
- modules/index.md - the module catalog and import rules.

smtp - send mail (SMTP client)

Import with `import "smtp.j" as smtp;` . An SMTP **send** client: the line-oriented command/response dialogue of RFC 5321 over the `net` system library, with plaintext / implicit-TLS / STARTTLS transport and SASL AUTH (PLAIN / LOGIN / XOAUTH2 / CRAM-MD5 / SCRAM-SHA-1 / SCRAM-SHA-256). The message body is any string, typically built by the `mime` module. Because it uses `net` , this module needs the default **jennifer** binary; on the stock `jennifer-tiny` a send raises a friendly error.

> **On jennifer-tiny:** "needs the default `jennifer` binary" refers to the **stock** tiny build, which ships without a network driver - not a TinyGo limitation. A `jennifer-tiny` rebuilt with a network stack runs this module too; see the note on `net` and TinyGo.

```
import "smtp.j" as smtp;
import "mime.j" as mime;

def msg as mime.Part init mime.text("text/plain", "Hello!");
$msg = mime.withHeader($msg, "Subject", "Hi");

def opts as smtp.Options init smtp.Options{host: "mail.example.com", port: 587,
    security: "starttls", clientName: "me.example.com",
    user: "me@example.com", pass: "secret"};
smtp.send($opts, "me@example.com", ["you@example.com"], mime.encode($msg));
```

Runnable: `examples/modules/smtp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/smtp_demo.j).

Surface

Call / type	Notes
smtp.Options	host, port, security, clientName, user, pass, auth, allowInsecureAuth.
smtp.send(opts, from, recipients, message)	One-shot: open a session, deliver message, close. Throws on a server rejection.
smtp.Session	A live, authenticated connection (conn, open) for delivering many messages over one handshake.
smtp.open(opts)	Connect, greet, STARTTLS, and authenticate once; returns a ready Session.
smtp.sendOn(session, from, recipients, message)	Deliver one message over an open session (RSET + MAIL / RCPT / DATA); reusable afterwards.
smtp.close(session)	QUIT (best-effort) and close the session's socket.

`Options` fields:

Field	Notes
host	Server hostname.
port	Server port (25 / 587 plaintext or STARTTLS, 465 implicit TLS).
security	A transport.Security: .None (plaintext), .Starttls (upgrade after EHLO), or .Tls (implicit TLS on connect). Needs import "transport.j" as transport;.
clientName	The EHLO identity; defaults to "localhost" when empty.
auth	SASL mechanism: "" (none when user is empty, else PLAIN), "auto" (negotiate the strongest mechanism EHLO advertises, falling back to PLAIN), "plain", "login", "xoauth2", "cram" (CRAM-MD5), "scram-sha-1", or "scram-sha-256".
user	SASL username; "" with auth: "" skips authentication.

pass	SASL password.
allowInsecureAuth	Force SASL AUTH over an unencrypted (security: "none") connection. Default false: credentials are refused over plaintext (use "tls" / "starttls", or set this to force).

What send does

One call runs the whole delivery, throwing a catchable `Error` (kind `"smtp"`) the moment the server rejects a step:

1. Connect per `security` (`net.connect`, or `net.connectTLS` for `"tls"`), bounded by a connection-establishment timeout.
2. Read the `220` greeting, send `EHLO`.
3. For `"starttls"`: confirm the `EHLO` response actually advertised `STARTTLS` (an anti-downgrade check - a MITM that strips the capability to keep the session in plaintext is refused), then `STARTTLS`, `net.startTLS`, and a second `EHLO`.
4. Authenticate per `auth` (via the `sasl` mechanisms): `AUTH PLAIN`, the `AUTH LOGIN` two-step, `AUTH XOAUTH2` (an OAuth2 bearer token in `pass` - how Google / Microsoft 365 authenticate), `AUTH CRAM-MD5`, or the SCRAM challenge-response (`AUTH SCRAM-SHA-1` / `SCRAM-SHA-256`, whose server signature the client verifies).
5. `MAIL FROM:<from>`, one `RCPT TO:<r>` per recipient, `DATA`.
6. Send the message (CRLF-normalised and dot-stuffed) ended by `<CRLF>.<CRLF>`.
7. `QUIT` and close.

The `from` / `recipients` are the **envelope** (who the server routes to), separate from the `From:` / `To:` header lines in the message - set both. Each envelope address is validated before the connection opens: control characters and CRLF (command injection) are rejected, and the address must be a well-formed `local@domain` per RFC 5321 (a missing `@`, an empty local part or domain, or a length over 254 throws).

When `auth` runs over an unencrypted (`security: "none"`) connection, `send` **refuses** to hand SASL credentials to the server unless `allowInsecureAuth: true` is set - so a password does not cross the wire in the clear by accident. `"tls"` and a completed `"starttls"` upgrade are encrypted and unaffected.

Certificate verification for `"tls"` / `"starttls"` is the `net` default (on; see net.md for the opt-out).

Persistent session (many messages, one handshake)

`smtp.send` is the one-shot convenience: it `open`s a session, delivers one message, and `close`s. To send a **queue** of messages, do the handshake (connect + STARTTLS + AUTH) once with `smtp.open`, then `smtp.sendOn` each message, and `smtp.close` at the end - so N messages pay one TLS + auth round-trip instead of N:

```
def s as smtp.Session init smtp.open($opts);
for (def m in $queue) {
    smtp.sendOn($s, $m.from, $m.recipients, $m.message);
}
smtp.close($s);
```

The `Session` is value-semantic, but its `conn` is a shared `net.Conn` handle, so passing `$s` to each `sendOn` reuses the same socket. Each `sendOn` issues a `RSET` first, so a rejected message cannot bleed into the next. `sendOn` on a closed session throws `kind "smtp"`; `close` sends a best-effort `QUIT` (a dead connection is closed regardless). A rejected `sendOn` leaves the session open - `close` it (or keep sending) as you choose.

Errors

A rejection at any step throws `Error{kind: "smtp", message: "..."}` carrying the step and the server's reply, so wrap untrusted sends in `try` / `catch`:

```
try {
    smtp.send($opts, $from, $rcpts, $wire);
} catch (e) {
    io.printf("send failed: %s\n", $e.message);
}
```

A connection failure (host down, port blocked) surfaces as the underlying `net` error through the same path.

Testing

The pure protocol logic - reply-code parsing (including multi-line `250-` continuations), `AUTH PLAIN` base64, and dot-stuffing - is unit-tested in the overlay. The networked `send` path is covered end to end by an in-process fake SMTP server in the Go test suite (so it runs in CI without an external server); a live send against a real daemon is the demo's job.

Out of scope

- **Send only.** Receiving is POP3 / IMAP (later sub-milestones); this module does not fetch mail.
- **Auth mechanisms** : PLAIN, LOGIN, XOAUTH2, CRAM-MD5, and SCRAM-SHA-1 / SCRAM-SHA-256 (via `sasl`). `auth: "auto"` selects the strongest the server's `EHLO` advertises. Server-side SASLprep is not applied.
- **No connection reuse / pipelining.** `send` opens, delivers, and closes one connection per call.
- **Non-ASCII local parts only.** An internationalized **domain** in the host or an envelope address (`user@münchen.de`) is IDNA-encoded to its `xn--` form automatically (via `idna`). A non-ASCII **local part** (before the `@`) still **throws** - it needs SMTPUTF8 (RFC 6531), a later step - rather than sending a misrouted address.

Timeouts

Reads carry a 30 s idle timeout (a deadline re-armed before each read), so a hung server fails with a catchable error instead of blocking the caller forever. The initial connect and the STARTTLS handshake are bounded by their own connection-establishment timeout, so a slow or unreachable server fails the dial rather than blocking it indefinitely.

See also

- mime.md - build the `message` (headers, multipart, encodings).
- net.md - the transport (`connect` / `connectTLS` / `startTLS`) and TLS options `smtp` builds on.
- modules/index.md - the module catalog and import rules.

snmp - SNMP v1 / v2c client

Import with `import "snmp.j" as snmp;` . An SNMP manager (client) that queries an agent over UDP: `get` , `getNext` , `set` , and subtree `walk` , with community-string authentication. The wire messages are ASN.1 BER, built and parsed with the `asn1` library; the transport is `net` UDP. Needs the default `jennifer` binary (`net`).

```
import "snmp.j" as snmp;
use io;

def c as snmp.Client init snmp.client("192.0.2.10", "public");
def vbs as list of snmp.Varbind init snmp.get($c, ["1.3.6.1.2.1.1.1.0"]);
io.printf("%s = %s\n", $vbs[0].oid, $vbs[0].value);   # sysDescr

# Walk a subtree.
for (def vb in snmp.walk($c, "1.3.6.1.2.1.1")) {
    io.printf("%s [%s] %s\n", $vb.oid, $vb.type, $vb.value);
}
```

Runnable: `examples/modules/snmp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/snmp_demo.j).

Client

```
export def struct Client {
    address as string, community as string,
    version as int, timeoutMs as int, retries as int
};
```

Call	Returns	
snmp.client(host, community)	Client	Defaults: UDP port 161, SNMP v2c, 2s timeout, 1 retry.
snmp.clientWith(address, community, version, timeoutMs, retries)	Client	Full control; address is "host:port", version is snmp.VERSION1 or snmp.VERSION2C.

Constants: `snmp.VERSION1` (`0`) and `snmp.VERSION2C` (`1`). v2c is the default; many older or embedded devices (some printers, sensors) answer **v1 only** - if a query times out, try `clientWith(..., snmp.VERSION1, ...)` .

Requests

Each call sends one PDU and returns the response bindings, throwing a `snmp` -kind `Error` if the agent reports an error status or does not answer within `timeoutMs` (after `retries` extra attempts).

Call	Returns	
snmp.get(c, oids)	list of Varbind	GET the given OIDs (a list of string).
snmp.getNext(c, oids)	list of Varbind	GETNEXT: the binding lexically after each OID.
snmp.set(c, varbinds)	list of Varbind	SET the bindings (build them with intVar / stringVar / oidVar); returns the agent's echo.
snmp.walk(c, rootOid)	list of Varbind	Repeated GETNEXT from rootOid until the OID leaves the subtree or the agent signals endOfMibView.

Bindings

```
export def struct Varbind { oid as string, type as string, value as string,
number as int };
```

A returned `Varbind` carries the OID, the SNMP value `type` , a string `value` rendering, and `number` (the

integer for a numeric type, else `0`):

type	value	number
integer	decimal string	the value
octetString	UTF-8 text if all printable, else 0x-prefixed hex	0
oid	dotted OID	0
null	""	0
counter32 / gauge32 / timeTicks / counter64	decimal string (or 0x-hex if >= 2^63)	the value (0 if it exceeds int64)
ipAddress	dotted a.b.c.d	0
opaque	hex	0
noSuchObject / noSuchInstance / endOfMibView	""	0

For a SET, build typed bindings:

Call	Builds
snmp.intVar(oid, n)	an INTEGER binding
snmp.stringVar(oid, s)	an OCTET STRING binding
snmp.oidVar(oid, target)	an OBJECT IDENTIFIER binding

Agent (server)

The module is also an **agent** (server): it answers GET / GETNEXT / SET for a MIB you supply. This is a hardware simulator for testing an SNMP tool, a self-contained example (no external device), or a way to expose your own app's metrics to an SNMP-native monitoring system.

```
import "snmp.j" as snmp;
use net;
use task;

# A MIB: one binding per OID, any SNMP value type.
def a as snmp.Agent init snmp.agent("public", snmp.VERSION2C, [
    snmp.stringVar("1.3.6.1.2.1.1.1.0", "Jennifer SNMP agent"),
    snmp.varbind("1.3.6.1.2.1.1.3.0", "timeTicks", "", 424242),
    snmp.intVar("1.3.6.1.2.1.1.7.0", 72)
]);

# Serve on a loopback socket in a spawned goroutine, then query it.
use channel;
def sock as net.UDPSocket init net.listenUDP("127.0.0.1:0");
def stop as channel of bool init channel.make(1);
def server as task of null init spawn { snmp.serveOn($a, $sock, $stop); };
def c as snmp.Client init snmp.clientWith(net.address($sock), "public", snmp.
VERSION2C, 2000, 3);
snmp.get($c, ["1.3.6.1.2.1.1.1.0"]);      # -> "Jennifer SNMP agent"
channel.send($stop, true);                # graceful shutdown ...
task.wait($server);                       # ... then join the agent
```

Call	Returns	
snmp.agent(community, version, bindings)	Agent	Build an agent from a list of Varbind MIB (build entries with varbind / intVar / stringVar / oidVar).

snmp.serve(a, address)	(blocks)	Bind address (":161") and serve forever. Run as the program's main loop, or in a spawn.
snmp.serveOn(a, socket, stop)	(blocks)	Serve on an already-bound net.UDPSocket until shut down (bind first, then spawn this, to avoid a start-up race). Stop it with channel.send(stop, true) on a channel of bool (capacity >= 1); the loop returns within ~250 ms, so task.wait the handle for a graceful join.

The agent answers **GET** (a missing OID is a v2c `noSuchObject` exception, or a `v1` `noSuchName` error-status), **GETNEXT** (the next OID in numeric order - so `walk` works; past the end is `endOfMibView`), and **SET** (updates an existing binding; an unknown OID errors). A request with the wrong community, or a malformed datagram, is dropped silently, as a real agent does.

> **Security.** Like any SNMP agent, this one is a UDP responder: a request can elicit a larger response (amplification), and the reply goes to the datagram's source address, which can be spoofed (reflection). The community string is plaintext (v1 / v2c has no real authentication). Treat the agent as a simulator / test tool or an internal-metrics endpoint - **do not expose it on an untrusted network** without a firewall or source allowlist in front of it.

Runnable: `examples/modules/snmp_agent_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/snmp_agent_demo.j) runs an agent and a client in one process.

Scope

SNMP v1 / v2c only - **no SNMPv3 / USM** (user-based security), which is the authentication / privacy model a later tier would add. v1 / v2c has no privacy or integrity either: the community travels in plaintext, and a response is matched only by request-id (a random 31-bit value), not by source address - so a network-level observer can read the community and a well-placed attacker can spoof a reply. Treat the community as a filter, not a secret. GETBULK is not built (a `walk` uses GETNEXT). `number` is `int64` -bounded, so a `Counter64` (or other unsigned type) whose value is >= `2^63` is returned as a `0x` -prefixed **hex** `value` string with `number 0` , rather than a wrong negative number. There is no MIB compiler: OIDs are dotted numeric strings, and values are returned by SNMP type, not resolved to named objects. The agent answers GET / GETNEXT / SET but does not send **traps** / notifications, and does not implement **GETBULK** (`walk` uses GETNEXT); its MIB is a flat `list of Varbind` , so lookup is linear (fine for the hundreds of OIDs a simulator holds, not a huge tree). Every OID in the MIB is **writable** by a client with the community - there is no read-only distinction, so filter out any OID you do not want settable when you build the agent's bindings.

See also

The `asn1` library (the BER/DER codec underneath) and `net` (the UDP transport); the module index.

sqlmigrate - version-tracked schema migrations over sql

Import with `import "sqlmigrate.j" as mig;` . A version-tracked schema-migration runner over the `sql` library: it applies ordered, reversible DDL migrations and records what ran in a `schema_migrations` tracking table, so each migration runs **exactly once** and can be rolled back.

It is deliberately **decoupled from the orm mapper** . A migration's `up` / `down` are plain `list of string` DDL statements, so you build them with `orm` 's DDL helpers (`orm.createTable` / `orm.dropTable` / `orm.addColumn` / ...) **or** hand-written SQL - the runner never touches a `Schema` or a `Query` . That keeps it usable for any `sql` program, ORM or not.

```
import "sqlmigrate.j" as mig;

def m1 as mig.Migration init mig.Migration{
    version: "001",
    description: "create users",
    up: ["CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)"],
    down: ["DROP TABLE users"]
};
def applied as int init mig.migrate($conn, [$m1]);   # 1 the first run, 0 after
```

Runnable: `examples/modules/sqlmigrate_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/sqlmigrate_demo.j).

The Migration struct

Field	Type	Meaning
version	string	The ordering key ([A-Za-z0-9._-]).
description	string	A human-readable label, recorded in the tracking table.
up	list of string	The forward DDL statements.
down	list of string	The reverse DDL statements.

Migrations are ordered by `version` **lexically** , so zero-pad numeric versions (`"001"` , `"002"` , ...) or use a sortable timestamp (`"20260101120000"`).

The runner

Each function takes a raw `sql.Connection` (not an `orm.Session`) because the runner owns transaction control: each migration's `up` (or `down`) statements **plus** its tracking-table write run inside **one** `sql` transaction, so a failed statement rolls the whole step back and leaves `schema_migrations` consistent.

Call	Returns	Notes
mig.migrate(conn, migrations)	int	Apply every pending migration in version order; returns how many ran. Idempotent - applied versions are skipped.
mig.rollbackMigrations(conn, migrations, steps)	int	Reverse the steps most-recently-applied migrations (newest first); returns how many ran.
mig.migrationStatus(conn, migrations)	list of MigrationStatus	The applied / pending state of every migration, in version order.

`MigrationStatus` is `{version as string, description as string, applied as bool}` . The tracking table (`version VARCHAR(255) PRIMARY KEY, description TEXT`) is created on first use and is accepted verbatim by both MySQL and PostgreSQL, so the runner needs no dialect.

```
# Build the DDL with orm's helpers, or hand-write it - the runner runs strings.
def m2 as mig.Migration init mig.Migration{
    version: "002",
    description: "add unique email",
    up: [
        orm.addColumn("users", "email", orm.ColumnKind.String, orm.Dialect.
Postgres),
        orm.createIndex("idxUsersEmail", "users", ["email"], true)
    ],
    down: [
        orm.dropIndex("idxUsersEmail", "users", orm.Dialect.Postgres),
        orm.dropColumn("users", "email")
    ]
};
def plan as list of mig.Migration init [$m1, $m2];
mig.migrate($conn, $plan);                  # applies 001 then 002
mig.rollbackMigrations($conn, $plan, 1);    # reverses 002
```

Safety

- **Version and description are injection-safe.** The `version` is allowlisted (`[A-Za-z0-9._-]`) and the `description` is escaped as a quoted SQL literal (backslash / control characters rejected), both before reaching the tracking-table SQL. The `up` / `down` DDL strings are the caller's own (built by `orm` 's allowlist-checked helpers or hand-written); the runner executes them verbatim.
- **Atomic per migration.** A migration that fails partway leaves nothing recorded and nothing half-applied - the whole step rolls back.

Needs `sql` , so the **default jennifer** binary.

statsd - StatsD metrics client

Import with `import "statsd.j" as statsd;` . Emit `metric:value|type` lines to a StatsD / Datadog / Telegraf agent over UDP. This is the **push** counterpart to a pull-based scrape: it is **fire-and-forget** (UDP, no reply, no error when no agent is listening), so a metric costs one datagram and never blocks the program. Needs the default `jennifer` binary (`net`).

```
import "statsd.j" as statsd;

def c as statsd.Client init statsd.clientWith("127.0.0.1:8125", "web");
statsd.increment($c, "requests");        # web.requests:1|c
statsd.timing($c, "response", 42);       # web.response:42|ms
statsd.gauge($c, "queue.depth", 7);      # web.queue.depth:7|g
statsd.close($c);
```

Runnable: `examples/modules/statsd_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/statsd_demo.j).

Client

```
def struct statsd.Client {
    socket as net.UDPSocket,   # the sending socket
    address as string,         # the agent "host:port"
    prefix as string           # a metric-name namespace ("" for none)
};
```

A `Client` bundles the sending socket, the agent address, and an optional metric-name **prefix** . Value-copies share the underlying socket (the usual handle carve-out to value semantics), so copying a `Client` is safe and cheap. The prefix is joined to every metric name with a `.` separator, so prefix `web` and metric `hits` send `web.hits` ; an empty prefix sends the bare name.

Call	Returns	
statsd.client(host)	Client	connect to host:8125 (the default port), no prefix
statsd.clientWith(address, prefix)	Client	connect to a full host:port with a metric-name prefix
statsd.close(c)		close the sending socket

Metrics

Call	Wire line	Type
statsd.count(c, name, value)	name:value|c	counter delta (value may be negative)
statsd.increment(c, name)	name:1|c	counter +1
statsd.decrement(c, name)	name:-1|c	counter -1
statsd.gauge(c, name, value)	name:value|g	absolute gauge
statsd.timing(c, name, ms)	name:ms|ms	timer, milliseconds
statsd.set(c, name, value)	name:value|s	unique-member set (agent counts distinct values)

All six are fire-and-forget: they format one line, send one datagram, and return. `count` / `increment` / `decrement` adjust a counter; `gauge` sets an absolute value; `timing` records a duration the agent aggregates into percentiles; `set` records a distinct member (e.g. a user id) the agent counts uniquely. The base verbs take integer counter / gauge values and string `set` members; the extensions below add sample rates, tags, float values, and batching.

Sample rates

A `|@rate` suffix tells the agent this metric was emitted from only a fraction of the events, so it scales the received value back up by `1/rate` . Emit it from a `*Rate` verb; a `rate` of `1.0` (or any value `>= 1`) samples everything and sends no suffix.

Call	Wire line
statsd.countRate(c, name, value, rate)	name:value|c|@rate
statsd.timingRate(c, name, ms, rate)	name:ms|ms|@rate

```
statsd.countRate($c, "hits", 1, 0.1);       # web.hits:1|c|@0.1
statsd.timingRate($c, "response", 42, 0.5); # web.response:42|ms|@0.5
```

The rate is formatted compactly (`0.1` , not `0.100000`).

Tags (DogStatsD)

The Datadog `|#key:value,key2:value2` extension attaches dimensions to a metric. Pass a `map of string to string` (rendered in insertion order) to a `*Tagged` verb; the tag suffix comes last on the line, after any sample rate.

Call	Wire line
statsd.countTagged(c, name, value, tags)	name:value|c|#k:v,...
statsd.incrementTagged(c, name, tags)	name:1|c|#k:v,...
statsd.decrementTagged(c, name, tags)	name:-1|c|#k:v,...
statsd.gaugeTagged(c, name, value, tags)	name:value|g|#k:v,...
statsd.timingTagged(c, name, ms, tags)	name:ms|ms|#k:v,...
statsd.setTagged(c, name, value, tags)	name:value|s|#k:v,...

```
def tags as map of string to string init {"env": "prod", "host": "h1"};
statsd.countTagged($c, "hits", 1, $tags);  # web.hits:1|c|#env:prod,host:h1
```

Tag keys and values are control-character validated on the wire line just like the metric name, so a tag carrying a newline, `|` , or `,` (which would forge an extra metric or tag) is a positioned `statsd` error.

Float values

StatsD also accepts fractional counter / gauge values (e.g. a load average).

Call	Wire line
statsd.countFloat(c, name, value)	name:value|c
statsd.gaugeFloat(c, name, value)	name:value|g

```
statsd.gaugeFloat($c, "load", 3.5);        # web.load:3.5|g
```

The float is formatted compactly (no trailing padding).

Batching

`Batch` packs several metrics into **one** UDP datagram (the StatsD multi-metric packet, lines separated by `\n`). It is a value-semantic builder: each `add*` verb returns a fresh `Batch` with the line appended, and `flush` sends the whole packet through a client. The prefix is captured from the client at `batch` time.

```
def b as statsd.Batch init statsd.batch($c);
$b = statsd.addCount($b, "hits", 3);    # web.hits:3|c
$b = statsd.addGauge($b, "queue", 7);   # web.queue:7|g
$b = statsd.addIncrement($b, "errors"); # web.errors:1|c
statsd.flush($c, $b);                   # one datagram, three lines
```

Call	Returns	
statsd.batch(c)	Batch	start an empty accumulator, capturing the client prefix
statsd.addCount(b, name, value)	Batch	append name:value|c
statsd.addIncrement(b, name)	Batch	append name:1|c
statsd.addDecrement(b, name)	Batch	append name:-1|c
statsd.addGauge(b, name, value)	Batch	append name:value|g
statsd.addTiming(b, name, ms)	Batch	append name:ms|ms
statsd.addSet(b, name, value)	Batch	append name:value|s
statsd.flush(c, b)		send all lines in one datagram (an empty batch sends nothing)

Each line is individually validated as it is added, so a batched packet can no more forge an extra metric than a single-metric send.

Scope

- **Fire-and-forget only.** UDP means a lost or unheard datagram is silent by design - there is no delivery confirmation and no error when the agent is down. Use it for metrics, not for data you must not lose.
- **One datagram per verb (except Batch).** The single-metric verbs each send their own datagram; use a `Batch` to coalesce several metrics into one packet.

See also

- net.md - the UDP surface (`listenUDP` / `sendTo`) the client is built on.
- modules/index.md - the module catalog and import rules.

telegram - Telegram Bot API client

Import with `import "telegram.j" as telegram;` . Drive a Telegram bot over the Bot API (https://core.telegram.org/bots/api): send messages and photos, identify the bot, and long-poll for incoming updates. Built on the `http` module + `json` , so it needs the default `jennifer` binary. Larger than the one-shot notifiers (`slack` / `discord` / `gotify`) - `getUpdates` drives a stateful receive loop. An API error (`{"ok": false, ...}`) throws `Error{kind: "telegram"}` .

```
import "telegram.j" as telegram;

def bot as telegram.Bot init telegram.bot("123456:ABC-DEF...");   # token from
@BotFather
telegram.sendMessage($bot, 12345678, "hello from Jennifer");

def updates as list of telegram.Update init telegram.getUpdates($bot, 0, 30);
```

Runnable: `examples/modules/telegram_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/telegram_demo.j).

The bot

```
def struct telegram.Bot { token as string, baseUrl as string };
```

Call	Returns	
telegram.bot(token)	Bot	a bot against the public Telegram API
telegram.botWith(token, baseUrl)	Bot	a bot against a custom API base (a self-hosted Bot API server, or a test endpoint)

The token is a secret - read it from the environment, never commit it. Every call POSTs `application/x-www-form-urlencoded` params to `baseUrl/bot<token>/<method>` and verifies the `{"ok": true}` envelope, throwing on an API error with the server's `description` .

Sending

Call	Returns	
telegram.sendMessage(bot, chatId, text)	Message	send a text message
telegram.sendMessageWith(bot, chatId, text, parseMode)	Message	with a parse mode ("Markdown", "MarkdownV2", "HTML", or "")
telegram.sendMessageWithKeyboard(bot, chatId, text, rows)	Message	with an inline keyboard attached (see below)
telegram.sendPhoto(bot, chatId, photo, caption)	Message	send a photo by URL or file id, with an optional caption
telegram.sendPhotoFile(bot, chatId, filePath, contentType)	Message	upload a photo from a local file (multipart)
telegram.sendDocumentFile(bot, chatId, filePath, contentType)	Message	upload any document from a local file (multipart)
telegram.sendChatAction(bot, chatId, action)	bool	show activity ("typing", "upload_photo", ...)
telegram.answerCallbackQuery(bot, callbackId, text)	bool	acknowledge a button press (text = optional notification)
telegram.getMe(bot)	User	the bot's own identity (a good token / connectivity check)

`chatId` is an integer (Telegram user, group, or channel id - channel ids are large and negative, which fits Jennifer's 64-bit `int`). A returned `Message` carries the text-relevant fields:

```
def struct telegram.Message {
    messageId as int,   # the message id
    chatId as int,      # the chat it belongs to
    text as string,     # the text ("" for non-text messages)
    date as int         # send time as a Unix timestamp
};
def struct telegram.User {
    id as int, isBot as bool, firstName as string, username as string
};
```

Inline keyboards

Attach a grid of buttons to a message. A `Button` carries either a `url` (opens a link) or a `callbackData` (fires a `callback_query` back to the bot); build one with `urlButton` / `inlineButton` rather than the raw literal. Rows are a `list of list of Button` - each inner list is one row.

Call	Returns	
telegram.urlButton(text, url)	Button	a button that opens url
telegram.inlineButton(text, callbackData)	Button	a button that sends callbackData back as a callback_query
telegram.renderInlineKeyboard(rows)	string	render rows to the reply_markup JSON (pure, no network)

```
def rows as list of list of telegram.Button init [
    [ telegram.urlButton("Docs", "https://example.org"),
      telegram.inlineButton("Ping", "ping") ],
    [ telegram.inlineButton("Close", "close") ]
];
telegram.sendMessageWithKeyboard($bot, 12345678, "pick one", $rows);
```

`renderInlineKeyboard` is pure - it produces the exact JSON `sendMessageWithKeyboard` sends as the `reply_markup` parameter (a URL button emits `{"text":...,"url":...}` , a callback button `{"text":...,"callback_data":...}`), so you can assert its shape without a network round-trip.

Callback queries

When a user presses a callback button, Telegram delivers a `callback_query` update. `getUpdates` sets `hasMessage` false for it (it is not a text message); reach the raw JSON via a direct `http` call, then parse it:

Call	Returns	
telegram.parseCallbackQuery(update)	CallbackQuery	parse a raw update json.Value (pure, no network)
telegram.answerCallbackQuery(bot, callbackId, text)	bool	ack the press (text = optional brief notification)

```
def struct telegram.CallbackQuery {
    id as string,       # pass to answerCallbackQuery
    from as User,       # who pressed the button
    data as string,     # the button's callbackData
    messageId as int    # the message the button is attached to (0 if absent)
};
```

Always `answerCallbackQuery` a press, even with an empty `text` - otherwise the client shows a spinner until it times out.

Local-file upload

`sendPhoto` takes a URL or a Telegram file id; to upload a file from disk, use `sendPhotoFile` / `sendDocumentFile` . They read the file with `fs.readBytes` , build a `multipart/form-data` body via the `multipart` module (a `chat_id` field plus the file part, named from the path's basename), and POST it. Pass `contentType` as `""` to let Telegram sniff the format.

```
telegram.sendPhotoFile($bot, 12345678, "chart.png", "image/png");
telegram.sendDocumentFile($bot, 12345678, "report.pdf", "application/pdf");
```

`telegram.buildUpload(field, chatId, filename, contentType, data)` is the pure builder underneath - given the file bytes it returns a `multipart.Built` (`contentType` header + `body`), so the upload shape is testable with no disk or network.

Receiving (long-poll)

`telegram.getUpdates(bot, offset, timeout)` long-polls for pending updates. Pass `offset` as the last processed `updateId` + `1` (`0` on the first call) and `timeout` as the wait in seconds; the HTTP read is bounded a few seconds beyond that. Returns a `list of Update` :

```
def struct telegram.Update {
    updateId as int,       # advance the next poll offset to this + 1
    hasMessage as bool,    # whether this update carries a text-message
    message as Message     # the message (zero-valued when hasMessage is false)
};
```

The receive-loop pattern - fetch, process, advance the offset past each update:

```
def offset as int init 0;
def updates as list of telegram.Update init telegram.getUpdates($bot, $offset,
30);
for (def u in $updates) {
    $offset = $u.updateId + 1;
    if ($u.hasMessage and len($u.message.text) > 0) {
        telegram.sendMessage($bot, $u.message.chatId, "echo: " + $u.message.text);
    }
}
# next loop: telegram.getUpdates($bot, $offset, 30)
```

Security: token redaction

The bot token is a secret that rides in the request URL **path** (`baseUrl/bot<token>/<method>`). A transport error from the underlying `http` layer can carry that URL - and thus the token - in its message, so this module scrubs it: every error rethrown from an HTTP call passes through `telegram.redactToken(s, token)` , which replaces the `bot<token>` path segment with `bot<redacted>` and any bare token occurrence with `<redacted>` . A caught `Error{kind: "telegram"}` (or a leaked log line) therefore never exposes the token. `redactToken` is exported so you can apply the same scrubbing to your own log output.

Scope

- **Long-poll only** , no webhook receiver (that needs a public HTTPS server; compose `web` / `httpd` yourself).
- **Text-centric updates.** `Update` surfaces the `message` (text) shape; `edited_message` , `channel_post` , and inline queries set `hasMessage` false - reach the raw JSON via a direct `http` call. A `callback_query` (a button press) is parsed by `parseCallbackQuery` .
- **UTF-8 upload bodies.** `sendPhotoFile` / `sendDocumentFile` send the multipart body through the text `http` client, so a file whose bytes are not valid UTF-8 is rejected on send; the pure `buildUpload` builder is byte-exact regardless. A byte-safe multipart send path is a follow-on.
- **No message editing or deletion** in this version.

See also

- http.md - the HTTP client this module builds on.
- slack.md / discord.md / gotify.md - the one-shot notifier siblings.
- modules/index.md - the module catalog and import rules.

tengine - a text template engine

Import with `import "tengine.j" as tengine;` . A text template engine for lightweight-CMS-style rendering - a subset of Go's `text/template` (the Go / Hugo style) - evaluated directly over a `json.Value` data tree. There is no compile step and no AST: the engine re-scans block bodies as it renders, which is fine at page / CMS scale. It uses only the compiled-in `json` / `strings` / `lists` / `maps` / `convert` libraries, so it runs on **both** binaries.

```
import "tengine.j" as tengine;
use json;

def set as tengine.Set init tengine.newSet();
$set = tengine.add($set, "base", "<h1>{{ .title }}</h1>{{ template \"body\" .
}}");
$set = tengine.add($set, "page", "{{ define \"body\" }}<p>{{ .msg | html }}</p>{{
end }}");
def out as string init tengine.render($set, "base",
    json.decode("{\"title\":\"Hi\",\"msg\":\"a<b\"}"));
# out == "<h1>Hi</h1><p>a&lt;b</p>"
```

Runnable: `examples/modules/tengine_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/tengine_demo.j).

Security: output is not auto-escaped

`tengine` is modelled on Go's `text/template` , **not** `html/template` : it does **no automatic HTML escaping** . Whatever a value contains is emitted verbatim, `<script>` and all. Rendering untrusted data (a form field, a URL parameter, a database row a user controls) straight into an HTML page is therefore a cross-site-scripting (XSS) hole.

Escape every untrusted value that reaches HTML yourself, with the built-in `html` pipe:

```
# SAFE: the user-controlled field is escaped before it lands in the page.
$set = tengine.add($set, "page", "<p>{{ .comment | html }}</p>");

# UNSAFE: .comment is emitted raw - a comment of "<script>...</script>" runs.
$set = tengine.add($set, "page", "<p>{{ .comment }}</p>");
```

The `html` pipe escapes `&` , `<` , `>` , `'` , and `"` . For attribute values and URLs the same caution applies: build links through `html.safeUrl` (which rejects `javascript:` and other dangerous schemes) rather than interpolating a raw URL into an `href` . When a template mixes trusted markup with untrusted data, escape at the boundary - the engine cannot tell the two apart for you.

Data model

Templates render against a **json.Value** node (use `json.decode` to build one). Inside a template three things address data:

- `.` - the *current* node. `{{ . }}` outputs it; `{{ .a.b }}` reads `/a/b` from it (a dotted path is a JSON Pointer). `.` is rebound by `with` , `range` , and `template` .
- **$** - the *root* node passed to `render` (or to the enclosing `template` call). `{{ $.site.title }}` reaches the top-level data even from inside a loop.
- **$name** - a variable (see Variables).

A missing key renders as **empty** , not an error. Output is **not auto-escaped** (this mirrors `text/template` , not `html/template`): pipe untrusted values through `html` in an HTML context.

Truthiness (for `if` / `with` and the range `else`): a value is true when it is non-null, a non-empty string, a non-zero number, `true` , or a non-empty list or map.

API

Call	Returns	Notes
tengine.newSet()	Set	A fresh, empty template set.
tengine.add(set, name, src)	Set	Register src under name; any {{ define }} blocks become their own entries. Value-semantic (returns a new Set).
tengine.render(set, entry, data)	string	Render entry against a json.Value.

`Set` is value-semantic, so build a page per request by `add` ing the base layout, the partials, and the page (which `{{ define }}` s the sections the base pulls in), then `render` the base.

Actions

Action	Effect
{{ .a.b }} / {{ . }} / {{ $.x }} / {{ $var }}	Output a value (see pipes).
{{ if COND }} A {{ else if COND }} B {{ else }} C {{ end }}	Conditional with any number of else ifs and an optional final else.
{{ range .items }} A {{ else }} B {{ end }}	Render A once per element (of a list) or value (of a map, insertion order), rebinding . each time; B for an empty collection.
{{ range $i, $e := .items }} ... {{ end }}	As above, also binding $i (index / key) and $e (element). {{ range $e := .items }} binds just $e.
{{ with .x }} A {{ else }} B {{ end }}	Rebind . to .x for A when truthy, else B.
{{ $x := PIPE }}	Assign a variable (produces no output).
{{ define "name" }} ... {{ end }}	Define a named template (collected when the source is added).
{{ template "name" . }}	Render a named template with the given node as . (and $).
{{ block "name" . }} default {{ end }}	Render the set's name template if one exists, otherwise the inline default.
{{/* comment */}}	Dropped from the output.

Conditionals

A condition is truthiness of a value, or a call to a comparison / boolean function. Functions are prefix, and arguments may be parenthesised to nest:

Function	Meaning
eq a b / ne a b	Equal / not equal (numbers compare numerically, strings and bools by value).
lt / le / gt / ge	Ordered comparison (numeric when both are numbers, else lexical).
and x y ... / or x y ...	Boolean and / or over any number of truthy tests.
not x	Boolean negation.

```
{{ if eq .type "post" }}...{{ end }}
{{ if and .published (not .draft) }}...{{ end }}
{{ if or (eq .section "blog") (eq .section "news") }}...{{ end }}
{{ if gt .count 0 }}{{ .count }} items{{ else }}empty{{ end }}
```

Arguments are terms: `.path` , `$` , `$.path` , `$var` , a `"string"` , a number, or `true` / `false` .

Variables

`{{ $name := PIPE }}` binds a variable to a pipeline's value; `{{ $name }}` reads it, and `{{ $name.a.b }}` indexes into it. A variable is visible from its assignment to the end of the current template (and inside the blocks it encloses) - the usual way to keep the root, or a computed value, reachable inside a `range` :

```
{{ $site := .site }}
{{ range .posts }}
  <a href="{{ $site.baseUrl }}/{{ .slug }}">{{ .title }}</a>
{{ end }}
```

`range $i, $e := .items` is the same mechanism: it binds the index / key and the element per iteration.

Pipes

An output or assignment value may be piped through one or more functions: `{{ .title | trim | title }}` , `{{ .tags | join ", " }}` .

Pipe	Effect
upper / lower	Case conversion.
title	Upper-case the first letter of each word.
trim	Strip surrounding whitespace.
html	Escape & < > " ' as HTML entities.
urlize	Slugify: lower-case, keep alphanumerics, collapse spaces / dashes / underscores to single dashes.
default X	The value if it is truthy, otherwise X (a fallback for optional fields).
truncate N	The first N characters, with ... appended when it shortened (excerpts).
join SEP	Join a list's elements into a string with SEP between them.
len	The length: characters of a string, or elements of a list / map.
printf FORMAT	Format the piped value per FORMAT (see below).

An unknown pipe throws a catchable `Error` (kind `"tengine"`).

Nesting (control blocks, `template` / `block` includes, `range` bodies) is capped at 256 levels; past the cap the engine throws a catchable `Error` (kind `"tengine"`) instead of recursing without bound, so a template that includes itself - directly or through a partner - fails cleanly even when templates come from untrusted authors.

printf

`printf` formats values with a `text/template` -style format string. It works both as a **function** - `{{ printf "%s: %d" .name .count }}` - and as a **pipe** , where the piped value is the last argument: `{{ .n | printf "%02d" }}` renders `07` . Verbs: `%s` / `%v` (string), `%d` (integer), `%f` (float), `%t` (bool), and `%%` (a literal `%`). Each verb accepts the flags `-` (left-align) and `0` (zero-pad), a width, and a `.precision` (decimal places for `%f` , or a maximum length for `%s`):

```
{{ printf "%.2f" .price }}     -> 3.50
{{ printf "%-10s|" .name }}    -> "Ada       |"
{{ range $i, $t := .items }}{{ printf "%02d. %s\n" $i $t.title }}{{ end }}
```

(This is a self-contained subset, not the full Go / `io.printf` verb set - no `%x` / `%e` / `%q` , and no `*` dynamic width.)

Whitespace-trim markers

A leading `{{-` trims the whitespace immediately **before** the action; a trailing `-}}` trims the whitespace **after** it - the same as Go. This keeps generated output tidy without cramming the template onto one line:

```
<ul>
{{- range .items }}
  <li>{{ . }}</li>
{{- end }}
</ul>
```

renders `<ul>\n <li>a</li>\n <li>b</li>\n</ul>` for `items = ["a", "b"]` .

Layout inheritance

The `define` / `template` / `block` trio gives Go/Hugo-style layout inheritance. A base layout names the holes; a page fills them:

```
def set as tengine.Set init tengine.newSet();
$set = tengine.add($set, "base",
    "<html><body>{{ block \"content\" . }}<p>default</p>{{ end }}</body></html>");
$set = tengine.add($set, "page",
    "{{ define \"content\" }}<h1>{{ .heading }}</h1>{{ end }}");
def out as string init tengine.render($set, "base", json.
decode("{\"heading\":\"Hello\"}"));
# <html><body><h1>Hello</h1></body></html>
```

`{{ block "x" . }}...{{ end }}` renders the set's x if a page defined one, else its own inline body - the override hook. `add` pulls each `{{ define }}` out into its own named entry, so the order you `add` pages does not matter. Each `template` / `block` invocation starts a fresh variable scope, and `$` is reset to the node it was called with.

Scope

A focused subset - enough to drive a lightweight CMS (list and single pages, menus, conditional sections, excerpts), not a general programming language:

- **Pipe / function arguments are literal or path terms** (`default "x"` , `truncate 20` , `printf "%02d" .n`); no `*` dynamic widths.
- **No user-defined functions** and no method calls - the built-in comparison / boolean / `printf` functions and pipes are the whole vocabulary.
- **Fixed data source.** The data tree is the `json.Value` you pass; there is no file inclusion or front-matter parsing (compose the tree yourself, e.g. with `toml` / `json` for front matter and `markdown` for bodies).
- **No auto-escaping.** Escaping is explicit via the `html` pipe (a `text/template` , not an `html/template`).

Adding a pipe or a comparison function is one branch in the engine.

See also

- json.md - the `json.Value` data tree templates render over.
- markdown.md - render Markdown bodies to HTML for a template.
- toml.md - parse front matter / config into a data tree.
- html.md - build an HTML tree programmatically instead of from text.
- modules/index.md - the module catalog and import rules.

totp - time-based one-time passwords

Import with `import "totp.j" as totp;` . Generate and verify **TOTP** codes (RFC 6238 (https://www.rfc-editor.org/rfc/rfc6238) over RFC 4226 (https://www.rfc-editor.org/rfc/rfc4226) HOTP) - the six-digit two-factor codes an authenticator app shows. Both sides share a **secret** (a base32 string) and, from the current time, compute the same short numeric code independently. Pure `.j` ; runs on **both** binaries.

Built on `hash.hmac` (HMAC-SHA1 by default; SHA-256 / SHA-512 optional), `encoding` (base32 secrets), and `time` (the 30-second step); the dynamic-truncation step uses `bytes` + bitwise operators.

```
import "totp.j" as totp;

def o as totp.Options;                                  # zero-value: 6 digits, 30 s,
SHA-1
def code as string init totp.generate("JBSWY3DPEHPK3PXP", $o);
def ok as bool init totp.verify("JBSWY3DPEHPK3PXP", $code, $o);
```

Runnable: `examples/modules/totp_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/totp_demo.j).

Options

A `totp.Options` carries the parameters; a zero-value struct (`def o as totp.Options;`) means the common authenticator defaults.

Field	Effect
digits (int)	Code length; 0 means 6.
period (int)	Time step in seconds; 0 means 30.
algorithm (totp.Algorithm)	HMAC digest: totp.Algorithm.Sha1 (the zero-value default), .Sha256, or .Sha512.

The `secret` is a **base32** string - the same value an authenticator app stores. Spaces are ignored, letters are upper-cased, and missing = padding is supplied, so an app's grouped, unpadded secret (`JBSW Y3DP EHPK 3PXP`) decodes fine.

Functions

Call	Returns	Notes
totp.generateSecret()	string	A fresh, crypto-grade random secret at the RFC 6238-recommended 20-byte / 160-bit length, as an unpadded base32 string.
totp.generateSecretN(nbytes)	string	Same, with an explicit byte length (nbytes must be positive).
totp.hotp(secret, counter)	string	The raw RFC 4226 HOTP: the six-digit HMAC-SHA1 code for an explicit counter (the building block TOTP layers a clock over).
totp.generate(secret, opts)	string	The code for the current time.
totp.generateAt(secret, unixSeconds, opts)	string	The code for an explicit Unix time. Deterministic - use it in tests.
totp.verify(secret, code, opts)	bool	True if code is valid for the current time (+/-1-step skew).
totp.verifyAt(secret, code, unixSeconds, opts)	bool	True if code is valid for an explicit Unix time (+/-1-step skew). Deterministic.
totp.verifyWindow(secret, code, window, opts)	bool	Like verify, but accepts a window-step skew on each side of now.

totp.verifyWindowAt(secret, code, unixSeconds, window, opts)	bool	Like verifyAt, with an explicit window. Deterministic.
totp.uri(issuer, account, secret, opts)	string	The otpauth://totp/... provisioning URI (what a QR code encodes).

`verify` / `verifyAt` accept a **+/-1-step skew window** : a code from the immediately previous or next time step still passes, so a small clock drift between the two sides does not reject a legitimate code. A code two or more steps away is rejected.

Use `verifyWindow` / `verifyWindowAt` to set the tolerance explicitly: `window` is the number of period-steps accepted on each side of now, so `window` 0 checks only the current step and `window` 2 tolerates up to a minute of drift (at the default 30-second period). Widen it only as far as the drift you actually need - every extra step is another valid code an attacker may guess.

Generating a secret

`totp.generateSecret()` draws 20 bytes (the RFC 6238-recommended length for SHA-1 TOTP) from a cryptographic source (`crypto.randBytes`) and returns them as an unpadded base32 string - the format an authenticator app stores. It round-trips: the result works directly with `generate` / `verify` . Use `generateSecretN(nbytes)` for a different key length.

```
def secret as string init totp.generateSecret();    # e.g. "JBSWY3DPEHPK3PXP..."
def code as string init totp.generate($secret, $o); # both sides now agree
```

HOTP

`totp.hotp(secret, counter)` is the counter-based RFC 4226 one-time password TOTP is built from (six digits, HMAC-SHA1). Use it directly for event-based HOTP, or to pin against the RFC 4226 Appendix D test vectors; TOTP is exactly this over `unixSeconds // period` .

`generate` / `verify` read the host clock via `time` ; `generateAt` / `verifyAt` take the time as an argument, which is what makes them deterministic (and what the RFC 6238 Appendix B test vectors pin the module against).

Provisioning URI

`totp.uri` builds the string an authenticator app enrols by scanning a QR code. The label is `issuer:account` , and the issuer / account are percent-encoded:

```
totp.uri("ACME Corp", "jane@acme.example", "JBSWY3DPEHPK3PXP", $o)
# otpauth://totp/ACME%20Corp:jane%40acme.
example?secret=JBSWY3DPEHPK3PXP&issuer=ACME%20Corp&algorithm=SHA1&digits=6&period=
30
```

Render that URI as a QR code (any QR generator) and the app is enrolled; the app and `totp.verify` then agree on the code for each 30-second window.

Security notes

The caller is the verifier, and RFC 6238 section 5.2 puts two duties on the verifier that this module does not perform for you. This module only *computes and compares* codes; it keeps no state across calls. You MUST, in your own code:

1. **Prevent replay (single-use).** A code stays valid for the whole validity window (a step, plus the skew window). Record each code that successfully authenticates and reject a second use of the *same* code within that window - otherwise an attacker who observes one code can reuse it until it expires.
2. **Rate-limit verification attempts.** Cap the number of failed verifications per secret (and lock out or back off after a threshold) so an attacker cannot brute-force the small numeric code space. `verify` is already constant-time per attempt, but that does not bound *how many* attempts an attacker gets - only you can.

Both are stateful policy that belongs to your application (a store of used codes, an attempt counter); the module has no place to keep it and never sees your request flow.

- The secret is the shared key: store it like a password, and transmit the provisioning URI over a secure channel only.
- Generate a *new* secret with `totp.generateSecret()` (crypto-grade random, RFC 6238-recommended 20 bytes, base32-encoded) rather than rolling your own - never use `math.rand*` , which is not crypto-grade.
- Keep the skew window as narrow as the real clock drift requires: each extra step (`verifyWindow`) is another simultaneously-valid code.
- SHA-1 is the default because authenticator apps default to it; it is a safe choice for HMAC despite being broken for collision resistance.

See also

- hash.md - the `hmac` primitive TOTP is built on.
- encoding.md - base32 for secrets.
- time.md - the clock the step counter uses.
- modules/index.md - the module catalog and import rules.

transport - shared connection-security mode

Import with `import "transport.j" as transport;` . A single shared enum for the connection **security mode** used by every network-client module that opens a socket - `smtp` , `pop` , `imap` , `redis` , `amqp` , `mqtt` . One typed value instead of a stringly-typed `security` field per module, so a typo is a compile-time error, not a silent plaintext downgrade. Pure `.j` (just an enum + one helper); runs on **both** binaries.

```
import "transport.j" as transport;
import "smtp.j" as smtp;

def o as smtp.Options init smtp.Options{
    host: "smtp.example.com", port: 587,
    security: transport.Security.Starttls,
    clientName: "me", user: "u", pass: "p", auth: "", allowInsecureAuth: false};
```

The Security enum

```
export def enum transport.Security { None, Tls, Starttls };
```

Variant	Meaning
Security.None	Plaintext - no encryption.
Security.Tls	Implicit TLS negotiated on connect (SMTPS / POP3S / IMAPS / rediss / AMQPS / mqtts).
Security.Starttls	Connect in plaintext, then upgrade in-band (SMTP STARTTLS, POP3 STLS, IMAP STARTTLS).

The **zero value** is `Security.None` (the first variant), so a zero-value `Options` is plaintext.

Only the protocols with an in-band upgrade command accept `Starttls` : `smtp` , `pop` , and `imap` . `redis` , `amqp` , and `mqtt` have no such command and **reject** `Security.Starttls` with a catchable `Error` (their `match` handles the variant explicitly rather than silently treating it as plaintext) - use `Security.Tls` for the TLS-on-connect port instead.

Helper

Call	Returns	
transport.encrypted(s)	bool	true for Tls / Starttls, false for None

`encrypted` is the "is this connection protected?" test - e.g. `smtp` uses it to refuse sending SASL credentials over a `None` connection unless `allowInsecureAuth` is set.

Received-data cap

`transport.MAX_RECEIVED_BYTES` (64 MiB) is the tree-wide ceiling on a single received payload, and `transport.checkReceiveSize(n, what)` enforces it: it raises a catchable `Error{kind: "transport"}` naming `what` when n is negative or over the cap. A wire client calls it on a peer-declared body / literal size before reading, and on an accumulating buffer's length inside a read loop, so a hostile or desynced server cannot drive the recover-less interpreter to an unbounded allocation. One shared constant and check, so the whole client suite caps identically instead of each module inventing its own.

```
def bodySize as int init decodeContentBodySize($frame);
transport.checkReceiveSize($bodySize, "amqp message body");   # throws if > 64 MiB
```

Why a separate module

The security mode is the one field shared verbatim across every socket client, so it lives in one place rather than duplicated as six sibling enums (the one-way-per-thing stance). Because Jennifer has no re-export, a program that builds a client `Options` imports `transport` alongside the client module - the small cost of a single shared

type.

See also

- smtp.md / pop.md / imap.md - the mail clients (support all three modes).
- redis.md / amqp.md / mqtt.md - the store / broker clients (None / Tls).
- ldap.md - the LDAP client (None / Tls / Starttls).
- net.md - the TLS transport underneath.
- modules/index.md - the module catalog and import rules.

uri - URL / URI parsing, building, and query strings (RFC 3986)

Import with `import "uri.j" as uri;` . The shared URL layer the network modules (`http` , `rest` , `s3` , `oauth` , `influxdb` , ...) build on instead of re-splitting strings and hand-rolling percent-encoding. `parse` splits a URL into its parts; `build` reassembles one; `encode` / `decode` and `encodeForm` / `decodeForm` are the two web encodings; `buildQuery` / `parseQuery` convert between a `map of string to string` and a query string; and `resolve` applies a relative reference to a base URL. Pure Jennifer over `strings` + `encoding` + `convert` - no Go, no network - so it runs on **either binary** .

```
import "uri.j" as uri;

def u as uri.Uri init uri.parse("https://user@example.com:8443/a/b?x=1#top");
# $u.scheme "https", $u.user "user", $u.host "example.com", $u.port "8443",
# $u.path "/a/b", $u.query "x=1", $u.fragment "top"

def q as string init uri.buildQuery({"name": "a b", "tag": "x&y"});  #
"name=a+b&tag=x%26y"
def abs as string init uri.resolve("http://h/a/b/page.html", "../img.png");  #
"http://h/a/img.png"
```

Runnable: `examples/modules/uri_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/uri_demo.j)

The Uri struct

`parse` returns, and `build` consumes, a value-semantic `Uri` . Missing components are the empty string, so a relative URL has an empty scheme and host. `port` is a `string` ("" when absent) rather than an `int` , to distinguish "no port" from port 0.

Field	Type	Notes
scheme	string	the scheme without the trailing ":" (e.g. "https")
user	string	the userinfo before "@" ("" when absent)
host	string	the host name or IP literal ("" when absent)
port	string	the port after ":" ("" when absent)
path	string	the path, including its leading "/" when present
query	string	the raw query string without the leading "?"
fragment	string	the fragment without the leading "#"

Surface

Call	Returns	Notes
uri.parse(raw)	Uri	Split a URL into its parts. Absent components are "".
uri.build(u)	string	Reassemble a URL from a Uri (the inverse of parse); no re-encoding.
uri.encode(s)	string	RFC 3986 percent-encode (space -> %20). For a path segment.
uri.decode(s)	string	Reverse percent-encoding; a literal "+" stays "+".
uri.encodeForm(s)	string	application/x-www-form-urlencoded (space -> +). For a query value / form body.
uri.decodeForm(s)	string	Reverse form-encoding; a "+" decodes to a space.

uri.buildQuery(params)	string	A map -> key=value&... (form-encoded, insertion order), no leading "?".
uri.parseQuery(q)	map of string to string	A query string -> a map (form-decoded); a bare "key" maps to "".
uri.resolve(base, ref)	string	Resolve a relative reference against a base URL (RFC 3986 §5).

Percent vs form encoding

Two byte-level encodings share the same unreserved set (`A-Z a-z 0-9 - . _ ~` , left literal) but differ on the space:

- **encode / decode** are **RFC 3986 percent-encoding** : a space is `%20` , and a literal "+" is preserved on decode. Use these for **path segments** and anywhere the RFC 3986 rules apply.
- **encodeForm / decodeForm** are **application/x-www-form-urlencoded** : a space is + (so a literal "+" encodes as `%2B`). Use these for **query-string values** and HTML form bodies. `buildQuery` / `parseQuery` use them, matching the query-string convention every browser and HTTP client follows.

Both sit on the `encoding` library's `uri-percent` / `uri-form` codecs, so a program that only needs the raw byte transform can call `encoding.toText(b, "uri-percent")` directly. A malformed `%` escape is a catchable error.

Reference resolution (RFC 3986 §5)

`resolve(base, ref)` turns a possibly-relative reference into an absolute URL, the operation a feed reader or crawler needs to follow relative links:

```
def base as string init "http://h/a/b/page.html";
uri.resolve($base, "sibling");     # "http://h/a/b/sibling"
uri.resolve($base, "../img.png");  # "http://h/a/img.png"
uri.resolve($base, "/root/x");     # "http://h/root/x"
uri.resolve($base, "//other/z");   # "http://other/z"  (network-path reference)
uri.resolve($base, "https://x/y"); # "https://x/y"     (absolute ref, returned as-
is)
```

It applies the RFC 3986 §5.2.4 `remove_dot_segments` algorithm, so `.` and `..` segments collapse (`/a/b/../c` -> `/a/c`).

Notes and scope

- **IPv6 literals** keep their brackets in `host` (`[::1]`) with the port split out, per RFC 3986: `uri.parse("http://[::1]:9000/x")` gives `host "[::1]"` , `port "9000"` .
- **build does not re-encode.** Components are emitted verbatim, so a `Uri` you assembled by hand must already carry encoded values (use `encode` / `encodeForm` yourself). This keeps `parse` -> `build` an exact round-trip.
- **No scheme registry / normalization.** `uri` does not lowercase the scheme, drop a default port, or know that `http` implies port 80 - it is a syntactic parser, not a URL normalizer.

See also

- encoding.md - the `uri-percent` / `uri-form` byte-level codecs this module builds on.
- rest.md / http.md - the HTTP clients that consume `buildQuery` and `resolve` .
- modules/index.md - the module catalog and import rules.

validate - declarative data validation

Import with `import "validate.j" as validate;` . Validate a `map of string to string` (a form body, a query string, a config) against a rule set and get a structured failure list back, instead of ad-hoc per-field `if` checks scattered through a handler. Rules compose as value-semantic descriptors built by the `validate.required` / `isInt` / `min` / `email` / ... family. Pure Jennifer over `regex` + `uri` + `time` + `password` + `convert` + `lists` + `strings` + `maps` , so it runs on **both binaries** .

```
import "validate.j" as validate;

def rules as map of string to list of validate.Rule init {
    "email": [validate.required(), validate.email()],
    "age":   [validate.isInt(), validate.min(0.0), validate.max(150.0)],
    "role":  [validate.oneOf(["reader", "author", "admin"])]
};

def errs as list of validate.Failure init validate.check($form, $rules);
if (len($errs) > 0) {
    for (def m in validate.messages($errs)) { io.printf("%s\n", $m); }
}
```

Runnable: `examples/modules/validate_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/validate_demo.j)

The model

A rule set is a `map of string to list of Rule` : each field maps to the rules it must satisfy, in order. `validate.check(data, rules)` returns a `list of Failure` (`{field, rule, message}`) - empty means valid; `validate.ok(data, rules)` is the bool short-circuit.

Values are **strings** - the shape `web.bodyForm` , a query string, or `dotenv` produce. A type rule (`isInt` / `isFloat` / `isBool`) checks the string *parses* ; `min` / `max` parse then compare; `minLen` / `maxLen` measure the string (in runes).

Two conventions to know:

- **Only fields named in the rule set are checked.** Extra keys in `data` are ignored.
- **An absent or blank field passes every rule except required.** So an optional field left empty is valid; pair `required` with the other rules when a value must be present *and* well-formed.

Rules

Builder	Passes when the value...
validate.required()	is present and non-empty
validate.isInt()	parses as an integer
validate.isFloat()	parses as a number
validate.isBool()	is exactly "true" or "false"
validate.min(n)	is numeric and >= n (a float)
validate.max(n)	is numeric and <= n
validate.minLen(n)	is at least n characters (an int)
validate.maxLen(n)	is at most n characters
validate.pattern(re)	matches the RE2 source re (anchor with ^/$)
validate.email()	looks like an email address
validate.url()	is an absolute URL (a scheme and a host, via uri)

validate.datetime(format)	is a real date/time in the strftime format
validate.oneOf(allowed)	is one of allowed (a whitelist)
validate.noneOf(blocked)	is not one of blocked (a blacklist)
validate.password(policy)	meets a password.Schema policy (length + classes)
validate.custom(fn, msg)	the predicate fn(value) returns true (else msg)
validate.withMessage(r, m)	(modifier) replaces rule r's message with m

`noneOf` matches exactly and case-sensitively, like `oneOf` ; for a case-insensitive blacklist use a `pattern` with the `(?i)` flag (`validate.pattern("(?i)^(admin|root)$")`) or normalise the value first.

`custom` takes a first-class **func** value - a top-level `func(v as string)` returning a bool:

```
func isHandle(v as string) { return validate.ok({"h": $v}, {"h": [validate.
pattern("^[a-z0-9_]+$")]}); }
# ...
"username": [validate.required(), validate.custom(isHandle, "must be a valid
handle")]
```

The predicate is **exception-safe** : if it throws (say a naive `convert.toInt` on a non-numeric value), the field is treated as invalid rather than the exception escaping `check` .

`validate.datetime(format)` uses the `time` library's `strftime` codes, and checks both the format and calendar validity (month 13 / day 32 fail):

format	Accepts
"%d.%m.%Y"	25.12.2026
"%m/%d/%Y"	12/25/2026
"%Y-%m-%d"	2026-12-25 (ISO)
"%Y-%m-%d %H:%M"	2026-12-25 14:30

The default message is generic ("must be a valid date"); pair it with `withMessage` for a user-facing hint: `validate.withMessage(validate.datetime("%d.%m.%Y"), "use DD.MM.YYYY")` .

`validate.password(policy)` delegates to the `password` module's `validate` : build a `password.Schema` with `password.schema()` and its `with*` builders (length range, per-class minimums), and the rule enforces it. The failure message is the policy's own failed-rule reasons joined with `"; "` (e.g. `too short (minimum 8); needs at least 1 uppercase`); collapse them to one hint with `withMessage` :

```
import "password.j" as password;
import "validate.j" as validate;

# at least 8 chars, at least one lower / upper / digit (symbols optional)
def policy as password.Schema init
    password.withMinimums(password.withLength(password.schema(), 8, 64), 1, 1, 1,
0);

def rules as map of string to list of validate.Rule init {
    "password": [validate.required(), validate.password($policy)]
};
```

Because Jennifer has no closures, the policy travels in the `Rule` itself (a `schema` field), so `validate` imports `password` - it stays pure and runs on both binaries (the rule only reads a policy, it never generates).

Reading failures

Call	Returns	Use

validate.check(data, r)	list of Failure	every failure (empty = valid)
validate.ok(data, r)	bool	a quick valid / invalid gate
validate.messages(errs)	list of string	"field: message" lines for a log/flash
validate.byField(errs)	map of string to list of string	field -> its messages, to re-render a form
validate.localize(errs, t)	list of Failure	re-message with your own templates (i18n)

A `Failure` is `{field, rule, param, message}` . `rule` is the failed rule's kind (e.g. `"required"` , `"min"`) - a **stable id** for machine-readable responses or your own message table - and `param` is the rule's argument in string form (a threshold, a joined choices list), the piece a localized message interpolates.

Non-English (or custom) messages

The built-in `message` is English. Because `Failure` carries a stable `rule` id and its `param` , you can re-render every message however you like with `validate.localize(errs, templates)` : `templates` maps a rule id to a template where `%param%` and `%field%` are substituted. A rule **not** in the map keeps its default message, so a partial map only overrides what it names, and the result is a `list of Failure` (feed it to `messages` / `byField`):

```
def de as map of string to string init {
    "required": "ist erforderlich",
    "email":    "muss eine gültige E-Mail-Adresse sein",
    "min":      "muss mindestens %param% sein",
    "minLen":   "muss mindestens %param% Zeichen lang sein",
    "oneOf":    "muss eines von %param% sein"
};
def msgs as list of string init validate.messages(validate.localize($errs, $de));
```

The markers are `%name%` -style (brace-free) on purpose: they never collide with the language's `{expr}` string interpolation, so a template reads the same in a cooked or raw string. Substitution is a single left-to-right pass (a substituted value is never re-scanned), and a literal `%` is written `%%` - valid in cooked or raw alike, since the string engine never touches `%` (it owns `\` and `{}` , the module owns `%`).

For full internationalization, build the `templates` map from `intl` - one `intl.tr("validate.min")` per rule id, so the active locale picks the language - then hand it to `localize` . The rule ids (`required` , `isInt` , `isFloat` , `isBool` , `min` , `max` , `minLen` , `maxLen` , `pattern` , `email` , `url` , `datetime` , `oneOf` , `noneOf` , `password` , `custom`) are the catalog keys. (A `password` rule's default message is the `password` module's own English reasons; a `"password"` template replaces the whole message.)

Pairs with

- **web** - validate `web.bodyForm($ctx)` and respond `422` with `validate.byField` when invalid.
- **rest** - guard a request body before sending or after receiving.
- **dotenv** - validate a loaded config map.

Scope

Values are strings, so validation is over the string form (what forms / queries / config give you). A nested `json.Value` is not validated directly - pull the fields you need into a `map of string to string` first. There is no cross-field rule (e.g. "password == confirm") in v1; express that with a `custom` rule over the whole map, or a follow-up check.

See also

- regex.md / uri.md / time.md / password.md - the `pattern` / `email` / `url` / `datetime` / `password` backends.

- intl.md - the translation catalog to feed `localize` for non-English messages.
- modules/index.md - the module catalog and import rules.

vcard - vCard (RFC 6350) contacts build and parse

Import with `import "vcard.j" as vcard;` . Build contact `Card` s and encode them to **vCard 4.0** text (`VCARD` records), and parse that text back. The contacts counterpart to `ical` : the two share the same content-line codec (TEXT escaping, 75-character line folding). Pure Jennifer over `strings` / `lists` - no Go engine, so it runs on **both** binaries.

```
import "vcard.j" as vcard;

def c as vcard.Card init vcard.card("Ada Lovelace");
$c = vcard.withName($c, "Lovelace", "Ada");
$c = vcard.addEmail($c, "ada@example.com");
def text as string init vcard.encode($c);   # BEGIN:VCARD ... END:VCARD
```

Runnable: `examples/modules/vcard_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/vcard_demo.j).

Types

Both structs have public fields (read them directly - `$c.emails` , `$c.family`); the builder functions are the conventional way to construct them.

```
def struct vcard.Card {
    formattedName as string,                # FN (required by vCard 4.0)
    family as string, given as string,      # N family / given
    additional as string,                   # N additional (middle) name(s)
    prefixes as string, suffixes as string, # N honorific prefixes / suffixes
    nickname as string,                     # NICKNAME ("" when unset)
    organization as string,                 # ORG ("" when unset)
    title as string,                        # TITLE ("" when unset)
    emails as list of Typed,                # EMAIL (0..n, each with an optional
TYPE)
    phones as list of Typed,                # TEL (0..n, each with an optional
TYPE)
    addresses as list of Address,           # ADR (0..n)
    url as string,                          # URL ("" when unset)
    bday as string,                         # BDAY ("" when unset)
    photo as string,                        # PHOTO URI ("" when unset)
    categories as list of string,           # CATEGORIES tags
    note as string                          # NOTE ("" when unset)
};
def struct vcard.Typed { value as string, type as string };   # a value +
optional TYPE ("work" / "home" / ...)
def struct vcard.Address {
    street as string, locality as string, region as string,
    postalCode as string, country as string, type as string
};
```

Building

The builders are **value-semantic** - each returns a fresh copy and never mutates its argument, so you thread them:

Call	Returns	
vcard.card(formattedName)	Card	a card with just its FN display name

vcard.withName(c, family, given)	Card	set the N family / given
vcard.withFullName(c, family, given, additional, prefixes, suffixes)	Card	set all five N components
vcard.withNickname(c, nickname)	Card	set the NICKNAME
vcard.withOrg(c, organization, title)	Card	set ORG and TITLE
vcard.addEmail(c, email) / addEmailTyped(c, email, type)	Card	append an EMAIL (with optional TYPE)
vcard.addPhone(c, phone) / addPhoneTyped(c, phone, type)	Card	append a TEL (with optional TYPE)
vcard.address(...) / addressTyped(..., type)	Address	build an address (with optional TYPE)
vcard.addAddress(c, address)	Card	append an ADR
vcard.withBday(c, bday) / withPhoto(c, uri)	Card	set BDAY / PHOTO
vcard.addCategory(c, tag)	Card	append a CATEGORIES tag
vcard.withUrl(c, url)	Card	set the URL
vcard.withNote(c, note)	Card	set the NOTE

```
def c as vcard.Card init vcard.card("Grace Hopper");
$c = vcard.withName($c, "Hopper", "Grace");
$c = vcard.withOrg($c, "US Navy", "Rear Admiral");
$c = vcard.addEmail($c, "grace@navy.mil");
$c = vcard.addAddress($c, vcard.address("1 Navy Way", "Arlington", "VA", "22202",
"USA"));
```

Encoding and parsing

Call	Returns	
vcard.encode(c)	string	one card as a VCARD (CRLF-terminated)
vcard.encodeAll(cards)	string	many cards concatenated
vcard.parse(text)	list of Card	parse one or many VCARDs

`parse(encode(card))` round-trips the data. `parse` always returns a **list** (a vCard file may hold many cards, or none). `encode` writes `VERSION:4.0` , escapes text values, folds long lines, and omits empty optional fields. `parse` unfolds folded lines, ignores property parameters (the `;KEY=VALUE` after a name, e.g. `EMAIL;TYPE=work`), reads the structured `N` / `ADR` values, and unescapes text.

Structured values

`N` (name), `ADR` (address), and `ORG` are **structured** : components are separated by `;` . `N` is `Family;Given;Additional;Prefix;Suffix` (this module sets family and given); `ADR` is `POBox;Extended;Street;Locality;Region;PostalCode;Country` (this module sets the last five, leaving PO box and extended empty); `ORG` is `Organization;Unit;...` (this module sets and reads the first component). A `;` or `,` inside a component value is escaped, so it never splits the structure.

Scope

- **A contact subset.** `FN` / `N` / `ORG` / `TITLE` / `EMAIL` / `TEL` / `ADR` / `URL` / `NOTE` . No `BDAY` / `PHOTO` / `GEO` / `TZ` / `KIND` / grouping, no parameter round-tripping (a `TYPE=work` on input is dropped, not preserved), and the `N` additional / prefix / suffix and `ADR` PO-box / extended components are not modelled. The escaping / folding / structured-value discipline is the reusable core.
- **vCard 4.0 output.** `encode` always writes `VERSION:4.0` ; `parse` reads any version's properties (it does not enforce the version line).

See also

- ical.md - the calendar counterpart sharing the content-line codec.
- strings.md - the text surface the codec is built on.
- modules/index.md - the module catalog and import rules.

web - a small HTTP framework

```
import "web.j" as web;
```

An ergonomic HTTP framework over the `httpd` server engine. Register routes against handler **func values** , then `web.run` owns the accept loop, matches each request, and dispatches to the handler - so a web app reads as a set of handlers plus a route table, not a hand-written server loop.

Needs the default `jennifer` binary (the `httpd` engine is net-backed; the constrained `jennifer-tiny` has no network stack).

Concurrency: requests are handled concurrently

The serve loop handles requests **concurrently** : it accepts a request, hands it to its own `spawn` ed worker, and immediately accepts the next. A handler that blocks - a `sql` query, an outbound `http` / `rest` call, `memcache` , a slow file read - no longer stalls the requests behind it; each runs on its own worker. In- flight concurrency is bounded by the `httpd` engine's accept rate.

Dispatch calls each handler `func` value, which runs in its home (entry-program) context, and that cross-boundary call path is **race-safe** : each worker carries its own call-depth counter, and reads of shared program state (top-level constants, variables, and the maps / lists they hold) are safe to do from many handlers at once.

The one thing that is **not** safe by construction is a handler **writing** a shared top-level variable of the entry program. Unlike a top-level `spawn` (which deep-copies the scope it captures, so a data race is impossible), handler workers run against the *same* entry-program globals - so two handlers that mutate one top-level `def` at overlapping times race on it, exactly like threads sharing memory in any language. Keep per-request mutable state where it belongs:

- **In the request / response.** The `web.Context` and its helpers are per-request.
- **In a synchronized store** , not a bare global a handler assigns. Server-side session state goes through the `session` module; a counter or cache that several handlers update needs its own synchronization, not a top-level `def` .
- **Offload slow, response-independent work** (sending mail, writing an audit log) to a `spawn` the handler does not wait on - now stacked on top of the per-request worker.

Reads of shared config set once at startup (a routing table, a settings map) are fine; it is concurrent *writes* to one global that you must avoid.

Shutdown and handler lifecycle

Two consequences of per-request workers are worth knowing:

- **Shutdown does not drain in-flight workers.** `httpd.shutdown` (from another task or a signal handler) stops the accept loop; workers already running are *not* waited on. Their clients still get a clean `503` (the engine unblocks them), but a handler mid-way through response-independent work (a started write, an outbound call) is abandoned when the process exits. If you need in-flight requests to finish, quiesce the load before shutting down.
- **exit from a handler does not stop the server.** Each worker is `task.discard` ed, and `discard` suppresses a task's `exit` (standard task semantics), so `exit` / `exit N` called inside a handler is inert - one request cannot unilaterally terminate a multi-request server. Stop the server with `httpd.shutdown` (optionally from an `os.catchSignal` handler), not `exit` .

Handlers

A handler is a top-level method taking a `web.Context` :

```
import "web.j" as web;

func showUser(ctx as web.Context) {
    web.text($ctx, 200, "user " + web.param($ctx, "id"));
}

def app as web.App init web.new();
$app = web.get($app, "/users/:id", showUser);
web.run($app, ":8080");
```

A route stores the handler as a **func value** (pass a method by its bare name, `web.get($app, "/x", showUser)`). A func value carries its home interpreter, so a handler defined in the entry program is called back in the entry program's context across the module boundary - it resolves its own imports and can build the app's own structs. A bare name that is not a defined method is a parse-time error at the call site, so a mistyped handler fails before the server ever starts.

The web.Context

Everything a handler needs hangs off the `web.Context` it receives, so it rarely touches `httpd` directly:

Call	Returns	
web.method($ctx)	string	Request method.
web.path($ctx)	string	Request path.
web.param($ctx, name)	string	A captured :param ("" if none).
web.query($ctx, name)	string	A query-string parameter.
web.header($ctx, name)	string	A request header.
web.body($ctx)	bytes	The raw request body (binary-safe).
web.bodyJson($ctx)	json.Value	The request body decoded as JSON.
web.form($ctx)	map of string to string	The x-www-form-urlencoded body, decoded.
web.formValue($ctx, name)	string	One form field ("" if absent).
web.multipartForm($ctx)	list of multipart.Part	The multipart/form-data body (file uploads), parsed via the multipart module. Import multipart.j in the handler to name multipart.Part / use multipart.isFile / multipart.text.
web.remoteAddr($ctx)	string	The client's host:port.
web.setHeader($ctx, name, value)	null	Set a response header (before responding).
web.respond($ctx, status, body)	null	Send a response (body is a string).
web.text($ctx, status, body)	null	Respond with text/plain.
web.html($ctx, status, body)	null	Respond with text/html.
web.sendJson($ctx, status, doc)	null	Respond with application/json from a json.Value.
web.sendGzip($ctx, status, body)	null	Respond, gzip-compressed when the client accepts it.
web.redirect($ctx, status, location)	null	Redirect (301/302/303/307/308) with a Location header.
web.serveFile($ctx, path)	null	Respond with a file from disk.
web.serveDir($ctx, root)	null	Serve static files from a directory root.

Cookies

Pure HTTP-header work over `httpd` , no extra dependency:

Call	Returns	
web.cookie($ctx, name)	string	The request cookie's value ("" if absent).
web.setCookie($ctx, name, value, opts)	null	Emit a Set-Cookie response header.

`opts` is a `web.CookieOptions` - `path` , `domain` , `maxAge` (int seconds; `0` omits the attribute, negative expires the cookie now), `httpOnly` , `secure` , `sameSite` (`"Lax"` / `"Strict"` / `"None"` / `""`). A zero-value struct is a plain session cookie. Several `setCookie` calls emit several `Set-Cookie` headers (they are not collapsed).

Sessions

`web` owns only the session **id cookie** ; the session **data** lives in a store the **app** owns, so `web` forces no store or network dependency (it never imports `session` / `memcache`). That keeps a `web` app that uses no sessions at `httpd` + `meta` + `json` + collections.

Call	Returns	
web.sessionId($ctx, cookieName)	string	The request's session id, minting a new UUID + HttpOnly / SameSite=Lax / path-/ cookie on first use.
web.renewSession($ctx, cookieName)	string	Rotate the id: mint a fresh UUID and set it as the cookie, returning the new id.

`sessionId` only accepts an incoming cookie that matches the UUID shape it mints; a malformed or non-UUID value is discarded and a fresh id issued. That check stops an attacker from smuggling metacharacters through the id into a store keyed by it, but it does **not** by itself stop session **fixation** - a well-formed UUID an attacker plants (via a sibling subdomain or a stray XSS) still passes. The fixation defence is to **rotate** the id: after any privilege change - a successful login, a logout, a role change - call `web.renewSession` and move your session data to the returned id.

Call `web.sessionId` once per request and pair the id with your own store - for multi-process serving the `session` module over `memcache` ; for a single process anything the app holds:

```
# The app owns the store; web just resolves the id cookie.
func profile(ctx as web.Context) {
    def id as string init web.sessionId($ctx, "sid");
    def data as map of string to string init session.load($store, $id);   # app's
store
    # ... read / write $data ...
    session.save($store, $id, $data, 3600);
    web.text($ctx, 200, "ok\n");
}
```

Stateless signed-cookie sessions (data in the cookie, no store) wait on a `crypto` library for real HMAC - see horizon.md.

Registering routes

Each registrar returns a **new** App (value semantics), so build the app by threading it through:

Call	
web.new()	An empty app.
web.get($app, pattern, handler)	Register a GET route.
web.post / web.put / web.patch / web.delete	The other verbs.
web.route($app, method, pattern, handler)	The general form.

web.before($app, handler)	Add a middleware (runs before each route handler).
web.notFound($app, handler)	A custom handler for unmatched requests (default: a plain 404).
web.onError($app, handler)	An error handler for a throwing route / middleware (default: log to stderr).

A `HEAD` request is served by the matching **GET** route automatically (HTTP requires it, and health checks / caches rely on it); the response body is suppressed by the server, so the handler needs no special-casing.

Patterns are `/` -separated. A segment beginning with `:` captures a single parameter: `/users/:id/posts/:pid` matches `/users/7/posts/9` and captures `id` = `7` , `pid` = `9` . A trailing segment beginning with `*` is a **wildcard** that captures the entire remainder (joined by `/` , possibly empty): `/files/*path` matches `/files/css/app.css` with `path` = `css/app.css` , and `/*path` is a catch-all - a natural **SPA fallback** . Read either with `web.param($ctx, name)` .

The first matching route wins, so register specific routes **before** a wildcard (a greedy `/*path` registered first would swallow everything). A wildcard is only special as the last pattern segment.

```
$app = web.get($app, "/static/*path", serveStatic); # nested static files
$app = web.get($app, "/*page", spaIndex);           # fallback, registered last
```

Middleware

A middleware is a handler that runs before the route handler and returns a `bool` : `true` to continue, or **respond and return false** to halt (e.g. an auth gate):

```
func requireKey(ctx as web.Context) {
    if (web.header($ctx, "X-Api-Key") == "secret") {
        return true;
    }
    web.text($ctx, 401, "unauthorized\n");
    return false;
}

$app = web.before($app, requireKey);
```

Every request is answered exactly once: if a handler throws or forgets to respond, the framework sends a `500` so the connection never hangs.

Error handling

When a handler or middleware throws, `web` contains the error (so one bad request never stops the accept loop) and answers `500` . The error is **not** dropped: by default its kind / message / position is written to **stderr** . Register an error handler to route it somewhere else (a log file, an alerting service):

```
func logError(err as Error) {
    io.eprintf("request failed: %s: %s (%s:%d)\n", $err.kind, $err.message, $err.
file, $err.line);
}
$app = web.onError($app, logError);
```

The handler receives the thrown value (the `Error` struct by convention) and runs after the `500` is already committed, so it is for observability, not for changing the response. An error thrown by the error handler itself is caught and logged, never re-raised.

Reading form bodies

`web.form($ctx)` parses an `application/x-www-form-urlencoded` body into a map, and `web.formValue($ctx, name)` reads one field. Both decode **leniently** : a `%FF` escape or a latin-1 / binary field

(legal urlencoded input) maps byte for byte rather than throwing, so a non-UTF-8 body is usable instead of a `500` . `web.csrfCheck` only falls back to the `csrf` form field for an actual urlencoded body - a JSON or multipart request is left to the `X-CSRF-Token` header.

Authentication

`web` parses the incoming `Authorization` header; **checking** the credentials (against your user store) and sending the `401` challenge stay app code.

Call	Returns	
web.basicAuth($ctx)	BasicCredentials	Decode Authorization: Basic base64(user:pass).
web.bearerToken($ctx)	string	The token from Authorization: Bearer <token> ("" if absent).

`BasicCredentials` is `{ user, password, present }` ; `present` is false when the header was missing or malformed. A Basic-auth gate is a middleware:

```
func requireLogin(ctx as web.Context) {
    def cred as web.BasicCredentials init web.basicAuth($ctx);
    if ($cred.present and checkUser($cred.user, $cred.password)) {
        return true;
    }
    web.setHeader($ctx, "WWW-Authenticate", "Basic realm=\"app\"");   # the 401
challenge is a two-liner
    web.text($ctx, 401, "unauthorized\n");
    return false;
}
$app = web.before($app, requireLogin);
```

For **bearer** tokens, `web.bearerToken($ctx)` extracts the token; validate it yourself (an opaque lookup, or `jwt.verify` once the `jwt` module lands). Client -side auth (sending `Authorization`) lives in the `rest` module (`rest.basic` / `rest.bearer`). **Digest** auth is not supported (a legacy, challenge/nonce scheme; use Basic over TLS or a bearer token).

CSRF

Stateless, HMAC-signed double-submit tokens. `web` holds no secret or session state - the **app supplies a secret** (a stable per-deployment string) and opts in with a middleware. A token is `<random>.<hmac(secret, random)>` , minted into the `csrf` cookie and echoed by the client; a request is accepted only when the submitted token equals the cookie *and* its signature verifies, so a forger without the secret cannot mint one.

Call	Returns	
web.csrfToken($ctx, secret)	string	Mint a token, set the csrf cookie, return it for the form / an X-CSRF-Token header.
web.csrfCheck($ctx, secret)	bool	True when the request carries a valid token.

Mint the token in the GET handler that renders a form; guard the unsafe methods with a middleware:

```
def const CSRF as string init os.getEnv("CSRF_SECRET");   # a stable secret

func showForm(ctx as web.Context) {
    def token as string init web.csrfToken($ctx, CSRF);
    web.html($ctx, 200, "<form method=post><input type=hidden name=csrf value=" +
$token + ">...</form>");
}

func guardCsrf(ctx as web.Context) {
    def m as string init web.method($ctx);
    if ($m == "GET" or $m == "HEAD") {
        return true;                      # safe methods
    }
    if (web.csrfCheck($ctx, CSRF)) {
        return true;
    }
    web.text($ctx, 403, "CSRF check failed\n");
    return false;
}
$app = web.before($app, guardCsrf);
```

The submitted token is read from the `X-CSRF-Token` header (for JSON / fetch clients) or the `csrf` form field (for HTML forms).

CORS

`web.cors($app, opts) -> App` sets a cross-origin policy for the whole app. When it is set, the serve loop adds the `Access-Control-*` headers to **every** response and answers a preflight `OPTIONS` request with a `204` before routing - so CORS is a one-line, app-wide policy, not something each handler repeats.

```
def opts as web.CorsOptions;
$opts.allowOrigin = "*";
$opts.allowMethods = "GET, POST, PUT, DELETE, OPTIONS";
$opts.allowHeaders = "Content-Type, Authorization";
$app = web.cors($app, $opts);
```

`opts` is a `web.CorsOptions` - `allowOrigin` (`"*"` or a specific origin; `""` leaves CORS off), `allowMethods` , `allowHeaders` , `allowCredentials` (bool), and `maxAge` (int seconds, `0` omits it). A zero-value struct is CORS off, so `web.new()` starts with no policy.

Caching

Static files are already cached: `web.serveFile` rides Go's file server, which sets `ETag` / `Last-Modified` and answers `If-None-Match` / `If-Modified-Since` (and `Range`) on its own. **Plain cache headers** are a one-liner - `web.setHeader($ctx, "Cache-Control", "max-age=3600")` - so there is no wrapper for them.

For a **dynamic** response, `web.etag($ctx, tag) -> bool` handles the conditional GET: it sets the `ETag` header and, if the request's `If-None-Match` matches, answers `304 Not Modified` and returns `true` so the handler stops before sending the body. `tag` is your choice of validator - a content hash (via `hash`), a database row version, an mtime - so `web` needs no hashing of its own.

```
func page(ctx as web.Context) {
    def body init render();
    def tag init hashOf($body);           # your validator
    if (web.etag($ctx, $tag)) {
        return;                            # 304 sent; skip the body
    }
    web.text($ctx, 200, $body);
}
```

The match is a simple exact / `*` comparison; the RFC 7232 comma-list and weak (`W/`) tag forms are not parsed.

Compression

`web.sendGzip($ctx, status, body)` answers with the body gzip-compressed when the client's `Accept-Encoding` names gzip, otherwise plain. It always sets `Vary: Accept-Encoding` (so caches don't cross the wires) and, when compressing, `Content-Encoding: gzip` . Set the `Content-Type` yourself first. Worth it for large text / JSON / HTML; skip it for already-compressed payloads (images, archives).

```
func report(ctx as web.Context) {
    web.setHeader($ctx, "Content-Type", "application/json");
    web.sendGzip($ctx, 200, json.encode($bigDocument));
}
```

Serving

Call	
web.run($app, addr)	Listen on addr and serve forever (blocks; interrupt to stop).
web.serveOn($app, srv)	Serve on an already-listening httpd.Server you hold - so you can shut it down (or serve from a spawn).

`web.run` is the common path. Use `web.serveOn` when you want the server handle, e.g. to shut down from another task:

```
def srv as httpd.Server init httpd.listen(":8080");
def worker as task of null init spawn { web.serveOn($app, $srv); };
# ... later ...
httpd.shutdown($srv);
task.wait($worker);
```

Running with jennifer serve

`jennifer serve app.j` runs a web app; `--watch` restarts it whenever the entry file changes - a Hugo-style edit / reload loop:

```
jennifer serve app.j            # run the app
jennifer serve app.j --watch    # reload on change
```

`--watch` is not web-specific: it re-runs *any* program on every change to the entry file, so it doubles as an autorun / edit-and-rerun loop for plain scripts too. See the `serve` command.

See also

- `httpd` - the server engine web is built on.
- `http` - the HTTP client module (talk to other servers).
- `json` - encode response bodies for `web.sendJson` .
- `rest` , `session` , `ratelimit` - companions for a fuller serving stack.

webapi - a JSON-API conventions layer over web

`webapi` sits on top of the bundled `web` module and supplies the pieces every JSON API otherwise re-implements by hand: a uniform error envelope, request validation (reusing `validate`), versioned route mounting, pluggable bearer auth and rate limiting, content negotiation, and pagination. `web` stays the HTTP framework; `webapi` carries the JSON-API opinions, so an application can take one without the other.

```
import "web.j" as web;
import "webapi.j" as webapi;

def app as web.App init web.new();
def api as webapi.Api init webapi.new();
$api = webapi.mount($api, 1, "/v1");
$api = webapi.alias($api, 1);
$api = webapi.authenticator($api, verifyToken);
$api = webapi.get($api, "/deck/:name", getDeck, webapi.public());
$api = webapi.post($api, "/publish", publish, webapi.Spec{
    summary: "publish a deck version",
    auth: webapi.Auth.Bearer, scopes: ["publish"],
    rules: {"tag": [validate.required(), validate.maxLen(16)]},
    rateLimit: 30, produces: webapi.Produces.Json
});

func apiGuard(ctx as web.Context) { return webapi.guard($api, $ctx); }
$app = webapi.install($api, $app, apiGuard);
web.run($app, ":8080");
```

Runnable: `examples/modules/webapi_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/webapi_demo.j).

The builder

`webapi.new()` returns a value-semantic `Api` you finish before the server runs; every builder returns a fresh `Api` .

Call	Does
webapi.mount(a, version, path)	serve the routes under path (labelled version)
webapi.alias(a, version)	additionally serve version at the bare root
webapi.deprecate(a, version, sunset)	mark a version deprecated with a date
webapi.authenticator(a, fn)	the token-verifier func value
webapi.limiter(a, fn)	the rate-limit counter func value
webapi.get/post/put/patch/delete(a, pattern, handler, spec)	register a route (handler a func value) with a Spec
webapi.feature(a, name)	label the last route for the discovery document
webapi.install(a, app, guard)	register every route (once per mount) + wire the guard func value

`webapi.public()` returns a zero `Spec` - an unauthenticated JSON route with no scopes, rules, or rate limit.

The Spec

```
export def struct Spec {
    summary as string,
    auth as Auth,                                       # Auth.None (public) or
Auth.Bearer
    scopes as list of string,                           # permissions the identity
needs
    rules as map of string to list of validate.Rule,  # request validation
    rateLimit as int,                                   # per identity per window;
0 = none
    produces as Produces                                # Produces.Json / Html /
Negotiate
};
```

`Auth` and `Produces` are enums, not strings, so a `match` on them is exhaustiveness-checked and a typo is a compile error.

The guard, and why the shim

Routes, the authenticator, the limiter, and the guard are all `func` values, called through their home interpreter so they resolve their own imports and can build a `webapi.Identity` across the module boundary. The one thing a `func` value still cannot do is **capture** the built `Api` (Jennifer has no closures yet), so `install` wires the enforcement as a `before` middleware whose `func` value you supply, backed by a one-line entry-program shim that binds the `Api` :

```
func apiGuard(ctx as web.Context) { return webapi.guard($api, $ctx); }
```

The guard, per request, matches the route, authenticates, checks scopes, validates the body, and rate-limits - answering the request and returning false on any failure, or true to run the handler. A request matching no API route passes through untouched.

The authenticator and limiter are entry-program `func` values, called through their home interpreter (so the authenticator can construct and return a `webapi.Identity`). Their signatures:

```
func verifyToken(token as string) -> webapi.Identity        # ok:false = reject
func countHit(key as string, limit as int) -> bool           # true = allowed
```

The `Spec` evaluation itself is a **pure** function you can test without a server: `webapi.evaluate(spec, identity, data) -> Decision` .

Error envelopes

Every failure uses one shape, so a client parses one thing:

```
{ "error": "no such deck" }
{ "error": "invalid request", "failures": [{ "field": "tag", "rule": "required",
"message": "is required" }] }
```

Call	Does
webapi.fail(ctx, status, message)	send the envelope at a status
webapi.failWith(ctx, status, message, failures)	with validate.Failure detail
webapi.notFound(ctx, message)	404
webapi.denied(ctx, message)	403
webapi.unauthorized(ctx, message)	401 with a WWW-Authenticate header
webapi.onError(app, name)	register a 500-envelope error handler

Request data

The guard validates a route's `rules` before the handler runs (a failure short-circuits to `422`), so a handler never

has to ask whether its input is sane. Data is read from the query, then the JSON body's top-level scalars, then the form body.

Call	Does
webapi.queryData(ctx, fields)	named query parameters as a string map
webapi.formData(ctx)	the form body as a string map
webapi.jsonData(ctx, fields)	named top-level JSON fields, flattened to strings
webapi.validated(a, ctx)	inside a handler: the checked data for this request
webapi.identity(a, ctx)	inside a handler: the authenticated caller

A nested or array-valued body is the handler's own business, read with `web.bodyJson` .

Negotiation, pagination, discovery

Call	Does
webapi.wants(ctx)	"json" or "html" from Accept
webapi.sendJson(ctx, status, value)	send a json.Value
webapi.page(ctx, defaultLimit, maxLimit)	parse + clamp offset/limit -> Page
webapi.sendPage(ctx, items, page, total)	a {items, offset, limit, total} response
webapi.discovery(a, registry, specVersion)	a discovery json.Value derived from the route table

`page` clamps `limit` to `maxLimit` so a client cannot ask for the whole table. `discovery` builds its `apis[]` and `features[]` from what is actually registered, so the advertised surface cannot drift from the routes that exist.

Placement

- Depends on `web` , `validate` , `json` , `strings` , `lists` , `maps` , `convert` , `meta` - all bundled, so no new external dependency.
- Needs the default `jennifer` binary (`web` needs `net`); it carries # `pragma-jennifer-capability: net` and is unavailable under `jennifer-tiny` .

See also: `web` , `validate` , `json` .

webhook - HMAC-signed webhooks

Import with `import "webhook.j" as webhook;` . Sign and verify webhook deliveries the GitHub / Stripe way - the **X-Hub-Signature-256** convention. A sender computes `sha256=<hex>` , the hex HMAC-SHA256 of the exact request body keyed by a shared secret, and puts it in a header; the receiver recomputes it over the body it got and compares, confirming the delivery is authentic and untampered.

`sign` / `verify` are pure and run on **both** binaries; `send` POSTs a signed payload and needs the default **jennifer** binary (`net` via `http`).

```
import "webhook.j" as webhook;

def sig as string init webhook.sign("{\"event\":\"push\"}", "topsecret");
# -> "sha256=..."  (put this in the X-Hub-Signature-256 header)

def ok as bool init webhook.verify("{\"event\":\"push\"}", $sig, "topsecret");
```

Runnable: `examples/modules/webhook_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/webhook_demo.j).

Functions

Call	Returns	Notes
webhook.sign(payload, secret)	string	sha256= + hex HMAC-SHA256 of payload keyed by secret.
webhook.verify(payload, signature, secret)	bool	True if signature matches (constant-time compare).
webhook.send(url, payload, secret)	http.Response	POST payload to url with the signature header set. Needs the default binary.

The signature covers the **raw body bytes** - sign and verify the exact string you send / receive, before any parsing. A receiver that re-serializes the body first can compute a different signature and reject a valid delivery.

`verify` uses a constant-time comparison, so a check does not leak via timing how many leading characters of the signature were correct. It returns `false` (never throws) for a wrong secret, a tampered payload, or a malformed signature.

Timestamped, replay-protected schemes

The GitHub-style `sign` / `verify` above authenticate the body but carry no timestamp, so a captured request can be replayed forever. The schemes below fold a **unix timestamp** into the signed data and reject a delivery whose timestamp drifts more than `toleranceSeconds` from `now` - the timestamp tolerance **is** the replay defence. All of them constant-time compare, run on **both** binaries, and are pure: pass `now` and `timestamp` as explicit `int` unix seconds (a script reads the clock from `time` and passes it in), which keeps the functions testable and deterministic.

Call	Returns	Notes
webhook.stripeSign(secret, body, timestamp)	string	Stripe t=<ts>,v1=<hexsig> header; signs <ts>.<body> HMAC-SHA256, hex.
webhook.stripeVerify(secret, body, header, toleranceSeconds, now)	bool	Parse t + the v1= sig(s), recompute, constant-time compare; true only if a sig matches and |now - t| <= tolerance.
webhook.slackSign(secret, body, timestamp)	string	Slack v0=<hexsig>; signs "v0:" + <ts> + ":" + <body> HMAC-SHA256, hex.

webhook.slackVerify(secret, body, timestamp, signature, toleranceSeconds, now)	bool	Recompute over the X-Slack-Request-Timestamp value; true only if it matches and the timestamp is fresh (e.g. 300s).
webhook.timestampedSign(secret, body, timestamp, algo, encoding)	string	Generic: sign <ts>.<body> with algo ("sha1"/"sha256") and encoding ("hex"/"base64").
webhook.timestampedVerify(secret, body, timestamp, signature, algo, encoding, toleranceSeconds, now)	bool	Constant-time compare + freshness check for the generic scheme.

```
import "webhook.j" as webhook;
use time;

def body as string init "{\"event\":\"payment\"}";
def secret as string init "whsec_...";
def now as int init time.unix(time.now());

# Sign (sender side): stamp with the current time.
def header as string init webhook.stripeSign($secret, $body, $now);
# -> "t=...,v1=..."  (put this in the Stripe-Signature header)

# Verify (receiver side): recompute, constant-time compare, reject stale.
def ok as bool init webhook.stripeVerify($secret, $body, $header, 300, $now);
```

Stripe (`t=,v1=`): the signed payload is `<timestamp>` + `"."` + `<body>` , HMAC-SHA256, hex. A header may carry several `v1=` values (a signing-secret rotation); `stripeVerify` accepts the delivery if **any** matches and the `t` value is fresh. A missing `t=` or `v0=` / `v1=` part, or a non-numeric timestamp, yields `false` rather than throwing.

Slack (`v0=`): the base string is `"v0:"` + `<timestamp>` + `":"` + `<body>` , HMAC-SHA256, hex; the timestamp travels separately in `X-Slack-Request-Timestamp` (parse it to an `int` and pass it in). Slack's own guidance is a 5-minute (`300`) tolerance.

Generic (`timestampedSign` / `timestampedVerify`): the small composable core

- sign `<timestamp>` + `"."` + `<body>` with any supported digest and encoding, so

a GitHub-style `sha1` / `sha256` scheme or a base64 scheme is one call. Hex is case-folded on the built-in schemes; base64 is compared exactly.

Every verify does a **constant-time** compare (via `crypto.hmacEqual`), so no scheme leaks through timing how many leading bytes of the signature matched, and the freshness window is checked independently so a replayed-but-correctly-signed request is still rejected.

Sending

`webhook.send` posts the payload as `application/json` with the `X-Hub-Signature-256` header, and returns the receiver's `http.Response` (status / headers / body). Reading the result needs `import "http.j"` for the type:

```
import "webhook.j" as webhook;
import "http.j" as http;

def r as http.Response init webhook.send("https://example.com/hook",
    "{\"event\":\"push\"}", "topsecret");
io.printf("delivered: %d\n", $r.status);
```

A non-2xx status comes back as a value to branch on; a network failure throws a positioned `http` / `net` error

(wrap in `try` / `catch`).

Notes and scope

- **SHA-256, X-Hub-Signature-256.** This is the modern GitHub convention. The legacy `X-Hub-Signature` (SHA-1) header is not emitted; sign with SHA-256.
- **The secret is the shared key** - store it like a password and distribute it over a secure channel. Randomness for a new secret is out of scope (use a cryptographic source; `math.rand*` is not crypto-grade).
- **Content type is application/json** for `send` . For a different body type, `sign` the payload yourself and post it with your own headers via `http` .

See also

- hash.md - the `hmac` primitive the signature is built on.
- http.md - the client `send` posts through.
- totp.md - the other `hash.hmac` -based module (2FA codes).
- modules/index.md - the module catalog and import rules.

websocket - RFC 6455 WebSocket client

Import with `import "websocket.j" as websocket;` . Open a WebSocket connection, send and receive text / binary messages, and close it - a full RFC 6455 client over `net` . `connect` performs the HTTP `Upgrade` handshake and **verifies** the server's `Sec-WebSocket-Accept` (SHA-1 + base64 of the client key); `send` / `sendBytes` write masked frames (client-to-server frames must be masked); `receive` reads the next message, answering pings with pongs and reassembling fragmented messages. Needs the default `jennifer` binary. A protocol error or dropped connection throws `Error{kind: "websocket"}` .

```
import "websocket.j" as websocket;

def ws as websocket.Conn init websocket.connect("wss://ws.postman-echo.com/raw");
websocket.send($ws, "hello");
def m as websocket.Message init websocket.receive($ws);   # m.kind "text", m.text
"hello"
websocket.close($ws);
```

Runnable: `examples/modules/websocket_demo.j` (https://github.com/jennifer-language/jennifer/blob/main/examples/modules/websocket_demo.j).

Connecting

`ws://` is a plain TCP connection; `wss://` is TLS (`net.connectTLS`). The port defaults to 80 / 443 and may be overridden in the URL (`ws://host:9001/path`).

Call	Returns	
websocket.connect(url)	Conn	handshake with the default receive timeout (30 s)
websocket.connectWith(url, timeoutMs)	Conn	handshake with an explicit per-receive timeout

```
def struct websocket.Conn { socket as net.Conn, timeoutMs as int };
```

The handshake sends a random 16-byte `Sec-WebSocket-Key` and rejects the connection (throws) unless the response is `101` and its `Sec-WebSocket-Accept` matches `base64(SHA1(key + GUID))` .

Sending and receiving

Call	Returns	
websocket.send(c, text)		send a text message
websocket.sendBytes(c, data)		send a binary message
websocket.ping(c)		send a ping
websocket.receive(c)	Message	read the next message
websocket.close(c)		send a close frame and shut the socket

Outgoing frames are masked with a fresh key (as the spec requires) and length- encoded in the 7-bit, 16-bit, or 64-bit form automatically. `receive` returns a `Message` :

```
def struct websocket.Message {
    kind as string,   # "text", "binary", "close", or "pong"
    text as string,   # decoded text (for a "text" message; "" otherwise)
    data as bytes     # the raw payload bytes
};
```

`receive` transparently answers a **ping** with a pong and keeps reading, and reassembles a **fragmented** message (continuation frames) into one `Message` . A **close** frame surfaces as `kind` == `"close"` - stop reading and `close` the connection. Each `receive` is bounded by the connection's `timeoutMs` ; a timeout throws.

```
def m as websocket.Message init websocket.receive($ws);
if ($m.kind == "close") {
    websocket.close($ws);
} elseif ($m.kind == "text") {
    # handle $m.text
}
```

Scope

- **Client only.** A server-side upgrade would need an `httpd` connection-hijack hook (a separate, larger piece).
- **Crypto-grade masking key / nonce.** The 4-byte frame mask and the handshake nonce come from the `crypto` library, as RFC 6455 (section 5.3) requires: the mask must be "derived from a strong source of entropy" and hard to predict frame to frame.
- **No permessage-deflate / extensions, no subprotocol negotiation.** The `Sec-WebSocket-Protocol` / `-Extensions` headers are not sent.
- **receive blocks per message** up to `timeoutMs` ; there is no non-blocking poll (drive one message at a time, or run `receive` in a `spawn`).
- **Bounded connect and message size.** The opening handshake is bounded by a connection-establishment timeout (a slow or unreachable server fails the dial), and a received message is capped at 64 MiB - a peer that declares an oversized frame or reassembled message is refused with a catchable error, not an unbounded allocation.

See also

- net.md - the TCP / TLS transport this is built on.
- http.md - the HTTP/1.1 client (a different, request/response protocol over the same `net`).
- modules/index.md - the module catalog and import rules.

Technical reference

Jennifer Interpreter - Technical Documentation

Internals of the Jennifer interpreter. This directory is split by topic; each page is small enough to read in one sitting.

Design stances

The interpreter's shape follows from seven language-design decisions. When the implementation seems unusual - the `def` -site bare-name rule, the strict no-truthiness check at every `if` / `while` / `for` , deep-copy on list/map assignment, the per-topic library directories - the reasons trace back to one of them.

See docs/design-stances.md for the canonical table.

Contents

- Lexer - hand-written scanner, token kinds, position tracking, identifier rules.
- Grammar and parser - the EBNF the parser accepts, plus the AST node table and parser implementation notes.
- Preprocessor - how `import "file.j";` is expanded before the parser runs.
- Interpreter - runtime values, scoped environment, execution model, library/builtins, error model.
- CLI - subcommands (`run` , `repl` , `tokens` , `ast` , `fmt` , `lint` , `profile` , `test` , `version` , `help`), each with its own page (REPL, Inspection, Formatter, Linter, Profiler, Test runner) linked from the index, plus version injection.
- Testing - which packages test what.
- File map - one-line description of every file in the repository.
- Rejected features - proposals that were turned down, with the reasoning so they don't come back.
- Design decisions - features that ship despite looking like a stance violation at first glance; the reasoning that shows they aren't one.
- TinyGo notes - the constraints TinyGo imposes and how the codebase respects them.
- Benchmarks - single-binary throughput (Go vs TinyGo), parallelism, and the memory / page-fault trade, from the reference machine.
- Security model - what a Jennifer script can do by design, what counts as a real vulnerability, and how to run untrusted code.
- Glossary - canonical project terminology (function/method, library/module, list/array, ...). The terms this file uses match it.

Pipeline

The compiler/interpreter is a six-stage pipeline (scope analysis sits between parse and evaluation):

```
    source (string)
        |
        ?
    +--------+
    | lexer  |   internal/lexer
    +--------+
        | []Token
        ?
    +--------------+
    | preprocessor |   internal/preproc   (splices file imports)
    +--------------+
        | []Token
        ?
    +--------+
    | parser |   internal/parser
    +--------+
        | *Program (AST)
        ?
    +----------+
    | resolver |   internal/parser (Resolve)
    +----------+    scope pass: numbers slots per frame,
        |           annotates every reference with (Depth, Slot),
        |           promotes undefined-variable / shadowing errors
        |           to parse-time diagnostics. Idempotent; REPL
        |           bypasses and falls back to name-based lookup.
        | resolved *Program
        ?
    +-------------+
    | interpreter |   internal/interpreter + internal/lib/io (and other libs)
    +-------------+
        |
        ?
      stdout / runtime error
```

The CLI lives in `cmd/jennifer/main.go` and orchestrates these stages. For the user-facing language reference (types, control flow, libraries), see ../user-guide/. For the style rules `jennifer fmt` enforces, see ../user-guide/style-guide.md.

Lexer (internal/lexer)

A hand-written, single-pass scanner.

Token types

Group	Tokens
Markers	EOF, ILLEGAL
Literal values	INT, FLOAT, STRING, STRING_INTERP, TRUE, FALSE, NULL
Identifiers	IDENT, VARREF
Declaration keywords	DEFINE (def), FUNC, AS, INIT, CONST, RETURN
Import keywords	USE, IMPORT
Control-flow keywords	IF, ELSEIF, ELSE, WHILE, FOR
Type keywords	INT_TYPE, FLOAT_TYPE, STRING_TYPE, BOOL_TYPE, LIST, MAP
Type structure keywords	OF, TO
Iteration keyword	IN
Keyword operators	AND, OR, NOT
Arithmetic operators	PLUS (+), MINUS (-), STAR (*), SLASH (/), DIV (//), PERCENT (%)
Comparison operators	LT (<), GT (>), LE (<=), GE (>=), EQ (==), NEQ (!=)
Assignment	ASSIGN (=)
Grouping and punctuation	LBRACE ({), RBRACE (}), LPAREN ((), RPAREN ()), LBRACKET ([), RBRACKET (]), SEMI (;), COMMA (,), COLON (:), DOT (.), DOTDOT (..)

`def` introduces a variable or constant binding (TOKEN_DEFINE); `func` introduces a method (TOKEN_FUNC). `import` (TOKEN_IMPORT) is for **file imports** (`import "path.j";`); `use` (TOKEN_USE) is for **library imports** (`use io;`). `DOT` (`.`) no longer appears in import syntax (paths are strings now) and is reserved for future expression use. Comparison tokens `LE` , `GE` , `EQ` , `NEQ` are two-character (<= , >= , == , !=) and are recognized by a one-character lookahead from < , > , = , `!` . `!` exists **only** as the lead of != (logical negation is the word `not`); a bare `!` is a positioned lex error whose message points at both `not` and != . `RETURN` is the keyword behind `return [EXPR];` (see grammar.md).

`VARREF` carries the variable name *without* the leading $. `STRING` carries the value *without* surrounding quotes. A **double-quoted** literal is **cooked** - escape sequences are already processed: `\n` `\r` `\t` `\\` `\"` `\'` `\0` `\{` `\}` , plus the Unicode escapes `\uXXXX` (exactly 4 hex digits, the BMP) and `\UXXXXXXXX` (exactly 8 hex digits, any plane). `readUnicodeEscape` rejects a surrogate, a value above `U+10FFFF` , and a short / non-hex digit run as a positioned lex error (matching `convert.fromCodepoint`). A **single-quoted** literal is **raw** - its content is verbatim (no escape processing, no interpolation) from the opening `'` to the next `'` . The two delimiters are thus not interchangeable; `readString` branches on the quote into `readRawString` or `readCookedString` .

A cooked string also **interpolates** : an unescaped `{expr}` is a slot. `readCookedString` decomposes the literal into ordered literal chunks and expression slots. With no slots it returns a plain `STRING` (the decoded value); with one or more it returns `STRING_INTERP` , whose `Parts []StringPart` carry the literal chunks and the raw expression source of each slot plus that slot's **absolute** `Line` / `Col` (so the parser's sub-lex / sub-parse annotates references and reports errors at the real location - `TokenizeAt` seeds the position). `scanInterpSlot` brace-counts nested `{}` (a map literal in a slot balances) and copies a nested string literal verbatim (`copyNestedString`) so a brace or quote inside it never disturbs the count; `\{` / `\}` are literal braces and a bare unescaped `}` is a positioned lex error. A raw `'...'` string is returned verbatim as a plain `STRING` , never interpolated.

Whitespace handling

Spaces, tabs and newlines are discarded between tokens; they only ever advance `Line` / `Col` for position

tracking. There is no indentation-significant mode and no off-side rule. The user-facing consequence is documented in user-guide/syntax.md > Tokens and whitespace; the rule is load-bearing for `jennifer fmt`, which trusts that re-emitting the token stream with canonical spacing produces a semantically identical program.

The only place whitespace is *retained* is inside string literals - `readString` reads byte-by-byte until the closing quote, so a literal space, tab, or even a raw `\n` between the quotes becomes part of the string value (in both the cooked `"..."` and the raw `'...'` form). In a raw single-quoted literal a backslash is just a byte, so `'\n'` is the two-character string `\n`, and a multi-line block needs no escapes at all; `fmt` re-emits every literal in its original delimiter and spelling (via `Token.Raw`).

Comments and blank lines are emitted as **trivia tokens** (`TOKEN_COMMENT_LINE`, `TOKEN_COMMENT_BLOCK`, `TOKEN_COMMENT_SHEBANG`, `TOKEN_BLANK_LINE`) so `jennifer fmt` can round-trip them. The preprocessor and parser strip these tokens at entry; the formatter walks the raw lexer stream. See Comments below.

Position tracking

Every token records `Line` and `Col` (both 1-based) and `File` (the absolute path supplied to `TokenizeWithFile`, or `""` for unattributed input). The `advance()` helper bumps `line` on `\n` and otherwise bumps `col`. `File` flows from the token to the AST node (every node embeds `pos{File, Line, Col}`), so errors raised inside an imported `.j` still point at the imported file - see Interpreter > Errors and positions.

Keywords

The lexer's keyword map covers: `def` `func` `as` `init` `const` `import` `use` `return` `if` `elseif` `else` `while` `for` `true` `false` `null` `and` `or` `not` `int` `float` `string` `bool`. Anything else lexed as a word stays a `TOKEN_IDENT`. `define` is **not** a keyword and lexes as a plain identifier. `div` was removed when `//` took over floor division.

Comments

`# ...` runs to end of line and emits `TOKEN_COMMENT_LINE`; the special case of `#!` on line 1 col 1 emits `TOKEN_COMMENT_SHEBANG` instead so the formatter can re-emit the shebang verbatim at the file head. `/* ... */` emits `TOKEN_COMMENT_BLOCK` and **nests** via a depth counter (increment on `/*`, decrement on `*/`, exit at depth 0). Unterminated nested comments error positionally at the outermost `/*` so the message points at where the user meant to start.

Each comment token's `Lexeme` carries the verbatim source text including the delimiters (`# ...`, `/* ... */`, `#! ...`) so the formatter round-trips byte-for-byte.

Runs of blank lines collapse to one `TOKEN_BLANK_LINE` per run - matching the style rule "never more than one consecutive blank line".

was chosen (over the C/Java `//` style) so the floor-division operator `//` is unambiguous and a Jennifer file can begin with a Unix shebang (`#!/usr/bin/env -S jennifer run`).

A `.` peeks for a second `.`: two together lex as one `DOTDOT` (the half-open range operator `a..b`), a single `.` stays `DOT`. Number scanning already requires a digit after `.` for a fraction, so `1..5` lexes as `INT DOTDOT INT` (not `1. .5`) with no special-casing.

A decimal number may carry a scientific-notation exponent, `[eE][+-]?` followed by one or more digits (`6.022e23`, `1.6e-19`, `1e10`). An exponent makes the literal a `FLOAT` even with no fractional part, so `1e10` is a float where a bare `1` is an int; the exponent takes no `_` separators (the mantissa still does, `1_000.5e3`), and a digit followed by `e` / `E` with no exponent digit (`1e`, `1e+`) is a positioned lex error, since it has no other valid reading. Only the decimal path scans an exponent - `0x` / `0o` / `0b` literals return earlier, so `0xe5` 's e stays a hex digit. The stored `Lexeme` keeps the exponent's source case (`2.5E8` stays `2.5E8`, which `strconv.ParseFloat` accepts) and strips only the mantissa's `_`; `Raw` keeps the exact source so `fmt` re-emits `6.022e23` / `1.6E-19` verbatim.

The parser's `strconv.ParseFloat` governs the magnitude boundaries. An **overflowing** exponent (`1e400`) is a positioned parse error, never an `Infinity` (the strict stance rejects non-finite results). An **underflowing** one (

`1e-400` , below the smallest denormal ~ `5e-324`) rounds to `0.0` with no error - `0.0` is a finite, correctly-rounded value, unlike the non-finite `Infinity` overflow would give, so it is the same value every IEEE-754 language yields there. The asymmetry is deliberate: the strict check bans the non-finite, not a finite zero.

Identifier rule

Every identifier is **letter-initial** (a token that starts with a digit is always a number, which keeps lexing unambiguous). Variable, method, parameter and library names are `[A-Za-z][A-Za-z0-9]{0,63}` - a letter, then letters and digits, no underscores. Constants use a looser form: uppercase chunks separated by single `_` characters, with in-chunk digits - `[A-Z][A-Z0-9]*(_[A-Z][A-Z0-9]*)*` . Each chunk starts with an uppercase letter (never a digit), and every `_` must be immediately followed by that letter, so leading, trailing, consecutive, and underscore-then-digit forms are all rejected (`SHA256` / `SCRAM_SHA256` are legal, `AES_256` is not - write `AES256`).

The lexer reflects this by accepting digits and `_` as continuation characters for bare IDENT tokens (so `sha256` and `MAX_RETRIES` are each a single token) but rejecting any identifier that *ends* with `_` . The full per-kind rule is then enforced by the parser at each def / use site - variables, methods, parameters, library names and call callees may not contain `_` ; constants may, with the leading- `_` case already excluded by `isIdentStart` . `$var` references go through a separate lexer path (`readVarRef`) whose `isIdentPart` allows the leading letter plus digits but not `_` , so `$x2` lexes and `$foo_bar` lex-errors directly.

Resource limit: the token budget

`TokenizeWithFile` caps how many tokens a single source file may produce at `limits.MaxTokens` (build-tag split, like the parser's nesting cap: `1<<22` on the default binary, `1<<18` on `jennifer-tiny`). Past the cap it returns a positioned, catchable `LexError` ("token budget exceeded"). This bounds the front of the pipeline against a *token bomb* : a small, dense file (a few MB of `1+1+1+1;`) amplifies to millions of ~72-byte `Token` structs - roughly 100x the input byte count - which the parser then walks again. The over-long-identifier early-exit (a single lexeme is capped at 64 chars) bounds one token; this bounds the token *stream* .

It complements the preprocessor's spliced-token budget (`maxSplicedTokens`), which counts tokens across `include` splices but does **not** count the entry file's own tokens - so a giant root program with no includes would otherwise reach the parser unbounded. The cap is a package var defaulting to the const only so a test can lower it without generating millions of tokens; production code never writes it.

Grammar and parser

The authoritative grammar for what the parser accepts, plus a quick tour of the AST node table and the parser's structure.

The grammar, in two notations

The grammar lives in two companion files, one per notation:

- **EBNF** - the declarative view: what shapes the language contains. Start here to learn or look up the syntax.
- **PEG** - the operational view: ordered choice and predicates encoding exactly the decisions the recursive-descent parser makes (which alternative wins, and which orderings are load-bearing). Start here to understand or modify the parser.

Both describe the token stream **after** preprocessing (`include` splices are already expanded; `use` / `import` do reach the parser), with terminals written as lexer token classes (Lexer > Token types). They describe the same language and must be kept in sync - when the syntax changes, update **both** , in the same commit that changes the parser. The parser (`internal/parser/parser.go`) is the source of truth.

The semantic rules no grammar can express follow here.

Semantic notes that aren't expressed in the grammar:

- Two separate keywords: `def` introduces a binding (variable or constant); `func` introduces a method. There's no longer any lookahead disambiguation in this area - the parser dispatches purely on the keyword.
- The name in `defineStmt` is a bare `IDENT` . Writing `def $x as int` produces a parse error with a hint to drop the `$` (it's reserved for use-site references).
- Operator precedence (lowest to highest): `or` , `and` , unary `not` , comparison < > <= >= == != , bitwise or `|` , bitwise xor `^` , bitwise and `&` , shifts << >> , additive `+` `-` , multiplicative `*` `/` `//` `%` , unary `-` and `~` . Binary operators are left-associative; `not` and the prefix operators are right-associative (`not not x` , `--x` , and `-~x` are all valid). The bitwise family (`|` `^` `&` << >> `~`) operates on `int` only.
- `and` and `or` **short-circuit** : the right operand is not evaluated when the left already decides the result. Both operands must be `bool` .
- Unary `not` requires `bool` ; unary `-` requires `int` or `float` .
- Comparison operators produce `bool` ; `if` / `while` / `for` conditions **must** be `bool` (no implicit truthiness). The ordering operators (< > <= >=) compare two numbers or two strings (strings lexicographically by UTF-8 byte); a string/number mix is a type error.
- Mixed `int` / `float` arithmetic promotes `int` to `float` ; the result is `float` . `%` requires int operands. `+` on two `string` values concatenates.
- **`/` (true division) always returns float** (Python 3 semantics). For integer-result division use `//` : `5 / 2` = `2.5` , `5 // 2` = `2` . `//` on float operands returns the floor as a float (`5.7 // 2.0` = `2.0`). Line comments are `#` (not `//`), which leaves `//` free for the operator and lets a Jennifer file start with a shebang (`#!/usr/bin/env -S jennifer run`).
- Floats always display with a `.` so the type stays visible: `5.0` prints as `"5.0"` , not `"5"` . See `interpreter.DisplayFloat` .
- Methods may only be defined at the top level. Variable definitions, assignments, control flow, and expression statements may appear at the top level or inside a block.
- Each `block` (`{...}`) introduces a new lexical scope. A binding is visible from its `def` to the end of the enclosing block, and is inherited by inner blocks; inner scopes **cannot redeclare** a name already visible.
- `for` opens a private scope for its `init` , `cond` , `step` , and body so the init variable does not leak out of the loop.
- There is **no required entry point** . Top-level statements execute in source order. Methods are hoisted (collected before any top-level statement runs) so they can be called regardless of textual order.
- Method bodies inherit the global scope as their outer scope, so top-level variables are visible inside methods (subject to the no-shadowing rule).
- Method parameters use bare `IDENT` (no `$`), same as variable definitions. Writing `func f($x as int)`

errors with "parameter name has no $ ".
- Call sites type-check arguments against the declared parameter types at runtime; both arity and per-argument kind are checked.
- Method return values are dynamically typed - methods don't declare a return type, and callers receive whatever value the body returns (or `null` for a bare `return;` or a body that falls off the end).
- A bare `IDENT` in expression position is parsed as a `CallExpr` if immediately followed by `(` , otherwise as a `ConstRefExpr` . At runtime the latter resolves to a constant in scope, else to a **first-class function value** if the name is a top-level method (the `func` type), else a name that resolves to a variable produces a helpful error ("use `$name` "); anything else is an undefined-name error.
- A postfix `(...)` on any value expression (not a bare method-name call, which parseCallTail consumes) is a **CallValueExpr** - a call through a function value: `$f(x)` , `$fns0(x)` , `makeAdder(1)(2)` . The callee must evaluate to a non-null `func` ; arity and argument kinds are checked at the call site exactly as for a named call.
- **Lists are array-backed sequences, not Lisp-style linked lists** : `def xs as list of int init [1, 2, 3]` . Element access is `$xs[i]` , 0-indexed, in-bounds-checked. Out-of-bounds reads and writes are positioned runtime errors.
- **Maps preserve insertion order** : iteration via `for (def k in $m)` visits keys in the order they were first inserted; updating an existing key does not move it; appending a new key extends. Reads of missing keys are runtime errors - use `has($m, key)` to test.
- **Lists and maps are value-typed** : `$ys = $xs;` copies, function parameters bind by copy, and `const` is deep (constness extends to every nested element). Aliasing is impossible; mutations through `$xs[i] = ...` only affect that binding.
- **Index assignment** (`$xs[i][j] = ...`) walks the chain on a copy of the root binding's value, applies the write, and stores the result back via `env.Assign` . The const-target check fires once against the root binding; deep constness falls out of the value-semantics invariant.
- **Iteration** (`for (def x in $coll)`) opens a fresh scope per iteration. The loop variable is bound to each element (list) or key (map). The collection is evaluated once at loop entry; concurrent mutation of the original binding during iteration doesn't affect the walk because the iterator works against a snapshot.
- **{ is overloaded** : it opens a block in statement position and a map literal in expression position. The parser disambiguates by context; the formatter (which doesn't run the parser) tracks the classification through a small stack so both forms get the right indentation and spacing.

Parser (internal/parser)

Recursive descent with precedence climbing for binary operators. The grammar the parser implements is documented twice: as EBNF (the shapes) and as PEG (the decision order); the PEG is the closer mirror of the code, down to which `case` order and peeks are load-bearing.

The exported entry points (`Parse` , `ParseTokens`) return a raw `*Program` without running the scope-analysis pass. Callers that intend to execute the program must invoke `parser.Resolve(prog)` themselves (`Interpreter.Run` does this automatically). Splitting the two lets grammar tests focus on parse trees without wiring up scope context for every fragment; see scope analysis below.

AST nodes

Node	Kind	Fields
Program	root	Imports []*ImportStmt, Methods []*MethodDef, Structs []*StructDef, Enums []*EnumDef, TopLevel []Stmt, NumGlobals int
ImportStmt	stmt	Name, AsName (empty unless use NAME as ALIAS;)
MethodDef	stmt	Name, Params []Param, Body *Block
Param	-	Name, Type
StructDef	stmt	Name, Fields []StructField (top-level only, hoisted before execution)
StructField	-	Name, Type (each field of a struct definition)

EnumDef	stmt	Name, Variants []EnumVariant (top-level only, hoisted; a sum type)
EnumVariant	-	Name, Fields []StructField (payload-less when empty)
Block	stmt	Stmts []Stmt, NumSlots int (hint used by NewEnvironmentSized)
DefineStmt	stmt	IsConst, VarName, VarType Type, InitExpr Expr (nil = uninit), Slot int (-1 = unresolved)
AssignStmt	stmt	VarName, Value Expr, Depth, Slot (both -1 = unresolved)
IndexAssignStmt	stmt	Target *IndexExpr, Value Expr - $xs[i][j] = ... (chain may include FieldAccessExpr nodes)
FieldAssignStmt	stmt	Target *FieldAccessExpr, Value Expr - $p.field = ...
TryStmt	stmt	Body *Block, CatchName, CatchBody *Block, CatchSlot (slot for CatchName in the handler frame) - try { ... } catch (NAME) { ... }
ThrowStmt	stmt	Value Expr - throw EXPR;
AppendStmt	stmt	Target *VarExpr, Value Expr - $xs[] = item;
ReturnStmt	stmt	Value Expr (nil for bare return;)
IfStmt	stmt	Cond, Then *Block, ElseIfs []Expr, ElseIfBodies []*Block, Else *Block
MatchStmt	stmt	Subject Expr, Arms []MatchArm, Else *Block (nil if absent) - match (EXPR) { when ... else ... }
MatchArm	node	Values []Expr (value arm, compared by ==) OR Variant/Bind/BindSlot (enum-pattern arm, set by the resolver), Body *Block
WhileStmt	stmt	Cond, Body *Block
ForStmt	stmt	Init Stmt, Cond Expr, Step Stmt, Body *Block (any may be nil)
ForEachStmt	stmt	VarName, Coll Expr, Body *Block, IterSlot (slot for the iterator in each iteration frame)
ExprStmt	stmt	Expr
IntLit	expr	Value int64
FloatLit	expr	Value float64
StringLit	expr	Value string
BoolLit	expr	Value bool
NullLit	expr	-
VarExpr	expr	Name (no $), Depth, Slot (both -1 = unresolved, use name lookup) - mutable-variable reference
ConstRefExpr	expr	Name, Depth, Slot (-1 = unresolved) - bare-IDENT reference; interpreter expects it to resolve to a constant
CallExpr	expr	Callee, Args []Expr, Method *MethodDef (pre-resolved pointer for hoisted user methods; nil for builtins and resolver-less paths)
CallValueExpr	expr	Callee Expr, Args []Expr (a call through a function value: $f(x); the callee evaluates to a func at run time)
LenExpr	expr	Operand Expr - len(EXPR) language built-in
QualifiedCallExpr	expr	Prefix, Callee, Args []Expr, Fn any (pre-resolved Builtin; nil for resolver-less paths)
QualifiedConstRefExpr	expr	Prefix, Name, Const any (pre-resolved Value; nil for resolver-less paths)
BinaryExpr	expr	Op BinaryOp, Left, Right, Folded Expr (pre-computed fold result; nil for runtime-only exprs)
UnaryExpr	expr	Op UnaryOp (OpNeg/OpNot/OpBitNot), Operand, Folded Expr
ListLit	expr	Elements []Expr - [1, 2, 3]

MapLit	expr	Keys []Expr, Values []Expr (parallel) - {"a": 1}
IndexExpr	expr	Target Expr, Index Expr - $xs[i], chained
RangeExpr	expr	Lo Expr, Hi Expr - half-open lo..hi; materialises list of int (or iterates lazily as a for-each source)
SliceExpr	expr	Target Expr, Lo Expr, Hi Expr (either endpoint nil for an open end) - $xs[a..b], $xs[a..], $xs[..b], $xs[..]
StructLit	expr	NS, Enum, Name, Bare, Fields []StructLitField - Point{...} / lib.Point{...}, reused for enum construction Shape.Circle{...} (NS=enum) and cross-module mod.Shape.Circle{...} (Enum set); Bare marks a payload-less form
StructLitField	-	Name, Expr (one named field in a struct literal)
FieldAccessExpr	expr	Target Expr, Field - $p.field, chainable with IndexExpr

Every node embeds a `pos{File, Line, Col}` for error reporting and exposes it via `Node.Pos()` (line/col) and `Node.Filename()` (file path). The file is populated from the originating token so cross-file diagnostics work.

`Sprint(node)` produces a stable textual representation used by tests.

Scope analysis

`internal/parser/resolver.go` is a post-parse pass that walks the AST and fills in the slot fields (`Depth` , `Slot` , `NumSlots` , `Program.NumGlobals` , `Block.NumSlots` , etc.). It also promotes two classes of error from first-execution runtime errors to positioned parse-time diagnostics:

- **Undefined variables** - `Resolve` walks its own scope stack in parallel with the AST and reports any `VarExpr` / `AssignStmt` whose name isn't in scope.
- **Shadowing** - a `def` (variable or constant) whose name is already visible in an enclosing scope. Same rule the runtime's name-based `Define` used to enforce; now caught earlier.

The resolver is idempotent: running twice on the same AST produces the same annotations. `Interpreter.Run` calls it before any structural check; `EvalInteractive` (REPL) does not (each REPL turn lacks the accumulated global context that would let resolution succeed). The runtime handles the resolver-less path by leaving all slot fields at the `-1` sentinel and using name-based Environment methods.

Scope-frame model. The resolver tracks scopes as a stack. Each frame carries a name -> slot map and a `count` allocator. A frame is `isRoot=true` at the boundaries where the runtime chain jumps directly to globals (the globals frame itself, and a method's callFrame). Reference lookup walks innermost-out, respects those root boundaries, and terminates at globals.

Three scope-shape carve-outs where the resolver deliberately deviates from "one AST scope = one runtime frame" to stay aligned with the interpreter:

- **try body** runs in the enclosing env at runtime; the resolver walks its stmts inline in the current scope rather than pushing a fresh frame.
- **For-header init** lands in `forEnv` (a frame the resolver pushes for the header), body lands in a nested body-frame (pushed by `resolveBlock`).
- **Spawn body** is skipped entirely. The runtime's two-frame spawn snapshot doesn't line up with a static single-frame view of the enclosing scope, so references inside a spawn body stay at `(Depth=-1, Slot=-1)` and the interpreter falls back to name-based lookup at runtime.

Method-call pre-resolution. The resolver also pre-fills `CallExpr.Method` whenever the callee names a hoisted top-level user method. The interpreter's `evalCall` consults the pointer first and skips the `i.methods` hash lookup on every recursive call. Builtins keep `Method` = `nil` because the namespaced / global registries need the runtime `use` -activation check.

Namespaced-call pre-resolution. For `QualifiedCallExpr.Fn` and `QualifiedConstRefExpr.Const` the pre-fill happens on the interpreter side, not in the parser resolver, because the namespace / import tables don't

exist until `processImports` has run. `Interpreter.resolveQualifiedRefs` is a second pass invoked from `Interpreter.Run` after `processImports` that walks the same AST and stamps the exact `Builtin` / `Value` a call would otherwise look up.

Constant folding. `internal/parser/fold.go` runs from inside `Resolve` as a post-step on `BinaryExpr` / `UnaryExpr` . When both operands are literal (checked transitively through their own `Folded` fields via `asLit`), the operator is applied at parse time and the result stamped on `Folded` as a fresh literal node. Chains collapse in a single pass - `((1+2)*3)+4` folds to `IntLit(11)` . Operations that would error at runtime (division by zero, negative shift count, unknown op) leave `Folded` nil so the runtime hits the error at its actual source position.

See interpreter.md > Environment for the runtime side.

Grammar - EBNF

The declarative view of the grammar: what shapes the language contains, in ISO-style EBNF. For how the parser *decides* between alternatives (ordered choice, lookahead), see the PEG grammar - the two describe the same language and must be kept in sync; the parser (`internal/parser/parser.go`) is the source of truth for both. Semantic notes that no grammar expresses (precedence prose, scoping, value semantics) stay in grammar.md.

This grammar describes the token stream **after** preprocessing - file splices (`include STRING ;`) are expanded before the parser runs, so they don't appear here. Library imports (`use IDENT ;`) and module imports (`import STRING [ "as" IDENT ] ;`) do reach the parser: `use` becomes an `ImportStmt` , `import` a `ModuleImportStmt` .

Terminals in CAPITALS are token classes from the lexer (see Lexer > Token types); quoted strings are keywords or punctuation that match the corresponding token's lexeme.

```
program     = { useStmt | moduleImport | exported | methodDef | structDef |
enumDef | statement } EOF ;
exported    = "export" ( methodDef | structDef | enumDef | constDefine ) ;
                                       (* `export` publishes a name from a
                                          module; it may only precede a
                                          `func`, `def struct`, `def enum`, or
                                          `def const`.
                                          Whether a program may contain
                                          `export` at all (module vs script)
                                          is decided at load time, not by the
                                          grammar. *)
useStmt     = "use" IDENT [ "as" IDENT ] ";" ;      (* library import; the
                                                       optional "as ALIAS"
                                                       renames the namespace
                                                       at the use site *)
moduleImport = "import" STRING [ "as" IDENT ] ";" ;  (* module import; the
                                                       STRING path must end
                                                       in ".j". Top-level
                                                       only - a module is a
                                                       declaration, so an
                                                       import inside a block
                                                       is a parse error *)
methodDef   = "func" IDENT "(" [ paramList ] ")" block ;
paramList   = param { "," param } ;
param       = IDENT "as" type ;
block       = "{" { statement } "}" ;

structDef   = "def" "struct" IDENT "{" [ structField { "," structField } [ "," ]
] "}" ";" ;
                                       (* top-level only;
                                          IDENT names the struct type;
                                          zero fields parse. Hoisted
                                          before the first top-level
                                          statement runs. *)
enumDef     = "def" "enum" IDENT "{" enumVariant { "," enumVariant } [ "," ] "}"
";" ;
                                       (* top-level only; a sum type. IDENT
                                          names the enum; at least one variant.
                                          Hoisted like structDef. A value of the
                                          type is exactly one variant. *)
enumVariant = IDENT [ "{" structField { "," structField } [ "," ] "}" ] ;
                                       (* a payload-less tag, or a variant with
                                          named fields (a mini-struct). *)
structField = wordName "as" type ;     (* a field name may be any
                                          identifier-shaped word (see
                                          wordName below) - `from` / `to`
                                          are fine; camelCase (no `_`)
                                          is enforced as a check *)

statement   = defineStmt
            | assignStmt
            | indexAssign
            | fieldAssign
```

```
             | appendStmt
             | returnStmt
             | ifStmt
             | whileStmt
             | forStmt
             | forEachStmt
             | repeatStmt
             | matchStmt
             | breakStmt
             | continueStmt
             | exitStmt
             | tryStmt
             | throwStmt
             | deferStmt
             | exprStmt ;

tryStmt     = "try" block "catch" "(" IDENT ")" block ;
                                     (* IDENT is the catch
                                        binding, follows the
                                        iteration-variable name rule
                                        (letters only). No `finally`
                                        in v1. *)
throwStmt   = "throw" expr ";" ;
                                     (* expr may produce any
                                        value; convention is an
                                        `Error` struct. *)
deferStmt   = ( "defer" | "errdefer" ) deferCall ";" ;
deferCall   = "(" deferCall ")" | qualifiedCall | call ;
                                     (* exactly a call - a plain call
                                        (`cleanup()`) or a namespaced /
                                        module call (`fs.close($f)`);
                                        grouping parens are transparent
                                        (`defer (f());` is accepted),
                                        any other expr is a parse
                                        error. Args evaluate at the
                                        defer site; the call runs
                                        when the enclosing block
                                        exits, LIFO. `defer` runs on
                                        every exit path; `errdefer`
                                        only when the block exits
                                        with a propagating error (a
                                        throw / runtime error - not
                                        return, break, continue, or
                                        exit). Block-scoped. *)

returnStmt  = "return" [ expr ] ";" ;
breakStmt   = "break" ";" ;          (* exits the innermost loop *)
continueStmt = "continue" ";" ;      (* skips to the next iteration *)
exitStmt    = "exit" [ expr ] ";" ;  (* terminates the program; the
                                        optional int expr is the exit
                                        code (0 when omitted) *)

constDefine = "def" "const" IDENT "as" type "init" expr ";" ; (* the const
```

```
                                        form of defineStmt - the only `def`
                                        an `export` may mark *)
defineStmt  = "def" [ "const" ] IDENT "as" type [ "init" expr ] ";" ;
                                     (* constants require "init" and an
                                        uppercase name matching
                                        [A-Z]+(_[A-Z]+)* (uppercase
                                        chunks joined by single `_`;
                                        no leading, trailing or
                                        consecutive `_`); variables may
                                        omit "init" and get zero-value,
                                        and use the letters-only IDENT
                                        form *)

assignStmt  = VARREF "=" expr ";" ;

indexAssign = VARREF lvalueTail { lvalueTail } "[" expr "]" "=" expr ";" ;
                                     (* l-value chain ending in `[index]`;
                                        root is a VARREF. Tail
                                        steps may freely mix `[index]`
                                        and `.field`. *)

fieldAssign = VARREF lvalueTail { lvalueTail } "." wordName "=" expr ";" ;
                                     (* l-value chain ending in `.field`.
                                        Root is a VARREF; tail
                                        may mix `[index]` and `.field`. *)

lvalueTail  = "[" expr "]" | "." wordName ;

appendStmt  = VARREF "[" "]" "=" expr ";" ;
                                     (* append sugar: write-only
                                        target meaning "the position
                                        just past the end of the
                                        list"; read use `e[]` is a
                                        parse error. Only one bare
                                        VARREF root - chained forms
                                        like `$xs[0][]` are not supported
                                        (yet). *)

ifStmt      = "if" "(" expr ")" block
              { "elseif" "(" expr ")" block }
              [ "else" block ] ;

whileStmt   = "while" "(" expr ")" block ;

forStmt     = "for" "(" [ defineStmt | assignStmt | ";" ]
                        [ expr ] ";"
                        [ assignNoSemi ]
                  ")" block ;
assignNoSemi = VARREF "=" expr ;       (* same shape as assignStmt without
trailing ";" *)

forEachStmt = "for" "(" "def" IDENT "in" expr ")" block ;
                                     (* iterates list elements (in order)
```

```
                                        or map keys (insertion order);
                                        the iteration variable is a fresh
                                        binding in the body's scope *)

repeatStmt  = "repeat" block "until" "(" expr ")" ";" ;
                                     (* post-test loop: the body runs at
                                        least once; exits when the
                                        condition is true *)

matchStmt   = "match" "(" expr ")" "{"
                  { "when" ( valueArm | patternArm ) block }
                  [ "else" block ]
              "}" ;
valueArm    = expr { "," expr } ;      (* one or more values compared to the
                                        subject by `==` *)
patternArm  = IDENT [ "(" IDENT ")" ] ; (* an enum-variant pattern: a variant
                                        name, optionally binding the payload
                                        into a fresh name *)
                                     (* Two arm forms, disambiguated at resolve
                                        time by the subject's type. When the
                                        subject is a variable / parameter of a
                                        known enum type, every arm is a
                                        patternArm (variant + optional binder),
                                        checked for exhaustiveness (cover every
                                        variant or add `else`). Otherwise arms
                                        are valueArms: the subject is compared
                                        to each value by `==`, first match wins,
                                        `else` optional and last, no
                                        fall-through. A bare `Name` value
                                        followed by `{` reads the `{` as the arm
                                        block - parenthesize a struct-literal
                                        value. *)

exprStmt    = expr ";" ;

type        = primType | listType | mapType | taskType | channelType | funcType |
structType ;
primType    = "int" | "float" | "string" | "bool" | "null" | "bytes" ;
funcType    = "func" ;                (* a first-class function value; the
                                        type carries no signature - arity
                                        and argument kinds are checked at
                                        the call site *)
channelType = "channel" "of" type ;   (* a CSP channel. `channel` is a
                                        CONTEXTUAL keyword - a type only in
                                        this `channel of T` position; the `of`
                                        disambiguates it from a struct name, so
                                        `channel` stays an ordinary identifier
                                        everywhere else *)
listType    = "list" "of" type ;
mapType     = "map" "of" type "to" type ;
taskType    = "task" "of" type ;      (* `task of T` - handle to
                                        a `spawn`ed computation. Same
                                        shape as `list of T`; recurses
```

```
                                        the same way (`task of list of
                                        int` is legal). *)
                                    (* recursive; nesting like
                                        `list of list of int` and
                                        `map of string to list of int`
                                        falls out naturally *)
structType  = IDENT [ "." IDENT ] ;    (* User-defined struct OR enum type
                                        (bare IDENT), or a library-/module-
                                        provided namespaced type
                                        (`IDENT.IDENT`). The parser cannot tell
                                        a struct name from an enum name here;
                                        both are resolved at runtime against
                                        the struct / enum / NSStructs tables.
                                        Unknown names are positioned errors. *)

expr        = rangeExpr ;
rangeExpr   = orExpr [ ".." orExpr ] ;
                                    (* half-open range `lo..hi` -> `[lo, hi)`.
                                        Non-associative (`a..b..c` is an
                                        error) and looser than every binary
                                        operator, so `1+1..2*3` parses as
                                        `(1+1)..(2*3)`. Both bounds int; a
                                        range materialises a `list of int`,
                                        or - as a for-each source - iterates
                                        lazily. `lo > hi` is a runtime error;
                                        `lo == hi` is empty. *)
orExpr      = andExpr { "or" andExpr } ;
andExpr     = notExpr { "and" notExpr } ;
notExpr     = "not" notExpr | compExpr ;
compExpr    = bitOrExpr { ("<" | ">" | "<=" | ">=" | "==" | "!=") bitOrExpr } ;
bitOrExpr   = bitXorExpr { "|" bitXorExpr } ;
bitXorExpr  = bitAndExpr { "^" bitAndExpr } ;
bitAndExpr  = shiftExpr { "&" shiftExpr } ;
shiftExpr   = addExpr { ("<<" | ">>") addExpr } ;
                                    (* the bitwise family (`|` `^` `&`
                                        `<<` `>>`, unary `~`) operates on
                                        int only; precedence sits between
                                        comparison and additive, tightest
                                        first: shift, then `&`, `^`, `|` *)
addExpr     = mulExpr { ("+" | "-") mulExpr } ;
mulExpr     = unaryExpr { ("*" | "/" | "//" | "%") unaryExpr } ;
unaryExpr   = ("-" | "~") unaryExpr | primary ;
primary     = ( INT | FLOAT | STRING | interpString | "true" | "false" | "null"
              | VARREF | qualifiedCall | qualifiedConstRef | taskCall
              | call | structLit | constRef | "(" expr ")"
              | listLit | mapLit | lenExpr | spawnExpr )
              { "[" ( expr | sliceTail ) "]" | "." wordName | callArgs } ;

interpString = STRING_INTERP ;         (* a cooked "..." string carrying one or
                                        more `{expr}` interpolation slots. The
                                        lexer emits it as one TOKEN_STRING_
INTERP
                                        whose Parts alternate literal chunks and
```

```
                                        expression-source slots; the parser
                                        sub-parses each slot to a single `expr`
                                        and builds an InterpStringExpr (lowered
                                        to concat + convert.toString). A slot is
                                        one expr, not a statement; `\{`/`\}` are
                                        literal braces; a bare `}` is a lex
                                        error; a raw '...' string never
                                        interpolates. STRING is a cooked or raw
                                        literal with no slots. *)
                                    (* any primary can be index-,
                                        slice-, field-, or call-chained. A
                                        `[...]` holds either an index `expr`
                                        or a `sliceTail` (contains a `..`);
                                        a trailing `callArgs` is a call
                                        through a function value
                                        (CallValueExpr): `$f(x)`,
                                        `$fns[0](x)`, `makeAdder(1)(2)`. *)
callArgs    = "(" [ expr { "," expr } ] ")" ;
sliceTail   = orExpr ".." [ orExpr ]   (* `[a..]`, `[a..b]` *)
            | ".." [ orExpr ] ;        (* `[..]`, `[..b]` - endpoints parse at
                                        orExpr so a bool-keyed comparison
                                        index `$m[$a == $b]` stays an index,
                                        not a slice. Slice yields a fresh,
                                        value-semantic copy (never a view);
                                        open ends default to 0 / len; strict
                                        `0 <= lo <= hi <= len` bounds check;
                                        works on list / bytes / string
                                        (rune-indexed). Read-only:
                                        `$xs[a..b] = ...` is a parse error. *)
spawnExpr   = "spawn" block ;          (* launches the block as a
                                        goroutine and evaluates
                                        immediately to a `task of T`
                                        where T is the body's return
                                        type at the use site. Bare
                                        `return;` produces `task of
                                        null`. Value-semantics
                                        capture: every binding visible
                                        at the spawn site is
                                        deep-copied into a fresh frame
                                        at launch. *)
lenExpr     = "len" "(" expr ")" ;     (* polymorphic
                                        structural-length built-in
                                        (string / list / map /
                                        bytes). Reserved keyword,
                                        not a library function; the
                                        `core` library that once
                                        hosted it no longer exists. *)
structLit   = IDENT [ "." wordName [ "." wordName ] ] "{" [ structLitField { ","
structLitField } [ "," ] ] "}" ;
                                    (* struct OR enum literal (same node).
                                        Bare IDENT names a user struct;
                                        `IDENT.IDENT` is a namespaced struct
                                        (`os.Result{...}`) OR a local enum
```

```
                                        variant (`Shape.Circle{...}`);
                                        `IDENT.IDENT.IDENT` is a cross-module
                                        enum variant (`mod.Shape.Circle{...}`).
                                        The enum-vs-struct decision is made at
                                        eval from the tables, not from
                                        capitalisation. A payload-less enum
                                        variant (`Shape.Empty`) has no `{...}`
                                        and reaches eval as a constRef the
                                        interpreter resolves against the enum.
                                        The `{` after IDENT in expression
                                        position is the tie-breaker against
                                        `constRef`. `P{}` parses; "every field
                                        present exactly once" is a post-parse
                                        check. *)
structLitField = wordName ":" expr ;
call        = IDENT "(" [ expr { "," expr } ] ")" ;
qualifiedCall      = IDENT "." wordName "(" [ expr { "," expr } ] ")" ;
qualifiedConstRef  = IDENT "." wordName ;
                                     (* qualifiedCall / qualifiedConstRef:
                                        IDENT "." wordName, then `(` decides
                                        which. Resolved against the
                                        namespaced-builtin / constant
                                        registry, gated by `use lib;`
                                        (or alias-aware equivalent). *)
taskCall    = "task" "." wordName "(" [ expr { "," expr } ] ")" ;
                                     (* the `task` library's namespace. `task`
                                        is a type keyword, not an IDENT, so
                                        `qualifiedCall` (IDENT-led) cannot
                                        match it - hence its own production.
                                        Expression position only, as the
                                        namespace prefix (`task.wait($t)`). *)
constRef    = IDENT ;                (* bare-IDENT: constant reference; the
                                        parser disambiguates `call` vs
                                        `qualifiedCall` vs `constRef` by
                                        peeking for "." / "(". *)
wordName    = IDENT | KEYWORD ;      (* any identifier-shaped token: a plain
                                        IDENT or a keyword spelled like one
                                        (every Jennifer keyword is). Name
                                        positions - after a ".", and field
                                        names in struct definitions and
                                        literals - are contextually
                                        unambiguous, so reserved words are
                                        valid names there:
                                        `strings.repeat(...)`,
                                        `def struct Route { from as Point,
                                        to as Point };`, `$r.to`. *)
listLit     = "[" [ expr { "," expr } [ "," ] ] "]" ;
mapLit      = "{" [ expr ":" expr { "," expr ":" expr } [ "," ] ] "}" ;
                                     (* `{` is also a block opener; only
                                        legal as a map literal in
                                        expression position, where the
                                        parser is unambiguous *)
```

A type keyword (`int` , `float` , `string` , `bool` , `bytes`) has **no expression-position meaning** : the parser reports a positioned error pointing at the `convert` library (`convert.toInt(v)` , ...; `convert.bytesFromString(s, codec)` for bytes). `task` is the one type keyword with an expression role - solely as the `task.` namespace prefix (`task.wait($t)`).

Grammar - PEG

The operational view of the grammar: a Parsing Expression Grammar whose **ordered choice** and **predicates** encode exactly the decisions `internal/parser/parser.go` makes (a recursive-descent parser with fixed lookahead *is* a PEG evaluator, minus the backtracking it never needs). Where the EBNF says *what shapes exist* , this grammar says *which alternative wins and why* . The two describe the same language and must be kept in sync; the parser is the source of truth for both.

Like the EBNF, this describes the token stream **after** preprocessing (`include` splices are already expanded). Terminals in CAPITALS are token classes from the lexer (Lexer > Token types); quoted strings are keywords or punctuation matching one token. Because the terminals are whole tokens, the classic PEG longest-match ordering concern (`"<="` before `"<"`) does not arise here - the lexer already decided; ordering below matters only where two alternatives share a *token* prefix.

Notation: `<-` defines, `/` is ordered choice (first match wins, no reconsidering), `? * +` repeat greedily, `&e` succeeds iff e matches (consuming nothing), `!e` succeeds iff e does not match (consuming nothing).

```
# --- program structure ------------------------------------------------------

Program      <- (UseStmt / ModuleImport / Exported / MethodDef / StructDef
                / EnumDef / Statement)* EOF
Exported     <- "export" (MethodDef / StructDef / EnumDef / ConstDefine)
UseStmt      <- "use" IDENT ("as" IDENT)? ";"
ModuleImport <- "import" STRING ("as" IDENT)? ";"
MethodDef    <- "func" IDENT "(" ParamList? ")" Block
ParamList    <- Param ("," Param)*
Param        <- IDENT "as" Type
Block        <- "{" Statement* "}"

StructDef    <- "def" "struct" IDENT "{" (StructField ("," StructField)* ","?)?
"}" ";"
StructField  <- WordName "as" Type
                # zero fields parse (`def struct E { };`); a field name may be
                # any identifier-shaped word (WordName, below) - `from` / `to`
                # are fine - with camelCase (no `_`) enforced as a check

EnumDef      <- "def" "enum" IDENT "{" EnumVariant ("," EnumVariant)* ","? "}" ";"
EnumVariant  <- IDENT ("{" StructField ("," StructField)* ","? "}")?
                # a sum type; top-level only, hoisted like StructDef. At least
                # one variant. Each variant is a payload-less tag or carries a
                # mini-struct field list.

# --- statements -------------------------------------------------------------
# The parser dispatches on the leading token; the keyword alternatives are
# mutually exclusive, so their relative order is free. Within the VARREF-
# rooted group and against ExprStmt the order IS load-bearing: ExprStmt must
# come last (`$x = 1;` would otherwise never be an assignment), and
# ForEachStmt must precede ForStmt (both start `for` "(" - the parser peeks
# for `def IDENT in`, which ordered choice reproduces).

Statement    <- DefineStmt
              / IfStmt / WhileStmt / ForEachStmt / ForStmt / RepeatStmt /
MatchStmt
              / BreakStmt / ContinueStmt / ReturnStmt / ExitStmt
              / TryStmt / ThrowStmt / DeferStmt
              / AssignStmt / AppendStmt / IndexAssign / FieldAssign
              / ExprStmt

ConstDefine  <- "def" "const" IDENT "as" Type "init" Expr ";"
DefineStmt   <- "def" "const"? IDENT "as" Type ("init" Expr)? ";"
                # constants require "init" and the UPPER_CASE name shape;
                # both are checked by the parser, not the grammar

IfStmt       <- "if" "(" Expr ")" Block
                ("elseif" "(" Expr ")" Block)*
                ("else" Block)?
WhileStmt    <- "while" "(" Expr ")" Block
ForEachStmt  <- "for" "(" "def" IDENT "in" Expr ")" Block
ForStmt      <- "for" "(" ForInit Expr? ";" ForStep? ")" Block
ForInit      <- DefineStmt / AssignStmt / ";"
```

```
                # define / assign consume their own ";"; a lone ";" is the
                # empty init
ForStep      <- VARREF "=" Expr        # no trailing ";" - the ")" ends it
RepeatStmt   <- "repeat" Block "until" "(" Expr ")" ";"
MatchStmt    <- "match" "(" Expr ")" "{"
                 ("when" (Expr ("," Expr)*) Block)*
                 ("else" Block)?
                "}"
                # the subject and each `when` value parse with struct-literal
                # suppression on (a bare `Name` followed by `{` is a value +
                # the block, not `Name{...}`); the flag is re-enabled inside
                # any "(" / "[" / call / list / map, mirroring Go's exprLev.
                # `else` must be last (enforced by the parser, not the grammar).
                # When the subject has a known enum type, the resolver
                # reinterprets each arm's single `Expr` as a variant pattern -
                # `Name` (payload-less) or `Name(bind)` (payload binder) - and
                # checks exhaustiveness. This is a resolve-time rewrite, not a
                # separate grammar production: a pattern arm parses as an
                # ordinary value expression (a ConstRef or a Call).
BreakStmt    <- "break" ";"
ContinueStmt <- "continue" ";"
ReturnStmt   <- "return" Expr? ";"
ExitStmt     <- "exit" Expr? ";"
TryStmt      <- "try" Block "catch" "(" IDENT ")" Block
ThrowStmt    <- "throw" Expr ";"
DeferStmt    <- ("defer" / "errdefer") DeferCall ";"
DeferCall    <- "(" DeferCall ")" / QualifiedCall / PlainCall
                # exactly a call - the parser parses a full expression and
                # rejects any node that is not a call. Grouping parens
                # evaluate to their inner node, so `defer (f());` is accepted
                # (hence the recursive first alternative); a postfix chain
                # (`f()[0]`, `f().g`) is not a call node and fails.

# The four VARREF-rooted statements. AssignStmt first mirrors the parser's
# `$x =` peek; AppendStmt's "[" "]" prefix cannot be an IndexAssign (an index
# expression never starts at "]"), but trying it first keeps the intent
# obvious. In `(LvalueStep !"=")*` the not-predicate stops the greedy star
# one step early, leaving the FINAL `[index]` / `.field` - the one directly
# before "=" - for the explicit tail. Without the predicate a greedy star
# would swallow the last step and the alternative could never match.

AssignStmt   <- VARREF "=" Expr ";"
AppendStmt   <- VARREF "[" "]" "=" Expr ";"
IndexAssign  <- VARREF (LvalueStep !"=")* "[" Expr "]" "=" Expr ";"
FieldAssign  <- VARREF (LvalueStep !"=")* "." WordName "=" Expr ";"
LvalueStep   <- "[" Expr "]" / "." WordName

ExprStmt     <- Expr ";"

# --- types ------------------------------------------------------------------
# The compound heads (`list` / `map` / `task`) and the primitive keywords are
# distinct tokens, so this choice never backtracks; StructType last catches
# the bare / namespaced IDENT forms.
```

```
Type         <- ListType / MapType / TaskType / ChannelType / FuncType / PrimType
/ StructType
ListType     <- "list" "of" Type
MapType      <- "map" "of" Type "to" Type
TaskType     <- "task" "of" Type
ChannelType  <- "channel" "of" Type   # contextual keyword: `channel` is a type
                                      # only here (in `channel of T`); elsewhere
it
                                      # is an ordinary identifier
FuncType     <- "func"                # first-class function value (no signature)
PrimType     <- "int" / "float" / "string" / "bool" / "null" / "bytes"
StructType   <- IDENT ("." IDENT)?

# --- expressions ------------------------------------------------------------
# One rule per precedence level, loosest first (the same ladder the parser
# climbs): or, and, not, comparison, |, ^, &, shifts, additive,
# multiplicative, unary. `A (op A)*` yields left association; NotExpr and
# Unary recurse right.

Expr         <- RangeExpr
RangeExpr    <- OrExpr (".." OrExpr)?   # half-open `lo..hi` -> `[lo, hi)`;
                                        # non-associative (a second `..` is an
                                        # error), looser than every binary op.
                                        # Materialises `list of int`, or - as a
                                        # for-each source - iterates lazily.
OrExpr       <- AndExpr ("or" AndExpr)*
AndExpr      <- NotExpr ("and" NotExpr)*
NotExpr      <- "not" NotExpr / CompExpr
CompExpr     <- BitOrExpr (("<" / ">" / "<=" / ">=" / "==" / "!=") BitOrExpr)*
BitOrExpr    <- BitXorExpr ("|" BitXorExpr)*
BitXorExpr   <- BitAndExpr ("^" BitAndExpr)*
BitAndExpr   <- ShiftExpr ("&" ShiftExpr)*
ShiftExpr    <- AddExpr (("<<" / ">>") AddExpr)*
AddExpr      <- MulExpr (("+" / "-") MulExpr)*
MulExpr      <- Unary (("*" / "/" / "//" / "%") Unary)*
Unary        <- ("-" / "~") Unary / Primary

# Any primary can be postfix-chained with indexing, field access, and a call
# through a function value. The chain belongs to Primary, not Atom, so
# `f()[0].x` and `$fns[0](x)` parse as expected.

Primary      <- Atom Postfix*
Postfix      <- "[" (SliceTail / Expr) "]" / "." WordName / CallArgs
CallArgs     <- "(" (Expr ("," Expr)*)? ")"
                                        # a call through a function value
                                        # (CallValueExpr): `$f(x)`, `$fns[0](x)`,
                                        # `makeAdder(1)(2)`
SliceTail    <- OrExpr ".." OrExpr? / ".." OrExpr?
                                        # `$xs[a..b]`, `$xs[a..]`, `$xs[..b]`,
                                        # `$xs[..]`. Endpoints parse at OrExpr so
                                        # a bool-keyed comparison index
                                        # `$m[$a == $b]` stays an index. A slice
```

```
                                        # is a fresh value-semantic copy (never a
                                        # view); read-only (`$xs[a..b] = ...` is a
                                        # parse error).

# Atom is where the parser's peek-based disambiguation lives; here it is
# ordered choice over alternatives that share prefixes. IdentExpr and
# TaskExpr expand below.

Atom         <- INT / FLOAT / STRING / "true" / "false" / "null"
              / LenExpr / SpawnExpr / TaskExpr
              / VARREF / IdentExpr
              / "(" Expr ")" / ListLit / MapLit
                 # a type keyword (int / float / string / bool / bytes) is NOT
                 # an atom: expression position rejects it with a positioned
                 # error pointing at convert.toInt / convert.toFloat / ...

LenExpr      <- "len" "(" Expr ")"
SpawnExpr    <- "spawn" Block

# After a bare IDENT the next token decides, in this order: "." starts a
# qualified form, "{" a struct literal, "(" a plain call, anything else a
# constant reference. The order is load-bearing: QualifiedSuffix must be
# tried before StructBody / CallArgs (a "." wins over everything), and the
# bare-IDENT constant reference must come last or it would always win.
# After the "." the same trick repeats: "(" makes it a call, "{" a
# namespaced struct literal, else it is a qualified constant.

IdentExpr    <- IDENT IdentSuffix?
IdentSuffix  <- QualifiedSuffix / StructBody / CallArgs
QualifiedSuffix <- "." WordName (CallArgs / StructBody)?

# WordName: any identifier-shaped token - a plain IDENT or a keyword spelled
# like one (every Jennifer keyword is). Name positions (after a ".", field
# names in struct definitions and literals) are contextually unambiguous, so
# reserved words are valid names there: `strings.repeat(...)`,
# `def struct Route { from as Point, to as Point };`, `$r.to`.

WordName     <- IDENT / KEYWORD

# `task` is a type keyword, not an IDENT, so the IDENT-led call rules
# (PlainCall / QualifiedCall) never match it; TaskExpr below is the one
# production that consumes `task` in expression position - and only as the
# `task` library's namespace prefix.

TaskExpr     <- "task" QualifiedSuffix

# Named the same as their EBNF counterparts:

PlainCall     <- IDENT CallArgs
QualifiedCall <- IDENT "." WordName CallArgs
CallArgs      <- "(" (Expr ("," Expr)*)? ")"
StructBody    <- "{" (StructLitField ("," StructLitField)* ","?)? "}"
StructLitField <- WordName ":" Expr
```

```
                # `P{}` parses; "every field present exactly once" is a
                # post-parse check (duplicates at parse time, missing fields
                # at evaluation)

ListLit      <- "[" (Expr ("," Expr)* ","?)? "]"
MapLit       <- "{" (MapEntry ("," MapEntry)* ","?)? "}"
MapEntry     <- Expr ":" Expr
                # "{" is a block opener in statement position and a map
                # literal in expression position; the split between
                # Statement and Expr contexts keeps the two from meeting
```

Reading the load-bearing orderings

Three places where PEG's first-match-wins is doing real work, and what the parser does at the same juncture:

- **Statement: ExprStmt last.** The parser peeks: `$x` followed by = is an assignment, `$xs[...]...` ending in = `expr ;` an index/field assignment, and only a `$x` chain *not* followed by = falls through to an expression statement. Ordered choice reproduces that priority; putting `ExprStmt` earlier would shadow every assignment form.
- **IdentSuffix / QualifiedSuffix: call before constant.** The parser decides `call` vs `constRef` (and `qualifiedCall` vs `qualifiedConstRef`) by peeking for `(` . In PEG the bare-name alternative simply comes last: if `CallArgs` matches, it was a call; only when nothing longer matches does the name stand alone.
- **IndexAssign / FieldAssign: (LvalueStep !"=")*.** PEG repetition is greedy and never gives back, so `LvalueStep* "[" Expr "]"` could not work - the star would consume the final step. The `!"="` predicate stops the star exactly one step before the = , which is precisely the parser's approach of scanning the l-value chain while watching for the assignment that terminates it.

Preprocessor (internal/preproc)

Sits between the lexer and the parser. Its only job is to expand `include` file splices and pass `use` (library) and `import` (module) statements through unchanged.

Algorithm

1. Strip trivia tokens (comments, blank lines) so the recognizers can rely on adjacent tokens being meaningful.
2. Walk the token stream.
3. When `INCLUDE STRING SEMI` is found (a textual file splice):
   - Verify the string ends in `.j` .
   - Resolve the path (relative to the current file's directory, or absolute if it starts with `/`).
   - Reject if the path was already visited up the include chain (circular include).
   - Read the file, lex it (with file-tagged tokens), recursively preprocess it.
   - Splice the result (dropping the trailing `EOF`) at this point.
4. When `IMPORT ...` is found (a module import): pass the tokens through unchanged. `import` is a real statement handled by the parser (`ModuleImportStmt`) and interpreter (the module loader), not a textual splice. The one thing checked here is the common unquoted mistake (`validateModuleImport`).
5. When `USE IDENT SEMI` is found: pass through unchanged. The parser turns it into an `ImportStmt` node.
6. Helpful errors for common mistakes:
   - `include IDENT;` (library form with `include`) -> "use `use NAME;` for system libraries".
   - `include IDENT.j;` (unquoted file form) -> "file splices take a string literal".
   - `include "foo.go";` (wrong extension) -> "include path must end with `.j` ".
   - `import IDENT;` (library form with `import`) -> " `import` takes a quoted module path; for a system library use `use NAME;` ".
   - `import IDENT.j;` (unquoted module path) -> "module paths are quoted: `import \"name.j\";` ".
   - `use IDENT.j;` (file form with `use`) -> "for files use `include \"name.j\";` ".

Edge cases

- An `include` path must literally end in `.j` . `include "foo.go";` is rejected.
- `include` paths may contain `/` for subdirectories. Absolute paths are accepted as-is; relative paths resolve from the including file's directory.
- Circular includes are detected by tracking absolute paths visited along the current chain. (A module `import` cycle is detected separately, by the loader in `internal/interpreter` ; the preprocessor never opens an imported module - it only forwards its tokens.)

Interpreter (internal/interpreter)

A tree-walking evaluator.

Runtime values

`Value` is a tagged union (single concrete struct) rather than a Go interface hierarchy. This avoids `reflect` and method-table indirection, which keeps the binary small and predictable under TinyGo.

```
type Value struct {
    Kind    ValueKind  # KindNull | KindInt | KindFloat | KindString |
                      #  KindBool | KindList | KindMap | KindBytes |
                      #  KindStruct (M13.1)
    Int     int64
    Float   float64
    Str     string
    Bool    bool
    List    []Value      # KindList:   element data
    Map     []MapEntry   # KindMap:    insertion-ordered entries
    Bytes   []byte       # KindBytes:  raw bytes
    Fields  []StructField # KindStruct: ordered (Name, Value) per definition
    StructName string    # KindStruct: matches the StructDef name
    ElemTyp *parser.Type # KindList: declared element type (stamped)
    KeyTyp  *parser.Type # KindMap:  declared key type   (stamped)
    ValTyp  *parser.Type # KindMap:  declared value type (stamped)
}
```

`ZeroFor(t parser.Type)` returns the zero value for each declared type and is used when a `def` omits its `init` clause: `0` , `0.0` , `""` , `false` , `null` , an empty `[]` list (typed by the declaration), or an empty `{}` map. `Value.AsFloat()` handles int->float promotion for arithmetic and comparison; `Value.Equal()` implements == (same-kind comparison, plus the numeric-promotion rule across `int` and `float` ; deep-equal for lists; order-insensitive key->value equality for maps).

Parameterized Type

`parser.Type` is a recursive struct: `{ Kind TypeKind; Element, KeyType, ValType *Type }` . Compound types nest naturally (`list of list of int`). Equality (`Type.Equal`) is structural.

Value semantics

Lists and maps are **value-typed** in Jennifer: `$ys = $xs;` behaves as a copy, function parameters bind by copy, and `const` is deep. No aliasing is observable from user code.

Eager-copy value semantics. Value semantics rest entirely on eager deep copies at every store site, not on copy-on-write:

- Every binding site takes a private copy before storing: `execDefine` / `execAssign` via `eagerCopy` , parameter binding via `bindParamValue` , and `snapshotForSpawn` via `DeepCopy` for the goroutine-boundary crossing. Library builtins whose pattern is "copy then mutate freely" (`lists.shuffle` , `lists.reverse` , ...) `Copy()` first. So no two live bindings ever share a compound backing.
- Because of that invariant the mutation sites (`execAppend` , `execIndexAssign` , `execFieldAssign`) fetch the target via `GetBinding` and mutate the binding's **own** backing in place - no per-write copy - which keeps append-in-a-loop amortised O(N).
- `Value.Copy()` is the public deep-copy alias; the engine is `DeepCopy()` .
- One optimisation trims a redundant copy: a fresh list / map / struct literal RHS is already private (its evaluator `Copy()` s every element into a brand-new container), so `execDefine` / `execAssign` skip the whole-value eager copy for it (`rhsFreshLiteral`) and only stamp the declared type. Var / index / field reads, const refs, and calls can hand back a reference into a live binding, so those are still eager-copied.

Read-only-parameter borrow. The one carve-out to "parameter binding always copies": when a method never writes a parameter, its argument is bound by **alias** rather than deep-copied, since a read-only alias is observationally identical to a copy. `Resolve` runs `markBorrowableParams` per method - a statement-level write-scan of the body (descending through nested control flow; because no shadowing is allowed, a written name is unambiguously the parameter, and `spawn` bodies mutate their own deep-copied snapshot so are correctly skipped) - and sets `Param.Borrow` when the parameter is never written **and** its type is *borrow-safe* : a compound whose `stampDeclaredType` does not recurse into element backing (a flat `list` / `map` , or a struct / `bytes` ; a `list of list` falls back to copying so the stamp cannot mutate the shared backing). `Interpreter.bindArg` aliases when `Param.Borrow && (i.isModule || i.entryGlobalsImmutable || m.GlobalSafe)` and copies (`bindParamValue`) otherwise, per callee method `m` (`methodBorrowCtx`). The gate is the soundness boundary: "never writes the parameter" alone is not enough, because if the argument aliases a mutable global and the body mutates that global, a borrowed read would see the change where a copy would not. So borrow is enabled only where no mutable global can be aliased and mutated during the call, which holds in three cases:

- a **module** (top level is declarations-only, and cross-module arguments are copies at the boundary);
- a **single-file script that declares no mutable top-level def** (`hasMutableTopLevelGlobal` false, recorded as `entryGlobalsImmutable` in `Run`), so a script's own read-only helpers benefit without being moved into a module;
- a specific **method proven to mutate no global transitively** (`m.GlobalSafe`), even in a script that holds mutable globals elsewhere. `computeEntryGlobalSafe` (run after module load) scans each entry method for a local hazard - a write to a non-local name, a `CallValueExpr` , a module call (which can re-enter the host via `meta.callMain`), or a callback builtin that can invoke `.j` code (the higher-order `lists` layer, `meta.call` / `meta.callMain` , `testing.run` / `runWith` / `assertThrows` , keyed by canonical namespace) - then a fixpoint propagates the hazard along named-call edges. The walker defaults an unrecognised node to hazardous, so a new node never silently escapes; `spawn` bodies are skipped (they mutate a snapshot). The callback set must be a denylist because a func-value argument is not statically visible (a bare method name is a `ConstRefExpr` , and curried func returns carry no type) - the invariant "any builtin invoking `.j` code via `BuiltinCtx.Invoke` / `CallByName*` / `CallHostWith*` is listed" lives in `globalsafe.go` .

The REPL (whose global set grows across inputs) keeps the copying bind. This is the extension of the scalar copy-elision to compound Kinds that `bindParamValue` was documented as needing a mutation-safety proof for. It removes accidental quadratics in read-only helpers (a `markdown` block collector that binds the whole line list once per block; the `countNodes` budget walk; a per-element script helper over a growing list). Pinned by `internal/parser/borrow_test.go` (write-scan + type gate), `internal/interpreter/borrow_test.go` (soundness gates + value- semantics parity for module, globals-free script, and mutable-globals script), and `globalsafe_internal_test.go` / `borrow_internal_test.go` (the `GlobalSafe` fixpoint and the `entryGlobalsImmutable` wiring).

A shared-marker copy-on-write protocol (`Value.shared *bool` + `Share()` / `Ensure()`) was tried here and removed. It was inert: a value receiver plus by-value `Environment.Get` / `GetAt` reads meant the flag was set on a throwaway copy and never reached the stored binding, so `Ensure` never detached and correctness always came from the eager copies above. Reintroducing it write-through (store `*Binding` so the marker propagates) was considered and rejected - see rejected.md.

Borrow and that COW are near-opposites, which is why one is rejected and the other ships. COW is dynamic and optimistic: share every value by default, then *detect a mutation at runtime and deep-copy at that moment* - so it needs a live marker on every read/write and it keeps a **mutable** shared backing around until a write splits it, exactly the shared mutable state the eager-copy model is designed to avoid. Borrow is static and conservative: share only the bindings a resolve-time proof shows are **never** written, so there is no detach, no runtime machinery, and the shared backing is **immutable** for the alias's lifetime (as safe to share as a `const`). COW was also rejected as barely-reachable - because every store copies, two live bindings almost never alias, so its deferred-copy case rarely occurs; borrow targets the opposite, ubiquitous case, since parameter binding is a store site that *always* copies and read-only helpers are everywhere. Borrow is a second narrow, provably-safe

copy elision layered on top of eager copies (like `rhsFreshLiteral`), not a replacement of the model the way COW would have been.

Aliasing correctness is exercised by `internal/interpreter/value_alias_test.go` (https://github.com/jennifer-language/jennifer/blob/main/internal/interpreter/value_alias_test.go) - every "shared then mutated" corner case (nested lists, structs containing lists, function-argument mutation, chained lvalues, etc.). Anyone changing the mutation logic must add coverage there.

Type stamping

After a literal like `[1, 2, 3]` evaluates, the resulting `Value` has no `ElemTyp`. The declared type lives only on the receiving binding's `DeclType`. To make subsequent operations - index-writes, parameter checks, iteration types - have inner-type info without re-consulting the declaration, the interpreter calls `stampDeclaredType(v, declType)` at every binding boundary. The helper writes `ElemTyp` / `KeyTyp` / `ValTyp` onto the value and recurses into nested compound elements so deep type tracking is preserved for nested `$xs[i][j] = ...` writes.

Index access

- **Reads** (`evalIndex`): out-of-bounds list indices and missing map keys are positioned runtime errors. Reads return the slot value by copy via Go's struct semantics, but the inner slice headers still alias - which is fine because reads can't mutate anything.
- **Writes** (`execIndexAssign` / `execFieldAssign`): both route through the unified `collectLvalueSteps` + `applyLvalueWrite` walker. The walker descends through the chain (any mix of `[index]` and `.field` nodes) on a fresh copy of the root binding, writes via `writeIndexedSlot` (index leaf) or the per-struct-field type check (field leaf), then commits back through `env.Assign`. Const enforcement fires once against the root binding (deep constness). Map writes to a missing key extend the map (insertion order preserved); writes to an existing key update in place. Struct field writes are type-checked against the declared field type stored in the `StructDef`.

Structs

User-defined struct types live in `Interpreter.structs` (`map[string]*parser.StructDef`), populated at `Run` time by hoisting every top-level `def struct` before any other statement executes. This mirrors the method-hoisting pass, and the same duplicate-name check applies (`Run` rejects two structs with the same name; the REPL silently redefines).

Per-instance values use `KindStruct` and carry the struct name plus a `[]StructField` (each entry is `{Name, Value}`). The field slice is stored in declaration order so `%v` rendering and `Equal` checks are deterministic. `Copy()` deep-copies every field so value semantics hold; `MatchesDeclared` matches by name (`p` typed `Point` matches `Point`-typed declarations only).

The `def x as Name;` no-init path is handled by `zeroStructFor`, which recurses through nested struct-typed fields so every leaf field gets its declared zero. Unknown struct names are rejected here and at `execDefine` time before the init expression is evaluated, so the user sees `"unknown struct type"` rather than a misleading type-mismatch error.

Module structs resolve by their canonical path, not their stem: `zeroStructFor` and `lookupStructDef` take the `ModPath` carried on the value / declared type and resolve through `moduleByPath`, falling back to the stem-keyed `moduleByNS` only when no path is available. This is what keeps field writes and zero-init working when two loaded modules share a file stem (`a/util.j` and `b/util.j`) - a stem-keyed lookup is ambiguous there and `moduleByNS` deliberately returns nil rather than guessing.

Library-provided namespaced structs

Libraries register their own struct types via `Interpreter.RegisterNamespacedStruct(libName, structName, fields)`. The definition lands in `i.NSStructs` keyed by `nsKey{NS, Name}`, parallel to `NSBuiltins` and `NSConstants`. Users write `def x as os.Result;` for the type and `os.Result{ ... }` for the literal; both forms resolve at use time via `resolveNamespacePrefix` so aliases (`use os as o; def x as

`o.Result;`) work the same way as they do for namespaced function calls and constants.

At runtime the value carries both `StructName` and an optional `StructNS` tag. `MatchesDeclared` and `Equal` compare on the `(NS, Name)` pair so a library `os.Result` is a distinct type from a user-defined `Result` ; `Display` prefixes the namespace (`os.Result{exitCode: 0, ...}`). Field access, chained lvalues (`$r.exitCode` , `$line.from.x = 5;`), value semantics, and deep `const` all reuse the user-struct machinery - only the type-resolution path differs.

User code may not register namespaced structs; the API is Go-side only. Programs that want to declare their own structs keep using the `def struct Name { ... };` bare form.

Iteration

`execForEach` opens a fresh per-iteration scope so the loop variable binding doesn't leak out and `def` -rebindings don't accumulate. It borrows that frame from `envPool` (like `execBlock`) rather than allocating a fresh map + slot slice each pass; `execFor` pools its header frame the same way. For lists it walks elements in order; for maps it walks keys in insertion order. The underlying map representation is an ordered slice (`[]MapEntry`) rather than a Go `map[K]V` precisely to make this iteration deterministic and testable.

Map hash index. Because `Map` is an ordered slice, a naive `indexInto` / `writeIndexedSlot` linear-scans it with `Value.Equal` per entry, so building a map with `$m[$k] = $v` over N keys is O(N^2). Each map `Value` carries an unexported side index `mapIdx` `map[string]int` (encoded scalar key -> position) that turns lookup, update, and the existence check on insert into O(1), so the build is O(N). The index is *advisory* : it is consulted only when `mapIndexUsable()` confirms it is complete - non-nil and `len(mapIdx) == len(Map)` - and every operation falls back to the linear scan otherwise, which is always correct. That length stamp is what makes the index safe under value semantics: a map copied by value shares the slice header until it diverges, a duplicate-key literal has no 1:1 index, and a non-hashable key (float, or any compound) is never indexed - all three fail the stamp and scan. Only hashable scalar keys (string / int / bool / null) are encoded, mirroring `Value.Equal` exactly so distinct keys never collide; float is excluded (NaN, -0.0, precision). The index is (re)built by `DeepCopy` (every binding-boundary copy owns a fresh one), by `evalMapLit` , and lazily on the first indexed write.

Environment

`Environment` is a parent-linked scope frame. Each frame carries two storage backends, but a given binding lives in exactly one of them:

- **slots []Binding** - the slot-indexed view and the primary storage for **resolved** code. Sized from `Block.NumSlots` at frame construction (`NewEnvironmentSized`) or grown on demand from `DefineAt` . A resolved binding (`Slot >= 0`) is written here and **only** here - not the name map - so a call / block frame does no per-binding heap allocation (the M21.12 per-frame allocation work). Each `Binding` carries its own `Name` , so the rare name-based readers can find a slot binding by scanning this (small, pooled) slice.
- **vars map[string]Binding** - the name-keyed view, now the **fallback only** : it holds bindings the resolver did not stamp with a slot (`Slot < 0`) - the REPL (each turn is a fresh parse with no resolver context) and ad-hoc / hand-built ASTs. Resolved frames leave it empty.
- **root *Environment** - cached pointer to the outermost ancestor. Set at construction via `rootFor(parent, self)` ; both `NewEnvironment*` and `borrowBlockEnv` populate it. `effectiveGlobal(env)` reads this field in O(1) with a defensive parent-chain walk as fallback. `releaseBlockEnv` clears it on release so pooled envs don't retain refs across borrow cycles.

`Binding{Value, DeclType, IsConst, Slot, Name}` carries `Slot` (`-1` for name-only `Define` ; non-negative for resolver-stamped `DefineAt`) and `Name` (the identifier, so a slot-backed binding is self-describing without a parallel name-map entry).

Storing the declared type lets `Assign` reject type-mismatching writes (you cannot assign a string to a variable declared as int).

`lookupLocal(name)` is the shared per-frame lookup: it scans `slots` by `Name` first (the resolved common

case), then falls back to `vars`. All the name-based readers below walk the parent chain calling it.

Name-based API (fallback path - REPL, spawn snapshot, ad-hoc AST):

- **Define(name, val, declType, isConst)** - adds to the current frame's `vars`; errors if the name exists *anywhere in the chain* (the spec forbids shadowing). Sets `Binding.Slot = -1`.
- **Assign(name, val)** - walks up the chain (`lookupLocal`) to find the binding; errors if it is a constant, the value's kind doesn't match its declared type, or the name is undefined. Writes back to the slot it came from, or to `vars` for a name-only binding.
- **Get(name)** and **GetBinding(name)** - walk up the chain via `lookupLocal`.
- **copyBindingsInto(dst)** - reconstructs a frame's name->value view (slots by `Name`, then `vars`) as detached deep copies; the one place `snapshotForSpawn` rebuilds captured scope from slot storage (globals are slot-backed too).

Slot-based API (fast path):

- **DefineAt(slot, name, val, declType, isConst)** - for `slot >= 0` installs the binding at `slots[slot]` (growing the slice if needed) with `Binding.Name = name` and does **not** touch `vars`; for `slot < 0` it takes the name-map fallback (with the enclosing-scope shadow check).
- **GetAt(depth, slot, name)** - walks `depth` parents then indexes `slots[slot]`. Falls back to `vars[name]` at the same depth when the slot is out of range (covers execution paths added to a resolver-less frame).
- **GetBindingAt(depth, slot, name)** - metadata companion to `GetAt`.
- **AssignAt(depth, slot, name, val)** - const / type-mismatch checks match the name path; on success writes to `slots[slot]` (the slot is authoritative for a slot-backed binding).

`NewEnvironmentSized(parent, numSlots)` is the constructor that pre-sizes `slots` from the resolver's hint, avoiding a grow on every `DefineAt` in hot loops. `NewEnvironment(parent)` (no size) is still used by REPL paths and by ad-hoc paths where the slot count isn't known upfront.

`execBlock` opens a fresh child `Environment` for each `{...}` block, so variables declared inside don't leak out. `for` opens its own scope wrapping init/cond/step/body so the init variable is visible throughout the loop without escaping it.

Resolver / runtime scope alignment

The resolver's static scope stack has to match the runtime's env chain exactly, or `(Depth, Slot)` addresses land in the wrong place. Three carve-outs where the resolver deviates from "one AST scope = one runtime frame":

- **try body runs in the enclosing env**, not a fresh frame: `execTry` calls `execStmts(body.Stmts, env)` directly. The resolver walks try-body statements inline in the current scope to match. Only the catch handler gets a proper scope push (matches the runtime's `catchEnv := NewEnvironment(env)`).
- **For-header init lives in forEnv**, body lives in a nested block frame: `execFor` creates one env for the header and `execBlock` nests another for the body. The resolver tracks them as separate scopes.
- **Spawn bodies are deliberately unresolved.** The runtime's `snapshotForSpawn` produces a two-frame duplex (globals-snap + locals-snap) that doesn't align with the resolver's single-frame view of "the enclosing scope." Rather than invent depth arithmetic to reconcile the two, the resolver skips spawn-body statements entirely and every reference inside falls back to name-based lookup at runtime. Spawn is coarse-grained concurrency dispatch, not hot-loop territory, so the perf regression is bounded.

Method call frames

Three compounded moves cut the recursive-call cost:

- **Environment pool.** `environment.go` exports `borrowBlockEnv` / `releaseBlockEnv` on top of a package-level `sync.Pool`. Every `execBlock`, every `evalCall`, and every `CallByName` borrows a frame on entry and returns it before returning to the caller. Release zeroes both the `vars` map (delete-in-place) and every used slot entry so the pool doesn't retain compound-value backings live between uses. Jennifer has no closures - no value, no library, no AST node can capture an env pointer past its

block's dynamic extent - so the pool is safe by construction. The two envs that outlive their creating call (`i.global` , the goroutine-root snapshots from `snapshotForSpawn`) stay on the non-pooled `NewEnvironment` path. `snapshotForSpawn` copies the *launching* goroutine's own root frame - `effectiveGlobal(env)` , which is `i.global` in serial code but the enclosing spawn's detached global snapshot inside a spawn body - never the live `i.global` , so a nested spawn snapshotting on a background goroutine can't race the main goroutine's global writes.

- **Pre-resolved callees.** `CallExpr` carries a `Method *MethodDef` pointer (see `internal/parser/ast.go`). During `Resolve` the resolver stamps the pointer when the callee names a hoisted top-level user method. `evalCall` consults the pointer first; only when it's `nil` (REPL turns, hand-built ASTs) does it fall back to `i.methods[c.Callee]` . Builtins keep the pointer `nil` because the namespaced / global registries still need the `use` -activation check on every call.
- **Slot-based parameter binding.** The resolver's `resolveMethod` assigns parameters to slots `0..N-1` in the call frame. At runtime `evalCall` borrows the call frame via `borrowBlockEnv(effectiveGlobal(env), len(m.Params))` and binds each parameter through `DefineAt(idx, ...)` . No map hashing per parameter; the resolver's slot numbers align with the interpreter's storage layout automatically.

Namespaced-call fast paths

Four more moves compound on top of the method-call frame work:

- **Pre-resolved namespaced calls / constants.** `QualifiedCallExpr` carries an opaque `Fn any` field; `QualifiedConstRefExpr` carries an opaque `Const any` field (see `internal/parser/ast.go` , kept `any` to avoid a parser -> interpreter import cycle). Because `parser.Resolve` runs before `processImports` , the pre-fill can't live in the resolver pass. `Interpreter.resolveQualifiedRefs` runs from `Run` right after `processImports` and walks the AST once, stamping the exact `Builtin` / `Value` the runtime would otherwise look up per call. `evalQualifiedCall` and `evalQualifiedConst` use the pre-filled pointer via type assertion; unresolvable prefixes stay nil and hit the original `resolveNamespacePrefix` + `NSBuiltins` path with its original error messages.
- **Int-int / float-float comparison fast paths.** `evalComparison` now checks both operand Kinds before falling through to `AsFloat` , mirroring the int-int block that already lived in `evalArithmetic` . Numeric loops (`$i < N`) skip two float conversions per iteration.
- **Immutable-Value copy elision in arg binding.** `evalCall` 's arg loop routes through `bindParamValue(v, declType)` . For scalar Kinds (`int` / `float` / `bool` / `null` / `string`) it returns `v` unchanged - both `Value.Copy` and `stampDeclaredType` were no-ops for those Kinds. Compound Kinds still go through the full copy + stamp path so value semantics + declared-type propagation stay correct.
- **effectiveGlobal caching via Environment.root.** Environment grew a `root *Environment` field set at construction time by the `rootFor(parent, self)` helper. Both `NewEnvironment*` and `borrowBlockEnv` populate it. `effectiveGlobal(env)` becomes an O(1) field read with a defensive parent-chain walk as fallback for hand-built envs. `releaseBlockEnv` clears `root = nil` on release so the pool doesn't retain a reference across borrow cycles.

Expression micro-optimizations

Two moves close out the optimization pass:

- **Constant folding at parse time.** `BinaryExpr` and `UnaryExpr` gained a `Folded Expr` field (`internal/parser/ast.go`). The resolver runs `tryFoldBinary` / `tryFoldUnary` (`internal/parser/fold.go`) after each subtree's operands resolve; when both operands are literal (transitively through their own `Folded` fields via the `asLit` helper), the operation runs at parse time and the result gets stamped on `Folded` . `evalBinary` / `evalUnary` check `Folded` first and delegate to it, skipping the operand walk + op switch. Runtime errors (division by zero, negative shift count) leave the node unfolded so the runtime hits the same error at the same source position - the fold pass never surfaces a parse-time error the runtime wouldn't have raised. Two exactness carve-outs mirror the runtime: an int/int comparison folds on the exact `int64` values (not a lossy `float64`), and a **mixed int/float comparison is left unfolded entirely** so the runtime's exact `compareIntFloat` decides it - promoting the int to `float64` at

fold time would lose precision above 2^53 and make `9007199254740993 == 9007199254740992.0` wrongly fold to `true`. `!=` folds wherever `==` does (it is the negation). Mixed int/float *arithmetic* still folds (the int->float promotion there matches the language's arithmetic semantics).

String interpolation

A cooked `"..."` string with one or more `{expr}` slots lexes to a single `TOKEN_STRING_INTERP` whose `Parts []lexer.StringPart` alternate literal chunks and expression-source slots. `readCookedString` drives this: `\{` / `\}` decode to literal braces, an unescaped `{` calls `scanInterpSlot` (which brace-counts nested `{}` so a map literal in a slot balances, and copies a nested string literal verbatim via `copyNestedString` so its braces / quotes do not disturb the count), and a bare `}` is a positioned lex error. Each expression slot records its **absolute** source line/col; the parser's `buildInterpString` sub-lexes it with `lexer.TokenizeAt` (seeding that position so diagnostics and resolver annotations point at the real spot) and sub-parses it to a single `expr` - a trailing non-EOF token is a "slot must be a single expression" error, an empty slot is rejected. The result is a `parser.InterpStringExpr` (`Parts []InterpPart`, each a literal `Lit` or an `Expr`). The resolver descends into each slot's `Expr` (undefined-variable / shadowing stay parse-time errors), so `lint` and `profile` - which walk the same tree - see slot expressions as ordinary code (`lint` adds the `L204` interpolation-slot-complexity check; `profile` attributes a slot's call to the slot's own position). Evaluation stringifies each slot value with `Display` (the `convert.toString` form, so no `use convert` is needed and any kind interpolates) and concatenates - the "sugar over concat + `convert.toString`" the milestone describes, kept as a node so the toolchain is not blind to slot code. A raw `'...'` string never interpolates (it lexes to a plain `TOKEN_STRING`); `fmt` re-emits a `TOKEN_STRING_INTERP` from its captured `Raw`, so slots survive a format verbatim.

Execution model

1. `Interpreter.Run(prog)` calls `parser.Resolve(prog)` first so the AST carries `(Depth, Slot)` annotations before any structural check runs. Resolve is idempotent: re-running on an already-resolved program produces the same annotations. Any undefined-variable or shadowing error surfaces here as a positioned parse-time diagnostic, not a runtime error.
2. Records `Imports` into `i.imported`, activating library namespaces. Module aliases are not loaded yet - that happens in step 4 - which is why the qualified-ref pre-resolution is deferred until after it.
3. Collects every `MethodDef` into `i.methods` (methods are hoisted: callable regardless of source order). During collection it enforces two rules: no duplicate method names, and no method name that collides with a registered builtin whose owning library has been imported (the no-shadowing rule extended to builtins - see `evalCall` below).
4. Creates the global `Environment` (`i.global`). Before executing the body, `loadModuleImports(prog)` loads and initialises every `import "..."` module (see Module loading), so an imported module is fully initialised before the importer's body runs. Then `resolveQualifiedRefs(prog)` walks the AST once and pre-fills every `QualifiedCallExpr.Fn` / `QualifiedConstRefExpr.Const` against the now-populated namespace AND module-alias tables. A library builtin, a library const, a module-alias const (`m.CONST` - its boundary-retagged, deep-const value cached on the node), and a module-alias method (`m.fn(...)` - a `moduleMethodTarget{mod, *MethodDef}` cached in `Fn`, dispatched through `dispatchModuleMethod` -> `CallMethodWith`) all resolve to a direct dispatch instead of a per-access lookup (an unexported / missing name stays unstamped, so its runtime error is unchanged). This pass runs after BOTH import mechanisms (the tables didn't exist during Resolve) and is skipped by the REPL, which builds its namespaces incrementally. Finally executes `prog.TopLevel` statements in source order in the global scope.
5. Method calls execute the body in a fresh call frame whose parent is `effectiveGlobal(env)` (an O(1) `env.root` field read; in serial code that's `i.global`, inside a `spawn` body it's the spawn-globals snapshot). The call frame is borrowed from the environment pool pre-sized to the parameter count; parameters bind through `DefineAt` into slots `0..N-1`; the callee is looked up via the pre-resolved `CallExpr.Method` pointer when set, falling back to `i.methods` when it isn't (REPL turns). Scalar-Kind arguments skip the Copy + declared-type-stamp step via `bindParamValue`. Top-level variables are visible

inside methods (subject to the no-shadowing rule). The body's return value (bare `return;` -> `null` ; `return EXPR;` -> the expression's value; falling off the end -> `null`) propagates back to the caller.

`EvalInteractive` (the REPL entry point) skips step 1 - each REPL turn is a fresh parse whose scope can't be resolved without the accumulated global context from prior turns. The runtime handles this by leaving resolver annotations at their `-1` sentinel and using the name-based Environment API. The perf cost is limited to REPL sessions, which aren't hot loops.

There is no required entry point. A program with only imports and method defs is valid and runs to completion immediately (those methods are simply never called).

Module loading

`import "PATH.j" [as NAME];` (an `ast.ModuleImportStmt`) loads another `.j` file as a module. `internal/interpreter/module.go` holds the loader; `internal/module` holds the path resolver.

`EnableModules(baseDir, searchDirs, load, setup)` wires the system onto the root interpreter and builds a `moduleReg` shared across the whole run:

- `cache map[string]*loadedModule` - the run-once table keyed by resolved absolute path.
- `stack []string` - canonical paths currently loading, for cycle detection.
- `search []string` - the module search path (system module dir, then each `-I` dir).
- `load func(string) (*parser.Program, error)` - lex + preprocess + parse a resolved file. The CLI passes `main.go` 's `loadModuleProgram` ; tests pass an equivalent.
- `setup func(*Interpreter)` - install the standard library into a module's fresh sub-interpreter. The CLI passes `installLibraries` .

`loadModuleImports(prog)` runs from `Run` (step 4) before the body. For each import it calls `loadModule(path, at)` :

1. `module.Resolve(path, baseDir, search)` -> canonical absolute path (local `./` / `../` and absolute `/` resolve directly; a bare name walks `search` , where a name found in two search dirs is a hard error). Resolution errors are positioned at the import statement.
2. **Cycle check** - if the canonical path is already on `reg.stack` , error `module cycle: A -> B -> C -> A` naming every edge.
3. **Run-once** - if it's already in `reg.cache` , return the cached `*loadedModule` without re-running.
4. `reg.load(canonical)` parses the module (parse errors stay positioned in that file).
5. A **fresh sub-interpreter** is the module's own scope: `sub := New(); reg.setup(sub); sub.modReg = reg; sub.baseDir = dir(canonical)` . Sharing `reg` means the sub's own imports use the same cache and stack.
6. Push the canonical path, `sub.Run(modProg)` (which recurses into the sub's imports, then runs its body), pop the path, and cache the result.

The recursion is what delivers the guarantees: **run-once** from the cache, **depth-first post-order init** from initialising a module before its importer's body, and **cycle detection** from the load stack. Load errors (a parse error or a `throw` during a module's top level) propagate out of `Run` as ordinary errors and fail the program; they are **not** catchable, because an `import` is a top-level declaration, not an expression, so it cannot sit inside a `try` / `catch` (the parser rejects `import` in a block).

Module scope and namespacing

A module's top level is **declarations-only** : `checkModuleDeclarationsOnly` (run in `loadModule` after parse) allows only `def` `const` in `TopLevel` - structs, methods, and imports live in their own `Program` slices - so a mutable `def` or a free-standing statement is a positioned load-time error. Scripts run through the CLI never reach this check, so a `jennifer run` program keeps mutable top-level `def` and free-standing statements.

`loadModuleImports` binds each import's alias (the `as NAME` clause, or the file stem via `moduleStem`) into the importer's `moduleAliases` (`map[string]*loadedModule`), collision-checked against library prefixes (`nsPrefixes`) and other module aliases. Consumer-side resolution rides the existing qualified-reference eval

layer, since the parser resolver and `resolveQualifiedRefs` already defer unknown prefixes to runtime:

- `evalQualifiedCall` checks `moduleAliases` before the library path and, on a hit, dispatches `alias.fn(args)` through `callModuleMethod` - arguments evaluated in the consumer's env, then `m.interp.CallByNameWith` runs the body against the **module's** globals and methods. Arity / type mismatches reposition at the consumer's call site; a `runtime error` , `throw` , or `exit` from the module body propagates unchanged.
- `evalQualifiedConst` reads `alias.CONST` from the module's global scope (`m.interp.global.GetBinding`).

`use` non-transitivity, run-once sharing, and `-race` safety all fall out of the fresh-sub-interpreter-per-module model: a module's interpreter holds only immutable constants and read-only methods, so concurrent calls from parallel `spawn` bodies share no mutable state.

Exports, visibility, and cross-module struct identity

`export` marks a top-level `func` / `def` `struct` / `def` `const` as a module's public surface (`parseExported`); `collectExports` records the set on the `loadedModule` , and `callModuleMethod` / `moduleConst` / `evalStructLit` / the `def` -type check reject an unexported `alias.member` with a positioned "not exported from module" error. `export` is illegal in a script: `Run` calls `rejectExportInScript` unless `isModule` is set (only `loadModule` , and the `jennifer test` overlay via `SetModuleContext` , set it). `checkReferentialClosure` (at load) rejects an exported struct field or exported function parameter typed as a *private* module struct; only the module's own bare struct types are checked, so library / namespaced types cross freely.

Cross-module struct identity is boundary translation, not internal tagging. A module's own structs stay bare (`StructNS ""`) inside the module, where all the existing struct machinery works unchanged. `retagStructs` re-tags them to `(module-stem, name)` as a value crosses *out* to an importer (a `callModuleMethod` return, a `moduleConst` read) and back to bare on the way *in* (a `callModuleMethod` argument), recursing through struct fields, list elements, and map values; library / other-module structs (a different namespace) are untouched. So the consumer can type a module struct (`def p as points.Point` , or `def ps as list of points.Point` ; the `def` -check's `resolveDeclaredStructNS` stamps the type's namespace to the module stem, recursing into list / map / task element types so a collection element type matches the identity `retagStructs` gives the values), construct one (`points.Point{...}` in `evalStructLit`), read its fields, and pass it back - all type-checking - while `a.Point` and `b.Point` stay distinct `(namespace, name)` pairs. Struct identity is keyed by the module's **canonical path** , carried in the `ModPath` field on both `Value` and `parser.Type` (empty for library / user structs) and compared by `Value.Equal` / `MatchesDeclared` ; `StructNS` holds the file **stem** purely for display, so two module files sharing a basename (`a/util.j` , `b/util.j` , or `@jennifer/benchmark` vs `@alice/benchmark`) are genuinely distinct types while both still render as `benchmark.Point` . The boundary retag threads `(StructNS, ModPath)` so a foreign struct that only shares the stem is left untouched, and method parameter types are stamped alongside `def` types so a `func f(s as mod.Struct)` param carries the same identity the passed value does. The retag copies only compound values at the boundary (module calls are not a hot path). Declared-type stamping happens once, single-threaded, before execution: `resolveDeclaredTypesOnce` (run from `Run` after `loadModuleImports`) walks every declared type - top-level, method bodies, and spawn bodies - stamps it, and marks the AST node `parser.Type.Resolved` . The per-execution `resolveDeclaredStructNS` in `execDefine` early-returns on a resolved node, so a `def` inside a loop, or a shared method / spawn body reached from concurrent goroutines, re-reads the marker instead of re-stamping - no write-write race on the shared type node. The pass is best-effort (an unresolvable type is left for `execDefine` to error on at its original position), and the marker also sidesteps a latent idempotency gap: re-resolving an already-stamped *library* alias would otherwise hit the "canonical namespace is aliased" rejection. (An importer alias to a module stem is also recognised on any later pass via `moduleByNS` .)

Func values across the module boundary

A `func` value is a handle to a top-level method (`Value.Fn *parser.MethodDef`). It also carries

Value.FnHome *Interpreter - the interpreter that *defined* the method - stamped when a bare method name is evaluated as a value (`i.methods[name]` -> `FuncVal(m)` with `FnHome = i`). This matters only across a module boundary: a func value is a value like any other, so it is routinely passed *into* a module (a `lists.map` predicate, an `args.dispatch` handler, a `webapi` authenticator) or returned *out* of one. Without a home, `evalCallValue` would run the body in the **caller's** interpreter, whose namespace table does not carry the func's own `use` / `import` aliases - so a body that references `strings.upper` or constructs `mymod.Thing{...}` would fail with "unknown namespace" against the wrong interpreter.

So `evalCallValue` , when `fv.FnHome != nil && fv.FnHome != i` , dispatches the call to the home: it evaluates the arguments in the caller, then runs `home.callMethodWithDepth(fv.Fn, env.depth, args...)` - the body executes in its home context, its aliases resolve, and `env.depth` threads the caller's call-depth counter so recursion bouncing through a cross-boundary func call still trips the catchable guard (as it does through `dispatchModuleMethod` / `meta.callMain`). Same-interpreter calls (`FnHome == i` , or a legacy `nil` home) keep the ordinary `callUserMethod` path with no retag.

Struct arguments and the return value cross the boundary through **crossRetag(v, from, to)** , which reuses `retagStructs` (above) in a two-pass form: `to` 's own structs - tagged with `to` 's identity as `from` sees them - are un-tagged to bare inside `to` , then `from` 's own structs - bare inside `from` - take `from` 's identity so `to` recognises them as that module's structs. Because a struct belongs to exactly one module, the two passes target disjoint owners, so the *same* helper serves both directions (arguments crossing in, the result crossing out) and every pairing (host->module, module->host, module->module). The webapi authenticator is the worked case: a host func value returning a `webapi.Identity` (built through the host's `webapi` alias, so tagged `(webapi-stem, webapi-path)` in the host) is called from inside the `webapi` module and its return is retagged bare, exactly the identity the module's own code expects. `FnHome` is a shared pointer, copied with the (immutable) func value like `Fn` , and never mutated. Pinned by `cmd/jennifer/funcvalue_home_test.go` and `args_dispatch_test.go` .

Builtins and libraries

Each library lives in its own Go package under `internal/lib/<name>/` and registers its functions (and constants) on the interpreter. User-facing reference docs are split per library:

- libraries/io.md - `printf` , `sprintf` , format verbs
- libraries/convert.md - `int` , `float` , `string` , `bool` , `typeOf`
- libraries/math.md - `math.abs` , `min` , `max` , `sqrt` , `pow` , `floor` , `ceil` , `round` , `rand` , `randInt` , `randSeed` ; constants `math.PI` , `math.E`
- libraries/strings.md - `upper` , `lower` , `contains` , `startsWith` , `endsWith` , `indexOf` , `trim` / `trimLeft` / `trimRight` , `replace` , `repeat` , `substring` , `split` , `chars` , `join`
- libraries/os.md - `os.PLATFORM` , `os.ARCH` , `os.EOL` , `os.DIRSEP` , `os.PATHSEP` , `os.ARGS` , `os.getEnv` , `os.hasFlag` , `os.flag` , `os.run` , `os.spawn` , `os.wait` , `os.poll` , `os.kill`
- libraries/meta.md - `meta.VERSION` (build version), `meta.BUILD` (toolchain)
- libraries/index.md - catalog and organizing principles
- `len(EXPR)` is a language built-in primary, not a library function. See grammar.md.

What follows is the implementation contract, not the user-facing API.

Library functions are Go closures registered with the interpreter:

```
type BuiltinCtx struct {
    Out    io.Writer  // stdout-like effects write here
    In     io.Reader  // stdin-consuming builtins read here
    InREPL bool       // true when the call originates from the REPL
}
type Builtin func(ctx BuiltinCtx, args []Value) (Value, error)

# In a library package:
func Install(in *interpreter.Interpreter) {
    in.Register("io", "printf", printf)
    in.Register("io", "sprintf", sprintf)
    in.Register("io", "readLine", readLine)
    in.Register("io", "eof", eofFn)
}
```

`BuiltinCtx` replaces an earlier `(out io.Writer, args)` signature to give input-consuming builtins symmetric access to stdin and the REPL flag. `Interpreter.In` defaults to `os.Stdin` ; the REPL sets `Interpreter.InREPL = true` so `readLine` / `eof` refuse rather than racing the line editor for input.

`Interpreter.Builtins` stores `builtinEntry{Lib, Fn}` per name. A call to `foo(...)` resolves in this order:

1. User-defined method `foo` in `i.methods` .
2. Builtin `foo` - **but only if its owning library has been used** . The error otherwise quotes the right library name: `foo` requires `use <lib>;` .

The no-shadowing check at hoist time uses the same lookup: a user method that collides with an imported library's builtin is rejected.

User-defined **constants** (via `def const NAME as TYPE init EXPR;`) live in the same Environment as variables and resolve through `evalExpr` 's `ConstRefExpr` case (bare-identifier lookup). They participate in the no-shadowing rule like everything else.

Library-provided constants (`math.PI` , `math.E` , `time.UTC` , `time.PROGRAM_START` , `os.PLATFORM` , ...) are namespaced and registered through `RegisterNamespacedConst` . They resolve through `QualifiedConstRefExpr` - see the "Namespaced libraries" subsection below. The `RegisterConst` flat-namespace constant API and the bare-IDENT `ConstRefExpr` fallback for library constants are no longer used by any shipping library; the fallback path remains in the interpreter as exported API surface pending a final cleanup pass.

Namespaced libraries

Domain libraries register through the namespaced API:

```
in.RegisterNamespaced("os", "platform", platformFn)
in.RegisterNamespacedConst("os", "JENNIFER_OS", interpreter.StringVal("linux"))
```

Both entries are keyed by `(namespace, name)` in `NSBuiltins` / `NSConstants` . The library's name doubles as the namespace prefix (future libraries may decouple them, but today they always match). Registering through the namespaced API also flags the lib in `knownNamespaces` .

Only libraries flagged in knownNamespaces may be aliased. `processImports` rejects `use NAME as ALIAS;` for any library that registered exclusively through `Register` / `RegisterConst` (the flat API) with the message `library NAME has no namespaced builtins; + " as ALIAS " + aliasing is meaningless here` . The flat libraries (`io` , `convert` , `math` , `strings` , `core`) all fall into this category - they have no prefix to rename, and silently accepting an `as` clause would create the misleading impression of an alias-shaped escape hatch.

`processImports` builds two maps from each `use NAME [as ALIAS];` :

- `nsPrefixes[prefix]` = `canonicalNamespace` - the prefix that's active at call sites; `prefix` == `canonical` for `use os;` , `prefix` == `alias` for `use os as o;` .
- `nsAliasedAway[canonical]` = `alias` - records that the canonical name has been shadowed by an alias, so a later `os.foo()` after `use os as o;` errors with a `did you mean` + " `o` " + `?` hint.

Resolution at a `QualifiedCallExpr` / `QualifiedConstRefExpr` goes through `resolveNamespacePrefix(prefix)` :

1. If `prefix` is in `nsPrefixes` , use the canonical namespace it points at.
2. Else, if `prefix` is in `nsAliasedAway` , emit the "did you mean `<alias>` ?" hint.
3. Else, if `prefix` is the canonical name of a *known* namespaced lib the program forgot to `use` , emit a `requires` + " `use prefix;` " reminder.
4. Else, emit `unknown namespace` .

The no-shadowing rule for top-level methods (`checkMethodNoShadow`) adds one more clause: a method name that matches an active namespace prefix is rejected (`func os() {}` errors after `use os;` , but is fine after `use os as o;` because only o is reserved as a prefix).

The five essential flat libraries (`io` , `convert` , `math` , `strings` , `core`) intentionally do *not* use the namespaced API - their names stay bare for ergonomics.

For the user-facing API of each library, follow the links above. Below are the implementation-only notes worth knowing as a maintainer.

internal/lib/io : `printf` and `sprintf` share a `formatArgs` helper with three shapes - 0 args errors; first-arg-is-string triggers format substitution; single non-string arg writes the value's `Display()` form (the "just print this value" shortcut). `printf` writes to `Interpreter.Out` ; `sprintf` returns a `KindString` value and ignores the writer.

internal/lib/math : `floor` / `ceil` / `round` accept int (identity) or float and return `int` . `round` uses Go's `math.Round` (half away from zero). `math.PI` and `math.E` are registered via `RegisterNamespacedConst` and resolved through `QualifiedConstRefExpr` like every other namespaced constant; the namespace prefix is reserved for the rest of the program once `use math;` runs.

internal/lib/convert : conversions live behind the namespace - `convert.toInt(v)` , `convert.toFloat(v)` , `convert.toString(v)` , `convert.toBool(v)` . A bare type keyword in expression position (`int(...)` , `bool(...)`) is a positioned parse error whose hint names the matching `convert` builtin. `typeOf` is a normal IDENT call. `convert.toBool(v)` implements canonical-only conversion at all source kinds (`0` / `1` for int, `0.0` / `1.0` for float, `"true"` / `"false"` for string) - non-canonical values produce a positioned error, not silent coercion.

internal/lib/strings : all indices and lengths are **rune-based** (Unicode code points), implemented via `unicode/utf8` . `len` returns the rune count; `indexOf` returns a rune index (not the byte index Go's `strings.Index` produces - we translate); `substring` uses a small `byteOffsetForRune` helper to convert rune-indexed bounds back to byte slicing on the underlying string. `repeat` guards against multiplication overflow before calling Go's `strings.Repeat` to avoid the panic in the standard library. The Go package is named `stringslib` to avoid colliding with the standard `strings` package, which it depends on heavily.

Runtime errors

`*runtimeError` carries optional `File` / `Line` / `Col` and a `Kind` tag (defaults to `"runtime"` when the originating site doesn't specialise it). Errors render as `runtime error at FILE:L:C: <msg>` (or `runtime error at L:C: <msg>` when the file is unknown). All five Jennifer error types - `*lexer.LexError` , `*preproc.PreprocessError` , `*parser.ParseError` , `*runtimeError` , and `*ErrorSignal` - implement a small `Position() (file string, line, col int)` interface. The CLI uses that interface (no string parsing) to look up the right file and print a caret under the offending source line.

Catchable errors

`try { body } catch (NAME) { handler }` runs the body and, on an error, binds the thrown value to `$NAME` in

a fresh per-handler scope. Two sentinel paths can produce the catchable error:

- ***ErrorSignal** - raised by `throw EXPR;` (`execThrow`). Carries the thrown `Value` plus the throw's source position. Uncaught signals reach the CLI through the same `positioned` interface as `*runtimeError` .
- ***runtimeError** - raised by any builtin or language operation (out-of-bounds index, missing map key, type mismatch, etc.). When one reaches an enclosing `try` , `execTry` wraps it via `runtimeErrorToValue` into an `Error` struct (`kind` , `message` , `file` , `line` , `col`) and binds it like any other thrown value.

`*ExitSignal` is **not** routed through this path - the spec puts process exit outside the recoverable-error scope, so `execTry` propagates it untouched. `blockResult` flags (`hasReturn` , `hasBreak` , `hasContinue`) flow through `execTry` unchanged so the surrounding method / loop sees them.

The canonical `Error` struct is auto-hoisted into `i.structs` by both `Run` and `EvalInteractive` before any user struct definition runs (`canonicalErrorStructDef()`). User code may not redefine it - the existing duplicate-struct check fires with `struct "Error" is defined more than once` .

`runtimeError.Kind` is the symbolic tag surfaced as `$err.kind` in the catch block. The current shipping default is `"runtime"` ; specific tags (`"out_of_bounds"` , `"type_mismatch"` , etc.) get filled in per call site as user code grows demand for finer dispatch.

Errors and positions (cross-file)

The pipeline plumbs file information through three layers:

1. The lexer attaches the source file path to every token (`Token.File`). `TokenizeWithFile(source, file)` is the entry point; the no-arg `Tokenize` leaves `File` blank for unattributed input.
2. The preprocessor preserves each spliced token's `File` field when resolving `import "path.j";` , so tokens from an imported file keep that file's path, line, and column.
3. The parser propagates `File` from tokens to every AST node (each `pos` struct carries `File` , `Line` , `Col`). Synthesized nodes (e.g. `BinaryExpr`) copy the file from the left operand.

When the interpreter raises a `*runtimeError` , it pulls file/line/col from the offending node via a small `posFor(node)` helper. The CLI's `printErrorContext` type-asserts the `positioned` interface, and if the reported file differs from the program's main file it loads that file from disk before slicing out the snippet to display.

Concurrency

The `spawn` keyword, the `task of T` type kind, and the `task` library together form Jennifer's first concurrency surface. The user-facing model is docs/user-guide/concurrency.md; this section describes the runtime side.

Goroutine mapping

`spawn { ... }` is a primary expression: `parser.SpawnExpr` carries the body as `[]Stmt` . The interpreter handles it through `evalSpawn` :

1. Build a fresh capture environment with `snapshotForSpawn(env)` .
2. Allocate a `TaskState` with a freshly made `Done chan struct{}` .
3. Register the state in the interpreter's per-run task registry.
4. `go i.runSpawn(state, ex, spawnEnv)` .
5. Return a `Value` of kind `KindTask` wrapping the state pointer.

`runSpawn` closes `state.Done` from a `defer` so all observers (`task.wait` , `task.waitAll` , `task.waitAny` , the exit-time scan) see the close as a happens-before edge before reading `state.Result` or `state.Err` . The goroutine itself executes the body via the existing `execBlock` over the captured env; this is the same path top-level statements take, so the spawn body sees the full interpreter (libraries, structs, method definitions, namespacing).

`return EXPR;` in the body becomes `state.Result` . A `blockResult` with `hasReturn=false` but no error means an implicit `null` return (matches method-call semantics). `break` or `continue` that escapes its loop

inside the body becomes a positioned error (" `break` outside a loop" / " `continue` outside a loop") via `unhandledLoopFlowError` ; loop-flow can't cross the spawn boundary, mirroring how it can't cross a method-call boundary.

Value-semantics capture

`snapshotForSpawn(env)` is the data-race story. It builds a two-frame chain: a "globals" frame holding deep copies of every `i.global` binding, and a "locals" frame chained on top holding deep copies of every non-global binding visible at the spawn site. The spawn body runs against the locals frame; user-method call frames inside the spawn parent through `effectiveGlobal` and land on the globals frame.

```
func (i *Interpreter) snapshotForSpawn(env *Environment) *Environment {
    globalSnap := NewEnvironment(nil)
    for name, b := range i.global.vars {
        globalSnap.vars[name] = b.deepCopy() // globals -> own frame
    }
    localSnap := NewEnvironment(globalSnap)
    for cur := env; cur != nil && cur != i.global; cur = cur.parent {
        for name, b := range cur.vars {
            if _, seen := localSnap.vars[name]; seen { continue }
            localSnap.vars[name] = b.deepCopy()
        }
    }
    return localSnap
}
```

The two-frame shape is what lets user-function calls inside the spawn keep their normal scoping. A user method's call frame inherits from the *global* surface only - never from the caller's locals (Jennifer's "no inheriting caller scope" model). Inside a spawn, the call frame's parent comes from `effectiveGlobal(env)` :

```
func effectiveGlobal(env *Environment) *Environment {
    cur := env
    for cur != nil && cur.parent != nil {
        cur = cur.parent
    }
    return cur
}
```

In serial code `env` chains to `i.global` , so `effectiveGlobal` returns `i.global` . In a spawn body `env` chains to the snapshot's globals frame, so `effectiveGlobal` returns that frame. Both paths honour the no-shadowing rule the same way (parameters never collide with captured locals, only with true globals), and the spawn body's user-method calls are race-free because they never touch the live `i.global` the parent goroutine may be writing.

Deep-copy reuses the same `Value.Copy()` path as `$ys = $xs;` and function-parameter binding, so lists, maps, bytes, and structs copy at any depth.

The one exception is `KindTask` itself. A `task of T` value deliberately copies the *pointer* to the underlying `TaskState` , not the state - multiple variables pointing at "the same spawn" must observe it together. Without this, `def u as task of T init $t;` would clone the in-flight goroutine handle and break observation counting.

Task registry and loud-fail

The interpreter carries

```
type Interpreter struct {
    // ...
    tasksMu sync.Mutex
    tasks   []*TaskState
}
```

`evalSpawn` calls `registerTask(state)` . Each `TaskState` carries an `Observed bool` flag that any of the three "I saw this" operations flips: `task.wait` (both on success return and on rethrow), `task.discard` , and `task.waitAll` (drains every survivor before re-raising).

`Interpreter.UnwaitedTaskErrors()` runs at the end of CLI execution:

```
func (i *Interpreter) UnwaitedTaskErrors() []error {
    i.tasksMu.Lock()
    snapshot := append([]*TaskState(nil), i.tasks...)
    i.tasksMu.Unlock()
    var errs []error
    for _, t := range snapshot {
        if t == nil || t.Observed { continue }
        <-t.Done                        // happens-before edge
        if t.Err != nil { errs = append(errs, t.Err) }
    }
    return errs
}
```

It deliberately blocks on `<-t.Done` for every unobserved task. The "no footguns" rationale: a non-blocking scan could miss a late-arriving error and silently exit cleanly. Blocking buys the loud-fail guarantee at the cost of hanging the program when an unobserved goroutine never finishes (a `spawn { while (true) {} }` without `task.discard`). The user-guide flags this as a footgun of its own; the runtime trade-off favours soundness.

`cmd/jennifer/main.go` consumes the slice: after `Run(prog)` returns, it walks `UnwaitedTaskErrors()` , prints each one to stderr in `spawn error (unwaited): MSG` form, and bumps the process exit code if any were present. `ExitSignal` from a body is special-cased in the loud-fail surface (treat as a normal program-level exit, not a "task error") so user-explicit shutdowns don't print spurious "unwaited" lines.

task library Go layer

`internal/lib/task` registers five namespaced builtins through the standard `RegisterNamespaced` path:

Builtin	Path
task.wait	block on <-state.Done; MarkObserved; return Result or wrap Err as runtimeError
task.poll	BoolVal(state.IsDone()) via the non-blocking select on state.Done
task.discard	MarkObserved; return Null() immediately (does not block)
task.waitAll	iterate list, wait each, mark all observed; return list-of-results or first error in order
task.waitAny	[]reflect.SelectCase over the list, reflect.Select, return chosen index

`reflect.Select` is the one place the runtime drops into reflect; acceptable because the list length is dynamic and `select { ... }` on a variable arm count has no other Go-level construction. The TinyGo target supports it for chan-receive cases; verified by the package tests passing under both compilers.

`MarkObserved` is a thin wrapper around setting the flag under the registry mutex (no atomics - the field is read only by the exit-time scan, which already takes the mutex). The pattern is "observation = explicit consent that this task's outcome is yours"; the loud-fail path is the only place reads happen outside the consenting frame.

Type stamping for task of T

`parser.TypeTask` joins `TypeList` / `TypeMap` / `TypeBytes` / `TypeStruct` in the `Type.Kind` enum.

`Type.Element` holds the `T` for `task of T` ; `Type.String` and `Type.Equal` handle recursion the same way as `list of T` . `MatchesDeclared` rejects non-task values and (when the declared element type is concrete) walks the wrapped task's `ElemTyp` to enforce element-type compatibility - so `def t as task of int init spawn { return "x"; };` fails at the use site, not deep inside the spawn body.

CLI integration

`main.go` (batch path), `repl.go` (interactive path), `fmt_test.go::runProgramOutput` (golden-test harness), and `examples_test.go` all `tasklib.Install(in)` alongside the other libraries. The REPL path also calls `UnwaitedTaskErrors()` between inputs - a spawn that errored in line N surfaces before the prompt for line N+1, so the REPL session can't accumulate silent failures.

What's deferred

The runtime side has more breathing room than the user-facing surface. The deferred pieces:

- **Channels.** No `chan T` type, no `send` / `recv` builtins. The spawn/task pair handles the common cases; a channel primitive would add real bookkeeping and is a later candidate.
- **Cancellation.** No way for an outsider to stop a running spawn body. Open design question (cooperative vs hard abort vs structured-concurrency tree).
- **Structured concurrency.** No automatic scope-bounded termination. The loud-fail registry is the lighter-weight answer.
- **Timeouts.** Compose with a `time.sleep` sentinel + `task.waitAny` ; a higher-level helper may ship later.
- **Refcounted copy-on-write for Value.** The O(N) deep-copy cost of spawning over a large captured collection is a known cost of the value-semantics model and the same cost that hits `$ys = $xs;` in serial code. A refcounted copy-on-mutation optimisation in the `Value` runtime would help both paths; not scheduled.

CLI (cmd/jennifer)

```
jennifer run [flags] <file.j> [args...]  run a Jennifer program
jennifer run -             read source from stdin
jennifer repl              interactive REPL
jennifer tokens <file.j>   dump the lexer's token stream
jennifer ast <file.j>      dump the preprocessed AST as JSON
jennifer fmt [-w] <file.j>...  format source per style-guide.md (-w rewrites in
place)
jennifer lint <file.j>     report compile-legal but suspect patterns
jennifer profile <file.j>  profile hit counts and wall-clock per source position
jennifer test <file.j>     discover and run the file's test methods
jennifer version [-v]      print the build version (-v adds module-path layers)
jennifer help              show usage
```

Subcommand reference

The `jennifer` command-line tool bundles the Jennifer interpreter and its full development toolchain into a single binary. Beyond running `.j` programs, it provides an interactive REPL, a source-code formatter, a linter, a profiler, a test runner, and lexer-token and AST inspection - so a whole Jennifer workflow (write, run, format, lint, profile, test) needs no extra tools. Each subcommand is summarised below and documented in depth on its own page. The development subcommands (`tokens` , `ast` , `fmt` , `lint` , `profile` , `test`) and `serve` live in the default `jennifer` binary only; `run` , `repl` , and `version` work on both `jennifer` and `jennifer-tiny` .

Subcommand	What it does	Details
run <file.j>	Run a Jennifer program (- reads source from stdin).	Module resolution flags
repl	Interactive read-eval-print loop, with a line editor and history.	REPL
tokens <file.j>	Dump the lexer's token stream.	Inspection
ast <file.j>	Dump the preprocessed AST as JSON.	Inspection
fmt [-w] <file.j>...	Format source per the style guide (stdout; -w/--write rewrites in place).	Formatter
lint <file.j>	Report compile-legal but suspect patterns.	Linter
profile <file.j>	Per-position hit counts and wall-clock timings.	Profiler
test <file.j>	Discover and run the file's test methods.	Test runner
serve <file.j>	Run a program; --watch re-runs it on change (a web-app reloader, or an autorun loop for any script).	The serve command
version [-v]	Print the build version (-v adds module-path layers).	Version injection
help	Show usage.	-

Module resolution flags (run)

`jennifer run` accepts interpreter flags before the source file; anything after the file is the program's own `os.ARGS` .

- `--sysmoddir DIR` (or `--sysmoddir=DIR`) - the system module directory for bare `import "name.j";` . Overrides `JENNIFER_SYSMODDIR` , which overrides the compile-time default. A named (CLI or env) dir that is missing or not a directory refuses to start; the compile-time default is best-effort. The resolved value is `meta.SYSMODDIR` .
- `-I DIR` (or `-I=DIR` , repeatable) - add a directory to the module search path after the system dir. A `-I` dir *adds* names; a module name appearing in two search dirs is a hard error at load. (Resolution lives in `internal/module` ; the `import` statement consumes it via the loader wired in `main.go` 's `runFile` -

`in.EnableModules(baseDir, searchDirs, loadModuleProgram, installLibraries)`, where `searchDirs` is the system dir followed by each `-I` dir.)

- `--vendor DIR` (or `--vendor=DIR`) - the vendor root for `@scope/package` deck imports. Overrides `JENNIFER_VENDOR`, which overrides the upward walk to the nearest `vendor/` directory above the program. `@scope/package/` expands to `<vendorRoot>/scope/package/package.j` (see the import spec); wired via `in.SetVendorRoot(module.FindVendorRoot(vendorFlag, baseDir))`. `repl` / `test` use the upward walk (no flag).
- `--env PROFILE` (or `--env=PROFILE`) - a **run profile**, pure sugar that sets `JENNIFER_ENV=PROFILE` in the process environment before `Run`, so `jennifer run --env=prod app.j` is identical to `JENNIFER_ENV=prod jennifer run app.j`. No interpreter change - the program reads the profile from the one env var (the `dotenv` module's `.env.<profile>` selection is the first consumer). The label is validated (`[A-Za-z0-9_-]`, 1-64 chars, mirroring dotenv's own `validProfile`), which also blocks a `.env.<profile>` path traversal; an explicit `--env` overrides any inherited `JENNIFER_ENV`. Parsed in `main.go`'s `parseRunArgs` / `validRunProfile`.

`jennifer version -v` reports every system directory the resolver uses - the system module dir and the vendor root - each with the layers (compile default / `JENNIFER_SYSMODDIR`, and env / `vendor/`-walk) behind it.

- Verifies the `.j` extension - unless the file's first line is a `#!` shebang, in which case any name is accepted (so an executable script installed under a bare command name, e.g. `/usr/bin/mytool`, runs without a `.j` suffix). Only the entry point is exempt; `import` / `include` targets still require `.j`.
- Reads the file, parses, runs
- On error: prints the message and a source-context caret on stderr, exits `1`
- Bad usage exits `2`
- `jennifer help` includes a `Version:` line so the build is identifiable at a glance
- `tokens`, `ast`, `fmt`, `lint`, `profile`, `test`, and `serve` are present only in the default `jennifer` binary (the run-only `jennifer-tiny` build stubs them); the first six are development subcommands with their own pages below, and `serve` runs a net-backed `web` app so it, too, is default-only.

The serve command

`jennifer serve <file.j>` runs a program the same way `run` does - there is no entry point, so the file's top-level executes in order and serving happens only because the program itself calls `web.run(...)`. On its own, `serve` adds just a banner (printed after a clean parse). Its reason to exist is `--watch`:

```
jennifer serve app.j            # run the app
jennifer serve app.j --watch    # re-run on every change to the entry file
```

With `--watch`, `serve` runs the program in a **child process** and restarts it whenever the entry file changes: it polls the file's modification time and, on a change, kills the child and starts a fresh one. Ctrl-C stops the loop.

Two uses fall out of the same mechanism:

- **A web-app reloader.** A long-running `web.run` server never exits on its own; save a handler and `--watch` restarts it against the new code - the Hugo-style edit / reload loop.
- **A general autorun loop.** For a program that *finishes* (any script, not just a server), the child exits, `serve` prints `app exited; waiting for a change to reload...`, and parks until the next edit - so `jennifer serve --watch script.j` is an edit-and-rerun harness for anything: save, see the output, save again. The watcher deliberately stays alive after a clean exit (a loop that quit the moment a server died on a syntax error would defeat the point).

Only the **entry file** is watched today - changes to `include`d files or imported modules do not trigger a reload. `serve` is default-binary-only (the `httpd` engine is net-backed, and `--watch` uses `os/exec`); see the `web` module.

Shell pipelines and aliases

Because `jennifer run -` reads a program from stdin (the `run -` form above), Jennifer drops into a shell pipeline like any other filter. One caveat sets the shape: stdin can carry *either* the program *or* the data, not both. So a one-liner alias pipes the program in and passes the data as an argument (read back through `os.ARGS`); a reusable filter keeps the program in a file and leaves stdin free for the data.

Inline, program piped via run -. A `json-pretty` that reformats a single JSON argument. The program arrives on stdin, so the JSON is `os.ARGS[1]` (`ARGS[0]` is the `-`):

```
alias json-pretty="printf '%s' 'use json; use os; use io; io.printf(\"%s\\n\",
json.encodePretty(json.decode(os.ARGS[1])));' | jennifer run -"

json-pretty '{"b":2,"a":1}'
# {
#   "b": 2,
#   "a": 1
# }
```

Use `printf '%s'` rather than `echo` to pass the program verbatim, so the `\n` reaches Jennifer as a two-character escape instead of being expanded by the shell.

Reusable filter, data on stdin. For a true pipe (`... | json-pretty` , `json-pretty < file.json`) keep the program in a file and let stdin carry the data. Save this as, say, `~/.local/share/jennifer/json-pretty.j` :

```
use json;
use io;

def src as string init "";
while (not io.eof()) {
    $src = $src + io.readLine() + "\n";
}
io.printf("%s\n", json.encodePretty(json.decode($src)));
```

```
alias json-pretty='jennifer run ~/.local/share/jennifer/json-pretty.j'

echo '{"b":2,"a":1}' | json-pretty
curl -s https://api.example.com/thing | json-pretty
```

The same shape extends to any decode / re-encode pair: swap `json` for `xml` once that library lands (M20.2) and the file becomes a `pretty-xml` .

Version injection

`internal/version.Version` holds the build version as a single `string` . The default is `"dev"` ; the `Makefile` runs `scripts/gen-version.sh` before each build, which writes `internal/version/version_gen.go` containing a small `init()` that overwrites `Version` with the output of `scripts/version.sh` (a `git describe --tags --long` derivative; see ../libraries/meta.md for the format).

This codegen path replaces the more conventional `go build -ldflags "-X .Version=..."` because TinyGo 0.41 silently ignores the `-X` directive. Codegen works identically on both toolchains. The generated file is `.gitignore` d so the repository never carries a stale copy.

Two consumers read `version.Version` :

- `cmd/jennifer/main.go` prints it in the `help` banner and as the body of the `version` subcommand.
- `internal/lib/meta/metalib.go` mirrors it into the interpreter as the `meta.VERSION` constant. The `meta` library is opt-in like every other library: `use meta; io.printf("%s\n", meta.VERSION);` .

`go test ./...` skips codegen and uses the default `"dev"` . The meta-lib test only checks that the constant

matches `version.Version` , not a specific value, so it stays robust across builds.

REPL (cmd/jennifer/repl.go)

The REPL drives a read-eval-print loop on top of the standard pipeline. Each input is lexed, preprocessed, parsed, and fed to `Interpreter.EvalInteractive` (not `Run`). `EvalInteractive` differs from `Run` in three documented ways: the global env is lazy-initialized and preserved across calls, library imports and method definitions are idempotent / re-assignable so the user can iterate, and the value of a trailing `ExprStmt` is returned so the loop can print it.

One safety restriction: input that defines methods or structs, or adds `use` / `import` statements, is rejected while a spawned task is still running. Spawn bodies resolve method calls, struct lookups, and namespace prefixes by name from their own goroutines, so mutating those shared tables mid-flight would be a data race (Go's fatal "concurrent map read and map write"). Observe the task first - `task.wait($t)` or `task.discard($t)` - or let it finish; plain statements evaluate normally in the meantime (spawn snapshots are isolated from the live global frame).

Both import kinds work at the prompt: `use LIB;` activates a library namespace, and `import "PATH.j";` loads a module (`runRepl` calls `EnableModules` with the current directory as the local-import base and the system module dir as the search path, so `./mod.j` resolves against the cwd and a bare `mod.j` through the search path). `EvalInteractive` calls `loadModuleImports` in REPL mode, which no-ops a re-submitted `import` of the same module under the same alias (a module is run-once / cached) while still rejecting an alias bound to a different module. Because caching is by resolved path, editing a module file and re-importing it in the same session serves the cached version - restart the REPL to pick up edits.

Echoing a value

Because `EvalInteractive` returns the trailing `ExprStmt` 's value, you can inspect any variable by typing **its bare reference followed by `;`** - the REPL prints the value's `Display()` form:

```
>>> def x as int init 41;
>>> $x;
41
>>> def doc as json.Value init json.decode("{\"a\":[1,2]}");
>>> $doc;
{"a":[1,2]}
```

This is a REPL-only convenience: `Run` (the batch path) evaluates an expression statement but discards its value, so a bare `$x;` in a `.j` **script prints nothing**. To show a value from a script, format it explicitly - `io.printf("%v\n", $x);` (or `io.sprintf($x)` / `convert.toString($x)`). Opaque values render through their registered displayer, so `$doc;` shows a `json.Value` as its JSON rather than `<json.Value>`.

Multi-line input is handled by a small `inputComplete(tokens)` helper that balances `{` / `(` against `}` / `)` (using the lexer's tokens so string and comment contents are ignored) and requires the input to end in `;` or `}`. Anything else triggers a `...` continuation prompt. Unbalanced *closing* delimiters intentionally fall through to the parser for diagnosis since no amount of additional input would fix them.

REPL input is tagged with the synthetic file label `<repl>`. The cross-file-error snippet loader in `printErrorContext` treats `<repl>` like `<stdin>`: no external file lookup is attempted, and the current input buffer is used as the snippet source. Lex errors discard the buffer (since they cannot become valid by reading more); parse and runtime errors print and the loop continues.

`:quit` / `:exit` / EOF terminate cleanly; `:help` prints a short reminder. Directives are only recognized at a fresh prompt so a literal `:quit` inside a block doesn't short-circuit.

Line editor (cmd/jennifer/lineedit.go, cmd/jennifer/history.go)

When stdin is a terminal the REPL installs raw mode via `golang.org/x/term` and reads lines through a small built-in editor. The editor is a single state machine over a rune buffer plus a cursor index; each keystroke updates the state and triggers a `redraw()`.

Supported input:

Key	Action
Printable rune	Insert at cursor
Backspace, Ctrl+H	Delete char before cursor
Delete (CSI 3~)	Delete char at cursor
Left / Right (CSI D / C)	Move cursor by one char
Home / End (CSI H / F)	Jump to line start / end
Ctrl+A / Ctrl+E	Same as Home / End
Ctrl+Left / Ctrl+Right	Move by word
Alt+B / Alt+F	Same as Ctrl+Left / Ctrl+Right (macOS terminals send these for option-arrow)
Ctrl+W, Ctrl+Backspace	Delete word backward
Ctrl+U	Kill from line start to cursor
Ctrl+K	Kill from cursor to line end
Up / Down	History navigation
Ctrl+C	Cancel current line (fresh prompt)
Ctrl+D on empty buffer	EOF (exits the REPL)
Ctrl+D on non-empty buffer	Forward-delete

Word boundaries use a small punctuation + whitespace ruleset that's predictable for source-code editing without needing a full Unicode word-break implementation. History is an in-memory ring (`replHistory` , 100 entries by default, adjacent duplicates collapsed); on-disk persistence is a future enhancement.

Non-TTY stdin falls back to the original `bufio` line reader. This keeps `echo ... | jennifer repl` and integration tests working unchanged - the editor would do nothing useful on a non-interactive stream anyway.

Raw mode disables the kernel's `OPOST` flag, so `\n` written to stdout no longer auto-translates to `\r\n` . The REPL works around this with a tiny `crlfWriter` wrapper that performs the translation in user space for error/help/result prints. Cooked-mode output (the banner printed before raw mode is entered, and anything after raw mode is restored) goes to `os.Stderr` / `os.Stdout` directly.

The editor only handles single-line editing. Multi-line input via the continuation prompt (`...`) is still driven by the surrounding REPL loop's `inputComplete()` check, so unclosed `{` / `(` accumulate across calls to `editor.readLine` .

Syntax highlighting on commit (cmd/jennifer/highlight.go)

While you type, the line is drawn plain. On Enter, if colour is enabled, the editor redraws the committed line one last time with syntax highlighting (`redrawCommitted` -> `highlightLine`) before the newline, so the source shows coloured just above its output. Highlighting only at commit (not per keystroke) keeps the edit path cheap and sidesteps recolouring half-typed, unlexable input.

`highlightLine` lexes the line and wraps each token's source span in an ANSI SGR colour (keyword, type, string, number, `$var` , comment; other tokens stay default). It slices by each token's 1-based **rune column** rather than its lexeme, so the user's exact spacing is preserved and a processed `TOKEN_STRING` lexeme (quotes stripped, escapes resolved) can't desync the offsets. A span runs to the next token's start, so trailing whitespace inherits the token's colour - invisible for foreground-only codes. The colours are zero-width escapes, so the editor's cursor-column arithmetic is unaffected; the commit redraw skips the cursor-back step because a newline follows immediately. On any lex error (e.g. an unterminated string mid-edit) `highlightLine` returns the input unchanged, so the line always echoes verbatim.

Colour is gated by `colorEnabled()` : stdout must be a TTY and `NO_COLOR` (https://no-color.org) must be

unset. The editor already requires a TTY stdin, so the two together mean colour appears only in a genuine interactive session; piped or redirected output stays plain.

Part of the CLI reference.

Inspection: tokens and ast

`cmd/jennifer/dump.go` and `cmd/jennifer/astjson.go` implement two read-only inspection subcommands over the front of the pipeline: `tokens` stops after the lexer, `ast` after the parser. They make the two intermediate representations the interpreter builds visible, which is as much a teaching aid as a debugging one. Both live only in the default `jennifer` binary (the run-only `jennifer-tiny` build stubs them).

The examples below use this three-line program, `snippet.j` :

```
use io;
def x as int init 41;
io.printf("%d\n", $x + 1);
```

tokens - the lexer's stream

`tokens` runs only the lexer and prints one token per line in column-aligned `LINE:COL TYPE [lexeme]` form - useful for tracing a scanning issue:

```
$ jennifer tokens snippet.j
1:1   USE
1:5   IDENT     "io"
1:7   SEMI
2:1   DEF
2:5   IDENT     "x"
2:7   AS
2:10  INT_TYPE
2:14  INIT
2:19  INT       "41"
2:21  SEMI
3:1   IDENT     "io"
3:3   DOT
3:4   IDENT     "printf"
3:10  LPAREN
3:11  STRING    "%d\n"
3:17  COMMA
3:19  VARREF    "x"
3:22  PLUS
3:24  INT       "1"
3:25  RPAREN
3:26  SEMI
4:1   EOF
```

A few things the stream makes concrete: every token records its source `LINE:COL` ; the type keyword `int` scans to its own `INT_TYPE` , distinct from the `INT` literal `41` ; the `$x` use-site is a single `VARREF "x"` with the sigil already consumed; and the stream always terminates in `EOF` .

ast - the parsed tree as JSON

`ast` runs lex + preproc + parse and writes the AST as two-space-indented JSON. Every node carries `type` , `file` , `line` , `col` , plus its node-specific fields:

```
$ jennifer ast snippet.j
```

```
{
  "type": "Program",
  "line": 1,
  "col": 1,
  "imports": [
    { "type": "ImportStmt", "line": 1, "col": 1, "name": "io" }
  ],
  "moduleImports": [],
  "methods": [],
  "topLevel": [
    {
      "type": "DefineStmt",
      "line": 2, "col": 1,
      "isConst": false,
      "exported": false,
      "varName": "x",
      "varType": "int",
      "init": { "type": "IntLit", "line": 2, "col": 19, "value": 41 }
    },
    {
      "type": "ExprStmt",
      "line": 3, "col": 1,
      "expr": {
        "type": "QualifiedCallExpr",
        "line": 3, "col": 1,
        "prefix": "io",
        "callee": "printf",
        "args": [
          { "type": "StringLit", "line": 3, "col": 11, "value": "%d\n" },
          {
            "type": "BinaryExpr", "line": 3, "col": 19, "op": "+",
            "left":  { "type": "VarExpr", "line": 3, "col": 19, "name": "x" },
            "right": { "type": "IntLit",  "line": 3, "col": 24, "value": 1 }
          }
        ]
      }
    }
  ]
}
```

Each node also carries a `file` field (the resolved absolute source path), elided above for width. Because preproc runs before the parse, any `include` d file is already spliced and `import` statements are resolved, so the tree is exactly what the interpreter walks: `$x + 1` as a `BinaryExpr` over a `VarExpr` and an `IntLit` , and `io.printf(...)` as a single `QualifiedCallExpr` with a `prefix` / `callee` pair.

Implementation

The JSON emitter is hand-rolled in `astjson.go` 's `emitNode` (a switch over every concrete AST type). We avoid `encoding/json` because its reflect-based marshaling is fragile under TinyGo and at odds with the tagged-union `Value` discipline used elsewhere; a switch over ~20 node kinds is small enough to keep readable. Each field-emitter (`emitStringField` , `emitBoolField` , `emitNodeListField` , etc.) writes `"key": value,` and the closing `endObj` trims the trailing comma so the output is valid JSON.

Part of the CLI reference.

Formatter (cmd/jennifer/fmt.go)

`jennifer fmt [-w|-l] <file.j>...` rewrites Jennifer source into the one canonical style defined in ../user-guide/style-guide.md. By default it prints the result to **stdout** ; `-w` (or `--write`) rewrites the named files in place; `-l` (or `--check`) checks without rewriting:

```
jennifer fmt prog.j            # preview the formatted source on stdout
jennifer fmt prog.j > out.j    # or redirect it (pipe to `sponge`, an editor, ...)
jennifer fmt -w prog.j         # rewrite prog.j in place
jennifer fmt -w a.j b.j        # rewrite several named files in place
jennifer fmt --check src/*.j   # list files that are not formatted; exit non-zero
```

`-w` skips a file whose content is already canonical (no needless mtime churn) and preserves each file's mode; it prints `formatted <path>` to stderr for each file it changes. Sending several files to stdout without `-w` is refused as ambiguous.

`-l` / `--check` is the non-mutating **gate** mode: it writes the path of each file whose canonical form differs to **stdout** , one per line, mutates nothing, and sets the exit code to `0` (all formatted) / `1` (some need formatting) / `2` (a read or lex error) - the same `0` / `1` / `2` split `jennifer lint` uses, so a combined `fmt --check` + `lint` CI gate reads uniformly. It is mutually exclusive with `-w` . There is one write flag (`-w` / `--write` , no `-i` synonym) and one check flag (`-l` / `--check`), and no style options: like `gofmt` , the formatter is opinionated by design, so a whole codebase reads the same and diffs stay minimal.

The write is careful about what it touches:

- **Atomic + verified.** The formatted text is written to a temp file in the same directory and `rename` d over the original, so a crash or a full disk can never leave a truncated source. Before that rename, `fmt` re-lexes its own output and refuses to write if the non-trivia token stream (values *and* surface spellings) differs from the input - a runtime guarantee that a format only ever changes whitespace, never a program.
- **Symlinks are followed.** `fmt -w link.j` rewrites the link's target and leaves `link.j` a symlink (it does not replace the link with a regular file).
- **Read-only files are left alone.** A source with no write permission (`chmod -w`) is refused with exit `1`, matching `gofmt` ; `chmod +w` it to format it.
- **Line endings.** The output uses LF; a CRLF file is rewritten with LF endings (Jennifer source is LF).

Selecting files: use the shell

`fmt` formats exactly the files you name; it does **no** globbing or directory walking of its own (a directory argument is a usage error). Selecting files is the shell's job - that is the one obvious way to do it, and it composes with every other command. A filemask works flat, and recursively via `**` :

```
# bash / zsh
jennifer fmt -w src/*.j                # flat: every .j in src/
shopt -s globstar                      # bash: enable ** (zsh has it on)
jennifer fmt -w src/**/*.j             # recursive: every .j under src/
jennifer fmt -w src/**/mikrotik*.j     # recursive filemask

# fish (** is recursive by default, no setup)
jennifer fmt -w src/**.j               # recursive: every .j under src/
jennifer fmt -w src/**/mikrotik*.j     # recursive filemask
```

For anything the glob can't express, pipe from `find` : `find src -name '*.j' | xargs jennifer fmt -w` .

Before / after

The formatter fixes spacing, indentation, brace placement, and statement splitting in one pass, while leaving your intent (parentheses, comments, imports) intact:

```
use io;import "helpers.j" as h;
# greet the world
def   x as int init 21 ;
if($x>0){io.printf("pos\n") ;}else{ io.printf("neg\n");}


def y as int init ($x + 1)*2;   # keep the parens
def z as int init -$x;
for(def i as int init 0;$i<3;$i=$i+1){io.printf("%d\n",$i);}
func add(a as int,b as int){return $a+$b;}
```

becomes:

```
use io;
import "helpers.j" as h;
# greet the world
def x as int init 21;
if ($x > 0) {
    io.printf("pos\n");
} else {
    io.printf("neg\n");
}

def y as int init ($x + 1) * 2; # keep the parens
def z as int init -$x;
for (def i as int init 0; $i < 3; $i = $i + 1) {
    io.printf("%d\n", $i);
}
func add(a as int, b as int) {
    return $a + $b;
}
```

What fmt normalises

Aspect	Canonical form
Statements	one per line, each terminated by ; (use io;import ...; splits onto two lines).
Indentation	4 spaces per block level; a } dedents before it is written, so it lands at the outer level.
Operator spacing	a single space around binary operators ($a + $b, $x > 0); none around a unary - (-$x).
Punctuation	one space after each , and after the ;s in a for header; no space before a ;.
Blocks	{ follows its header with a space (if ($c) {}); the body is indented; } sits on its own line.
else / elseif	cuddle the preceding brace on one line (} else {).
for header	the two ; stay on the header line (for (init; cond; step)), not split across lines.
Calls / params	arguments and parameters get one space after each comma (add(a as int, b as int), f(a, b)).
Strings	re-quoted with double quotes and standard escapes (quoteJenniferString, mirroring the lexer's readString).
Blank lines	a run of blank lines collapses to a single one.

What fmt deliberately preserves

Formatting is layout-only; it never rewrites meaning. Three things are kept exactly as written:

Kept as written	Why

import "file.j"; statements	fmt works on the token stream before preprocessing, so imports are re-emitted, not inlined - the opposite of a splice.
User-written parentheses	($x + 1) * 2 keeps its grouping; an AST-based formatter would erase parens the grammar makes redundant.
Comments (and blank lines)	# and nesting /* */ comments survive as trivia: a leading comment stays on its own line, a trailing one stays on the same line.

How it works

`fmt` is **token-level, not AST-level** - it walks the lexer's token stream rather than the parsed tree. That choice is what makes the two preservation guarantees above possible:

- **import survives.** The preprocessor consumes file imports before the parser sees them; an AST formatter would inline every one. The token walker sees `IMPORT` tokens unchanged and re-emits them.
- **User parens survive.** The AST records grouping only through nesting, so redundant parens vanish. A token walker preserves `LPAREN` / `RPAREN` exactly.

`formatTokens(tokens)` drives a small state machine (`fmtState`): for each token it computes the separator (`writeSeparator` - none, a space, or a newline-plus-indent) and then writes the token's canonical spelling (`writeToken`). The key state fields:

Field	Role
indent	current block depth; bumps on {, drops on } (the closing brace dedents before it is written).
prevIsOperand	answers "is the next - binary or unary?" - flipped by isOperandToken after every emit.
prevIsUnaryMinus	suppresses the right-side space after a - that was ruled unary, so -$x stays tight.
insideForHeader	a small backward scan that lets the two ;s inside for (...; ...; ...) stay on the same line.

Comments and blank lines flow through the same machine: the lexer emits them as trivia tokens, and `emitTrivia` writes them in place without disturbing the surrounding state (leading comments on their own line at the current indent, trailing same-line comments inline, blank-line runs collapsed to one; block comments may nest).

Part of the CLI reference.

Linter (cmd/jennifer/lint.go, internal/lint)

`jennifer lint <file.j>` reports patterns that are compile-legal but stylistically or semantically suspect - the slot between `fmt` (which normalises lexical shape) and the parser (which rejects the outright illegal). The checks live in `internal/lint` ; the subcommand in `cmd/jennifer/lint.go` wraps them with file I/O, config resolution, and output rendering.

The check set is grouped by concern, each check with a stable ID so suppression and configuration stay portable and greppable. The leading digit is the group: **L0nn** source errors (the file doesn't parse, or a directive is malformed), **L1nn** correctness, **L2nn** complexity and style, **L3nn** API lifecycle.

ID	Check	Severity	Flags
L001	lex-error	error	the source could not be tokenized
L002	parse-error	error	the source could not be parsed
L003	preproc-error	error	an include could not be spliced
L004	invalid-directive	error	a malformed or unknown-ID # lint-disable comment
L101	unused-local	warning	a local def binding never read (skips spawn-body declarations)
L102	dead-code-after-terminator	warning	a statement after return/throw/exit/break/continue
L103	empty-catch	warning	a catch block with no body
L104	throw-non-error	warning	a throw whose value isn't statically an Error
L105	constant-condition	warning	if (true), while (true) with no escape, if ($x == $x), ...
L106	unused-import	warning	a use / import whose namespace is never referenced (call, constant, value, or type)
L107	undefined-call	warning	an unqualified foo(...) whose name is not a defined method (the call analogue of L002; namespaced / $f() calls excluded)
L201	method-too-long	info	method body over the statement threshold (default 60)
L202	nesting-too-deep	info	block nesting over the depth threshold (default 4)
L203	line-too-long	info	a source line over the column limit (default 100)
L204	interpolation-slot-too-complex	info	a call or other non-trivial expression inside a {expr} string slot
L301	deprecation	warning	reserved family, empty until an API is deprecated
L302	removed-api	warning	use of a removed API (e.g. use core;)
L303	invalid-requirement-header	warning	a malformed / duplicate / unknown # pragma-jennifer-* directive

The **L0nn source errors are always on and not user-selectable** : they are produced by the pipeline (lex / preprocess / parse) or the suppression pass (`L004`), not by an AST walk, so `--checks` can't enable or exclude them and they carry a nil `run` . `registry` in `lint.go` marks every other check `selectable` ; `selectableIDs()` is what `--checks` resolves against, `KnownIDs()` (all IDs) is what directive/config validation checks against. Adding a check takes the next free number in its group; a new group (say `L4nn` for a portability family) is a new leading digit.

Traversal. The parser exposes no generic visitor, so `internal/lint` carries two: a flat `walker` (`walk.go`) with list/stmt/expr hooks for checks that match node shapes (L102/L103/L106/L107/L201/L202/L105 - L106 also walks declared types, which the node walker does not visit, to catch a namespace used only in a type annotation), and a scope-aware traversal (`scope.go`) mirroring the resolver's frame model for the checks that need binding visibility (L101/L104). Both descend into `SpawnExpr.Body` , which the resolver deliberately skips: a read inside a spawn still marks an outer local used, but a *declaration* inside a spawn is left unreported (the

resolver's spawn carve-out, which the linter inherits). The linter runs on the parsed AST alone - it does not call `parser.Resolve`, so it can lint code that would fail resolution, and it tracks its own bindings and declared types.

Format-honest errors. A lex / preprocess / parse failure is not a stderr bail-out: `lintComputeDiags` turns it into an `L0nn` source finding that renders in whatever `--format` was asked for, so a `--format=json` pipeline always receives valid output saying why the file couldn't be checked. `stripPositionPrefix` peels the `FILE:LINE:COL:` that the pipeline errors embed, since the finding carries those as fields.

Module-overlay splice. Linting a `MODULE_test.j` white-box overlay splices the sibling `MODULE.j` in front of the test file before parsing - the same splice `jennifer test` applies (`overlayBaseFor` / `tokenizeForSplice` / `spliceTokens`) - so the scope analysis `lintComputeDiags` runs (`parser.Resolve`: undefined names, shadowing, enum-match binding and exhaustiveness) sees the module's private consts / structs / enums / methods. Without it the resolver reports false positives the test run never hits: e.g. a `when Variant(bind)` arm over a module-private enum goes unrecognised, so the bound name reads as an undefined variable (`L002`). Findings that land in the spliced base module (or its own includes) are dropped from the overlay's report - they belong to `lint MODULE.j`, so a module's findings are never double-counted. A file that is not a `*_test.j` overlay, or whose sibling module is absent, is linted unchanged.

Severity and exit code. A finding at or above `SeverityFloor` (warning) makes the run exit 1; an info-only run exits 0. Exit 2 is reserved for an invocation failure with no source position - bad flags, unreadable file, or a bad `--checks` / `.jennifer-lint` - which prints to stderr. A source error (`L0nn`, severity error) is a finding, so it exits 1, not 2. Same triaging shape as `gofmt -l` / `shellcheck`.

Suppression. `# lint-disable: L101` (trailing) silences an ID on that line; `# lint-disable-file: L101, L102` silences file-wide. There is no blanket disable-all - a directive names IDs, on the line the finding anchors to (the `func` line for `L201`, the block-introducer line for `L202`). Because the parser strips comments, `applySuppressions` reads directives off the raw `lexer.TokenizeWithFile` stream and correlates them to findings by `file` / `line`. A malformed or unknown-ID directive is **continue-and-report**: it becomes an `L004` finding, suppresses nothing (so the finding it meant to silence still surfaces), and the run keeps going. A doubled marker (`## lint-disable: ...`) is an ordinary comment, not a directive.

Selection and suppression are orthogonal layers, and suppression always wins locally: `--checks` gates which checks *run*, then `applySuppressions` filters the findings they produced. So `--checks=L203` with a `# lint-disable: L203` on one line runs L203 everywhere but silences that one line - suppression can only ever remove findings, never add them, so there is no conflict to resolve.

Configuration. `--checks=IDS` (per run) or a `.jennifer-lint` file at the tree root (per project) select checks with one `IDS` / `!IDS` direction - all includes ("run only these") or all excludes ("run everything except"); mixing is an error. Unknown IDs are always an error; naming an always-on `L0nn` source error in `--checks` is rejected too. Error messages are terse - `unknown check ID "L999"`, no catalog dump; `jennifer lint --help` lists the catalog. `--format=human|json|github` picks the output shape: positioned carets (reusing `printErrorContextTo`), a JSON array of `{id,file,line,col,message,severity}` objects, or GitHub Actions annotations. **Multi-file --format=json aggregates every file's findings into one array** (a stream of per-file `[...]` documents would not be valid JSON); human and github stream per file.

TinyGo. The subcommand is build-tag split: `lint.go` (`!tinygo`) carries the real implementation and is the only importer of `internal/lint`, so the whole AST-walking machinery stays out of the `jennifer-tiny` binary; `devtools_tinygo.go` (`tinygo`) stubs `runLint` to a friendly pointer at the default `jennifer` binary, alongside the other dev-tool stubs, mirroring the `os.run` / `net` pattern.

Part of the CLI reference.

Profiler (cmd/jennifer/profile.go, internal/profile)

`jennifer profile <prog.j>` runs the program with the evaluator instrumented and attributes work back to Jennifer source positions (`file:line:col`) - the gap `go tool pprof` leaves, since it profiles the *interpreter binary* , not the `.j` program inside it. The program's own output is redirected to **stderr** so the profile owns **stdout** cleanly, even in the binary form:

```
jennifer profile prog.j                       # table to stdout, program output
to stderr
jennifer profile --allocs prog.j              # value-semantics (copy) profile
jennifer profile --format=pprof p.j > p.pb.gz # gzipped protobuf for go tool
pprof / speedscope
```

Statement profile

The default profile records, per source position, how many times a statement ran and how long it took - **self** (this statement alone) versus **cumulative** (this statement plus everything it called), highest self-time first:

```
$ jennifer profile examples/profile.j
Jennifer statement profile (wall-clock, self = excluding nested statements)

  HITS        SELF          CUM  POSITION
     1  2.208054ms  16.438373ms  examples/profile.j:76:1
     1   321.116µs  10.776339ms  examples/profile.j:36:5
   200  5.292611ms   5.553693ms  examples/profile.j:38:9
   200  4.755398ms   4.755398ms  examples/profile.j:37:9
     8  1.438971ms   1.438971ms  examples/profile.j:51:30
     ...
```

Column	Meaning
HITS	How many times the statement at this position executed.
SELF	Wall-clock spent in this statement, excluding nested statements it called.
CUM	Cumulative wall-clock: this statement plus everything it called.
POSITION	The file:line:col of the statement. Rows are sorted by SELF descending.

A high `SELF` is a hot line; a high `CUM` with low `SELF` is a line that mostly waits on the work it dispatches.

Below the table the statement profile also reports the **maximum call depth** - the deepest chain of nested method calls the run reached, overall and per deepest call site (top 10):

```
Max call depth (deepest chain of nested method calls): 12
  DEPTH  CALL SITE
     12  examples/profile.j:2:62
      1  examples/profile.j:3:19
```

It shares the interpreter's call-depth counter with the catchable recursion guard (`limits.MaxCallDepth` , build-tag split - 10000 on the default binary, 24 on `jennifer-tiny`), so a program that trips a stack-limit / deep-recursion error can be re-run under `profile` to see exactly which call site drove the depth. The section is silent for a program that made no method call.

Modes and formats

Flag	Effect
(default)	Statement profile: hit counts + self / cumulative time per position.
--allocs	Value-semantics profile instead: where compound values are copied (see below).
--format=table	Human-readable text (default).

--format=pprof	Gzipped protobuf, hand-encoded to keep the zero-dependency stance; go tool pprof and speedscope.app read it.
--format=trace	Chrome-trace JSON of the method-call timeline (open in chrome://tracing / Perfetto).
--sysmoddir D	System module directory for bare imports (defaults to the built-in path).
-I DIR	Add DIR to the module search path (repeatable), same as run.
--vendor DIR	Vendor root for @scope/package deck imports.

Unknown `--format` and the unsupported `--allocs --format=trace` combination (allocation events have no timeline) are rejected at argument parse, not deferred to output.

Programs that use modules

`profile` enables `import` exactly as `run` does, so a program that reaches for a shipped module (`markdown` , `http` , ...) profiles normally. The module's own statements are instrumented too and attributed to their own file positions, so a hot line inside `markdown.j` appears in the table beside the caller's lines. The self / cumulative split is corrected at the module-call boundary: a call into a module counts toward the caller's *cumulative* time but not its *self* time (the module's own lines carry that), so nothing is double-counted. (`test --coverage` reuses the profiler but leaves module instrumentation off, so a coverage report still measures only the entry program.)

After an error that executed nothing (a failed import, a bad path), no profile is written to stdout - the stderr error is the whole result, and in particular `--format=pprof` does not leave a valid-but-empty gzip behind. A run that executed some statements before erroring still emits its partial profile.

Allocation profile (--allocs)

Because Jennifer is value-semantic, copies are where hidden cost hides. `--allocs` reports two copy paths per source position:

```
$ jennifer profile --allocs examples/profile.j
Jennifer allocation profile (value-semantics copies)

Eager copies - a def / assignment / parameter binding that deep-copied a compound
value:
  COUNT  POSITION
    200  examples/profile.j:38:28
    200  examples/profile.j:37:9
     50  examples/profile.j:69:9
     ...
  453 copies across 6 sites

Spawn-frame deep copies - a scope snapshot captured at spawn launch:
  (none)
```

Copy path	What it is
Eager copies	A def / assignment / parameter binding that deep-copies a compound value (Value.Copy()). Where the real allocation cost lives. A fresh list / map / struct literal RHS is already private, so binding it is not counted (no redundant copy).
Spawn-frame copies	The scope snapshot taken when a spawn launches (snapshotForSpawn).

`examples/profile.j` exercises both - read it to see where value semantics turn a store into real allocation.

Reading a parallel profile

`spawn` bodies are profiled too (each on its own goroutine, onto the shared, mutex-guarded collector with a per-goroutine self/cumulative accumulator). Two things to keep in mind:

- **Self time aggregates across goroutines** , so total self time can exceed wall-clock elapsed. Four workers each spending 5s at one position report ~20s of self time there though only ~5s of wall-clock passed - time-at-position summed over all goroutines, like `pprof` 's CPU time exceeding wall time.
- **Blocking counts as self time.** A statement that waits (`task.wait` / `task.waitAll` on in-flight workers, or `time.sleep`) attributes that wall-clock wait to itself, so a `waitAll` line can show large self time with a hit count of one. It is real elapsed time the statement occupied, not computation.

Instrumentation (implementation)

The interpreter carries an optional `Profiler` interface (`internal/interpreter/profiling.go`) and three gate flags; `nil` means no profiling, the only hot-path cost being a nil check. The concrete collector lives in `internal/profile` and is injected only by this subcommand, so no profiling machinery compiles into either binary's run path. Hook points:

- **execStmt** wraps `execStmtRaw` , timing each statement. A `profChild` accumulator splits self from cumulative time (the standard nested-timing subtraction). It lives on each goroutine's **root Environment** (`env.root.profChild`), not the shared `Interpreter` , so parallel `spawn` bodies each accumulate into their own snapshot root instead of racing one field; the collector's maps are mutex-guarded for the same reason.
- **evalCall** times each method-call body for the trace timeline.
- **eagerCopy** records an eager deep copy at each value-storage site (`def` / assignment / parameter binding) when the value is a compound.
- **evalSpawn** times the `snapshotForSpawn` deep copy.

`evalExpr` is deliberately *not* timed: a `time.Now()` around every literal read would swamp the profile with its own overhead.

TinyGo. Build-tag split like the linter: `profile.go` (`!tinygo`) is the only importer of `internal/profile` ; `devtools_tinygo.go` stubs the subcommand in the run-only `jennifer-tiny` binary.

Part of the CLI reference.

Test runner (cmd/jennifer/test.go, internal/lib/testing)

`jennifer test <file.j>` discovers a file's `test*` methods, runs each with optional `setUp` / `tearDown` hooks, and reports pass / fail. It is the `testing` library's assertion vocabulary and runner primitives wired into a subcommand - "primitives in Go, orchestration as subcommand", the same shape as `fmt` / `lint` / `profile` .

A test file

```
use testing;

def state as int init 0;

func setUp() {              # runs before every test*
    $state = 10;
}

func testAddsUp() {
    testing.assertEqual(2 + 3, 5);
    testing.assertEqual($state, 10);
}

func testGreeting() {
    testing.assertContains("hello world", "world");
    testing.assertTrue(len("abc") == 3);
}

func testDeliberateFail() {
    testing.assertEqual(2 + 2, 5);      # oops
}
```

Running it prints one line per test and a summary; a failure names the assertion, its source position, and the mismatch:

```
$ jennifer test mathtest.j
PASS testAddsUp (0 ms)
FAIL testDeliberateFail (0 ms)
     [assertion] mathtest.j:20:5 assertEqual: 4 != 5
PASS testGreeting (0 ms)

2 passed, 1 failed, 3 total
```

Discovery and hooks

- A **test** is any top-level `func` whose name begins with `test` (`testAddsUp` , `testGreeting` , ...). The file's top level runs once first, so `def` s and `use` / `import` are in place before any test.
- **setUp** (if defined) runs before each test; **tearDown** after each. Both are optional, looked up by name, and absent by default - the shape of an xUnit fixture without the class.

Flags

Flag	Effect
--filter=REGEX	Run only tests whose name matches REGEX (e.g. --filter=testGreeting).
--format=text	Human-readable, one line per test (default).
--format=tap	TAP version 14 - CI-friendly, with a YAML diagnostic block per failure.
--format=junit	JUnit XML - for CI dashboards that consume it.

--isolated	Run each test in a fresh interpreter subprocess: clean state per test, coarser reporting (a pass/fail line, no in-process kind / position).
--coverage[=FMT]	Also report statement coverage: which executable positions in the tested file(s) ran. text (default) prints a per-file and total percentage plus the never-executed positions; json emits a machine-readable form. Runs in-process (so it overrides --isolated).

Coverage

`--coverage` reuses the profiler's per-position hit data (no second counting path): the tests run with statement profiling live, and the report intersects the recorded hits with every executable statement position statically walked from the AST. Coverage is scoped to the tested program's files, so an imported module that merely ran does not skew it. A module overlay (`MODULE_test.j`) reports the module file and the test file separately.

```
$ jennifer test --coverage mathlib_test.j
... test report ...

Coverage (statements):
  .../mathlib.j       12/14  (85.7%)
    uncovered: 33:5, 41:9
  .../mathlib_test.j  8/8  (100.0%)
  total: 20/22 (90.9%)
```

`--coverage=json` puts the machine-readable report on **stdout** (files, per-file `covered` / `total` / `percent` / `uncovered` positions, and the grand total); the human test report moves to **stderr** so a tool can parse stdout directly - the same "machine format owns stdout" rule `jennifer profile --format=pprof` uses.

The same run in TAP:

```
$ jennifer test --format=tap mathtest.j
TAP version 14
1..3
ok 1 - testAddsUp
not ok 2 - testDeliberateFail
  ---
  kind: assertion
  message: assertEqual: 4 != 5
  file: mathtest.j
  line: 20
  col: 5
  ...
ok 3 - testGreeting
```

Exit codes

Code	Meaning
0	All tests passed.
1	At least one test failed.
2	Runner error (parse / lex / IO) - no tests ran.

The `0` / `1` / `2` split matches `jennifer lint` , so a CI step can treat "failures" and "the tool broke" differently.

Assertions

Six builtins from the `testing` library (`internal/lib/testing/assertions.go` , built into **both** binaries). On failure each throws the canonical `Error{kind: "assertion"}` positioned at the call site, which the runner catches and classifies exactly like a user `throw` :

Assertion	Passes when
testing.assertEqual(a, b)	a equals b (value and kind).
testing.assertNotEqual(a, b)	a differs from b.
testing.assertTrue(cond)	cond (a bool) is true.
testing.assertFalse(cond)	cond is false.
testing.assertContains(container, x)	x occurs in the string / list container.
testing.assertThrows(name, kind)	calling the zero-arg method name throws an Error of that kind.

`assertThrows` takes the method **name** as a string (Jennifer has no function references - `myTest` is a name, not a value), and the interpreter's `CallByName` invokes it. `testing.runWith(name, args)` / `CallByNameWith` bind arguments for framework dispatchers; the zero-arg `CallByName` is the entry point the runner itself uses.

Flow (implementation)

The subcommand parses, preprocesses, and runs the file (methods hoist, top level executes for setup), then:

- **Discovery** - `Interpreter.MethodNames()` filtered to `test*` , overridable with `--filter` .
- **Per test** - `setUp` if present, run the test through `CallByName` (timing + `ClassifyError` into a record), `tearDown` if present.
- **Report** - `--format=text|tap|junit` via `testinglib.RenderReport` ; the exit-code split above.
- **--isolated** - `os.Executable()` re-invokes the binary as `jennifer test --testing-single METHOD FILE.j` (runs exactly one method, prints a one-line result); the parent records the child's exit code and line.

Enabling interpreter change. A Go builtin can now raise a *catchable* Jennifer error. `evalCall` / `evalQualifiedCall` previously flattened any builtin error into a fresh `runtimeError` (losing a custom `kind`); now `builtinError` passes an `*ErrorSignal` / `*ExitSignal` through unwrapped - so `interpreter.RaiseError(kind, msg, ...)` throws a real Jennifer error - and `BuiltinCtx` carries the call-site `file/line/col` so it anchors at the call. No existing builtin returned a signal, so the change is additive.

TinyGo. The subcommand is build-tag split like the other dev tools: `test.go` (`!tinygo`) has it, `devtools_tinygo.go` stubs it. The assertion vocabulary and runner primitives live in the always-built `testing` library, so a hand-written TinyGo suite can still call `testing.assertEqual` / `testing.run` directly.

Part of the CLI reference.

Testing

Ordered by the pipeline, then libraries, then dev tooling and the CLI.

Package	What it tests
internal/lexer	Token-by-token output for fixed inputs; trivia (comments, blank lines); error cases
internal/preproc	include file splicing and path resolution; circular-include detection; trivia handling
internal/parser	AST shape via Sprint; operator precedence; the resolver's scope / slot pass with shadowing + undefined-variable errors; constant folding; parse error cases
internal/interpreter	Full programs in-memory with stdout captured; value-semantics aliasing (eager deep copies) under -race; CallByName / CallByNameWith dispatch and RaiseError classification
internal/lib/crc	CRC-32 / CRC-64 checksums over bytes; codec-table lookups and aliases
internal/lib/encoding	toText / fromText (hex, base64); charset encode / decode; isAscii and length introspection
internal/lib/fs	Whole-file read / write / append; metadata (stat); directory ops; buffered File handles, against temp dirs
internal/lib/hash	MD5 / SHA-1 / SHA-256 one-shot compute and streaming update / finalize
internal/lib/io	Install registers printf / sprintf; format verbs and the modifier grammar; arity and format errors
internal/lib/lists	push / pop / sort / reverse / slice / concat / range; non-mutating (value) semantics
internal/lib/maps	keys / values / has / delete / merge; insertion order; missing-key errors
internal/lib/meta	meta.VERSION matches version.Version; meta.BUILD matches the compiler tag
internal/lib/net	TCP / UDP loopback round-trips and DNS lookups (build-tag gated; the TinyGo stub returns friendly errors)
internal/lib/os	env / args / flag helpers; external-process run / spawn / wait / poll / kill
internal/lib/regex	RE2 matches / find / findAll / replace / split; positional + named captures; the pattern cache
internal/lib/task	spawn observation - wait / poll / discard / waitAll / waitAny - under -race
internal/lib/testing	Assertion vocabulary (assert*) throwing Error{kind:"assertion"}; run / runWith failure classification into testing.Result; text / TAP / JUnit report rendering
internal/lib/time	Constructors / accessors / arithmetic round-trip; ISO weekday remapping; deterministic via a nowFunc package-var override
internal/lint	The L001-L010 checks; # lint-disable suppression; --checks / .jennifer-lint selection and unknown-ID rejection
internal/profile	Collector aggregation (self / cumulative, hit counts); table, Chrome-trace, and pprof rendering (gzip + string-table checks)
cmd/jennifer	Golden test that runs every examples/*.j and compares stdout to examples/expected/*.txt; REPL inputComplete helper; AST-JSON validity; formatter idempotence + behavior preservation; cross-file error reporting

`internal/lib/convert` , `internal/lib/math` , and `internal/lib/strings` have no dedicated `_test.go` ; they are exercised end to end through the golden `examples/*.j` suite and the interpreter's in-memory program tests. `internal/version` is generated code (`version_gen.go`), verified indirectly through `internal/lib/meta` .

Run everything with `go test ./...` . Concurrency-touching packages (`internal/interpreter` , `internal/lib/task`) should also run under `go test -race ./...` .

File map

Where each piece of the codebase lives, grouped by area. The 6-stage pipeline (lexer -> preproc -> parser -> resolver -> interpreter -> libraries) is described in interpreter.md; this page is the file-level index.

CLI (cmd/jennifer/)

File	Description
main.go	CLI + source-context error formatting; argv forwarding to the user program.
repl.go	Interactive REPL loop (TTY + bufio fallback).
lineedit.go	Raw-mode line editor (cursor keys, word ops).
history.go	In-memory REPL history ring buffer.
dump.go	tokens and ast subcommands.
astjson.go	Hand-rolled AST -> JSON emitter.
fmt.go	Token-level source formatter; comment + blank-line preservation.
lint.go	lint subcommand: pipeline + config + suppression + JSON / human / GitHub rendering.
profile.go	profile subcommand: table / pprof / trace output.
test.go	test subcommand: test* discovery, setUp / tearDown, --isolated subprocess mode, text / TAP / JUnit reports.
devtools_tinygo.go	TinyGo build: run-only stubs for tokens / ast / fmt / lint / profile / test.

CLI tests

File	Description
examples_test.go	Golden-file integration test over top-level examples/*.j (skips files without an expected/).
module_overlay_test.go	Runs every modules/*_test.j overlay under jennifer test.
repl_test.go	REPL inputComplete unit tests.
lineedit_test.go	Line-editor state-machine tests.
dump_test.go	AST-JSON validity and token-name tests.
fmt_test.go	Formatter idempotence + behaviour + comment-preservation tests.
cross_file_error_test.go	Cross-file error reporting tests.
highlight_test.go	Docs-site highlight-def generation test.
smtp_send_test.go / pop_recv_test.go / imap_recv_test.go / mail_xoauth2_test.go	Mail-module integration tests: smtp / pop / imap / XOAUTH2 against in-process fake servers.
redis_test.go / resque_test.go	redis / resque integration tests against an in-process RESP server.
memcache_test.go / session_test.go / ratelimit_test.go	memcache and its reference modules against an in-process memcached fake.
http_test.go / gotify_test.go / rest_test.go / oauth_test.go	http client + gotify / rest / oauth against an in-process net/http server.

Lexer, preprocessor, module resolution (internal/)

File	Description
lexer/token.go	Token type definitions (incl. trivia tokens).

lexer/lexer.go	Scanner with optional file tagging; nested block comments.
lexer/lexer_test.go	Lexer tests.
preproc/preproc.go	File-import preprocessor; trivia stripping.
preproc/preproc_test.go	Preprocessor tests.
module/resolve.go	Module path classification + resolution.
module/sysmoddir.go	System module dir: CLI / env / compile precedence + validation.
module/resolve_test.go	Resolver + sysmoddir tests.

Parser (internal/parser/)

File	Description
ast.go	AST node types + Sprint (incl. namespaced struct types).
parser.go	Recursive-descent parser; namespaced struct literals.
resolver.go	Parse-time scope / slot resolution; undefined-variable + shadowing errors.
fold.go	Parse-time constant folding of literal-only subtrees.
parser_test.go / fold_test.go	Parser and constant-folding tests (incl. "runtime error stays runtime").

Interpreter (internal/interpreter/)

File	Description
value.go	Runtime Value tagged union (incl. struct + namespaced-struct + KindObject tags).
environment.go	Scoped symbol table (name map + slot slice).
interpreter.go	Tree-walking evaluator; namespaced-struct + object registration and resolution.
interpreter_test.go	End-to-end interpreter tests.
interactive_test.go	REPL EvalInteractive tests.
namespace_test.go	Namespaced builtins + constants tests.
namespaced_struct_test.go	Namespaced struct tests (synthetic widgets lib).
structs_test.go	User-defined struct tests.
trycatch_test.go	try / catch / throw tests.
spawn_test.go	spawn / task of T runtime + registry tests.
control_flow_test.go	break / continue / repeat / exit tests.
collections_test.go	Lists, maps, iteration tests.
compound_test.go	Nested list-of-list / map-of-list / chained-index tests.
bytes_test.go	Bytes type + non-decimal literals + bit-op tests.
callbyname_test.go	CallByName / CallByNameWith dispatch + RaiseError classification tests.
value_alias_test.go	Shared-marker COW alias-stress tests.
collection_typing_test.go	Generic collection validated entry-by-entry against the declared element type at def-init + assign.

Standard libraries (internal/lib/)

File	Description

io/iolib.go, io/format.go, io/input.go	io: entrypoint; printf / sprintf + verb-modifier mini-language; readLine / readBytes / readChars / eof.
convert/convert.go	convert: toInt / toFloat / toString / toBool / typeOf / objectType + UTF-8 codecs + codepoint bridges.
math/mathlib.go	math: abs / min / max / sqrt / pow / floor / ceil / round / rand*; PI / E.
strings/stringslib.go	strings: case, predicates, slicing, split / chars / join.
lists/listslib.go	lists: push / pop / first / last / head / tail / reverse / sort / contains / concat / slice / shuffle / range.
maps/mapslib.go	maps: keys / values / has / delete / merge.
os/oslib.go, os/exec.go	os: PLATFORM / ARCH / EOL / DIRSEP / PATHSEP / ARGS + getEnv / hasFlag / flag / isTerminal; external-program execution (run / spawn / wait / poll / kill).
meta/metalib.go	meta: VERSION / BUILD / SYSMODDIR (interpreter self-identity).
time/timelib.go, time/zone.go, time/format.go	time: Time / Duration, Unix + calendar + arithmetic; fixed-offset Zone, UTC / local / inZone; strftime format / parse + ISO round-trip.
hash/hashlib.go	hash: MD5 / SHA-1 / SHA-256 / SHA-384 / SHA-512 one-shot + streaming via a codec table.
crc/crclib.go	crc: CRC-32 / CRC-64 (big-endian) one-shot + streaming.
compress/compresslib.go	compress: pack / unpack (gzip / zlib / deflate) + streaming (compress.Stream); optional level.
archive/archivelib.go	archive: tar / zip / tar.gz pack / unpack over bytes (archive.Entry); no fs dependency.
encoding/encodinglib.go, encoding/codecs.go, encoding/codecs_gen.go, encoding/gen_codecs.go	encoding: introspection + toText / fromText + encode / decode; hand-written ascii / ebcdic + generated ISO-8859-N / Windows-125N tables (codecs_gen.go is generated, do not edit; gen_codecs.go is the go:build ignore codegen).
json/jsonlib.go, json/jsondecode.go	json: RFC 8259 encode / encodePretty + Value emitter; recursive-descent decoder -> opaque json.Value (KindObject), positioned errors.
task/tasklib.go	task: wait / poll / discard / waitAll / waitAny over task of T handles.
fs/fslib.go, fs/handles.go	fs: one-shot read / write / append + metadata + dir ops; buffered fs.File handles.
net/netlib.go, net/netlib_std.go, net/netlib_tinygo.go	net: shared install / dispatch; !tinygo full TCP / TLS / UDP / DNS; tinygo friendly-error stubs.
regex/regexlib.go	regex: matches / find / findAll / replace / split / escape + regex.Match + 128-entry LRU pattern cache.
testing/testinglib.go, testing/assertions.go	testing: run / runWith / results / reset / report (text / TAP / JUnit) + exit interception; the six assert* builtins.
uuid/uuidlib.go	uuid: generate("v4"/"v7") + parse / isValid / version + NIL (RFC 9562).
crypto/cryptolib.go	crypto: crypto-grade random (randBytes / randInt), constant-time hmacEqual, key derivation (hkdf / pbkdf2), AEAD encrypt / decrypt (AES-256-GCM), Ed25519 signKeypair / sign / verify; registers the asymmetric rsa* / ecdsa* surface.

crypto/cryptolib_asym_std.go, crypto/cryptolib_asym_tiny.go	crypto RSA / ECDSA over PEM keys (rsaSign / rsaVerify / ecdsaSign / ecdsaVerify, for JWT RS* / ES*): !tinygo real over crypto/rsa + crypto/ecdsa + crypto/x509; tinygo friendly-error stub (build-tag split like net).
crypto/cryptolib_acme_std.go, crypto/cryptolib_acme_tiny.go	crypto key generation, CSR, and canonical JWK (rsaGenerateKey / ecGenerateKey / jwkPublic / csr, for the acme module): same !tinygo real / tinygo stub split.
intl/intllib.go	intl: locale catalogs - load(lang, catalog), setLocale / locale, tr(key[, params]) with %name% interpolation.
term/termlib.go, term/termlib_std.go, term/termlib_tinygo.go	term: raw mode (makeRaw / restore -> term.State), size, raw byte reads; !tinygo real termios (+ RestoreAll for the CLI signal path), tinygo stubs.
httpd/httpdlib.go, httpd/httpdlib_std.go, httpd/httpdlib_tinygo.go	httpd: HTTP/1.1 server engine over net/http (TLS, graceful shutdown) - pull loop listen / accept / respond + request accessors; tinygo stubs.
xml/xmllib.go, xml/xmldecode.go, xml/xmlaccess.go	xml: decode / encode / encodePretty over an opaque xml.Value element tree (ordered attributes, children, mixed text).
yaml/yamllib.go, yaml/yamldecode.go, yaml/yamlencode.go, yaml/yamlaccess.go, yaml/yamlwrite.go	yaml: YAML 1.2 decode / decodeAll / encode / encodePretty over an opaque yaml.Value (on gopkg.in/yaml.v3); same read / walk / write surface as json.
toml/tomllib.go, toml/tomldecode.go, toml/tomlaccess.go, toml/tomlwrite.go	toml: TOML 1.0 encode / encodePretty / decode, same Value walk / write surface as json, plus asDatetime.
devio/devio.go	Shared plumbing for the handle-based libraries (serial / spi / i2c / gpio, also sql): integer-handle structs + typed positional-argument extraction. Not a Jennifer library itself (no namespace, no Install).
serial/seriallib.go, serial/seriallib_linux.go, serial/seriallib_other.go	serial: termios serial-port I/O - open / openWith -> serial.Port, read / write / flush / close. Linux-only; stubs elsewhere.
spi/spilib.go, spi/spilib_linux.go, spi/spilib_other.go	spi: SPI devices - open -> spi.Device, configure(mode, speedHz), full-duplex transfer, close. Linux-only; stubs elsewhere.
i2c/i2clib.go, i2c/i2clib_linux.go, i2c/i2clib_other.go	i2c: the I2C bus - open(path, addr) -> i2c.Bus, read / write / readReg / writeReg, close. Linux-only; stubs elsewhere.
gpio/gpiolib.go, gpio/gpiolib_linux.go, gpio/gpiolib_other.go	gpio: /dev/gpiochipN GPIO v2 lines - pin-keyed setup / read / write / release + IN / OUT. Linux-only; stubs elsewhere.
sql/sqllib.go, sql/sqllib_std.go, sql/sqllib_tiny.go	sql: MySQL / MariaDB + PostgreSQL client over database/sql (placeholder-only binding, pull cursor + typed accessors, transactions, prepared statements); !tinygo imports the drivers, tinygo stubs so the driver trees never compile there.
*/…_test.go	Each library has a co-located _test.go with its unit tests (canonical vectors, round-trips, boundary errors).

Tooling internals

File	Description
internal/stdlib/stdlib.go	InstallAll: the single registration point for every system library; run / repl / profile / test all call it (the seam for build-time selection).
internal/lint/lint.go	lint checks: Diagnostic + grouped-ID registry (L0nn source / L1nn correctness / L2nn style / L3nn lifecycle) + KnownIDs / selectable / Catalog.
internal/lint/run.go	Check driver + threshold Config + source-error diagnostic builder.
internal/lint/checks.go	The individual check implementations.

internal/lint/scope.go	Scope-aware traversal for binding-visibility checks (L101 unused-local, L104 throw-non-error).
internal/lint/walk.go	Flat AST walker for node-shape checks (L102 / L103 / L105 / L201 / L202).
internal/lint/config.go	--checks / .jennifer-lint selection parsing (selectable IDs only).
internal/lint/suppress.go	# lint-disable directive parsing + application; a malformed / unknown-ID directive becomes an L004 finding.
internal/profile/profile.go	profile collector: per-position hit counts + wall-clock (+ optional alloc counting).
internal/profile/render.go	Table + Chrome-trace renderers.
internal/profile/pprof.go	Gzipped-pprof output (hand-encoded, zero-dependency).

Shared limits

File	Description
internal/limits/nesting_std.go	MaxNestingDepth = 1000 + MaxCallDepth = 10000 (!tinygo): the recursive-descent nesting cap (language parser + json / toml / xml decoders) and the evalCall method-call-depth cap on the default binary.
internal/limits/nesting_tiny.go	MaxNestingDepth = 64 + MaxCallDepth = 48 (tinygo): the same caps lowered to stay below jennifer-tiny's fixed 4 MB stack. See docs/technical/tinygo.md.

Version & build

File	Description
internal/version/version.go	Default Version = "dev".
internal/version/version_gen.go	GENERATED by scripts/gen-version.sh (gitignored).
scripts/version.sh	Computes the version string from git state.
scripts/gen-version.sh	Writes version_gen.go before each build.
scripts/gen-module-docs.j	Jennifer program that renders the per-module API reference (docs/api/) + the docs/SUMMARY.md "Module API Reference" region from each module's doc comments, via the docblock module. Run at publish time by docs.yml before grimoire build --pdf (not committed); test.yml runs it as a smoke-check.
docs/api/	GENERATED by scripts/gen-module-docs.j at docs-build time (gitignored, not committed). One API page per shipped module plus index.md; the committed SUMMARY.md keeps an empty api:start/api:end region the generator fills.
scripts/gen-coverage-docs.sh	Shell generator for the test-coverage report: go test -cover (system libraries) + jennifer test --coverage (modules) into a Markdown page. Run at docs-build time by docs.yml after building the interpreter; excluded from the PDF.
docs/technical/coverage.md	GENERATED by scripts/gen-coverage-docs.sh at docs-build time (gitignored, not committed). Library + module statement-coverage tables; the committed SUMMARY.md keeps an empty coverage:start/coverage:end region the generator fills.
Makefile	build (both binaries) / build-tinygo / build-go / test / clean / version.

Jennifer-coded modules (modules/)

Distributable `.j` modules brought in with `import` . Each `X.j` ships a co-located `X_test.j` white-box overlay (run under `jennifer test`) and a `examples/modules/X_demo.j` demo; a reference doc lives under `docs/modules/` .

Module	Description
ansi.j	Terminal styling as string wrappers (color / style / rgb / strip); TTY-aware.
csv.j	RFC 4180 CSV parse / format (+ any delimiter) + header-keyed records.
semver.j	Strict SemVer 2.0.0 parse / compare / sort / increment.
html.j	Build an HTML element tree and render escaped HTML5.
markdown.j	CommonMark subset -> HTML / ANSI, plus Markdown authoring helpers.
mime.j	Build + parse MIME messages (RFC 5322 / 2045, RFC 2047 encoded-words).
smtp.j / pop.j / imap.j	Mail clients over net: send (SMTP), receive (POP3 / IMAP4rev1).
sasl.j	SASL auth encoders (plain / login / bearer XOAUTH2), pure base64.
idna.j	Internationalized domain names (Punycode toAscii / toUnicode).
redis.j	Redis RESP2 client over net.
resque.j	Resque-wire-compatible background jobs (on redis).
memcache.j	memcached text-protocol client over net.
session.j	Server-side sessions (on memcache + uuid + json).
ratelimit.j	Fixed-window rate limiter (on memcache).
http.j	HTTP/1.1 client over net (https via TLS).
gotify.j	Gotify push notifications (on http).
rest.j	Ergonomic REST layer (on http + json).
oauth.j	OAuth2 client - get-a-token grants (on http + json; feeds sasl).

Examples

Path	Description
examples/*.j	Example programs.
examples/expected/*.txt	Expected stdout per example (no entry == not in the golden suite).
examples/with_import/	Subdirectory demonstrating file imports.
examples/showcase/	Helpers spliced by showcase.j.
examples/modules/*_demo.j	Runnable demo per shipped module (smoke-run in CI; not golden-checked).

TinyGo notes

Jennifer ships as two binaries built from the same source:

- `jennifer` - default, built with the standard Go toolchain. Full host-feature surface: file I/O, `os/exec` , network stack, everything.
- `jennifer-tiny` - constrained variant, built with TinyGo. Smaller binary, embeddable; the stock build ships without `os/exec` or a network stack (a build choice, not a hard TinyGo limit - see TinyGo restrictions below).

`make build` produces both. Use `make build-go` or `make build-tinygo` for just one; both regenerate the version file before compiling.

The language is written to stay TinyGo-clean even though the default binary is standard-Go. The `jennifer-tiny` build sits in CI so any change that breaks TinyGo compatibility surfaces immediately. A few constraints shape the implementation:

- **No reflect-heavy code.** Tagged-union `Value` instead of interfaces with type assertions in hot paths. `task.waitAny` uses `reflect.Select` (verified under TinyGo); other new reflect uses need justification.
- **No text/template.** Not needed yet; would drag in fragile runtime paths under TinyGo.
- **No encoding/json for in-binary serialization.** The reflect-based marshaler is fragile under TinyGo, so the AST JSON emitter is hand-rolled (see CLI > Inspection).
- **Goroutines are allowed** (used for `spawn`), but need `-stack-size=4mb` under TinyGo - see the goroutine-stack note below.
- **No -ldflags "-X package.var=value".** TinyGo 0.41 silently ignores the `-X` directive. Use the codegen path (`scripts/gen-version.sh` -> `internal/version/version_gen.go`) for build-time string injection. See CLI > Version injection.
- **No hard dependencies on a hosted runtime.** `jennifer-tiny` targets embedded systems, minimal containers, and small-footprint scripting hosts where ambient stdin, dynamic linking, and a full hosted runtime are not guaranteed.
- **testing runs under regular go test.** TinyGo's `testing` support is partial; we develop and verify with `go test ./...` .

Verify both builds after non-trivial changes:

```
make build
./jennifer      run examples/hello.j   # default (standard Go); full host features
./jennifer-tiny run examples/hello.j   # constrained (TinyGo); no os/exec, no net
```

TinyGo restrictions

A few standard-library features depend on TinyGo runtime support that isn't there today. Calls into them from `jennifer-tiny` error with a friendly Jennifer-level message pointing the user at the default `jennifer` binary. The default binary always supports the full surface.

Library	Affected names	What happens on jennifer-tiny
os	os.run, os.spawn, os.wait, os.poll, os.kill	Runtime error pointing at the default jennifer binary. The os/exec subprocess surface: unimplemented in TinyGo on host targets, and absent by nature on embedded / WASM. Not the same "recompile" story as net - see the note below.
net	Every entry point (TCP, UDP, DNS)	Runtime error pointing at the default jennifer binary. Our stock jennifer-tiny registers no netdev driver, so net is stubbed. Build-tag split: netlib_tinygo.go returns friendly errors. Not a hard TinyGo limit - see the note below.
httpd	Every entry point (listen, accept, respond, ...)	Runtime error pointing at the default jennifer binary. The HTTP/1.1 server engine is over Go net/http, so it is stubbed for the same reason as net (no netdev driver): httpdlib_tinygo.go returns friendly errors, and a rebuild with a network stack restores it.

term	Every entry point (makeRaw, restore, size, readByte)	Runtime error pointing at the default jennifer binary. Terminal control needs golang.org/x/term (which the tiny build excludes) and a controlling TTY (which a minimal / embedded target may not have). Build-tag split like net: termlib_tinygo.go returns friendly errors.
serial	Every entry point (open, read, write, ...)	Runtime error pointing at the default jennifer binary on Linux. Serial-port termios I/O is a Linux /dev + ioctl feature over golang.org/x/sys/unix, which the tiny build does not carry. Build-tag split linux && !tinygo (real) / else (stub).
spi	Every entry point (open, configure, transfer, close)	Runtime error pointing at the default jennifer binary on Linux. SPI_IOC_MESSAGE ioctl; same build-tag split as serial.
i2c	Every entry point (open, read, write, readReg, ...)	Runtime error pointing at the default jennifer binary on Linux. I2C_SLAVE ioctl; same build-tag split as serial.
gpio	Every entry point (setup, read, write, release, chip)	Runtime error pointing at the default jennifer binary on Linux. The /dev/gpiochipN GPIO v2 line ioctls; same build-tag split as serial. (The sysfs-backed gpio module is the portable default and runs on both binaries.)
sql	Every entry point (open, query, exec, ...)	Runtime error pointing at the default jennifer binary. The MySQL / Postgres drivers are heavyweight dependency trees that TinyGo does not compile, so sqllib_tiny.go stubs the whole surface (they are unreachable on stock jennifer-tiny anyway - no network stack). Build-tag split like net: sqllib_std.go (!tinygo) imports the drivers, sqllib_tiny.go (tinygo) returns friendly errors.
crypto	Only the RSA / ECDSA surface: rsaSign / rsaVerify / ecdsaSign / ecdsaVerify, plus rsaGenerateKey / ecGenerateKey / jwkPublic / csr	Runtime error pointing at the default jennifer binary. These pull in crypto/rsa, crypto/ecdsa, and crypto/x509 (PEM parsing / CSR), which are off the TinyGo build; cryptolib_asym_tiny.go / cryptolib_acme_tiny.go stub them. The rest of crypto - random, hmacEqual, HKDF / PBKDF2, AES-GCM, and Ed25519 sign / verify - runs on both binaries. (So JWT's HS* / EdDSA work on jennifer-tiny but RS* / ES* do not, and acme needs the default binary.)

net on TinyGo is a build choice, not a hard limit

The "no network" state is a property of the **stock jennifer-tiny build** , not of TinyGo itself. TinyGo compiles most of `net.Dial` / `net.Listen` ; what it does not do on a default target is register a **netdev driver** at runtime (the pluggable network device interface its `net` package dials through), and our stock build ships none - so we compile the `tinygo` -tagged `netlib_tinygo.go` stub that returns a friendly error instead of failing cryptically deep in Go's `net` .

Anyone who needs networking on the tiny binary can restore it by **rebuilding with a network stack** : target (or link in) a registered netdev driver - or a net-capable target such as one exposing a host socket layer - and drop the `tinygo` build tag on `net` so the real implementation compiles in. With a network stack present, `net` and **every net-backed module** (`smtp` , `pop` , `imap` , `redis` , `resque` , `memcache` , `session` , `ratelimit` , ...) run on `jennifer-tiny` too. So read "needs the default `jennifer` binary" as "needs a build that includes a network stack" - the stock `jennifer` has one, the stock `jennifer-tiny` does not. (UDP is the one genuinely thinner spot: `net.ListenPacket` is not part of TinyGo's surface today, so a rebuild covers TCP / DNS more readily than UDP.)

os/exec on TinyGo is a platform limit, not a switch

The `os` restriction is narrower than it looks: it is only the **os/exec subprocess surface** - `os.run` / `os.spawn` / `os.wait` / `os.poll` / `os.kill` . Everything else in `os` (env, args, flags, the `PLATFORM` / `ARCH` / `EOL` / `DIRSEP` / `PATHSEP` / `ARGS` values) works fully on both binaries.

Do **not** read the `net` note above as applying here. `net` needs a pluggable *driver* you can supply; `os/exec` needs a whole **host operating system with a process model** - fork/exec, a process table, executables on a filesystem. There is no component to link in. Two cases:

- **Host-OS TinyGo target (Linux / macOS / Windows):** a TinyGo **standard-library maturity gap** - the `os/exec` fork/exec path is not implemented yet. If TinyGo upstream adds it, a host-targeted `jennifer-tiny` could gain `os.run` / `os.spawn` ; that is upstream work, not a rebuild switch on our side.
- **Embedded / bare-metal / WASM / WASI targets** (what `jennifer-tiny` exists for): there is **no process model at all** - nothing to fork, no other programs, no `exec` syscall. So the subprocess surface is *fundamentally inapplicable* , a hard platform limit rather than a missing piece. It stays unavailable there, permanently.

This also fits the deployment target: minimal containers and embedded scripting hosts generally should *not* shell out to external processes (no shell, no other executables), so the restriction aligns with where `jennifer-tiny` runs rather than fighting it. In short: `net` = a driver you can supply and rebuild around; `os/exec` = a host capability that is a TinyGo gap on host targets and simply absent on embedded / WASM.

The constants and the env / argv / flag helpers in `os` (`os.PLATFORM` , `os.ARCH` , `os.EOL` , `os.DIRSEP` , `os.PATHSEP` , `os.ARGS` , `os.getEnv` , `os.hasFlag` , `os.flag`) all work fully on both binaries. Every shipped library except the three stubbed ones above (`net` , `httpd` , `term`) and the `os/exec` slice of `os` has **full TinyGo support on both binaries** - `io` , `convert` , `math` , `strings` , `lists` , `maps` , `meta` , `time` , `hash` , `crc` , `crypto` , `compress` , `archive` , `encoding` , `json` , `toml` , `xml` , `yaml` , `intl` , `task` , `fs` , `regex` , `testing` , and `uuid` . `yaml` is the one carrying a third-party dependency (`gopkg.in/yaml.v3`), verified to build *and* run under TinyGo 0.41; every other library is Go standard library or hand-rolled.

Development subcommands are default-binary only. `jennifer-tiny` is a run-only interpreter: `run` and `repl` execute Jennifer source, but the development subcommands `tokens` , `ast` , `fmt` , `lint` , `profile` , and `test` are build-tag-excluded (`cmd/jennifer/devtools_tinygo.go`) and return a friendly error pointing at the default `jennifer` binary. They pull in lexer-dump, AST-JSON, formatter, and lint machinery that a minimal-footprint embedding has no use for; build the standard-Go `jennifer` binary for development work.

TinyGo goroutine stack . Jennifer's tree-walking evaluator wraps each Jennifer-level call in many Go-stack frames (`execBlock` + `evalCall` + `evalExpr` + ...), so even a modest-depth recursion (fib 23) easily exceeds TinyGo's default goroutine stack of ~8KB and segfaults. The Makefile passes `-stack-size=4mb` to `tinygo build` for `jennifer-tiny` so it can run recursive `spawn` bodies (and the parallel section of `examples/benchmark.j`). The default `jennifer` binary doesn't need this - Go's goroutine stacks grow automatically.

That fixed 4 MB stack also sets a hard ceiling on how deeply the tree-walker can recurse over *nested data* : a deeply-nested source literal, or a deeply-nested `json` / `toml` / `xml` document, drives one recursive descent per level, and the interpreter has no `recover()` , so an overflow is a fatal crash rather than a catchable error. Every recursive-descent parser (the language parser and the three hand-rolled decoders) therefore enforces a shared nesting cap, `internal/limits.MaxNestingDepth` . It is build-tag split: 1000 on the default binary (growable stack), and 64 on `jennifer-tiny` , which sits below the depth where the heaviest shape (a nested map literal) overflows the stack (at the earlier 2 MB it survived 96 and segfaulted near 128; the 4 MB stack roughly doubles that floor, so 64 has even more margin). Exceeding the cap is a positioned parse error / catchable decode error on both binaries. `yaml` keeps its own pre-parse guard (it is backed by a Go dependency, not a hand-rolled descent).

The same 4 MB ceiling limits how deep *Jennifer method calls* can nest at runtime - each call stacks many Go frames in the tree-walker - so the interpreter enforces a second, sibling cap in the same package, `internal/limits.MaxCallDepth` . The counter is a per-call-chain `*int` threaded down the frames (fresh at each goroutine root) and bumped on **every** method entry - both an ordinary call (`evalCall`) and a cross-boundary dispatch (a module call, `meta.call` / `meta.callMain`) - so recursion that bounces through a dispatch boundary accumulates on one chain and is caught too, rather than resetting per hop and segfaulting (which matters most on `jennifer-tiny` 's low cap). It too is build-tag split, but lower than the nesting cap because a call frame is heavier than one nesting step: 10000 on the default binary (a heavy recursive body crashes Go's growable stack near 50k), and 48 on `jennifer-tiny` (whose stack was raised from 2 MB to 4 MB for this, on which a fib-shaped or heavy body segfaults near depth 75, while the deepest recursion a shipped example reaches - `examples/benchmark.j` 's serial `fib(23)` - is depth 24). Exceeding it is a catchable "call

stack too deep" runtime error - the analogue of Python's `RecursionError` - instead of a segfault; each `spawn` body gets its own counter on its goroutine.

TinyGo scheduler . `jennifer-tiny` pins the cooperative single-thread scheduler (`-scheduler=tasks` in the Makefile). `spawn` works fully (semantics, loud-fail, registry), but every goroutine shares one OS thread, so it gives concurrency without multi-core parallelism: **parallel speedups stay close to 1.0** , and `-stack-size=4mb` reliably covers recursive `spawn` bodies. The pin is deliberate - the threads-capable default briefly showed real multi-core speedups (161% CPU) but segfaulted on recursive `spawn` bodies, because `-stack-size` doesn't govern OS-thread stacks. Real multi-core on `jennifer-tiny` is separate future work, not a default flip; the default `jennifer` binary already reaches multi-core speedup via Go's scheduler.

Future library work will grow the restrictions table if further TinyGo runtime gaps surface. Each new gap lands with the same friendly-message pattern.

Binary size

The constrained build is the smaller one, which is the point of `jennifer-tiny` targeting minimal-footprint deployments. Sizes from `make build` on linux/amd64 (Go 1.26.5, TinyGo 0.41.1, unstripped). The absolute numbers move with toolchain version and platform; the ratio is the stable part.

Binary	Size
jennifer	~23.3 MB (23,253,990 bytes)
jennifer-tiny	~10.4 MB (10,385,072 bytes)

`jennifer-tiny` comes in at ~45% of the default binary (just under half the size). Most of that gap is TinyGo's smaller runtime versus the standard Go runtime, plus the network-, `os/exec` -, and database-driver-backed libraries the tiny build stubs out; the run-only trim (excluding the `tokens` / `ast` / `fmt` / `lint` / `profile` / `test` development subcommands) shaves an incremental slice on top. The gap has widened as the default binary grew (the database drivers and crypto surface land only on the standard build), so the ratio drifts with each library the tiny build stubs.

These are unstripped `make build` (dev) sizes. Release builds strip: the Go binary adds `-trimpath -ldflags "-s -w"` (down to ~16.0 MB, a ~31% cut) and the TinyGo binary adds `-no-debug` (down to ~4.5 MB, a ~57% cut). Shipped artifacts are therefore well under the dev numbers above; dev builds keep symbols for debugging.

Benchmarks

Single-binary throughput - `jennifer` (Go) vs `jennifer-tiny` (TinyGo) across the serial and parallel workloads of `examples/benchmark.j` , plus the memory / page-fault trade - lives in its own doc: benchmark.md.

Benchmarks

Single-binary throughput for the two builds - `jennifer` (standard Go, the default) and `jennifer-tiny` (TinyGo, the constrained variant) - measured with `examples/benchmark.j` .

Reference numbers are from `examples/benchmark.j` (version `0.23.0-dev+16.f857e8a`) on an **AMD Ryzen 5 7600X3D** (6 cores, 12 threads; bare metal, desktop idle) - the machine the suite prints in its own header, from `os.NCPU` plus a `/proc/cpuinfo` read done in Jennifer. The numbers are machine-specific; the **ratios and the shape** of the comparison are the stable part, not the absolute milliseconds.

The suite is **versioned** (`--suite` `v1` / `v2` / `all`) so its reference series stays comparable over time. **v1** is the interpreter-throughput workload; **v2** is the numeric-library workload (`stats` / `linalg` / `math` / `ml`), reported in its own section below. Each version's workloads are frozen once they have published reference numbers.

The v1 suite has two sections. The **serial** section is single-threaded by design; the **parallel** section fans out to `PARALLEL_WORKERS = 4` spawn tasks per workload. The interpreter build is the current one - eager-copy value semantics, lexical slot resolution, parse-time constant folding, the advisory map hash index, per-frame allocation elimination (slot-backed bindings), and the `binary` bulk-byte primitives - so append-in-a-loop is amortised O(N) with in-place growth, keyed map access is O(1), a call frame allocates nothing per binding, and a byte scan can drop into a single Go call.

> These are reference-machine figures, recorded from a supplied run, **not** produced in CI - benchmark timings are too machine- and load-sensitive to pin in a test. Regenerate them on the reference machine after a perf-relevant change (`jennifer run examples/benchmark.j --suite all --format json`); do not run the suite on other hardware for documentation numbers.

Suite v1 - jennifer (standard-Go binary, default)

```
=== Jennifer benchmark suite ===
build:    go
version:  0.23.0-dev+16.f857e8a
cpu:      AMD Ryzen 5 7600X3D 6-Core Processor (12 cores)
platform: linux/amd64
host:     bare metal (no VM / container hints)

------------------------------------------------------------------------
Workload                                  base        iters      time_ms
------------------------------------------------------------------------
fib(N) recursive                            23            1           57
primes up to LIMIT                      100000            1        18342
newton sqrt batch                        10000        10000          309
monte carlo pi                          500000       500000          821
list sort/reverse/slice                  10000          500         1075
struct list build+read                   10000        10000           30
string join                              10000        10000           11
map insert+read                          10000        10000           37
byte scan naive (per-byte)              500000           10         6442
byte scan binary.indexOf (Go)           500000           10            0
------------------------------------------------------------------------
total                                                               27124

Parallel comparison (workers = 4, scheduler = go)
------------------------------------------------------------------------
Workload                             serial_ms       par_ms      speedup
------------------------------------------------------------------------
primes up to LIMIT                       18342         6543         2.80
newton sqrt batch                          309           74         4.18
monte carlo pi                             821          232         3.54
fib(N) x workers                           228           63         3.62
------------------------------------------------------------------------
```

Go runs the serial section single-threaded while its concurrent GC works on other cores, then reaches genuine multi-core throughput in the parallel section (four `spawn` workers on separate OS threads).

Suite v1 - jennifer-tiny (TinyGo binary)

```
=== Jennifer benchmark suite ===
build:    tinygo
version:  0.23.0-dev+16.f857e8a
cpu:      AMD Ryzen 5 7600X3D 6-Core Processor (1 cores)
platform: linux/amd64
host:     bare metal (no VM / container hints)

------------------------------------------------------------------------
Workload                                  base        iters      time_ms
------------------------------------------------------------------------
fib(N) recursive                            23            1           53
primes up to LIMIT                      100000            1        15451
newton sqrt batch                        10000        10000          270
monte carlo pi                          500000       500000          818
list sort/reverse/slice                  10000          500         1620
struct list build+read                   10000        10000           33
string join                              10000        10000           18
map insert+read                          10000        10000           45
byte scan naive (per-byte)              500000           10         4286
byte scan binary.indexOf (Go)           500000           10            2
------------------------------------------------------------------------
total                                                               22596

Parallel comparison (workers = 4, scheduler = tinygo)
------------------------------------------------------------------------
Workload                             serial_ms       par_ms      speedup
------------------------------------------------------------------------
primes up to LIMIT                       15451        15261         1.01
newton sqrt batch                          270          283         0.95
monte carlo pi                             818          845         0.97
fib(N) x workers                           212          229         0.93
------------------------------------------------------------------------
```

`os.NCPU` reports `1` here - honest about the cooperative `-scheduler=tasks` build's usable parallelism, not the 12 threads the machine has. Every goroutine runs on one OS thread, so the parallel column hovers at ~1.0 by design, dipping slightly below on rows where spawn setup has no parallel payoff to offset it.

v1 per-workload comparison (serial section)

Ratios are `tiny_ms / go_ms` ; > 1.0 means `jennifer-tiny` is slower, < 1.0 (shown in *italics*) means it is faster.

Workload	tiny (ms)	go (ms)	Ratio	Where the time goes
fib(N) recursive	53	57	0.9x	Call-heavy dispatch; TinyGo's tighter loop edges it. Both effectively free (sub-60 ms) thanks to the per-frame allocation elimination.
primes up to LIMIT	15451	18342	0.8x	The long numeric dispatch loop, the single biggest row. TinyGo leads it by ~2.9 s this run.
newton sqrt batch	270	309	0.9x	Float arithmetic + dispatch; TinyGo ahead.
monte carlo pi	818	821	1.0x	Float arithmetic + RNG calls; a dead heat.
list sort/reverse/slice	1620	1075	1.5x	Allocation-heavy. Go's concurrent GC absorbs the sort/reverse/slice churn better than TinyGo's simpler collector - the one clear Go win in the serial section.

struct list build+read	33	30	1.1x	Append hot loop is O(1); both effectively free (sub-40 ms).
string join	18	11	1.6x	Build-up-a-string pattern is O(1); both free (sub-20 ms).
map insert+read	45	37	1.2x	The advisory map hash index keeps keyed insert+read O(1); both sub-50 ms.
byte scan naive (per-byte)	4286	6442	0.7x	A per-byte tree-walker scan of ~500 KB x 10 - the cost the binary library exists to remove. TinyGo's tighter dispatch wins it by ~2.2 s.
byte scan binary.indexOf (Go)	2	0	-	The same scan through one binary.indexOf call (Go bytes.Index, SIMD). Both effectively 0 ms - see below.
total	22596	27124	0.8x	TinyGo posts the lower serial total (~17%), carried mainly by primes and the naive byte-scan row.

TinyGo takes the serial section clearly this run (22.6 s vs 27.1 s, ~17% lower), where its tighter dispatch loop leads the long numeric workloads - `primes` (-2.9 s), the naive `byte scan` (-2.2 s), plus `newton` and `fib` . Go's win is the allocation-heavy `list` row (+545 ms) and the tiny structural rows (`struct` / `string` / `map` , all sub-50 ms). **Strip the two byte-scan rows and the classic-workload totals are 18308 ms (TinyGo) vs 20682 ms (Go)** - TinyGo still ahead, so this run's serial lead is not just the byte scan. (The absolute milliseconds shift run to run; the durable signal is that the two builds trade wins by workload shape - TinyGo the tight compute loops, Go the allocation churn and the parallel section below.)

The bulk-byte rows

The two `byte scan` rows are the `binary` library's throughput demonstration: the same search for a 6-byte needle at the **end** of a ~500 KB buffer, 10 times (so every scan traverses the whole buffer).

- **Naive, per byte** (a `.j` loop comparing bytes one at a time): 6442 ms (Go) / 4286 ms (TinyGo). This is the per-byte tree-walker cost byte-oriented code used to pay.
- **binary.indexOf** (one call into Go's assembly/SIMD `bytes.Index`): **0 ms** (Go) / 2 ms (TinyGo).

The Go row reads `0` because it is genuinely sub-millisecond - ~500 KB x 10 = ~5 MB scanned at ~20 GB/s finishes in a couple hundred microseconds, and the suite reports whole milliseconds, so it floors to `0` (TinyGo's 2 is the same work, just above the 1 ms rounding line). That is the point: pushing a per-byte loop into one Go call is a ~40,000x speedup on this workload.

v1 parallel section

Speedup is `serial_ms / par_ms` ; > 1.0 means the four-worker version beat serial. Go gets real multi-core speedup; TinyGo's cooperative scheduler stays at ~1.0 by design.

Workload	Go serial (ms)	Go par (ms)	Go speedup	TinyGo serial (ms)	TinyGo par (ms)	TinyGo speedup
primes up to LIMIT	18342	6543	2.80	15451	15261	1.01
newton sqrt batch	309	74	4.18	270	283	0.95
monte carlo pi	821	232	3.54	818	845	0.97
fib(N) x workers	228	63	3.62	212	229	0.93

Go reaches real multi-core speedup (2.8x-4.2x on four workers). `jennifer-tiny` pins the cooperative scheduler, so `spawn` there is concurrency without multi-core throughput: its column sits at ~1.0, a touch under on rows where the spawn setup has no parallel payoff to hide behind. Use the default binary when parallel throughput matters.

The parallel section reverses the end-to-end picture. TinyGo has the lower *serial* total (22.6 s vs 27.1 s), but Go crunches the parallel section in **~6.9 s** (6543 + 74 + 232 + 63) where TinyGo takes **~16.6 s** (no parallelism). On the wall clock the full `--suite all` run finishes in **34.35 s on Go vs 39.43 s on TinyGo** (see below): a

lower single-thread total does not mean a faster end-to-end run once any `spawn` parallelism is in play.

Suite v2 - numeric libraries (stats / linalg / math / ml)

The v2 workloads exercise the numeric stack: the Go-side compute plus the Jennifer <-> Go marshaling of scalars and matrices. Both builds run it serially (no parallel section).

```
=== Jennifer benchmark suite (--suite v2, jennifer) ===
-------------------------------------------------------------------------
Workload                                base        iters      time_ms
-------------------------------------------------------------------------
stats normal cdf                       50000        50000           94
math regGammaP                         50000        50000          108
linalg matmul NxN                         30          300           25
linalg solve NxN                          30          300            7
ml linreg fit+predict                    500          200           25
ml kMeans cluster                        500           40            9
-------------------------------------------------------------------------
total                                                               268
```

```
=== Jennifer benchmark suite (--suite v2, jennifer-tiny) ===
-------------------------------------------------------------------------
Workload                                base        iters      time_ms
-------------------------------------------------------------------------
stats normal cdf                       50000        50000           91
math regGammaP                         50000        50000          116
linalg matmul NxN                         30          300           24
linalg solve NxN                          30          300            4
ml linreg fit+predict                    500          200           28
ml kMeans cluster                        500           40            5
-------------------------------------------------------------------------
total                                                               268
```

The two builds are effectively tied on v2 (268 ms each). The distribution rows (`stats normal cdf` , `math regGammaP`) dominate the total because they are the interpreter-heavy ones - 50 000 tree-walked iterations each, one Go call per iteration. The `linalg` / `ml` rows are cheap despite doing real matrix work: the compute is one Go call over a matrix built once outside the timed loop, so the per-iteration cost is mostly the list marshaling, and the Go-side algorithms (matmul, Gaussian solve, normal-equations fit) are far below the dispatch cost. The takeaway: the numeric libraries carry their weight in Go; the tree-walker overhead is in the per-call loop around them, not the math.

Memory, page faults, and wall clock

GNU `/bin/time` on the full `--suite all` run (v1 + v2) on the reference machine:

Metric	jennifer (Go)	jennifer-tiny (TinyGo)
wall clock (real)	34.35 s	39.43 s
user / system	48.50 s / 0.11 s	39.21 s / 0.03 s
CPU	141%	99%
peak resident (RSS)	~44 MB	~123 MB
minor page faults	~26,800	~22,700
major page faults	0	0

- **Go finishes faster end-to-end** (34.35 s vs 39.43 s) while spending more CPU-seconds doing it (48.5 s user at 141% CPU). The gap between `user` and `real` is the concurrent GC on other cores plus the four `spawn`

workers running truly in parallel during the parallel section. TinyGo's `user ~= real` at 99% CPU confirms single-thread execution: the cooperative scheduler runs every goroutine on one OS thread.

- **TinyGo uses ~2.8x the peak RSS** (~123 MB vs ~44 MB). Its cooperative scheduler reserves each goroutine's full `-stack-size` (4 MB, sized above the catchable call-depth cap) up front, so the four parallel `spawn` workers hold ~16 MB of reserved stack between them. Go grows goroutine stacks on demand from ~8 KB, so its footprint stays small and flat.
- **Minor page faults are comparable** this run (~26.8k Go, ~22.7k TinyGo), Go slightly higher - the allocation churn of the four parallel workers plus the v2 numeric workloads (matrix / model allocation) landing on Go's growable heap. Neither build takes any major faults.

Picking a binary

- **Single-thread compute:** the two builds trade wins by workload shape. TinyGo leads the long tight numeric loops (`primes` , `newton`) and the naive byte scan; Go leads the allocation-heavy `list` row and the small structural rows. Which build posts the lower serial *total* shifts run to run.
- **End-to-end wall clock / any spawn parallelism:** the default **jennifer** , every time - it is the only build with real multi-core throughput (2.8x-4.2x here) and finishes the whole suite faster despite a higher serial total.
- **Footprint:** `jennifer` for a small, on-demand memory profile (~44 MB peak); `jennifer-tiny` trades ~3x the RSS for its smaller *binary* and embeddability, not for a smaller runtime footprint.
- **Byte-oriented work:** reach for the `binary` library (and `net.readAll` / `readN`) on either build - the `binary.indexOf` row shows a per-byte loop collapsing to effectively free.
- **Numeric work:** `stats` / `linalg` / `math` / `ml` cost the same on both builds (v2 is a tie); the per-call dispatch loop dominates, not the Go-side math.

Security model

Jennifer is a **general-purpose programming language** , not a sandbox. This page states what that means for security, so that a real bug is told apart from an intended capability.

The trust boundary: the script is trusted

A Jennifer program has **full access to the host** , by design, exactly like a Python, Ruby, or Node program:

- `fs` reads, writes, and deletes any path the process can reach.
- `net` / `sql` / `http` dial any address or database.
- `os.run` / `os.spawn` execute any program.
- `meta.call` / `meta.callMain` invoke a function by runtime name.

These are language capabilities, not vulnerabilities. There is no trust boundary between a script and the host: *the script is the trusted party.* "A Jennifer program can read `/etc/passwd` " is the same statement as "a Python program can call `open('/etc/passwd')` " - true, intended, and not a defect. Automated security scanners frequently mislabel this surface as "path traversal", "SSRF", or "RCE"; in a trusted-script model it is none of those.

The corollary: **do not run untrusted Jennifer code on a host you care about.** If you need to run code you do not trust, isolate the *whole interpreter* at the OS level (a container, a VM, a seccomp/AppArmor profile, an unprivileged user with a restricted filesystem). A capability-based in-process sandbox - an allow-listed `fs` root, a dial policy, an exec allow-list - is planned as `DRAFT#11` in horizon.md; until it lands, OS-level isolation is the answer.

What is a security bug

A real vulnerability is one that exists **even under the trusted-script model** - where untrusted *data* , not untrusted *code* , causes harm. The clear cases:

- **Protocol injection.** A module that puts a caller-supplied value onto a protocol wire without sanitizing it. A program using the module *correctly* - an email form calling `smtp.send` , a route parameter reaching a cookie - is exploitable through the data. So the network / protocol modules reject control characters at the wire boundary (`smtp` addresses / EHLO name, the `http` request method, the `websocket` handshake URL, `web` cookie `Path` / `Domain`), reject an unsupported JWT `crit` header, JSON-escape hand-built JWS, and cap a received message size.
- **Uncatchable crashes / DoS from untrusted input.** Parsing an attacker-shaped document must fail with a *catchable* error, not a fatal crash. The `json` / `toml` / `xml` decoders and the language parser share a nesting cap; the `http` client caps response-body size; deep Jennifer recursion is bounded by a call-depth cap (`limits.MaxCallDepth`) that raises a catchable "call stack too deep" error instead of overflowing the Go stack; and the lexer caps the token stream a single source file may produce (`limits.MaxTokens`), so a dense "token bomb" (a few MB of tiny tokens that amplifies ~100x into Token structs) is a catchable lex error rather than an out-of-memory kill.
- **Cryptographic weakness.** Predictable "random", a timing side channel, an algorithm-confusion or downgrade bypass. `crypto` uses a crypto-grade random source, constant-time comparison, AEAD-only symmetric encryption, and length-validated keys; `jwt.verify` pins the expected algorithm.

These are treated as bugs and fixed with regression tests.

Guidance for programs that face untrusted input

Even with a trusted script, a program can expose a *trust boundary of its own* - an HTTP handler, a job runner, a form. In that program:

- **Validate and bound untrusted data** before handing it to a module (sizes, formats, allowed values). The modules harden the wire, but a program should not pass arbitrary user input to `os.run` , `fs.*` , `meta.call` , or a `serveFile` path.
- **Do not echo raw interpreter or module error text to untrusted end-users.** Error messages carry source

positions and internal names for debugging; that is intended for the operator, not for a remote user. Catch errors and return a generic message at the trust boundary.
- **Wrap network I/O in try / catch** and set timeouts; a remote peer is untrusted.
- **Keep secrets out of world-readable files.** On-disk tokens should use owner-only permissions.

Reporting

See SECURITY.md for how to report a suspected vulnerability, and what is in and out of scope.

Design decisions

Decisions that ship in the language but look, at first glance, like they conflict with one of Jennifer's seven design stances. Each entry explains why the feature is *not* the kind of thing the stance was written to reject. The negative counterpart is Rejected features: proposals that *were* turned down because they really did clash with a stance.

When in doubt, the stances list in ../user-guide/index.md is authoritative for users; this file is the reasoning record for maintainers.

The $xs[] = item; append form

Stance #1 ("one way per thing") normally rejects sugar that creates a parallel API. `$xs[] = item;` and `$xs = lists.push($xs, item);` do compile to the same operation, so the form looks suspect under that rule. It ships anyway because the three properties below set it apart from the rejected `$i++` / `+=` family - the form is not a parallel API, it's the index-write syntax growing one more legal position.

1. **$xs[] re-uses an existing operator slot; it is not a new operator.** `$xs[i] = item;` already targets a list position via the `[...]` index-write syntax. `$xs[] = item;` extends that same operator to one position the existing syntax didn't cover - "just past the end" - by passing an empty index. No new token is introduced. Compare `$i++` : that proposed a *new* operator (`++`) competing with the canonical `$i = $i + 1;` . The bracket form has no new token to learn, no precedence to memorize, and no parse rule that wouldn't exist anyway.
2. **Index-write semantics, not function-call semantics.** `$xs[i] = item;` mutates the binding's list in place. `$xs[] = item;` extends that in-place behaviour to the append position, where the function-call form (`$xs = lists.push($xs, item);`) needs an explicit reassignment to commit the new list back into the binding. So the bracket form isn't a "shortcut for `lists.push` " so much as the index-write syntax growing one more legal position. The two forms have genuinely different shapes: one is a write statement that mutates a binding, the other is an expression that returns a new list.
3. **Write-only; no expression-context footgun.** `$xs[]` cannot appear on the right-hand side of any expression - reading "the element just past the end" has no meaning and is rejected at parse time. `$i++` 's real problem was that pre/post forms differ only in expression context, which is where the bugs hid. `$xs[]` has no expression context to hide in, so the analogous footgun cannot exist.

What this means for `lists.push` : it stays in the language and is canonical for any context that needs the post-append list as an expression value (passing it into another call, chaining transformations). The two spellings are not parallel APIs that do the same thing in the same context; they fit different syntactic positions - the bracket form for the in-place write statement, the function form for the expression value. That's also why the same argument doesn't license a `bytes.push` removal once `$b[] = byte;` ships: any future code that needs "a new bytes value with this byte appended" as an expression still wants the function form.

XOR (^) as its own operator

Stance #1 ("one way per thing") would normally argue against shipping an operator that's algebraically derivable from operators we already have - XOR is `(a | b) & ~(a & b)` in terms of the other bitwise primitives. It ships anyway because XOR is a CPU primitive with unique algebraic properties that show up at every use site:

- **Self-inverse** : `$a ^ $a == 0` .
- **Round-trip** : `($a ^ $b) ^ $b == $a` - the canonical reversible transform (cheap obfuscation, parity bits, the classic in-place swap trick).
- **Bit-toggle** : `$flags ^ $mask` flips exactly the bits set in the mask, leaving the rest alone.

Forcing every XOR use site to write the three-operator composition would be the `a - b == a + (-b)` argument: we still ship `-` because the composed form obscures the intent at every call site. Same logic applies here.

len is a language built-in, not a library

`len(EXPR)` is a reserved keyword and a primary expression in the grammar, not a function in any library.

Stance #2 ("explicit over implicit") would normally argue that every name should be explicitly imported - which is exactly what every library obeys (`use io;` , `use math;` , etc.). The pre-M15.4 design had the `core` library auto-loaded as the one exception to this rule, so that `len` could be called without ceremony. M15.4 chose a different answer: promote `len` to a language built-in so the exception disappears, instead of preserving the auto-loaded library.

The case for keeping `core` auto-loaded (the path we didn't take):

- Minimal language surface area (one stronger reserved word avoided).
- The auto-loaded library was already justified once; doubling down is cheaper than redesigning.
- A future library that wants the same exception ("polymorphic structural primitive every program needs") could be added the same way.

The case for the built-in (what we ship):

- **Stance #2 alignment is now uniform.** Every name a Jennifer program reaches for either lives in the language (operators, keywords, `len`) or behind an explicit `use lib;` . There is no third category. A reader can audit a `.j` file's imports and know every external name in scope.
- **No special-case library machinery.** `RegisterGlobal` , `globalFnsByLib` , the alias-meaningless-for-globals-only-lib rule, the "library 'core' is automatically available" rejection, the "skip core from the available-libs error message" filter - all of that infrastructure existed to support one auto-loaded library. With `len` promoted to a built-in, none of it is required.
- **Future polymorphic primitives have a clear home.** If `len` -like behaviour ever needs a sibling (e.g. a future `empty(v)`), the decision is the same: language built-in or topic library, not "expand the auto-loaded list."
- **No core to keep tightening.** M15.1 moved `JENNIFER_VERSION` out of `core` into `meta` ; the charter discussion ("what qualifies for `core` ?") had already started chipping at the exception. Removing `core` entirely closes the question instead of arguing it indefinitely.

Tradeoffs accepted:

- **Another reserved word.** `len` is now a keyword - users can't define `func len() {}` . The same restriction existed under the old model (the M5-era "shadows builtin" runtime check), just enforced one phase later.
- **Migration churn for any out-of-tree code.** Source that wrote `use core;` errors with a friendly migration hint; sources that defined `func len()` get a parse error pointing at the keyword rename. Pre-1.0 covers both.

`RegisterGlobal` / `RegisterGlobalConst` remain on the interpreter as exported API for compatibility, but no shipping library calls them; the in-tree consumer is gone. A later cleanup pass removes the infrastructure once the M10 collision-rule tests that exercise it migrate.

Half-open ranges

`lists.range(start, end)` is half-open: `lists.range(1, 100)` returns `[1, 2, ..., 99]` (99 elements; 100 excluded). The English-reading stance Jennifer has applied to syntax (`repeat ... until` , word operators, `as` / `init` / `in`) would argue for the closed form - "from 1 to 100" in English includes 100. We deliberately don't extend that stance to value-generating runtime operations.

The English-reading argument applies cleanly to *syntax* , which is read once when learning the language. `lists.range` is a *runtime operation* whose semantics are read at every use site, and the cost of getting it wrong is paid every time the user composes ranges, partitions an iteration, or aligns a range with indexing. The half-open form makes those operations easier; the closed form makes the function name read more naturally in isolation. We pick the operation-friendly form.

The case for half-open:

- **Index alignment.** `lists.range(0, len($xs))` yields exactly the valid 0-based indices for an `len($xs)` -element list. Closed would force `lists.range(0, len($xs) - 1)` - the off-by-one trap the half-open form was invented to eliminate.

- **Composability.** `lists.concat(lists.range(a, b), lists.range(b, c))` is exactly `lists.range(a, c)` . Partitioning a range at any point composes cleanly with no duplication and no `+1` adjustment. Closed would either duplicate `b` or require `lists.range(b + 1, c)` on every partition.
- **Stepping uniformity.** Half-open stepping is always "emit while inside the open end" with no "did the step land?" question. The user never has to reason about whether `end - start` divides evenly by `step` .
- **Consistency with the rest of the stdlib.** `lists.slice` , `strings.substring` , and 0-based indexing are all half-open. A closed `range` would make it the only exception, forcing every user to remember the special case.
- **CS-tradition languages all picked half-open.** Python `range` , Go slice indexing, Rust `..` , C++ STL iterators `[begin, end)` , JavaScript libraries. The "natural-syntax languages picked closed" framing is misleading: Ruby ships *both* (`..` closed, `...` half-open), Swift ships *both* (`...` , `..<`), Kotlin ships *both* (`..` , `until`). When you can ship only one because of stance #1, half-open is the more general choice - closed is recoverable as `lists.range(start, end + 1)` , but the composition and index-alignment properties of half-open are not recoverable from closed.

The English-reading stance still wins for syntax (it costs nothing at runtime), but it's the wrong tie-breaker for a value-generating operation that's about behaviour, not prose. This entry exists to record the call for future tie-breakers: when an operation's semantics matter at every use site, the operation-friendly form beats the prose-friendly form.

We deliberately don't ship a closed variant (`lists.rangeInclusive` or similar) - stance #1 rejects parallel APIs, and the closed form is recoverable as `lists.range(start, end + 1)` when the user wants "count 1 to N inclusive." `lists.range(end)` with a single-arg default-start form is also not shipped (stance #2: explicit over implicit).

The sql library's heavyweight driver dependencies

Jennifer states a dependency-free discipline for the library layer: every standard library is pure-stdlib, the two carve-outs being `gopkg.in/yaml.v3` (a parser too big to hand-roll) and the CLI-scoped `golang.org/x/term` . The `sql` library breaks further: it takes two **real dependency trees** - `go-sql-driver/mysql` and `jackc/pgx` - which looks like a clear violation.

It ships anyway, and the reasoning is deliberate, not slid in:

- **Both drivers are pure-Go.** No cgo, so static builds, cross-compilation, and the best-effort macOS / Windows artifacts stay clean. The one *embedded* engine that would need a multi-MB, TinyGo-hostile dependency - SQLite - is excluded and parked in horizon behind a build tag, precisely to keep this line from being crossed casually.
- **The deciding factor is correctness maturity, not convenience.** MySQL and Postgres are open TCP wire protocols, so a client *is* writable in pure Jennifer (the same shape as `redis` / `imap`). But the mature drivers have absorbed a decade of protocol long-tail - every auth plugin (`caching_sha2_password` / the RSA path, SCRAM-SHA-256), charset handling, NULL semantics, multi-result sets, server-version quirks - that a hand-rolled `.j` client would re-derive one edge case at a time. For databases users depend on daily, that maturity is worth the dependency; the auth crypto becomes the driver's problem, not the language's.
- **It is not a performance concession.** A database client is latency-bound (network round-trip + server execution dominate); client-side decode is the cost center only when streaming 10^5+ rows, a bulk workload Jennifer is the wrong tool for regardless of driver. So this is not "reach for a fast Go library" - it is "reach for a *correct* one".
- **It is TinyGo-clean by construction.** The build-tag split (`sqllib_std.go` imports the drivers, `sqllib_tiny.go` stubs them) means `jennifer-tiny` never compiles the trees; the language stays TinyGo-clean and the interpreter core is untouched.

The precedent (`yaml`) took one pure-Go dependency for a parser too big to hand-roll; `sql` extends the same judgment to a client too intricate to hand-roll *correctly* for engines people trust with production data. The bar stays high: a new heavyweight dependency needs this same "correctness maturity that a hand-roll would re-derive slowly and get subtly wrong" justification, not mere convenience.

Per-goroutine call-depth counter (concurrent web dispatch)

Turning the web framework from serial to concurrent request handling looked like a one-line change: wrap each request in a `spawn` . The naive version worked

- a fast request issued mid-way through a slow handler dropped from ~1700 ms to

~3 ms - but `go test -race` reported a data race on the call-depth counter. The fix that makes concurrent dispatch safe turns on understanding why.

The call-depth guard (the `RecursionError` analogue that raises a catchable error before the Go goroutine stack overflows fatally) counts nested method calls. It originally lived as a plain `int` on the per-goroutine **root env** (`i.global` in serial code, a `spawn` snapshot's globals frame inside a spawn), which is correct for ordinary spawn bodies: each snapshot is a distinct root, so parallel spawns never share the counter.

Handler dispatch breaks that assumption. A handler is dispatched through `meta.callMain` into the **entry program's** interpreter, and `CallMethodWith` re-roots the handler frame at that interpreter's *shared* `host.global` . So the counter a handler's nested calls bump is `host.global` 's single field - and several `spawn` ed workers dispatching handlers into the one shared host all increment it at once. The value-semantics guarantee that makes ordinary spawns race-free ("each spawn deep-copies its scope") does **not** extend across the `meta.callMain` boundary, because the whole point of that boundary is to reach the *shared* entry program, not a copy.

The fix makes the counter a `*int` **threaded down the logical call chain** rather than read off whatever root a frame happens to sit on. `evalCall` stamps the caller frame's counter onto the callee frame (`callFrame.depth = env.depth`) instead of reading `effectiveGlobal` 's shared one, so a chain shares one counter while concurrent chains stay isolated.

The first cut of this minted a *fresh* counter at each cross-interpreter dispatch entry (`CallMethodWith`) - which is race-free, but wrong in a subtler way that an adversarial review caught: it **reset the depth at every boundary crossing** , so recursion that bounces through a dispatch builtin (`meta.call` self-recursion, a handler that re-dispatches itself via `meta.callMain` , a module method that calls back into the host) never accumulated past a single crossing and grew the Go stack unbounded - converting a **catchable** "call stack too deep" error into a **fatal, unrecoverable** Go stack overflow (a whole-process crash, e.g. a web handler taking down the entire server). A fresh counter per crossing is the wrong knob precisely because the milestone's own step-2 note warned "must NOT reset depth in a way that breaks meta.call recursion detection."

So the counter is instead **threaded across the dispatch boundary too** , not reset. `dispatchModuleMethod` passes the consumer's `env.depth` into the module interpreter's `callMethodWithDepth` ; the `meta.call` / `meta.callMain` builtins carry the caller's counter through `BuiltinCtx.Depth` into `CallByNameWithDepth` / `CallHostWithDepth` ; and those dispatch entries increment it (a crossing adds Go frames like any call). Only a genuine root entry with no caller chain - the CLI invoking the entry program, a `testing` harness - mints a fresh counter. The caller's counter is always goroutine-local (a spawn worker's own, or `i.global` 's for the main goroutine), so threading it keeps concurrent chains isolated *and* lets a single logical chain accumulate across every boundary. The result is strictly better than the pre-change reach: recursion that bounces purely through `meta.call` (never guarded before, because those builtins bypassed `evalCall` 's increment) is now catchable too.

Two consequences worth stating. First, the audit that gated the change (run under `-race`): the *other* shared interpreter state a concurrently-dispatched handler can reach is safe - the map hash-index is built only on an indexed *write* (a read never mutates the shared value), the resolver's `CallExpr.Method` / `Fn` caches are stamped single-threaded before any spawn and only read afterwards, the profiler's collector is mutex-guarded (and off the serve path anyway), and `diagReq` is atomic. Second, the one genuinely-unsafe pattern is left to the user and documented, not papered over: a handler that **writes** a shared top-level variable of the entry program races another handler doing the same, because handler workers run against the same host globals (no per-goroutine snapshot across `meta.callMain`). That is the threads-sharing-memory hazard every language has; the guidance is to keep per-request mutable state in the request/response or a synchronized store, not a bare global a handler assigns.

The orm module maps rows as map of string to string, relations via a side Result

The `orm` module is a **Data Mapper** : a row is a `map of string to string` keyed by column name, not a user struct, and `orm.load` returns eager-loaded relations in a separate `Result` holder rather than nesting them into the row map. Both look, at first glance, like they fight stance #4 (rich, honest types) - why hand back stringly-typed maps instead of typed records?

Because the language has **no struct reflection** . A generic mapper cannot populate an arbitrary user struct field-by-field at runtime (there is no way to enumerate a struct's fields or set one by name), so a typed-row API would have to be hand-written per table by the caller - which is exactly what the caller does today by reading `row["name"]` and rebuilding a typed value explicitly. The map is the honest shape for "a row of unknown-at-compile-time columns": the database coerces the string values to column types, and values bind only through placeholders, so the safety story is intact. A typed-struct row form is recorded as rejected (it needs the missing reflection), to revisit if the language ever grows field reflection.

The **side Result** falls out of the same constraint. A `map of string to string` is homogeneous - it cannot hold both scalar columns *and* child-row lists without becoming a heterogeneous `map of string to any` , and Jennifer has no `any` (that is the json.Value opaque-tree decision restated). So eager-loaded associations live in an identity-map-style `Result` read through `orm.related` / `relatedOne` , keeping the row map a clean scalar record. It is the same "no language-level top type; walk heterogeneous data with explicit accessors" stance the `json` / `toml` / `yaml` libraries take.

Scientific-notation float literals

Stance #1 ("one way per thing") normally rejects a second spelling for a value that already has a canonical form, and `1.5e3` / `1500.0` are, on their face, exactly that. Scientific notation ships anyway, for the same three reasons the hex / octal / binary integer literals do (which stance #1 also let through):

1. **A notation that carries domain intent is not a parallel API.** `0o755` says "permission bits", `0xDEAD_BEEF` says "bit pattern", and `6.022e23` says "physical magnitude" - each communicates something the plain-decimal form does not. Stance #1 rejects parallel APIs for the *same job* (`$i++` vs `$i = $i + 1`), not multiple notations that each read as a different kind of quantity. If hex-for-ints passes, exponent-for-floats passes by the identical argument.
2. **For a whole class of values there is no other practical form.** `1e-300` cannot be written with a decimal point (300 zeros), and tiny p-values / physical constants are everyday values once the `math` special functions and the `stats` distribution / inference layer land. So the notation is *enabling* , not merely convenient - the redundant-looking mid-range (`1.5e3`) is the minority case.
3. **It closes a round-trip gap rather than opening one.** The interpreter already *prints* exponent form for extreme magnitudes (`math.pow(10, 21)` displays as `1e+21` , `1e-08` , `...e-301`) but, before this, could not parse that spelling back - a literal it emits was a syntax error on input. Adding the literal makes output readable back as source, which serves the "strict / no surprises at boundaries" stance (#4), not undermines it.

Mechanically it is a lexer-only change (`readNumber` scans an optional `[eE][+-]?digits` suffix; the exponent alone makes the token a `FLOAT`), additive and non-breaking (`1e10` was a juxtaposition parse error before), and strict at the edge: an overflowing exponent (`1e400`) is a positioned parse error via the parser's existing `strconv.ParseFloat` , never an `Infinity` .

The magnitude boundaries are deliberately **asymmetric** , and this is not an oversight in the strictness. Overflow (`1e400`) errors because it produces `Infinity` , a non-finite value stance #4 bans outright (the same reason `math.pow` rejects an infinite result). Underflow (`1e-400` , below the smallest denormal ~ `5e-324`) rounds to `0.0` with no error, because `0.0` is a *finite, correctly-rounded* value - a real number that behaves normally downstream, not the "NaN / silent garbage" the stance is written against. Erroring on it would misapply the stance (a finite zero is not non-finite), diverge from every IEEE-754 language (Python / C / Go / JS all yield `0.0` there, none error), and buy nothing: a program that then divides by the underflowed `0.0` hits the existing

division-by-zero error anyway. So the rule is precisely "reject the non-finite (`Inf` / `NaN`), accept a finite result" - applied consistently, the tiny side stays a value and the huge side is the error.

String interpolation owns {}; the template libraries move to %name%

String interpolation (M24.19) makes an unescaped `{expr}` inside a cooked `"..."` string an evaluation slot. That reserves `{}` for the language, and it *appears* to violate stance #1 (one obvious way) twice - once by adding a second way to build a string beside + concatenation, and once by colliding with the `{name}` placeholder markers `intl` and `validate` used. Both are reasoned, not overlooked.

Why interpolation despite + already existing. Concatenation and interpolation are not the same job. + joins two *strings* and requires an explicit `convert.toString` on every non-string operand; interpolation places a *value beside its label* and stringifies in place. The recurring real-world shape - `"user " + convert.toString($name) + " has " + convert.toString($count) + " items"`

- is exactly what interpolation removes: `"user {$name} has {$count} items"` . It is

sugar (it lowers to concat + `convert.toString`), so it adds no new semantics, and the toolchain still sees each slot as ordinary code (the resolver, `lint` , and `profile` all descend into it). The `f"..."` opt-in prefix was rejected (`rejected.md`): the cooked / raw delimiter split already is the opt-in, so a raw `'...'` string is the "no interpolation" form.

Why the template libraries move off {} to %name%. Two libraries used a `{name}` placeholder for named substitution - `intl.tr` and the `validate` module's `localize` . Once a cooked `"..."` interpolates, a template written as a cooked string (`"Hello, {name}"`) would try to evaluate `{name}` as an expression (and error on the undefined `name`). The fix keeps the language feature primary and moves the two *secondary* libraries to a brace-free **%name%** marker (with %% escaping a literal %), fully decoupled from interpolation and its `\{` escape - a %name% template then reads the same in a cooked or a raw string. A sweep confirmed these were the only two `{name}` -placeholder consumers (JSON / TOML / YAML braces are serialization, `io` 's %a braces are output delimiters). This is a pre-1.0 break to shipped libraries, taken because the language owning `{}` is worth more than a shipped library's marker syntax; see the M24.19 milestone.

markdown.toPdf is folded into markdown, not a separate module

M24.21 added Markdown -> PDF layout. The question was where it lives: a separate `mdpdf` module over `markdown` + `pdf` , or folded into one of them. It is **folded into markdown** as `toPdf` / `toPdfWith` / `renderPdf` , beside the existing `toHtml` / `toAnsi` .

The argument *against* folding is real and was measured. `markdown` gains `import "./pdf.j"` , which pulls in `pdf` plus the `font` TrueType parser, so **every** `markdown` import - even a `toHtml` -only program - now loads ~2,700 extra lines of `.j` . Indicative cost on one machine: import time roughly doubles (~24 ms -> ~48 ms) and peak RSS grows ~6 MB (+37%). It is a one-time, per-process **import** cost, not a per-render one (`toHtml` runs at the same speed either way) - but it is a permanent tax the common case pays for a feature the rare case uses.

Folding into `pdf` instead was rejected outright: it inverts the layering (a low-level PDF writer would embed a markup parser) and would tax every `pdf` user with `markdown` + `html` + `ansi` .

`markdown` was chosen anyway because it is **already a multi-format renderer** - `toHtml` and `toAnsi` set the precedent, and `toPdf` is simply the third output, so one import and one namespace give a caller every rendering path (`markdown.toHtml` / `toAnsi` / `toPdf`). The unified, discoverable surface was judged worth the import tax; a program that needs `markdown` without PDF and cares about the few-ms / few-MB cost can take the single `.j` file and strip the `pdf` import. See the M24.21 milestone.

Rejected features

Proposals that were considered and explicitly turned down. Recorded here so the same ideas don't come back as fresh suggestions next session.

finally clause (try { } finally { })

Considered as the cleanup construct, alongside `defer` , when resource-holding libraries (`fs` , `term` , the planned `sql`) made "release this no matter how the block exits" a real need.

Rejected in favor of `defer` (which shipped) because:

- **Java's footgun.** A `return` or `throw` inside a `finally` silently swallows the try's exception or overrides its return value - a notorious, hard-to-spot bug. `defer CALL(args);` takes a *single call* , not a block, so that hazard is unrepresentable rather than merely discouraged.
- **defer co-locates acquire and release** (`open` then `defer close` on the next line) and scales to N resources by LIFO unwinding, where `finally` nests into pyramids and separates open (top) from close (bottom).
- **It fits the language.** With value semantics and handle-into-registry resources, `defer` 's "capture args now, run the call at block exit" needs no closures (which Jennifer lacks); the model falls out of value semantics.
- **One-obvious-way.** Shipping both `finally` and `defer` would be two constructs for one job. Student familiarity is `finally` 's only real edge, and `defer` is trivially teachable ("cleanup that runs when the block exits").

See M20.7 for the `defer` design.

Reference-capturing (mutable-capture) closures

Distinct from the **value-capturing** closure literal on the roadmap (`DRAFT#28`): what is rejected here is the JS / Python flavor, where a closure captures its enclosing bindings **by live reference** so a mutation on either side is visible to the other. Value-capture (a deep-copy snapshot at literal-evaluation time, the way `spawn` already captures) is accepted; reference-capture is not.

Rejected because:

- **It breaks the value-semantics pillar.** A live captured binding is aliasing by another name - two holders share one mutable backing, which is exactly what "assignment and argument passing copy; no aliasing" rules out everywhere else in the language.
- **It invalidates optimizations documented as safe because there are no such captures.** The `sync.Pool` frame recycling (implementation-note 14), the read-only-parameter borrow analysis (implementation-note 24), and the `spawn` deep-copy capture all rest on the invariant "no value, no library, no AST node can capture an env pointer past its lifetime." Reference-capture reintroduces exactly that pointer, so a pooled / borrowed / snapshotted frame could be mutated through a surviving closure.
- **Value-capture already covers the real need** - inline lambdas and currying - without the aliasing. The only thing lost is live mutable capture, which in a value-semantic language reads as spooky-action-at-a-distance more than a feature.

The **snapshot surprise** value-capture carries (a captured variable is frozen at creation, so a later outer mutation is invisible inside the closure) is the deliberate, teachable trade for keeping value semantics, and is the same model `spawn` already establishes. See `DRAFT#28` for the accepted design.

Increment / decrement (++/--)

Considered: postfix `$i++` and prefix `++$i` .

Rejected because:

- The pre/post distinction is a real footgun - the two forms differ only in expression context, which is exactly where bugs hide. Swift removed `++` / `--` in version 3 (2016) for this exact reason.
- The savings are tiny (three characters) and only apply to `+1` / `-1` .

- Python rejected them from the start and the language hasn't suffered.
- `$i = $i + 1;` is verbose but unambiguous; the readability cost is small.

Compound assignment (+=, -=, *=, /=, //=, %=)

Considered as an alternative to ++ / `--` .

Rejected because:

- Several operators to add and remember for marginal ergonomic gain over `$x = $x + E;` .
- Slippery slope: would we also need a string-concat += ? An `and=` ? Where does the family end?
- Keeping a single assignment shape (`$x = EXPR;`) makes source code uniform and matches Jennifer's "one way to do each thing" stance.

Chained assignment ($x = $y = 0;)

Considered as a shorthand for setting several existing variables to one value, alongside the other assignment sugar above.

Rejected because:

- **Same family, same verdict.** It is one more "assignment sugar" for marginal payoff, so declining it keeps company with ++ / `--` and the += family and preserves the single assignment shape (`$x = EXPR;`).
- **The common case is already covered.** Jennifer variables are born with a value (`def x as int init 0;`), so initializing several to the same value is a declaration concern, not an assignment one. Chaining would only help *re-assigning* several already-declared variables at once - a narrow case that `$x = 0; $y = 0;` handles in one extra line.
- **Assignment is a statement, not an expression.** `$y = 0` has no value, so `$x = $y = 0;` does not parse today; supporting it is a real grammar addition, not free sugar. The two ways to add it both lose: the safe Python-style form (special `lvalue (= lvalue)* = EXPR;` grammar, RHS evaluated once) is pure sugar for the above non-need, and the C-style form (assignment-as-expression) is a bigger semantic change that reopens the `if ($x = 0)` typo-for- == footgun Jennifer avoids by having no assignment-expression at all.

The chosen solution: **initialize at declaration (def ... init) or assign per line ($x = 0; $y = 0;).** (Note: had it existed, value semantics would give `$x = $y = $someList` independent copies, sidestepping Python's shared-alias surprise - a cleaner behavior, but still not enough to justify the sugar.)

Ternary operator (cond ? a : b)

Considered as a way to write expression-position conditional selection without bouncing through an intermediate variable:

```
# verbose - status quo
def grade as string;
if ($score >= 90) { $grade = "A"; } else { $grade = "B"; }

# what a ternary would let us write
def grade as string init $score >= 90 ? "A" : "B";
```

Python's `a if c else b` and the C-family `c ? a : b` cover the same need.

Rejected because:

- **Parallel API to if/else.** Stance #1 is strict in Jennifer (++ was rejected even though it's syntactically distinct from `$i = $i + 1`). "Pick value A or B based on a condition" is one operation; `if` / `else` is the canonical spelling. Adding a second syntax form would be the same kind of parallel API that ++ and += were rejected for.
- **Closest stance neighbor agrees.** Go is the only mainstream language built on a "small, explicit" stance comparable to Jennifer's, and Go's designers rejected ternary explicitly: "The reason `?:` is absent from Go is that the language's designers had seen the operation used too often to create impenetrably complex

expressions. The if-else form, although longer, is unquestionably clearer." Real Go code lives without it; Jennifer programs will too.
- **Nesting is the footgun.** `a ? b : c ? d : e ? f : g` is parseable but unreadable. Once shipped, the ternary will end up in code like this and there's no way to take it back. Rejecting upfront saves us the migration.
- **Verbose form has search/step affordances.** A multi-line `if` / `else` is grep-friendly, debuggable, and editable line by line. The condensed expression form is none of these. For one saved line of source across the whole program, the cost is too high.
- **Escape hatch already exists.** A user who really wants ternary-shaped code can write a one-line helper `func pick(c as bool, a as int, b as int) { if ($c) { return $a; } return $b; }` . Both arguments evaluate eagerly (no short-circuit), but for the cases where ternary is genuinely better this is fine.

The "make `if` itself an expression" alternative (Rust-style: `def x init if (c) { 1 } else { 2 };`) was also considered and deferred indefinitely. It's the cleaner long-term answer if expression-position conditionals ever become genuinely needed - it extends an existing construct rather than introducing a new operator - but it's a much larger change (blocks have to evaluate to their last expression; type-checking gets harder; `if` without `else` and `if` containing `return` / `exit` need defined semantics). We don't owe ourselves that complexity for "save one line of source" ergonomics.

Range literal syntax ([1..9])

Considered as a shorthand for constructing a `list of int` sequence:

```
# what range literal would let us write
def xs as list of int init [1..9];
for (def i in [1..len($items)]) { ... }
```

Borrowed from Haskell / Kotlin / Ruby's syntactic form.

Rejected because:

- **Parallel API to the [1, 2, 3] list literal.** Two syntaxes for "construct a `list of int` " - same family as `++` , `+=` , and ternary. Stance #1 has rejected every previous instance.
- **Hides materialization cost (stance #2).** `[1..big_number]` silently allocates a million-element list. The explicit `for (def i init 1; $i <= n; ...)` loop iterates without materializing, and the cost shows up at the call site instead of being buried in a two-character `..` operator. A library function call (`lists.range(...)`) makes the allocation visible too; the literal form does not.
- **New single-purpose operator.** Jennifer hasn't introduced an operator that works in exactly one context before; every other operator (`+ - * / // % & | ^ ~ << >> < > <= >= == and or not`) works across multiple types or contexts. `..` only does integer ranges. Once it ships we'd have to defend "why doesn't `[1.0..9.0]` work? why not `["a".."z"]` ?" - each extension another design discussion.
- **Pattern set elsewhere.** Jennifer ships `lists.head` / `lists.tail` instead of Python `xs[:n]` / `xs[n:]` slicing syntax; `strings.substring` instead of `s[i:j]` ; index-write `$xs[i] = v;` instead of pythonic `$xs[i:i+1] = [v]` . Library functions over syntax for collection operations is the established Jennifer pattern.

The chosen rule: ship `lists.range(start, end)` (M15.0) as the canonical way to allocate an integer sequence. Use site reads as "this allocates a list" instead of hiding behind two characters of punctuation. See milestones.md > M15 (M15.0).

printf data-transformation modifiers

Considered during the M7 format-verb-modifier design: extending the modifier list with options that transform the *value* rather than its *visual representation* . Examples from the original draft:

- `%s|case=upper|lower|title|snake|kebab|camel|pascal|leet`
- `%s|slice=START:END`
- `%s|md=italic|bold|code|strike|header1|header2|...|link(URL)|...`
- `%s|md=table(...)` and the wider `%a|json=*` / `%a|xml=*` / `%a|yaml=*` family - serialisation modifiers

for the aggregate verb. %a itself shipped in M11 with presentation-only modifiers (sep , kv , open , close , depth , null=skip); the json= / xml= / yaml= family remains rejected as data transformation that belongs in dedicated libraries.
- null=sql (SQL-specific spelling of NULL)

Rejected for the modifier system because:

- **Mission creep.** The guiding rule for printf modifiers is "shape the printed representation, not the value." %d|base=2 is presentation (the int 5 becomes the glyph sequence 101). %s|case=upper is data transformation ("abc" becomes "ABC" - a different string value). Once one transform is in, every string/number/aggregate manipulation becomes a candidate modifier and the format-string spec swallows the rest of the standard library.
- **Parallel API to the libraries.** %s|case=upper is already strings.upper($s) . Two ways to do the same thing breaks Jennifer's "one way per thing" stance and means every future string helper has to decide whether to ship as a function, a modifier, or both.
- **Domain leakage.** null=sql picks one application domain to bake into the formatter. null=literal("NULL") is already general - letting the user spell their own NULL keeps SQL, CSV, JSON, and any future format out of the printf spec.
- **md=* is a separate library.** Markdown rendering belongs in a future markdown library that returns strings, so the result composes with printf , sprintf , string concat, file writing - anywhere a string goes. Folding it into %s modifiers would lock markdown output to print sites.

printf literal-pipe lookahead

Considered during M7 as a way to soften the breaking change to pre-M7 format strings: treat the | after a verb as a literal whenever the *next* byte isn't a lowercase letter, so "%s|%s" would keep working because |% isn't a key start.

Rejected because the rule is context-sensitive in exactly the wrong direction. A user who writes "%s|fill text" (intending literal |) would suddenly hit a parse error because fill is a valid-looking key, while "%s|9 lives" would silently keep working because 9 isn't a letter. The footgun moves around with whatever word follows the verb.

The chosen rule is the strict one: | immediately after a verb always starts a modifier list. To write a literal | in that position, double it (||), parallel to the %% escape for a literal % . The rule is uniform and easy to remember; the migration cost was small (five test strings in this repo).

Verbatim print builtin (io.print / io.println)

Considered: a plain io.print(s) / io.println(s) that writes a string with no format interpretation - the safe primitive for "just emit this string," since a dynamic value passed as io.printf(s) misparses any % it contains (a generated password, user input, or file bytes containing %c reads as an unknown verb, %s as a missing argument).

Rejected because:

- It is a second way to do a job printf already covers. Any string prints with printf("%s", s) , so print(s) is convenience sugar over that - exactly the "no two printf flavors for the same job" case stance 1 (one way per thing) rules out.
- Keeping a single output entry point (printf / sprintf / eprintf) keeps the surface uniform; a reader never wonders which printer a program uses.
- The counter-argument - that a verbatim printer is a *distinct* primitive (no format language at all) rather than a printf flavor - is real but does not clear stance 1's bar: the observable job ("put this string on stdout") is the same, and the language prefers one canonical spelling even when a shorthand would be safer.

The accepted trade-off: printf("%s", s) is the **mandatory** idiom for any dynamic or untrusted string, and printf(s) on such a value is a latent bug. Documented in io.md so the footgun is called out at the source rather than papered over with a second builtin.

Methods on structs

Considered during M15.5 (`time` library) planning: let structs declare methods that receive `self` implicitly, so accessors and small operations on a struct value read as `$t.year()` instead of `time.year($t)` . The trigger was the time library's many calendar accessors; the same shape would later cover `hash.Stream.update` , `os.Process.kill` , and every other library that holds onto state behind a struct.

Rejected because:

- **It's the start of object orientation, not a syntactic shortcut.** Methods carry an implicit `self` binding and re-open the question of polymorphism, dispatch (single? double?), inheritance vs composition, interfaces / traits, visibility modifiers, and constructors. Once any one of those is shipped the others become a "why not also" thread. Jennifer is procedural with value types; the small, explicit shape is the language's identity, not a placeholder for an OO upgrade.
- **It invalidates every shipped library's call shape.** Today `lists.push($xs, x)` , `maps.has($m, k)` , `strings.upper($s)` , `hash.update($s, $b)` , `os.run($argv)` are all `lib.verb(receiver, args...)` . Adding `$receiver.verb(args...)` alongside would force each library to choose - or worse, ship both spellings and create the parallel-API problem stance #1 rejects. Picking method-form everywhere would mean rewriting every library and every example.
- **Stances #1 and #2 already cover the ergonomics complaint.** "One way per thing" - if methods exist the function form becomes the second way. "Explicit over implicit" - the function call shows the library name at the call site; the method form hides it behind dispatch on `$receiver` 's type. The cost of `time.year($t)` vs `$t.year()` is one extra word; the cost of OO is a different language.

The chosen rule: structs have field access only. Operations on struct values live as functions in the owning library (`time.year($t)` , `hash.update($s, $b)` , `os.kill($p)`). If ergonomics ever genuinely justify a receiver syntax, the language change is large enough to merit its own milestone with its own rejected.md trail of what the OO surface would *not* include.

os.exit(n)

Considered during the M11 / M15.1 planning: ship process exit as both the language statement `exit EXPR;` (M11) and as a library function `os.exit(n)` (planned for M15.1). The argument for keeping both was that the language statement might be redefined under a future embedding (a WASM sandbox, a host application driving the interpreter) to mean "return from the interpreter," while `os.exit(n)` would always mean "kill the host process" with no possibility of redefinition. Same argument as C's `exit()` vs `_exit()` .

Rejected because:

- **They collapse to the same thing today.** On a hosted OS the two forms have identical observable behaviour - same exit code, same stdout flush, same termination. Two spellings for one behaviour violates Jennifer's "one way per thing" stance immediately, in exchange for a divergence that hasn't been needed yet.
- **The embedding case is hypothetical.** A WASM-sandbox build and a host-driven embedding are long-horizon items; designing the public API for them now locks in a duplicate that the actual embeddings may not even want (an embedded host might redefine `exit EXPR;` *and* `os.exit(n)` the same way, leaving the distinction useless but still in the language).
- **The statement form is the right home.** Process exit is control-flow, parallel to `return;` : terminating execution from any reachable point. Wrapping the same primitive in a library function would read as "call into the OS" when it's actually "stop running." The statement form keeps the intent visible.

The chosen rule: `exit EXPR;` (and bare `exit;`) is the only spelling. If a future embedding does need a "always kill the host" escape hatch, it ships then with a name that says what it does (`os.kill()` , `os.hardExit()`), not as a near-duplicate of the existing statement.

Implicit use NAME; fallback chain (M8+)

Considered during the M8-and-beyond roadmap discussion: have `use http;` search the system libraries first,

then any installed WASM libraries, then a `http.j` on the file-import path, taking the first one found. Same call-site spelling regardless of where the implementation actually lives.

Rejected because:

- **It violates "explicit over implicit."** Two programs with the same source text would resolve `use http;` to different implementations depending on what's installed in the environment. At the call site (`http.get(...)`) the reader has no way to tell which `http` is in scope.
- **Silent precedence shifts break things.** Installing a WASM `http` package would shadow a (slower / different-flavoured) system `http` . The user's program would change behaviour with no source edit and no visible diff.
- **Debuggability suffers.** "Why does `http.get` behave this way?" becomes "which `http` did the resolver pick this time?" - a question that depends on the environment, not the code.

The chosen rule is explicit prefixes - the load source is visible at the `use` site:

- `use net;` -> system library only.
- `use wasm:libname;` -> WASM library (when that milestone lands).
- `import "path/foo.j";` -> file import (textual splice today; module-aware when M17 lands).

Users who genuinely want one entry point can write a tiny Jennifer-coded shim that picks an implementation explicitly; that's a per-program decision, not a language-wide default.

FFI as a single milestone

Considered: a dedicated "FFI" milestone covering everything users typically mean by foreign function interface - calling C libraries, calling Go libraries, integrating with OS APIs, embedding Jennifer in host applications, reusing existing ecosystems.

Rejected because it's three different problems wearing one name, and lumping them together obscures that two are already addressed and the third doesn't fit:

- **"Call Go libraries" is already the library mechanism.** Every `<pkg>.Install(in)` call registers Go functions as Jennifer builtins. That *is* FFI in everything but name. Anyone wanting to extend Jennifer with Go code writes a topic library today; no new surface needed.
- **"Integrate with OS APIs" is the topic-library job.** `os` , `fs` , `net` , and friends wrap Go's stdlib (which wraps the OS). Adding more OS surface means filling out those libraries, not inventing an FFI keyword.
- **"Reuse existing ecosystems / call C libraries" lands through WASM (M19), not cgo.** TinyGo's cgo support is partial on hosted targets and absent on WASI / baremetal; small-footprint embedding targets typically lack a userspace libc to link against at all. The WASM runtime milestone already plans sandboxed module loading, which sidesteps the ABI / marshalling / ownership fight entirely.
- **"Embed Jennifer in host applications" is a real gap - but it's a Go-side polish job, not a language feature.** It gets its own placeholder in the Long horizon list under "Host-embedding API", separate from FFI as conventionally meant.

The chosen rule: no FFI milestone. Three named homes instead - topic libraries (Go + OS surface), WASM (M19, third-party ecosystems), and a future host-embedding API milestone (driving the interpreter from Go).

References, interior mutability, shared mutable state

Considered as escape hatches from the deep-copy cost that stance #5 (value semantics for collections) imposes on graph algorithms, large data structures, and zero-copy workflows. Three different shapes, considered together because they all relax the same invariant:

- **Mutable references.** A `&xs` form that lets a callee write through to the caller's binding (Go / C++ semantics).
- **Interior mutability.** A wrapper type (Rust's `Cell` / `RefCell`) carrying a mutable cell behind an otherwise-immutable value, settable from anywhere holding the wrapper.
- **Shared mutable state.** A first-class "shared list" or "shared map" kind whose writes are visible to every binding that points at the same underlying storage.

Rejected because:

- **They break the local-reasoning guarantee that value semantics is the whole point of.** Today a reader of `func f(xs as list of int)` knows the caller's binding cannot be mutated by `f` - no aliasing, no surprises. Any of the three above forces every reader to ask "does this callee have a writable handle on my data?" at every call site. The cost is paid by every program, not just the ones that need the optimization.
- **Rust gets away with this only because the borrow checker pays the bill.** Aliased mutation in Rust is sound because the type system rejects programs where two writable handles coexist. Jennifer has no such checker and the language's whole point is local readability without a type-system tax; adding aliased mutation without the checker is just C semantics with prettier syntax.
- **Pre-1.0 softening is a one-way door.** Once any of these ships, the "no aliasing" guarantee is gone for every future reader. Even gating them behind a keyword (`shared` , `&mut`) forces every other Jennifer programmer to learn the aliased-mutation rules to read other people's code.
- **The performance gap is recoverable without semantic change.** Copy-on-write, arena allocation, and read-only slice views (`xs[1..5]` as a non-owning window that errors on assignment) close most of the gap that motivates the proposals. Those are interpreter-internal optimizations - they preserve "no aliasing" at the user level and are recorded as the "Performance & memory" placeholder in the Long horizon list. Graph algorithms specifically can use the M13.1 struct mechanism with explicit ID-keyed maps (`map of int to Node`) for parent / child links, which is the conventional fix in value-typed languages.
- **Shared state for concurrency belongs to M16.0.** The concurrency milestone already plans channel / queue / spawn primitives that handle the cross-task communication case without exposing shared mutable values to single-threaded code.

The chosen rule: stance #5 stands without exception. The "Performance & memory" Long horizon entry covers the optimizations that close the cost gap; users who genuinely need aliased mutation are using the wrong language.

Auto-invoked module setup hook (M17)

Considered while designing the module system (M17). A module could declare a magic `func setup()` (or `init`) that the loader calls automatically the first time the module is imported, giving a defined place for one-time initialization.

Rejected because:

- **Redundant with def const ... init EXPR;.** Value-producing one-time work is already expressible: `export def const TABLE as list of int init buildTable();` where `buildTable` is a private module `func` that runs a loop and returns the result. The initializer runs exactly once at load (M17's run-once semantics), so a separate hook buys nothing for the common case.
- **A pure-side-effect hook reintroduces the footgun M17 removed.** M17 modules are declarations-only with no mutable top-level state, so `setup()` could only do I/O or seed a shared source - exactly the "import does work" surprise the declarations-only rule is meant to eliminate (the lesson behind Python's "imports should be cheap").
- **Cuts against "no required entry point."** Jennifer deliberately has no auto-invoked entry point; an auto-called module hook is the same idea by another name.
- **Stance #1 (one way).** `def const ... init ...;` already runs code once at load; a hook is a second spelling for "run at import."

The chosen rule: a module that genuinely needs imperative setup exports an ordinary function the consumer calls explicitly (`points.prepare();`) - more explicit (stance #2) and it keeps the work off the import path. No new language surface, no auto-invocation.

Directory-as-module and cross-file re-export (M17)

Considered for multi-file modules: let a directory be a module, imported by directory name (`import "bigmod" as bigmod;`) resolving to a conventional entry file, and/or let several files in the module each `export` with the module's public surface being their union (a cross-file re-export / package model, as in Python packages, Go

packages, or ES module re-exports).

Rejected because:

- **include-assembly already covers it with one mechanism.** A multi-file module is a single entry `.j` file that `include`s its parts (subdirectories allowed); the splice shares one module scope and one `export` surface, and the consumer imports just the entry file. That is the M10 textual-splice mechanism plus M17.0's declarations-only-and-export rule, with no new surface. See M17.1.
- **Stance #1 (one way).** Directory-as-module would be a second, parallel way to compose a module on top of include-assembly, for the same outcome.
- **Re-export reopens a closed question.** Cross-file re-export is the same "reach a name from elsewhere and republish it" shape as the `from FOO import BAR` symbol import M17.2 already turned down; adding it here would reintroduce that surface by another name.
- **Relaxing must-end-in-.j for directory imports** removes the lexical marker that keeps import strings unambiguous, for an ergonomic gain include-assembly already delivers.

The chosen rule: multi-file modules assemble via `include` behind one entry file (M17.1). Directory-as-module and cross-file re-export can be revisited as their own milestone if real-world module sizes ever make include-assembly unwieldy.

Mandatory public/private keyword on module declarations (M17.4)

Considered as a stronger form of M17.4's export model: instead of private-by-default with a single `export` marker, require an explicit `public` or `private` keyword on every top-level `def` `const` , `def` `struct` , and `func` in a module (omitting it would be a parse error), on the "explicit over implicit" (stance #2) argument.

Rejected because:

- **Stance #2 is already satisfied by export.** The property that must be explicit is the module's public surface - "what does this expose?" - and `export` answers it by grep (`grep '^export' mod.j` is the whole public API). The only implicit fact left is "unmarked means private," a one-line documented rule, not a hidden surprise.
- **Private-by-default is the fail-safe direction.** A forgotten marker keeps a name internal; there is no path where omission leaks a name into the public surface, which is the failure mode the M17.3/M17.4 supply-chain-hygiene argument cares about.
- **Every cited language uses marker-for-public, default-private.** Rust `pub` , TypeScript `export` , Go capitalization, Java package-private + `public` - none requires a visibility keyword on every declaration. The ecosystem converged on exactly the chosen model.
- **Cheaper script-to-module promotion.** Promoting a script means adding `export` only to the names being committed to a public API (M17.4's deliberate "I now have a public API" moment). Mandatory keywords would force annotating every top-level name on promotion, a noisier diff for no safety gain.
- **Stance #1 (one way).** Mandatory `public` / `private` doubles the visibility vocabulary; `export` -or-nothing is one axis, one token. (This is also why the parallel `public` synonym for `export` and a standalone `private` marker are rejected - see M17.4.)

Implicit map-to-struct coercion at binding boundaries (M16.9 json typed decode)

Considered: letting a `map of string to V` value materialize a struct at a typed binding *implicitly* - `def p as Point init json.decode(s);` silently filling `Point` 's fields from the decoded object, erroring on missing / unknown / wrong-type fields. It was the original M16.9 shape for typed JSON decode. **Only the implicit form is rejected here** - an *explicit* map-to-struct conversion (a spelled-out call or a struct-literal spread, where the reader sees the conversion at the call site) is the sanctioned path, deferred to Long horizon in milestones.md.

Rejected because:

- **Jennifer does no coercion at binding boundaries, anywhere.** The boundary (`MatchesDeclared` , at def-init, assignment, params, field writes) is strict: even `def x as float init 5;` errors - you write `5.0` . A

map-to-struct coercion would be the first exception, and a cross-cutting one (it fires wherever a struct type meets a map value), softening a stance the rest of the language holds uniformly.
- **json.decode can't see the caller's declared type** (builtins don't receive it), so the coercion could only live in the interpreter's binding path - it is a language feature, not a library detail, and would apply to every map, not just decoded ones.
- **The explicit rebuild is short and self-documenting.** `def p as Point init Point{ x: $m["x"], y: $m["y"] };` names the schema at the call site, matching Jennifer's write-it-out style; the encode direction (`json.encode($p)`) is unaffected.

So `json.decode` returns generic Values (objects to `map of string to V`); typed targets are rebuilt by hand, or - once specced - through the *explicit* map-to-struct conversion (deferred; Long horizon). See M16.9 in milestones.md and libraries/json.md.

A language any / top type (M16.16 json.Value)

Considered: a general `any` type keyword - a top type every value matches - as the home for heterogeneous data, its concrete driver being mixed-shape JSON (`def x as any;` , `list of any` , `map of string to any` , with a runtime-checked extraction back into concrete slots). **Rejected in favour of confining the dynamism to an opaque, destructure-only json.Value tree** (M16.16).

Rejected because:

- **It is a language-wide type-system opt-out.** An `any` value is usable at every expression site - operators dispatch on the runtime kind, so `def x as any init 5; def y as any init 3; $x + $y;` just works. A beginner can declare everything `any` and sail past the type system entirely; the strict, teachable identity ("you always declare what you store") is gutted to serve one library's need.
- **The friction fixes are worse than the disease.** Making `any` opaque (rejecting it at every operator so each use demands an explicit narrow) claws the strictness back, but only by bolting a whole second set of narrowing rules onto the type system - heavy machinery for what is really a per-source problem.
- **The dynamism is per-source, not universal.** Heterogeneous data comes from specific boundaries (a decoded JSON document; later, a DB row). Each earns its own labelled, opaque, destructure-only type - `json.Value` , a future `sql.Row` - so the escape hatch stays visible and local, never a shared language feature.

So there is no `any` keyword; heterogeneous JSON lives in `json.Value` , walked with explicit accessors. See M16.16 in milestones.md.

printf i18n / string translation in format strings (M20.4 intl)

The proposal: since stance 3 makes `printf` about *presentation* , and translating `"hello"` to `"hallo"` presents the same meaning in another form, extend `(s)printf` to look up translations - a format-string-level translation layer.

Rejected because it fails the stance's own test - "shape how a value is **rendered** " (kept) vs "transform the value itself" (a library call, like `upper` / markdown rendering):

- **It is substitution, not rendering.** `%d | base=2` renders 255 as `11111111` - the same value, a pure function of value + modifier, no state. Translation *replaces* the string with a different one fetched from a catalog keyed by the current locale; the output has no mechanical relation to the input. That is transformation, sitting next to `upper` and markdown rendering, which the stance already makes library calls.
- **It smuggles in global state.** `printf` modifiers are pure; translation needs an ambient locale + loaded catalogs, violating stance 2 (explicit) and stance 7 (namespaced, no globals), and it couples `printf` to `intl` , killing the orthogonality stance 3 exists to protect.
- **Even gettext keeps it out of printf.** The canonical `_()` design translates the *format string* first, then formats the result: `io.printf(intl.tr("Hello, %s"), name)` . Translation and formatting already compose cleanly, so there is no gap for a `printf` extension to fill.

So translation stays `intl.tr()` (M20.4), composing with `printf` as above. Locale-aware *value* formatting (a

number with the locale's grouping / decimal, a date in the locale's order) genuinely is presentation and is *not* rejected - it is a separate, open `printf` question for later.

Write-through copy-on-write for compound Values (M19.2)

Value semantics rest on eager deep copies at every store site (`def` / assignment / parameter binding / spawn snapshot), so no two live bindings ever share a compound backing and the mutation sites can grow a binding's own backing in place - append-in-a-loop stays amortised O(N). An earlier attempt added a copy-on-write layer on top: a `Value.shared *bool` marker set by `Share()` on every `VarExpr` read and honoured by `Ensure()` at each mutation site, meaning to defer the deep copy until an actually-aliased value is mutated. It shipped **inert** - `Share()` has a value receiver and `Environment.Get` / `GetAt` return the binding's `Value` by value, so the flag was set on a throwaway copy and never reached the stored binding; `Ensure` never detached. Correctness never depended on it, and every compound read heap-allocated a `*bool` that went nowhere. It was removed.

The **write-through** version that would make COW actually fire - store `*Binding` (or otherwise let the marker propagate back to the stored value) so a mutation site genuinely detaches an aliased backing - was considered and rejected:

- **The eager-copy invariant already makes aliasing rare.** Because every store copies, the only values that are ever aliased are transient rvalues that get copied again at the next store. A binding almost never holds a value that another live binding also holds, so the deferred-copy path COW optimises is nearly never taken - it would add machinery for a case that eager copies have already designed away.
- **It complicates the hot path it claims to help.** Propagating the marker means threading `*Binding` through `Environment` reads and every element / field slot, turning simple by-value reads into pointer bookkeeping and reintroducing shared mutable state the interpreter is otherwise free of.
- **The measured win is on append-in-a-loop, which already works.** In-place growth of a binding's own backing (safe precisely because nothing else aliases it) gives the O(N) append without any marker at all.

So Jennifer keeps eager copies plus one narrow optimisation - a fresh list / map / struct literal RHS is already private, so the binding site skips the redundant whole-value copy (`rhsFreshLiteral`). If profiling ever shows real aliasing-heavy workloads paying for redundant copies, refcounted COW can be revisited then.

Sandboxing the stdlib, removing TLS skipVerify, and SecureString (security review)

A bot-generated security audit flagged the standard library's host access - `fs` reading / writing any path, `net.connect` dialing any address, `sql.open` to any DSN, `os.run` executing programs, `meta.call` invoking a function by name

- as CRITICAL / HIGH "path traversal / SSRF / RCE / arbitrary-dispatch"

vulnerabilities, and proposed removing or gating them (a `SetRoot` fs jail, a dial policy, an exec allow-list, removing the TLS `skipVerify` option, a zeroable `SecureString`).

These are **not vulnerabilities** ; they are the intended surface of a general-purpose language, exactly as Python's `open` / `socket` / `subprocess` / `getattr` are. There is no trust boundary being crossed: the script *is* the trusted party. A script's ability to read a file is a capability, not a defect. Correspondingly:

- **No fs jail / dial policy / exec allow-list by default.** Restricting the stdlib to sandbox *untrusted* scripts is a real, separate goal - it is the planned capability sandbox, parked as `horizon.md` DRAFT#11 - not a fix for a bug in the trusted-script model. The genuine module-level injection / DoS bugs the same review found (where untrusted *data* , not untrusted *code* , reaches a wire) are fixed in M21.7; the recursion-crash robustness gap in M21.8.
- **The TLS skipVerify option stays.** Certificate verification is **on by default** ; `net.TLSOptions.skipVerify` is an explicit, documented opt-out for self-signed / development endpoints (with `caCert` PEM-pinning as the narrower alternative) - the same role as Go's `InsecureSkipVerify` , `curl -k` , or Python's `verify=False` . Removing it would break legitimate use and is not how any mainstream TLS client behaves.

- **No SecureString zeroable-secret type.** Jennifer strings are immutable and garbage-collected, so an in-place wipe is not expressible, and no mainstream language zeroes secret strings in the interpreter. The pragmatic protection (owner-only file permissions for on-disk secrets) is M21.9, not a new type.

The review's one sound architectural point - that the security model should be *stated* - is adopted: M21.7 adds a `SECURITY.md` / security-model doc making the "scripts have full host access; sandbox untrusted ones" contract explicit.

A modsupport / modulecompanion catch-all library (M21.10)

Considered while planning M21.10 (byte-oriented throughput): a single Go system library - working names `modsupport` / `modulecompanion` - that would house the performance-critical inner routines pulled out of various `.j` modules (mime boundary scanning, http body accumulation, redis / imap framing, ...), even the module-specific ones, so a slow `.j` hot loop could be replaced by one Go call in a shared home.

Rejected because:

- **It is organized by the wrong axis.** The stdlib is topic-based (I/O to `io` / `fs` / `net` , byte / number / string transforms to `binary` / `math` / `strings` , ...). A "perf helpers for modules" library is organized by *who calls it* and *why* , not by topic - the classic `util` / `common` grab-bag that every mature codebase regrets, accreting unrelated functions with no coherent mental model.
- **The premise is mostly false.** The slow routines are rarely module-specific: mime scanning, http accumulation, and redis / imap framing all reduce to a small set of *general* byte primitives (`readAll` / `concat` / `slice` / `find`) that belong in a topic library (`binary`) and serve every module plus user code. That is M21.10.2, and it is the correct granularity.
- **It leaks module internals into the global public surface.** A `modsupport.mimeScanBoundary` builtin would appear in the cheatsheet and be callable by any program, yet mean anything only inside mime - an implementation detail published as API.
- **It unions dependencies and TinyGo build cost.** One library housing many modules' helpers pulls in the union of their Go dependencies; a program using only mime would compile redis's helpers too, unless the grab-bag is itself build-tag-split (complexity stacked on complexity). Topic libraries keep dependencies localized.
- **It invites speculative accretion.** "This might be slow, put it in `modsupport` " is exactly the unmeasured optimization the M21.10.1 benchmark exists to prevent.

The chosen rule: a hot `.j` routine that needs Go goes to a **topic library** as a general primitive (`binary` for `concat` / `slice` / `find` / `readAll` , `encoding` for codecs); a module whose *core* is inherently Go work is **promoted to a Go library** (the `json` / `toml` / `xml` / `yaml` precedent - "a char parser belongs in Go"); a genuinely module-specific, irreducible, *measured* hot path folds into the nearest topic library or ships as a build-tag-split helper co-located with that concern - never a global grab-bag. Only routines that measure hot in M21.10.1 move.

Multiple returns / destructuring

Considered as an ergonomic shorthand (it lived in the horizon idea collection): let a method return several values (`return $a, $b;`) and a `def` / assignment bind them positionally (`def q as int, r as int init divmod($a, $b);` , `$x, $y = $y, $x;`), optionally extended to struct / list destructuring.

Rejected in favor of **structs** - which the language already has - because:

- **Structs already cover it, and read better.** A `def struct` with named fields is the sanctioned way to return several values, and the names are self-documenting (`$d.quotient` / `$d.remainder` , not "the second one"). A struct return also stays a single typed `Value` , so it composes everywhere - stored, passed, nested, printed - whereas a multi-return only works at the bind site. Adding multi-return would be a *second* way to do what structs already do, which the "one obvious way" / minimalism stances push back on.
- **The dominant motivation does not apply here.** Multiple returns earn their keep in languages that return errors as a value (`(value, error)`). Jennifer raises errors with `throw` and handles them with `try` / `catch` , so errors are never returned - that removes the biggest use case and leaves only small anonymous

tuples (divmod, min / max), exactly what a two-field struct expresses.

- **It is not free plumbing, and the cost is Jennifer-specific.** Methods declare no return type or arity (a body may `return $a, $b;` on one path and `return $c;` on another), so return arity would become a *runtime* contract: a call site would newly have to care how many values came back, and a single-vs-multi mismatch would be a positioned runtime error. Keeping the tagged-union `Value` model clean would also force a statement-position-only restriction (no `f(g())` nesting) plus a slice-carrying `ReturnSignal` and a parallel call-eval path - real machinery for a narrow convenience structs already serve.

The chosen solution: **return a struct.**

```
def struct DivMod { quotient as int, remainder as int };
func divmod(a as int, b as int) {
    return DivMod{ quotient: $a // $b, remainder: $a % $b };
}
def d as DivMod init divmod(17, 5);   # $d.quotient == 3, $d.remainder == 2
```

Per-algorithm digest / checksum shortcuts (hash.sha256(b), crc.crc32(b))

Considered once digit-bearing identifiers (M22.2) made the names spellable: per-algorithm one-shot methods - `hash.md5(b)` / `sha1(b)` / `sha256(b)` / `sha384(b)` / `sha512(b)` and `crc.crc32(b)` / `crc64(b)` - beside the existing `hash.compute(b, algo)` / `crc.compute(b, algo)` .

Rejected on **stance #1 (one way per thing)** because:

- **They are a parallel API for the same job.** `hash.sha256(b)` and `hash.compute(b, "sha256")` produce the identical bytes; that is exactly the two-ways redundancy the stance rejects (its own examples are ++ / += / two `printf` flavors).
- **The rest of the family is irreducibly algorithm-as-value, so the shortcuts would make it less consistent, not more.** `hash.hmac(_, _, algo)` , `crypto.pbkdf2(_, _, _, _, algo)` , and `crypto.hkdf(_, _, _, _, algo)` all take the algorithm as a runtime string and cannot cleanly go static (SHA-1 vs SHA-256 vs SHA-512 across three functions is a large surface, and the algorithm is *negotiated* - SCRAM, JWT `alg` , TLS suites pick it on the wire). Splitting only `compute` would leave the crypto/hash surface half per-method, half algo-string.
- **"Runtime algorithm => use if()" breaks the one real caller.** `sasl.j` threads a server-negotiated `$s.algo` through `compute` , `hmac` , *and* `pbkdf2` together; forcing a per-method dispatch ladder there (while `hmac` / `pbkdf2` beside it still take the string) is worse on every axis.

The chosen form: the algorithm is always a value, uniformly - `hash.compute(b, algo)` , `hash.hmac(k, m, algo)` , `crypto.pbkdf2(..., algo)` , `crypto.hkdf(..., algo)` - so a fixed algorithm is a string literal and a negotiated one is a variable, with no new surface and no dispatch ladder.

Version-suffixing the mail protocol modules (pop -> pop3, imap -> imap4)

Considered after digit-bearing identifiers (M22.2) enabled the digit renames (`iic` -> `i2c` , `storage` -> `s3` , `uuid.v4()` / `v7()` , `pbkdf2`): rename the POP3 client module `pop` to `pop3` for wire-protocol precision.

Rejected ; the mail suite stays `pop` / `imap` / `smtp` . The digit renames all fixed a name where the digit is the **canonical identity or disambiguates a live variant** : I2C is literally the bus name, S3 the product name, and `sha256` / `sha512` / `uuid.v4` / `pbkdf2` distinguish co-existing variants - dropping the digit was wrong or ambiguous. `pop` / `imap` are the opposite case: POP3 and IMAP4 are the **sole living versions** of their protocols (POP1/POP2 and IMAP1-3 are dead), so the bare name disambiguates nothing and the digit is noise, not identity.

The tell is `imap` : renaming `pop` -> `pop3` for completeness would force `imap` -> `imap4` for consistency (nobody writes `imap4`), while `smtp` has no version at all - yielding the inconsistent trio `pop3` / `imap4` / `smtp` . Keeping `pop` / `imap` / `smtp` is the internally consistent choice (all bare, colloquial, single-live-version protocol names) and matches the universal convention (Python's `poplib` / `imaplib` / `smtplib` are POP3 / IMAP4 clients, named bare).

The rule: include the digit when it is the canonical identity or disambiguates a live variant (`s3` , `i2c` , `sha256` , `uuid.v4` , `pbkdf2`); omit it - bare name - when a single living version makes the bare name unambiguous and colloquial (`pop` , `imap` , `smtp`).

Parallel sequence-number + UID verbs in imap

Rejected. M23.3 briefly shipped a UID-addressed twin for every sequence-number `imap` verb (`fetch` + `uidFetch` , `search` + `uidSearch` , `copy` + `uidCopy` , `move` + `uidMove` , and so on - twenty verbs for ten operations). Collapsed to **UID-only addressing** (each verb sends its `UID` form and takes a UID) on **stance #1 (one way per thing)** :

- **The twins are a parallel API for the same job.** `fetch(session, n)` and `uidFetch(session, uid)` retrieve a message; they differ only in which identifier space the `int` names. That is the two-ways redundancy the stance rejects.
- **It also violated stance #2 (explicit over implicit).** With both, a bare `fetch($s, 42)` hid *which* id space `42` was in - a sequence number and a UID are different numbers for the same message, so mixing them is a silent bug.
- **Sequence-number addressing is the footgun the milestone existed to fix.** An `EXPUNGE` renumbers sequence numbers, silently breaking "fetch only what's new since last run" - the exact correctness problem M23.3 set out to solve. Keeping it as a co-equal addressing scheme preserved the trap.

The chosen form: one addressing scheme, UID, which is stable across expunges and sessions. Sequence numbers survive only where the server *emits* them as data - the `selectFolder` count and IDLE `EXISTS` / `EXPUNGE` push numbers - never as an addressing input a caller hands back in. A session-level `useUid` mode (the imapclient approach) was also considered and rejected: it fixes stance #1 but doubles down on stance #2's hidden-mode problem.

A r"..." raw-string prefix (Python-style)

Considered when splitting the two string delimiters so each does one job (raw vs cooked, M23.14). Python spells a raw string `r"..."` ; Jennifer could have kept both delimiters cooked and added an `r` prefix for the raw variant.

Rejected in favor of **making the delimiter itself the mode** - `'...'` is raw, `"..."` is cooked - because:

- **The delimiter is already free.** Before M23.14 the two quote characters were fully redundant (both cooked the same escapes), which stance #1 rules out. Giving each a distinct job spends an existing, otherwise-wasted lexical distinction rather than inventing new syntax.
- **No new lexer state.** A prefix needs the lexer (and every tool that classifies a token) to look *behind* the opening quote for an `r` , and to decide what a bare `r` identifier immediately before a string means. Branching on the quote character (`raw := quote == '\''`) is a one-line, local change with no ambiguity.
- **One obvious way.** A prefix would leave *three* ways to write a string (`"..."` , `'...'` , `r"..."`) and a redundant `r'...'` . The delimiter-is-the-mode rule gives exactly two, one per job.

The one ergonomic cost - a raw literal cannot contain a `'` - is handled by switching to the cooked form (`"it's"`), which is rare in practice. See M23.14.

An f"..." interpolation prefix (Python / C# style)

Considered when designing string interpolation (M24.19). Python marks an interpolating string `f"..."` ; Jennifer could have gated `{expr}` slots behind an `f` prefix, leaving a plain `"..."` non-interpolating.

Rejected in favor of **interpolating every cooked "..." string** (a raw `'...'` never interpolates), because:

- **The cooked / raw split already is the opt-in.** Jennifer had just spent its two quote characters on distinct jobs (raw vs cooked, M23.14). A cooked string is already the "process this" mode and a raw string the "verbatim" mode, so a raw `'...'` *is* the no-interpolation form. An `f` prefix would add a third axis on top of an existing binary one - exactly the redundancy stance #1 rules out, and the same reasoning that rejected the `r"..."` prefix above.
- **No new lexer state / bare-f ambiguity.** A prefix forces the lexer and every token classifier to look *behind*

the opening quote and decide what a bare `f` identifier immediately before a string means. Branching on the delimiter needs neither.
- **One obvious way.** With a prefix there would be `"..."` , `f"..."` , `'...'` , and a pointless `f'...'` . Delimiter-is-the-mode gives exactly two forms, one per job.

The cost - every existing cooked string holding a literal brace had to migrate to a raw string or `\{` / `\}` - was a one-time, mechanical, pre-1.0 break, paid once. See M24.19.

orm typed-struct row mapping, Active Record, and whereRaw

Three `orm` shapes were considered during its M23.15 workover and turned down.

Typed-struct row mapping (`orm.find(...) -> User` populating a user struct) was rejected because Jennifer has **no struct reflection** : a generic mapper cannot enumerate a struct's fields or set one by runtime name, so it cannot fill an arbitrary target. Rows stay `map of string to string` (the honest shape for compile-time-unknown columns); the caller rebuilds a typed value explicitly when it wants one. Reasoned in design-decisions.md; revisit only if the language grows field reflection.

Active Record (a row that carries `save()` / `delete()` methods, mutating itself) was rejected on two counts: Jennifer values are **value-semantic and method-free** (a struct cannot carry behaviour, and a `map` row certainly cannot), and a module holds **no state** (there is nowhere for a row to remember "its" connection). Persistence is therefore always an explicit `orm.insert(session, s, record)` call - Data Mapper, not Active Record - which also keeps the which-connection and which-transaction questions in the caller's hands.

orm.whereRaw(q, fragment, params) - a raw-SQL WHERE escape hatch - was rejected because it cannot keep orm's core guarantee. Every other token orm puts into SQL unparameterized (identifiers, operators, aggregate functions, join kinds) is checked against a fixed allowlist, so even a hand-built `Query` literal cannot inject. A raw fragment is arbitrary SQL: validating it would mean parsing SQL, and not validating it makes `whereRaw` the one place a caller's string reaches the database unchecked. The structured builders (`where` / `whereIn` / `whereNull` / `whereBetween` / ...) cover the common cases; genuinely bespoke SQL belongs one layer down in the `sql` library, which is explicitly the "you write the statement, values bind through placeholders" surface. Keeping raw SQL out of orm is what lets orm promise "no injection, even from a hand-built query".

Project

Jennifer - Milestones

Development is split into milestones. Each milestone produces a *working* interpreter that runs a strictly larger subset of the language.

> **Note on compaction.** Older milestone entries are periodically **compacted** - condensed to a short summary (and sometimes grouped) - to keep this file readable and its length in check. The compacted text records *what* shipped, not the full implementation design. For the detailed design, rationale, and step-by-step evolution of any milestone, **dig into the git history** (the commits for that milestone) - that is the authoritative record of how it was built.

M1 - End-to-end MVP

Smallest vertical slice that proves the pipeline (source -> tokens -> preprocessed tokens -> AST -> result):

- Types: `int` , `string`
- `def x as int init 5;` , `$var` references, method defs (zero-arg, top level), `import "file.j";` , `use io;` , single-arg `printf`
- Arithmetic `+ - * / %` on ints; comments # and `/* */`
- Source-context caret in error messages
- Golden-file integration test and TinyGo build verified

Exit criterion: `./jennifer run examples/hello.j` prints `42` .

M2 - Types, constants, scoping, control flow

Rounds out the "ordinary" feature set:

- New types `float` , `null` , `bool` with literals `3.14` , `null` , `true` , `false`
- Uninitialized `def x as T;` gives `T` 's zero value
- `def const NAME as TYPE init VALUE;` (reassignment is an error)
- Nested block scoping; inner scopes cannot redeclare visible names
- Assignment statement `$x = EXPR;`
- Comparison `< > <= >= ==` , `+` for string concat, `int` <-> `float` promotion
- Escape parsing in `'...'` strings (previously only `"..."`)
- Control flow: `if` / `elseif` / `else` , `while` , `for` , all requiring `bool` conditions (no implicit truthiness)

M3 - Methods with parameters and return values

- `func name(a as int, b as string) { ... }` with typed parameters, by-value argument passing, call-site arity + type checks
- `return;` and `return EXPR;` ; recursion works
- `sprintf` and format verbs `%d %f %s %t %v %%` for both `printf` and `sprintf`
- The omnibus `stdlib` retired in favor of topic-based libraries; `io` is the first.

M4 - Polish & ergonomics

- Logical operators `and` , `or` , `not` (word-based, short-circuit)
- Unary minus

- Python-3 division: `/` always returns `float` ; new `div` keyword for floor division (`//` is taken by line comments)
- Floats always display with a decimal (`5.0` , not `5`) so the type stays visible
- New libraries (all `use` -gated): `convert` , `math` , `strings`
- Interpreter gained `RegisterConst` so libraries can expose constants (`PI` , `E`).

M5 - Interpreter improvements

- **Cross-file error sources** - errors raised inside an imported `.j` display the line from the imported file. See technical/interpreter.md > Errors and positions.
- **REPL** - `jennifer repl` , persistent globals/methods/imports across inputs, multi-line input via brace balancing, expression results printed. See technical/cli_repl.md > REPL.
- **REPL line editor** - cursor keys, Home/End, word motions, Ctrl+W / Ctrl+U / Ctrl+K, in-memory history (Up/Down), Ctrl+C cancel. Non-TTY stdin falls back to plain line reading. See technical/cli_repl.md > Line editor.
- **Auto-loaded core library** - new library kind, pre-imported at startup; writing `use core;` is a runtime error. Contents: `JENNIFER_VERSION` (a `git describe` -derived build-version constant) and `len` (polymorphic over strings now; lists/maps in M6). `len` moved here from `strings` . (M15.4 later promoted `len` to a language built-in and deleted `core` ; see M15.4 for the migration.) Version injection details at technical/cli.md.
- **Formatter** - `jennifer fmt` re-emits canonical source per user-guide/style-guide.md. Token-level walker so file imports and user-written parentheses survive. See technical/cli_fmt.md > Formatter.
- **Inspection subcommands** - `jennifer tokens <file>` dumps the lexer output; `jennifer ast <file>` dumps the preprocessed AST as JSON. See technical/cli_inspect.md > Inspection.
- **Underscore-in-constants** - constant names became `[A-Z]+(_[A-Z]+)*` , enabling `MAX_RETRIES` and the `JENNIFER_VERSION` rename. See technical/lexer.md > Identifier rule.
- **Documentation overhaul** - `docs/technical.md` split into `docs/technical/<topic>.md` ; `docs/lib_*.md` moved to `docs/libraries/` ; new `docs/user-guide/style-guide.md` .

M6 - Lists and maps

Two new compound types - `list` and `map` - plus the strings library functions deferred until compound types existed.

- **Syntax** : `def xs as list of int init [1, 2, 3];` , `def m as map of string to int init {"a": 1};` . Index read/write `$xs[i]` , `$m["k"]` , chains `$g[i][j]` . Iteration via `for (def x in $coll) { ... }` (new keyword `in`). New tokens `[ ]` `:` and keywords `list` , `map` , `of` , `to` , `in` .
- **Semantics** : value-typed (copy on assignment and on function-parameter binding; no aliasing); `const` is deep (`$NUMS[0] = ...` is a runtime error if `NUMS` is `const`); out-of-bounds list reads/writes and missing map keys are positioned runtime errors; map iteration is insertion-order deterministic.
- **Type system** : `parser.Type` became a recursive struct (`Element` , `KeyType` , `ValType` `*Type` slots), so nesting like `list of list of int` falls out without depth cap. 3+ levels is a documented code smell.
- **Stdlib** : `core.len` extended to lists and maps; `core.has(m, key)` for membership tests; `strings.split` , `strings.chars` , `strings.join` finished.
- **Tooling** : formatter handles `[...]` / `{...}` per user-guide/style-guide.md (no inner padding, space after `,` / `:` , block-vs-map disambiguation via a small brace stack); AST JSON emitter handles `ListLit` , `MapLit` , `IndexExpr` , `IndexAssignStmt` , `ForEachStmt` .

See user-guide/types-and-values.md > Lists and maps for the user-facing tour, and technical/grammar.md / technical/interpreter.md for the implementation contract.

M7 - printf modifiers, stdin input, comment/division swap

A breaking syntax change to free up `//` for integer division and to allow shebangs, the long-promised format-verb modifier system, and the first stdin-reading builtins.

- **Comments and integer division** (**BREAKING**). Line comments moved from `//` to `#` , freeing `//` for floor division (Python 3 shape). `div` keyword removed. A Jennifer file can now begin with `#!/usr/bin/env -S jennifer run` .
- **(s)printf format-verb modifiers.** Each format verb except `%v` accepts a pipe-separated, order-independent flag list: `%verb[|key=value]*` . Modifiers shape *presentation* only - data transformations (`case=upper` on strings, markdown rendering, etc.) are explicitly out of scope; libraries do that work. Verbs gained: `pad` / `max` / `align` / `mode` (`%s`); `pad` / `fill` / `align` / `base` / `sign` / `group` / `sep` (`%d`); `prec` / `trim` / `sci` / `pad` / `align` / `sign` (`%f`); `case` (`%t`); shared `null=empty|null|literal(...)` across all four typed verbs. `%v` deliberately takes none.
- **Format-string breaking change.** `|` immediately after a verb now starts a modifier list. Pre-M7 strings with `|` as a literal separator (`"%d|%d"`) need either a different separator or the `||` escape (parallels `%%`).
- **io stdin input.** New builtins `readLine()` , `readLine(prompt)` , `eof()` - one-line-at-a-time reads with an explicit EOF predicate (`while (not eof()) { ... }`). Refuses inside the REPL since the line editor owns stdin.
- **Internals.** Builtin signature changed from `func(out io.Writer, args)` to `func(ctx BuiltinCtx, args)` so stdin and the REPL flag are plumbed symmetrically with stdout. Mechanical refactor across the ~30 existing builtins.

See:

- libraries/io.md - full modifier and input reference.
- technical/lexer.md and technical/grammar.md - the comment / division syntax change.
- technical/rejected.md - what the modifier system deliberately doesn't do (data transformations, `%a` aggregate, `null=sql` / `null=skip`) and why the literal-pipe lookahead alternative was turned down.
- technical/interpreter.md > Builtins and libraries - the `BuiltinCtx` signature.

M8 - System library namespacing

A hybrid namespace model so domain libraries can ship without polluting the bare-name pool, plus the first real namespaced library (`os`) so the machinery has a non-synthetic exercise.

- **Hybrid model.** Essential libraries (`io` , `convert` , `math` , `strings` , auto-loaded `core`) stay flat - their builtins are bare names. Domain libraries register through a new namespaced API (`RegisterNamespaced` / `RegisterNamespacedConst`) and are addressed by `prefix.name(...)` / `prefix.NAME` . The library's name doubles as the namespace prefix.
- **Qualified calls and constants.** New AST nodes `QualifiedCallExpr` and `QualifiedConstRefExpr` ; parsed as `IDENT "." IDENT` (then `(` decides). Lookup is keyed by `(namespace, name)` and gated by `use lib;` .
- **use NAME as ALIAS; aliasing.** Optional `as` clause on `use` . Rename-not-addition: after `use bio as b;` only `b.` resolves, `bio.foo()` errors with a "did you mean b ?" hint; the canonical name `bio` is freed for ordinary identifier use. Matches Python's `import foo as bar` . Aliasing is rejected for flat libraries (`use math as m;` errors as meaningless).
- **Namespace prefix is a reserved identifier.** After bare `use bio;` , `func bio() {}` errors with `shadows imported namespace 'bio'` . After `use bio as b;` , only `b` is reserved.
- **No migration.** The change is purely additive; all five flat essentials continue to work unchanged.
- **Demo library os (minimal slice).** First namespaced library: `os.platform() -> string` , `os.getEnv(name) -> string` , `os.JENNIFER_LF` , `os.JENNIFER_OS` . Two functions plus two constants - enough to exercise namespaced zero-arg calls, namespaced calls with arguments, namespaced constants, and aliasing end-to-end. Expands in M15.1.

See:

- libraries/os.md - the shipping demo library.
- libraries/index.md - flat vs namespaced policy and the rule for library authors.
- user-guide/imports.md > Namespaced libraries and aliasing - user-facing reference for `use NAME [as ALIAS];` and qualified calls.
- user-guide/style-guide.md > Namespaced calls - spacing convention around `.` .
- technical/grammar.md - EBNF for `qualifiedCall` / `qualifiedConstRef` and the `use ... as ...` shape; AST table entries for the new nodes.
- technical/interpreter.md > Namespaced libraries (M8) - registration API, `nsPrefixes` / `nsAliasedAway` resolution tables, no-shadowing rule for namespace prefixes.

M9 - Collection operations

Two new namespaced libraries cover the M6-deferred list/map manipulation helpers, a small append sugar shortens the common write pattern, and two follow-on breaking changes tidy up the flat-vs-namespaced split.

- **lists library** (`use lists;` , namespaced). `lists.push` , `lists.pop` , `lists.first` , `lists.last` , `lists.head` , `lists.tail` , `lists.reverse` , `lists.sort` , `lists.contains` , `lists.concat` , `lists.slice` . Non-mutating - every function returns a new list. `sort` accepts numeric, string, or bool elements (mixed int/float promotes; other mixes error); comparator-based sort is deferred until methods are first-class.
- **maps library** (`use maps;` , namespaced). `maps.keys` , `maps.values` , `maps.has` , `maps.delete` , `maps.merge` . Same shape. `maps.delete` of a missing key errors (strict at boundaries, matching `$m[missing]`); `maps.merge` layers the second arg over the first.
- **Sugar: $xs[] = item;** - write-only target meaning "just past the end of the list". Equivalent to `$xs = lists.push($xs, item);` . Reads of `$xs[]` and chained forms (`$xs[0][]`) are parse errors; non-list targets error at runtime. New AST node `AppendStmt` .
- **BREAKING:** `has()` moved from `core` to `maps` as `maps.has(m, key)` . Bare `has(...)` callers now need `use maps;` and the qualified form. `has` was the only non-polymorphic name in core; `len` stays because it genuinely spans string / list / map.
- **BREAKING:** `strings` library moved from flat to namespaced. `upper(s)` -> `strings.upper(s)` , `contains(s, sub)` -> `strings.contains(s, sub)` , etc. across all 15 functions. `use strings;` itself is unchanged. The M8 library-author rule named exactly these collision-prone verbs (`contains` , `split` , `replace` , `join`); acting on it now keeps callers off the wrong shape before more libraries arrive. After M9 the remaining flat libraries are `io` , `convert` , `math` , and auto-loaded `core` .

See:

- libraries/lists.md / libraries/maps.md - function reference for each new library.
- libraries/strings.md - now namespaced (M9 migration note at top).
- libraries/index.md - updated flat-vs-namespaced catalog and the library-author rule.
- user-guide/imports.md and user-guide/types-and-values.md > The $xs[] append sugar - user-facing reference.
- technical/grammar.md - EBNF and AST entry for `AppendStmt` .

M10 - Namespace-first library architecture

A pre-language-completion API-shape correction: every library is now namespaced, with bare-name globals reserved as a narrow `core` -only exception. Small implementation surface, large API shape; pre-1.0 is the window for this kind of change.

- **BREAKING:** `io` , `math` , `convert` migrate to namespaced-only. `printf(x)` -> `io.printf(x)` , `sqrt(x)` -> `math.sqrt(x)` , etc. The "io is special, keep it flat" alternative was considered and rejected at kickoff to

keep a uniform "every call carries its library name" rule. `strings` , `lists` , `maps` , `os` were already namespaced (M9/M8).

- **BREAKING:** `convert` 's four conversion callees are renamed to `convert.toInt` , `convert.toFloat` , `convert.toString` , `convert.toBool` so they don't collide with the type keywords (`int` , `float` , `string` , `bool`); `convert.typeOf` keeps its name. The `to` -prefix also reads as English ("convert to int") at the call site.
- **BREAKING:** file-splice keyword `import` -> `include` . `include "x.j";` is the textual splice; the `import` keyword is reserved for the M17 module system and produces a migration-hint error today. Mixing-mistake diagnostics updated.
- **BREAKING for embedders:** registration API renamed. `Register` / `RegisterConst` -> `RegisterGlobal` / `RegisterGlobalConst` , making their role explicit ("expose this name globally"). The namespaced API (`RegisterNamespaced` / `RegisterNamespacedConst`) keeps its name and is the recommended default. Per-library storage (`globalFnsByLib` , `globalConstsByLib`) so two libraries with the same global name can no longer silently overwrite each other at Install time; the resolution map is populated by `processImports` when a library activates.
- **math absorbs the planned non-crypto random helpers** : `math.rand()` , `math.randInt(lo, hi)` , `math.randSeed(n)` . Three functions don't justify their own library under the new threshold (next bullet); pseudo-random fits `math` 's pure-numeric charter. The crypto-grade variant still ships in M20.1 `crypto` . The originally planned M14.2 `random` library is removed.
- **core is the only library publishing bare-name globals.** `len` and `JENNIFER_VERSION` only - no `core.len` / `core.JENNIFER_VERSION` qualified form, because shipping the same name two ways violates stance #1. `core` is the auto-loaded escape hatch, and its asymmetric exposure is the whole point.
- **Three globals-publishing rules in processImports** , all forward-looking (inert today since `core` is the only globals-publishing library and can't be `use` d):
  1. Duplicate `use` of a globals-publishing library is rejected (`library 'X' already in scope`); REPL no-ops a repeat.
  2. `use X as Y;` where `X` has globals but no namespaced names is rejected as meaningless.
  3. Two active libraries publishing the same global name are rejected at the second `use` (`library "B" collides with already-active library "A" on global "VER"`). The pre-M10 flat-only-alias-meaningless check is removed for the general case but kept as rule 2.
- **Library-author guidance updated.** The `docs/libraries/index.md` "flat vs namespaced" framing is retired; the new policy is "every library is namespaced; only `core` ships globals via `RegisterGlobal` ." The "deserves its own library" threshold is raised from M8's "3+" to **"5+ functions or constants"** : anything smaller folds into the most-related existing library. The non-crypto random helpers (3 functions) are the first case the new rule caught.

See:

- libraries/io.md, libraries/math.md, libraries/convert.md - migrated library references.
- libraries/index.md - retired flat-vs-namespaced framing; new library-author policy and 5+ threshold.
- user-guide/imports.md - `use` and `include` keyword reference; namespaced-call and aliasing rules.
- user-guide/types-and-values.md - `convert.toInt` / `convert.toFloat` example placement in the "explicit conversions" section.
- technical/rejected.md - "Methods on structs" (M14.3 trigger; recorded here in M10's wake because M10's review touched the same call-shape question) and other related rejected alternatives.

No new language features land here - that's M11.

M11 - Control-flow completion

Closes the biggest daily-use gap in the language and rounds out the printf modifier table at the same time. Five new keywords (`break` , `continue` , `repeat` , `until` , `exit`) and two new printf features.

- **break; / continue;** in every loop kind (`while` / `for` / `for-each` / `repeat`). Innermost loop only; misuse

outside a loop or across a method-call boundary is a positioned runtime error. `continue` in C-style `for` still runs the step before re-checking the condition (matches C/Go).
- **repeat { } until (cond);** post-test loop. New keywords `repeat` and `until` ; `do { } while ...` considered and rejected because the inverted condition is the whole point of switching to `until` .
- **exit; / exit EXPR;** terminate the whole program (exit code 0 / EXPR-as-int). Distinct from `return` (method-scoped): skips every caller frame and remaining top-level statement. Implemented as an `ExitSignal` sentinel error the CLI translates into the OS exit status.
- **Bundled: printf %s|align=center** rounds out the align set. Rejected on every other typed verb (centred numbers break columnar output).
- **Bundled: printf %a aggregate verb** for lists and maps (deferred from M7; unblocked by M6 + M9). Modifiers: `sep` , `kv` , `open` , `close` , `depth=N` , `null=skip` . The modifier-list parser was extended with a `"..."` quoted-value form (`%a|sep=", "`) so values can contain spaces / reserved characters; standard `\n \r \t \\ \"` escapes.
- **Post-dot name relaxation.** Reserved words read as identifiers in the name slot of a qualified call (`strings.repeat` , `lists.break` if anyone wrote one), preserving the `strings.repeat` library function after `repeat` was reserved as a loop keyword.

See:

- user-guide/control-flow.md - `repeat` / `until` , `break` / `continue` scope rules, `exit` vs `return` .
- libraries/io.md - %a modifier table, `%s|align=center` example, quoted modifier values.
- technical/rejected.md - `%a|json=*` / `%a|xml=*` / `%a|yaml=*` (serialisation modifiers stayed rejected even after %a itself shipped) and the `do { } while` shape for the post-test loop.

M12 - Bytes and bit operators

Adds the buffer-shaped primitive and the bit-twiddling vocabulary the standard library needs for hashing, encoding, crypto, and network code in later milestones.

- **New primitive type bytes** - mutable byte sequence; value semantics on assignment / parameter binding; deep-const. Reads yield `int` in `[0, 255]` ; writes accept the same range and reject anything else. Append via the existing M9 `$b[] = byte;` sugar. `len($b)` returns the byte count.
- **New convert.bytesFromString(s, codec) and convert.stringFromBytes(b, codec)** - bytes <-> string codecs. Only `"utf-8"` today (further codecs ship in M15.7 `encoding`). Invalid UTF-8 input is an error - no silent replacement characters.
- **Bit operators on int** : `& | ^ ~ << >>` . Python-style precedence (comparison < `|` < `^` < `&` < shifts < `+ -`), so `$x & 0xff == 0` parses as `($x & 0xff) == 0` . `~` is bitwise NOT. Shifts are arithmetic; negative count rejected; count >= 64 saturates to 0 / -1. `^` ships as a primitive operator (CPU primitive with unique algebraic properties - same justification `-` has against being composable from + and unary `-`).
- **Non-decimal integer literals** : hex `0xff` , octal `0o755` , binary `0b1010_0110` . `_` accepted between digits in any base (including decimal `1_000_000` and float mantissas). Never adjacent to the prefix or another `_` . Lexer-only change.
- **Resolves M7-deferred stdin builtins** : `io.readBytes(n) -> bytes` (exact n; partial at EOF then `io.eof()` becomes true) and `io.readChars(n) -> string` (n runes, UTF-8 decoded). Both compose with M7's `io.eof()` unchanged.

See:

- user-guide/types-and-values.md - `bytes` type, value semantics, index-write rules.
- libraries/convert.md - codec functions, UTF-8 strictness.
- libraries/io.md - `io.readBytes` , `io.readChars` .
- user-guide/control-flow.md - bit-operator precedence table.
- user-guide/syntax.md - non-decimal literals + digit separator.

M13-M13.2 - structs and catchable errors

The composite-data milestone, batched in dependency order: M13.1 ships the struct mechanism, M13.2 the

recoverable-error story built on it (the canonical error value is a struct), together unblocking composite library returns. Design detail: git history, the `user-guide` (types-and-values, control-flow) and technical/interpreter.md docs, and `examples/structs.j` / `examples/trycatch.j` (plus the matching `showcase.j` sections).

M#	Topic	Summary
M13.1	structs / records	def struct Name { field as type, ... }; at top level (hoisted before the first statement; duplicate names error in Run, silently redefine in the REPL). Literals Name{ field: expr, ... } with every field required; def x as Name; zero-fills, recursing nested struct fields. Field read $p.field / write $p.field = ...;; lvalue chains mix [index] and .field freely ($L.from.x = 5;, $bag.items[0] = 99;) through one shared index/field walker. Value semantics like lists / maps; const deep at any depth. Strict at boundaries - unknown struct type, missing / unknown field at a literal, field-type mismatch on write, and field access on a non-struct are all positioned errors. Runtime: a KindStruct tagged-union value.
M13.2	try / catch / throw	Catchable errors (keywords try / catch / throw). try { body } catch (NAME) { handler } binds the thrown value to $NAME in a fresh per-handler scope; throw EXPR; raises any value, convention the auto-hoisted Error{kind, message, file, line, col} struct. The runtime errors today's builtins / ops raise (out-of-bounds, missing key, type mismatch) are wrapped into Error on entry to the catch (kind defaults to "runtime" until a site opts into a tag); throw $err; in a catch re-raises to the enclosing try. Not catchable: exit (propagates through try), and return / break / continue (control flow, flow through try unchanged). No finally, no typed catch in v1. Internals: an ErrorSignal sentinel parallels ExitSignal, runtimeError.Kind threads the tag, and the Error struct is reserved (user code may not redefine it).

M14 - Lexer comment + blank-line preservation

Closes the two M5-deferred items (`fmt drops comments` , `fmt drops blank lines`). No language change - the runtime still never sees comments.

- Lexer emits trivia tokens (`TOKEN_COMMENT_LINE` , `TOKEN_COMMENT_BLOCK` , `TOKEN_COMMENT_SHEBANG` , `TOKEN_BLANK_LINE`). Shebang on line 1 col 1 is its own kind; runs of blank lines collapse to one.
- Preprocessor and parser strip trivia at entry; `jennifer fmt` walks the raw lexer stream and re-emits trivia via a dedicated `emitTrivia` path that doesn't disturb the surrounding state machine.
- Block comments nest via depth counter; unterminated comments error at the outermost `/*` .
- Token-level over AST-level: the original spec proposed AST-attached `LeadingComments` / `TrailingComment` slots and a `jennifer ast --with-comments` flag - dropped in favour of the simpler token-level path. Add them back if a future doc generator needs structured per-statement attachment.

See:

- user-guide/style-guide.md - Comments section (block comments nest; inline-comment spacing exception).
- technical/lexer.md - trivia emission, shebang detection, nesting depth counter.
- technical/cli_fmt.md - `fmt` 's trivia re-emission.

M15 - foundational libraries + first public release

Nine sub-milestones - two language (M15.2, M15.4), the rest library / tooling / release - that built out the foundational stdlib and shipped the first public release. Design detail: git history and the linked per-library docs. Two API patterns established here recur across later libraries:

- **Codec-table shape** - the algorithm / format / codec is a string argument (`hash.compute(b, algo)` , `crc.compute(b, algo)` , `encoding.encode(s, codec)` , `encoding.toText(b, format)`), collapsing parallel verbs into one (stance #1) and sidestepping the letters-only identifier rule that rejects `hash.md5(...)` .
- **Integer-handle struct for opaque resources** - a namespaced struct with a single `id as int` indexing a Go-side map (`os.Process` , `hash.Stream` , `crc.Stream`).

M#	Topic	Summary
M15.0	existing-library extensions	lists.shuffle(xs) (Fisher-Yates, respects math.randSeed) and lists.range(start, end[, step]) (half-open; the single-arg form is deliberately omitted, stance #2).
M15.1	os reshape + meta	Immutable per-run host facts became uppercase constants (PLATFORM / ARCH / EOL / DIRSEP / PATHSEP / ARGS), operations stay functions (getEnv / hasFlag / flag); dropped the JENNIFER_ prefix. New meta library for interpreter identity (VERSION / BUILD); the CLI forwards trailing args to os.ARGS (script path at index 0). Breaking renames (JENNIFER_VERSION -> meta.VERSION, os.platform() -> os.PLATFORM, os.JENNIFER_LF -> os.EOL), old names now plain "undefined" errors.
M15.2	library-provided namespaced structs (language)	def x as lib.Name; type syntax + lib.Name{field: ...} literals + the Go Interpreter.RegisterNamespacedStruct API, reusing M13.1's value-semantics / deep-const / strict-boundary machinery (only resolution differs). User code can't register structs (Go-side only); no methods-on-structs / inheritance. Unblocks os.Result/Process, time.*, the hash/crc streams, later fs/net.
M15.3	os external-program execution	First consumer of M15.2. os.Result{exitCode, stdout, stderr} + os.Process{pid}; os.run(argv) -> Result (blocking), os.spawn(argv) -> Process (non-blocking) + wait / poll / kill. argv is always list of string (no shell parsing - use ["sh", "-c", $cmd] explicitly). Non-zero exit codes are values, not errors. First user-visible two-binary split: jennifer-tiny has no os/exec, so it returns a friendly "use the default jennifer binary" error.
M15.4	len built-in, core removed (language)	Promoted len(EXPR) from the auto-loaded core library to a reserved keyword + primary expression (polymorphic over string / list / map / bytes). Deleted internal/lib/core/; use core; returns a friendly migration error pointing at the built-in and meta.VERSION/BUILD. Every library now lives behind a use NAME; (stance #2, no exceptions).
M15.5	time	Instants, durations, fixed-offset zones, strftime format / parse, ISO 8601 round-trip. Structs time.Time{nanos, offset} (private fields), time.Duration{nanos}, time.Zone{offset, name} (public, so an IANA/DST companion can build them). Granularity is a formatting property, not a type; Unix timestamps are constructor/accessor pairs, not a type. IANA/DST stay out of the fixed-offset core (a Go-backed extension, not a .j data map). Three parts: core + Unix + calendar + 1-based-ISO-weekday + arithmetic; strftime (%Y %m %d %H %M %S %z %a %A %b %B %j %u %%) + time.zone/inZone + time.UTC const alongside time.utc() + time.iso/fromIso; and examples/benchmark.j (the TinyGo-vs-Go suite; the sieve became trial-division since value-semantic list mutation makes a sieve O(N^2)).
M15.6	hash + crc	Two parallel codec-table libraries: hash (crypto-style digests md5 / sha1 / sha256), crc (crc32 IEEE / crc64 ECMA) - the split keeps "transport integrity" vs "content addressing" visible at the import line (mirrors Go's crypto/* vs hash/crc*). Output is raw bytes. compute(b, algo) one-shot + stream(algo) / update / finalize (the integer-handle pattern). No convenience wrappers like hash.md5String - compose convert + encoding (stance #1). Struct hashing deferred.
M15.7	encoding	Introspection (isAscii / lenBytes / lenRunes), binary-to-text toText / fromText (hex / base64 / base64-url), character encode / decode / codecs (ascii / iso-8859-1 / windows-1252 / ebcdic). The cross-kind UTF-8 pair stays in convert (M12); encoding owns the codec proliferation. Exact-match codec / format names (the original alias / case-normalisation layer was later dropped, stance #2); Windows-1252's five undefined positions reject symmetrically. The long-tail single-byte codecs shipped later in M16.15.

M15.8	distribution + first public release	Packaging / CI / release only, no language change. CI: PR gate (go vet + gofmt + go test ./... + make build + per-binary smoke + em-dash scan); release on bare-semver tags cross-compiling linux/{amd64,arm64}, QEMU-smoke-testing the non-native arch, running the benchmark, publishing a draft Release. Packaging under packaging/{debian,arch,mime,man}/ (.deb via build-deb.sh with man pages + text/x-jennifer MIME; AUR PKGBUILD-bin / -git, auto-filled by the pipeline). Docs site via pinned mdBook 0.5.3 -> GitHub Pages. Conventions kept: bare semver tags (no v prefix), no top-level LICENSE (LGPL text in packaging/debian/copyright + a README link). Deferred (not gated): the macOS / Windows cross-build and a real apt repo stay in Requirements for 1.0.0 stable.

M16 - I/O libraries and developer tooling

System libraries that touch the OS or do significant compute, opened by the `spawn` concurrency primitive (M16.0) the I/O libraries build on, then a lint / profile / test developer-tooling trio and a run of self-contained data libraries. Design detail: git history and the linked per-library / per-tool docs.

M#	Topic	Summary
M16.0	lightweight concurrency	spawn { ... } (block primary expression), task of T (new compound kind), the task library (wait / poll / discard / waitAll / waitAny). Goroutine-backed but race-free by construction: snapshotForSpawn deep-copies a globals+locals snapshot at launch; tasks share only the TaskState pointer (the one value-semantics carve-out). A per-run registry loud-fails unobserved task errors at exit (an undiscarded non-terminating spawn hangs at exit - the documented trade-off); task.wait re-raises a body error at the wait site; waitAny is the runtime's only reflect.Select. TinyGo builds with -stack-size=4mb -scheduler=tasks.
M16.1	fs	Blocking filesystem I/O composed with spawn (no *Async): whole-file read/write/append (String/Bytes), metadata (exists/isFile/isDir/stat -> fs.Stat), dir ops with the two-verb recursion split (mkdir/mkdirAll, remove/removeAll, rename/list/walk), buffered fs.File handles (open/readLine/.../close, eof peeks a byte). Path- vs handle-form verbs dispatch on the first-arg kind; fs.File shares state across copies (handle carve-out).
M16.2	net	TCP (connect/listen/accept/readBytes/writeBytes), UDP (listenUDP/sendTo/recvFrom), DNS (lookup/reverseLookup); polymorphic close/address over three handle registries; blocking calls compose with spawn (accept-loop-per-connection). Build-tag split: jennifer-tiny returns friendly-error stubs (no netdev in TinyGo).
M16.3	regex	RE2 (Go regexp, linear-time) over strings: matches/find/findAll/replace/split/escape; regex.Match{text,start,end,groups,groupsNamed} (rune indices, start=-1 = no match); implicit 128-entry LRU pattern cache. Both binaries.
M16.4	testing (primitives)	The irreducible system side a .j test framework needs: testing.run(name) invokes a zero-arg user method via Interpreter.CallByName, times it, classifies failures into a testing.Result; results/reset (mutex-guarded); report (text / TAP / JUnit). The one place exit is caught (Go-level, so language try/catch still can't).
M16.5	interpreter performance pass	Five sub-parts, behaviour unchanged: .1 shared-marker COW on compound Values (append-in-a-loop O(N^2) -> amortised O(N)); .2 parse-time lexical slot resolution ((Depth,Slot) coordinates + a slots slice; undefined/shadowing promoted to parse-time errors); .3 pooled + pre-resolved + slot-bound method-call frames; .4 namespaced-call / comparison / arg-bind / root-cache fast paths; .5 compile-time constant folding + a Share() scalar fast path. Numbers in tinygo.md.
M16.6	lint	jennifer lint flags compile-legal-but-suspect patterns: grouped IDs (L0nn source errors / L1nn correctness / L2nn style / L3nn lifecycle), # lint-disable[-file]: IDS suppression, --checks / .jennifer-lint config, human / JSON / GitHub output (a JSON pipeline stays valid even on a source error); exit 0/1/2. !tinygo.

M16.7	profile	jennifer profile attributes work to .j source positions (what go tool pprof can't): a statement profile (hit count + self/cumulative wall-clock) and an --allocs value-copy profile; table / pprof (hand-encoded gzipped protobuf) / Chrome-trace output; program output to stderr so the profile owns stdout. !tinygo.
M16.8	testing framework consolidation	An assertion vocabulary on M16.4 (assertEqual ... assertThrows, throwing Error{kind:"assertion"} at the call site), CallByNameWith/runWith arg dispatch, and the jennifer test subcommand (test* discovery or --filter, setUp/tearDown, --isolated per-test subprocess, text/TAP/JUnit, exit 0/1/2). Builtins can now raise a catchable error via interpreter.RaiseError.
M16.9	json	Hand-rolled RFC 8259 encode/decode onto the tagged-union Value (no encoding/json, no reflect). encode/encodePretty/decode; structs and map of string to V -> objects, bytes -> base64, integral numbers -> int else float. Also closed a type hole: a generic collection (a fresh literal or decode result) is validated entry-by-entry against the declared element type at every binding boundary. Decode's return shape was later superseded by M16.16 (json.Value).
M16.10	uuid	RFC 9562: generate("v4") (random) / generate("v7") (time-ordered) - the version a string arg since identifiers were letters-only - plus parse/isValid/version and constant NIL. Randomness was later repointed from math to crypto's crypto-grade source (M20.1), making v4/v7 unguessable; M22.3 renamed generate("v4") -> v4().
M16.11	compress	Byte-stream size reduction (distinct from encoding's representation codecs): pack/unpack for gzip/zlib/deflate with an optional fast/default/best level, plus a streaming compress.Stream. Go compress/*, TinyGo-clean.
M16.12	archive	tar / zip containers over bytes (no fs, value-semantic): pack/unpack (verbs shared with compress) for tar/zip/tar.gz; a bundle is a list of archive.Entry{name,data,mode,mtime}. Go archive/tar+archive/zip.
M16.13	os.isTerminal	os.isTerminal("stdout"/"stderr"/"stdin") -> bool (the ANSI-colour gate) via the char-device mode bit (os.ModeCharDevice) - pure stdlib (keeps x/term CLI-scoped), TinyGo-clean; an unstattable stream reports false.
M16.14	net TLS	net.connectTLS(address) (implicit) and net.startTLS(conn) (in-place STARTTLS upgrade), both yielding the transport-agnostic net.Conn. Cert verification on by default, net.TLSOptions{skipVerify, caCert} opt-out. Go crypto/tls on the !tinygo build (stubbed on tiny).
M16.15	encoding completion	toText/fromText gained quoted-printable, base32/base32-hex, ascii85, z85; the full ISO-8859-{1..16} / Windows-{1250..1258} single-byte codecs, generated from the Unicode mapping files (gen_codecs.go -> codecs_gen.go) so only ascii/ebcdic stay hand-written. Exact-match codec/format names (the normalisation layer was dropped, stance #2).
M16.16	json.Value	The strict home for heterogeneous JSON without a language top type: json.decode returns an opaque json.Value - the first KindObject (the opaque sibling of KindStruct: discriminated by (namespace, name), minted only by a library, rejecting operators / [i] / .field). convert.typeOf -> "object", convert.objectType -> "json.Value". Reads + non-mutating writes share JSON Pointer (RFC 6901): typeOf/get/has/keys/length/as*/isNull and map/list/set/insert/append/remove/move (strict / no-vivify, - end-marker), node types in list/map vocabulary; a displayer renders a handle as its JSON. json.decode's return type changed (a pre-1.0 break); the decoder's number grammar tightened to json.org. No any keyword (rationale in rejected.md).

M17 - module system for Jennifer-coded libraries

Jennifer-coded libraries get their own namespace, scope, and explicit `export` s via a real **module boundary** (`import "x.j" as x;` , a parser

- interpreter feature) beside the textual `include "x.j";` splice (a preprocessor

operation for composing one module from several files). Settled cross-cutting decisions (turned-down

alternatives in rejected.md): each module is its own resolution context (own `use` set, namespace + export tables); the module top level is **declarations-only** (no mutable state, so `spawn` capture is unaffected); the one global `Error` stands (modules add distinctly-named error structs, never redefine it); **private by default** , a leading `export` publishes (no `public` / `private` keyword); multi-file modules assemble via `include` behind one entry file (no directory-as-module); a module needs a filesystem (an FS-less `jennifer-tiny` host fails with the ordinary search-path error). Design detail: git history, imports.md, interpreter.md.

M#	Topic	Summary
M17.1	source tree + resolution	internal/module Classify + Resolve map an import path (local ./ / ../, absolute /, or a bare name walked on the search path) to a canonical absolute path, rejecting an ambiguous name (found in two search dirs) and a not-found. The system module dir resolves --sysmoddir > JENNIFER_SYSMODDIR > compile default (surfaced as meta.SYSMODDIR; a named-but-missing dir refuses to start, the compile default is best-effort); -I DIR (repeatable) appends after it. jennifer version -v reports the layers.
M17.2	import statement + loader	import "path.j" [as NAME]; is a real statement (ModuleImportStmt - the preprocessor passes it through, the parser builds the node). The loader runs each module in a fresh sub-interpreter sharing one moduleReg, so run-once (cached by canonical path), depth-first post-order init, and cycle detection (erroring with every edge named) all fall out of the recursion. Load-time errors (a parse error or a throwing def const init) aren't catchable (import is a declaration, not an expression). fmt / ast / tokens round-trip an import line.
M17.3	module scope + namespacing	Declarations-only top level (checkModuleDeclarationsOnly: only def const / def struct / func / use / import; scripts keep both). loadModuleImports binds each alias (the as NAME, or the file stem) into moduleAliases, collision-checked against library prefixes. Consumer resolution rides the qualified-reference eval layer: evalQualifiedCall / evalQualifiedConst dispatch alias.fn(args) into the module's own interpreter via CallByNameWith (args evaluated in the consumer, body run against the module's globals + methods) and read alias.CONST. use non-transitivity, run-once sharing, and -race safety all follow from the fresh-sub-interpreter-per-module model.
M17.4	exports + visibility	export publishes a top-level func / def struct / def const; unmarked names stay private (reaching mod.helper is a positioned "not exported" error), and export in a run script is rejected. checkReferentialClosure rejects an exported field / parameter typed as a private module struct (library / namespaced types cross freely). Cross-module struct identity = boundary translation (retagStructs): a module's structs are bare inside it and re-tagged to (module-stem, name) as a value crosses out to an importer and back, so def p as mod.Point, mod.Point{...}, field reads, and pass-back all type-check while a.Point / b.Point stay distinct; the retag also tags the element-type metadata a list / map carries (retagType). A co-located MODULE_test.j overlay (a token splice in jennifer test) runs white-box tests against the module's private names.
M17.5	ansi module	First module built on the system (pure .j, one use os; across the boundary; a real dogfood of import / export / resolution). Exports color / bgColor / style (bold / dim / italic / underline / reverse) / rgb truecolor / strip, plus per-colour and per-style shortcuts (ansi.red(s), ansi.bold(s), ...). The ESC byte is built from a one-byte bytes; strip uses regex; unknown names throw. Stateless + TTY-aware: enabled() re-reads NO_COLOR / FORCE_COLOR / os.isTerminal("stdout") per call (no toggle state; degrades to always-on when os.isTerminal is absent). Colour is a string wrapper, so a `%s
M17.6	semver module	Strict SemVer 2.0.0 (https://semver.org) as a second pure-.j reference module (use strings / convert / regex, so both binaries), and the base a future jvc package manager needs. Exports Version{major, minor, patch, prerelease, build} + parse (throws on invalid) / isValid / toString; compare / lt / eq / gt (full SemVer precedence: numeric core, prerelease ranks below release, build ignored); isStable (0.y.z unstable by convention) / isPrerelease; incMajor / incMinor / incPatch; and sort (own pass over compare, since lists.sort is scalar-only). Strict - a loose 1.2.3.4 is rejected. parse uses the anchored RE2 pattern with named groups; precedence + sort are hand-written (the algorithmic dogfood). Building it surfaced + fixed the M17.4 retagType gap (a consumer list of semver.Version handed back into a module list of Version). Range / constraint matching (^1.2.0, >=1.0.0, ~1.2.3) deferred to jvc.

M18 - Jennifer-coded modules

Built atop the existing system libraries. Each one ships as a Jennifer **module** under `modules/` (the directory introduced in M17); none of them are compiled into the interpreter binary. Sub-milestones in priority order.

Forty sub-milestones (with their nested parts) shipped as pure-Jennifer `modules/` (except where noted as a Go **system library**), each with the standard discipline: a 100%-passing `*_test.j` overlay, a `cmd/jennifer/*_test.go` integration test, a `docs/modules/*.md` reference, an `examples/modules/*_demo.j` , and catalog / README / `JENNIFER.md` entries. Per-function detail lives in docs/modules/; this table is the milestone-number index (numbers were assigned in rough priority order).

M#	Module(s)	Surface
M18.1	csv	RFC 4180 parse / format (+ *With for any delimiter), header-keyed toRecords / fromRecords.
M18.2	htmlwriter	build an HTML element tree and render escaped HTML5 (element / text / raw / render).
M18.3	markdown	Markdown -> HTML.
M18.4.1/.7	mime	RFC 5322 / 2045 message build + parse, incl. RFC 2047 encoded-words.
M18.4.2/.4	smtp / pop / imap	mail send + POP3 / IMAP receive over net (plaintext / STARTTLS / implicit TLS).
M18.4.5/.6	sasl / idna	SASL auth encoders (incl. XOAUTH2); Punycode / IDNA domains.
M18.5	redis	RESP2 client over net.
M18.5.1	resque	Resque-wire-compatible background jobs on redis.
M18.6	memcache	memcached text-protocol client over net.
M18.6.1/.2	session / ratelimit	server-side sessions + fixed-window rate limiting on memcache.
M18.7	http	HTTP/1.1 client over net (https:// via TLS).
M18.7.1/.3	gotify / rest / oauth	push notifications; ergonomic REST layer; OAuth2 get-a-token - all on http.
M18.8	toml (library)	RFC TOML 1.0 encode / decode; opaque toml.Value, JSON-Pointer walk. TinyGo-clean.
M18.9.1	httpd (library)	HTTP/1.1 server engine over net/http; pull-loop accept / respond.
M18.9.2	web + jennifer serve	.j routing framework over httpd (routes by handler name, :param, middleware, web.Context), dispatched by meta.callMain; serve runs / --watch-reloads a program.
M18.10	flatdb	file-backed JSON document store over json + fs; JSON-Pointer query / edit; crash-atomic save.
M18.11	gpio	Linux GPIO over the sysfs / character-device interface.
M18.12	docblock	the Jennifer doc-comment format + parser (FileDoc tree, drift diagnostics).
M18.13	mqtt	MQTT 3.1.1 pub / sub client over net.
M18.14	prometheus	metrics exposition (produce) + retrieval (query the HTTP API).
M18.15	label	industrial label printing: build / render (ZPL + cab JScript) / emit pipeline.
M18.16	web cookies + sessions	cookie helpers + cookie-keyed sessions on the web framework.
M18.17	totp	RFC 6238 TOTP: generate / verify / uri. Over hash.hmac + encoding + time.
M18.18	webhook	GitHub X-Hub-Signature-256 HMAC sign / verify (pure) + send (over http).

M18.19	s3	S3-compatible object storage over http (AWS SigV4): connect / get / put / delete / listObjects. One module for AWS S3 + MinIO / R2 / B2. Renamed bucket -> s3 once the M22.2 digit-identifier rule made s3 a legal namespace (cf. M22.3's iic -> i2c); a pre-1.0 break, one batch across the module / overlay / Go test / demo / docs.
M18.20	dotenv	.env config: parse / read / load (into env via os.setEnv). Over fs + strings + os.
M18.21	cron	parse cron expressions; next(schedule, after) / matches. A calculator over time.
M18.22	log	leveled structured logging (debug..error; text / logfmt / json) to stdout / stderr / file / RFC 5424 syslog.
M18.23	ical	iCalendar (RFC 5545) build + parse: a Calendar of VEVENTs, escaped + line-folded, dates through time.
M18.24	vcard	vCard (RFC 6350) contacts build + parse; shares the content-line codec (ical_vcard_shared.inc.j) with ical.
M18.25	jsonl	JSON Lines (NDJSON): encode / decode + whole-file + streaming Reader, over json + fs.
M18.26	ipnet	IPv4 / IPv6 addresses + CIDR math: parseAddress / toString (RFC 5952) / parse / contains / netmask / broadcast.
M18.27	ntp	SNTP network-time client over UDP: query / queryWith -> Result (server time + clock offset + round-trip delay).
M18.28	statsd	fire-and-forget StatsD metrics over UDP (count / gauge / timing / set); the push counterpart to prometheus.
M18.29	influxdb	InfluxDB 1.x client on http: line-protocol Point builders + write; query -> parsed Series.
M18.30	slack / discord	incoming-webhook chat notifiers on http: plain send + Block Kit / embed builders (sendMessage).
M18.31	telegram	Telegram Bot API on http: sendMessage / sendPhoto / getMe, getUpdates long-poll (stateful receive loop).
M18.32	websocket	RFC 6455 client over net (ws:// / wss://): handshake + masked send / receive (auto-pong, fragment reassembly).
M18.33	amqp	AMQP 0-9-1 client for RabbitMQ over net: handshake, declareQueue, publish, get (Basic.Get pull), ack.
M18.34	multipart	multipart/form-data (RFC 7578) build + parse (binary-safe); web.multipartForm pairs it with web.
M18.35	pdfwriter	generate PDF documents (text / lines / rects, Standard-14 fonts, FlateDecode via compress); byte-identical output.
M18.36	bloom / ringbuffer	data structures: a Bloom filter (probabilistic set) + a fixed-capacity ring buffer (bounded FIFO).
M18.37	tengine	a text/template-subset engine over a json.Value tree (if / range / with / pipes / layout inheritance).
M18.38	barcode	QR (Reed-Solomon over GF(256), masking, versions 1-10) + 1D (code128 / code39 / ean13 / ean8 / itf); SVG / PNG / terminal.
M18.39	mikrotik	MikroTik RouterOS API client over net: sentence-based binary framing, talk / print / run, plaintext + MD5 login.
M18.40	password	password generate / validate / score against a policy Schema; entropy-based complexity (non-crypto RNG).

Enabling changes these modules pulled into the system side (each documented under its library):

- **net.setDeadline** - a read/write deadline for socket timeouts (M18.13; later extended to UDP sockets for `ntp`, M18.27).
- **io.eprintf** - the stdout- `printf` twin that writes to stderr (a new `Interpreter.Err` / `BuiltinCtx.Err`

writer), the stderr sink `log` builds on (M18.22).

- **toml** and **httpd** - two new Go **system libraries** (a char-by-char TOML parser and a `net/http` server engine both belong in Go, not a `.j` module); M18.8 / M18.9.1.
- **meta.callMain / meta.definedMain** - resolve a method against the entry program (retagging module-own struct args across the boundary), the capability the `web` framework dispatches handlers through (M18.9.2).
- **hash.hmac** (RFC 2104) and the **sha512** digest - the HMAC primitive `totp` / `webhook` build on (and that `jwt` / SigV4 will reuse).

M19 - cross-cutting tooling

The catch-all bucket for interpreter / tooling work belonging to neither the M18 `.j` modules nor the M20 Go system libraries: M19.1-M19.5 a correctness / performance hardening pass over the core + libraries; M19.6 the coverage tool; M19.7 the `@scope/package` vendored-module resolver; M19.8 the one-time org / vanity-path relocation; M19.9 the audit-driven hardening pass. No `reflect` , no TinyGo-cleanliness break. Design detail: git history and the linked docs.

M#	Topic	Summary
M19.1	interpreter concurrency-safety	Both interpreter data races fixed, each pinned by a -race stress test. snapshotForSpawn snapshots the launching goroutine's own root frame (effectiveGlobal(env)), not the live i.global, so a nested spawn no longer races the main goroutine's global writes. Declared struct types are stamped once, single-threaded, before any statement (resolveDeclaredTypesOnce, after loadModuleImports) with a parser.Type.Resolved marker, so the per-execution re-resolve is a read-only no-op (also fixes a latent aliased-library-struct "canonical is aliased" rejection). Error timing unchanged.
M19.2	value representation cleanup	Removed the inert copy-on-write machinery (Value.shared, Share(), Ensure(), ensureCOW, the per-VarExpr-read Share()); the mutation sites grow the binding's own backing in place, reads return the binding directly. Value semantics rest (as before) on eager deep copies at every store site; the write-through alternative is in rejected.md. Dead COW reporting stripped from the --allocs profiler. A fresh list / map / struct literal RHS is already private, so execDefine / execAssign skip the redundant copy (rhsFreshLiteral), proven by a profiler-backed test.
M19.3	runtime perf: maps + call/loop hot path	Maps gained an advisory hash index (Value.mapIdx, encoded scalar key -> position) guarded by a len(mapIdx) == len(Map) stamp, so $m[$k] = $v over N keys is O(N) not O(N^2) while insertion order + value semantics are untouched (any stale / duplicate-key / non-hashable case fails the stamp and falls to the linear scan; a 100k-key build + for-each runs sub-second where the quadratic path took minutes). Plus: execForEach / execFor borrow frames from envPool; DefineAt skips the shadow walk on the slot path; Run pre-sizes i.global's slots; the three mutation sites write the root binding through (Depth, Slot); lists.reverse/head/tail/slice/concat shallow-copy instead of deep-copying an argument they overwrite.
M19.4	resource lifecycle + numeric strictness	os.spawn handles keyed by a monotonic id, not the OS pid (a recycled pid can't alias a handle or misroute wait/poll/kill); the reaper drains captured buffers to strings and drops the live *bytes.Buffers (idempotent wait / poll-after-wait preserved). Numeric strictness: convert.toInt and math.floor/ceil/round reject NaN / +/-Inf / out-of-int64 floats; math.abs(MinInt64) errors; the toml decoder errors on an int past int64 (json keeps its deliberate fallback); the most-negative int literal -9223372036854775808 parses to MinInt64 (folded at unary-minus with a 2^63 range check).

M19.5	module struct identity: canonical path	Module structs were tagged only by the file stem, so two modules sharing a basename (a/util.j, b/util.j, or two @scope/package decks) produced identical (namespace, name) identity and a foreign struct passed the other's type check. Identity is now keyed by the module's canonical (resolved) path: Value and parser.Type gained a ModPath field that Equal / MatchesDeclared compare alongside StructNS (which stays the stem, for display, so %v still reads benchmark.Point); the boundary retag + method-parameter stamping thread the path. Two imports of the same file stay one type; different files differ - no import error.
M19.6	.j code coverage	`jennifer test --coverage[=text
M19.7	@scope/package resolution (vendored decks)	A leading @ is a vendored-deck reference expanded by one function (resolveVendor): the @ swaps in the vendor root and a reference not ending in .j gets the package-named entry appended, so @jennifer/bitcoin, @jennifer/bitcoin/, and @jennifer/bitcoin/utils.j all reduce to a plain absolute path (after which run-once cache + M19.5 path identity are untouched). The entry is <package>.j, so moduleStem gives the package name and the display namespace / default alias fall out (import "@jennifer/bitcoin/" binds bitcoin.); two same-package decks across scopes are distinct types, colliding only on the default alias (resolve with as). Vendor root via FindVendorRoot: --vendor > JENNIFER_VENDOR > nearest vendor/ above the program. Path safety: @ front-only, no ./.., the file must stay inside the deck; a missing root is a guided error. The jvc manager over this is M26.
M19.8	relocation: jennifer-language org + vanity path	A one-time mechanical relocation, no language / interpreter behavior change: the repo moves to a jennifer-language GitHub org, and (separately) the Go module path moves off GitHub to a vanity path jennifer-lang.dev/jennifer served by a go-import meta page, so module identity is host-independent. Two distinct targets: the Go module path -> the vanity domain (go.mod + 112 .go imports rewritten; the meta page maps it to the org repo, plus a go-source tag for pkg.go.dev); human-facing URLs -> github.com/jennifer-language/jennifer (the -lang.dev vs -language org spelling is deliberate; the redirect keeps old links working, but canonical in-tree URLs are updated). Metadata / CI / packaging swept until grep -rn 'mplx/jennifer-lang' is empty; the first-party deck scope placeholder flips @mplx/ -> @jennifer/ (doc-only until jvc).
M19.9	audit-driven correctness + hardening	A systematic severity-ordered sweep of ~190 findings from a full bug / performance audit of internal/ + modules/, each fix with a regression test (Go test or _test.j overlay). Crash / safety: OS-entropy RNG seeding (predictable UUIDs / session ids / passwords fixed; math.randSeed stays the deterministic opt-in), json / toml decode nesting caps, a try-body scope fix (a throw-skipped def reads undefined, not null), tengine recursion guards, archive zip-slip + aggregate-decompression caps, a JSON-pointer overflow guard, the two interpreter -race races + REPL-vs-spawn table mutation. Correctness: lvalue writes re-fetch their root after the RHS, index / append stamp the element type, path-keyed module structs, barcode (Code 128 stop / EAN check digit / Code 39 * / QR mask 3), pdfwriter WinAnsi / Info encoding, a full toml conformance pass, http / web (CSRF / cookie / CORS / ETag) hardening, quote-aware vcard / ical. Performance: the O(N^2) accumulation patterns retired (map hash index, json object decode, wire-framing reads in redis / amqp / mqtt / mikrotik / imap / websocket, list-join builders in csv / barcode / influxdb / jsonl / statsd, GF(256) inline). Lifecycle: os.release + capped child output, non-blocking net.eof + mutex-guarded net.Conn, httpd admission bounds / must-respond timeout / TLS-1.2 floor / safe unix-socket unlink, per-stream mutex + a discard verb on hash / crc / compress streams, lint descending into spawn / repeat. Plus six coordinated pre-1.0 strictness breaks (each with tests + operator / scoping docs): % is floored (Python; -7 % 3 == 2, 7 % -3 == -2); integer arithmetic overflow is a positioned error, not a silent wrap; a duplicate map-literal key is an error; mixed int / float comparison is exact (no lossy promotion, so 9007199254740993 == 9007199254740992.0 is false); a method may not share a top-level var / const name (no-shadowing both directions); reading a constant with the $ sigil ($MAX) is a parse error.

M20 - system libraries

Go **system libraries** (cryptographic primitives, plus formats too heavy or too reflect-bound for a Jennifer-coded `.j` module - the `json` pattern, M16.9), plus two cleanup keywords (`defer` / `errdefer`) and Unix-signal support that the libraries needed. Ten sub-milestones, each shipped with the standard discipline: a `cmd/jennifer` / `*_test.go` suite, a `docs/libraries/*.md` reference, cheatsheet + `JENNIFER.md` entries, and a runnable example. Per-function detail lives in docs/libraries/; this table is the milestone-number index.

M#	Library / feature	Surface
M20.1	crypto	Security primitives above hash, Go stdlib only: crypto-grade randBytes / randInt (rejection-sampled, unseedable), constant-time hmacEqual, key derivation hkdf / pbkdf (algo sha1/256/512). Repointed uuid's random source here, so v4 / v7 are unguessable.
M20.2	xml	Hand-rolled XML encode / decode over an opaque xml.Value (KindObject) element tree; read (tag / text / attr / children) + XPath-style get / findAll / has (name / name[k] / *) + build (element / setAttr / setText / append). No encoding/xml.
M20.3	yaml	YAML 1.2 over yaml.Value, the same read / walk / write surface as json / toml + asDatetime; decode / decodeAll, anchors / aliases by value, << merge keys. Backed by gopkg.in/yaml.v3 (the one config parser that earns a dep); pre-parse depth + node-budget guards.
M20.4	intl	Message catalogs + locale-aware tr(key[, params]) ({name} interpolation, locale -> base -> default -> key fallback); load / setLocale / locale. A system library (global mutable state + O(1) Go-map lookup); single-pass, output-capped interpolation.
M20.5	term	Terminal host control for TUIs: raw mode (makeRaw / restore, single-use State), size -> Size{rows, cols}, readByte. Over golang.org/x/term, build-tag split (stub on jennifer-tiny); refused in the REPL.
M20.6	signals	Cooperative Unix signals: SIGUSR1 -> live interpreter diagnostics (dump-and-continue); os.catchSignal / os.gotSignal (opt-in int / term / hup / usr2) for graceful shutdown; CLI terminal-restore-on-abort. Establishes the checkpoint hook the loop-cancellation follow-on will build on.
M20.7	defer / errdefer	defer CALL(args); - single call, args snapshotted, block-scoped LIFO, runs on every exit path, never crosses the method / spawn boundary (no finally - rejected). errdefer (Zig-style) fires only on a propagating error; adopted by the seven connect-then-handshake modules. Paired with the new fs.sync.
M20.8	serial / spi / iic / gpio	Device I/O over Linux /dev + ioctl (x/sys/unix), build-tag split linux && !tinygo: three buses + character-device GPIO (reusing the sysfs M18.11 module's pin-keyed shape). I2C is iic (letters-only). Shared plumbing in internal/lib/devio; ioctl struct layouts pinned to the kernel ABI by size assertions.
M20.9	sql	Relational client over database/sql: MySQL / MariaDB (go-sql-driver/mysql) + PostgreSQL (jackc/pgx), both pure-Go. open -> Connection, query / exec, pull cursor + typed as* accessors, begin / commit / rollback, prepared statements. Values bind only through placeholders. The first heavyweight library-layer dependency (design-decisions.md); build-tag split, default binary only.
M20.10	crypto AE + signatures	AES-256-GCM encrypt / decrypt (32-byte key, nonce prepended, AEAD-only) + Ed25519 signKeypair / sign / verify (crypto.Keypair); added sha384 to hash. All Go stdlib, TinyGo-clean. Safe-by-construction: AEAD-only, one algorithm per verb, length-validated keys, internal nonce. Out: password hashing, raw block modes, RSA / ECDSA, x509, AAD.

Cross-cutting additions these pulled into the language / system side (each documented under its library):

- **defer / errdefer** (M20.7) - the deterministic-cleanup keywords; the `finally` rejection and the `printf` -for-translation rejection (M20.4) are recorded in rejected.md.
- **fs.sync** (M20.7) - fsync a write / append handle to the device, distinct from `close` 's reach-the-OS: durability you check, versus cleanup you defer.

- **sha384 / sha512** (M20.10) - the SHA-2 digests filled out across `hash.compute` / `hmac` / streaming; MD5 / SHA-1 stay checksum-only.
- **The nesting cap** (`internal/limits.MaxNestingDepth`) shared by the language parser and the `json` / `toml` / `xml` decoders - build-tag split so it stays below `jennifer-tiny` 's fixed-stack crash point (see technical/tinygo.md).
- **The SIGUSR1 diagnostics checkpoint hook** (M20.6) - the loop-iteration / call checkpoint the deferred cooperative loop-cancellation follow-on (interruptible loops, REPL Ctrl-C, terminating-signal terminal restore) builds on.

M21 - general backlog

The catch-all bucket: milestones that fit no other track - not a Jennifer-coded module (M18), not interpreter / tooling work (M19), not a Go system library (M20), and not a beyond-1.0.0 idea (those live in the horizon collection). Anything worth recording with no natural home landed here, and graduated out once a cluster grew big enough to earn its own bucket. Thirteen sub-milestones across Jennifer-coded modules, security hardening, interpreter / language work, byte throughput, and Windows packaging. Each shipped module carries the standard discipline (a 100%-passing `*_test.j` overlay, a `cmd/jennifer/*_test.go` integration test, `docs/modules/*.md` + catalog + `JENNIFER.md` entries, a runnable demo); per-item detail lives in those docs - this table is the milestone-number index.

M#	Topic	Summary
M21.1	screen module	Terminal UI. Output-only layer (both binaries): a value-semantic cell Buffer (text / textColor / box / fill / hline / vline), ANSI control builders, and a flicker-free render / diff paint loop (repaints only changed row-runs). Interactive layer (term, default binary): a pure decodeKey(seq) -> Key (printable / arrows / nav / F1-F12 / ctrl-* / alt-*) plus nextKey / begin / end / size over raw mode. 0-based coords; drawing past an edge is clipped. The curses / bubbletea subset.
M21.2	feed module	RSS 2.0 + Atom 1.0 build and parse in one module (format chosen on build, sniffed on parse); a value-semantic Feed of Entry plus fetch(url). Rides xml (parse / escaped build) and time (RFC 822 / 3339 dates). Hardened for untrusted feeds: xml's nesting cap + no-custom-entity decode, lenient dates, and a new 64 MiB http body cap closing an unbounded-body OOM.
M21.3	jwt module	JWT (RFC 7519) sign / verify / decode / header, claims a json.Value; ten algorithms (HS* / RS* / ES* / EdDSA). verify pins the expected alg (blocks algorithm-confusion), enforces exp / nbf, HMAC-compares constant-time. Brought M20.10's deferred asymmetric crypto due: extends crypto with rsaSign / rsaVerify / ecdsaSign / ecdsaVerify over PEM keys (build-tag split - RS* / ES* need the default binary; HS* / EdDSA on both).
M21.4	acme module	ACME (RFC 8555) client over http + json: directory / account / order / authorization / challenge (HTTP-01 keyAuthorization + DNS-01 dnsRecord math) / accept / finalize(CSR) / downloadCertificate. Per-request JWS + a fresh anti-replay nonce; CA problem docs surface as a catchable Error. Second crypto expansion: rsaGenerateKey / ecGenerateKey, jwkPublic (RFC 7638 thumbprint), csr (PKCS#10). Default binary only.
M21.5	orm module	Data Mapper over sql (value-semantic, method-less structs rule out Active Record): repository CRUD keyed off an explicit orm.Schema (no reflection), a non-mutating functional query builder (from / where / orderBy / limit / join -> toSql) with per-dialect placeholders (mysql / postgres backend selector, injection-safe), records as map of string to string, plus a createTable DDL emitter. Enabling sql change: toDriverArgs spreads a single list argument (Jennifer has no spread).
M21.6	font module	Pure-.j TrueType / SFNT parser (no Go - bytes + bitwise + fs, both binaries): parse / open -> Font, then unitsPerEm / name / advance / glyphPath (SVG d) / glyph (contours). Parses head / cmap (fmt 4 + 12) / maxp / hhea / hmtx / loca / glyf (simple and composite) / name. Closed the dogfood gap: scripts/genwordmark.j reproduces the wordmark byte-for-byte without fontTools.

M21.7	injection / DoS hardening	Real wire-boundary fixes, each with a regression test: CRLF / control rejection in smtp (envelope + EHLO), http (method), websocket (handshake URL), web (cookie Path / Domain); jwt.verify rejects an unsupported crit; acme control-char escaping + validated nonce; websocket 64 MiB message cap (refused before allocation, 1009 close). Shipped the security model (SECURITY.md + security-model.md): full host access is by design, untrusted data on a wire is the bug class, the untrusted-code sandbox is DRAFT#11.
M21.8	call-depth limit + profiler metric	Deep recursion that overflowed the Go stack (fatal crash / jennifer-tiny segfault) now raises a catchable "call stack too deep" - Python's RecursionError analogue. The counter lives on the per-goroutine root env (spawn-safe); the cap is build-tag split in internal/limits (10000 default / 48 tiny, whose stack rose 2 -> 4 MiB). Feeds a jennifer profile max-call-depth metric.
M21.9	network + credential hardening	Sensible-default / optional checks, one regression test each: an optional timeoutMs on net.connect / connectTLS / startTLS, fs.chmod / chown, web Secure cookies, smtp no-cleartext-auth + anti-downgrade STARTTLS check + envelope validation, oauth 0600 token file, jwt.verifyWith (iss / aud), 64 MiB received-data caps across the network clients, amqp AMQPS.
M21.10	byte-oriented throughput	The binary library (concat / slice / indexOf / contains / split / startsWith / endsWith - the byte counterpart to strings, value-semantic, TinyGo-clean) plus net.readAll / readN (bulk reads with catchable size / close-mid-frame caps). Reworked http / mqtt / imap onto bulk reads; a binary.indexOf benchmark fixture.
M21.11	range syntax (..)	Half-open lo..hi (int bounds), three materializing / value-semantic uses: list construction (0..n), lazy for-each (for i in 0..n, no list built), and slicing ($xs[a..b] + open forms, over list / bytes / string). lo > hi a positioned error; materialisation bounded by a catchable limits.MaxRangeElements (int64-span-overflow-safe, not the uncatchable makeslice panic). New RangeExpr / SliceExpr AST; fmt emits .. tight.
M21.12	per-frame allocation elimination	A call / block frame now does no per-binding or per-call heap allocation: a slot-backed binding (Slot >= 0) writes only the pooled slots slice (the identifier travels in Binding.Name; the rare name-based readers scan the small slot slice via lookupLocal), and evalCall binds args interleaved with no intermediate []Value. Recursive fib ~300k -> ~59 allocs/op; Go's minor page faults fell ~7x. Value semantics and the vars fallback (REPL) intact; guarded by TestFrameAllocationsStayLow.
M21.13	Windows installer	An Inno Setup script (packaging/windows/jennifer.iss) built by a windows-latest CI job into jennifer-<ver>-setup.exe: per-user (no admin), adds to PATH, bundles the system modules + sets JENNIFER_SYSMODDIR, opt-in .j association, Apps & Features uninstaller. Unsigned, best-effort unsupported build. scripts/build-windows-installer.sh recreates it locally via Wine. Promoting Windows to supported is the follow-on, M28.1.

Cross-cutting threads:

- **The crypto asymmetric surface** grew here, not in M20: M21.3 added `rsaSign` / `rsaVerify` / `ecdsaSign` / `ecdsaVerify` (PEM; PKCS#1 v1.5 / JOSE R||S) for JWT interop, and M21.4 added `rsaGenerateKey` / `ecGenerateKey` / `jwkPublic` / `csr` for ACME - all build-tag split off the TinyGo build (`crypto/x509` is absent there), so the asymmetric verbs are default-binary only while the symmetric primitives and Ed25519 stay on both.
- **A 64 MiB received-data cap** became the standard DoS guard across the network clients (`http` / `redis` / `pop` / `imap` / `mqtt` / `websocket` , in M21.2 / M21.7 / M21.9), so an attacker-declared length or an unbounded stream fails catchably instead of OOMing.
- **Not all of M21 is modules.** M21.8 / M21.11 / M21.12 are interpreter and language work (the call-depth guard, range syntax, the allocation model), M21.10 is the throughput library, and M21.13 is packaging - which is exactly why this bucket is the general catch-all rather than an M18 module run.
- **The sql list-spread** (M21.5) - `toDriverArgs` spreading a single `list` argument into the placeholder sequence - is the one small language-adjacent enabler this track pulled in, for runtime-shaped parameterized queries.

M22 - additional libraries and language refinements

Post-backlog work belonging to neither the M20 system-library set nor the M21 catch-all: focused standard-library and `.j` -module additions or enhancements, plus small language cleanups, each landing when the need was concrete. Eighteen sub-milestones. Each shipped with the standard discipline (a library adds a Go package + `internal/stdlib.InstallAll` line + `docs/libraries/` reference + cheatsheet rows; a `.j` module ships a 100%-passing `*_test.j` overlay + a `cmd/jennifer/*_test.go` integration test + `docs/modules/` doc + catalog + `JENNIFER.md` bullet + a demo; a language feature updates the spec + grammar EBNF/PEG + the editor highlighters). Per-item surface detail lives in those docs; this table is the milestone-number index.

M#	Topic	Summary
M22.1	path library	OS-aware path manipulation over Go path/filepath, the pure-string counterpart to fs's I/O: base / dir / ext / stem / join (variadic) / clean / isAbs / split. Manipulation subset only (no disk-touching Abs / Glob / Walk), so no build-tag split, TinyGo-clean, both binaries. Host separator (portable). Explicitly not a filename sanitizer.
M22.2	digits in identifiers	Relaxed letters-only to letter-initial [A-Za-z][A-Za-z0-9]* (still no _, <= 64); constants [A-Z][A-Z0-9]*(_[A-Z][A-Z0-9]*)* (digits within a chunk). Additive / non-breaking - so sha256 / SHA256 / HTTP2 / SCRAM_SHA256 legal, AES_256 still illegal (write AES256). One IDENT token, so it applies uniformly (vars / params / methods / struct type and field names / use as / import as aliases); constants the one separately-lexed class. Graduated DRAFT#20; unlocks the M22.3 renames.
M22.3	library renames	The breaking renames the digit rule unlocked, one pre-1.0 batch (no deprecation window): use iic; -> use i2c;; uuid.generate("v4"/"v7") -> uuid.v4() / v7() (generate removed); crypto.pbkdf -> pbkdf2. Not renamed: binary (the bytes keyword, not a digit), intl (JS Intl). Each batch updated the library, its overlay + Go test, docs, cheatsheet, and every caller. Rejected (stance #1): per-algorithm digest shortcuts (hash.sha256 / crc.crc32) - the hash / crypto family is irreducibly algorithm-as-value (SCRAM / JWT / TLS negotiate the hash), so compute(b, algo) stays canonical (rejected.md).
M22.4	match statement	Multi-way value dispatch match (EXPR) { when V [, V ...] { } ... else { } }: subject evaluated once, strict == (Value.Equal), first match wins, values short-circuit left-to-right, no fall-through, not a break target (break / continue act on the enclosing loop), optional else last, no-match-no-else is a no-op; a statement, each arm its own scope. Keywords match / when (pre-1.0 break). fmt lays arms out flat (each when / else on its own line, not cuddled). New MatchStmt; the header { ambiguity (when Name { vs a Name{...} literal) is resolved by the parser's noStructLit flag. Designed to grow into M22.5 patterns.
M22.5	sum types (enums) + pattern match	def enum Name { Variant [{ field as type, ... }], ... }; (top-level, hoisted like def struct); Name.Variant{...} / Name.Variant construction; match gains variant patterns when Variant(bind) { } binding the payload into a fresh per-arm scope, with exhaustiveness checked at resolve time for a local / same-module enum subject. New Value KindEnum mirroring KindStruct (value semantics, deep-const, cross-module identity by canonical path, retag), tagged-union / reflect-free. Case-agnostic naming - Prefix.Member resolved from the tables at eval, not capitalisation, so no PascalCase rule is forced on a teaching language (only the pre-existing "ALL-CAPS is a constant" rule still applies). Graduated DRAFT#19.

M22.6	TLS options for http / rest	Reach an HTTPS host with a self-signed / private-CA cert. http.TlsOptions{skipVerify, caCert} + send variants http.requestTls / requestWithTls; rest.Client.tls field + a rest.client(baseUrl) constructor (the added required field breaks the bare literal) + rest.withCA(c, pem) (preferred) / rest.insecure(c). Secure by default (verification stays on unless explicitly relaxed). Pure .j plumbing to net.connectTLS (M16.14) - no interpreter or system-library change. Verified end-to-end against a self-signed loopback (http_tls_test.go). Dogfoods M22.9 (http.TlsOptions as a struct field across main -> rest -> http).
M22.7	graphql client module	Thin GraphQL client over http / rest: client(endpoint) + bearer / basic / header / withCA / insecure builders, query(c, query, variables) -> json.Value (POST {query, variables}; result under /data). Gets the GraphQL convention right - a non-empty top-level errors array is an HTTP 200, not a non-2xx - raising a graphql error with the joined messages; a non-2xx also raises. queryNamed / tryQueryNamed add an operationName; tryQuery / tryQueryNamed return the raw envelope (no raise on GraphQL errors) for structured-error handling via exported hasErrors / errorMessages + json accessors. POSTs the endpoint verbatim via http.requestTls (rest's joinUrl would append a trailing slash). The GraphQL dependency the M27 Unraid deck consumes.
M22.8	self-referential struct guard	A struct containing itself by value (direct or mutual) has no finite zero value and used to fatally stack-overflow when its zero / a literal was built; Interpreter.checkStructCycles (a gray/black DFS over direct struct-typed fields, run after hoisting at both Run and EvalInteractive) now rejects it at hoist time with a positioned error pointing at list of Self. Recursion through a list / map / task field and ordinary nesting stay legal.
M22.9	module structs as struct fields	A module struct used as a struct field type now type-checks (was rejected "expects geo.Point, got struct" though it worked as a variable type). resolveDeclaredTypesOnce now also stamps struct field types with the module's (stem, path) identity (recursing into list / map elements), and a module struct's own sibling-struct field types retag to the module identity at the boundary check (construction + field assignment, via retagType). Value semantics + chained lvalues into a nested module-struct field work.
M22.10	byte-capable http download	http could not fetch a binary body (the response was always convert.stringFromBytes(_, "utf-8"), which throws on non-UTF-8). Added a byte path reusing the already byte-exact framing: http.BytesResponse (body as bytes) + requestBytes / requestWithBytes / getBytes (parseResponse split into a parseRaw byte core + a text decoder). Text verbs unchanged (still throw on non-UTF-8, by design); rest stays text / JSON. Pinned by http_bytes_test.go (a gzip round-trips with matching sha256).
M22.11	hardening: injection & output-encoding	From two security / robustness audits (internal/ + modules/; per-finding detail - severity, reproducer, sites - in the two report files). orm identifier / operator allowlists; imap / pop CRLF rejection; statsd metric validation; htmlwriter tag / attr checks + exported safeUrl; json.encode HTML-escaping (< > & U+2028 U+2029); a new constant-time hash.equal; http.parseUrl authority split; dotenv env-name validation (OM-021); ipnet v4-mapped ::ffff:0:0/96 fold (unmap). Reasoned non-literal: a tengine no-auto-escape SECURITY warning, not an auto-escape mode (OM-012, keeps text/template semantics); OM-010 folds only the well-defined ::ffff:0:0/96 mapped form.
M22.12	hardening: network / resource / path robustness	64 MiB caps on server-declared lengths; connect / read timeouts + cleartext warnings; a net.readAll default cap; net.startTLS locking; bounded handle registries + a sql query deadline + DSN-password redaction; archive per-entry budget; httpd.serveDir traversal rejection + unix:-socket perms; the json / toml / yaml write-API depth guard. Reasoned non-literal: OF-007 applied full bounds + deadline + teardown to sql (the worst instance) and registry bounds to fs / os / compress / net.

M22.13	hardening: web framework	web.sessionId trusts only a minted-UUID-shaped cookie + web.renewSession; the web.onError hook (else stderr); lenient form / percent decoding + csrf content-type gate; a HEAD served by the matching GET route; httpd per-request timer cleanup. Reasoned non-literal: OM-003 shipped serial for v1 with loud docs because concurrent dispatch raced shared interpreter state - later removed properly in M22.17; OF-006 (unrestricted meta.call) shipped a docs allowlist pattern + meta.md example, not a new primitive (a two-line .j allowlist suffices; web only dispatches author-registered handler names).
M22.14	imap criteria-based search	imap.search(session) -> imap.search(session, criteria) (breaking; an empty imap.criteria() = the old SEARCH ALL). imap.Criteria filters hybrid: server-side fields map to one IMAP SEARCH (substring on subject/from/to/text, a since / before day-range as time.Time, flags, size - all ANDed, one round-trip), client-side fields refine the candidates by fetching only headers / BODYSTRUCTURE (subjectRegex / fromRegex, hasAttachments heuristic). Injection-safe (quoteArg + control-checked line); a time-of-day bound is transparently refined client-side against each candidate's INTERNALDATE. Pure .j.
M22.15	imap browse / APPEND / rename	Rounded imap into a full read / browse / manage / save client: folders (LIST) + status (STATUS, no select), append / appendWith (APPEND via the synchronizing-literal continuation flow). Terminology rename (pre-1.0 breaking): mailbox -> folder (selectFolder / createFolder / the Folder struct); the LIST verb is folders (list is a reserved type keyword).
M22.16	core hardening sweep	Small internal/ correctness / resource / performance residuals, one reviewed pass: sql cursor deadline split from the acquire / statement deadline (caller-settable sql.setQueryTimeout); httpd registry maxServers bound + catchable "too many open"; md5 / sha1 labelled non-cryptographic; the json / toml / yaml write-depth guard checks only the touched node (not a full re-scan) and folds three exceedsDepth copies into a shared helper; an include total-token cap (a diamond that re-includes the previous file expands 2^n); the printf field cap lowered from 1<<20 + closed-form padding; incremental json / xml decoder error positions.
M22.17	web hardening (concurrent dispatch)	Turned web from strictly-serial into safely-concurrent, interpreter-first (four ordered steps, each with a gating test; reasoning in design-decisions.md). (1) Error crosses meta.callMain intact - track a module's declared structs separately from the auto-injected Error, retag only the former. (2) Race-safe dispatch - the call-depth counter became a per-chain *int threaded down the frames (fresh at each goroutine root, incremented at evalCall and every cross-boundary dispatch), fixing both the -race data race and a fatal re-entry Go-stack overflow; the rest of the reachable shared state (map hash-index reads, resolver caches, profiler, diag) audited clean. (3) spawn-per-request in web (errors caught inside; task.discard prunes; concurrency bounded by httpd). (4) web.onError fail-safe - always stderr and the hook, which now binds as Error. The 1700 ms -> 3 ms latency probe is the regression gate. Supersedes the M22.13 OM-003 "web stays serial" note.
M22.18	dotenv layering, profiles, interpolation	Grew dotenv into a layered loader: readCascade / resolve / loadCascade / autoload merge .env -> .env.local -> .env.<profile> -> .env.<profile>.local from one fixed dir (no walk-up, closing the file-hijack class), with a real OS env var always winning over a file value; profile from JENNIFER_ENV (empty = base files only). Enhanced parse: backward-reference ${VAR} (unquoted + double-quoted, resolving earlier keys -> real OS env -> "", so cycles are impossible; no $(...) command substitution), multi-line double-quoted values (positioned unterminated-quote error). Strict profile validation ^[A-Za-z0-9_-]{1,64}$ (no traversal). Single-file load keeps unconditional-override (the primitive); the cascade loaders are real-env-wins. os.getEnv(k) != "" is the "already set" test (no os.hasEnv; "" counts as unset). Consolidated the ${VAR} / multi-line items parked in M23.8.

M23 - module improvements

Done. M22 lifted a handful of modules (`imap` , `http`); M23 generalized that across the module ecosystem. A survey of all 63 `.j` modules found the gaps **cluster into cross-cutting themes** rather than scattering per

module, so the work was organized by theme - a shared pattern (a receive loop, a persistent connection, a backend selector) built once and applied to every module that needs it. Fifteen sub-milestones. Each shipped the standard per-module discipline (a 100%-passing `*_test.j` overlay, a `cmd/jennifer/*_test.go` integration test where a live server applies, `docs/modules/` + catalog + `JENNIFER.md` entries, a demo, both binaries build); per-item surface detail lives in those docs, and the biggest sub-milestones keep their own reference docs. This table is the milestone-number index. **Deliberate non-goals:** `password` hashing (needs `x/crypto`) and fully-typed `orm` rows (awaits struct reflection).

M#	Topic	Summary
M23.1	streaming / server-push read loops	One cooperative receive-loop-over-net shape - a blocking receive* (+ timeout-bounded poll), no callbacks, the app opting into concurrency via spawn - across redis (pub/sub + one-round-trip pipeline + multi/exec + scan), amqp (Basic.Consume + exchanges + publisher confirms), mqtt (QoS-1 + retained + Last-Will + reconnect), mikrotik (.tag-correlated + /listen), imap (RFC 2177 IDLE). Wire framing factored into pure encode/parse (100% overlay); live loops on mock-server Go tests. Fixed redis coalesced-frame buffering and imap stale-deadline. Residual: a value-semantic Session can't retain a cross-call buffer (a buffered net reader is the general fix).
M23.2	connection reuse / persistent sessions	Reusable connections so a loop stops re-handshaking: http.Session (reused net.Conn + cookie jar, framed one-response reader) + a policy http.send over http.Options (3xx redirects, cookie jar, 429/5xx retry+backoff); rest.Client routes through it (tls folded into Options, pre-1.0 break; paginate / paginateCursor walk every page); smtp split into open / sendOn / close so N messages pay one TLS+auth handshake. Pinned by keep-alive / cookie / pagination / session-reuse Go tests.
M23.3	stable-identity verbs	Volatile sequence numbers broke "fetch only what's new": imap went UID-only (every verb sends its UID form + search returns UIDs + atomic move + ranged fetchPartial; pre-1.0 break, a seq+UID twin set rejected on stance #1), pop added uidl -> MessageId + top / reset / noop (additive). Pinned by exact-wire-command fake-server tests.
M23.4	byte-exact binary values	redis / memcache threw on a non-UTF-8 bulk value; added bytes-valued setBytes / getBytes (text verbs unchanged, still strict-throw) plus typed redis hash/list/set helpers and memcache getMulti / gets / cas. NUL/CR/LF/0xFF byte-count-framed round-trip tests.
M23.5	selectable backends	session / ratelimit were memcache-only; added the kv system library (in-process per-key-TTL store, integer handle shared across spawn, openFile persisted) and the kvstore module selector (Store a sum-type enum Memcache/Redis/Local, exhaustiveness-matched), and moved session (values now a json.Value) and ratelimit (fixedWindow / slidingWindow -> Result) onto it (both pre-1.0 break). Interpreter fix: a module enum as a struct-field type across a boundary.
M23.6	format & coverage completeness	The broadest track (13 pieces) - the deepest per-module gaps. ipnet subnet math + scope classifier; orm ordinary-query surface (select / aggregate / join / groupBy+having, render-time allowlist re-check); markdown images / blockquotes / nested lists; vcard full N / TYPE; ical recurrence + TZID + VTODO/VALARM; mime charset-on-decode + RFC 2231; feed enclosures + author/categories; s3 presign + byte bodies + multipart; barcode UPC / code93 / DataMatrix + QR 11-40; font a CFF/OTTO backend + an O(1)-per-query fix; pdfwriter embedded TrueType fonts + image XObjects + text layout. Several pre-1.0 struct-shape breaks; each validated against an independent reference.
M23.7	observability completeness	prometheus histogram + summary types (cumulative buckets, nearest-rank quantiles, exact text format, observeAt / pushgatewayPath); statsd *Rate / *Tagged / *Float verbs + a Batch datagram packer + a control-char-validated prefix; influxdb 2.x / 3.x via a Version enum (Flux queryFlux, token redaction). Each validated against an independent reference parser.

M23.8	ergonomic papercuts + notifier richness	Cheap high-value wins across 15 modules: log child logger + fatal; cron named months/weekdays + @-macros; jwt verifyLeeway / verifyWithKeys / verifyJwks; totp generateSecret / hotp / verifyWindow; bloom optimal / serialize; csv formatSafe + Dialect + streaming; richer discord / telegram / slack / gotify messages; webhook replay-protected signing; and more. Core fall-out: http.requestRawBody / requestRawBodyTls (a byte request body, for telegram uploads).
M23.9	fmt: shape-aware wrapping + raw-literal fidelity	Rebuilt jennifer fmt (token-stream, no AST): width-aware wrapping (one element/arg per line past 100 cols, struct/map over 6 members, calls hug), inline single-statement when arms, operator-chain fill-break), tight Go-style literal spacing, and raw-literal fidelity via a lexer Token.Raw (digit separators / base prefix / quote style / embedded newlines survive verbatim). -w is atomic + self-verifying (re-lexes its output, refuses to corrupt). Corpus reflowed (~700 -> ~215 over-limit lines). Pre-1.0 break (canonical output changed); TestFmtPreservesTokenStream proves a format changes only whitespace.
M23.10	interactive stdin for os.run / os.spawn	os.run(argv, stdin) feeds an optional trailing string / bytes to the child's stdin then closes it (variadic - one-arg calls unchanged), output drained into the 16 MiB-capped buffers - deadlock-free by construction. Unblocks a stateless filter / subprocess exchange (M23.13's mcp.connectStdio built on it). Deferred: interactive streaming pipes on a spawned Process.
M23.11	jsonrpc module	JSON-RPC 2.0 client + server, pure .j over json + http. Client call / notify (json.Value params/results, every failure a unified catchable Error); a transport-agnostic handle(requestBody) runs the whole protocol (single / notification / batch / reserved codes) dispatching each method to a top-level func by name via meta.callMain; a thrown handler yields a generic -32603 (detail stays server-side). Chosen over gRPC (protobuf + HTTP/2 + codegen would be a heavy Go library, not a .j module).
M23.12	match / enum adoption	Applied match / enum to the genuine closed-variant-set cases (open sets stayed strings): non-breaking match in 14 modules; a non-breaking private enum (markdown); breaking API enums (htmlwriter.NodeKind, prometheus.MetricType, barcode.SymbolKind, orm.Dialect / ColumnKind, totp.Algorithm); and a shared transport.Security{None,Tls,Starttls} replacing the stringly-typed security field on six socket clients (the mail three accept all modes, the other three reject Starttls). Exhaustiveness-checked.
M23.13	mcp module (Model Context Protocol, stateless)	A server exposing tools / resources / prompts (server / addTool / addResource / addPrompt, handle / serveStdio) plus a client (HTTP connect / stdio connectStdio over os.run) sharing one call surface; MCP is JSON-RPC 2.0 so the client reuses jsonrpc.call. tools/call is allow-listed (only a registered handler; a thrown handler yields a generic message). Validated end-to-end against the official MCP Python SDK. Not planned: the stateful Streamable-HTTP transport (needs an SSE push primitive httpd lacks).
M23.14	raw single-quoted string literals	Breaking split so each delimiter does one job: "..." stays cooked (escapes processed), '...' becomes raw (verbatim to the next ', spanning newlines - a free heredoc); embed a ' via the cooked form ("it's"). No r"..." prefix (rejected - the delimiter is the mode). Lexer-confined (raw := quote == '\''); src is []rune so multibyte content is exact; Token.Raw round-trips (incl. multi-line). Migration a no-op; docs / grammar / four editor highlighters updated.
M23.15	orm: a batteries-included data-mapper ORM	Lifted orm from a thin query builder to a full ORM (still Data Mapper): a Session unit-of-work + column-attribute DDL builders (.1); migrations split into the sibling sqlmigrate module (.1b); associations + joinRelation (.2); eager loading in a fixed 1+R queries (.3); a write path - upsert / insertMany / insertReturning / updateWhere / deleteWhere / save (.4); finders + whereNull / whereBetween / distinct / page (.5). Rows stay map of string to string (no reflection), relations attach via a side Result (no any); whereRaw / typed-struct mapping / Active Record rejected. Keeps its own docs/modules/orm.md + sqlmigrate.md.

Cross-cutting threads:

- **Not all of M23 is modules.** M23.9 (`fmt`), M23.10 (`os.run` stdin), and M23.14 (raw single-quoted strings) are tooling / interpreter / language work; the rest is the module ecosystem.
- **Build once, apply everywhere** was the payoff of organizing by theme: a shared receive-loop shape (M23.1) and a shared backend selector (the `kv` library + `kvstore` enum, M23.5), each built once and threaded through every module that needs it.
- **match / enum** (M22.4 / M22.5) landed across the ecosystem in M23.12, and the largest sub-milestone - M23.15's `orm` + the new `sqlmigrate` - is a full batteries-included ORM in its own right.

M24 - language, concurrency, and libraries

Done. The first batch of horizon drafts graduated into a scheduled track, then grew into the broadest single milestone: a CLI ergonomic, the language's biggest expressiveness feature (first-class functions), a concurrency-coordination layer, a numeric / ML library stack, two ASN.1 protocol clients, and a run of library / language refinements. Twenty-three sub-milestones. Each shipped with the standard discipline - spec + grammar EBNF/PEG + editor highlighters where a language feature lands; a Go package + `internal/stdlib.InstallAll` line + `docs/libraries/` reference + cheatsheet where a library lands; a 100%-passing `*_test.j` overlay + `cmd/jennifer/*_test.go` + `docs/modules/` + catalog + `JENNIFER.md` + demo where a `.j` module lands; both binaries build; the full test close-out. Per-item surface detail lives in those docs - this table is the milestone-number index.

M#	Topic	Summary
M24.1	run profiles / --env flag	jennifer run --env=prod script.j sets JENNIFER_ENV=prod before Run - identical to the env var, no interpreter change (dotenv's .env.<profile> selection is the first consumer). cmd/jennifer-only: parseRunArgs + a validRunProfile allowlist ([A-Za-z0-9_-], 1-64, blocks a .env.<profile> traversal); an explicit flag overrides an inherited env. Both binaries. Graduated DRAFT#26.
M24.2	first-class functions	Closed the largest expressiveness gap - a function held in a value (type func), immutable, not a pointer (so the value-semantics stance is untouched); a bare method name in expression position is the value, a name + (is a call (no &NAME sigil). New KindFunc / TypeFunc; a CallValueExpr postfix-(node calls any function-valued expression ($f(x), $fns0 (x), makeAdder(1)(2)). Enables the higher-order lists layer (map / filter / reduce / find / any / all / sortBy, via BuiltinCtx.Invoke). Deferred: Tier 2 closures + migrating string-name dispatch. Graduated DRAFT#18.
M24.3	concurrency coordination	Gave spawn / task cancellation, bounded waits, and channels. task.cancel / cancelled - cooperative, observed at a shared loopCheckpoint (a catchable "task cancelled" at loop safe points; one atomic-nil check per iteration, hot path untouched); waitTimeout / waitAnyTimeout throw on timeout. channel of T + the channel library (make / send / recv / close / select / len / capacity): a KindChannel shares a *ChannelState while send deep-copies the value in (conduit shared, data copied), and channel is a contextual keyword. Buffer capacity capped (OOM guard); send-on-closed / double-close catchable. Graduated DRAFT#21. Deferred: cancellable channel ops, index-returning select.
M24.4	stats library	26 descriptive statistics over list of int / list of float - central tendency (geo / harmonic / weighted means, modes), spread (population + sample* Bessel, iqr, mad), shape (skewness / excess kurtosis), order (percentile / quartiles), sum / zscore, bivariate (correlation / covariance), describe -> stats.Summary. Strict like math (an undefined result is a catchable error, never a NaN); real reductions -> float, selections preserve the input kind. Pure Go, TinyGo-clean. Graduated DRAFT#5.

M24.5	ml library	A classical / predictive ML core (scikit-learn-lite) over stats / linalg, not deep learning. Fit/predict: a fit fn -> an opaque ml.Model handle applied with predict / transform / predictProba / free. Regression (linear / ridge / lasso / kNN / tree / forest), classifiers (kNN / naiveBayes / logistic / CART tree / forest), kMeans, pca, scalers; metrics (accuracy / precision / recall / f1 / rocAuc / rmse / r2 / ...); trainTestSplit / kFold / polynomialFeatures. Random draws from math's seedable source; hyper-parameters bounded (a runaway value -> catchable). Pure Go, both binaries.
M24.6	linalg library	Linear algebra over Jennifer's value types, the stats companion. Vectors (list of float): dot / distance / cross / normalize; matrices (list of list of float): transpose / trace / determinant / inverse / solve / identity / zeros / shape; polymorphic norm / scale / add / sub; matmul dispatches on operand shape. Direct algorithms (Gaussian / Gauss-Jordan), no gonum. Strict (dimension mismatch / singular / non-finite -> catchable); every value bounded by MaxMatrixElements. Pure Go, both binaries.
M24.7	asn1 library	ASN.1 BER decode / DER encode, the byte enabler for LDAP / SNMP. Hand-rolled (Go's encoding/asn1 is DER-only + reflect-bound). An opaque asn1.Value element tree (like json / xml) walked by (node, pointer) accessors whose tokens are child indices (tagClass / tagNumber / asInt / asOid / ...); typed constructors (integer / oid / sequence / tagged EXPLICIT / retag IMPLICIT). A decode-node budget + nesting cap turn a bomb into a catchable error; output byte-verified against Go's encoding/asn1. Pure Go, both binaries.
M24.8	snmp client + agent	SNMP v1 / v2c client and agent (modules/snmp.j) over asn1 + net UDP. Client get / getNext / set / walk -> list of Varbind typed by SNMP type (counter32 / gauge32 / timeTicks via asn1.retag); request-id check, deadline + retries. Agent agent / serve answers GET / GETNEXT / SET for a MIB (GETNEXT in numeric OID order so a walk traverses it). Codec factored pure (socketless overlay); live loopback + real-hardware verified. No v3 / traps / GETBULK. Default binary only.
M24.9	ldap client + directory server	LDAP v3 (RFC 4511) client and lightweight directory server over asn1 BER + net, LDAPS / StartTLS via transport.Security. Client bind / SASL bindSasl, search (RFC 4515 parseFilter or constructors) / searchPaged, writes add / modify / delete / modifyDn / passwordModify. Server answers simple bind (password schemes) + filtered search - read-only over the wire, mutable from code via a shared kv store (in-memory or file-backed), enough to back Authelia. Codec factored pure. Default binary only.
M24.10	math foundations + special fns	Filled the math gaps for 1.0, folded in. Everyday: trig (+ inverses, atan2), hyperbolic, exp / log (exp / ln / log2 / arbitrary-base log), cbrt / hypot / sign, combinatorics (factorial / comb / perm / gcd / lcm, exact-int overflow-checked), TAU. Special: erf / gamma / lgamma / beta + the regularized incomplete gamma (regGammaP / regGammaQ) / beta (regBetaI) the CDF engine needs, hand-rolled to machine precision. Strict (domain / overflow / non-finite -> catchable). Go stdlib base; both binaries.
M24.11	distributions + inferential stats	The classical numerical-inference layer (the scipy.stats / Excel surface) folded into stats on M24.10's specials - no separate prob library. Distributions (normal / t / chi-square / F / binomial / Poisson): pdf/pmf, cdf, quantile (Acklam + bisection), Box-Muller normalSample. Inference: linearRegression / multipleRegression, confidenceInterval, proportionCi (Wald / Wilson / Clopper-Pearson), tTest / chiSquareTest / fTest / anova, histogram -> Regression / Interval / Test structs. Every p-value finite-guarded, clamped [0, 1]; a degenerate input -> catchable. Pinned to scipy; both binaries.
M24.12	scientific-notation float literals	Float literals accept an [eE][+-]?digits exponent (6.022e23, 1e10 - the exponent makes a literal a float even with no fraction), no _ in the exponent, decimal-only (0xe5 keeps e a hex digit). Lexer-only, additive / non-breaking. Strict at the edge: overflow (1e400) a positioned parse error (never Inf); underflow (1e-400) rounds to a finite 0.0. Reasoned in design-decisions.md.

M24.13	uri module + encoding codecs	URL handling factored - byte encoding in encoding, URL semantics in a .j module. encoding gains "uri-percent" (RFC 3986) + "uri-form" (form-urlencoded, space +) toText / fromText codecs. The uri module (pure .j, both binaries): parse -> Uri / build (round-trips), encode / decode / encodeForm / decodeForm, buildQuery / parseQuery, RFC 3986 resolve(base, ref). Seven modules de-duplicated onto them. Pre-1.0 break: rest query strings now form-encode a space as +.
M24.14	args CLI parser module	A declarative argparse-style parser (the "write your tools in Jennifer" gap os.flag / os.ARGS left). A value-semantic Parser: typed optional flags (flag / intFlag / boolFlag / countFlag -vvv / listFlag), nargs positionals, subcommands, version; args.parse($p, os.ARGS) -> a Result with typed accessors. Normalises --flag=value, bundled shorts, -- end-of-flags; an error is a catchable Error{kind:"args"} (not exit(2)), -h / --version set done. Pure .j, both binaries. 32-test overlay.
M24.15	validate module	Declarative validation of a map of string to string -> a structured failure list. Value-semantic rules (required / isInt / min / maxLen / pattern / email / url / datetime / oneOf / password / custom func predicate + withMessage), grouped per field; validate.check -> list of Failure, plus ok / messages / byField / localize (i18n %param% markers). An absent / blank field passes every rule but required. Pure .j over regex / uri / time / password; both binaries. 16-test overlay.
M24.16	module suite hardening	A security / robustness pass whose audit findings collapsed to systemic root causes, each fixed once across its cluster (~35 fixes / ~24 modules, both binaries clean), remotely-exploitable first. Received-data caps (shared transport.checkReceiveSize); injection (label digits-only command class, CRLF / control stripping in log / ical / vcard, barcode colour); a cross-origin redirect credential leak (http.send drops Authorization / Cookie behind allowCrossOriginRedirect); typed error propagation; numeric clamps; correctness (semver.satisfies(v, garbage) -> false).
M24.17	html module (rebrand + parser)	Fold HTML building and parsing into one bare-named html. Rebrand htmlwriter -> html (pre-1.0 break; a format module is named for the format, like xml / json). A tolerant hand-rolled parse (pure .j, no x/net/html) producing the same transparent Node the writers do (build / parse round-trip): void / self-closing / unquoted-boolean attrs, mismatched-nesting auto-close, comments, DOCTYPE, raw script, entities; depth + node budget. Added parse / attrOf / hasAttr + XPath-ish get / findAll / has. Both binaries; 23-test overlay.
M24.18	markdown reader	Gave markdown reading (the consistency move M24.17 set up). markdown.parse(md) -> a public Node (a string kind, recursive over children) walked by the xml / html vocabulary (typeOf / children / text / level / attr / get / findAll / has); the private Block / Span model is converted, inline spans eagerly expanded (a link's href walkable), the fence language captured. One model: toHtml / toAnsi became render(parse(md), ...) wrappers (old renderers deleted; byte-identical output). Nesting cap + node budget. Pure .j, both binaries; +16 overlay tests.
M24.19	string interpolation	Cooked-string interpolation "total: {$sum}, up {strings.upper($x)}" - sugar over concat + convert.toString. Only a cooked "..." interpolates, each {expr} one expression (a statement / empty {} is a parse error); a raw '...' never (the off form); a literal brace is \{ / \}; f"..." rejected. Part 1 moved the two {name}-template consumers (intl.tr, validate) to a %name% marker. Part 2 added the lexer TOKEN_STRING_INTERP (Parts) + the parser sub-lex / sub-parse of each slot -> InterpStringExpr. The whole toolchain treats a slot as real code (resolver / lint L204 / profile); every literal-brace cooked string was migrated (prefer raw). Graduated the horizon entry; both binaries.

M24.20	module version + capability header	A per-file # pragma-jennifer-<key>: <value> header stating the minimum interpreter version + required host capabilities, checked at read time (the lightweight sibling of the M26 jvc / deck.toml constraints, no package manager). version (>=0.25.0, a min-compare via version.AtLeast; any dev build bypasses) + capability (net / exec / sql, from a build-tag-split set, queryable as meta.CAPABILITIES / hasCapability). Enforced per file at three first-read seams (CLI / include / loadModule), with bounded diagnostics; a malformed or duplicate-version directive is a hard error. Ships an L303 lint + fmt canonicalisation. Every module carries a >=0.24.0 floor; capability: only where mandatory (an optional backend stays ungated). No interpreter-core change.
M24.21	Markdown -> PDF (markdown.toPdf)	The markup-driven document story for pdf, folded into markdown as a third target beside toHtml / toAnsi: toPdf(md) / toPdfWith(md, opts) / renderPdf(doc, opts) lay a document out to a paginated PDF over pdf's layout primitives. A flow engine threads a value-semantic Layout through per-block renderers (heading, word-wrapped paragraph with per-run fonts, nested lists, ruled GFM table, code, blockquote), paginating at the bottom margin; PdfOptions sets size / margins / standard-14 fonts. Folded in (reasoned in design-decisions.md) at ~2x markdown import time / +~6 MB RSS; needed M24.18's parse tree. Graduated DRAFT#13. Pure .j, both binaries; +15 overlay tests. Dogfooded by gen-module-docs.j -> jennifer-module-api.pdf.
M24.22	dot module	Graphviz DOT graph description: build a graph of nodes and edges with attributes (digraph / graph, node / nodeWith, edge / edgeWith, graphAttr / nodeAttr / edgeAttr) and render it to .dot text for an external Graphviz tool to lay out (dot -Tsvg). Value-semantic builders; DOT-escaped strings; emits the description only (graph layout is Graphviz's job, deliberately not reimplemented). Pure .j over strings / lists, both binaries.
M24.23	plot module	Data plotting to SVG: a unified chart(series, opts) renders one or more Series (line / points / both / area, dashed, error bars, marker shapes) on shared axes with a positioned legend; line / scatter / bar / bars (grouped / stacked / diverging) / histogram wrap it. Automatic "nice" ticks, log scales, a time-labelled date axis, reference lines (hline / vline), fonts / margins, data labels, <title> hover tooltips, and save. The visual companion to stats / ml; pure .j over math / time / fs / strings / lists / convert, both binaries.

Cross-cutting threads:

- **Graduated horizon drafts.** M24 drained the first scheduled batch from the horizon collection: `DRAFT#26` (M24.1 `--env`), `DRAFT#18` (M24.2 first-class functions), `DRAFT#21` (M24.3 concurrency coordination), `DRAFT#5` (M24.4 `stats`), and `DRAFT#13` (M24.21 Markdown -> PDF), plus the string-interpolation entry (M24.19).
- **A numeric / statistical stack** landed together: `stats` (M24.4) + `linalg` (M24.6) + `ml` (M24.5) + the `math` special functions (M24.10) + the distributions / inference layer folded back into `stats` (M24.11) - each strict (an undefined result is a catchable error, never a NaN) and bounded by a materialised-value cap.
- **An ASN.1 protocol stack:** the hand-rolled `asn1` byte layer (M24.7) enabled the `snmp` (M24.8) and `ldap` (M24.9) clients-and-servers, each factoring its codec pure so the overlay round-trips without a socket.
- **Consistency across the format modules.** The `html` rebrand + parser (M24.17) and the `markdown` reader (M24.18) gave both modules the same build-and-parse `Node` model as `xml` / `json` , which M24.21's `toPdf` then consumed.
- **Not all of M24 is libraries.** M24.2 (first-class functions), M24.3 (cancellation / channels), M24.12 (scientific-notation literals), and M24.19 (string interpolation) are language / interpreter work; M24.1 is a CLI flag; M24.16 / M24.20 are a cross-suite hardening pass and a read-time guard.

M25 - read-only-parameter borrow (compound copy elision)

Parameter binding deep-copies compound arguments (`list` , `map` , struct) to uphold value semantics. When the callee **never writes the parameter** - no `$p = ...` , `$p[i] = ...` , `$p[] = ...` , `$p.f = ...` anywhere in its

body - that copy is pure waste: a read-only alias to the caller's backing is observationally identical to a copy, so the parameter can be bound by alias. This is the standard value-semantic-language optimization (the "borrow" half of copy-on-write), narrower and more clearly sound than the shared-marker COW that was tried and reverted as inert (see technical/rejected.md): it never detaches-on-write, because a borrowed parameter is by construction never written. No `.j` changes and no new syntax - a pure interpreter optimization. Split into the borrow itself (M25.1) and the escape analysis that widens where it is sound (M25.2). Independent of the horizon bytecode-VM direction but composes with it.

M25.1 - borrow in modules and globals-free scripts

Done. The borrow, enabled in the two contexts where it is sound without a whole-program analysis.

- **The soundness subtlety.** "Never writes the parameter" alone is *not* sufficient: if the argument aliases a mutable global and the body mutates that global, a borrowed read observes the mutation where a copy would not (`def g ...; func f(p) { $g[0] = 9; return $p[0]; } f($g)`). The synchronous call model rules out the *caller* changing the backing mid-call, but not the *callee* changing it through a global alias. Borrow is therefore enabled only where no mutable global can exist to alias: (a) a **module** , whose top level is declarations-only (and whose cross-boundary arguments are copies anyway); and (b) a **single-file script that declares no mutable top-level def** - checked once by `hasMutableTopLevelGlobal` , so a script's own read-only helpers benefit without being moved into a module. A script *with* a mutable global keeps the copying bind (M25.2 is what lifts that).
- **Mechanism.** Extends the existing **scalar** copy-elision in `bindParamValue` to compound Kinds - the "mutation-safety proof" that elision path was documented as requiring. `Resolve` runs `markBorrowableParams` per method: a statement-level write-scan of the body (through nested control flow; no-shadowing makes a written name unambiguously the parameter; `spawn` bodies mutate their own snapshot so are correctly skipped) sets `Param.Borrow` when the parameter is never written and its type is *borrow-safe* - a compound whose `stampDeclaredType` does not recurse into element backing (a flat `list` / `map` , or a struct/bytes; a `list of list` falls back to copy so the stamp can't mutate the shared backing). `Interpreter.bindArg` aliases when `Param.Borrow && (i.isModule || i.entryGlobalsImmutable)` , else copies as before.
- **What it buys.** It removes a class of accidental quadratics transparently. The `markdown` block collectors (`collectFence` / `collectQuote` / `collectList` / `tableFrom` , all `lines as list of string`) deep-copied the whole line list once per block (lines x blocks); parsing a 160-chapter synthetic book as one document drops from **5.1 s to 0.85 s** and turns from super-linear (2.9x per input doubling) to linear (2.0x). The `countNodes` budget walk borrows its `Node` argument, and a globals-free script's own read-only helpers go linear the same way. The `.j` -level workaround does not pay off: a count-during-construction rewrite of the `countNodes` guard measured a **wash** (9.5 s vs 9.6 s on a 100k-node document), because eliminating the read-only walk forces threading the count back through the builders, re-introducing an equal volume of copies. No reference-semantic handle a pure `.j` module can hang a large list on exists, so the copy is only removable below the language.
- **Discipline.** TinyGo-clean, reflect-free, strict behaviour parity, race-clean. `internal/parser/borrow_test.go` pins the write-scan and type gate (a false positive would alias a mutated parameter); `internal/interpreter/borrow_test.go` and `borrow_internal_test.go` pin the soundness gate (a script with a mutable global keeps copy semantics; a globals-free script and a module borrow) and the value-semantics parity (borrowed reads correct, caller values intact, returned borrows copied at the receiver).

M25.2 - per-function escape analysis for scripts with mutable globals

Done. M25.1 gates a whole script all-or-nothing on "does it declare any mutable top-level global". M25.2 adds a per-function decision on top: borrow a never-written parameter in any method proven not to mutate a global **transitively** , even in a script that has mutable globals elsewhere.

- **The analysis** (`computeEntryGlobalSafe` , run after module load so the module-alias and namespace tables exist). Each entry-program method is scanned for a **local hazard** , then a fixpoint propagates it along named-call edges (`CallExpr.Method`), so a method that reaches a hazardous method is itself

hazardous. A local hazard is: a write whose root is not a method-local name (a mutable global write - a `const` target is already rejected, so any non-local write hits a mutable global); a `CallValueExpr` (dynamic func-value dispatch); a **module** call (a module can re-enter the host via `meta.callMain` and reach a host global write); a **callback builtin** (the higher-order `lists` layer, which invokes a func-value argument, and the by-name dispatchers `meta.call` / `meta.callMain` / `testing.run` / `runWith` / `assertThrows` - keyed by canonical namespace, so an aliased `use lists as l` is caught); or an unresolved bare call. The walker **defaults an unrecognised node to hazardous** , so a new AST node never silently escapes. `spawn` bodies are skipped (they mutate a deep-copied snapshot, not the live globals). A method with no hazard is stamped `MethodDef.GlobalSafe` , and `methodBorrowCtx` widens the bind gate to `i.isModule || i.entryGlobalsImmutable || m.GlobalSafe` .

- **Why callback builtins are a denylist.** A func value is a `ConstRefExpr` at the AST level (indistinguishable from a constant), and func values also arrive through `func` -typed variables and untyped curried returns (`makeAdder(1)` - methods declare no return type), so a func-value *argument* cannot be spotted statically. The sound alternative is to name the builtins that can invoke `.j` code (via `BuiltinCtx.Invoke` or `CallByName*` / `CallHostWith*`). The invariant Conservative by construction: a script function that calls a module or uses a func value does not borrow (M25.1 still covers it when the script is globals-free or the call is into a module, whose own helpers borrow via isModule).
  - any such builtin must be listed - lives beside the set in `globalsafe.go` .
- **What it buys.** A read-only helper in a script that holds mutable globals goes linear: a per-element `peek(xs, i)` scanned over a growing list drops from quadratic to linear once `peek` / `scan` are proven `GlobalSafe` .
- **Discipline.** `internal/interpreter/globalsafe_internal_test.go` pins the flag (pure / direct-write / index+append-write / nested-block / transitive / mutual recursion / cycle-reaching-a-writer / function-value-call); `borrow_test.go` pins the value-semantics soundness end to end (direct, transitive-via-callee, `meta.call` , and `lists.map` global mutations all keep copy semantics; a GlobalSafe helper in a mutable-globals script borrows correctly). Race-clean.

M26 - jvc package manager (decks)

Planned. The package manager for Jennifer and its distribution: the tool itself (`M26.1`), shipping a known-good copy inside every interpreter release (`M26.2`), gating that release on jvc's own test suite (`M26.3`), and the lockfile forward-compatibility guard the two programs need once the interpreter reads jvc's output (`M26.4`).

M26.1 - the package manager itself

Planned. A package manager for Jennifer, in the shape of PHP's Composer (or Rust's Cargo): declare dependencies in a manifest, `jvc install` resolves and fetches them, and the app imports what it pulled. Installing an app becomes `git clone` + `jvc install` , and `jvc update` advances within the declared constraints.

- **Packages are "decks".** A deck is a distributable, versioned bundle of `.j` modules, published to a public **deck repository / registry** (provided later) that `jvc` resolves and fetches from - packagist-style; a deck can also come straight from a git URL.
- **Naming conventions.** A deck has two separate identities. Its **canonical name** is the `@vendor/deckname` scope in `deck.toml` (what imports and jvc key on) - kept clean, with no "deck" word: an official deck is `@jennifer/routeros` , imported `import "@jennifer/routeros/";` -> `routeros.*` . Its **GitHub repo name** is cosmetic (jvc reads `deck.toml` , not the repo name), so the "deck" marker lives there and only there:
  - **Official** decks (in the `jennifer-language` org) use a **deck- prefix** - `jennifer-language/deck-routeros` . The prefix is the " *not core, but official deck* " statement: it clusters the decks and keeps the org's top level readable (core `jennifer` / `homebrew-tap` vs `deck-*`). The org already supplies "jennifer", so `deck-` is the useful signal (deck vs core).
  - **Community** decks (any account) can be named anything - jvc only needs it configured - but the **suggested** form is a **jennifer- prefix** , `alice/jennifer-routeros` : on a personal account there is no Jennifer namespace, so "which ecosystem" is the useful signal instead.

    - **All** deck repos carry the GitHub **topic jennifer-deck** for discovery, which works regardless of the repo name.
- **Installed into the vendor/ tree M19.7 resolves.** `jvc` writes decks into the project-local `vendor/` tree that the interpreter already addresses through the `@scope/package` import form and vendor-root discovery shipped in M19.7 - so a hand-populated `vendor/` imports before `jvc` exists, and `jvc` is just the manager layered over that resolver. Nothing is global; each app owns its decks beside it.
- **The one remaining language-surface question: inline version selectors.** M19.7 resolves `import "@jennifer/supercms/" as cms;` (the trailing `/` expands to the package-named entry `supercms/supercms.j`) against whatever is installed. `jvc` supplies the *default* - plain `import @jennifer/supercms;` takes the version `jvc` resolved (declared in `deck.toml` , pinned in the lockfile), version-transparent, which is what almost every script wants. The opt-in for side-by-side versions is a **per-import selector** matched against the installed set (never triggering a fetch): `@jennifer/supercms=1.2.3` (exact), `>=1.2.3` / `~` / `^` (a `semver` constraint over what is installed), or `#cefa234` (a git commit); one script can pin `=1.x` while another pins `=2.x` , and two versions in one file take distinct `as` aliases. An unsatisfiable selector errors pointing at `jvc install` , not a silent download. **Cost:** the selector is **new grammar** (the lexer reads `@vendor/deck` + `semver` op + `#commit` as one token up to `;`); the plain M19.7 string-path form is the no-new-grammar fallback that loses only the inline selector.
- **deck.toml manifest + lockfile.** `deck.toml` (TOML, so it needs the `toml` library) declares required decks and constraints (`bitcoin = ">=1.2.0"`), and `jvc` produces a **camcorder.lock** pinning exact resolved versions (content hash per deck) so `git clone` + `jvc install` is reproducible. Dependency sets split by section (`[prod]` / `[dev]` , `jvc install --prod`), with a **taxative** (the section is the exact set) vs **additive** (base plus the section's extras) mode still to design.
- **jvc owns the lifecycle:** dependency resolution (semver constraint solving across the graph), downloading, `jvc update` (advance to the newest constraint-satisfying versions, rewrite `camcorder.lock`), integrity pinning, and the publish flow to the registry.

Migrating bundled modules out to decks. Once decks exist, niche or product-specific modules that ship bundled today should graduate *out* into decks (the archetype is `gotify` - a single-product push integration every install need not carry); language-fundamental modules stay bundled. Moving one changes its import, so it is a breaking change under semver: within 1.x ship it **both ways** (bundled + @ -deck) with the bundled copy marked `@deprecated` so imports re-point at their own pace, let the two drift without breaking, and **remove the bundled copy in 2.0.0** . Conversely, *new* third-party service integrations ship **as decks from the start** rather than as core modules (core stays general primitives; specific-vendor clients live in the ecosystem) - the `M27` candidate-deck ecosystem collects the list (GitLab, GitHub, Steam, TheMovieDB, Jellyfin, Frigate, RouterOS, ...).

Requires: the public deck **registry** (separate infrastructure, provided later); its language-side prerequisites - the `@scope/package` resolver + vendor root (`M19.7`), `toml` , the module system, and `semver` 's range surface - have shipped.

M26.2 - bundle jvc in tagged releases

Planned. Ship a known-good jvc inside every tagged `jennifer` release, so `install Jennifer` yields a working deck manager with no second step, while `jvc app install` can still shadow the bundled copy for anyone who needs a fix ahead of the next language release (the bundled copy is then the rescue path if a self-installed jvc breaks). jvc is **Jennifer source** (`.j` files, ~17 modules, no build step, no per-platform artifact), so bundling is copying text - but it uses `http` / `os.run` / `archive` , so it runs on the **default jennifer binary only** ; `jennifer-tiny` (no net / exec) does not carry it, and the launcher must `exec jennifer` , never the tiny build.

- **Vendored copy, pinned to a commit.** The jvc source is vendored into the jennifer tree (a `sync` script refreshes it) with a CI check that the copy matches a **pinned jvc commit SHA** - a commit, never a branch or a tag, since a tag is a mutable pointer (the same rule the deck resolver applies to git-sourced decks). A git submodule is the alternative if keeping upstream canonical and the pin visible in `git log` outweighs the `--recursive` friction. Fetch-at-build is rejected (a release build must stay offline / air-gapped, and

packagers dislike network); a separate companion tarball is rejected (it leaves a fresh install with no jvc, so it does not meet the goal).

- **Install as source, beside the bundled modules.** `$PREFIX/share/jennifer/jvc/` holds jvc's `cli/*.j`, and `$PREFIX/bin/jvc` is a one-line launcher (`exec jennifer run $PREFIX/share/jennifer/jvc/cli/jvc.j "$@"`), threaded through the `.deb` / OCI image / tarball / Homebrew packaging the same way the system modules already are. The shippable set is the clean `cli/` directory once jvc moves its registry-writing `publish` path off the `server/` maintenance verbs (a client CLI should not bundle the registry server's admin code); until then it is `cli/*.j` plus `server/{store,admin}.j` .
- **Report the pair in jennifer version.** `jennifer 0.25.0 (jvc 0.3.0)` , so a bug report names both halves. The jvc pin is stamped at build time through the **version codegen path** (`gen-version.sh -> internal/version/version_gen.go`), **not** `-ldflags -X` , which TinyGo silently ignores (implementation-note 7). jvc's own `jvc version` stays the authority on which copy is running vs installed (PATH order decides silently; its provenance report is the mitigation for that and for a `~/.local/bin` shadow).

Requires: `M26.1` (a taggable jvc to pin). Do not bundle ahead of it.

M26.3 - jvc suite as a release gate

Planned. Run jvc's test overlays against the release-candidate interpreter in `release.yml` ; a red suite blocks the tag. jvc is the language's most complete integration test - its assertions exercise `toml` / `json` / `semver` / `archive` / `http` / `fs` / `os` / `path` / `hash` / `encoding` / `convert` / `strings` / `lists` / `maps` / `docblock` / `flatdb` , git through `exec` , and the module system itself (cross-module struct identity, the `@vendor` resolver, the capability pragmas), so a change that breaks the language's own package manager cannot ship. This gate is what makes bundling valuable rather than merely convenient.

- **jvc is a first-class caller.** The gate fits the existing breaking-change discipline (one batch updates a library, its overlays, docs, and every caller): jvc becomes another caller updated in the same batch. An **unintended** break is a hard red caught in the language CI instead of jvc's; an **intended** break (a pre-1.0 milestone is free to make one) is handled by advancing the pinned jvc commit to a version already green against the RC, in that same batch - the gate passes because the pin moved, not because anything reverted.
- **The pin policy is written, not implicit.** A release pins a jvc commit **known-green against that RC** . An intended interpreter break requires a compatible jvc commit to exist before the tag; if jvc cannot keep up, the release waits or holds the pin back and documents the incompatibility. The language cut thereby depends on jvc being green - an acceptable reverse dependency for one maintainer who owns both repositories, named here so it is a deliberate choice.

Requires: `M26.2` (a pinned, bundled jvc) and jvc's overlay suite. The two recent breaks it would have caught - cooked-string `{expr}` interpolation turning embedded JSON literals into lex errors, and the `fmt` / `lint` line-width disagreement (`fmt` not counting a trailing `{`) - motivate it; the latter is already fixed in the tree, which the gate would confirm ships.

M26.4 - lockfile forward-compatibility guard (design-open)

Planned. Once the interpreter's vendor resolver reads jvc's `camcorder.lock` at import time, a newer jvc can write a `lockfileVersion` the bundled resolver predates. The forward-compat rule both sides agree on: the interpreter **refuses an unknown lockfileVersion with a clear message** rather than misreading it, settled before the first jvc that changes the format ships.

- **Open design question, deliberately deferred.** Nothing in `internal/` reads `camcorder.lock` today; the lockfile contract is spec, not code. Whether the **core resolver** should parse jvc's lockfile `[engines]` at all is unsettled - the interpreter already enforces version floors per file via the `# pragma-jennifer-version:` header and exposes `meta.CAPABILITIES` , so engine-floor enforcement may stay jvc's job and the interpreter's only lockfile duty be the `lockfileVersion` refusal above. Enforcing package-manager policy in the core resolver runs against the minimal-core stance, so this is resolved when `M26.1` 's resolver work is designed, not as part of bundling.

Requires: `M26.1` (the lockfile format and the resolver that would read it).

M27 - candidate decks (deck ecosystem)

Planned. A running parking lot of deck ideas - third-party service and integration clients that, once `jvc` / decks (M26) land, ship as **decks** rather than core `modules/` . The rule: core stays *general primitives* (protocols, formats, infrastructure - `http` , `graphql` , `csv`); a client for one *specific vendor or service* lives in the ecosystem as a deck (independently versioned, community-maintainable, so vendor API churn never touches the core). This list is demand-driven and open-ended, a collection not a commitment.

Most are thin clients over `http` / `rest` + `json` (plus `xml` where a vendor returns XML, and the `graphql` module where the API is GraphQL). Each is a login / token step, a generic `call(path, params) -> json.Value` , and a handful of conveniences; a fat typed wrapper is explicitly **not** the plan - these APIs are enormous and firmware-versioned, so a thin client ages far better.

M27.1 - self-hosted infrastructure

A LAN appliance, usually a **self-signed** cert. Per-vendor maturity differs and should set the order, not the vendor:

- **routeros** - a *full* RouterOS abstraction layer (MikroTik), well beyond the small bundled `mikrotik.j` API client. Already in progress; the likely **first published deck** .
- **proxmox** - Proxmox VE: documented REST / JSON, API-token header auth (`PVEAPIToken=...`). Clean.
- **vmware** - vCenter / vSphere (vCSA): the vSphere Automation **REST** API (JSON, session auth), clean and tractable like Proxmox. A standalone **ESXi** host is the messier case - historically only the SOAP / VMOMI Web Services API (XML / SOAP, heavier), with just partial REST on 8.x - so vCenter is the target, a bare ESXi host best-effort.
- **synology** - DSM: the cleanest NAS API - a documented Web API (`SYNO.API.Auth` login -> session `sid` , JSON responses).
- **unraid** - the newer official API is **GraphQL** over HTTP with an API key.
- **qnap** - QTS: the messiest - much of the useful surface is undocumented, reverse-engineered CGI with XML responses and a legacy hashed-password auth; firmware-fragile and hard to keep green. Lowest priority, on real need only.
- **ugreen** - NASync / UGOS Pro: **no official API** - only the internal token-based API the web GUI uses (log in for a token, then GET), reverse- engineered by the community (a working Home Assistant integration exists). Same keep-green risk as `qnap` ; UGOS is new and evolving, so revisit if UGREEN ever ships official docs.
- **jellyfin** - the self-hosted media server's REST API (API-key / token auth).
- **frigate** - the Frigate NVR REST API (events / config / recordings), often paired with its MQTT feed (the `mqtt` module).

M27.2 - public / SaaS APIs

- **gitlab** / **github** - dev-platform clients; both expose REST **and** GraphQL, token auth.
- **steam** - the Steam Web API (JSON, `?key=` auth); parts of the surface are community-reverse-engineered, which is exactly why it belongs in a deck, not core.
- **themoviedb** - TMDB's clean, well-documented REST JSON API (bearer / key auth).

M27.3 - daily helpers

A grab-bag of small, everyday helper decks - the little utilities a script reaches for often, each a tiny pure- `.j` deck.

- **cli spinners** - a small deck of terminal progress spinners for a long-running task: a catalogue of spinner styles (dots / line / bar / braille / arc / ...), each individually **colourable** (through `ansi`) and driven at its **own speed** (frame interval). Self-suppresses when stdout is not a TTY (`os.isTerminal`), so piped output stays clean.

M27.4 - bioinformatics

A sequence-manipulation deck for DNA / RNA / protein, modelled on the Sequence Manipulation Suite (https://stothardresearch.ca/sequence-manipulation-suite/) (SMS) tool catalogue - the classic, comprehensive reference for this surface. Almost all of it is pure string / list / map work, so it is a natural **pure-.j deck** (leaning on `strings` , `regex` , `lists` , `maps` , `math` , and `stats` , no Go), dogfooding the language and community-maintainable as tables and algorithms are added. Grouped SMS-style:

- **Transforms** - `complement` , `reverseComplement` , `reverse` , `transcribe` (DNA <-> RNA), `translate` (codon table), six-frame translation, `splitCodons` , amino-acid `oneToThree` / `threeToOne` .
- **Composition & properties** - `gcContent` , base / amino-acid composition, `dnaStats` / `proteinStats` summaries, `molecularWeight` (DNA / RNA / protein), `meltingTemp` (Wallace + nearest-neighbour), `isoelectricPoint` (pI), `gravy` (hydropathy), `codonUsage` . The thermodynamic / pI / nearest-neighbour formulas want `exp` / `ln` / `log` , so they pull in `M24.10` (`math` foundations).
- **Search** - exact and IUPAC-ambiguity pattern find (via `regex` character classes), fuzzy search (n mismatches), ORF finder (six frames), CpG islands, restriction-site search and `restrictionDigest` (fragments) over a bundled enzyme table.
- **Formats** - FASTA parse / write (a `Record{id, description, sequence}` list), FASTQ reads (sequence + quality), format conversion, `filterDna` / `filterProtein` (strip non-sequence characters).
- **Manipulation** - `randomDna` / `randomProtein` (length + optional composition), composition-preserving `shuffle` , point `mutate` (rate), range / sliding-window extraction.
- **Comparison** - pairwise `alignGlobal` (Needleman-Wunsch) / `alignLocal` (Smith-Waterman) with percent `identity` / `similarity` , plus `hammingDistance` / `editDistance` . Alignment is the one `O(nm)` hot loop - fine in `.j` for the small sequences a deck user handles; a candidate Go primitive only if whole-genome throughput is ever needed.
- **Reference data** - the standard genetic code + alternative codon tables, IUPAC ambiguity codes, per-residue property tables (MW / pKa / hydropathy), and a common-enzyme table - all plain `.j` maps a community can extend.

FASTA / FASTQ I/O and alignment cover the "molecule structures" the original note gestured at; PDB / 3-D structure parsing stays out of v1 (a much larger, separate effort).

M27.5 - forensic / statistical genetics

The statistical-genetics sibling of the sequence deck (M27.4): it works on **profiles** (an unordered allele pair per autosomal STR locus, plus uniparental **haplotypes** for mtDNA / Y-STR) and reference frequency / count data, not on sequences, so it shares no code and no audience with SMS. Pure `.j` (probability arithmetic over allele-frequency maps, no Go, TinyGo-clean), leaning on `math` and on `xml` / text parsing to ingest a published frequency table.

- **Match probabilities** - Hardy-Weinberg single-locus genotype frequencies (homozygote `p^2` , heterozygote `2 pa pb`), with the NRC II theta / Fst sub-population correction (Balding-Nichols), multiplied across loci: `randomMatchProbability` (RMP), the single-source match `LR = 1 / RMP` , and `cpi` / `cpe` (combined probability of inclusion / exclusion) for mixtures. A minimum-allele-frequency floor (`5 / 2N` , so the table keeps each locus's `N`) handles rare or unobserved alleles.
- **Lineage markers (mtDNA / Y-STR)** - mitochondrial and Y-chromosome markers are haploid, non-recombining, and uniparentally inherited, so match probability is **not** Hardy-Weinberg but a direct **haplotype count** : frequency `k / N` in a reference database, with a **Clopper-Pearson** exact-binomial upper bound as the conservative estimate (which pulls in the `beta` distribution `M24.11` adds to `stats`). Deliberately **database-independent** , so the deck supplies the estimator (`haplotypeFrequency(k, N)` / `lineageMatchProbability`) and the caller feeds the count from whatever database they queried. mtDNA adds a haplotype coded as differences from the **rCRS** (e.g. `263G 315.1C`) plus the substantive `align` step that renders a raw sequence into that standard nomenclature; Y-STR is a per-locus repeat-count vector with exact / single-step comparison.
- **Kinship likelihood ratios** - relationships encoded as IBD coefficients (kappa0 / kappa1 / kappa2: parent-child `(0, 1, 0)` , full sibs `(1/4, 1/2, 1/4)` , half-sib / avuncular / grandparent `(1/2, 1/2, 0)` ,

first cousins (`3/4, 1/4, 0`), ...). `kinshipLR(a, b, relationship, db)` tests one relationship against another; `paternityIndex(mother, child, allegedFather, db)` gives a combined paternity index (CPI) and `probabilityOfPaternity` (`W = CPI / (CPI + 1)`), with a stepwise STR mutation model to survive a lone inconsistency. General pedigrees (beyond pairs / trios) need a peeling engine - **Elston-Stewart** - the one substantial algorithm, the Familias / `forrel` -style capability.

Mixture deconvolution (multi-contributor, drop-in / drop-out - the EuroForMix space) is a larger, later effort layered on this base.

A concrete frequency source to wire in first, as a sample: the ENFSI STR reference database **STRidER (https://strider.online/frequencies)** - a `loadFrequencies(xml, "strider")` over the `xml` library, with its formulae page (https://strider.online/formulae) and online calculator as the exact-conventions spec and a validation target.

M27.6 - NGS / high-throughput sequencing

Unlike the pure- `.j` decks above, NGS breaks the model on two axes - **scale** (FASTQ.gz files run 10s of GB, BAM 100s, so everything **streams**, never load-into-memory) and **compute** (alignment / assembly / variant calling are heavily SIMD-optimised C a tree-walker is ~100-1000x too slow to replace). So the deck is deliberately the **glue and light-I/O layer**, not the heavy engine, with a different posture from the pure- `.j` decks: **Go-backed streaming parsers** (the decompress-and-parse hot loop in Go, the `net.readAll` / `binary` pattern) with the per-record logic in `.j`. In scope:

- **Streaming format I/O** - FASTQ (gzipped, over `compress` + `fs` handles) and the tab-delimited **SAM** / **VCF** / **BED** / **GFF** / **GTF** text formats: parse / filter / convert as a stream, so a 50 GB file never lands in memory.
- **QC + preprocessing** (FastQC / fastp-lite) - per-base quality and read-length / GC distributions, adapter detection, quality / adapter trimming, length filtering, subsampling: one streaming pass over FASTQ.
- **Interval ops** (bedtools-lite) - overlap / intersect / merge over BED / GFF features.
- **Pipeline orchestration** - the real sweet spot: NGS is fundamentally `bwa | samtools | gatk` glued together, and Jennifer already has `os.run` / `os.spawn` + `spawn` concurrency, so it can drive the standard tools, parse their output, manage intermediate files, fan out over samples in parallel, and handle errors - a Snakemake-lite in a real language.

Out of scope (wrap and pipe the native tool, do not reimplement): read **alignment** (BWA / Bowtie2 / minimap2), de-novo **assembly** (SPAdes), **variant calling** (GATK), and heavy **BAM / CRAM** handling - which additionally needs **BGZF** (blocked gzip for random access), a `compress` gap and a Go addition whose standalone value is limited without the downstream compute the deck omits.

M28 - multiplatform: promote macOS / Windows to supported

Linux is the only *supported* platform, but best-effort **unsupported** macOS / Windows binaries (the standard-Go `jennifer`, via cross-compile) already ship each release - so the work is not "add the ports" but **promoting them to supported**, which graduates the "cross-build for macOS / Windows" 1.0.0 distribution requirement. A portability audit found the surface small: separators / EOL / `$HOME` / temp are already `runtime.GOOS` -derived (`internal/lib/os/oslib.go`), `os/exec` keys on `runtime.Compiler != "tinygo"` (not GOOS, so it is enabled on Windows), and signals + the four Linux-only hardware libs (`serial` / `spi` / `i2c` / `gpio`) stub cleanly on non-Linux (`*_other.go`). **Extra distribution packaging** (a Homebrew tap, Snap, Nix flake, Flatpak / AppImage) stays a per-format nice-to-have, shipped only when a user asks and a maintainer keeps it green - none blocks a release.

M28.1 - Windows: promote to supported

Planned. The **Windows** track is a handful of concrete gaps, not a rewrite:

- **Exe-relative module default.** `compileDefaultSysmoddir` bakes one hardcoded POSIX path (`/usr/share/jennifer/modules`, `internal/module/sysmoddir.go`) into a Windows binary; give it a Windows-native, exe-relative default (`<dir(os.Executable())>\share\jennifer\modules`) via a

build-tag split (`sysmoddir_windows.go` / `_unix.go`), so a portable-zip user's `import "name.j";` resolves with no env var (the `M21.13` installer's `JENNIFER_SYSMODDIR` stays the explicit override; precedence unchanged).

- **Per-OS golden strategy.** `examples/expected/osinfo.txt` is the sole platform-pinned golden (pins `linux` / `amd64` / `/` / `:` , compared byte-exact in `cmd/jennifer/examples_test.go`); add per-OS expected-file selection or a `runtime.GOOS` -gated skip for the osinfo canary (already flagged at `examples/osinfo.j`).
- **fs.chmod / fs.chown on Windows.** Define the Windows behaviour (a friendly catchable error is acceptable; the `chown` test is already Linux-gated), and document that signal-based graceful shutdown is limited on Windows (`signal_other.go` stubs `os.catchSignal`).
- **A windows-latest CI test job** running `go test ./...` so Windows correctness is actually verified (the exec suite self-skips off Linux; the osinfo golden is the known failure the item above resolves); once green, move windows/amd64 out of the `build-unsupported` matrix into the supported set and drop the "unsupported" labelling for that arch.

Requires: none.

M28.2 - macOS: promote to supported

Planned. The parallel case: the same "already ships unsupported, promote it" shape as `M28.1` , and simpler - macOS lacks even the module-path blocker Windows has (the POSIX exe-relative default resolves cleanly), and its separators / EOL / `$HOME` match Linux. A `macos-latest` CI test job running `go test ./...` verifies correctness (reusing the per-OS osinfo golden strategy from `M28.1`); once green, move darwin/amd64 + darwin/arm64 out of the `build-unsupported` matrix into the supported set and drop the "unsupported" labelling. **Requires:** none.

M29 - project governance, licensing, and contribution policy

Planned. A hard requirement for 1.0.0 stable (also listed under Requirements for 1.0.0 stable). The rules for *how the project is run and how outside contributions are taken* - organizational, not code. Untouched while the project is solo (one author, `Copyright (C) 2026 mplx <jennifer@mplx.dev>` , `LGPL-3.0-only` , no outside PRs), but it must be settled **before the first external contribution is merged** : several of the choices are hard to reverse once other people's copyrightable work is in the tree. The open questions, roughly by urgency:

- **Copyright-holder model.** Under distributed copyright (the default, no paperwork) every non-trivial contributor automatically holds copyright in their patch, so the tree becomes a mosaic of holders and any future relicensing needs each one's agreement. The alternatives are a **CLA** (contributor grants the project a broad license, keeps their own copyright) or an **assignment / CAA** (contributor transfers copyright to a single holder) - both consolidate the rights but add contributor friction, and assignment needs an entity to hold them. This is the decision that is expensive to undo.
- **The copyright notice.** Whether headers stay per-author (`(C) <name>`) or move to a collective label (`(C) The Jennifer Authors` , defined by git history). The trap to avoid: a two-file `AUTHORS` (holders) / `CONTRIBUTORS` (credit) split only carries information when a **work-for-hire** contributor exists (employer holds copyright, individual is merely credited); for an all-volunteer project the two lists are identical, so the split is pointless. Either keep no enumerated holder file (the collective label refers to git history) or consolidate ownership via CLA / assignment.
- **Relicensing headroom.** LGPL already lets anyone embed / link Jennifer without permission, so ordinary use never needs a contributor's sign-off. The only thing distributed copyright forecloses is issuing a *different* license - e.g. a commercial embedding exception for a deep-embedded `jennifer-tiny` target that cannot meet LGPL's static-relink terms. If keeping that option open matters (embedding is a first-class goal), a CLA is the tool; if "LGPL-only forever" is acceptable, distributed copyright is fine and the constraint never bites.
- **Contribution mechanics.** `CONTRIBUTING.md` , the sign-off mechanism (a lightweight **DCO** `Signed-off-by` line, which asserts "I have the right to submit this" without a license grant, vs a full

CLA-bot, which also grants one - the choice follows from the relicensing decision above), a code of conduct, and the PR / review workflow.
- **Project governance.** Who decides (BDFL vs a maintainer group), how commit rights are granted (judgment and sustained involvement, never an LOC or commit-count threshold - metrics are a bad proxy and get gamed), and a `MAINTAINERS` file once more than one decision-maker exists. Being listed as a contributor confers no authority; credit and governance are separate.
- **Name / mark.** Whether the "Jennifer" / `jennifer-lang` identity needs any trademark-style usage policy (forks, the deck registry) or stays informal.

The license itself stays **LGPL-3.0-only** unless a deliberate relicensing decision above changes it; this milestone is about the *process and ownership* around it, not a license change. Not legal advice - the chosen model should get a real legal review before it is published.

Requires: none (organizational, independent of the codebase). Socially paired with the M19.8 org move and triggered by the first external contribution, but no code prerequisite - which is why it is the one 1.0.0 requirement that can be settled at any time before the first outside PR.

Requirements for 1.0.0 stable

- **Project governance, licensing, and contribution policy** - the `M29` work: the copyright-holder model, copyright notice, relicensing headroom, `CONTRIBUTING.md` / sign-off, governance, and name / mark. A **hard requirement** for 1.0.0, and it must be settled before the first external contribution is merged (whichever comes first).
- **Cross-build for macOS / Windows.** The `M28` multiplatform track (`M28.1` Windows, `M28.2` macOS) does this; ships as soon as it lands.
- **Real apt repository** (replacing the "GitHub Release artifact" install of the M15.8 `.deb`) if user demand warrants the maintenance.
- **Container image (OCI).** Done - `.github/workflows/docker.yml` builds and pushes multi-arch (`linux/amd64` + `linux/arm64`) `ghcr.io/<owner>/jennifer` images on each release tag: a Debian-slim default (`:latest` / `:<ver>` , full host features) and a distroless `:static` variant. The image bundles both binaries and the system modules under the compile-default module dir, so a bare `import "name.j";` resolves with no env var. See `packaging/docker/` .
- **Homebrew tap (macOS, best-effort unsupported).** Available - `packaging/homebrew/jennifer.rb` builds the standard `jennifer` from source (so Intel + Apple Silicon work with no Gatekeeper friction) and bakes the version and module path via `-ldflags -X` , so a bare `import "name.j";` resolves with no env var. Published to the `jennifer-language/tap` tap by a manual `publish.sh` (the AUR convention; no CI secret). The `-UNSUPPORTED` macOS tarball stays alongside it.

The extra Linux / macOS distribution formats (Homebrew, Snap, Nix, Flatpak, AppImage, ...) are not requirements; they live in the horizon idea collection and ship when there's user demand and a maintainer willing to keep one green.

Long horizon

Ideas for development *beyond* 1.0.0 - embedding, a WASM runtime, specialised-domain libraries, and a grab-bag of smaller possibilities - live in their own collection, kept out of the near-term plan so this file stays focused on the road to 1.0.0. See the beyond-1.0.0 idea collection.

Beyond 1.0.0 - idea collection

Jennifer's near-term target is a rich, dependable set of language features, libraries, and modules - enough to tag a **1.0.0** . That work is tracked in milestones.md. This file collects ideas for *after* 1.0.0: directions worth recording so the design doesn't foreclose them, none committed to a timeline.

Two kinds:

- **Drafts** - concrete, already-shaped directions, grouped by theme and ordered roughly easiest-first. Each has a design; it just has no schedule.
- **Loose ideas** - a grab-bag of smaller or vaguer possibilities, loosely grouped, jotted down when they come up so they are not lost.

Nothing here is a commitment. An idea graduates into milestones.md if and when it earns a slot.

Drafts

Shaped directions, **grouped by theme** and ordered roughly **easiest to implement first** - language tweaks and additive libraries near the top, the big runtime and ecosystem restructures near the bottom. Each has a design; it just has no schedule.

Each carries a stable **DRAFT#** handle so it can be referenced and later graduated into a numbered milestone (graduating a `DRAFT#` gives it an `M` -number in `milestones.md`). Handles are **assigned once and retired on graduation** - never reused, and **never renumbered when the list is reordered** - so a reference stays valid for the life of the idea (which is why the numbers below run out of sequence). `DRAFT#` is deliberately *not* a milestone number; an idea only gets an `M` -number when it graduates into milestones.md.

Each draft also states its **Requires** - the `DRAFT#` handles and/or milestones (`M` -numbers) that must land first, or *none* - so its blockers are explicit before it is scheduled.

Language

DRAFT#28 - Anonymous function literals with value capture (Tier 2 of first-class functions)

`func` values already ship (`M24.2`): a function held in an immutable value, passed / returned / stored / called, powering the higher-order `lists` layer. What is missing - the "Tier 2" `M24.2` deferred - is the **anonymous function literal** (an inline lambda) and **capture** of enclosing bindings. Today a one-off transform must be hoisted to a named top-level method (`func dbl(n as int) { return $n * 2; }` just to write `lists.map($xs, dbl)`), and a middleware or handler cannot close over state, which is why the module ecosystem routes so much through **by-name dispatch** (`web` handlers, `args.dispatch` , `webapi` 's authenticator / limiter names + its `guard` shim). Both the ergonomic complaint ("no closures") and those internal workarounds are the same gap.

The design that fits: value capture, not reference capture. A literal `func(n as int) { return $n * $factor; }` in expression position evaluates to a `func` value whose captured bindings (`$factor`) are a **deep-copy snapshot taken at the moment the literal is evaluated** - exactly what `spawn` already does to its enclosing scope. Captured bindings are immutable snapshots, never live references, so no aliasing is introduced and value semantics hold unchanged. This is the only flavor consistent with the language; the reference-capturing (JS / Python) flavor is rejected (see rejected.md) because a live env pointer breaks value semantics and the frame-pool / borrow / spawn-snapshot optimizations documented as safe *because* there are no such captures.

It reuses machinery that already exists. Capture is `snapshotForSpawn` 's deep-copy, so the value-semantics-capture path is built. A closure body is a resolver / borrow-analysis **hazard boundary** the same way a `spawn` body is (skip descent, treat as opaque), and a captured closure is **excluded from the frame pool** the same way a spawn snapshot is - the carve-outs are the ones the concurrency work already established, applied to a new node. Across a module boundary a closure carries its defining interpreter like `FnHome` (implementation-note 25) already lets a `func` value do, so a captured closure resolves its own namespaces and

retags structs at the boundary.

The one teachable surprise is the snapshot: a captured variable is frozen at creation, so a later mutation of the outer variable is invisible inside the closure

- the identical mental model `spawn` already teaches, deliberately consistent

rather than the JS / Python live-binding behavior.

What it buys, beyond inline lambdas. Real currying actually captures (`makeAdder(1)(2)` today is call-through-expression syntax, not capture - this is what would make it capture). And it lets the module ecosystem replace by-name dispatch with closures that close over their state - a `webapi` guard could capture the `Api` directly instead of a name plus a one-line shim, `args.dispatch` could take closures - simplifying those APIs. Whether to *migrate* those modules is a separate follow-on, not part of this draft.

Syntax and semantics to settle at graduation: the literal spelling (`func(params) { body }` , mirroring a named `func` declaration without the name), whether parameter types are required (yes, for consistency with methods), no declared return type (consistent), and how `defer` / `spawn` inside a closure body behave.

Requires: `M24.2` (the `func` value, `KindFunc` , and `CallValueExpr` this extends). Reuses the `spawn` snapshot and the resolver / borrow carve-outs already in place; no new dependency.

Concurrency

DRAFT#25 - Go-side concurrent HTTP serve loop

The `web` framework dispatches each request into an entry-program handler `func` value (called in its home context), and `web.serveOn` runs that dispatch in a `.j` loop. Once `web` handles requests concurrently (each wrapped in a `spawn`), an optional refinement is to move only the **accept + bounded-worker-pool loop** into Go, as an `httpd` primitive (e.g. `httpd.serveLoop(server, handlerName)` , or a small engine-owned pool) that calls `Interpreter.CallHostWith` per request. Routing, middleware, and `web.Context` stay in `.j` .

Why it might pay: a worker pool with a tunable ceiling is more natural in Go than expressed as `.j` `spawn` s; back-pressure / queue-depth limits live where the sockets already are (`httpd` already bounds `maxInFlight`); and the hot accept loop stops paying tree-walker overhead per request. Why it is only a refinement, not a fix: a handler is still a `.j` method invoked on the shared host interpreter, so making that invocation race-safe is the actual prerequisite and does the real work - this only relocates the loop. Keep `web` a `.j` module either way: dogfooding is a first-class goal, and with no closures a handler is always a by-name entry method, so a Go rewrite of the router would buy little and lose test surface.

Developer tooling

DRAFT#27 - Static call-site checking in jennifer lint

`jennifer lint` resolves a call's *name* (`L107` `undefined-call`) but never checks its **arity or argument types** , so code that is compile-legal yet cannot run - a `bytes` passed to a `string` parameter, a call left one argument short after a signature changed - passes a lint gate in CI or a container build and surfaces only as a runtime error on the first request that exercises the path. The runtime diagnostic is already precise (`argument 3 to "token" must be string, got bytes`); the goal is to surface a **decidable subset** of it at lint time, emitting **only where the type is statically known** so the checks stay false-positive-free - the property that earns them a place in a CI gate. These land in the `L1nn` correctness band as `L108` (arity), `L109` (argument type), and `L110` (initializer type). The work splits into three tiers by how much is statically knowable, plus one part that stays undecidable.

Tier 1 - same-file, no language change

Arity and type mismatches against a **same-file user method** are fully decidable from the AST lint already walks. `L108` compares a call's argument count to the callee's `MethodDef.Params` (the callee resolves exactly the way `L107` does today). `L109` / `L110` compare an argument's - or a `def ... init` expression's - static type to the declared parameter / binding type, but **only** when that type is knowable without inference: a literal, or a variable / parameter / constant / struct-field reference whose declared type lint's own scope pass (`scope.go`) already

resolves. This catches the arity-drift bug (a parameter added, one call site missed) and same-file type mismatches with zero false positives, and ships as an additive lint change over the existing AST and scope machinery.

Tier 2 - cross-module (lint loads module ASTs)

Extend the same three checks across an `import` boundary, so `callee.wantsString(42)` is caught the way a same-file call is. This needs lint to **parse the imported module's AST** to read the callee's parameter types - which it does not do today: lint sees the import *declarations* (`c.prog.ModuleImports`) but never loads the target file. Reusing the interpreter's module resolver closes that gap; a bounded lift. This tier is what makes lint a trustworthy gate for a module-heavy repo, where a thin entry script is untested and all the checked logic sits behind module boundaries.

Tier 3 - library builtin signatures

The single change that closes the *originally reported* symptom - a library return type flowing into a typed slot, `web.body(ctx)` (`bytes`) passed to a `string` parameter. It is out of reach of Tiers 1 and 2 because **builtins carry no static signature** : a registered builtin is `type Builtin = func(ctx BuiltinCtx, args []Value) (Value, error)` , with no declared arity, argument types, or return type - each validates its own arguments at runtime, so the interpreter learns a builtin's types only by calling it. Adding a **static signature table** (arity + argument kinds + return kind), attached at registration or as a declarative side table, unlocks library-call arity, library-call *arguments* (`wantsString(crypto.randBytes(4))`), and library-call *initializers* (`def s as string init crypto.randBytes(4)`). It is bounded but real work across the ~40 libraries, must stay TinyGo-clean and not bloat `RegisterNamespaced` , and is independently valuable (editor tooling, doc / cheatsheet generation, possibly stricter runtime messages). The open design question is *how* to attach signatures without that bloat, which earns a `design-decisions.md` record.

The undecidable remainder

A **user method declares no return type** (deliberately - "the caller's `def x as T init f();` checks it"), so `f(g())` with `g` a user method stays undecidable in a single pass whatever the tiers above do. `v1` **skips** it (still valuable - the reported case was a library call, not a user-function nesting). Inferring a return type from a body's `return` statements is possible but multi-type / ambiguous bodies add false-positive risk; an optional return-type annotation is a genuine language change against a deliberate design choice, so either path needs its own decision and record rather than riding along with the checks.

Embedding, WASM, and sandboxing

DRAFT#1 - Public interpreter API for third-party embedding

Extract the interpreter core out from under `internal/` and expose a documented Go-side surface so external programs can embed Jennifer. Today `internal/interpreter` , `internal/parser` , `internal/lexer` , and `internal/lib/*` are unreachable from any module that isn't `jennifer-lang.dev/jennifer` - Go's `internal/` visibility rule is not a convention, it's a compile-time barrier. No submodule / require / replace workaround exists; embedding is impossible without a restructure.

It comes ahead of the WASM runtime (`DRAFT#2`) because a Go-side embedding API is a strictly smaller change (repository restructure, no new external dependency), it unblocks the most immediate embedding scenarios (scripting slot in a Go host, LSP / formatter tooling, test harnesses), and it does not foreclose WASM (`DRAFT#3`) - a plugin surface can layer on the same `pkg/` facade once Wazero (or similar) is in play.

Concretely. Add a `pkg/` top-level (working name; the final path settles at the start of this work):

- `pkg/interpreter` re-exports `Interpreter` , `Value` , error types, and the `Install(in *Interpreter)` registration API that every stdlib library already uses. The `internal/` packages stay as the implementation; `pkg/` is the stable facade with semver-covered surface once we ship 1.0.
- `pkg/lib/*` re-exports each shipped library (`convert` , `math` , `strings` , ...) so a host can install the ones it wants and leave out the rest. Non-breaking for the current CLI - `cmd/jennifer` picks up the same `Install` calls, just through `pkg/lib` shims instead of directly.

- Documented pluggable interfaces for the host-provided facilities the OS-touching libraries currently reach for:
  - `io.Writer` for `io.printf` output (already a `*Interpreter` field; formalize as an interface).
  - `io.Reader` for `io.readLine` / `io.readBytes` / `io.readChars` stdin.
  - `Clock` for `time.now()` / `time.local()` / `time.sleep` (the `nowFunc` / `sleepFunc` test hooks in `internal/lib/time` are the shape).
  - `Rand` for `math.rand*` / `lists.shuffle` (the shared random source).
  - Filesystem / network / process hooks left as future work - a host wanting those either installs the stdlib libraries as-is (accepting the Go `os` / `net` dependencies) or ships its own shims. A documented registration pattern is the deliverable; the shims themselves are per-host and out of scope here.

Stdlib-backed defaults. Each pluggable interface carries a working default so `pkg/interpreter.New()` plus `pkg/lib/io.Install(in)` produces a running interpreter without every embedder wiring up seven interfaces first. `Clock` defaults to Go's `time.Now` , `Rand` to a `math/rand` source, `io.Writer` to `os.Stdout` , `io.Reader` to `os.Stdin` . Hosts override only what they need. A `no-os` embedder replaces every default; a Slack-bot embedder swaps just `io.Writer` for its outgoing-message pipe and leaves the rest.

Boundary rules at the Install site. Three explicit error paths so hosts get loud, positioned failures instead of subtle misbehaviour:

- **Duplicate library Install at the Go level is rejected** , mirroring how a duplicate `use NAME;` errors at the Jennifer level (the duplicate- `use` rule, lifted). A host installing `pkg/lib/math` and then its own shim that also claims the `math` namespace fails at the second `Install` call, not silently overlaid.
- **Install and pluggable-interface setters are frozen once Run() starts.** Attempts to call `Install` , `SetClock` , `SetOut` , or friends after the interpreter has begun executing produce a positioned "cannot configure interpreter mid-run" error at the Go call site. The interpreter can then trust its host bindings for the rest of the run without defensive re-checks.
- **Host implementations are trusted at the interface boundary.** The interpreter uses whatever `Clock.Now()` or `Rand.Int63()` returns without validation - a broken host implementation is the host's problem, not the interpreter's. Stated so hosts don't expect defensive checks that aren't there and so downstream bug reports are triaged to the correct side of the API boundary.

Non-goals.

- A hosted no- `os` build target. Even with this restructure, the shipping stdlib libraries lean on Go's `os` / `net` / `time` packages. A truly bare-metal or `no-os` embedding can only use the pure-value libraries (`convert` , `math` , `strings` , `lists` , `maps` , `hash` , `crc` , `encoding` , `regex`) plus whatever host-provided shims the embedder wires up. That's a design constraint on the embedder, not a milestone on Jennifer's side.
- Semver freezing the public API. Jennifer stays pre-1.0 through this work; it documents what's exported and how libraries plug in, but breaking changes to that surface remain allowed until 1.0.0.

Motivation. Third-party embedding has multiple concrete consumers already imagined: scripting-language slot in a Go application, tooling that needs direct AST / interpreter access (LSP, formatter integrations, syntax highlighters), test harnesses that want to drive `.j` programs from Go, config-DSL runtimes, plugin systems for game engines and similar. None of them require an OS-free build; all of them need the `internal/` -> `pkg/` restructure. The `Install` pattern already works this way - every stdlib library is a `pkg.Install(in)` call. The missing piece is visibility, plus documented hooks for the pieces of host state currently exposed only as package-level test vars.

Requires: none - a self-contained restructure of the current codebase. Best sequenced once the core library / module surface has settled, so the `pkg/` facade is stable, but nothing blocks it.

DRAFT#2 - WASM runtime embedding

Wazero or similar inside the interpreter binary. TinyGo-size cost evaluated honestly before commitment. Without it, no WASM libraries (DRAFT#3).

Requires: none (embeds Wazero directly).

DRAFT#3 - WASM libraries

If the WASM runtime (DRAFT#2) ships, sandboxed plugins via `use wasm:libname;` . Each library its own piece.

Requires: DRAFT#2 (the runtime) and DRAFT#1 (the plugin surface layers on its `pkg/` facade).

DRAFT#11 - Sandbox

Restricted-capability execution.

Requires: none hard; relates to DRAFT#1 (embedding) and DRAFT#3 (WASM isolation).

Interpreter internals

DRAFT#17 - Bytecode execution model

The tree-walker re-walks the AST and re-resolves shapes on every pass through a hot loop; that per-operation dispatch + Value-copy overhead is the throughput ceiling for CPU-bound `.j` . Compile the resolved AST to a linear **bytecode** (or a register form) executed by a tight dispatch loop - a new pipeline stage between resolve and run - so a loop body is decoded once and then dispatched without re-walking the tree. This is the big structural lever, and the big effort.

- **Same semantics, same discipline.** Value semantics and the tagged-union `Value` stay exactly as they are; only how operations are sequenced and dispatched changes. Held to the same TinyGo-clean, reflect-free rule as the current evaluator, and to strict behaviour parity: the `spawn` snapshot, `defer` order, positioned errors, and the call-depth guard must all survive the rewrite, with the existing test suite as the conformance oracle.
- **Sequenced after the cheap win.** what remains for a VM is the residual CPU-bound `.j` (recursion, business-logic loops) no Go primitive covers. Pursued only when benchmark shows that residual is a real workload's bottleneck - not on spec.
- **Composes with arena.** The per-frame arena allocator is an independent memory-side optimization that pairs naturally here, since a bytecode VM restructures allocation anyway - but neither depends on the other.

Copy-on-write for compound Values is **not** part of this: it was tried (shared-marker COW, reverted as inert) and the write-through variant is rejected for reintroducing shared mutable state - see technical/rejected.md.

Loose ideas

A grab-bag, loosely grouped and recorded when it comes up.

Language sugar

- **Explicit map-to-struct conversion.** A spelled-out, validating way to turn a `json.Value` object (or a homogeneous `map of string to T`) into a typed struct - the sanctioned counterpart to the *rejected* implicit coercion (see technical/rejected.md). Deferred: once JSON is destructured through `json.Value` accessors, the by-hand rebuild covers the need, so a one-call form is a convenience, not a blocker. Two candidate shapes, decided on consistency not brevity - a `convert.toStruct($map, "Point")` library call (a two-arg, stringly-typed outlier in the otherwise one-arg `convert.toX` family, or else not self-contained if it reads the binding's declared type) versus a `Point{ ..$map }` struct-literal spread (names its type statically, at the cost of new literal syntax). Either way strict: every declared field present with a matching type, recursing into nested structs / lists / maps, value-semantic, no partial fills or defaults.
- **Decimal / bignum / money math.** A Go-backed arbitrary-precision base-10 `decimal` library (over `math/big`) with `from` / `add` / `sub` / `mul` / `div` / `round` / `compare` and an opaque `Decimal` value (the `KindObject` shape `json.Value` uses) - exact money arithmetic with no float rounding, kept a **library handle** rather than a new primitive so the core `int` / `float` model is untouched.

Library completions

- **Locale-aware string collation.** `strings.fold(s)` ships as the lightweight answer for accent-insensitive sort / search keys: strip common Latin diacritics (`Österreich` -> `Osterreich` , ß -> `ss`), pair it with `lists.sortBy` for a "good enough" locale-ish order. What it deliberately is *not* is full Unicode collation - the

Unicode Collation Algorithm (UCA) with per-locale CLDR tailorings, where German phonebook order puts ö at `oe`, Swedish sorts ö *after* `z`, and primary / secondary / tertiary weights separate base letter from case and accent. That is a data-heavy `collate` library (the scope of ICU / `golang.org/x/text/collate`), a real project best parked here until a concrete need justifies the CLDR data or the dependency; `strings.fold` covers the common Western-European case in the meantime.

- **encoding - the harder codecs.** The single-byte character codecs and binary-to-text formats all shipped; the deferred remainder, picked up only when a real program needs one: variable-width Asian encodings (`Shift-JIS`, `Big5`, `GB2312`, `GBK`, `GB18030`, `EUC-JP`, `EUC-KR`) - each a state machine with variant / ambiguity edge cases, a whole piece apiece; `UTF-16` / `UTF-16LE` / `UTF-16BE` / `UTF-32` (BOM, surrogate pairs, endianness); and `UTF-7` (mail-transport - though `quoted-printable` already shipped as a general codec).
- **Password hashing (Argon2id / bcrypt / scrypt).** The modern default for password *storage*, deferred out of the `crypto` library because it lives in `golang.org/x/crypto` (a dependency, unlike the stdlib KDFs `crypto` ships) and wants its own surface distinct from the KDFs: a self-describing `crypto.hashPassword(pw) -> string` (`$argon2id$...`) plus a constant-time `crypto.verifyPassword(pw, hash) -> bool`. Added when password storage is a concrete need, taking the `x/crypto` dependency then - crypto is the one place the dependency-free stance bends, since you never hand-roll it.
- **time: IANA / DST zones.** Real zone names (`Europe/Berlin`) with historically-correct daylight-saving resolution, added to the `time` **system library** - not a hand-maintained `.j` data map. A `.j` map is the wrong shape: abbreviations (`CST` is US Central *and* China Standard *and* Cuba Standard) don't identify a zone, and the real model is offset-per-(zone, instant) over a transition history that ships several updates a year. Back it with Go's `time.LoadLocation` + the embeddable `time/tzdata` (or the host's `/usr/share/zoneinfo`), so the database is the toolchain's problem and resolution is correct at any instant. Standard- `jennifer` only: TinyGo's `time` can't load zones, so `jennifer-tiny` stays fixed-offset (a build-tag split like `net`). Level 1 first - an offset-at-instant resolver (`time.offsetAt(name, $t)` / `time.zoneFor(name, $t) -> time.Zone`) that leaves the `time.Time {nanos, offset}` model untouched (the snapshot is fixed, so DST-crossing arithmetic must re-resolve); Level 2 - a zone-carrying `time.Time` with DST-correct arithmetic - is a larger, optional follow-up needing a Go-backed zone handle.
- **label: embed a bitmap image in the job.** Today `label.image` references an image already stored on the printer (by name). The heavier alternative is to embed the bitmap in the rendered job so a logo travels with the label and needs no pre-loading: convert a source image (PNG / mono bitmap) to each dialect's raster - cab embedded-ASCII image data, ZPL `^GF` graphic field - which needs image decoding plus 1-bit dithering / thresholding. That is a real raster-conversion capability (a Go-side helper or an `image` library), not the pure-text `.j` the rest of the module is, so it is a separate piece of work rather than another encoder branch. Until then, `label.image` (by reference) covers the stored-logo case.
- **SQLite (sql engine backend).** SQLite stays parked here, and it is worth being precise about *why*, because it is not the reason it first looks like. SQLite is **also** just a pure-Go `database/sql` driver - `modernc.org/sqlite`, registered with the same one-line `import _` as `go-sql-driver/mysql` or `pgx`, cross-compiling cleanly like any pure-Go package (the cgo `mattn/go-sqlite3`, which *does* break static / cross-compile / TinyGo and needs a C toolchain, is rejected in its favor). So integration effort and API are identical to the shipped drivers; SQLite is in every practical sense "just a third driver" for the same library, sharing its surface and opaque `sql.Row` result shape. The one real difference is **weight**. `modernc.org/sqlite` is the entire SQLite C source transpiled to Go plus `modernc.org/libc` (a Go libc reimplementation) - multiple MB of generated code, versus the hand-written, few-hundred-KB protocol clients already shipped. Baking that into every default `jennifer` bloats the binary for the many users who only ever touch a network database. That, and only that, is why SQLite is gated as a **build-tag opt-in** (`-tags sqlite`), surfaced as a `jennifer-full` release artifact - a build *variant* of the default binary, not a third supported brand. The binary ladder becomes `jennifer-tiny` (DBs stubbed) ? `jennifer` (MySQL + Postgres) ? `jennifer-full` (+ SQLite). The dependency break from "libraries stay dependency-free" is already accepted; SQLite adds size, not a new principle. **TinyGo** is the one place SQLite is categorically worse, and it is architectural, not a build choice: `modernc.org/sqlite` 's libc emulation (unsafe, goroutines, syscall-level memory management) cannot compile under TinyGo, and no TinyGo-compatible

SQLite exists. Unlike a wire-protocol database - which could in principle be reimplemented as pure `.j` over a net-enabled tiny rebuild - SQLite has no wire protocol and so can *never* reach the embeddable binary. That is the genuinely ironic gap: a local, file-based store is exactly what a minimal embedded target would most want, and it is the one database that binary can't have with current tooling.

- **FCGI.** `use FCGI as web;` library when `net` and `httpd` mature. Lets Jennifer host CGI / FastCGI workloads end-to-end.
- **i18n (CLDR formatting).** Locale-aware case folding, collation, number / date formatting, and BiDi - the CLDR-data-backed half of i18n, distinct from the message catalogs + `%name%` interpolation already shipped as the `intl` library (`M20.4`). Gated on the CLDR-data binary-size question (likely an optional library after the WASM runtime, so locale tables aren't baked into every build).

Runtime and tooling

- **tinygo_devtools build tag.** The dev subcommands (`tokens` / `ast` / `fmt` / `lint` / `profile` / `test`) are `!tinygo` for binary *size* , not compatibility - they are TinyGo-clean Go. A `//go:build !tinygo || tinygo_devtools` constraint (stub as `tinygo && !tinygo_devtools`) plus a `make build-tinygo-dev` target would let them run under the actual TinyGo runtime - e.g. to `profile` a TinyGo-specific perf or stack issue in situ. Pairs with the depth metric above: together they are "TinyGo runtime introspection." Deferred - build-tag complexity across ~6 files and a larger dev-tiny binary, for a diagnostic reached for only occasionally.
- **Build-time library selection.** Choose which system (Go) libraries are baked into a binary at compile time. Motivated by `jennifer-tiny` size (an embedded target needing only `io` + `math` shouldn't carry `net` / `regex` / `hash`) and by opt-in niche Go libraries that don't merit defaulting. The install point is already consolidated - every entry path (`run` / `repl` / `profile` / `test` and the test harnesses) calls `internal/stdlib.InstallAll` , so a library is one line there - and that is the seam a build-tag scheme would cut along: gate each entry behind `//go:build lib_net` (or a `minimal` / `full` profile) and grow `make build-minimal` / `make build TAGS=...` , exactly like the existing `!tinygo` dev-tool split. **Compile-time only** - Go's `plugin` package is Linux/macOS-only and unsupported by TinyGo, and dynamic linking contradicts `jennifer-tiny` 's no-hosted-runtime goal, so PHP-style loadable `.so` extensions are out. Two caveats to design for: (1) a trimmed build breaks the "any `.j` runs on any binary" portability promise (`use net;` becomes a runtime error), so the default build stays full and trimmed builds are an explicit opt-out - ideally with a `meta` -level "is library X present?" query for graceful degradation; (2) CI grows a couple of profiles (default / minimal), not 2^N. Complementary to, not a substitute for, the module system: `.j` -level extensibility (community / uncommon libraries writable in Jennifer) is the module system's job with zero binary cost; build-time selection is only for the curated Go-level core.
- **Binary AST cache (.jc files).** Pre-parsed loading for big programs and embedded scripting hosts. Its own effort when it lands - file-format design, versioning, and TinyGo-safe serialization are enough work to merit dedicated treatment. The text JSON form via `jennifer ast` is the placeholder until then.
- **Compile to Go (AOT).** Transpile the resolved AST to Go source and let the Go toolchain build a native binary - the pragmatic ahead-of-time path, reusing the existing lexer / parser / resolver front-end. It wins where a from-scratch native backend (an LLVM one, say) loses: Go's GC, goroutine scheduler, and the **entire Go-hosted stdlib** come for free, so the huge re-hosting cost (`net` / TLS / `crypto` / `sql` / `regex` / ...) never appears. The mappings are natural `defer` to `defer` , `throw` to error returns. The catch is the language's dynamism (untyped method returns, `meta.call` by name, opaque `json.Value`): a naive transpile keeps boxed tagged-union values, so it beats the tree-walker but is not C-speed without type-specialisation work. Relates to `DRAFT#17` (the in-process bytecode VM is the cheaper dispatch win) and `DRAFT#2` (WASM, the other compiled target); a hand-rolled LLVM backend is the most expensive, least differentiated version of this idea and stays a non-goal. Aside: `jennifer-tiny` already reaches native through LLVM transitively - TinyGo is an LLVM-based Go compiler.
  - value semantics to struct copies, `spawn` to a goroutine over a snapshot,
- **Advanced scheduling knobs.** CPU affinity, work-stealing pool sizing, NUMA awareness, `GOMAXPROCS` -equivalent runtime tuning. Runtime-config surface for the spawn scheduler, not new language features. Ships when a real use case forces it (the default - "let Go's scheduler decide" - handles every workload we've imagined so far).

- **Profiler: heap-per-position metric.** Out of scope for now: `--allocs` already proxies value-copy churn, and true per-position RSS needs `runtime.ReadMemStats` sampling, which is coarse under TinyGo.

Wild ideas

- **Inline assembler.**

Design stances

The seven decisions below shape every feature in Jennifer. They are deliberately uncompromising - "convenience" is rejected when it creates parallel ways to do the same thing, or hides what the code does. Every feature proposal is evaluated against these; a feature that violates a stance needs a strong justification (and, if turned down, an entry in technical/rejected.md). A feature that ships despite *appearing* to violate a stance gets a reasoning record in technical/design-decisions.md.

#	Stance	What it rules in / out
1	One way per thing.	Reject sugar that creates parallel APIs (no ++/--, no +=, no two printf flavors for the same job). One canonical form is easier to read than three convenient ones.
2	Explicit over implicit.	Sigils mark use-site references ($x), def carries the type, libraries are imported per topic (use io;; nothing auto-loads), conditions must be bool (no truthiness), conversions are spelled out (convert.toInt(v), convert.toFloat(v)). Nothing important hides.
3	Presentation, not transformation, in format strings.	printf verb modifiers shape how a value is rendered (%d|base=2, %f|prec=4). Transforming the value itself (upper, substring, markdown rendering) is a library call. Keeps printf small and orthogonal to the rest of the standard library.
4	Strict at boundaries.	Undefined math, missing map keys, out-of-bounds reads, and type mismatches are positioned runtime errors. No NaN, no silent garbage.
5	Value semantics for collections.	Lists and maps copy on assignment and on parameter binding - no aliasing. const is deep: it rejects both rebinding and content mutation at any depth.
6	No shadowing.	A name binds once in any visible scope. Inner scopes inherit outer bindings but cannot redeclare them.
7	Topic-based, opt-in libraries.	The standard library is split by topic, never bundled. Every library is enabled explicitly with use NAME; - no library auto-loads.

Jennifer glossary

This page is the authoritative list of project terminology. When more than one word could reasonably name the same concept (function vs method, library vs module, list vs array), the column **Term** is the one used everywhere in docs, comments, commit messages, and code identifiers. The other words are listed in **Meaning** so a search finds this page.

Pre-1.0 the list grows as the language does; new terms slot in alphabetically.

Term (singular / plural)	Kind	Meaning
bool / bools	type	Boolean primitive. Two values: true, false. Comparison operators produce one.
break	language	The break; statement; exits the innermost enclosing loop. Misuse outside a loop is a positioned runtime error. Not "leave", "stop".
builtin / builtins	implementation	A function or constant registered by a library via the Go-side Register* API. Distinguishes "ships with the interpreter" from user-defined func.
bytes	type	Mutable byte sequence. Indexing yields int in [0, 255]; writes accept the same range and reject anything else. Value-typed (deep-copy on assignment / parameter binding). Built from a string via convert.bytesFromString(s, "utf-8") or grown from def b as bytes; via $b[] = byte;. Not "buffer", "bytestring", "blob".
constant / constants	language	A name bound once at declaration with def const NAME ... and never reassigned. Constants are deep-immutable for lists, maps, and structs.
continue	language	The continue; statement; skips to the next iteration of the innermost loop. In C-style for, the step expression still runs before re-checking the condition. Not "next", "skip".
core library	historical	Removed. Formerly the single auto-loaded library that shipped len as a bare-name global; len is now a language built-in and the version constants moved to meta. Writing use core; produces a friendly migration error.
defer	language	The defer CALL(args); statement; schedules a single call to run when the enclosing block exits, on every exit path (fall-through, return, break, continue, throw, exit), last-registered-first. Arguments are evaluated at the defer line; the call runs at block exit. Jennifer's cleanup construct - there is no finally. Not "destructor", "finalizer".
discard	language	The task.discard($t) call; marks a task fire-and-forget so the exit-time loud-fail scan skips it. One of the ways to observe a task (alongside task.wait and task.waitAll). Not "drop", "detach".
errdefer	language	The errdefer CALL(args); statement; like defer but the call runs only when the enclosing block exits with a propagating error (a throw or a runtime error) - skipped on fall-through, return, break, continue, and exit. The undo half of an acquire whose resource must survive on success (a connect handshake). Shares one LIFO teardown stack with defer.
Error	type / value	The auto-hoisted struct Error { kind, message, file, line, col } that throw conventionally raises and try / catch binds; a runtime error is wrapped into it on entry to a catch. User code may not redefine it. Not "exception", "panic".
exit	language	The exit; (code 0) or exit EXPR; (int code) statement. Terminates the whole program; skips every caller frame and remaining top-level statement. Distinct from return (method-scoped) and not catchable by try.
float / floats	type	64-bit floating-point primitive. Mixed int/float arithmetic promotes to float.
function / functions	language	A named callable defined with func NAME(...) { ... }. Not "method" or "procedure". The func keyword is the canonical source.

global	implementation	A builtin registered via RegisterGlobal / RegisterGlobalConst; reachable as a bare name (no lib. prefix). The API is still on the interpreter but no shipping library uses it - every library is namespaced, and the polymorphic len is a language keyword. Kept for tests until a cleanup pass removes it.
import	language	The import "path.j" [as ALIAS]; statement that loads a .j file as a module: a real module boundary with run-once init and ALIAS.member access. Not include (textual splice) or use (system library).
include	language	The include "path.j"; statement that textually splices a file's tokens at the include site. Not "import" - import is the module system.
int / ints	type	64-bit signed integer primitive.
library / libraries	implementation	A topic-grouped set of builtins shipped inside the interpreter binary, written in Go. Enabled per topic via use NAME;. Not "package" or "module".
list / lists	type	Ordered, 0-indexed, mutable sequence. Element type fixed at declaration. Not "array", "vector", "sequence".
map / maps	type	Insertion-ordered key->value container, mutable. Key and value types fixed at declaration. Not "dictionary", "hashmap", "object".
method	non-term	Not used. Jennifer calls these "functions". The word "method" appears only when discussing other languages' OO methods.
module / modules	implementation	A distributable, Jennifer-coded library written in .j source and loaded via import "modules/foo.j" as foo;. Lives under modules/ in the source tree and ships separately from the interpreter binary. Not "package", "bundle", "plugin".
namespace / namespaces	language	The prefix a library introduces at call sites (os.getEnv(), lists.push(...)). Activated by use NAME; or remapped by use NAME as ALIAS;.
null	type / value	A type with a single value (null). Returned by bodyless return; and by a method that runs to the end.
object	type / kind	An opaque, library-owned value (e.g. json.Value) that carries data reachable only through the owning library's accessors, not through operators or [i] / .field. convert.typeOf reports the generic "object"; convert.objectType the specific registered name. Not "map", "struct".
observed	project	A task whose outcome (value or error) has been claimed via task.wait, task.waitAll, or task.discard. Only unobserved error-producing tasks trigger the exit-time loud-fail. Not "seen", "consumed".
overlay	project	A white-box test file MODULE_test.j that jennifer test splices after its module, so the tests reach the module's private functions by bare name. Every shipped module carries one that must pass 100%. Not "fixture", "harness".
parameter / parameters	language	A function's input slot, declared name as type in the function header. Not "argument" (which is the value passed at the call site).
repeat ... until	language	Post-test loop: repeat { ... } until (cond);. Body runs at least once, then until is checked after each pass; loop stops when cond is true. Chosen over do { } while !cond; so the condition reads as "loop until done."
sigil	language	The $ prefix on a variable use site. Marks "read or assign a mutable binding"; constants and functions do not take the sigil.
spawn	language	The spawn { ... } block expression. Runs its body concurrently with the rest of the program and returns immediately with a task of T handle. Captures its enclosing scope by deep copy (value-semantics capture). Not "async", "go", "launch".
stance / stances	project	One of the seven design rules. Tie-breakers for ambiguous design choices.
statement / statements	language	A ;-terminated unit of source: def, assignment, control flow, expression statement, include, use.

string / strings	type	Immutable UTF-8 text primitive. Both "..." and '...' are valid delimiters.
struct / structs	type	A composite value type with named, typed fields, declared def struct Name { field as type, ... };. Field access via $s.field. Value-typed and deep-const. No methods (see docs/technical/rejected.md).
task / tasks	type	A task of T handle to a spawned computation. Wraps either a pending goroutine, a final value, or a final error. Copies share the underlying handle (the one exception to Jennifer's value-semantics rule). Observed via the task library.
throw	language	The throw EXPR; statement that raises a catchable error (convention: the Error struct). Caught by try / catch. Not control flow - exit / return / break / continue are not catchable.
try ... catch	language	try { body } catch (NAME) { handler } runs body, and on a throw (or a runtime error, wrapped into Error) binds the value to NAME and runs handler. No finally, no typed catch. Not "rescue", "except".
use	language	The use NAME [as ALIAS]; statement that activates a system library's namespace at the call site. Not the same as include (textual splice) or import (module load).
variable / variables	language	A mutable, typed name bound by def NAME as TYPE [init EXPR];. Read and written through the $ sigil.

When you introduce a term that could plausibly mean different things across documents, add a row here in the same change. When you find an existing doc using a non-canonical synonym ("array" for "list", "package" for "library", "method" for "function"), the fix is to rewrite the doc to use the term in this table, not to add an alias.