

Jennifer

Jennifer is a small, experimental, interpreted programming language written in (Tiny)Go and ships as two binaries:

- **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).

But small is not bare. Jennifer is batteries-included: a broad standard library and a growing set of distributable modules cover what real programs actually need, so you build genuine tools, not toys. Text handling has full [regular expressions](#); structured data flows through [JSON](#); email is a complete stack - [SMTP](#) to send, [POP3](#) and [IMAP](#) to receive; in-memory data stores come through [Redis](#) and [memcached](#) clients; the web runs from an ergonomic [REST client](#) to turnkey integrations such as [Gotify](#) push notifications; and lightweight [concurrency](#) is built into the language via `spawn` and the `task` library. Browse the full [library catalog](#) and [module catalog](#) - both grow with every release.

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));
```

Write Jennifer with your editor and an AI assistant

Syntax highlighting ships in [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](#). The entire documentation in one file, regenerated from these pages on every build - handy for reading or searching offline.

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
<code>jennifer</code>	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 <code>spawn</code> parallelism is involved; see technical/tinygo.md > Single-binary benchmark results) and the reliable choice for multi-core parallel <code>spawn</code> . Required for <code>os.run</code> / <code>os.spawn</code> / <code>os.wait</code> / <code>os.poll</code> / <code>os.kill</code> and the whole <code>net</code> library.
<code>jennifer-tiny</code>	TinyGo	Constrained variant. Smaller binary, embeddable in minimal-footprint deployments (embedded systems, minimal containers, small-footprint scripting hosts). Missing <code>os/exec</code> (TinyGo runtime gap) and the network stack (no netdev driver). Also run-only: the <code>tokens</code> / <code>ast</code> / <code>fmt</code> / <code>lint</code> / <code>profile</code> / <code>test</code> development subcommands live only in the default binary. Calls into any of these surfaces return a friendly error pointing back at <code>jennifer</code> .

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](#), 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` 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](#):

```
# 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/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
```

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](#), 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.** No installer, man pages, MIME registration, or shell completion off Linux - the archive holds the executable plus `JENNIFER.md`, `README.md`, and the licence.

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](#) 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
```

`fmt` writes the formatted source to stdout. To rewrite in place, use your shell: `jennifer fmt foo.j > foo.j.new && mv foo.j.new foo.j`. 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/`. 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` and `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` (scope `source.jennifer`) from a thin language extension.
- **bat / Sublime Text** - the native `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` 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](#).

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 small 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](#).

Jennifer - User Guide

Jennifer is a small, experimental, interpreted programming language. 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](https://docs.jennifer-lang.org/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](https://docs.jennifer-lang.org/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".

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.)

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
<code>int</code>	<code>42</code> , <code>0xff</code> , <code>0o755</code> , <code>0b1010_0110</code> , <code>1_000</code>	<code>0</code>	64-bit signed; <code>_</code> may separate digits
<code>float</code>	<code>3.14</code> , <code>0.5</code> , <code>1_000.000_5</code>	<code>0.0</code>	64-bit; promoted from <code>int</code> in mixed math
<code>string</code>	<code>"hello"</code> , <code>'single quotes'</code>	<code>""</code>	Supports escape sequences
<code>bool</code>	<code>true</code> , <code>false</code>	<code>false</code>	Produced by comparison operators
<code>null</code>	<code>null</code>	<code>null</code>	A type with a single value (the unit)
<code>bytes</code>	<i>(no literal)</i>	<code>empty</code>	Mutable byte sequence; <code>element = int in [0, 255]</code> ; built via <code>convert.bytesFromString</code> or grown with <code>\$b[] = byte</code> ;
<code>list of T</code>	<code>[1, 2, 3]</code>	<code>[]</code>	Ordered sequence; 0-indexed; mutable
<code>map of K to V</code>	<code>{"a": 1, "b": 2}</code>	<code>{}</code>	Key → value; insertion-ordered; mutable
<code>user struct</code>	<code>Point{x: 1, y: 2}</code> (after <code>def struct Point ...;</code>)	every field zero	Named fixed set of typed fields; see Structs
<code>task of T</code>	<i>(no literal - produced by <code>spawn { ... }</code>)</i>	<i>(cannot be defaulted; must be initialised)</i>	Handle to a concurrent computation; observed via the task library. See Concurrency

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.

String escape sequences

Both `"..."` and `'...'` are valid string delimiters. The following escapes are recognized:

Escape	Meaning
<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	tab
<code>\\</code>	backslash
<code>\"</code>	double quote
<code>\'</code>	single quote
<code>\0</code>	null character

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
```

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.

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 3x3 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:

```
# 2x2 grid - two rows of two columns
def gridA as list of list of int init [[1, 2], [3, 4]];

# 3x3 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]` 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".

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.

Control flow

Operators

Operator	Meaning
<code>+</code>	addition (<code>int</code> / <code>float</code>); also concatenation on <code>string</code>
<code>-</code> , <code>*</code>	subtraction, multiplication (<code>int</code> / <code>float</code>)
<code>/</code>	true division - always returns <code>float</code>
<code>//</code>	floor (integer) division; <code>int // int -> int</code>
<code>%</code>	modulo (<code>int</code> only)
unary <code>-</code>	numeric negation (<code>int</code> / <code>float</code>)
<code><</code> , <code>></code> , <code><=</code> , <code>>=</code>	numeric comparison; result is <code>bool</code>
<code>==</code>	equality; same-kind plus <code>int</code> / <code>float</code> promotion; <code>bool</code>
<code>and</code> , <code>or</code>	logical; both operands <code>bool</code> ; short-circuit
<code>not</code>	unary logical negation; operand <code>bool</code>
<code>&</code> , <code>^</code>	<code>&</code> , <code>^</code>
<code><<</code> , <code>>></code>	left / arithmetic right shift on <code>int</code>
unary <code>~</code>	bitwise NOT on <code>int</code> (<code>~x == -x - 1</code>)

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;`.

(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]);
}
```

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.

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`.

break and continue

`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.

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
```

`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` in v1

Jennifer does not have a `finally` clause yet. The pattern is “do the cleanup explicitly in both branches” until a real cleanup need surfaces (probably with file handles in a future `fs` library).

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)	<code>task.wait(\$t)</code>
Check whether it's done yet	<code>task.poll(\$t)</code>
Fire and forget	<code>task.discard(\$t)</code>
Wait for a list of tasks	<code>task.waitAll(\$ts)</code>
First-to-finish from a list	<code>task.waitAny(\$ts)</code>

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

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
<code>io</code>	<code>use io;</code>	<code>io.printf</code> , <code>io.sprintf</code> , <code>io.readLine</code> , <code>io.eof</code> , and the for loop language
<code>convert</code>	<code>use convert;</code>	<code>convert.toInt</code> , <code>convert.toFloat</code> , <code>convert.toString</code> , <code>convert.typeOf</code> - explicit casts
<code>math</code>	<code>use math;</code>	<code>math.abs</code> , <code>math.min</code> , <code>math.max</code> , <code>math.sqrt</code> , <code>math.pow</code> , <code>math.round</code> , <code>math.rand</code> , <code>math.randint</code> , <code>math.randSeed</code> ; <code>math.E</code>
<code>strings</code>	<code>use strings;</code>	<code>strings.upper</code> , <code>lower</code> , <code>contains</code> , <code>startsWith</code> , <code>endsWith</code> , <code>trimLeft</code> , <code>trimRight</code> , <code>replace</code> , <code>repeat</code> , <code>substring</code> , <code>split</code>
<code>json</code>	<code>use json;</code>	RFC 8259 <code>json.encode</code> / <code>encodePretty</code> / <code>decode</code> ; structs and map to objects, <code>bytes</code> to base64, integral numbers decode to float, <code>decode</code> yields generic values (no map-to-struct coercion)
<code>toml</code>	<code>use toml;</code>	RFC-conformant TOML 1.0 <code>toml.encode</code> / <code>encodePretty</code> / <code>decode</code> ; <code>toml.Value</code> + <code>read</code> / <code>walk</code> / <code>write</code> surface as <code>json</code> (JSON Pointer)

Library	Enable with	Contents
		<code>toml.AsDatetime</code> (backed by <code>time.Time</code>) for date-times; the ships, not INI
<code>lists</code>	<code>use lists;</code>	<code>lists.push</code> , <code>pop</code> , <code>first</code> , <code>last</code> , <code>head</code> , <code>tail</code> , <code>reverse</code> , <code>sort</code> , <code>concat</code> , <code>slice</code> , <code>shuffle</code> , <code>range</code> - non-mutating helpers
<code>maps</code>	<code>use maps;</code>	<code>maps.keys</code> , <code>values</code> , <code>has</code> , <code>delete</code> , <code>merge</code> - non-mutating helpers
<code>os</code>	<code>use os;</code>	<code>os.Getenv</code> / <code>os.SetEnv</code> , <code>os.HasFlag</code> , <code>os.Flag</code> , <code>os.Run</code> / <code>spawn</code> , <code>constants</code> <code>os.PLATFORM</code> , <code>os.ARCH</code> , <code>os.EOL</code> , <code>os.DIRSEP</code> , <code>os.F</code>
<code>meta</code>	<code>use meta;</code>	<code>meta.VERSION</code> , <code>meta.BUILD</code> - interpreter-self-identity constants
<code>time</code>	<code>use time;</code>	instants, durations, calendar accessors, fixed-offset zones, strings, ISO; structs <code>time.Time</code> , <code>time.Duration</code> , <code>time.Zone</code> ; constants
<code>hash</code>	<code>use hash;</code>	<code>hash.compute(b, algo)</code> for "md5" / "sha1" / "sha256"; streams <code>hash.stream / update / finalize</code> ; struct <code>hash.Stream</code>
<code>archive</code>	<code>use archive;</code>	tar / zip containers over <code>bytes</code> (no <code>fs</code>): <code>archive.pack / unpack</code> for "tar" / "zip" / "tar.gz"; <code>bundle</code> is a list of <code>archive.Entry</code> (mode, mtime)
<code>compress</code>	<code>use compress;</code>	byte-stream compression: <code>pack / unpack</code> for "gzip" / "zlib" ("fast" / "default" / "best" level) + streaming (<code>compress.stream / update / finalize</code>); struct <code>compress.Stream</code>
<code>crc</code>	<code>use crc;</code>	<code>crc.compute(b, algo)</code> for "crc32" / "crc64" (big-endian by default); <code>crc.stream / update / finalize</code> ; struct <code>crc.Stream</code>
<code>encoding</code>	<code>use encoding;</code>	<code>isAscii</code> / <code>lenBytes</code> / <code>lenRunes</code> introspection; <code>toText</code> / <code>fromText</code> for "hex" / "base64" / "base64-url"; <code>encode / decode</code> for character sets "ascii" / "iso-8859-1" / "windows-1252" / "ebcdic"
<code>task</code>	<code>use task;</code>	<code>observe / join</code> task of T handles from <code>spawn { ... }.tasks</code> ; <code>task.discard</code> , <code>task.waitAll</code> , <code>task.waitAny</code>
<code>fs</code>	<code>use fs;</code>	filesystem I/O. Whole-file read/write/append (<code>readString</code> / <code>readBytes</code> / <code>writeString</code> / <code>writeBytes</code> / <code>append</code>), metadata (<code>exists</code> / <code>isFile</code> / <code>isDir</code> / <code>stat</code>), directory ops (<code>mkdir</code> / <code>mkdirAll</code> / <code>remove</code> / <code>removeAll</code> / <code>rename</code> / <code>list</code> / <code>walk</code>), streams (<code>open</code> / <code>readLine</code> / <code>readChars</code> / <code>readBytes</code> / <code>writeString</code> / <code>writeBytes</code>); structs <code>fs.Stat</code> , <code>fs.File</code>
<code>net</code>	<code>use net;</code>	TCP + UDP sockets and DNS. TCP (<code>connect</code> / <code>listen</code> / <code>accept</code> / <code>readBytes</code> / <code>writeBytes</code> / <code>eof</code> , <code>UDP</code> (<code>listenUDP</code> / <code>sendTo</code> / <code>recvFrom</code>), DNS (<code>lookup</code> / <code>reverseLookup</code> / <code>close</code> / <code>address</code>); structs <code>net.Conn</code> , <code>net.Listener</code> , <code>net.UDPConn</code>

Library	Enable with	Contents
		<code>net.Datagram</code> . Supported on the default <code>jennifer</code> binary; the <code>jennifer-tiny</code> returns friendly errors from every entry point
<code>httpd</code>	<code>use httpd;</code>	HTTP/1.1 server engine over <code>net/http</code> (TLS, HTTP/2, graceful shutdown). <code>httpd.listen / accept / respond</code> ; request accessors <code>method / path / query / header / body / remoteAddr</code> ; <code>setHeader</code> ; <code>serveFile / serveDir</code> , <code>shutdown</code> ; structs <code>httpd.Server</code> , <code>httpd.Request</code> . <code>jennifer</code> only; <code>jennifer-tiny</code> returns friendly errors
<code>regex</code>	<code>use regex;</code>	regular expressions over <code>string</code> (RE2 syntax). <code>matches / find / findAll / replace / split / escape</code> + <code>regex</code> . <code>Match</code> and named captures. Implicit LRU cache for compiled patterns in <code>matches</code>
<code>testing</code>	<code>use testing;</code>	test-runner primitives. <code>run / results / reset / report</code> + test runner <code>runtime</code> errors, <code>throws</code> , and <code>exit</code> inside test bodies. Three reporters: <code>"text"</code> , <code>"tap"</code> , <code>"junit"</code> . Foundation for the <code>.j</code> -side testing
<code>uuid</code>	<code>use uuid;</code>	RFC 9562 UUIDs. <code>uuid.generate("v4") / generate("v7")</code> + <code>parse / isValid / version</code> + constant <code>NIL</code> . Version tag is a string from <code>math</code> 's seedable source (not crypto-grade)

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/](#) 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).

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 = $i + 1)`.
- **No space inside parentheses:** `io.printf("hi")`, not `io.printf( "hi" )`.
- **One space between a keyword and its** `(: if (cond), while (cond), for (...)`. Function calls don't get this space: `io.printf(...)`.
- **No space inside `[ ]` or `{ }` list/map literals:** `[1, 2, 3]`, `{"a": 1, "b": 2}`, not `[ 1, 2, 3 ]` or `{ "a" : 1 }`. Empty literals are `[]` and `{}`. Same rule as `()`.
- **No space before `[`:** `$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`), `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 declarations expand to one field per line.** `def struct Point { x as int, y as int };` reflows to

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

Struct *literals* (`Point{x: 1, y: 2}`) stay inline - they read like map literals and the two form styles match.

- **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.

Line length

- **Soft limit: 100 columns.** **fmt** doesn't hard-wrap arbitrary code (that would risk changing meaning), but it will break at the binary joiners `+`, `and`, and `or` when a line has already grown past 100 columns. The break lands *after* the joiner - the operator hangs at end-of-line and the continuation is indented one level deeper:


```
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. Long argument lists, deeply nested calls, and everything else stay on one line unless you break them by hand; `fmt` isn't going to guess where.

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. Identifiers are `[A-Za-z]{1,64}` - no digits, no underscores.
- **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

- **Prefer double quotes:** `"hello"` over `'hello'`. Both forms parse escape sequences the same way, but mixing styles in one file reads as noise. Use single quotes only when the

string contains a `"` you'd otherwise need to escape.

- **Escape sequences are explicit:** `"\n"`, `"\t"`, `"\\"`. Don't rely on multi-line string literals - Jennifer doesn't have them.

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.

Source file conventions

- **`.j` extension** for all Jennifer source. The interpreter rejects anything else.
- **One SPDX header at the top** of every committed `.j` file (the project uses `LGPL-3.0-only` - see existing examples).
- **`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 (SPDX line, copyright, file description, 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](#) will enforce the spacing and file-encoding rules above automatically:

```
# .editorconfig
root = true

[*.]
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

```
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: 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 the shebang on line 1 all survive a `fmt` round-trip. The SPDX header (`# SPDX-License-Identifier: ...`) 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 <code>include</code> could not be spliced (preprocess error)
L004	a malformed or unknown-ID <code># lint-disable</code> comment

ID	Flags
L101	a local variable declared but never used
L102	code after a <code>return</code> / <code>throw</code> / <code>exit</code> / <code>break</code> / <code>continue</code>
L103	an empty <code>catch</code> block (an error caught and silently thrown away)
L104	a <code>throw</code> of something that isn't an <code>Error</code> struct
L105	a condition that is always true or always false (<code>if (\$x == \$x)</code> , ...)
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 <code>use core;</code>)

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.

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.**

Examples

The repository's `examples/` directory 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.

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	TinyGo	Contents
convert	<code>use convert;</code>	full	<code>convert.toInt</code> , <code>convert.toFloat</code> , <code>convert.toInt64</code> , <code>convert.typeOf</code> - explicit casts; canonical-only
archive	<code>use archive;</code>	full	<code>tar</code> / <code>zip</code> containers over <code>bytes</code> (no <code>fs</code>). <code>archive</code> is a list of <code>mode, mtime</code>
compress	<code>use compress;</code>	full	byte-stream compression. <code>compress.pack</code> / <code>unpack</code> ("gzip" / "zlib" / "deflate" (bytes in/out, optimal level) + streaming (<code>compress.stream</code> / <code>update</code> / <code>flush</code>) <code>compress.Stream</code>
crc	<code>use crc;</code>	full	<code>crc.compute(b, algo)</code> + streaming (<code>crc.stream</code>) "crc32", "crc64"; output is big-endian bytes; s
encoding	<code>use encoding;</code>	full	introspection (<code>isAscii</code> , <code>lenBytes</code> , <code>lenRunes</code>); l for "hex", "base64", "base64-url"; character "ascii", "iso-8859-1", "windows-1252", "eb

Library	Enable with	TinyGo	Contents
fs	<code>use fs;</code>	full	filesystem I/O. Whole-file <code>readString / readBytes / writeString / writeBy</code> <code>metadata exists / isFile / isDir / stat; dir ops</code> <code>mkdir / mkdirAll / remove / removeAll / rename /</code> <code>open / readLine / readChars / readBytes / writes</code> <code>structs fs.Stat, fs.File</code>
hash	<code>use hash;</code>	full	<code>hash.compute(b, algo) + streaming (hash.str</code> <code>"md5", "sha1", "sha256"; struct hash.Stream</code>
httpd	<code>use httpd;</code>	default only	HTTP/1.1 server engine over <code>net/http</code> (TLS, HTT loop: <code>httpd.listen / accept / respond; request</code> <code>method / path / query / header / body; serveFile</code> <code>jennifer-tiny</code> returns a friendly stub.
io	<code>use io;</code>	full	<code>io.printf, io.sprintf, io.readLine, io.eof</code> language
json	<code>use json;</code>	full	RFC 8259 <code>json.encode / encodePretty / decode .</code> map to objects, bytes to base64, integral numb Decode yields generic values (no map-to-struct c
lists	<code>use lists;</code>	full	<code>lists.push, pop, first, last, head, tail, i</code> <code>concat, slice, shuffle, range</code> - all return a n
maps	<code>use maps;</code>	full	<code>maps.keys, values, has, delete, merge</code> - all r
math	<code>use math;</code>	full	<code>math.abs, min, max, sqrt, pow, floor, ceil</code> <code>randSeed; constants math.PI, math.E</code>
meta	<code>use meta;</code>	full	<code>meta.VERSION, meta.BUILD</code> - interpreter-self-ide
net	<code>use net;</code>	stubs only	TCP <code>connect / listen / accept / readBytes / writ</code> <code>listenUDP / sendTo / recvFrom, DNS lookup / re</code> <code>close / address; structs net.Conn, net.Listen</code> <code>net.Datagram. jennifer-tiny</code> returns friendly binary for real net I/O.
os	<code>use os;</code>	partial	<code>os.getEnv, os.hasFlag, os.flag, os.run, os</code> <code>os.kill; constants os.PLATFORM, os.ARCH, os</code> <code>os.ARGS</code>
regex	<code>use regex;</code>	full	regular expressions over <code>string</code> (RE2 syntax). <code>regex.matches / find / findAll / replace / split</code> with positional and named captures. Implicit LRL
strings	<code>use strings;</code>	full	<code>strings.upper, lower, contains, startsWith</code> <code>trimLeft, trimRight, replace, repeat, subs</code>

(`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.
- 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](https://github.com/robertkrimen/otto/blob/master/technical/interpreter.md) > [Builtins and libraries](#).

For canonical terminology (library vs module, function vs method, list vs array, ...), see [../glossary.md](https://github.com/robertkrimen/otto/blob/master/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
<code>archive .pack(entries, fmt)</code>	Bundle a list of <code>archive.Entry</code> into bytes; <code>fmt</code> "tar" / "zip" / "tar.gz".
<code>archive .unpack(b, fmt)</code>	Read a bundle back into a list of <code>archive.Entry</code> .
<code>compress .finalize(stream)</code>	Close a streaming compressor; returns all compressed bytes.
<code>compress .pack(b, algo [, level])</code>	Compress bytes; <code>algo</code> "gzip" / "zlib" / "deflate", optional level "fast" / "default" / "best".
<code>compress .stream(algo [, level])</code>	Start a streaming compressor -> <code>compress.Stream</code> .
<code>compress .unpack(b, algo)</code>	Decompress bytes with <code>algo</code> .
<code>compress .update(stream, b)</code>	Feed one chunk into a streaming compressor.
<code>convert .fromCodepoint(n)</code>	One-rune string for Unicode code point <code>n</code> (whole range, 1-4 UTF-8 bytes); errors on out-of-range / surrogate.
<code>convert .toBool(v)</code>	Canonical conversion to <code>bool</code> (<code>0 / 1</code> , <code>0.0 / 1.0</code> , "true" / "false").
<code>convert .toCodepoint(char)</code>	Unicode code point (int) of a one-rune string; errors unless exactly one code point (not a grapheme cluster).
<code>convert .toFloat(v)</code>	Convert to float (int → float, float identity, string parses, bool → 1.0/0.0).

Call	What it does
<code>convert.ToInt(v)</code>	Convert to int (float truncates toward zero, string parses, bool → 1/0).
<code>convert.ToString(v)</code>	Convert to string (always succeeds; uses the value's display form).
<code>convert.TypeOf(v)</code>	Runtime kind as string ("int", "float", "string", "bool", "null", "list", "map", "object").
<code>convert.ObjectType(v)</code>	Specific registered name of an opaque object (e.g. "json.Value"); errors on a non-object.
<code>crc.compute(b, algo)</code>	One-shot checksum. <code>algo</code> is "crc32" or "crc64" . Returns big-endian bytes (4 or 8).
<code>crc.finalize(\$s)</code>	Final checksum as big-endian bytes; consumes the handle.
<code>crc.stream(algo)</code>	Allocate a <code>crc.Stream</code> for <code>algo</code> ; feed chunks via <code>crc.update</code> then close with <code>crc.finalize</code> .
<code>crc.update(\$s, \$bytes)</code>	Feed one chunk into a <code>crc.Stream</code> (mutates by side effect).
<code>encoding.codecs()</code>	Canonical character-codec names in registration order.
<code>encoding.decode(b, codec)</code>	Decode <code>bytes</code> from a character codec to a Jennifer string.
<code>encoding.encode(s, codec)</code>	Encode a Jennifer string into a character codec's bytes.
<code>encoding.fromText(s, format)</code>	Decode a binary-to-text format. <code>format</code> : "hex", "base32", "base32-hex", "base64", "base64-url", "ascii85", "z85", "quoted-printable" .
<code>encoding.isAscii(b)</code>	True iff every byte in <code>b</code> is < 0x80.
<code>encoding.lenBytes(s)</code>	UTF-8 byte length of <code>s</code> (pair with <code>len(s)</code> for rune count).
<code>encoding.lenRunes(b)</code>	Rune count of valid UTF-8 <code>bytes</code> ; errors on invalid UTF-8.
<code>encoding.toText(b, format)</code>	Encode <code>bytes</code> as printable text. <code>format</code> : "hex", "base32", "base32-hex",

Call	What it does
	"base64", "base64-url", "ascii85", "z85", "quoted-printable".
<code>fs .appendBytes(path, content)</code>	Append bytes to path; creates the file if missing.
<code>fs .appendString(path, content)</code>	Append UTF-8 string to path; creates the file if missing.
<code>fs .close(\$f)</code>	Close an <code>fs.File</code> handle; removes it from the registry.
<code>fs .eof(\$f)</code>	True iff the next read on <code>\$f</code> would error or return partial. Sticky.
<code>fs .exists(path)</code>	True if <code>path</code> resolves; permission errors still surface.
<code>fs .isDir(path)</code>	True iff <code>path</code> exists and is a directory.
<code>fs .isFile(path)</code>	True iff <code>path</code> exists and is a regular file.
<code>fs .list(path)</code>	Sorted entry names in <code>path</code> . Non-recursive; returns list of string.
<code>fs .mkdir(path)</code>	Create a single directory; errors if any parent is missing.
<code>fs .mkdirAll(path)</code>	Create <code>path</code> and every missing parent (like <code>mkdir -p</code>).
<code>fs .open(path, mode)</code>	Open <code>path</code> and return an <code>fs.File</code> . <code>mode</code> : "read", "write", "append".
<code>fs .readBytes(path) / .readBytes(\$f, n)</code>	Whole-file read (1 arg) or up to <code>n</code> bytes from handle (2 args). Partial + sticky-EOF on short handle reads.
<code>fs .readChars(\$f, n)</code>	Up to <code>n</code> runes from handle, UTF-8 decoded. Partial + sticky-EOF on short reads.
<code>fs .readLine(\$f)</code>	One line from handle, <code>\r\n / \n</code> stripped. Errors on EOF - check <code>fs.eof</code> first.
<code>fs .readString(path)</code>	Whole file as UTF-8; invalid UTF-8 is a positioned runtime error.
<code>fs .remove(path)</code>	Delete one file or empty directory. Non-empty dir errors.
<code>fs .removeAll(path)</code>	Recursive delete. Explicit second verb (no-footguns stance).

Call	What it does
<code>fs .rename(old, new)</code>	Same-filesystem rename; cross-fs is a boundary error.
<code>fs .stat(path)</code>	Returns <code>fs.Stat (path, size, isDir, mtimeNanos, mode)</code> . Missing path errors.
<code>fs .walk(path)</code>	Depth-first, sorted, includes <code>path</code> . Returns list of <code>fs.Stat</code> . Skips symlinks.
<code>fs .writeBytes(path, content) / .writeBytes(\$f, b)</code>	Whole-file overwrite (path form) or write via handle (fs.File form).
<code>fs .writeString(path, content) / .writeString(\$f, s)</code>	Whole-file overwrite (path form) or write via handle (fs.File form).
<code>hash .compute(b, algo)</code>	One-shot digest. <code>algo</code> is "md5", "sha1", "sha256", or "sha512". Returns raw bytes.
<code>hash .hmac(key, message, algo)</code>	Keyed-hash MAC (RFC 2104) over the same algorithms; raw bytes out. For JWT / TOTP / SigV4 / webhook signatures.
<code>hash .finalize(\$s)</code>	Final digest as bytes; consumes the handle (later calls error).
<code>hash .stream(algo)</code>	Allocate a <code>hash.Stream</code> for <code>algo</code> ; feed chunks via <code>hash.update</code> then close with <code>hash.finalize</code> .
<code>hash .update(\$s, \$bytes)</code>	Feed one chunk into a <code>hash.Stream</code> (mutates by side effect).
<code>httpd .listen(addr) / .listenTLS(addr, cert, key)</code>	Start an HTTP / HTTPS server -> <code>httpd.Server (":0" = ephemeral port)</code> . Default binary only.
<code>httpd .address(\$srv) / .shutdown(\$srv)</code>	Bound address of a server / graceful drain (unblocks parked <code>accept</code>).
<code>httpd .accept(\$srv)</code>	Block for the next request -> <code>httpd.Request</code> (the pull loop). Errors once the server is shut down.
<code>httpd .method(\$req) / .path(\$req) / .query(\$req, name) / .header(\$req, name) / .body(\$req) / .remoteAddr(\$req)</code>	Read the accepted request (<code>query / header -> ""</code> if absent; <code>body -> bytes</code>).
<code>httpd .setHeader(\$req, name, value) / .respond(\$req, status, body)</code>	Set a response header / send the response once (<code>body</code> is string or bytes).

Call	What it does
<code>httpd .serveFile(\$req, path) / .serveDir(\$req, root)</code>	Answer with a file / the file under <code>root</code> for the request path (<code>..</code> cannot escape <code>root</code>).
<code>io .eof()</code>	True if and only if the next <code>io.readLine()</code> would error. Pair with <code>while (not io.eof()) {...}</code> .
<code>io .printf(format, args...)</code>	Format-string write to stdout. Verbs: <code>%d %f %s %t %v %%</code> ; per-verb <code> key=value</code> modifiers (<code>pad</code> , <code>prec</code> , <code>base</code> , <code>null=*</code> , ...).
<code>io .printf(value)</code>	Write a value's display form to stdout.
<code>io .eprintf(format, args...)</code>	Like <code>printf</code> , but writes to stderr (diagnostics / logs that must not mix into stdout).
<code>io .readLine()</code>	Read one line from stdin (trailing newline stripped). Errors at EOF - check <code>io.eof()</code> first.
<code>io .readLine(prompt)</code>	Same as <code>io.readLine()</code> but writes <code>prompt</code> to stdout first.
<code>io .sprintf(format, args...)</code>	Format-string version of <code>sprintf</code> . Same verbs and <code> key=value</code> modifiers as <code>printf</code> .
<code>io .sprintf(value)</code>	Display-form of a value, returned as a string (doesn't write).
<code>len(v)</code> (<i>language built-in</i>)	Structural length: rune count (string), element count (list), entry count (map), byte count (bytes).
<code>json .decode(s)</code>	Parse JSON text into an opaque <code>json.Value</code> handle (walk it with the accessors below).
<code>json .encode(v)</code>	Compact JSON string for an encodable value (struct/map -> object, <code>bytes</code> -> base64, <code>json.Value</code> round-trips; task / non-string keys error).
<code>json .encodePretty(v)</code>	Like <code>encode</code> , 2-space indented.
<code>json .typeOf(v[, ptr])</code>	JSON type at an optional JSON Pointer: <code>null</code> <code>bool</code> <code>int</code> <code>float</code> <code>string</code> <code>list</code> <code>map</code> .

Call	What it does
<code>json .get(v[, ptr])</code>	Sub-node at a JSON Pointer, as a <code>json.Value</code> (walk stays opaque; no pointer = the node itself).
<code>json .has(v, ptr)</code>	Whether the JSON Pointer resolves to an existing node.
<code>json .keys(v[, ptr])</code>	<code>list of string</code> keys of the addressed map, in document order.
<code>json .length(v[, ptr])</code>	Element count of a list / entry count of a map at the pointer.
<code>json .asInt(v[, ptr]) / asFloat / asString / asBool</code>	Extract the addressed leaf as a typed value (strict; <code>asFloat</code> promotes an integral number).
<code>json .isNull(v[, ptr])</code>	Whether the addressed node is JSON <code>null</code> .
<code>json .map() / .list()</code>	A fresh empty JSON map / list <code>json.Value</code> - the explicit start of a document (writes never auto-vivify).
<code>json .set(v, ptr, val)</code>	Non-mutating: upsert a map key or replace an in-range list index; returns a new <code>json.Value</code> . Strict (no missing intermediates).
<code>json .insert(v, ptr, val)</code>	Insert into a list before index <code>ptr</code> (or <code>-</code> = at end); returns a new handle.
<code>json .append(v, ptr, val)</code>	Push onto the list addressed by <code>ptr</code> (sugar for insert at <code>/.../-</code>).
<code>json .remove(v, ptr)</code>	Drop the map key or list element at <code>ptr</code> ; returns a new handle.
<code>json .move(v, from, to)</code>	Relocate the subtree at <code>from</code> to <code>to</code> (read, remove, then <code>set</code>).
<code>lists .concat(a, b)</code>	New list with <code>a</code> 's elements followed by <code>b</code> 's.
<code>lists .contains(xs, item)</code>	True if <code>item</code> appears in <code>xs</code> (haystack, needle).
<code>lists .first(xs)</code>	Element at index 0. Empty input errors.
<code>lists .head(xs, n)</code>	New list of the first <code>n</code> elements.
<code>lists .last(xs)</code>	Element at the last index. Empty input errors.

Call	What it does
<code>lists .pop(xs)</code>	New list without the last element. Empty input errors.
<code>lists .push(xs, item)</code>	New list with <code>item</code> appended.
<code>lists .range(start, end[, step])</code>	Half-open list of consecutive ints; <code>end</code> excluded; <code>step</code> must match direction.
<code>lists .reverse(xs)</code>	New list with elements reversed.
<code>lists .shuffle(xs)</code>	Fisher-Yates; respects <code>math.randSeed</code> . Non-mutating.
<code>lists .slice(xs, start[, end])</code>	New sublist <code>[start, end)</code> ; <code>end</code> defaults to <code>len(xs)</code> .
<code>lists .sort(xs)</code>	New ascending-sorted list. Numeric / string / bool elements; mixed errors.
<code>lists .tail(xs, n)</code>	New list of the last <code>n</code> elements.
<code>maps .delete(m, key)</code>	New map without <code>key</code> . Missing key errors (strict at boundaries).
<code>maps .has(m, key)</code>	True if map <code>m</code> contains <code>key</code> . The non-erroring companion to <code>\$m[key]</code> .
<code>maps .keys(m)</code>	List of keys in insertion order.
<code>maps .merge(a, b)</code>	New map; <code>b</code> 's entries layered on top of <code>a</code> .
<code>maps .values(m)</code>	List of values in insertion order.
<code>math .abs(x)</code>	Absolute value of <code>x</code> (int → int, float → float).
<code>net .accept(\$listener)</code>	Block until a client connects to <code>\$listener</code> ; return the new <code>net.Conn</code> .
<code>net .address(\$h)</code>	Polymorphic. <code>Conn</code> -> peer address; <code>Listener</code> / <code>UDPSocket</code> -> local bound address.
<code>net .close(\$h)</code>	Polymorphic. Closes a <code>net.Conn</code> , <code>net.Listener</code> , OR <code>net.UDPSocket</code> .
<code>net .connect(address)</code>	TCP client: dial <code>"host:port"</code> and return a <code>net.Conn</code> .
<code>net .connectTLS(address)</code>	TLS client: dial <code>"host:port"</code> + handshake, verifying the cert against the host. <code>net.TLSOptions</code> for <code>caCert</code> / <code>skipVerify</code> .

Call	What it does
<code>net .startTLS(\$conn)</code>	Upgrade an open plaintext <code>net.Conn</code> to TLS in place (STARTTLS); host reused from connect; same handle.
<code>net .eof(\$conn)</code>	True iff the next read on <code>\$conn</code> would return partial or fail. Sticky.
<code>net .listen(address)</code>	Bind TCP "host:port" (use " :0 " for ephemeral). Returns a <code>net.Listener</code> .
<code>net .listenUDP(address)</code>	Bind a UDP socket. Returns a <code>net.UDPConn</code> ; usable as both client and server.
<code>net .lookup(host)</code>	DNS: resolve host to a list of string IPs.
<code>net .readBytes(\$conn, n)</code>	Read up to <code>n</code> bytes; blocks for at least one byte. Sticky-EOF on close.
<code>net .recvFrom(\$sock, n)</code>	Block for one UDP datagram, up to <code>n</code> bytes. Returns <code>net.Datagram{data, peer}</code> .
<code>net .reverseLookup(ip)</code>	Reverse DNS: IP address to a list of string of hostnames.
<code>net .sendTo(\$sock, peer, bytes)</code>	Send one UDP datagram to peer ("host:port").
<code>net .setDeadline(\$conn, ms)</code>	Arm a read/write deadline <code>ms</code> ms out (0 clears). A read past it fails with a catchable <code>read timed out</code> .
<code>net .writeBytes(\$conn, bytes)</code>	Blocking write of every byte to a <code>net.Conn</code> .
<code>regex .escape(s)</code>	Escape RE2 metacharacters so <code>s</code> matches literally when used as a pattern.
<code>regex .find(pattern, s)</code>	First match as <code>regex.Match</code> ; sentinel with <code>start=-1</code> if no match.
<code>regex .findAll(pattern, s)</code>	Every non-overlapping match; returns list of <code>regex.Match</code> .
<code>regex .matches(pattern, s)</code>	True iff <code>pattern</code> matches somewhere in <code>s</code> .
<code>regex .replace(pattern, s, replacement)</code>	Replace every match. <code>\$1</code> , <code>\${name}</code> expand to captured groups; <code>\$\$</code> is a literal <code>\$</code> .

Call	What it does
<code>regex .split(pattern, s)</code>	Split <code>s</code> at every match; returns list of string.
<code>math .ceil(x)</code>	Smallest int $\geq x$. Accepts int (identity) or float.
<code>math .floor(x)</code>	Largest int $\leq x$. Accepts int (identity) or float.
<code>math .max(a, b)</code>	Larger of two numbers; mixed int/float promotes to float.
<code>math .min(a, b)</code>	Smaller of two numbers; mixed int/float promotes to float.
<code>math .pow(x, y)</code>	<code>x</code> raised to <code>y</code> ; always float. Errors on NaN/Inf-producing inputs.
<code>math .round(x)</code>	Round to nearest int (half away from zero).
<code>math .sqrt(x)</code>	Square root; always float. Errors on negative input.
<code>os .flag(name)</code>	Value following <code>name</code> in <code>os.ARGV</code> , or "" if absent / at end. Exact-match (no <code>--foo=bar</code> parsing).
<code>os .getEnv(name)</code>	Read environment variable <code>name</code> . Unset $\rightarrow$ empty string, no error.
<code>os .setEnv(name, value)</code>	Set environment variable <code>name</code> for this process (and children it spawns). Invalid name errors.
<code>os .hasFlag(name)</code>	True if <code>name</code> appears as an exact element of <code>os.ARGV</code> .
<code>os .isTerminal(stream)</code>	Is <code>stream</code> ("stdout" / "stderr" / "stdin") an interactive terminal? Pipe/file $\rightarrow$ false.
<code>os .cwd()</code>	Absolute path of the current working directory.
<code>os .homeDir()</code>	Current user's home directory (<code>\$HOME</code> / <code>%USERPROFILE%</code>).
<code>os .tempDir()</code>	Temp-file directory (<code>\$TMPDIR</code> / <code>/tmp</code> ; <code>%TMP%</code> on Windows). Never errors.
<code>os .kill(p)</code>	Send SIGTERM to spawned process <code>\$p</code> .

Call	What it does
<code>os .poll(p)</code>	True if spawned process <code>\$p</code> has exited (a following <code>os.wait</code> returns immediately).
<code>os .run(argv)</code>	Blocking: run <code>argv</code> to completion, return <code>os.Result{exitCode, stdout, stderr}</code> .
<code>os .spawn(argv)</code>	Non-blocking: start <code>argv</code> , return <code>os.Process{pid}</code> handle.
<code>os .wait(p)</code>	Block until spawned process <code>\$p</code> exits; return <code>os.Result.Idempotent</code> .
<code>strings .chars(s)</code>	Split <code>s</code> into a list of string, one entry per Unicode code point.
<code>testing .assertContains(hay, needle)</code>	Throw <code>Error{kind:"assertion"}</code> unless <code>hay</code> contains <code>needle</code> : substring / list element / map key.
<code>testing .assertEqual(actual, expected)</code>	Throw unless deeply equal (lists / maps / structs compare by value).
<code>testing .assertFalse(cond)</code>	Throw unless <code>cond</code> (a bool) is false.
<code>testing .assertNotEqual(actual, expected)</code>	Throw unless not deeply equal.
<code>testing .assertThrows(name, kind)</code>	Throw unless the named zero-arg method throws an <code>Error</code> of that <code>kind</code> .
<code>testing .assertTrue(cond)</code>	Throw unless <code>cond</code> (a bool) is true.
<code>testing .report(results, format)</code>	Render results to "text", "tap", or "junit" (returns string).
<code>testing .reset()</code>	Clear the process-wide result accumulator.
<code>testing .results()</code>	Snapshot of the accumulator as list of <code>testing.Result</code> .
<code>testing .run(name)</code>	Invoke a zero-arg user method by name; catch every failure mode into a <code>testing.Result</code> .
<code>testing .runWith(name, args)</code>	Like <code>run</code> , binding the <code>args</code> list to the method's parameters (arity + type checked).
<code>strings .contains(s, sub)</code>	True if <code>s</code> contains the substring <code>sub</code> .
<code>strings .endsWith(s, suffix)</code>	True if <code>s</code> ends with <code>suffix</code> .
<code>strings .indexOf(s, sub)</code>	Rune index of first <code>sub</code> in <code>s</code> , or <code>-1</code> if absent.

Call	What it does
<code>strings .join(parts, sep)</code>	Concatenate list of string parts separated by <code>sep</code> . Inverse of <code>strings.split</code> .
<code>strings .lower(s)</code>	Lowercase <code>s</code> (Unicode-aware).
<code>strings .repeat(s, n)</code>	<code>n</code> non-negative copies of <code>s</code> concatenated.
<code>strings .replace(s, old, new)</code>	Replace all occurrences of <code>old</code> in <code>s</code> with <code>new</code> .
<code>strings .split(s, sep)</code>	Split <code>s</code> on non-empty <code>sep</code> ; returns list of string.
<code>strings .startsWith(s, prefix)</code>	True if <code>s</code> starts with <code>prefix</code> .
<code>strings .substring(s, start)</code>	Rune-indexed slice of <code>s</code> from <code>start</code> to end.
<code>strings .substring(s, start, end)</code>	Rune-indexed slice; exclusive <code>end</code> .
<code>strings .trim(s)</code>	Strip leading and trailing Unicode whitespace.
<code>strings .trimLeft(s)</code>	Strip leading whitespace.
<code>strings .trimRight(s)</code>	Strip trailing whitespace.
<code>strings .upper(s)</code>	Uppercase <code>s</code> (Unicode-aware).
<code>task .discard(\$t)</code>	Mark a <code>task</code> of <code>T</code> fire-and-forget; suppresses exit-time loud-fail. Returns null.
<code>task .poll(\$t)</code>	True if <code>\$t</code> has finished (non-blocking).
<code>task .wait(\$t)</code>	Block until <code>\$t</code> finishes; return its value or re-raise its error.
<code>task .waitAll(\$ts)</code>	Block for all tasks in <code>\$ts</code> ; results in list order; re-raises the first error if any.
<code>task .waitAny(\$ts)</code>	Block until any task in <code>\$ts</code> is done; return its index.
<code>time .add(\$t, \$d)</code>	<code>time.Time</code> shifted by duration <code>\$d</code> .
<code>time .after(\$a, \$b)</code>	True if <code>\$a</code> is strictly later than <code>\$b</code> .
<code>time .before(\$a, \$b)</code>	True if <code>\$a</code> is strictly earlier than <code>\$b</code> .
<code>time .day(\$t)</code>	Day of month, 1-31.
<code>time .equal(\$a, \$b)</code>	True if <code>\$a</code> and <code>\$b</code> are the same UTC instant.

Call	What it does
<code>time .format(\$t, layout)</code>	Strftime-style format. Codes: %Y %m %d %H %M %S %z %a %A %b %B %j %u %%. .
<code>time .fromHours(n)</code>	<code>time.Duration</code> of <code>n</code> hours.
<code>time .fromIso(s)</code>	Parse RFC 3339; accepts <code>z</code> or <code>+HH:MM</code> ; optional fractional seconds.
<code>time .fromMilliseconds(n)</code>	<code>time.Duration</code> of <code>n</code> milliseconds.
<code>time .fromMinutes(n)</code>	<code>time.Duration</code> of <code>n</code> minutes.
<code>time .fromSeconds(n)</code>	<code>time.Duration</code> of <code>n</code> seconds.
<code>time .fromUnix(seconds)</code>	<code>time.Time</code> at the given Unix second.
<code>time .fromUnixMillis(ms)</code>	<code>time.Time</code> at the given Unix millisecond.
<code>time .fromUnixNanos(ns)</code>	<code>time.Time</code> at the given Unix nanosecond.
<code>time .hour(\$t)</code>	Hour 0-23.
<code>time .hours(\$d)</code>	Span as whole hours (int).
<code>time .inZone(\$t, \$z)</code>	Re-render <code>\$t</code> in <code>\$z</code> 's wall-clock; UTC instant is preserved.
<code>time .iso(\$t)</code>	RFC 3339 string: <code>z</code> for UTC, <code>+HH:MM</code> otherwise; fractional seconds when non-zero.
<code>time .local()</code>	Host's current <code>time.Zone</code> (name + offset).
<code>time .milliseconds(\$d)</code>	Span as whole milliseconds (int).
<code>time .minute(\$t)</code>	Minute 0-59.
<code>time .minutes(\$d)</code>	Span as whole minutes (int).
<code>time .month(\$t)</code>	Calendar month, January = 1.
<code>time .nanosecond(\$t)</code>	Fractional second, 0-999_999_999.
<code>time .now()</code>	Current instant in the host's local zone (<code>time.Time</code>).
<code>time .parse(s, layout)</code>	Strict strftime-style parse. Same code set as format (<code>%j</code> / <code>%u</code> are format-only).
<code>time .second(\$t)</code>	Second 0-59.
<code>time .seconds(\$d)</code>	Span as whole seconds (int).
<code>time .sleep(\$d)</code>	Block the running task for <code>\$d</code> . Negative / zero returns immediately. Returns null.

Call	What it does
<code>time .sub(\$a, \$b)</code>	Signed <code>time.Duration</code> between two <code>time.Time</code> values.
<code>time .unix(\$t)</code>	Unix-second instant of <code>\$t</code> (int).
<code>time .unixMillis(\$t)</code>	Unix-millisecond instant of <code>\$t</code> (int).
<code>time .unixNanos(\$t)</code>	Unix-nanosecond instant of <code>\$t</code> (int).
<code>time .utc()</code>	Current instant in UTC (<code>time.Time</code>).
<code>time .weekday(\$t)</code>	ISO 8601 weekday: Monday = 1 ... Sunday = 7.
<code>time .year(\$t)</code>	Calendar year (int).
<code>time .zone(offset, name)</code>	Build a <code>time.Zone</code> from an integer offset (seconds east of UTC) and a display name.
<code>toml .decode(s)</code>	Parse TOML text into an opaque <code>toml.Value</code> handle (walk it with the accessors below).
<code>toml .encode(v) / .encodePretty(v)</code>	TOML string for a <code>toml.Value</code> (or native map / list / scalar); <code>encodePretty</code> blank-lines sections. Null value / non-table root errors.
<code>toml .typeOf(v[, ptr])</code>	Node type at an optional JSON Pointer: <code>null bool int float string list map datetime</code> .
<code>toml .get(v[, ptr])</code>	Sub-node at a JSON Pointer, as a <code>toml.Value</code> (walk stays opaque; no pointer = the node itself).
<code>toml .has(v, ptr)</code>	Whether the JSON Pointer resolves to an existing node.
<code>toml .keys(v[, ptr]) / .length(v[, ptr])</code>	<code>list of string</code> table keys in document order / element count of a list or table.
<code>toml .asInt(v[, ptr]) / asFloat / asString / asBool</code>	Extract the addressed leaf as a typed value (strict; <code>asFloat</code> promotes an int).
<code>toml .asDatetime(v[, ptr]) / .isDatetime(v[, ptr])</code>	A date-time node as a <code>time.Time</code> (needs <code>use time;</code>) / whether the node is a date-time.
<code>toml .map() / .list()</code>	A fresh empty table / array <code>toml.Value</code> - the explicit start of a document (writes never auto-vivify).

Call	What it does
<code>toml .set(v, ptr, val) / .insert / .append / .remove / .move</code>	Non-mutating edits by JSON Pointer; each returns a new <code>toml.Value</code> (strict / no missing intermediates).
<code>uuid .generate(v)</code>	New UUID string; <code>v</code> is "v4" (random) or "v7" (time-ordered).
<code>uuid .isValid(s)</code>	Whether <code>s</code> is a well-formed UUID string.
<code>uuid .parse(s)</code>	The 16 bytes of a UUID string; errors on malformed input.
<code>uuid .version(s)</code>	Version digit (4, 7, ...; 0 for NIL); errors on malformed input.

Constants

Name	Type	Value
<code>math .E</code>	<code>float</code>	Euler's number, 2.718281828459045.
<code>math .PI</code>	<code>float</code>	π , 3.141592653589793.
<code>meta .BUILD</code>	<code>string</code>	Which Go toolchain compiled the interpreter: "go" / "tinygo".
<code>meta .VERSION</code>	<code>string</code>	The interpreter's build version (e.g. "0.14.0").
<code>meta .SYSMODDIR</code>	<code>string</code>	Resolved system module directory (<code>-sysmoddir > JENNIFER_SYSMODDIR ></code> compile default).
<code>meta .call(name, args...)</code>	<code>value</code>	Invoke a top-level method by runtime name (arity + types checked); errors / <code>exit</code> propagate.
<code>meta .defined(name)</code>	<code>bool</code>	Whether a top-level method <code>name</code> exists.
<code>meta .callMain(name, args...) / .definedMain(name)</code>	<code>value / bool</code>	Like <code>call</code> / <code>defined</code> but against the entry program's methods (a module reaching its host's handlers).
<code>os .ARCH</code>	<code>string</code>	CPU architecture: "amd64", "arm64", "wasm", ...
<code>os .ARGS</code>	<code>list of string</code>	Argv. Index 0 is the script path, the rest are user args.

Name	Type	Value
<code>os .DIRSEP</code>	<code>string</code>	Path-component separator: <code>"/</code> Unix, <code>"\\"</code> Windows.
<code>os .EOL</code>	<code>string</code>	Platform line ending. <code>"\n"</code> Unix-likes, <code>"\r\n"</code> Windows.
<code>os .NCPU</code>	<code>int</code>	Logical CPUs usable by the process (<code>runtime.NumCPU</code>). <code>1</code> on <code>jennifer-tiny</code> (single-thread scheduler).
<code>os .PATHSEP</code>	<code>string</code>	PATH-list separator: <code>":"</code> Unix, <code>;"</code> Windows.
<code>os .PLATFORM</code>	<code>string</code>	OS tag: <code>"linux"</code> , <code>"darwin"</code> , <code>"windows"</code> , ...
<code>time .PROGRAM_START</code>	<code>time.Time</code>	Captured the moment the time library installed; “since program launched” anchor.
<code>time .UTC</code>	<code>time.Zone</code>	Canonical UTC: <code>Zone{offset: 0, name: "UTC"}</code> .
<code>uuid .NIL</code>	<code>string</code>	The all-zero UUID <code>00000000-0000-0000-0000-000000000000</code> .

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
<code>archive.pack(entries, format)</code>	<code>bytes</code>	Bundle a <code>list of archive.Entry</code> .
<code>archive.unpack(b, format)</code>	<code>list of archive.Entry</code>	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`).

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.

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
<code>compress.pack(b, algo [, level])</code>	<code>bytes</code>	Compress; <code>algo</code> is "gzip" / "zlib" / "deflate".
<code>compress.unpack(b, algo)</code>	<code>bytes</code>	Decompress; same <code>algo</code> .
<code>compress.stream(algo [, level])</code>	<code>compress.Stream</code>	Start a streaming compressor.
<code>compress.update(stream, b)</code>	<code>null</code>	Feed one chunk of input.
<code>compress.finalize(stream)</code>	<code>bytes</code>	Close and return the full compressed output.

`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
<code>convert.toInt(v)</code>	int / float / string / bool	identity / truncate / parse / <code>true</code> =1, <code>false</code> =0
<code>convert.toFloat(v)</code>	int / float / string / bool	convert / identity / parse / <code>true</code> =1.0, <code>false</code> =0.0
<code>convert.toString(v)</code>	any	always succeeds; uses the value's display form
<code>convert.toBool(v)</code>	bool / int / float / string	identity / canonical only (0 / 1 , 0.0 / 1.0 , "true" / "false")
<code>convert.typeOf(v)</code>	any	returns the kind as a string: "int" , "float" , ..., "list" , "map" , "object"
<code>convert.objectType(v)</code>	object	specific registered name of an opaque object, e.g. "json.Value" ; errors on a non-object

Call	Source kinds	Behavior
<code>convert.bytesFromString(s, codec)</code>	(string, string)	string to bytes; "utf-8" only (other codecs live in <code>encoding</code>)
<code>convert.stringFromBytes(b, codec)</code>	(bytes, string)	bytes to string; "utf-8" only; invalid UTF-8 is an error
<code>convert.toCodepoint(char)</code>	string	Unicode code point (int) of a one-rune string; errors unless the argument is exactly one code point
<code>convert.fromCodepoint(n)</code>	int	the one-rune string for code point <code>n</code> ; 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 <code>crc32.IEEE</code>).
"crc64"	8 bytes	Go <code>crc64.ECMA</code> (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
<code>crc.compute(b, algo)</code>	<code>bytes</code>	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
<code>crc.stream(algo)</code>	<code>crc.Stream</code>	Allocate a fresh handle for the named algorithm.
<code>crc.update(\$s, \$bytes)</code>	<code>null</code>	Feed one chunk. Mutates the handle's state by side effect.
<code>crc.finalize(\$s)</code>	<code>bytes</code>	Compute the checksum and consume the handle. 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.

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, and quoted-printable, `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($src, "hex"));           # 636166633a9
io.printf("%t\n", encoding.isAscii($src));                # false
io.printf("%x\n", encoding.encode("café", "iso-8859-1")); # 63 61 66 e9
```

Introspection

Call	Returns	Notes
<code>encoding.isAscii(b)</code>	<code>bool</code>	True iff every byte is <code>< 0x80</code> . Empty bytes is true.
<code>encoding.lenBytes(s)</code>	<code>int</code>	Byte length of <code>s</code> 's UTF-8 encoding. Pair with <code>len(s)</code> (runes).
<code>encoding.lenRunes(b)</code>	<code>int</code>	Rune count of valid UTF-8 <code>bytes</code> ; 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
<code>encoding.toText(b, fmt)</code>	<code>string</code>	Encode <code>bytes</code> as printable text.
<code>encoding.fromText(s, fmt)</code>	<code>bytes</code>	Decode back to <code>bytes</code> .

Formats

<code>fmt</code>	Standard	Notes
<code>"hex"</code>	base-16	Lowercase on encode; decode accepts upper and lower case. Two chars per byte.
<code>"base32"</code>	RFC 4648 §6	Standard alphabet (<code>A - Z</code> <code>2 - 7</code>), <code>=</code> padding.
<code>"base32-hex"</code>	RFC 4648 §7	Extended-hex alphabet (<code>0 - 9</code> <code>A - V</code>), <code>=</code> padding.
<code>"base64"</code>	RFC 4648 §4	Standard alphabet (<code>+</code> <code>/</code> , <code>=</code> padding).
<code>"base64-url"</code>	RFC 4648 §5	URL / filename-safe alphabet (<code>-</code> <code>_</code>).
<code>"ascii85"</code>	Adobe / btoa	Base-85, the <code>! .. u</code> alphabet, with <code>z</code> for an all-zero group.
<code>"z85"</code>	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).
<code>"quoted-printable"</code>	RFC 2045	MIME transfer encoding (see below).

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`.

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
<code>encoding.encode(s, codec)</code>	<code>bytes</code>	Errors when a rune has no byte in the codec (e.g. € in ASCII).
<code>encoding.decode(b, codec)</code>	<code>string</code>	Errors when a byte is undefined in the codec.
<code>encoding.codecs()</code>	<code>list of string</code>	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
<code>"ascii"</code>	7-bit US-ASCII. Rejects any byte or rune <code>>= 0x80</code> .
<code>"ebcdic"</code>	IBM Code Page 1047 (the Open Systems Latin-1 EBCDIC variant).

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

Codec	Part	Coverage
<code>"iso-8859-1"</code>	Latin-1	Western European (identity <code>U+0000..U+00FF</code>).
<code>"iso-8859-2"</code>	Latin-2	Central / Eastern European.

Codec	Part	Coverage
"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

Codec	Coverage
	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
<code>fs.readString(path)</code>	<code>string</code>	Whole file as UTF-8. Invalid UTF-8 is a positioned runtime error.
<code>fs.readBytes(path)</code>	<code>bytes</code>	Whole file as raw bytes. See below for the handle form.
<code>fs.writeString(path, content)</code>	<code>null</code>	Overwrites. Creates the file if missing. See below for the handle form.
<code>fs.writeBytes(path, content)</code>	<code>null</code>	Overwrites. Creates the file if missing. See below for the handle form.
<code>fs.appendString(path, content)</code>	<code>null</code>	Appends. Creates the file if missing.
<code>fs.appendBytes(path, content)</code>	<code>null</code>	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
<code>fs.exists(path)</code>	<code>bool</code>	True if the path resolves. Permission errors still surface.
<code>fs.isFile(path)</code>	<code>bool</code>	True iff the path exists and is a regular file. False for missing.
<code>fs.isDir(path)</code>	<code>bool</code>	True iff the path exists and is a directory. False for missing.
<code>fs.stat(path)</code>	<code>fs.Stat</code>	Missing path errors. Pair with <code>fs.exists</code> when tolerating absent files.

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."

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
<code>fs.mkdir(path)</code>	<code>null</code>	Errors if any parent is missing (matches POSIX <code>mkdir</code>).
<code>fs.mkdirAll(path)</code>	<code>null</code>	Creates every missing parent (matches POSIX <code>mkdir -p</code>).
<code>fs.remove(path)</code>	<code>null</code>	Removes one file or one empty directory. Non-empty dir errors.
<code>fs.removeAll(path)</code>	<code>null</code>	Recursive delete. Explicit second verb.
<code>fs.rename(old, new)</code>	<code>null</code>	Same-filesystem rename; cross-fs is a boundary error.
<code>fs.list(path)</code>	list of string	Sorted entry names. Non-recursive.
<code>fs.walk(path)</code>	list of <code>fs.Stat</code>	Depth-first, sorted, includes <code>path</code> 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
```

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. <code>fs.readLine</code> , <code>fs.readChars</code> , <code>fs.readBytes</code> allowed.

Mode	Semantics
"write"	Write, create+truncate. <code>fs.writeString</code> , <code>fs.writeBytes</code> allowed.
"append"	Write, create+append. Same write ops as "write".

Unknown mode strings error with the known set listed.

Handle surface

Call	Returns	Notes
<code>fs.open(path, mode)</code>	<code>fs.File</code>	Opens per the mode string.
<code>fs.close(\$f)</code>	<code>null</code>	Removes the handle from the registry; later ops on the id error.
<code>fs.readLine(\$f)</code>	<code>string</code>	One line; <code>\r\n</code> / <code>\n</code> stripped. Errors on EOF - check <code>fs.eof</code> first.
<code>fs.readChars(\$f, n)</code>	<code>string</code>	Up to <code>n</code> runes, UTF-8 decoded. Partial result on EOF, sticky-EOF flip.
<code>fs.readBytes(\$f, n)</code>	<code>bytes</code>	Up to <code>n</code> bytes. Partial result on EOF, sticky-EOF flip.
<code>fs.writeString(\$f, s)</code>	<code>null</code>	Read-mode handle errors.
<code>fs.writeBytes(\$f, b)</code>	<code>null</code>	Read-mode handle errors.
<code>fs.eof(\$f)</code>	<code>bool</code>	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.

Polymorphic verbs (path vs handle)

Three verbs are polymorphic on the first argument's kind:

Verb	1-arg form	2-arg form
<code>fs.readBytes</code>	<code>(path) -> whole</code>	<code>(\$f, n) -> partial handle read</code>
<code>fs.writeString</code>	-	<code>(path, content)</code> or <code>(\$f, s)</code>
<code>fs.writeBytes</code>	-	<code>(path, content)</code> or <code>(\$f, b)</code>

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.

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](https://jennifer.tech/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)` .
- **Symlink ops** (`fs.readlink` , `fs.symlink`).
- `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). 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	Broken for collision resistance; useful for integrity / caching.
"sha1"	20 bytes	Broken for collision resistance; still common in legacy formats.
"sha256"	32 bytes	The default choice for new code.
"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", "sha1", "sha256", "sha512"`.

One-shot

Call	Returns	Notes
<code>hash.compute(b, algo)</code>	<code>bytes</code>	Full digest of the entire input.
<code>hash.hmac(key, message, algo)</code>	<code>bytes</code>	Keyed-hash MAC (RFC 2104) over the same algorithms.

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).

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
<code>hash.stream(algo)</code>	<code>hash.Stream</code>	Allocate a fresh handle for the named algorithm.
<code>hash.update(\$s, \$bytes)</code>	<code>null</code>	Feed one chunk. Mutates the handle's state by side effect.
<code>hash.finalize(\$s)</code>	<code>bytes</code>	Compute the digest and consume the handle. 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](#) 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", "sha1", "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](https://github.com/tinygo-org/tinygo/blob/master/docs/technical/tinygo.md).

The pull loop

Jennifer has no first-class functions, so you cannot hand Go a request handler callback. 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
<code>httpd.listen(addr)</code>	<code>httpd.Server</code>	Start listening. <code>"127.0.0.1:8080"</code> (TCP), <code>":0"</code> (ephemeral TCP port), or <code>"unix:/run/app.sock"</code> (a Unix domain socket).
<code>httpd.listenTLS(addr, cert, key)</code>	<code>httpd.Server</code>	HTTPS; <code>cert</code> / <code>key</code> are PEM bytes. HTTP/2 negotiated automatically.
<code>httpd.address(srv)</code>	<code>string</code>	The actual bound address (resolve <code>":0"</code> to the chosen port).
<code>httpd.accept(srv)</code>	<code>httpd.Request</code>	Block for the next request. Errors once the server is shut down.
<code>httpd.method(req)</code>	<code>string</code>	<code>"GET"</code> , <code>"POST"</code> , ...
<code>httpd.path(req)</code>	<code>string</code>	URL path, e.g. <code>/users/42</code> .
<code>httpd.query(req, name)</code>	<code>string</code>	Query parameter (<code>""</code> if absent).
<code>httpd.header(req, name)</code>	<code>string</code>	Request header (<code>""</code> if absent; case-insensitive name).
<code>httpd.body(req)</code>	<code>bytes</code>	The request body (buffered, capped at 10 MiB).
<code>httpd.remoteAddr(req)</code>	<code>string</code>	Client <code>host:port</code> .
<code>httpd.setHeader(req, name, value)</code>	<code>null</code>	Set a response header (before <code>respond</code>).
<code>httpd.respond(req, status, body)</code>	<code>null</code>	Send the response; <code>body</code> is a <code>string</code> OR <code>bytes</code> .

Call	Returns	Notes
<code>httpd.serveFile(req, path)</code>	<code>null</code>	Answer with a file (content type, range requests handled by <code>net/http</code>).
<code>httpd.serveDir(req, root)</code>	<code>null</code>	Answer with the file under <code>root</code> matching the request path (<code>..</code> cannot escape <code>root</code>).
<code>httpd.shutdown(srv)</code>	<code>null</code>	Graceful drain: stop accepting, unblock parked <code>accept</code> 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`; `serveFile` answers with one specific file regardless of the request path.

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:

```
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 configurable limit and explicit read/idle/write timeout knobs are a planned follow-up.
- **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.

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` / `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 <code>null</code>
<code>null=empty</code>	<code>""</code>
<code>null=null</code>	<code>"null"</code>
<code>null=literal("X")</code>	<code>x</code> - 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
<code>pad</code>	non-negative integer	-	minimum rune width
<code>max</code>	non-negative integer	-	truncate to N runes
<code>align</code>	<code>left</code> , <code>right</code> , <code>center</code>	<code>left</code>	which side gets the pad spaces; <code>center</code> splits the pad evenly (odd leftover goes right)
<code>mode</code>	<code>raw</code> , <code>quote</code> , <code>escape</code>	<code>raw</code>	wrap in <code>"..."</code> (<code>quote</code>) / show escapes (<code>escape</code>)

Key	Values	Default	Effect
<code>null</code>	see above	-	substitute when value is <code>null</code>

`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.

<code>io.printf("[%s pad=8 align=right]\n", "hi");</code>	<code># [hi]</code>
<code>io.printf("[%s max=3]\n", "abcdef");</code>	<code># [abc]</code>
<code>io.printf("[%s mode=quote\n", "a\nb");</code>	<code># "a\nb"</code>

%d modifiers

Key	Values	Default	Effect
<code>pad</code>	non-negative integer	-	minimum width
<code>fill</code>	0	space	zero-pad between sign and digits; requires <code>align=right</code> (the default)
<code>align</code>	<code>left</code> , <code>right</code>	<code>right</code>	which side gets the pad
<code>base</code>	2, 8, 10, 16	10	digit base; hex uses lowercase
<code>sign</code>	<code>negative</code> , <code>always</code> , <code>space</code>	<code>negative</code>	sign for non-negative values
<code>group</code>	positive integer	-	digit-group size, reading right-to-left
<code>sep</code>	one of <code>_</code> , <code>,</code> , <code>.</code> , <code>-</code> , <code>:</code>	-	group separator; required with <code>group=</code> and vice versa
<code>null</code>	see above	-	substitute when value is <code>null</code>

<code>io.printf("%d base=2\n", 5);</code>	<code># 101</code>
<code>io.printf("%d base=16 group=4 sep=_\n", 3735928559);</code>	<code># dead_beef</code>
<code>io.printf("%d pad=5 fill=0 sign=always\n", 42);</code>	<code># +0042</code>

%f modifiers

Key	Values	Default	Effect
<code>prec</code>	non-negative integer	shortest	fraction digits (or mantissa fraction digits when <code>sci=true</code>)

Key	Values	Default	Effect
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)

Key	Values	Default	Effect
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.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](..user-guide/index.md), [../technical/interpreter.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
<code>json.encode(v)</code>	<code>string</code>	Compact JSON for any encodable value (including a <code>json.Value</code>).
<code>json.encodePretty(v)</code>	<code>string</code>	Same, 2-space indented; empty lists / maps stay <code>[] / {}</code> .
<code>json.decode(s)</code>	<code>json.Value</code>	Parse JSON text into an opaque handle (see below).

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
<code>json.typeOf(v[, ptr])</code>	<code>string</code>	The node's type: <code>null</code> <code>bool</code> <code>int</code> <code>float</code> <code>string</code> <code>list</code> <code>map</code> .
<code>json.get(v[, ptr])</code>	<code>json.Value</code>	The addressed sub-node, as a <code>json.Value</code> (walk stays opaque).
<code>json.has(v, ptr)</code>	<code>bool</code>	Whether the pointer resolves to an existing node.
<code>json.keys(v[, ptr])</code>	<code>list of string</code>	Keys of the addressed map, in document order.
<code>json.length(v[, ptr])</code>	<code>int</code>	Element count of a list, or entry count of a map.
<code>json.asInt(v[, ptr])</code>	<code>int</code>	A number node with no fractional part. A <code>float</code> node errors.
<code>json.asFloat(v[, ptr])</code>	<code>float</code>	A number node; an integral one promotes (JSON has one number type).
<code>json.asString(v[, ptr])</code>	<code>string</code>	A string node.

Call	Returns	Notes
<code>json.asBool(v[, ptr])</code>	<code>bool</code>	A <code>true</code> / <code>false</code> node.
<code>json.isNull(v[, ptr])</code>	<code>bool</code>	Whether the addressed node is JSON <code>null</code> .

Write surface - every verb is **non-mutating**, returning a fresh `json.Value`; the idiom is `$v = json.set($v, ...)`:

Call	Returns	Notes
<code>json.map()</code>	<code>json.Value</code>	A fresh empty JSON map (object) - the start of a document.
<code>json.list()</code>	<code>json.Value</code>	A fresh empty JSON list (array).
<code>json.set(v, ptr, val)</code>	<code>json.Value</code>	Upsert a map key, or replace an in-range list index, at <code>ptr</code> .
<code>json.insert(v, ptr, val)</code>	<code>json.Value</code>	Insert <code>val</code> into a list before index <code>ptr</code> (or <code>-</code> = at end).
<code>json.append(v, ptr, val)</code>	<code>json.Value</code>	Push <code>val</code> onto the list addressed by <code>ptr</code> (sugar for insert at <code>/.../-</code>).
<code>json.remove(v, ptr)</code>	<code>json.Value</code>	Drop the map key or list element at <code>ptr</code> .
<code>json.move(v, from, to)</code>	<code>json.Value</code>	Relocate the subtree at <code>from</code> to <code>to</code> (read, remove, then <code>set</code>).

Encoding

`json.encode` walks the value and writes its JSON image:

Jennifer	JSON
<code>null</code>	<code>null</code>
<code>bool</code>	<code>true</code> / <code>false</code>
<code>int</code>	integer number
<code>float</code>	number (always with a <code>.</code> or exponent, so it decodes back as <code>float</code>)
<code>string</code>	string (escaped)
<code>bytes</code>	base64 string
<code>list of T</code>	array

Jennifer	JSON
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).

```
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");  
$v = json.set($v, "/tags", json.list());  
$v = json.append($v, "/tags", "admin");  
$v = json.insert($v, "/tags/0", "root");  
$v = json.set($v, "/tags/1", "dev");  
$v = json.remove($v, "/name");  
$v = json.move($v, "/tags", "/roles");  
  
# {}  
# {"name":"ada"}  
# {"name":"ada","tags":[]}  
# ...,"tags":["admin"]  
# ...,"tags":["root","admin"]  
# replace index 1  
# drop a key  
# 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\\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.

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
<code>lists.push(xs, item)</code>	list	New list with <code>item</code> appended. In a build-a-list loop prefer the <code>\$xs[]</code> sugar ($O(N)$ vs this call’s $O(N^2)$).
<code>lists.pop(xs)</code>	list	New list without the last element. Empty input errors.
<code>lists.first(xs)</code>	element kind	Element at index <code>0</code> . Empty input errors.
<code>lists.last(xs)</code>	element kind	Element at the last index. Empty input errors.
<code>lists.head(xs, n)</code>	list	New list of the first <code>n</code> elements. <code>n</code> must be in <code>[0, len(xs)]</code> .

Call	Returns	Notes
<code>lists.tail(xs, n)</code>	list	New list of the last <code>n</code> elements. Same range constraint.
<code>lists.reverse(xs)</code>	list	New list, elements in reverse order.
<code>lists.sort(xs)</code>	list	New list sorted ascending. See “Sort” below.
<code>lists.contains(xs, item)</code>	bool	True iff <code>item</code> appears in <code>xs</code> under structural equality.
<code>lists.concat(a, b)</code>	list	<code>a</code> ’s elements followed by <code>b</code> ’s.
<code>lists.slice(xs, start)</code>	list	Elements from <code>start</code> to end (exclusive <code>end = len(xs)</code>).
<code>lists.slice(xs, start, end)</code>	list	Elements <code>[start, end)</code> . Out-of-range bounds error.
<code>lists.shuffle(xs)</code>	list	New list, elements in uniformly random order. See “Shuffle” below.
<code>lists.range(start, end)</code>	list of int	Half-open: <code>[start, start+1, ..., end-1]</code> . See “Range” below.
<code>lists.range(start, end, step)</code>	list of int	Walks by <code>step</code> while staying strictly before <code>end</code> . See “Range”.

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. Comparator-based sort (`lists.sortBy`) is deferred until methods are first-class values.

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
<code>maps.keys(m)</code>	list of <key type>	The map's keys, in insertion order.
<code>maps.values(m)</code>	list of <value type>	The map's values, in insertion order.
<code>maps.has(m, key)</code>	bool	Key membership test. The non-erroring companion to <code>\$m[key]</code> .
<code>maps.delete(m, key)</code>	map	New map without <code>key</code> . Missing key is an error - see below.
<code>maps.merge(a, b)</code>	map	New map: <code>a</code> 's entries with <code>b</code> 's layered on top (<code>b</code> 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;`. A small set of frequently-needed numeric functions plus the constants `math.PI` and `math.E`. The library is strict on undefined inputs - anything that would produce NaN or Infinity in IEEE arithmetic instead produces a positioned runtime error.

```
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
<code>math.abs(x)</code>	same type as <code>x</code>	<code>int</code> → <code>int</code> , <code>float</code> → <code>float</code> ; errors on <code>math.abs(MinInt64)</code> (no representable result)
<code>math.min(a, b)</code>	<code>int</code> or <code>float</code>	<code>int+int</code> → <code>int</code> ; mixed → <code>float</code>
<code>math.max(a, b)</code>	<code>int</code> or <code>float</code>	same rule as <code>min</code>
<code>math.sqrt(x)</code>	<code>float</code>	errors on negative input
<code>math.pow(x, y)</code>	<code>float</code>	errors if the result would be NaN or Infinity
<code>math.floor(x)</code>	<code>int</code>	toward $-\infty$; accepts <code>int</code> (identity); errors if the result does not fit in a 64-bit <code>int</code> (NaN / Inf / out of range)
<code>math.ceil(x)</code>	<code>int</code>	toward $+\infty$; same <code>int</code> -range error as <code>floor</code>
<code>math.round(x)</code>	<code>int</code>	half-away-from-zero (<code>math.round(2.5) = 3</code>); same <code>int</code> -range error as <code>floor</code>

`min` / `max` follow the same numeric-promotion rule as `+`: same-type arguments return that type; any `float` involved produces a `float`.

Constants

Name	Kind	Value
<code>math.PI</code>	float	3.141592653589793...
<code>math.E</code>	float	2.718281828459045...

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 <code>import s</code> look). Resolved from <code>--sysmoddir > JENNIFER_SYSMODDIR ></code> the compile-time default; <code>jennifer version -v</code> shows the layers.

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"

Repository state	<code>meta.VERSION</code> value
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 (<code>gc</code>) - the default <code>jennifer</code> binary
"tinygo"	Built with TinyGo - the constrained <code>jennifer-tiny</code> 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

Jennifer has no first-class functions: a bare call `greet(...)` is resolved at compile time, so you cannot dispatch 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	
<code>meta.call(name, args...)</code>	the method's return value	Invoke the method <code>name</code> with the given arguments (arity + declared types checked, as at a normal call site).
<code>meta.defined(name)</code>	bool	Whether a method <code>name</code> exists - validate a name before calling it.

Call	Returns	
<code>meta.callMain(name, args...)</code>	the method's return value	Like <code>call</code> , but resolves against the entry program's methods.
<code>meta.definedMain(name)</code>	bool	Like <code>defined</code> , 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`.

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 framework module dispatch to handlers the application defined - the `web` module registers routes against handler names and calls them with `meta.callMain`. 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
<code>net.connect(address)</code>	<code>net.Conn</code>	TCP client. <code>address</code> is "host:port" (Go's convention).
<code>net.listen(address)</code>	<code>net.Listener</code>	Bind and listen. <code>":0"</code> selects an ephemeral port.
<code>net.accept(\$listener)</code>	<code>net.Conn</code>	Blocking accept. Non-blocking use pairs with <code>spawn</code> .
<code>net.readBytes(\$conn, n)</code>	<code>bytes</code>	Blocks for at least one byte; returns whatever's available, capped at <code>n</code> . Sticky-EOF on close.
<code>net.writeBytes(\$conn, b)</code>	<code>null</code>	Blocking write of every byte.
<code>net.setDeadline(\$conn, ms)</code>	<code>null</code>	Arm a read/write deadline <code>ms</code> milliseconds out; <code>0</code> clears it. A read past the deadline fails with a distinguishable <code>read timed out</code> error. Accepts a <code>net.Conn</code> or a

Call	Returns	Notes
		<code>net.UDPSocket</code> (so a <code>recvFrom</code> can time out).
<code>net.eof(\$conn)</code>	<code>bool</code>	Looks ahead: true iff the next read would return partial or fail.
<code>net.address(\$conn)</code>	<code>string</code>	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.

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
<code>net.connectTLS(address)</code>	<code>net.Conn</code>	Dial <code>address</code> ("host:port" , same as <code>net.connect</code>) and complete a TLS handshake (implicit TLS: 465 / 993 / 995, HTTPS).
<code>net.startTLS(\$conn)</code>	<code>net.Conn</code>	Upgrade an open plaintext connection to TLS in place (STARTTLS: 587 / 143 / 110); same handle.

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
<code>net.listenUDP(address)</code>	<code>net.UDPSocket</code>	Bind a UDP port. Usable as both client and server (send from wherever you bound).
<code>net.sendTo(\$sock, peer, bytes)</code>	<code>null</code>	Send one datagram to <code>peer</code> ("host:port").
<code>net.recvFrom(\$sock, n)</code>	<code>net.Datagram</code>	Block for one datagram, up to <code>n</code> 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
<code>net.lookup(host)</code>	list of string	Resolve <code>host</code> to a list of IP addresses (v4 and v6 mixed).
<code>net.reverseLookup(ip)</code>	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.UDPSource` 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](https://tinygo.org/docs/reference/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
<code>os.getenv(name)</code>	string	Reads an environment variable. Unset variables return <code>""</code> , no error.
<code>os.setEnv(name, value)</code>	null	Sets an environment variable for this process (and any child it later spawns). An invalid name (empty, or containing <code>=</code> / <code>NUL</code>) is a positioned error.
<code>os.hasFlag(name)</code>	bool	True if <code>name</code> is an exact-match element of <code>os.ARGS</code> . See “Flag inspection” below.
<code>os.flag(name)</code>	string	The element immediately after <code>name</code> in <code>os.ARGS</code> , or <code>""</code> if absent or at end.
<code>os.run(argv)</code>	<code>os.Result</code>	Blocking. Run <code>argv</code> to completion; capture stdout/stderr. See “External programs” below.
<code>os.spawn(argv)</code>	<code>os.Process</code>	Non-blocking. Start <code>argv</code> , return a handle.

Call	Returns	Notes
<code>os.wait(p)</code>	<code>os.Result</code>	Block until <code>\$p</code> terminates; return captured streams + exit code. Idempotent.
<code>os.poll(p)</code>	<code>bool</code>	Non-blocking; true once <code>\$p</code> has exited (a following <code>os.wait</code> returns immediately).
<code>os.kill(p)</code>	<code>null</code>	Send SIGTERM to <code>\$p</code> .
<code>os.isTerminal(stream)</code>	<code>bool</code>	Is <code>stream</code> (<code>"stdout"</code> / <code>"stderr"</code> / <code>"stdin"</code>) an interactive terminal? See “Terminal detection”.
<code>os.cwd()</code>	<code>string</code>	Absolute path of the current working directory. Errors only if it can't be determined.
<code>os.homeDir()</code>	<code>string</code>	The current user's home directory (<code>\$HOME</code> on Unix, <code>%USERPROFILE%</code> on Windows). Errors if unresolved.
<code>os.tempDir()</code>	<code>string</code>	Directory for temporary files (<code>\$TMPDIR</code> or <code>/tmp</code> on Unix; the <code>%TMP%</code> / <code>%TEMP%</code> location on Windows). Never errors; the directory is not created.

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 {
    exitCode as int,
    stdout   as string,
    stderr   as string
};

def struct os.Process { pid as int };

# not actually written by users -
# the library registers it under
# the `os.` prefix.

# 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);

```

`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. `os.kill($p)` sends SIGTERM; a subsequent `os.wait` returns whatever the OS reports for the terminated child.

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
<code>os.PLATFORM</code>	string	Operating-system name as reported by the runtime: <code>"linux"</code> , <code>"darwin"</code> , <code>"windows"</code> , ...
<code>os.ARCH</code>	string	CPU architecture: <code>"amd64"</code> , <code>"arm64"</code> , <code>"wasm"</code> , ...
<code>os.NCPU</code>	int	Number of logical CPUs usable by the running process (<code>runtime.NumCPU</code>). Portable stdlib, so it stays OS-independent - it reports usable parallelism, not a raw core count: on <code>jennifer-tiny</code> (cooperative single-thread scheduler) it is <code>1</code> . For richer host details (CPU model, RAM), read the OS's own files from Jennifer, e.g. <code>/proc/cpuinfo</code> via <code>fs</code> on Linux, so the interpreter stays portable.
<code>os.EOL</code>	string	Platform line ending. <code>"\n"</code> on Unix-likes, <code>"\r\n"</code> on Windows.
<code>os.DIRSEP</code>	string	Path-component separator: <code>"/"</code> on Unix-likes, <code>"\"</code> on Windows.
<code>os.PATHSEP</code>	string	PATH-list separator (between entries in <code>\$PATH</code>): <code>":"</code> on Unix-likes, <code>;"</code> on Windows.

Process

Name	Kind	Value
<code>os.ARGS</code>	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](#).

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
<code>regex.matches(pattern, s)</code>	<code>bool</code>	Does <code>pattern</code> match anywhere in <code>s</code> ?
<code>regex.find(pattern, s)</code>	<code>regex.Match</code>	First match, or a sentinel with <code>start=-1</code> on no match.
<code>regex.findAll(pattern, s)</code>	list of <code>regex.Match</code>	Every non-overlapping match, left to right.
<code>regex.replace(pattern, s, replacement)</code>	<code>string</code>	Replace every match. <code>\$1</code> , <code>\${name}</code> expand to captured groups.
<code>regex.split(pattern, s)</code>	list of <code>string</code>	Split <code>s</code> at every match of <code>pattern</code> .
<code>regex.escape(s)</code>	<code>string</code>	Escape metacharacters so <code>s</code> 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)                  # named captures (see below)
  groupsNamed as map of string to string
};
```

`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

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
<code>.</code>	Any character except newline (add <code>(?s)</code> for dotall).
<code>^</code> / <code>\$</code>	Start / end of line (with <code>(?m)</code>) or of input (without).
<code>\d</code> <code>\w</code> <code>\s</code>	Digit / word char / whitespace.
<code>\D</code> <code>\W</code> <code>\S</code>	Their complements.
<code>[abc]</code> <code>[a-z]</code>	Character class.
<code>[^abc]</code>	Negated character class.

Pattern	Meaning
<code>a?</code> <code>a*</code> <code>a+</code>	0-or-1, 0-or-more, 1-or-more (greedy).
<code>a??</code> <code>a*?</code>	Lazy variants.
<code>a{n,m}</code>	Bounded repetition.
<code>`a</code>	<code>b`</code>
<code>(...)</code>	Grouping and positional capture.
<code>(?:...)</code>	Grouping without capture.
<code>(?P<name>...)</code>	Named capture.
<code>(?i)</code>	Case-insensitive flag.
<code>(?m)</code>	Multiline (<code>^</code> / <code>\$</code> match at line boundaries).
<code>(?s)</code>	Dotall (<code>.</code> 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.

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("héllö")` 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
<code>strings.upper(s)</code> , <code>strings.lower(s)</code>	string	Case conversion (Unicode-aware)
<code>strings.contains(s, sub)</code>	bool	Substring search
<code>strings.startsWith(s, prefix)</code>	bool	
<code>strings.endsWith(s, suffix)</code>	bool	
<code>strings.indexOf(s, sub)</code>	int	Rune index of first occurrence; -1 if not found
<code>strings.trim(s)</code>	string	Strip leading and trailing whitespace
<code>strings.trimLeft(s)</code> , <code>strings.trimRight(s)</code>	string	One-sided trim
<code>strings.replace(s, old, new)</code>	string	Replace all occurrences of <code>old</code> with <code>new</code>
<code>strings.repeat(s, n)</code>	string	<code>n</code> copies concatenated; <code>n</code> must be non-negative
<code>strings.substring(s, start)</code>	string	Rune-indexed; from <code>start</code> to the end of the string
<code>strings.substring(s, start, end)</code>	string	Rune-indexed; exclusive end
<code>strings.split(s, sep)</code>	list of string	Split on a non-empty separator; preserves empty segments
<code>strings.chars(s)</code>	list of string	One single-rune string per Unicode code point

Call	Returns	Notes
<code>strings.join(parts, sep)</code>	string	Concatenate a list of string with <code>sep</code> 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 = "héllö"`:

- `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.

See also: [../user-guide/index.md](..../user-guide/index.md), [../technical/interpreter.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](https://github.com/ocaml/jennifer/blob/master/doc/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
<code>task.wait(\$t)</code>	<code>T</code>	Block until <code>\$t</code> finishes; return its value, or re-raise its error.
<code>task.poll(\$t)</code>	<code>bool</code>	Non-blocking: true once <code>\$t</code> has completed (value or error available).
<code>task.discard(\$t)</code>	<code>null</code>	Mark <code>\$t</code> fire-and-forget so the exit-time loud-fail skips it. Does not block.
<code>task.waitAll(\$ts)</code>	list of <code>T</code>	Wait for every task in <code>\$ts</code> ; results in list order; re-raises the first error if any.
<code>task.waitAny(\$ts)</code>	<code>int</code>	Block until any task in <code>\$ts</code> is done; return its index. Caller follows up with <code>task.wait</code> .

`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.

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`.

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
<code>testing.run(name)</code>	<code>testing.Result</code>	Look up a zero-arg method by name, call it, catch every failure mode, append.
<code>testing.runWith(name, args)</code>	<code>testing.Result</code>	Like <code>run</code> , but binds the <code>args</code> list to the method's parameters (arity + declared-type checked). For framework dispatchers, not the zero-arg tests the runner discovers.
<code>testing.results()</code>	list of <code>testing.Result</code>	Snapshot of the accumulator. Value semantics - safe to modify.
<code>testing.reset()</code>	<code>null</code>	Clear the accumulator between independent runs.
<code>testing.report(results, format)</code>	<code>string</code>	Render <code>results</code> to <code>"text"</code> , <code>"tap"</code> , or <code>"junit"</code> .

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
<code>testing.AssertEqual(actual, expected)</code>	<code>actual != expected</code> (deep structural equality: lists / maps / structs compare by value).
<code>testing.AssertNotEqual(actual, expected)</code>	<code>actual == expected</code> .
<code>testing.AssertTrue(cond)</code>	<code>cond</code> is <code>false</code> (<code>cond</code> must be <code>bool</code>).
<code>testing.AssertFalse(cond)</code>	<code>cond</code> is <code>true</code> (<code>cond</code> must be <code>bool</code>).
<code>testing.AssertContains(haystack, needle)</code>	<code>needle</code> is absent: substring for a string, element for a list, key for a map (by haystack kind).
<code>testing.AssertThrows(name, kind)</code>	the named zero-arg method doesn't throw, or throws an <code>Error</code> whose <code>kind</code> 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 `throws`, 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 <code>throw</code> whose thrown value wasn't an <code>Error</code> struct.
"assertion" etc.	A <code>throw Error{kind: "assertion", ...}</code> - <code>errorKind</code> mirrors the struct's <code>kind</code> field.
"exit"	The test body called <code>exit</code> . <code>errorMessage</code> is <code>"exit code N"</code> .

Value	Meaning
"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
 - the `jennifer test` subcommand: `discovery`, `--filter`, `--format`, `--isolated`, and the enabling interpreter change.
- [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.

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
<code>time.now()</code>	<code>time.Time</code>	Current instant in the host's local zone.
<code>time.utc()</code>	<code>time.Time</code>	Current instant in UTC (offset = 0).
<code>time.fromUnix(seconds)</code>	<code>time.Time</code>	At UTC. <code>seconds</code> is <code>int</code> .
<code>time.fromUnixMillis(ms)</code>	<code>time.Time</code>	At UTC. <code>ms</code> is <code>int</code> .
<code>time.fromUnixNanos(ns)</code>	<code>time.Time</code>	At UTC. <code>ns</code> is <code>int</code> .
<code>time.fromIso(s)</code>	<code>time.Time</code>	Parse an RFC 3339 string. Accepts <code>Z</code> or <code>+HH:MM</code> , optional fractional seconds.
<code>time.parse(s, layout)</code>	<code>time.Time</code>	Parse using a strftime layout. See Format codes below.
<code>time.fromSeconds(n)</code>	<code>time.Duration</code>	Span of <code>n</code> seconds.
<code>time.fromMilliseconds(n)</code>	<code>time.Duration</code>	Span of <code>n</code> milliseconds.
<code>time.fromMinutes(n)</code>	<code>time.Duration</code>	Span of <code>n</code> minutes.
<code>time.fromHours(n)</code>	<code>time.Duration</code>	Span of <code>n</code> hours.
<code>time.zone(offset, name)</code>	<code>time.Zone</code>	Fixed-offset zone constructor. <code>offset</code> is seconds east of UTC.
<code>time.local()</code>	<code>time.Zone</code>	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
<code>time.unix(\$t)</code>	<code>int</code>	Unix seconds since 1970-01-01T00:00:00Z (truncates toward zero).
<code>time.unixMillis(\$t)</code>	<code>int</code>	Unix milliseconds.
<code>time.unixNanos(\$t)</code>	<code>int</code>	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
<code>time.year(\$t)</code>	full year	E.g. <code>2024</code> . No two-digit year form.
<code>time.month(\$t)</code>	1-12	January = 1.
<code>time.day(\$t)</code>	1-31	Day of month.
<code>time.hour(\$t)</code>	0-23	24-hour clock.
<code>time.minute(\$t)</code>	0-59	
<code>time.second(\$t)</code>	0-59	Whole seconds; sub-second precision lives in <code>nanosecond</code> / <code>unixNanos</code> .
<code>time.nanosecond(\$t)</code>	0-999_999_999	Fractional second.
<code>time.weekday(\$t)</code>	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
<code>time.seconds(\$d)</code>	Whole seconds in the span.
<code>time.milliseconds(\$d)</code>	Whole milliseconds.
<code>time.minutes(\$d)</code>	Whole minutes.

Call	Notes
<code>time.hours(\$d)</code>	Whole hours.

Arithmetic and comparison

Call	Returns	Notes
<code>time.add(\$t, \$d)</code>	<code>time.Time</code>	<code>\$t</code> shifted by <code>\$d</code> . Preserves the time's offset.
<code>time.sub(\$a, \$b)</code>	<code>time.Duration</code>	Signed: negative when <code>\$a</code> is earlier than <code>\$b</code> .
<code>time.before(\$a, \$b)</code>	<code>bool</code>	Strictly earlier (UTC instant compare).
<code>time.after(\$a, \$b)</code>	<code>bool</code>	Strictly later.
<code>time.equal(\$a, \$b)</code>	<code>bool</code>	Same UTC instant (the stored <code>offset</code> is ignored).
<code>time.sleep(\$d)</code>	<code>null</code>	Block the running task for the requested duration. Negative or zero <code>\$d</code> 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
<code>time.zone(offset, name)</code>	<code>time.Zone</code>	<code>offset</code> in seconds east of UTC; capped at +/- 26h.
<code>time.inZone(\$t, \$z)</code>	<code>time.Time</code>	Re-render <code>\$t</code> in <code>\$z</code> 's wall-clock. UTC instant is preserved.
<code>time.local()</code>	<code>time.Zone</code>	Host's current zone. Reads <code>time.Now().Zone()</code> once.
<code>time.UTC</code> (constant)	<code>time.Zone</code>	<code>Zone{offset: 0, name: "UTC"}</code> . 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
<code>%Y</code>	4-digit year	<code>2024</code>	exactly 4 digits
<code>%m</code>	Month 01-12	<code>06</code>	exactly 2 digits, 1..12

Code	Meaning	Format output	Parse expectation
%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](#) 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
<code>toml.decode(text)</code>	<code>toml.Value</code>	Parse a TOML document into an opaque handle.
<code>toml.encode(v)</code>	<code>string</code>	Render a <code>toml.Value</code> (or a native <code>map</code> / <code>list</code> / <code>scalar</code>) to TOML text.
<code>toml.encodePretty(v)</code>	<code>string</code>	Same, with a blank line separating <code>[table]</code> / <code>[[array]]</code> sections.
<code>toml.typeOf(v[, ptr])</code>	<code>string</code>	Node type: <code>null</code> / <code>bool</code> / <code>int</code> / <code>float</code> / <code>string</code> / <code>list</code> / <code>map</code> / <code>datetime</code> .
<code>toml.get(v[, ptr])</code>	<code>toml.Value</code>	The addressed sub-node, re-wrapped so a walk stays opaque.
<code>toml.has(v, ptr)</code>	<code>bool</code>	Whether the pointer resolves.
<code>toml.keys(v[, ptr])</code>	<code>list of string</code>	Keys of the addressed table, in document order.
<code>toml.length(v[, ptr])</code>	<code>int</code>	Element count of a list, entry count of a table.
<code>toml.asInt(v[, ptr])</code>	<code>int</code>	Strict: a float node errors.
<code>toml.asFloat(v[, ptr])</code>	<code>float</code>	An integer node promotes to float.
<code>toml.asString(v[, ptr])</code>	<code>string</code>	

Call	Returns	Notes
<code>toml.asBool(v[, ptr])</code>	<code>bool</code>	
<code>toml.asDatetime(v[, ptr])</code>	<code>time.Time</code>	Any of the four date-time forms as a <code>time.Time</code> (needs <code>use time;</code>).
<code>toml.isDatetime(v[, ptr])</code>	<code>bool</code>	Whether the addressed node is a date-time.
<code>toml.map()</code>	<code>toml.Value</code>	A fresh empty table, to build a document from scratch.
<code>toml.list()</code>	<code>toml.Value</code>	A fresh empty array.
<code>toml.set(v, ptr, val)</code>	<code>toml.Value</code>	Upsert a table key / replace an in-range list index.
<code>toml.insert(v, ptr, val)</code>	<code>toml.Value</code>	Insert into a list at an index or - (end).
<code>toml.append(v, ptr, val)</code>	<code>toml.Value</code>	Push onto the list at <code>ptr</code> .
<code>toml.remove(v, ptr)</code>	<code>toml.Value</code>	Remove the addressed key / element.
<code>toml.move(v, from, to)</code>	<code>toml.Value</code>	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.

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
<code>uuid.generate(v)</code>	<code>string</code>	New UUID; <code>v</code> is <code>"v4"</code> (random) or <code>"v7"</code> (time-ordered).
<code>uuid.parse(s)</code>	<code>bytes</code>	The 16 bytes of a well-formed UUID string; errors on malformed input.
<code>uuid.isValid(s)</code>	<code>bool</code>	Whether <code>s</code> is a well-formed UUID string.
<code>uuid.version(s)</code>	<code>int</code>	The version digit (4, 7, ...; 0 for <code>uuid.NIL</code>); errors on malformed input.
<code>uuid.NIL</code>	<code>string</code>	The all-zero UUID <code>"00000000-0000-0000-0000-000000000000"</code> .

The version is a **string argument** (`"v4"` / `"v7"`), not a `uuid.v4()` method - Jennifer identifiers are letters-only, so the variant lives in an argument, the same shape as `hash.compute(b, "sha-256")` or `encoding.toText(b, "base64")`.

```
use io;
use uuid;

def id as string init uuid.generate("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

- `"v4"` - fully random. Use for opaque identifiers with no ordering.

- **"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.generate("v7");  
# ... later ...  
def b as string init uuid.generate("v7");  
# $a < $b (string comparison reflects creation order)
```

Randomness caveat

v4/v7 randomness comes from `math`'s shared pseudo-random source - the same stream as `math.rand`, seedable with `math.randSeed`, and therefore **predictable**. Fine for identifiers; **not** for security tokens, session keys, or anything an attacker must not guess. A crypto-grade source lands with the future `crypto` library.

See also

- [math.md](#) - the shared RNG (`randSeed` also reseeds `uuid`).
- [cheatsheet.md](#) - every builtin at a glance.

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](https://jennifer-lang.org/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
<code>amqp</code>	<code>import "amqp.j";</code>	<code>no (net)</code>	an AMQP 0-9-1 client for RabbitMQ over <code>net</code> : <code>connect</code> runs the connection / channel handshake (SASL PLAIN), then <code>declareQueue - > QueueInfo</code> , <code>publish / publishText</code> (method + content-header + body frames), <code>get(c, queue, autoAck) -> Message</code> (synchronous <code>Basic.Get</code> pull), <code>ack</code> , <code>close</code> . Binary frame / method encoding hand-built with the bitwise ops. The largest protocol module. Needs the default binary.
<code>ansi</code>	<code>import "ansi.j";</code>	<code>full</code>	terminal styling as string wrappers. <code>color / bgColor / style / rgb / strip</code> plus per-colour and per-style shortcuts; TTY-aware.
<code>barcode</code>	<code>import "barcode.j";</code>	<code>full</code>	generate scannable codes as images. <code>encode(data, symbology, opts) -> Symbol</code> for <code>qr</code> (Reed-Solomon over GF(256), EC L/M/Q/H, versions 1-10, mask scoring, byte mode) and 1D <code>code128 / code39 / ean13 / ean8 / itf</code> ; render with <code>svg / png</code> (hand-encoded over <code>compress + crc</code>) / <code>terminal / matrix</code> . The GF(256) math lives in a private <code>barcode_ecc.j</code> . No image library. Both binaries.
<code>bloom</code>	<code>import "bloom.j";</code>	<code>full</code>	a Bloom filter (probabilistic set): <code>new(size, hashes)</code> , <code>add / addAll</code> , <code>mightContain</code> - no false negatives, possible false positives. Bits packed in <code>bytes</code> ; <code>k</code> positions

Module	Import with	TinyGo	Contents
			from double-hashing one SHA-256 digest. Value-semantic. Over hash ; both binaries.
bucket	import "bucket.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 .
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. 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.
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) 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 Diagnostic s). Reports drift (a @param naming no real parameter, a parameter with no @param) and orphans; string-literal- and nesting-

Module	Import with	TinyGo	Contents
			correct scanner. Data, not rendering.
<code>dotenv</code>	<code>import "dotenv.j";</code>	full	<code>.env</code> config files: <code>parse(text)</code> / <code>read(path)</code> -> map of string to string, and <code>load(path)</code> (<code>parse</code> + <code>os.setEnv</code> each). Handles # comments, <code>export</code> , single / double quoting, inline comments. Over <code>fs</code> + <code>strings</code> + <code>os</code> .
<code>flatdb</code>	<code>import "flatdb.j";</code>	full	a file-backed JSON document store over <code>json</code> + <code>fs</code> : open a file into a value-semantic DB, query / edit by JSON Pointer (<code>get</code> / <code>has</code> / <code>keys</code> / <code>length</code> / <code>set</code> / <code>append</code> / <code>remove</code>), <code>save</code> with a crash-atomic temp+rename. Crash-atomic snapshotting of small data, not a database engine.
<code>gotify</code>	<code>import "gotify.j";</code>	no (net)	push notifications to a Gotify server, on top of <code>http</code> : <code>push(cfg, title, message, priority)</code> POSTs the message form with the <code>X-Gotify-Key</code> header; value-semantic <code>Config</code> (<code>url</code> + <code>token</code>).
<code>gpio</code>	<code>import "gpio.j";</code>	full	Raspberry-Pi (Linux SBC) GPIO over <code>sysfs</code> (<code>fs</code> is the whole backend): <code>setup(pin, "in"/"out")</code> , <code>write(pin, 0/1)</code> , <code>read(pin)</code> , <code>release(pin)</code> . Stateless, pin-keyed; <code>JENNIFER_GPIO_BASE</code> overrides the <code>sysfs</code> root (tests / mounts). Absent-platform errors are clear, not crashes.
<code>htmlwriter</code>	<code>import "htmlwriter.j";</code>	full	build an HTML element tree and render escaped HTML5. <code>element</code> / <code>text</code> / <code>raw</code> / <code>attr</code> constructors, <code>render</code> / <code>renderAll</code> , <code>escape</code> ; void-element aware. A writer, not a parser.
<code>http</code>	<code>import "http.j";</code>	no (net)	an HTTP/1.1 client over <code>net</code> (<code>https://</code> via TLS): <code>method-</code>

Module	Import with	TinyGo	Contents
			agnostic request plus get / post / put / patch / delete / head / options -> Response (status , headers , body); Content-Length + chunked framing, header case-insensitive lookup. 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)	receive mail (IMAP4rev1 client) over net : LOGIN / SELECT / SEARCH / FETCH (with literals) / LOGOUT . connect / selectMailbox / search / fetch / logout , plus fetchAll ; messages parsed by mime . A reading subset.
influxdb	import "influxdb.j";	no (net)	an InfluxDB 1.x time-series client on http : build line-protocol points with value-semantic builders (point / tag / field / intField / stringField / boolField / at , mixed field types via pre-rendered

Module	Import with	TinyGo	Contents
			fragments), <code>line / write</code> them, and <code>query(client, influxql) -> Result</code> (a list of Series with stringified cells, prometheus-retrieval shape). Basic auth; nanosecond precision. Needs the default binary.
<code>ipnet</code>	<code>import "ipnet.j";</code>	full	IP addresses and CIDR networks, IPv4 and IPv6. <code>parseAddress / toString</code> (canonical, RFC 5952 for IPv6) / <code>version / equal</code> ; <code>parse(cidr) -> Network</code> (host bits zeroed), <code>contains(net, addr)</code> , <code>netmask / broadcast / networkString</code> . Addresses held as raw bytes (4 / 16); bitwise subnet math for allow-lists. Pure <code>.j</code> over strings + <code>convert</code> ; both binaries.
<code>jsonl</code>	<code>import "jsonl.j";</code>	full	JSON Lines (JSONL / NDJSON): newline-delimited JSON, one <code>json.Value</code> per line. <code>encode / decode</code> (compact JSON split / joined on <code>\n</code> , blank lines skipped, so <code>decode(encode(rows))</code> round-trips), whole-file <code>readFile / writeFile / appendFile</code> , and a streaming <code>Reader</code> (<code>openReader / hasMore / readRecord / closeReader</code>) for files too large to hold in memory. A thin framing layer over <code>json + fs</code> ; both binaries.
<code>label</code>	<code>import "label.j";</code>	partial	industrial label printing in a build / render / emit pipeline. Build a device-independent <code>Label</code> in millimetres (<code>new / text / barcode / box / image / quantity</code> ; barcodes <code>code128 / ean13 / itf / code39 / gs1-128 / datamatrix / qr</code>), <code>render(label, device)</code> to a selectable dialect (<code>"zpl"</code> Zebra, <code>"cab"</code> cab JScript), then emit -

Module	Import with	TinyGo	Contents
			send(host, port, rendered) to a printer's raw :9100 port. Build / render run on both binaries; send needs the default binary.
log	import "log.j";	partial	leveled, structured logging: a <code>Logger</code> carries a minimum level (<code>debug < info < warn < error</code>), a format (<code>text / logfmt / json</code>), and a sink; <code>debug / info / warn / error</code> (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. Sinks <code>new (stdout) / toStderr / toFile / toSyslog</code> (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 <code>htmlwriter</code>) and styled terminal text (through <code>ansi</code>) with <code>toHtml / toAnsi</code>, plus authoring helpers (<code>header / style / link / bullets / numbered / codeBlock / table</code>) that build Markdown, and <code>tablePretty</code> to align table source.
memcache	import "memcache.j";	no (net)	a memcached client (classic text protocol) over <code>net: set / add / get / delete / incr / decr / touch</code>, 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 <code>net</code> (8728 / api-ssl 8729): <code>connect</code> (plaintext login, MD5 fallback), <code>talk(s, command, attrs) -> list</code> of map of string to string (each !re row), <code>print read sugar, run</code>

Module	Import with	TinyGo	Contents
			for add / set / remove (returns the <code>!done =ret=</code>). Sentence-based binary framing (variable-length word codec) hand-built with the bitwise ops; <code>!trap</code> throws. Needs the default binary.
<code>mime</code>	<code>import "mime.j";</code>	full	build and parse MIME messages (RFC 5322 headers, multipart, quoted-printable / base64 transfer encodings, RFC 2047 encoded-words for non-ASCII headers). <code>text / attachment / multipart / encode / parse</code> ; the foundation the mail clients build on.
<code>mqtt</code>	<code>import "mqtt.j";</code>	no (net)	an MQTT 3.1.1 pub/sub client over <code>net</code> (<code>mqttps</code> via TLS): <code>connect -> Client</code> , then <code>subscribe / publish / publishBytes</code> (QoS 0), blocking <code>receive</code> and single-threaded <code>poll(client, timeoutMs)</code> (via <code>net.setDeadline</code>), <code>ping</code> , <code>disconnect</code> . Binary packet framing built with bitwise ops + <code>bytes</code> . Basics-first (QoS 0).
<code>multipart</code>	<code>import "multipart.j";</code>	full	build and parse <code>multipart/form-data</code> (RFC 7578) - the file-upload counterpart to <code>mime</code> . <code>field / file build Parts, build / buildWith -> Built{contentType, body}</code> (fresh or fixed boundary), <code>parse(contentType, body) -> list of Part</code> ; <code>text / isFile</code> read a part. Byte-level delimiter matching so binary bodies round-trip. Pairs with <code>web / http</code> . Pure <code>.j</code> ; both binaries.
<code>ntp</code>	<code>import "ntp.j";</code>	no (net)	a simple SNTP network-time client (RFC 4330 / 5905) over UDP: <code>query(host) / queryWith(address, timeoutMs) -> Result (serverTime + clock offset + round-trip delay)</code> . Packs

Module	Import with	TinyGo	Contents
			/ unpacks the 48-byte NTP packet with <code>bytes</code> + bitwise ops and converts the NTP epoch through <code>time</code> ; a lost reply times out (not hangs). Query-only (no clock discipline / daemon). Needs the default binary.
<code>oauth</code>	<code>import "oauth.j";</code>	no (net)	a generic OAuth2 client (the get-a-token half) on <code>http</code> + <code>json</code> : Client Credentials, Refresh Token, and Device Authorization grants, <code>google</code> / <code>microsoft</code> presets, expiry + on-disk token store. Tokens feed <code>sasl</code> XOAUTH2 for mail.
<code>password</code>	<code>import "password.j";</code>	full	generate / validate / score passwords against a policy <code>Schema</code> (classes, length range, per-class minimums, symbol set, exclude-ambiguous). <code>schema</code> + <code>with*</code> builders, <code>generate -> string</code> , <code>validate -> Report{valid, reasons}</code> , <code>complexity -> Strength{length, classes, poolSize, entropy, label}</code> (bits = length * log2(pool)). Non-crypto RNG (like <code>uuid</code> ; swaps to <code>crypto</code> later); pure <code>.j</code> , both binaries.
<code>pdfwriter</code>	<code>import "pdfwriter.j";</code>	full	generate simple PDF documents (text / lines / rectangles) the way <code>htmlwriter</code> / <code>label</code> generate their formats: <code>document</code> / <code>page</code> / <code>text</code> / <code>line</code> / <code>rect</code> / <code>color</code> / <code>addPage</code> builders, <code>info</code> metadata (+ <code>pdfDate</code>), <code>render()</code> -> <code>bytes</code> writing the PDF object / xref structure by hand with <code>FlateDecode</code> content streams (via <code>compress</code>). Standard-14 fonts; points, 0-255 RGB. Byte-identical (no timestamps), <code>qpdf</code> -clean output - golden-test friendly; both binaries.

Module	Import with	TinyGo	Contents
			A writer, not a reader (no embedded fonts / images yet).
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 / observe / render) builds a metric set and renders the Prometheus text format - 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.
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 .

Module	Import with	TinyGo	Contents
<code>rest</code>	<code>import "rest.j";</code>	no (net)	an ergonomic REST layer over <code>http + json</code> : a value-semantic <code>Client</code> (base URL + headers) and <code>get / post / put / patch / delete</code> (+ <code>getJSON / postJSON / ...</code>). Base-URL joining, query strings, Bearer / Basic auth.
<code>ringbuffer</code>	<code>import "ringbuffer.j";</code>	full	a fixed-capacity ring buffer (bounded FIFO of strings, overwrite-oldest when full): <code>new(capacity) , push / pop , first / last peek , size / capacity / isEmpty / isFull / toList</code> . A sliding window of recent items. Value-semantic. Over <code>lists</code> ; both binaries.
<code>sasl</code>	<code>import "sasl.j";</code>	full	SASL auth encoders shared by the mail clients: <code>plain / loginUser / loginPass / bearer</code> (XOAUTH2, the “use a token” half of OAuth2). Pure base64, no networking.
<code>semver</code>	<code>import "semver.j";</code>	full	strict Semantic Versioning 2.0.0, package-registry-grade. <code>parse / isValid / toString , compare / lt / lte / eq / neq / gt / gte / diff , isStable / isPrerelease , inc* , sort / rsort ; coerce / clean</code> for loose tags; and npm/Composer range matching - <code>satisfies</code> (caret / tilde / comparators / / hyphen / x-ranges, prerelease-aware), <code>maxSatisfying / minSatisfying / minVersion / validRange</code> , plus solver algebra <code>intersects / subset / gtr / ltr / outside / simplifyRange</code> (prerelease-precise). Struct <code>Version</code> .
<code>session</code>	<code>import "session.j";</code>	no (net)	server-side sessions on <code>memcache</code> : a map of string to string under <code>sess:ID</code> with a sliding TTL. <code>create / load / save / touch / destroy</code> ;

Module	Import with	TinyGo	Contents
			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 <code>http: send(webhookUrl, text)</code> for a plain message, or build a Block Kit message with <code>message / text / section / header / divider</code> and post it with <code>sendMessage</code> . <code>render</code> gives the JSON payload; strings JSON-escaped. Sibling of <code>discord / gotify</code> . Needs the default binary.
smtp	import "smtp.j";	no (net)	send mail (SMTP client) over <code>net: plaintext / STARTTLS / implicit TLS, AUTH PLAIN, MAIL FROM / RCPT TO / DATA</code> . <code>smtp.send(opts, from, recipients, message)</code> ; message built by <code>mime</code> .
statsd	import "statsd.j";	no (net)	a fire-and-forget StatsD metrics client over UDP: <code>client / clientWith -> Client</code> (agent address + optional name prefix), then <code>count / increment / decrement</code> (counter <code>c</code>), <code>gauge (g)</code> , <code>timing (ms)</code> , <code>set (s)</code> each emit one <code>metric:value type</code> datagram. The push counterpart to a pull-based scrape; no reply, no error when no agent listens. No sample rates / tags yet. Needs the default binary.
telegram	import "telegram.j";	no (net)	a Telegram Bot API client on <code>http + json: bot / botWith -> Bot</code> , <code>getMe</code> , <code>sendMessage / sendMessageWith</code> (parse mode) / <code>sendPhoto / sendChatAction -> Message / bool</code> , and <code>getUpdates(bot, offset, timeout)</code> long-poll -> <code>list of Update</code> for a stateful receive loop. Form-encoded params,

Module	Import with	TinyGo	Contents
			<code>{"ok":false}</code> throws. Needs the default binary.
<code>tengine</code>	<code>import "tengine.j";</code>	full	a lightweight-CMS text template engine (a subset of Go <code>text/template</code>) over a <code>json.Value</code> tree: <code>newSet</code> / <code>add</code> / <code>render</code> . <code>.path</code> / <code>\$ root</code> / <code>\$var</code> , <code>if</code> / <code>else if</code> with <code>eq</code> / <code>and</code> / <code>or</code> / <code>not</code> , <code>range</code> (with <code>\$i</code> , <code>\$e</code>) / <code>with</code> / <code>block</code> , <code>{{ \$x := }}</code> variables, <code>define</code> / <code>template</code> layout inheritance, <code>{{- -}}</code> trim markers, and pipes <code>upper</code> / <code>lower</code> / <code>title</code> / <code>trim</code> / <code>html</code> / <code>urlize</code> / <code>default</code> / <code>truncate</code> / <code>join</code> / <code>len</code> / <code>printf</code> .
<code>totp</code>	<code>import "totp.j";</code>	full	time-based one-time passwords (RFC 6238 / 4226): <code>generate</code> / <code>verify</code> (+/-1-step skew) and <code>generateAt</code> / <code>verifyAt</code> (explicit time), plus <code>uri</code> for the <code>otpauth://</code> provisioning string. base32 secrets; SHA-1 / SHA-256 / SHA-512. Over <code>hash.hmac</code> + <code>encoding</code> + <code>time</code> .
<code>vcard</code>	<code>import "vcard.j";</code>	full	vCard (RFC 6350, vCard 4.0) contacts build and parse: a <code>Card</code> of contact fields encoded to a <code>VCARD</code> and parsed back. <code>card</code> / <code>withName</code> / <code>withOrg</code> / <code>addEmail</code> / <code>addPhone</code> / <code>address</code> / <code>addAddress</code> / <code>withUrl</code> / <code>withNote</code> value-semantic builders, <code>encode</code> / <code>encodeAll</code> / <code>parse</code> (one or many cards). Structured <code>N</code> / <code>ADR</code> / <code>ORG</code> , RFC 6350 text escaping and 75-char line folding - shares the content-line codec with <code>ical</code> . Pure text over <code>strings</code> / <code>lists</code> ; both binaries. A contact subset (no <code>BDAY</code> / <code>PHOTO</code> / parameter round-trip).
<code>web</code>	<code>import "web.j";</code>	no (net)	a small HTTP framework over the <code>httpd</code> engine: register routes against handler methods by name (<code>web.get</code> / <code>post</code> / ...), <code>:param</code>

Module	Import with	TinyGo	Contents
			capture, middleware, <code>web.Context</code> request / response helpers; <code>web.run</code> owns the accept loop. Dispatch by <code>meta.callMain</code> . Pairs with <code>jennifer serve</code> .
<code>webhook</code>	<code>import "webhook.j";</code>	full (<code>send</code> <code>net</code>)	HMAC-signed webhooks (GitHub X-Hub-Signature-256): <code>sign(payload, secret)</code> / <code>verify(payload, signature, secret)</code> are pure (both binaries); <code>send(url, payload, secret)</code> POSTs the signed body via <code>http</code> (default binary). Over <code>hash.hmac</code> + <code>encoding</code> (hex).
<code>websocket</code>	<code>import "websocket.j";</code>	no (<code>net</code>)	an RFC 6455 WebSocket client over <code>net</code> (<code>ws://</code> / <code>wss://</code>): <code>connect</code> / <code>connectWith</code> do the HTTP Upgrade handshake (verifying the SHA-1 + base64 <code>Sec-WebSocket-Accept</code>), then <code>send</code> / <code>sendBytes</code> (masked frames) and <code>receive</code> -> <code>Message</code> (auto-pong, fragment reassembly), <code>ping</code> / <code>close</code> . Binary framing + masking with the bitwise ops over <code>hash</code> + <code>encoding</code> + <code>math</code> . 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.

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`.

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.

```
def struct amqp.Options { host as string, port as int, user as string, password as string, vhost as string };
```

Call	Returns	
<code>amqp.options(host, user, password)</code>	<code>Options</code>	defaults: port 5672, vhost "/"
<code>amqp.withPort(o, port)</code>	<code>Options</code>	copy with a different port
<code>amqp.withVhost(o, vhost)</code>	<code>Options</code>	copy with a different virtual host
<code>amqp.connect(opts)</code>	<code>Conn</code>	connect and open a channel
<code>amqp.close(c)</code>		<code>Connection.Close</code> and shut the socket

Queues and publishing

Call	Returns	
<code>amqp.declareQueue(c, name, durable)</code>	<code>QueueInfo</code>	declare a queue ("" name = server-generated); durable survives a restart
<code>amqp.publish(c, exchange, routingKey, body)</code>		publish a bytes body
<code>amqp.publishText(c, exchange, routingKey, text)</code>		publish a UTF-8 string

`declareQueue` returns `QueueInfo{name, messageCount, consumerCount}`. `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`.

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
<code>amqp.get(c, queue, autoAck)</code>	<code>Message</code> pull the next message (<code>empty</code> true when none)
<code>amqp.ack(c, deliveryTag)</code>	acknowledge a delivered message

Scope

- **Pull, not push.** Receiving is `Basic.Get` (one message per call); streaming `Basic.Consume` with server-pushed `Basic.Deliver` (an async loop) is a follow-on.
- **One channel, no publisher confirms / transactions.** A single channel (1) is opened; `publish` is fire-and-forget (no `Confirm.Select`).
- **No message properties.** Publishes carry an empty property set (no content-type, headers, or persistence flag on the message itself - queue durability is set at `declareQueue`).
- **SASL PLAIN only**, no TLS (`amqps`) in this version - 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` / `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](#).

Surface

Call	Returns	Notes
<code>ansi.color(s, name)</code>	<code>string</code>	Wrap <code>s</code> in the named foreground colour. Unknown name <code>throw S</code> .
<code>ansi.bgColor(s, name)</code>	<code>string</code>	Wrap <code>s</code> in the named background colour.
<code>ansi.style(s, name)</code>	<code>string</code>	Wrap <code>s</code> in a text style: <code>bold</code> / <code>dim</code> / <code>italic</code> / <code>underline</code> / <code>reverse</code> .
<code>ansi.rgb(s, r, g, b)</code>	<code>string</code>	24-bit truecolor foreground; each channel <code>0 - 255</code> .
<code>ansi.strip(s)</code>	<code>string</code>	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.

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`.

Encoding

`barcode.encode(data, symbology, opts) -> Symbol`. Symbologies:

Symbology	Kind	Notes
<code>qr</code>	2D	Reed-Solomon over GF(256), EC levels L/M/Q/H (<code>opts.ecLevel</code>), automatic version selection 1-10, data-mask scoring, byte mode (any UTF-8)
<code>code128</code>	1D	Code set B (ASCII 32-126), auto checksum
<code>code39</code>	1D	uppercase + digits + <code>- . \$ / + % , * </code> start/stop
<code>ean13</code>	1D	12 or 13 digits (check digit computed if omitted)
<code>ean8</code>	1D	7 or 8 digits
<code>itf</code>	1D	Interleaved 2 of 5, even digit count

```
def struct barcode.Symbol {
    kind as string,           # "matrix" (2D) or "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
};
```

`barcode.defaults()` gives scale 4, quiet 4, EC level M, black on white.

Call	Returns	
<code>barcode.svg(symbol, opts)</code>	<code>string</code>	resolution-independent SVG (embeds in HTML / email)
<code>barcode.png(symbol, opts)</code>	<code>bytes</code>	a monochrome (grayscale) PNG, hand-encoded over <code>compress + crc</code>
<code>barcode.terminal(symbol)</code>	<code>string</code>	Unicode half-block art for the CLI / REPL (2D only)
<code>barcode.matrix(symbol)</code>	<code>list of list of bool</code>	the raw 2D cells (e.g. to feed <code>label.image</code>)

`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, byte-mode codewords, and 1D bar patterns; 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 1-10** (up to ~270 bytes at level L). Byte mode only (universal); numeric / alphanumeric compaction and versions 11-40 are follow-ons.
- **The GF(256) / Reed-Solomon math** lives in a private `barcode_ecc.j` (`include d`), isolated so it can be extracted into an `ecc` module if a second consumer (e.g. Data Matrix) ever appears.
- **No general image library** - the only raster need is a monochrome bitmap, which the PNG encoder covers directly.
- **Data Matrix / Aztec / PDF417** are not included (a Data Matrix would reuse `barcode_ecc.j` inside this module).

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` + `bytes`; 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`.

Surface

```
def struct bloom.Filter { bits as bytes, size as int, hashes as int };
```

Call	Returns	
<code>bloom.new(size, hashes)</code>	<code>Filter</code>	an empty filter of <code>size</code> bits and <code>hashes</code> hash functions (both ≥ 1)
<code>bloom.add(f, item)</code>	<code>Filter</code>	a fresh filter with <code>item</code> recorded
<code>bloom.addAll(f, items)</code>	<code>Filter</code>	a fresh filter with every item of a <code>list of string</code> recorded
<code>bloom.mightContain(f, item)</code>	<code>bool</code>	<code>true</code> if the item might be present, <code>false</code> if it is definitely absent

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` $\approx -n * \ln(p) / (\ln 2)^2$ (about `9.6 * n` bits for `p = 1%`, `14.4 * n` for `p = 0.1%`).
- **hashes** `k` $\approx (m / n) * \ln 2$ (about 7 for `p = 1%`).

So for 10000 items at 1%: `size` $\approx$ 96000, `hashes` = 7.

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.

bucket - S3-compatible object storage

Import with `import "bucket.j" as bucket;`. 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`).

Named `bucket` (not `s3`) because a module namespace is letters-only - the same reason `pop` is not `pop3`.

```
import "bucket.j" as bucket;
import "http.j" as http;

def c as bucket.Client init bucket.connect(
    "https://s3.us-east-1.amazonaws.com", "us-east-1", accessKey, secretKey);

def put as http.Response init bucket.put($c, "mybucket", "hello.txt", "hi there");
def obj as http.Response init bucket.get($c, "mybucket", "hello.txt");
io.printf("%d\n%s\n", $obj.status, $obj.body);
```

Runnable: `examples/modules/bucket_demo.j`.

Client

`bucket.connect(endpoint, region, accessKey, secretKey)` returns a value-semantic `bucket.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 bucket.Client init bucket.connect(endpoint, region, key, secret);
$c.timeout = 5000; # fail a request that stalls for 5 s
```


Store	endpoint	region
AWS S3	<code>https://s3.<region>.amazonaws.com</code>	your bucket's region
MinIO	<code>http://host:9000</code>	us-east-1 (or as configured)
Cloudflare R2	<code>https://<account>.r2.cloudflarestorage.com</code>	auto
Backblaze B2	<code>https://s3.<region>.backblazeb2.com</code>	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
<code>bucket.get(client, bucket, key)</code>	GET	<code>body</code> is the object contents; a missing object is a 404.
<code>bucket.put(client, bucket, key, body)</code>	PUT	Upload / overwrite; 200 on success.
<code>bucket.delete(client, bucket, key)</code>	DELETE	204 on success.
<code>bucket.listObjects(client, bucket)</code>	GET ?list-type=2	<code>body</code> is the ListObjectsV2 XML.
<code>bucket.objectKeys(xml)</code>	-	Pull the <code><Key></code> values out of a <code>listObjects</code> body -> list of string.

(The list op is `listObjects`, not `list`, because `list` is a reserved type keyword.)

```
def r as http.Response init bucket.listObjects($c, "mybucket");
for (def k in bucket.objectKeys($r.body)) {
  io.printf("%s\n", $k);
}
```

Signing

Requests are signed with SigV4 for service `s3`: the canonical request covers the method, the URI-encoded path (object keys keep their `/`), the query, the signed headers `host` / `x-amz-content-sha256` / `x-amz-date`, 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 payload hash is a real SHA-256 of the body (not `UNSIGNED-PAYLOAD`), so the whole request is integrity-covered. The signature is pinned in the tests against two independent SigV4 implementations.

Scope

- **Path-style, SigV4, `s3` service.** Virtual-hosted addressing and the older SigV2 are out of scope.
- **String bodies.** Objects are sent / received as text (`http` 's current body type); a `bytes` body accessor is a planned follow-on, alongside a binary object path.
- **Core object ops.** Multipart upload, presigned URLs, bucket create / delete, and ACL / policy management are not covered.
- **`listObjects` returns the raw XML** (plus `objectKeys`); pagination (continuation tokens) and full metadata parsing are follow-ons.

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.

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`.

Functions

Call	Returns	Notes
<code>cron.parse(expr)</code>	<code>Schedule</code>	Parse a five-field expression.
<code>cron.matches(schedule, t)</code>	<code>bool</code>	Does the schedule fire at <code>t</code> ? (minute granularity - seconds are ignored).
<code>cron.next(schedule, after)</code>	<code>time.Time</code>	The next fire at or after <code>after</code> , keeping its zone offset.

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	<code>0</code> and <code>7</code> 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
<code>* * * * *</code>	every minute
<code>*/15 * * * *</code>	every 15 minutes
<code>0 9 * * 1-5</code>	09:00 on weekdays (Mon-Fri)
<code>0 0 1 * *</code>	midnight on the 1st of each month
<code>30 3 * * 0</code>	03:30 on Sundays
<code>0 0 13 * 5</code>	midnight on Friday the 13th (see below)

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.** No seconds field, no `@daily` / `@reboot` macros, 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](#).

Surface

Call	Returns	Notes
<code>csv.parse(s)</code>	list of list of string	Parse standard comma-delimited CSV into rows of fields.
<code>csv.parseWith(s, delim)</code>	list of list of string	Same, with a single-character delimiter (<code>"\t"</code> for TSV).
<code>csv.format(rows)</code>	string	Encode rows as comma-delimited CSV; quotes fields that need it.
<code>csv.formatWith(rows, delim)</code>	string	Same, with a single-character delimiter.
<code>csv.toRecords(rows)</code>	list of map of string to string	Treat row 0 as a header; map each later row to a header-keyed record.
<code>csv.fromRecords(header, records)</code>	list of list of string	Inverse: a header row followed by one row per record, in header order.

Parsing (RFC 4180)

`parse` and `parseWith` implement the [RFC 4180](#) 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.

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`. Streaming a file too large to hold in memory is also out of scope: `parse` takes a whole string. Read the file with `fs.readString` (or slurp stdin) and hand the text in.

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`, 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`.

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 string # pre-rendered embed JSON fragments
};
```

Call		Returns
<code>discord.message()</code>	<code>Message</code>	start an empty message

Call	Returns	
<code>discord.content(m, content)</code>	Message	set the top-level content
<code>discord.embed(m, title, description, color)</code>	Message	append an embed
<code>discord.render(m)</code>	string	render the JSON payload
<code>discord.sendMessage(webhookUrl, m)</code>	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.

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, and color. Embed fields, author, footer, thumbnail, and image 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`, 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 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) { ... }
```

The body is a **summary** line, an optional **description** (everything up to the first tag), then `@`-tags.

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, 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 }`
- `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 }`
- `Diagnostics { 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 `Diagnostics` 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 **neests `/* */` 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 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.
- `htmlwriter` - render parsed docs to HTML (a separate consumer, not part of `docblock`).

dotenv - .env configuration files

Import with `import "dotenv.j" as dotenv;`. Read the `KEY=VALUE` files that keep configuration and secrets out of source. `parse` turns text into a `map of string to string`, `read` parses a file, and `load` parses a file **and** sets each variable in the process environment. Over `fs + strings + os`; pure `.j`, runs on **both** binaries.

```
import "dotenv.j" as dotenv;

def cfg as map of string to string init dotenv.parse("PORT=8080\n#
note\nNAME=\"ada\"");
io.printf("%s\n", $cfg["PORT"]);      # 8080

dotenv.load(".env");                  # or set them in the environment
```

Runnable: `examples/modules/dotenv_demo.j`.

Functions

Call	Returns	Notes
<code>dotenv.parse(text)</code>	<code>map of string to string</code>	Parse <code>.env</code> text; touches nothing else.
<code>dotenv.read(path)</code>	<code>map of string to string</code>	Read and parse a file.
<code>dotenv.load(path)</code>	<code>map of string to string</code>	Read, parse, and <code>os.setEnv</code> each variable; returns the map.

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
PATH="/a\t/b"                  # double quotes expand \n \t \r
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); an unknown escape keeps its character.
- **Single quotes** are 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.

Scope

- **No variable interpolation.** `${OTHER}` inside a value is not expanded (the value is taken literally); reference other variables in your program instead.
- **No multi-line values.** Each assignment is one line.
- **load overwrites.** It sets every parsed variable with `os.setEnv`, replacing any existing value. To make `.env` a *default* (don't override a real environment variable), read the map and set only the keys `os.getEnv` reports as empty.

See also

- [fs.md](#) - the file read `read` / `load` use.
- [os.md](#) - `os.setEnv` / `os.getEnv` for the environment.
- [toml.md](#) - for richer, typed, nested configuration.
- [modules/index.md](#) - the module catalog and import rules.

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	
<code>flatdb.open(path)</code>	<code>DB</code>	Load the file (an empty store if it's absent, so first run never fails).
<code>flatdb.get(db, pointer)</code>	<code>json.Value</code>	The sub-document at a JSON Pointer ("" = the whole document).
<code>flatdb.has(db, pointer)</code>	<code>bool</code>	Whether the pointer resolves.
<code>flatdb.keys(db, pointer)</code>	list of string	Keys of the object at the pointer, in document order.
<code>flatdb.length(db, pointer)</code>	<code>int</code>	Element / entry count at the pointer.
<code>flatdb.set(db, pointer, value)</code>	<code>DB</code>	Upsert an object key / replace a list index (strict: no auto-vivify).
<code>flatdb.append(db, pointer, value)</code>	<code>DB</code>	Push onto the list at the pointer (create it first with <code>set</code>).
<code>flatdb.remove(db, pointer)</code>	<code>DB</code>	Drop the key / element at the pointer.
<code>flatdb.save(db)</code>	<code>null</code>	Write the document back, atomically (temp + rename).

`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](#).

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.

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](#) 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](#).

Surface

Call / type	Notes
<code>gotify.Config</code>	<code>url</code> (server, no trailing slash) and <code>token</code> (application key).
<code>gotify.push(cfg, title, message, priority)</code>	POST the message; returns the <code>http.Response</code> .

`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](#) contract, where `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.

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, "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	
<code>gpio.setup(pin, direction)</code>	<code>null</code>	Export <code>pin</code> and set its direction: <code>"in"</code> or <code>"out"</code> .
<code>gpio.write(pin, value)</code>	<code>null</code>	Set an output pin's value: <code>0</code> or <code>1</code> .
<code>gpio.read(pin)</code>	<code>int</code>	Read a pin's current value (<code>0</code> / <code>1</code>).
<code>gpio.release(pin)</code>	<code>null</code>	Unexport <code>pin</code> .

A bad direction (not `"in"` / `"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` drives the module against a temp-dir mock, and how `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.

See also

- `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.

htmlwriter - build and render an HTML tree

Import with `import "htmlwriter.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 "htmlwriter.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 & bye</p>
```

Runnable: `examples/modules/htmlwriter_demo.j`.

The node model

An HTML tree is `Node` values. A node is one of three kinds, tagged by its `kind` field, and is built with a constructor rather than a literal:

Kind	Built with	Renders as
"element"	<code>element(tag, attrs, children)</code>	<code><tag ...>children</tag></code> (or a void tag)
"text"	<code>text(s)</code>	<code>s</code> , HTML-escaped
"raw"	<code>raw(s)</code>	<code>s</code> , verbatim (already-trusted markup only)

```
export def struct Node {
  kind as string, 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 };
```

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`.

Surface

Call	Returns	Notes
<code>html.element(tag, attrs, children)</code>	<code>Node</code>	An element node. Pass <code>[]</code> for no attributes or no children.
<code>html.text(s)</code>	<code>Node</code>	A text node; <code>s</code> is HTML-escaped on render.
<code>html.raw(s)</code>	<code>Node</code>	A verbatim node; <code>s</code> is not escaped. Trusted markup only.
<code>html.attr(name, value)</code>	<code>Attr</code>	One attribute; <code>value</code> is escaped in attribute context on render.
<code>html.render(node)</code>	<code>string</code>	Serialize a node and its subtree to HTML5.
<code>html.renderAll(nodes)</code>	<code>string</code>	Serialize a list of <code>Node</code> fragment in order.
<code>html.escape(s)</code>	<code>string</code>	HTML-escape a bare string for text context (the helper <code>render</code> uses).

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.

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, []))); # 
```

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));
```

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](#).

Surface

`headers` is a map of string to string (pass `{}` for none); a `body` is a string ("" for none).

Call / type	Notes
<code>http.Response</code>	<code>status</code> (int), <code>statusText</code> , <code>headers</code> (lowercased keys), <code>body</code> .
<code>http.request(method, url, headers, body)</code>	The general request (default idle timeout); returns a <code>Response</code> .
<code>http.requestWith(method, url, headers, body, timeoutMs)</code>	As <code>request</code> , with an explicit per-read idle timeout (<code>0</code> = none).
<code>http.get(url, headers)</code>	GET.
<code>http.post(url, contentType, body, headers)</code>	POST; sets <code>Content-Type</code> .

Call / type	Notes
<code>http.put(url, contentType, body, headers)</code>	PUT; sets <code>Content-Type</code> .
<code>http.patch(url, contentType, body, headers)</code>	PATCH (partial update); sets <code>Content-Type</code> .
<code>http.delete(url, headers)</code>	DELETE.
<code>http.head(url, headers)</code>	HEAD (status + headers, no body).
<code>http.options(url, headers)</code>	OPTIONS (capability probe; read the <code>Allow</code> header).
<code>http.header(resp, name)</code>	Read a response header case-insensitively, or <code>""</code> if absent.

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.

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.

The body is returned as **text** (UTF-8). A JSON / HTML / XML body round-trips exactly (the whole body is decoded as one unit, so it is byte-exact); a binary body (an image, a gzip stream) is not decodable to a string and raises an error

- a `bytes` body accessor is a planned follow-on.

Timeouts

Every request 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.

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 are returned, not followed.** A 3xx comes back with its `Location` header; follow it yourself (auto-follow with a hop limit is a later add).

- **One request per connection.** No keep-alive, no connection pool.
- **No cookie jar, no automatic decompression, no multipart builder.** Set the headers and body you need directly.
- **Text bodies.** Binary responses need the planned `bytes` accessor.

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](#).

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 };
def struct ical.Event {
  uid as string,
  stamp as time.Time,      # DTSTAMP
  start as time.Time,      # DTSTART
  end as time.Time,        # DTEND
  summary as string,        # SUMMARY
  description as string,    # DESCRIPTION (" " when unset)
  location as string        # LOCATION (" " when unset)
};
```


Building

Call	Returns	
<code>ical.calendar()</code>	<code>Calendar</code>	an empty calendar with the default <code>PRODID</code>
<code>ical.calendarWith(prodid)</code>	<code>Calendar</code>	an empty calendar with a custom <code>PRODID</code>
<code>ical.event(uid, start, end, summary)</code>	<code>Event</code>	an event; <code>DTSTAMP</code> defaults to <code>start</code>
<code>ical.describe(ev, description)</code>	<code>Event</code>	a copy with the description set
<code>ical.locate(ev, location)</code>	<code>Event</code>	a copy with the location set
<code>ical.add(cal, ev)</code>	<code>Calendar</code>	a copy with the event 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	
<code>ical.encode(cal)</code>	<code>string</code>	the calendar as RFC 5545 text (CRLF-terminated)
<code>ical.parse(text)</code>	<code>Calendar</code>	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 non-empty. `parse` unfolds folded lines, ignores property parameters (the `;KEY=VALUE` after a name, e.g. `DTSTART;VALUE=DATE-TIME`), 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.

Scope

- **VEVENT only.** No `VTOD` / `VJOURNAL` / `VALARM` / `VTIMEZONE`, no recurrence rules (`RRULE`), no attendees / organizer. A focused calendar-of- events surface; the escaping / folding / date discipline is the reusable core.
- **UTC date-times.** Events are stored and emitted in UTC. There is no per-event `TZID` timezone reference (the `time` library ships fixed-offset zones only); a `TZID` parameter on input is ignored and the value read as-is.
- **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](#).

Surface

Call	Returns	Notes
<code>idna.toAscii(domain)</code>	<code>string</code>	Domain to its ASCII form; a Unicode label becomes <code>xn--...</code> , an ASCII label is lowercased.
<code>idna.toUnicode(domain)</code>	<code>string</code>	The inverse; an <code>xn--</code> label is decoded, others pass through.
<code>idna.isAscii(domain)</code>	<code>bool</code>	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 `LOGIN` auth. A useful **reading subset** - select a mailbox, search it, fetch whole messages - not the full protocol. 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](#).

Surface

A session is stateful: `connect`, `selectMailbox`, `search` / `fetch`, `logout`. `fetchAll` wraps the common “read every message in a mailbox” case.

Call / type	Notes
<code>imap.Options</code>	<code>host</code> , <code>port</code> , <code>security</code> , <code>user</code> , <code>pass</code> .
<code>imap.Session</code>	A live session over one connection (from <code>connect</code>).
<code>imap.connect(opts)</code>	Open a session: greeting, optional STARTTLS, <code>LOGIN</code> .
<code>imap.selectMailbox(session, name)</code>	<code>SELECT</code> a mailbox (e.g. <code>"INBOX"</code>); returns its message count.

Call / type	Notes
<code>imap.search(session)</code>	<code>SEARCH ALL</code> - the sequence numbers in the selected mailbox (<code>list of int</code>).
<code>imap.fetch(session, n)</code>	<code>FETCH n BODY.PEEK[]</code> - message <code>n</code> as a raw string, for <code>mime.parse</code> .
<code>imap.logout(session)</code>	<code>LOGOUT</code> and close.
<code>imap.fetchAll(opts, mailbox)</code>	Connect, select, retrieve every message, log out; <code>list of string</code> .

`Options.security` is `"none"` (143), `"tls"` (implicit TLS on connect, 993), or `"starttls"` .
`fetch` uses `BODY.PEEK[]` , so retrieving does **not** set the `\Seen` flag.

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.

Out of scope

This is a reading subset, not full IMAP4rev1:

- **Commands.** `LOGIN` / `SELECT` / `SEARCH ALL` / `FETCH BODY.PEEK[]` / `LOGOUT` . No partial fetch, `STORE` (flag changes), `COPY` , `APPEND` , `EXPUNGE` , mailbox management, or `IDLE` .
- **Auth.** `LOGIN` , or `AUTHENTICATE XOAUTH2` (`Options.auth = "xoauth2"` , via `sasl` , for Google / Microsoft 365). The SASL challenge-response mechanisms (`CRAM-MD5` / `SCRAM`) land with the `crypto` library.
- **Literals are read as 7-bit / ASCII.** MIME transfer encoding keeps mail bodies ASCII; raw 8-bit literals are not yet byte-exact.
- An internationalized (IDN) host is IDNA-encoded to its `xn--` form automatically (via `idna`).

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.

See also

- [mime.md](#) - parse a fetched message (`mime.parse`).
- [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 to InfluxDB (1.x) as line-protocol points, and run InfluxQL queries that come back as parsed `Series`. Built on the `http` module, so it needs the default `jennifer` binary. A failed request throws `Error{kind: "influxdb"}`.

```
import "influxdb.j" as influxdb;

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");
```

Runnable: `examples/modules/influxdb_demo.j`.

Client

```
def struct influxdb.Client {
    url as string,          # base URL, e.g. "http://localhost:8086"
    db as string,           # database name
    user as string,         # username ("" for no auth)
    password as string      # password
};
```

Call		Returns
<code>influxdb.client(url, db)</code>	<code>Client</code>	connect to a database, no authentication
<code>influxdb.clientWith(url, db, user, password)</code>	<code>Client</code>	with HTTP Basic-auth credentials

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	
<code>influxdb.point(measurement)</code>	<code>Point</code>	start a point (no tags/fields yet)
<code>influxdb.tag(p, key, value)</code>	<code>Point</code>	add an indexed string tag
<code>influxdb.field(p, key, value)</code>	<code>Point</code>	add a float field
<code>influxdb.intField(p, key, value)</code>	<code>Point</code>	add an int field (line-protocol <code>i</code> suffix)
<code>influxdb.stringField(p, key, value)</code>	<code>Point</code>	add a string field (quoted, escaped)
<code>influxdb.boolField(p, key, value)</code>	<code>Point</code>	add a bool field
<code>influxdb.at(p, unixNanos)</code>	<code>Point</code>	set an explicit timestamp (nanoseconds)
<code>influxdb.atTime(p, t)</code>	<code>Point</code>	set the timestamp from a <code>time.Time</code>
<code>influxdb.line(p)</code>	<code>string</code>	render one line-protocol line (throws if no fields)
<code>influxdb.write(c, points)</code>		write a list of <code>Point</code> 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. `write` posts to `/write?db=...&precision=ns` and throws on a non-2xx response, surfacing the server's `{"error": ...}` message when present.

```
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)
  }
}
```

Scope

- **InfluxDB 1.x line protocol + InfluxQL.** The 2.x Flux API and the `/api/v2/write` org/bucket model are not covered; a v2 backend would be a second selectable backend under stance 1, added on a concrete need.
- **Nanosecond write precision** (`precision=ns`); a point's timestamp is an integer nanosecond value (or a `time.Time` via `atTime`).
- **Stringified query cells.** The result keeps rows homogeneous (`list of list of string`) rather than exposing a typed cell union; you convert numeric columns yourself.
- **Basic auth only.** Token auth and TLS client certs are not wired; use a reverse proxy for those, or an unauthenticated local endpoint.

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, and subnet math (netmask, broadcast). 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`.

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	
<code>ipnet.parseAddress(s)</code>	<code>Address</code>	parse an IPv4 dotted-quad or IPv6 address
<code>ipnet.toString(addr)</code>	<code>string</code>	canonical text (RFC 5952 for IPv6)
<code>ipnet.version(addr)</code>	<code>int</code>	4 or 6
<code>ipnet.equal(a, b)</code>	<code>bool</code>	same version and bytes

`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.

CIDR networks

Call	Returns	
<code>ipnet.parse(cidr)</code>	<code>Network</code>	parse address/prefix (host bits zeroed)
<code>ipnet.networkString(net)</code>	<code>string</code>	render as address/prefix
<code>ipnet.contains(net, addr)</code>	<code>bool</code>	is the address in the network?
<code>ipnet.netmask(net)</code>	<code>Address</code>	the netmask (e.g. 255.255.255.0)
<code>ipnet.broadcast(net)</code>	<code>Address</code>	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; }
}
```

Errors

Malformed input throws `Error{kind: "ipnet"}` (a bad octet, too few / many groups, multiple `::`, a bad hex digit, a missing `/prefix`, or an out-of-range prefix) - 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.
- **Lenient decimal octets.** `parseAddress` reads IPv4 octets as plain decimal, so leading zeros are accepted as decimal (`192.168.001.001` = `192.168.1.1`), not rejected or read as octal.
- **IPv4-mapped IPv6 renders as hex.** `::ffff:192.168.1.1` round-trips at the byte level but `toString` renders it in pure hex form (`::ffff:c0a8:101`), not the dotted-quad convenience form.
- **No subnet-of-subnet test in v1.** `contains` tests an address in a network; network-in-network containment is not modelled yet.

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](#).

In-memory

Call	Returns	
<code>jsonl.encode(records)</code>	<code>string</code>	one compact JSON value per line, each newline-terminated
<code>jsonl.decode(text)</code>	list of <code>json.Value</code>	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	
<code>jsonl.readFile(path)</code>	list of <code>json.Value</code>	read and decode a whole JSONL file

Call	Returns	
<code>jsonl.writeFile(path, records)</code>	<code>null</code>	encode and write (replacing existing content)
<code>jsonl.appendFile(path, records)</code>	<code>null</code>	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	
<code>jsonl.openReader(path)</code>	<code>Reader</code>	open a file for streaming
<code>jsonl.hasMore(reader)</code>	<code>bool</code>	whether more input remains
<code>jsonl.readRecord(reader)</code>	<code>json.Value</code>	the next record (skips blank lines)
<code>jsonl.closeReader(reader)</code>	<code>null</code>	close the reader

`readRecord` mirrors `fs.readLine`: it throws `Error{kind: "jsonl"}` when the stream is exhausted, so guard it with `hasMore`.

```
def r as jsonl.Reader init jsonl.openReader("events.jsonl");
while (jsonl.hasMore($r)) {
  def rec as json.Value init jsonl.readRecord($r);
  # process one record without loading the whole file
}
jsonl.closeReader($r);
```

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.

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](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
<code>label.Label</code>	<code>width</code> , <code>height</code> (mm), <code>quantity</code> , <code>fields</code> .
<code>label.Field</code>	one placed field (<code>kind</code> "text"/"barcode"/"box"/"image").
<code>label.TextOptions</code>	<code>height</code> (mm), <code>points</code> (pt, wins over height), <code>rotation</code> (0/90/180/270), <code>bold</code> .
<code>label.new(width, height)</code>	A new empty label of that size in mm (quantity 1).

Call / type	Notes
<code>label.text(label, x, y, opts, content)</code>	Place text at (x, y); <code>opts</code> is a <code>TextOptions</code> .
<code>label.barcode(label, x, y, type, opts, data)</code>	Place a barcode (symbolologies + <code>opts</code> below).
<code>label.box(label, x, y, w, h, thickness)</code>	Place a rectangular outline (all mm).
<code>label.image(label, x, y, name)</code>	Place a pre-stored image by name (native size).
<code>label.quantity(label, n)</code>	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
<code>height</code> (float, mm)	Bar height (linear) or module size (2D); <code>0</code> uses the default (15 mm / 1 mm).
<code>checkDigit</code> (string)	Append an auto-computed check digit: <code>"mod10"</code> , <code>"mod11"</code> , <code>"mod16"</code> , <code>"mod36"</code> , <code>"mod43"</code> (<code>""</code> = none). On cab this is <code>+MODxx</code> ; on ZPL it toggles a symbology's native check (Code 39 / ITF) - Code 128 / EAN / GS1 carry the check digit in the data.
<code>errorLevel</code> (string)	2D error-correction level <code>"L"</code> / <code>"M"</code> / <code>"Q"</code> / <code>"H"</code> (<code>""</code> = default).
<code>hideText</code> (bool)	<code>true</code> suppresses a linear code's human-readable line.
<code>moduleWidth</code> (float, mm)	A linear code's narrow-element width; <code>0</code> uses the dialect default (cab writes it as <code>ne</code> ; ZPL uses its default <code>^BY</code> module width).

Field	Effect
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
<code>label.zpl(dpi)</code>	A ZPL target at the given printer resolution.
<code>label.cab()</code>	A cab target with default print-setup.
<code>label.cabWith(setup)</code>	A cab target carrying an explicit <code>label.CabSetup</code> .

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
<code>jobName</code> (string)	<code>J <name></code> (bare <code>J</code> when empty).
<code>heat</code> (int), <code>speed</code> (int), <code>mode</code> (string)	<code>H <heat>,<speed>,<mode></code> (omitted when all are zero/empty).
<code>orientation</code> (string)	<code>O <orientation></code> (e.g. "R" ; omitted when empty).
<code>sensor</code> (string)	the <code>S</code> photocell/sensor type (e.g. "l1" = die-cut with gap -> <code>S l1;...</code>).
<code>xOffset</code> , <code>yOffset</code> (float, mm)	the <code>S</code> horizontal / vertical origin offset.
<code>height</code> (float, mm)	the <code>S</code> label height (transport direction). width 0 derives the whole S line from the label size.
<code>pitch</code> (float, mm)	the <code>S</code> label pitch = label height + the gap between labels.
<code>width</code> (float, mm)	the <code>S</code> label width.
<code>columnPitch</code> (float, mm), <code>columns</code> (int)	multi-up dies: the horizontal column pitch and label count; appended only when <code>columns > 1</code> .

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
<code>label.send(host, port, rendered)</code>	Open a TCP connection and write the stream (raw <code>:9100</code>).

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.

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](#).

Loggers

A `log.Logger` is value-semantic; build one with a constructor that fixes the sink:

Constructor	Sink
<code>log.new(level, format)</code>	standard output
<code>log.toStderr(level, format)</code>	standard error
<code>logToFile(level, format, path)</code>	append to a file (created if missing)
<code>log.toSyslog(level, address, app)</code>	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	
<code>log.debug(logger, message, fields)</code>	verbose / development detail
<code>log.info(logger, message, fields)</code>	normal operation
<code>log.warn(logger, message, fields)</code>	something unexpected but handled
<code>log.error(logger, message, fields)</code>	a failure
<code>log.at(logger, level, message, fields)</code>	emit at a level chosen at runtime

A record below the logger's level is dropped before rendering, so guarded `log.debug` calls in hot paths cost only the level comparison.

Formats

The same record - timestamp, level, message, fields - in each format:

Format	Example
<code>text</code>	<code>2026-01-02T03:04:05Z INFO user logged in user=ada id=42</code>
<code>logfmt</code>	<code>time=2026-01-02T03:04:05Z level=info msg="user logged in" user=ada id=42</code>
<code>json</code>	<code>{"time":"2026-01-02T03:04:05Z","level":"info","msg":"user logged in","user":"ada","id":"42"}</code>

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 `htmlwriter` 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`.

Surface

Rendering (Markdown in, HTML / terminal text out):

Call	Returns	Notes
<code>markdown.toHtml(md)</code>	<code>string</code>	Render to HTML: block elements concatenated, no indentation.
<code>markdown.toAnsi(md)</code>	<code>string</code>	Render to terminal text with <code>ansi</code> styling (self-suppressing).

Authoring (build Markdown text - the inverse):

Call	Returns	Notes
<code>markdown.header(level, s)</code>	<code>string</code>	ATX heading; <code>level</code> is "h1" .. "h6" (throws otherwise).
<code>markdown.style(kind, s)</code>	<code>string</code>	Inline emphasis; <code>kind</code> is "bold" / "italic" / "code".
<code>markdown.link(text, url)</code>	<code>string</code>	<code>[text](url)</code> .
<code>markdown.bullets(items)</code>	<code>string</code>	Unordered list, one - item per line.

Call	Returns	Notes
<code>markdown.numbered(items)</code>	string	Ordered list, 1. item upward.
<code>markdown.codeBlock(text)</code>	string	Fenced code block around verbatim text.
<code>markdown.table(headings, aligns, rows)</code>	string	GFM table from column headings, per-column alignment, and rows.
<code>markdown.tablePretty(md)</code>	string	Reformat every table's source columns to line up; other lines untouched.

Supported Markdown

A deliberately small [CommonMark](#) 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		
Fenced code block	``` ... ```	<pre><code>	
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)		underline + (url)

HTML output

`toHtml` builds an `htmlwriter` 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&b=2">t</a> and &lt;b>& & <code>x&lt;y</code>  
</p>
```

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.

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 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-pretty 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:

- **Inline spans do not nest.** The content of `**...**`, ``...``, and a link's text is taken as plain text, so `**a b **` does not render the inner code span.
- No blockquotes, thematic breaks (`---`), images, reference links, autolinks, HTML passthrough, or setext (underlined) headings.
- No nested / indented lists; a list is a flat run of same-kind items.

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.

See also

- [htmlwriter.md](#) - the HTML backend `toHtml` renders through.
- [ansi.md](#) - the terminal styling `toAnsi` renders through.
- [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](#).

Surface

A session is stateful: `connect`, issue commands, `quit`. Every store carries an `exptime` in **seconds** (`0` = never expire, until evicted).

Call / type	Notes
<code>memcache.Options</code>	<code>host</code> , <code>port</code> (plaintext; the text protocol has no auth / TLS).
<code>memcache.Session</code>	A live session over one connection (from <code>connect</code>).
<code>memcache.connect(opts)</code>	Open a session.
<code>memcache.set(session, key, value, exptime)</code>	Store <code>value</code> (replacing any existing), TTL <code>exptime</code> seconds.

Call / type	Notes
<code>memcache.add(session, key, value, exptime)</code>	Store only if the key is absent ; returns whether it stored.
<code>memcache.get(session, key)</code>	The string value, or "" when the key is absent / expired.
<code>memcache.delete(session, key)</code>	Remove the key; returns whether it existed.
<code>memcache.incr(session, key, delta)</code>	Atomically add <code>delta</code> ; the new value, or -1 if the key is absent.
<code>memcache.decr(session, key, delta)</code>	Atomically subtract <code>delta</code> (not below 0); -1 if absent.
<code>memcache.touch(session, key, exptime)</code>	Re-arm the key's expiry; returns whether it existed.
<code>memcache.quit(session)</code>	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.

Values are read as UTF-8 text

The stored byte count is exact on the wire, but the parser reads a value back as UTF-8 text, so `get` is byte-exact for ASCII and UTF-8 values whose byte length equals their rune length - the common case (JSON, numbers, identifiers). A binary value, or one whose multi-byte runes make byte length differ from rune length, is not yet byte-exact; store such values base64-encoded (via `encoding`) until a byte-native read lands. This is the same limitation as `redis`.

Out of scope

- **A working subset**, not the full command set: `gets` / `cas`, `append` / `prepend`, `stats`, and multi-key `get` are reachable later; the basics 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"}`.

```
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`.

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
<code>mikrotik.options(host, user, password)</code>	<code>Options</code>	plain TCP, port 8728

Call	Returns	
<code>mikrotik.optionsTLS(host, user, password)</code>	Options	api-ssl (TLS), port 8729
<code>mikrotik.withPort(o, port)</code>	Options	copy with a different port
<code>mikrotik.connect(opts)</code>	Session	connect and log in
<code>mikrotik.close(s)</code>		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	
<code>mikrotik.talk(s, command, attrs)</code>	list of map of string to string	the general call - the <code>!re</code> reply rows
<code>mikrotik.print(s, path)</code>	list of map of string to string	read sugar for <code>path + "/print"</code>
<code>mikrotik.run(s, command, attrs)</code>	string	for add / set / remove - returns the <code>!done</code> <code>=ret=</code> (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);
```

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.

- **Synchronous `talk`**. Query words (`?name=value`) and `.tag` -multiplexed concurrent commands are follow-ons - each call runs to its `!done` before the next.
- **`!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`.

The Part model

A message is a `Part` tree. A part is either a **leaf** (headers plus a decoded-text `body` with a transfer `encoding`) 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
};
export def struct Header { name as string, value as string };
```

Bodies are held **decoded, as text**: `encode` applies the transfer encoding and `parse` removes it, so you always read plain content.

Surface

Call	Returns	Notes
<code>mime.text(contentType, body)</code>	<code>Part</code>	Leaf text part; 7bit if ASCII, else quoted-printable. Adds

Call	Returns	Notes
		charset=utf-8 .
<code>mime.attachment(filename, contentType, body)</code>	Part	Base64 leaf with a Content-Disposition filename.
<code>mime.multipart(subtype, boundary, parts)</code>	Part	Container (multipart/subtype) over one boundary.
<code>mime.withHeader(part, name, value)</code>	Part	Copy with a header set (case-insensitive replace, else append).
<code>mime.encode(part)</code>	string	Serialize to a CRLF MIME message with transfer encodings applied.
<code>mime.parse(text)</code>	Part	Parse a message: unfold headers, split multipart, transfer-decode.
<code>mime.headerValue(part, name)</code>	string	Header value (case-insensitive) or "" .
<code>mime.body(part)</code>	string	A leaf's decoded text body.
<code>mime.parts(part)</code>	list of Part	A container's child parts.
<code>mime.contentType(part)</code>	string	Media type without parameters (e.g. text/plain).
<code>mime.address(name, email)</code>	string	RFC 5322 mailbox: email , or Name <email> (name quoted, or RFC 2047-encoded when non-ASCII).
<code>mime.encodeWord(text)</code>	string	RFC 2047 UTF-8 base64 encoded-word(s), =?UTF-8?B?...?= , folded when long.
<code>mime.decodeWord(value)</code>	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));  
}
```

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?QmVyaWNodCBhdXMgTc08bmNoZW4=?=` 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?  
R3LDvM0fZSBhdXMgTc08bmNoZW4=?=
```

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:

- **Binary bodies.** A `Part` body is text (UTF-8); an `attachment` takes text content. True binary attachments (a `bytes` body, e.g. an image) are not yet supported - that needs a `bytes`-typed body field.
- **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`. 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: ""});
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);
```

Runnable: [examples/modules/mqtt_demo.j](#).

Surface

A client is stateful: `connect`, `subscribe` / `publish` / `receive`, `disconnect`.

Call / type	Notes
<code>mqtt.Options</code>	<code>host</code> , <code>port</code> , <code>clientId</code> , <code>keepalive</code> (seconds), <code>security</code> , <code>username</code> , <code>password</code> .
<code>mqtt.Client</code>	A live connection (from <code>connect</code>).
<code>mqtt.Message</code>	A received message: <code>topic</code> (string), <code>payload</code> (bytes).
<code>mqtt.connect(opts)</code>	Open a connection, send CONNECT, check the CONNACK return code.

Call / type	Notes
<code>mqtt.subscribe(client, topic)</code>	Subscribe to a topic filter at QoS 0 and wait for the SUBACK.
<code>mqtt.publish(client, topic, message)</code>	Publish a UTF-8 text message at QoS 0 (fire and forget).
<code>mqtt.publishBytes(client, topic, payload)</code>	Publish a raw <code>bytes</code> payload at QoS 0.
<code>mqtt.receive(client)</code>	Block until the next application message arrives; returns a <code>Message</code> .
<code>mqtt.poll(client, timeoutMs)</code>	Poll up to <code>timeoutMs</code> ms; returns a <code>list of Message</code> of length 0 or 1.
<code>mqtt.ping(client)</code>	Send a PINGREQ keepalive (fire and forget).
<code>mqtt.disconnect(client)</code>	Send DISCONNECT and close.

`Options.security` is `"none"` (plaintext, port 1883) or `"tls"` (implicit TLS, `mqttts`, port 8883). `username / password ""` omit the CONNECT credentials. A non-empty `clientId` identifies the session to the broker.

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` throws a catchable `Error` (kind `"mqtt"`) when the broker refuses the connection (a non-zero CONNACK code) or does not answer with a CONNACK; `subscribe` throws when the SUBACK reports failure. 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, and the PUBLISH parser (including the QoS>0 packet-id skip) - is unit-tested in the overlay (`modules/mqtt_test.j`). The networked connect / subscribe / publish / receive / poll round-trip 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

Basics-first (MQTT 3.1.1, QoS 0). Deferred until a workload needs them:

- **QoS 1 / 2** handshakes (PUBACK / PUBREC / PUBREL / PUBCOMP with persistent packet-id state).
- **Retained messages and the will.**
- **Auto-reconnect / session resumption.**
- **MQTT 5 properties.**

If full QoS 1/2 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

The CONNECT and SUBSCRIBE handshakes carry a 30 s timeout, so a broker that accepts the connection but never acknowledges fails instead of hanging. `poll(client, ms)` already bounds how long it waits for a message; `receive` blocks until one arrives.

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`.

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	
<code>multipart.field(name, value)</code>	<code>Part</code>	a plain text field
<code>multipart.file(name, filename, contentType, data)</code>	<code>Part</code>	a file part (body is <code>bytes</code>)
<code>multipart.text(p)</code>	<code>string</code>	decode a part's body as UTF-8 (for field values)

Call	Returns	
<code>multipart.isFile(p)</code>	<code>bool</code>	whether the part carries a filename

Building

Call	Returns	
<code>multipart.build(parts)</code>	<code>Built</code>	encode with a fresh random boundary
<code>multipart.buildWith(parts, boundary)</code>	<code>Built</code>	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`.

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 $((T_2 - T_1) + (T_3 - T_4)) / 2$ and `delay` as $(T_4 - T_1) - (T_3 - T_2)$, where T_1/T_4 are the local send/receive instants and T_2/T_3 are the server's receive/transmit timestamps. A positive `offset` means the local clock is behind the server.

Query

Call	Returns	
<code>ntp.query(host)</code>	<code>Result</code>	query <code>host:123</code> with a 5-second timeout
<code>ntp.queryWith(address, timeoutMs)</code>	<code>Result</code>	query a full <code>host:port</code> 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`.

Surface

Call / type	Notes
<code>oauth.Config</code>	<code>tokenUrl</code> , <code>deviceUrl</code> , <code>clientId</code> , <code>clientSecret</code> , <code>scope</code> .
<code>oauth.Token</code>	<code>accessToken</code> , <code>tokenType</code> , <code>refreshToken</code> , <code>scope</code> , <code>expiresAt</code> (Unix seconds; 0 = unknown).
<code>oauth.DeviceAuth</code>	<code>deviceCode</code> , <code>userCode</code> , <code>verificationUri</code> , <code>interval</code> , <code>expiresAt</code> .
<code>oauth.clientCredentials(config)</code>	The Client Credentials grant (a service as itself) -> <code>Token</code> .

Call / type	Notes
<code>oauth.refresh(config, refreshToken)</code>	Trade a refresh token for a new <code>Token</code> (keeps the refresh token when the server omits it).
<code>oauth.deviceStart(config)</code>	Begin the Device Authorization grant -> <code>DeviceAuth</code> (show the user the URL + code).
<code>oauth.deviceWait(config, deviceAuth)</code>	Poll until the user approves -> <code>Token</code> (backs off on <code>slow_down</code>).
<code>oauth.isExpired(token)</code>	Whether the token is past its expiry (30s skew buffer).
<code>oauth.google(clientId, clientSecret, scope)</code>	A <code>Config</code> with Google's endpoints.
<code>oauth.microsoft(tenant, clientId, clientSecret, scope)</code>	A <code>Config</code> with a Microsoft 365 / Entra tenant's endpoints.
<code>oauth.save(path, token)</code> / <code>oauth.load(path)</code>	Persist / reload a token as JSON (via <code>fs</code>).

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.

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.

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 `math` / `strings` / `lists` / `convert`; runs on both binaries.

Security note. Randomness comes from `math`'s shared, seedable, **non-cryptographic** RNG (the same source `uuid` draws from). It is predictable to an attacker who can reconstruct the seed, so generated passwords are **not** suitable for high-value secrets today. This swaps to a crypto-grade source when the `crypto` library lands. Use it for convenience passwords, fixtures, and policy checking - not for credentials that must resist a determined attacker.

```
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`.

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 0 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	
<code>password.schema()</code>	<code>Schema</code>	the strong default: 16 chars, all four classes, min 1 of each
<code>password.withLength(s, lo, hi)</code>	<code>Schema</code>	set the length range (<code>lo == hi</code> for a fixed length)
<code>password.withClasses(s, lo, up, dig, sym)</code>	<code>Schema</code>	enable/disable each class (bools)
<code>password.withMinimums(s, lo, up, dig, sym)</code>	<code>Schema</code>	set the per-class minimum counts
<code>password.withSymbolSet(s, chars)</code>	<code>Schema</code>	replace the symbol pool
<code>password.withoutAmbiguous(s)</code>	<code>Schema</code>	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

- **Non-crypto randomness** (see the security note) - swaps to `crypto` later.
- **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

- [uuid.md](#) - the same non-crypto RNG caveat and eventual crypto swap.
- [modules/index.md](#) - the module catalog and import rules.

pdfwriter - generate simple PDF documents

Import with `import "pdfwriter.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 `htmlwriter` / `label` generate their formats. Content streams are `FlateDecode`-compressed via `compress`. Pure Jennifer; runs on **both** binaries.

```
import "pdfwriter.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/pdfwriter_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	
<code>pdf.document()</code>	<code>Document</code>	an empty document
<code>pdf.page(width, height)</code>	<code>Page</code>	a blank page of the given size
<code>pdf.text(pg, x, y, font, size, str)</code>	<code>Page</code>	draw text at (x, y)
<code>pdf.line(pg, fromX, fromY, toX, toY)</code>	<code>Page</code>	draw a stroked line

Call	Returns	
<code>pdf.rect(pg, x, y, width, height, filled)</code>	Page	draw a rectangle (fill or stroke)
<code>pdf.color(pg, red, green, blue)</code>	Page	set fill + stroke colour for what follows
<code>pdf.addPage(doc, pg)</code>	Document	append a page
<code>pdf.render(doc)</code>	bytes	the finished PDF

`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: "pdfwriter"}`:

```
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.

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 pdfwriter")

Key	
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 pdfwriter"`; 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'
```

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, an Info dictionary when any metadata is set, a cross-reference table with correct byte offsets, and the trailer. 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: `pdfwriter` 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.
- **No embedded fonts or images** yet - a follow-on. Only the built-in base fonts, so no font file is embedded and non-Latin text is out of scope.
- **A writer, not a reader.** It generates PDFs; it does not parse them.

See also

- [compress.md](#) - the FlateDecode (`zlib`) streams.
- [htmlwriter.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.

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 `USER` / `PASS` auth. 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](#) .

Surface

A session is stateful: `connect`, issue commands, `quit`. `fetchAll` wraps the common “get every message” case.

Call / type	Notes
<code>pop.Options</code>	<code>host</code> , <code>port</code> , <code>security</code> , <code>user</code> , <code>pass</code> .
<code>pop.Session</code>	A live session over one connection (from <code>connect</code>).
<code>pop.Stat</code>	<code>count</code> and total <code>size</code> , from <code>stat</code> .

Call / type	Notes
<code>pop.connect(opts)</code>	Open a session: greet, optional STLS, USER / PASS .
<code>pop.stat(session)</code>	Mailbox Stat (STAT).
<code>pop.count(session)</code>	Just the message count.
<code>pop.sizes(session)</code>	list of int - each message's octet size, in order (LIST).
<code>pop.retrieve(session, n)</code>	Message n as a raw string (RETR), for mime.parse .
<code>pop.deleteMessage(session, n)</code>	Mark message n for deletion (DELE); removed at quit .
<code>pop.quit(session)</code>	End the session (commit deletions) and close.
<code>pop.fetchAll(opts)</code>	Connect, retrieve every message (no delete), quit; list of string .

`Options.security` is "none" (plaintext, port 110), "tls" (implicit TLS on connect, port 995), or "starttls" (STLS upgrade on 110).

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.

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`.
- **USER / PASS , or XOAUTH2** (`Options.auth = "xoauth2"` , via `sasl` , for Google / Microsoft 365). `APOP` (MD5 challenge) and the SASL challenge-response mechanisms land with the `crypto` library.
- **No TOP / UIDL** . Just `STAT` / `LIST` / `RETR` / `DELE` .
- An internationalized (IDN) host is IDNA-encoded to its `xn--` form automatically (via `idna`).

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.

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`.

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
<code>prometheus.Metric</code>	<code>name</code> , <code>help</code> , <code>type</code> ("counter"/"gauge"), <code>samples</code> .
<code>prometheus.Sample</code>	<code>labels</code> (map), <code>value</code> (float) - one rendered line.
<code>prometheus.counter(name, help)</code>	A new counter metric; throws on an invalid name.
<code>prometheus.gauge(name, help)</code>	A new gauge metric; throws on an invalid name.

Call / type	Notes
<code>prometheus.observe(metric, labels, value)</code>	Record a sample (upsert by label set); throws on an invalid label name.
<code>prometheus.render(metrics)</code>	Render a list of <code>Metric</code> as the text exposition format.

`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. `render` sorts label keys, so output is deterministic regardless of the order labels were inserted.

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
<code>prometheus.Result</code>	<code>resultType</code> + <code>series</code> (a list of <code>Series</code>).
<code>prometheus.Series</code>	<code>metric</code> (label map) + <code>values</code> (a list of <code>Point</code>).

Call / type	Notes
<code>prometheus.Point</code>	<code>timestamp</code> (float, Unix seconds) + <code>value</code> (float).
<code>prometheus.query(base, promql)</code>	Instant query (<code>/api/v1/query</code>) -> <code>Result</code> .
<code>prometheus.queryRange(base, promql, start, end, step)</code>	Range query (<code>/api/v1/query_range</code>) -> <code>Result</code> .

`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, and the upsert - 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

- **counter and gauge only.** `histogram` and `summary` (buckets / quantiles, the `_bucket` / `_sum` / `_count` child series) are a documented follow-on.
- **No registry / auto-collection.** The caller holds and assembles the metric set; there is no global default registry or process/Go collectors.
- **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 - a fixed-window rate limiter on memcached

Import with `import "ratelimit.j" as ratelimit;`. A **fixed-window rate limiter** on the `memcache` module - the sharpest use of memcached's distinctive strength: **atomic incr plus a per-key TTL**. Each key (a client IP, a user, an API token) counts hits in a time window; the counter is armed with the window's expiry when it is first created, so it resets on its own when the window ends - there is nothing to reap. Because it builds on `memcache` (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 "ratelimit.j" as ratelimit;
import "memcache.j" as memcache;

def mc as memcache.Session init memcache.connect(memcache.Options{
    host: "127.0.0.1", port: 11211});

# 100 requests per 60 seconds, per client IP
if (ratelimit.allow($mc, "ip:203.0.113.7", 100, 60)) {
    # ... serve the request ...
} else {
    # ... reject (e.g. HTTP 429) ...
}
```

Runnable: [examples/modules/ratelimit_demo.j](#).

Surface

Call	Notes
<code>ratelimit.allow(mc, key, limit, window)</code>	Record one hit; <code>true</code> if within <code>limit</code> for the current <code>window</code> (seconds).
<code>ratelimit.remaining(mc, key, limit)</code>	Hits left in the current window (the full <code>limit</code> when untouched, <code>0</code> once exhausted).

How the window works

`allow` does an atomic `incr` on the key. The window starts at the **first** hit: an absent counter is created (via `add`) carrying the window's TTL, and because a later `incr` does not re-arm the expiry, the counter dies exactly `window` seconds after that first hit - a clean fixed window with no background cleanup. The `incr`-then-`add` pair also closes the create race: if two callers both find the counter absent, only one `add` wins and the loser re-`incr`s, so no hit is lost.

```
def ok as bool init ratelimit.allow($mc, "user:ada", 5, 60); # 5 per minute
io.printf("%d left\n", ratelimit.remaining($mc, "user:ada", 5));
```

`allow` returns `true` for the first `limit` hits in a window and `false` afterwards; the counter keeps rising while denied, and `remaining` reports `0`, until the window expires and the budget refills.

Out of scope

- **Fixed window only.** The count resets at the window boundary, so up to `2 * limit` hits can land across two adjacent windows in the worst case. A **sliding window** or **token bucket** (smoother, burst-tolerant) is a later refinement.
- **Not a distributed clock.** The window is per key in one memcached; it does not coordinate wall-clock alignment across instances.
- **Volatile.** memcached can evict a counter under memory pressure, which resets that key's window early - acceptable for throttling, not for billing.

See also

- [memcache.md](#) - the `incr` / `add` / `get` primitives this uses.
- [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](#).

Surface

A session is stateful: `connect`, issue commands, `quit`.

Call / type	Notes
<code>redis.Options</code>	<code>host</code> , <code>port</code> , <code>security</code> , <code>user</code> , <code>password</code> , <code>db</code> .
<code>redis.Session</code>	A live session over one connection (from <code>connect</code>).
<code>redis.Reply</code>	A parsed reply: <code>kind</code> , <code>str</code> , <code>num</code> , <code>items</code> (see below).
<code>redis.connect(opts)</code>	Open a session; AUTH when <code>password</code> set, SELECT when <code>db > 0</code> .
<code>redis.command(session, args)</code>	Send any command (list of string); returns the raw <code>Reply</code> .

Call / type	Notes
<code>redis.get(session, key)</code>	GET - the string value, or "" when the key is missing.
<code>redis.set(session, key, value)</code>	SET .
<code>redis.del(session, key)</code>	DEL - number of keys removed (0 or 1).
<code>redis.exists(session, key)</code>	EXISTS - bool .
<code>redis.incr(session, key)</code>	INCR - the new value (int).
<code>redis.decr(session, key)</code>	DECR - the new value (int).
<code>redis.keys(session, pattern)</code>	KEYS - list of string matching a glob ("*" , "user:*").
<code>redis.ping(session)</code>	PING - the server's "PONG" .
<code>redis.quit(session)</code>	QUIT and close.

`Options.security` is "none" (plaintext, port 6379) or "tls" (implicit TLS, `rediss`). `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)

kind	RESP source	Read from
"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.

Bulk strings are read as UTF-8 text

RESP bulk-string lengths are byte counts, but the parser reads values as UTF-8 text. This is byte-exact for ASCII and UTF-8 string values - the common case (keys, JSON payloads, counters). A binary value whose byte length differs from its rune length is not yet byte-exact; store such values base64-encoded (via `encoding`) until a byte-native read lands.

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 - is unit-tested in the overlay (`modules/redis_test.j`). The networked session is 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, and the generic `command` for the rest. Lists / hashes / sets are reachable through `command`; typed helpers for them can follow.
- **No pipelining, pub/sub, or RESP3**. One request, one reply.
- **No connection pool**. One `Session` is one connection.
- **rediss TLS** rides `net`'s default certificate verification.

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 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`.

Surface

The module works over an existing `redis.Session` - it adds no transport of its own.

Call / type	Notes
<code>resque.Job</code>	A reserved job: <code>queue</code> , <code>class</code> , <code>args</code> (list of string).
<code>resque.enqueue(session, queue, class, args)</code>	Register <code>queue</code> and push a job (<code>class</code> + string <code>args</code>) onto it.
<code>resque.reserve(session, queues)</code>	Pop the next job from the first non-empty queue (priority order); an empty <code>Job</code> when all are drained.

Call / type	Notes
<code>resque.queueLength(session, queue)</code>	Pending jobs on one queue.
<code>resque.queues(session)</code>	Registered queue names (<code>list of string</code>).
<code>resque.size(session)</code>	Total pending jobs across every queue.
<code>resque.fail(session, job, message)</code>	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.

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:

- **Blocking reserve.** `reserve` polls; a `BLPOP`-based blocking wait (so a worker sleeps instead of spinning) is a later add.
- **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.Client{baseUrl: "https://api.example.com",
  headers: {"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](#).

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
<code>rest.Client</code>	<code>baseUrl</code> and default <code>headers</code> sent with every request.
<code>rest.Response</code>	<code>status</code> , <code>headers</code> , <code>body</code> .
<code>rest.get(c, path, query)</code>	GET; <code>query</code> is a map of string to string ({} for none).

Call / type	Notes
<code>rest.delete(c, path, query)</code>	DELETE.
<code>rest.post(c, path, contentType, body)</code>	POST with a raw body.
<code>rest.put(c, path, contentType, body)</code>	PUT with a raw body.
<code>rest.patch(c, path, contentType, body)</code>	PATCH with a raw body.
<code>rest.getJson(c, path, query)</code>	GET, decode the body -> <code>json.Value</code> .
<code>rest.postJson(c, path, body)</code>	POST a <code>json.Value</code> (encodes, sets <code>Content-Type</code>); -> <code>Response</code> .
<code>rest.putJson(c, path, body)</code>	PUT a <code>json.Value</code> .
<code>rest.patchJson(c, path, body)</code>	PATCH a <code>json.Value</code> .
<code>rest.bearer(token)</code>	An Authorization value: <code>Bearer <token></code> .
<code>rest.basic(user, pass)</code>	An Authorization value: <code>Basic <base64(user:pass)></code> .
<code>rest.withHeader(c, name, value)</code>	A copy of the client with one default header set.

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`.

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` will, however, `throw` if the body 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 cookie jar / stateful session.** A `Client` is a value threaded per call (a module has no mutable state); a stateful session that remembers cookies belongs on the `http` side (a system library may hold stateful handles; a module may not) and is deferred there.
- **No retry / backoff, no pagination helper, no auto-redirect** (redirects come back as their 3xx `Response`, from `http`).

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](#).

Surface

```
def struct ringbuffer.RingBuffer { items as list of string, capacity as int };
```

Call	Returns	
<code>ringbuffer.new(capacity)</code>	<code>RingBuffer</code>	an empty buffer of the given capacity (≥ 1)
<code>ringbuffer.push(rb, item)</code>	<code>RingBuffer</code>	append, dropping the oldest if already full
<code>ringbuffer.pop(rb)</code>	<code>RingBuffer</code>	remove the oldest (throws if empty)
<code>ringbuffer.first(rb)</code>	<code>string</code>	peek the oldest (throws if empty)
<code>ringbuffer.last(rb)</code>	<code>string</code>	peek the newest (throws if empty)
<code>ringbuffer.size(rb)</code>	<code>int</code>	the number of entries held
<code>ringbuffer.capacity(rb)</code>	<code>int</code>	the capacity
<code>ringbuffer.isEmpty(rb)</code>	<code>bool</code>	whether it holds no entries
<code>ringbuffer.isFull(rb)</code>	<code>bool</code>	whether it is at capacity
<code>ringbuffer.toList(rb)</code>	<code>list of string</code>	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.

sasl - SASL authentication encoders

Import with `import "sasl.j" as sasl;`. The crypto-free SASL mechanisms as pure base64 encoders, shared by the mail clients (`smtp` / `pop` / `imap`). These format the client tokens; the protocol clients run the mechanism-specific wire dialogue around them. **No networking and no crypto**, so this module is TinyGo-clean and 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` (and it is exercised end to end by the mail clients' demos, e.g. `smtp_demo.j`).

Surface

Call	Returns	Notes
<code>sasl.plain(user, pass)</code>	<code>string</code>	SASL PLAIN: base64 of <code>"\0user\0pass"</code> .
<code>sasl.loginUser(user)</code>	<code>string</code>	SASL LOGIN step 1: base64 of the username.
<code>sasl.loginPass(pass)</code>	<code>string</code>	SASL LOGIN step 2: base64 of the password.
<code>sasl.bearer(user, token)</code>	<code>string</code>	SASL XOAUTH2: an OAuth2 bearer-token response.

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

- **Crypto-free mechanisms only.** The challenge-response mechanisms (`SCRAM-SHA-256`, `CRAM-MD5`) need HMAC / PBKDF2 and join this module when the `crypto` library (M20.1) lands.
- **Encoders, not transport.** `sasl` formats tokens; the SMTP `AUTH`, IMAP `AUTHENTICATE`, and POP3 `AUTH` dialogue lives in the respective clients.

See also

- [smtp.md](#) / [pop.md](#) / [imap.md](#) - the mail clients that consume these encoders.
- [encoding.md](#) - the base64 codec `sasl` builds on.
- [modules/index.md](#) - the module catalog and import rules.

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](#) 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`.

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
<code>semver.parse(s)</code>	<code>Version</code>	Parse a strict version string; throw <code>s(kind: "value")</code> on invalid.

Call	Returns	Notes
<code>semver.isValid(s)</code>	<code>bool</code>	Whether <code>s</code> is a strict SemVer string (no throw).
<code>semver.coerce(s)</code>	<code>string</code>	Extract a version from loose text (a <code>v</code> -tag, a partial); <code>""</code> if none.
<code>semver.clean(s)</code>	<code>string</code>	Strict-normalise (trim, drop <code>v</code> / <code>=</code>); <code>""</code> if not a full version.
<code>semver.toString(v)</code>	<code>string</code>	Canonical form; round-trips <code>parse</code> .
<code>semver.compare(a, b)</code>	<code>int</code>	<code>-1</code> / <code>0</code> / <code>1</code> by SemVer precedence. Build metadata ignored.
<code>semver.lt(a, b)</code>	<code>bool</code>	<code>compare(a, b) < 0</code> .
<code>semver.eq(a, b)</code>	<code>bool</code>	<code>compare(a, b) == 0</code> (so <code>1.0.0+a</code> equals <code>1.0.0+b</code>).
<code>semver.gt(a, b)</code>	<code>bool</code>	<code>compare(a, b) > 0</code> .
<code>semver.gte(a, b)</code>	<code>bool</code>	<code>compare(a, b) >= 0</code> .
<code>semver.lte(a, b)</code>	<code>bool</code>	<code>compare(a, b) <= 0</code> .
<code>semver.neq(a, b)</code>	<code>bool</code>	<code>compare(a, b) != 0</code> .
<code>semver.diff(a, b)</code>	<code>string</code>	The change kind: <code>"major"</code> / <code>"minor"</code> / <code>"patch"</code> / <code>"prerelease"</code> / <code>""</code> .
<code>semver.isStable(v)</code>	<code>bool</code>	<code>major >= 1</code> and no prerelease. <code>0.y.z</code> is unstable by convention.
<code>semver.isPrerelease(v)</code>	<code>bool</code>	Whether a prerelease tag is present.
<code>semver.incMajor(v)</code>	<code>Version</code>	<code>major+1</code> ; resets minor / patch and clears prerelease + build.
<code>semver.incMinor(v)</code>	<code>Version</code>	<code>minor+1</code> ; resets patch and clears prerelease + build.
<code>semver.incPatch(v)</code>	<code>Version</code>	<code>patch+1</code> ; clears prerelease + build.
<code>semver.sort(vs)</code>	<code>list of Version</code>	A new list ordered ascending by precedence.
<code>semver.rsort(vs)</code>	<code>list of Version</code>	A new list ordered descending (highest first).

Call	Returns	Notes
<code>semver.satisfies(ver, range)</code>	<code>bool</code>	Whether the version string matches the range. See Ranges .
<code>semver.maxSatisfying(vers, range)</code>	<code>string</code>	Highest version string in <code>vers</code> matching <code>range</code> , or <code>""</code> .
<code>semver.minSatisfying(vers, range)</code>	<code>string</code>	Lowest version string in <code>vers</code> matching <code>range</code> , or <code>""</code> .
<code>semver.validRange(range)</code>	<code>bool</code>	Whether a range expression is well-formed.
<code>semver.minVersion(range)</code>	<code>string</code>	The lowest version that could satisfy <code>range</code> (its floor), or <code>""</code> .
<code>semver.intersects(a, b)</code>	<code>bool</code>	Whether two ranges share any satisfying version.
<code>semver.subset(inner, outer)</code>	<code>bool</code>	Whether every version in <code>inner</code> is also allowed by <code>outer</code> .
<code>semver.gtr(v, range)</code>	<code>bool</code>	Whether <code>v</code> is above the whole range.
<code>semver.ltr(v, range)</code>	<code>bool</code>	Whether <code>v</code> is below the whole range.
<code>semver.outside(v, range)</code>	<code>bool</code>	<code>gtr(v, range)</code> or <code>ltr(v, range)</code> .
<code>semver.simplifyRange(vers, range)</code>	<code>string</code>	The shortest range matching the same subset of <code>vers</code> .

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
<code>semver.intersects(a, b)</code>	do ranges <code>a</code> and <code>b</code> share any version? <code>^1.2.0 ∩ >=1.5.0 = true</code> ; <code>^1.2.0 ∩ ^2.0.0 = false</code>
<code>semver.subset(inner, outer)</code>	is every version in <code>inner</code> also in <code>outer</code> ? <code>subset("^1.5.0", "^1.0.0") = true</code>
<code>semver.gtr(v, range)</code>	is <code>v</code> above the whole range?
<code>semver.ltr(v, range)</code>	is <code>v</code> below the whole range?

Call	Question
<code>semver.outside(v, range)</code>	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 on memcached

Import with `import "session.j" as session;`. A server-side **session store** on the `memcache` module - the canonical memcached use. A session is a `map of string to string` held under a `sess:ID` key with a sliding TTL, so it expires on its own when idle. It threads three pieces together: `memcache` (store + TTL), `uuid` (the session ID), and `json` (encode the map). Because it builds on `memcache` (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 "session.j" as session;
import "memcache.j" as memcache;

def mc as memcache.Session init memcache.connect(memcache.Options{
    host: "127.0.0.1", port: 11211});

def id as string init session.create($mc, 1800);           # 30-minute session
def data as map of string to string init session.load($mc, $id);
$data["user"] = "ada";
session.save($mc, $id, $data, 1800);
```

Runnable: `examples/modules/session_demo.j`.

Surface

Each call takes a live `memcache.Session` and a session `id`; `ttl` is the expiry in **seconds**.

Call	Notes
<code>session.create(mc, ttl)</code>	Mint a new ID, store an empty session; returns the <code>id</code> (string).
<code>session.load(mc, id)</code>	The session's <code>map of string to string</code> , or an empty map when absent / expired.
<code>session.save(mc, id, data, ttl)</code>	Write the data map and re-arm the expiry to <code>ttl</code> seconds.

Call	Notes
<code>session.touch(mc, id, ttl)</code>	Re-arm the expiry without rewriting the data; returns whether it still existed.
<code>session.destroy(mc, id)</code>	Remove the session; returns whether it existed.

The typical request cycle: `load` the session at the start, read / write the map, `save` it at the end (which also slides the expiry). A quiet request that only needs to keep the session alive can `touch` instead of `save`.

Storage format

The data map is stored as **base64-wrapped JSON** under `sess:ID`. The base64 wrap keeps the cached value pure ASCII, so a session value with any UTF-8 text (`"name": "José"`) round-trips exactly - memcached's value read is byte-exact for ASCII, and base64 makes every value ASCII on the wire. The trade-off is that the cached blob is not human-readable and not interchangeable with another framework's session format; a PHP-session-compatible layout is a separate follow-on.

Caveats

- **Volatile.** memcached evicts under memory pressure and loses everything on restart, so a session can vanish before its TTL. Treat sessions as a cache of soft state (a shopping cart, a wizard step), not a store of record. Anything that must survive belongs in a database.
- **Session IDs are UUID v4 from a non-crypto RNG** (see the `uuid` module - randomness draws from `math`'s seedable source). That is fine for a cache key; a session ID used as an authentication token wants a cryptographic source, which lands with the planned `crypto` library.
- **String values only.** A session is a `map of string to string`; encode richer values (numbers, nested data) yourself, e.g. via `json` or `convert.toString`.

See also

- [memcache.md](#) - the cache client this builds on.
- [uuid.md](#) - the session-ID source.
- [json.md](#) - the map serialization.

- [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](#), 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`.

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](#) 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
<code>slack.message()</code>	<code>Message</code>	start an empty message
<code>slack.text(m, text)</code>	<code>Message</code>	set the fallback / notification text

Call	Returns	
<code>slack.section(m, markdown)</code>	Message	append a section block (mrkdwn)
<code>slack.header(m, heading)</code>	Message	append a header block (plain text)
<code>slack.divider(m)</code>	Message	append a divider block
<code>slack.render(m)</code>	string	render the JSON payload
<code>slack.sendMessage(webhookUrl, m)</code>	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.

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` / `divider`. Fields, accessories, images, and 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`. 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](#).

Surface

Call / type	Notes
<code>smtp.Options</code>	<code>host</code> , <code>port</code> , <code>security</code> , <code>clientName</code> , <code>user</code> , <code>pass</code> .
<code>smtp.send(opts, from, recipients, message)</code>	Deliver message to every recipient; throws on a server rejection.

`Options` fields:

Field	Notes
<code>host</code>	Server hostname.

Field	Notes
port	Server port (25 / 587 plaintext or STARTTLS, 465 implicit TLS).
security	"none" (plaintext), "starttls" (upgrade after EHLO), "tls" (implicit TLS on connect).
clientName	The EHLO identity; defaults to "localhost" when empty.
auth	SASL mechanism: "" (auto - PLAIN when user is set, else none), "plain", "login", or "xoauth2".
user	SASL username; "" with auth: "" skips authentication.
pass	SASL password.

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"`).
2. Read the `220` greeting, send `EHLO`.
3. For `"starttls"`: `STARTTLS`, then `net.startTLS` and a second `EHLO`.
4. Authenticate per `auth` (via the `sasl` encoders): `AUTH PLAIN`, the `AUTH LOGIN` two-step, or `AUTH XOAUTH2` (an OAuth2 bearer token in `pass` - how Google / Microsoft 365 authenticate).
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.

Certificate verification for `"tls"` / `"starttls"` is the `net` default (on; see [net.md](#) for the opt-out).

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.
- **PLAIN / LOGIN / XOAUTH2** (via `sasl`). The challenge-response mechanisms (`CRAM-MD5` , `SCRAM`) need the `crypto` library and land with it.
- **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.

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.

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` .

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	
<code>statsd.client(host)</code>	<code>Client</code>	connect to <code>host:8125</code> (the default port), no prefix
<code>statsd.clientWith(address, prefix)</code>	<code>Client</code>	connect to a full <code>host:port</code> with a metric-name prefix
<code>statsd.close(c)</code>		close the sending socket

Metrics

Call	Wire line	Type
<code>statsd.count(c, name, value)</code>	<code>name:value c</code>	counter delta (<code>value</code> may be negative)
<code>statsd.increment(c, name)</code>	<code>name:1 c</code>	counter +1
<code>statsd.decrement(c, name)</code>	<code>name:-1 c</code>	counter -1
<code>statsd.gauge(c, name, value)</code>	<code>name:value g</code>	absolute gauge
<code>statsd.timing(c, name, ms)</code>	<code>name:ms ms</code>	timer, milliseconds
<code>statsd.set(c, name, value)</code>	<code>name:value s</code>	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. Counter and gauge values are integers in this version; `set` values are strings (any unique identifier).

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.
- **No sample rates or tags in this version.** The StatsD `@rate` sampling suffix and Datadog `#tag:value` tags are not emitted; every call sends its metric unconditionally with no tags. Both are candidate additions.
- **Integer counter / gauge values.** Fractional gauges (e.g. a load average) would need a float-valued surface, not shipped here.
- **No batching.** One datagram per metric. Aggregating several metrics into a single packet is a possible follow-on.

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](#): 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](#).

The bot

```
def struct telegram.Bot { token as string, baseUrl as string };
```

Call	Returns	
<code>telegram.bot(token)</code>	<code>Bot</code>	a bot against the public Telegram API
<code>telegram.botWith(token, baseUrl)</code>	<code>Bot</code>	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	
<code>telegram.sendMessage(bot, chatId, text)</code>	<code>Message</code>	send a text message
<code>telegram.sendMessageWith(bot, chatId, text, parseMode)</code>	<code>Message</code>	with a parse mode ("Markdown" , "MarkdownV2" , "HTML" , or "")
<code>telegram.sendPhoto(bot, chatId, photo, caption)</code>	<code>Message</code>	send a photo by URL or file id, with an optional caption
<code>telegram.sendChatAction(bot, chatId, action)</code>	<code>bool</code>	show activity ("typing" , "upload_photo" , ...)
<code>telegram.getMe(bot)</code>	<code>User</code>	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
};
```

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)
```

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`, `callback_query`, and inline queries set `hasMessage` false - reach the raw JSON via a direct `http` call if you need them.
- **No multipart upload.** `sendPhoto` takes a URL or file id, not a local file (multipart `multipart/form-data` upload is a follow-on).
- **No inline keyboards / reply markup, 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<b</p>"
```

Runnable: `examples/modules/tengine_demo.j`.

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
<code>tengine.newSet()</code>	<code>Set</code>	A fresh, empty template set.
<code>tengine.add(set, name, src)</code>	<code>Set</code>	Register <code>src</code> under <code>name</code> ; any <code>{{ define }}</code> blocks become their own entries. Value-semantic (returns a new <code>Set</code>).
<code>tengine.render(set, entry, data)</code>	<code>string</code>	Render <code>entry</code> against a <code>json.Value</code> .

`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
<code>{{ .a.b }} / {{ . }} / {{ \$.x }} / {{ \$var }}</code>	Output a value (see pipes).
<code>{{ if COND }} A {{ else if COND }} B {{ else }} C {{ end }}</code>	Conditional with any number of <code>else if</code> s and an optional final <code>else</code> .
<code>{{ range .items }} A {{ else }} B {{ end }}</code>	Render <code>A</code> once per element (of a list) or value (of a map, insertion order), rebinding <code>.</code> each time; <code>B</code> for an empty collection.
<code>{{ range \$i, \$e := .items }} ... {{ end }}</code>	As above, also binding <code>\$i</code> (index / key) and <code>\$e</code> (element). <code>{{ range \$e := .items }}</code> binds just <code>\$e</code> .
<code>{{ with .x }} A {{ else }} B {{ end }}</code>	Rebind <code>.</code> to <code>.x</code> for <code>A</code> when truthy, else <code>B</code> .
<code>{{ \$x := PIPE }}</code>	Assign a variable (produces no output).
<code>{{ define "name" }} ... {{ end }}</code>	Define a named template (collected when the source is <code>add ed</code>).
<code>{{ template "name" . }}</code>	Render a named template with the given node as <code>.</code> (and <code>\$</code>).
<code>{{ block "name" . }}</code> <code>default {{ end }}</code>	Render the set's <code>name</code> template if one exists, otherwise the inline default.
<code>{{/* comment */}}</code>	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
<code>eq a b / ne a b</code>	Equal / not equal (numbers compare numerically, strings and bools by value).
<code>lt / le / gt / ge</code>	Ordered comparison (numeric when both are numbers, else lexical).
<code>and x y ... / or x y ...</code>	Boolean and / or over any number of truthy tests.
<code>not x</code>	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
<code>upper / lower</code>	Case conversion.
<code>title</code>	Upper-case the first letter of each word.
<code>trim</code>	Strip surrounding whitespace.
<code>html</code>	Escape <code>&</code> <code><</code> <code>></code> <code>"</code> <code>'</code> as HTML entities.
<code>urlize</code>	Slugify: lower-case, keep alphanumerics, collapse spaces / dashes / underscores to single dashes.
<code>default x</code>	The value if it is truthy, otherwise <code>x</code> (a fallback for optional fields).
<code>truncate N</code>	The first <code>N</code> characters, with <code>...</code> appended when it shortened (excerpts).
<code>join SEP</code>	Join a list's elements into a string with <code>SEP</code> between them.
<code>len</code>	The length: characters of a string, or elements of a list / map.
<code>printf</code> <code>FORMAT</code>	Format the piped value per <code>FORMAT</code> (see below).

An unknown pipe throws a catchable `Error` (kind `"tengine"`).

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.
- [htmlwriter.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](#) over [RFC 4226](#) 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](#).

Options

A `totp.Options` carries the parameters; a zero-value struct (`def o as totp.Options;`) means the common authenticator defaults.

Field	Effect
<code>digits</code> (int)	Code length; <code>0</code> means 6.
<code>period</code> (int)	Time step in seconds; <code>0</code> means 30.
<code>algorithm</code> (string)	HMAC digest: <code>"sha1"</code> (default), <code>"sha256"</code> , or <code>"sha512"</code> ; <code>""</code> means <code>"sha1"</code> .

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
<code>totp.generate(secret, opts)</code>	<code>string</code>	The code for the current time.
<code>totp.generateAt(secret, unixSeconds, opts)</code>	<code>string</code>	The code for an explicit Unix time. Deterministic - use it in tests.
<code>totp.verify(secret, code, opts)</code>	<code>bool</code>	True if <code>code</code> is valid for the current time.
<code>totp.verifyAt(secret, code, unixSeconds, opts)</code>	<code>bool</code>	True if <code>code</code> is valid for an explicit Unix time. Deterministic.
<code>totp.uri(issuer, account, secret, opts)</code>	<code>string</code>	The <code>otpauth://totp/...</code> 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.

`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 enrolls 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 secret is the shared key: store it like a password, and transmit the provisioning URI over a secure channel only.
- Randomness for a *new* secret is out of scope here - generate one from a cryptographic source and base32-encode it with `encoding.toText(bytes, "base32")`. (`math.rand*` is **not** crypto-grade; that lands with the future `crypto` library.)
- 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.

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`.

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,             # N family (last) name
  given as string,              # N given (first) name
  organization as string,       # ORG ("" when unset)
  title as string,              # TITLE ("" when unset)
  emails as list of string,      # EMAIL (0..n)
  phones as list of string,      # TEL (0..n)
  addresses as list of Address,  # ADR (0..n)
  url as string,                 # URL ("" when unset)
  note as string                 # NOTE ("" when unset)
};

def struct vcard.Address {
  street as string, locality as string, region as string,
  postalCode as string, country as string
};
```

Building

The builders are **value-semantic** - each returns a fresh copy and never mutates its argument, so you thread them:

Call	Returns	
<code>vcard.card(formattedName)</code>	Card	a card with just its <code>FN</code> display name
<code>vcard.withName(c, family, given)</code>	Card	set the structured <code>N</code> name
<code>vcard.withOrg(c, organization, title)</code>	Card	set <code>ORG</code> and <code>TITLE</code>
<code>vcard.addEmail(c, email)</code>	Card	append an <code>EMAIL</code>
<code>vcard.addPhone(c, phone)</code>	Card	append a <code>TEL</code>
<code>vcard.address(street, locality, region, postalCode, country)</code>	Address	build an address
<code>vcard.addAddress(c, address)</code>	Card	append an <code>ADR</code>
<code>vcard.withUrl(c, url)</code>	Card	set the <code>URL</code>
<code>vcard.withNote(c, note)</code>	Card	set the <code>NOTE</code>

```
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	
<code>vcard.encode(c)</code>	string	one card as a <code>VCARD</code> (CRLF-terminated)
<code>vcard.encodeAll(cards)</code>	string	many cards concatenated
<code>vcard.parse(text)</code>	list of Card	parse one or many <code>VCARD</code> s

`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 methods by name, 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).

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");
```

Because Jennifer has no first-class functions, a route stores the handler's **name** and dispatch happens by name via `meta.callMain` - the primitive that lets the framework module reach a handler defined in the entry program across the module boundary. You never register a function value; you register a name, and `web.get` / `web.route` check at registration time that the method exists.

The `web.Context`

Everything a handler needs hangs off the `web.Context` it receives, so it rarely touches `httpd` directly:

Call	Returns	
<code>web.method(\$ctx)</code>	<code>string</code>	Request method.
<code>web.path(\$ctx)</code>	<code>string</code>	Request path.
<code>web.param(\$ctx, name)</code>	<code>string</code>	A captured <code>:param</code> ("" if none).
<code>web.query(\$ctx, name)</code>	<code>string</code>	A query-string parameter.
<code>web.header(\$ctx, name)</code>	<code>string</code>	A request header.
<code>web.body(\$ctx)</code>	<code>bytes</code>	The raw request body (binary-safe).
<code>web.bodyJson(\$ctx)</code>	<code>json.Value</code>	The request body decoded as JSON.
<code>web.form(\$ctx)</code>	map of <code>string</code> to <code>string</code>	The <code>x-www-form-urlencoded</code> body, decoded.
<code>web.formValue(\$ctx, name)</code>	<code>string</code>	One form field ("" if absent).
<code>web.multipartForm(\$ctx)</code>	list of <code>multipart.Part</code>	The <code>multipart/form-data</code> body (file uploads), parsed via the <code>multipart</code> module. Import <code>multipart.j</code> in the handler to name <code>multipart.Part</code> / use <code>multipart.isFile</code> / <code>multipart.text</code> .
<code>web.remoteAddr(\$ctx)</code>	<code>string</code>	The client's <code>host:port</code> .
<code>web.setHeader(\$ctx, name, value)</code>	<code>null</code>	Set a response header (before responding).
<code>web.respond(\$ctx, status, body)</code>	<code>null</code>	Send a response (body is a string).
<code>web.text(\$ctx, status, body)</code>	<code>null</code>	Respond with <code>text/plain</code> .
<code>web.html(\$ctx, status, body)</code>	<code>null</code>	Respond with <code>text/html</code> .
<code>web.sendJson(\$ctx, status, doc)</code>	<code>null</code>	Respond with <code>application/json</code> from a <code>json.Value</code> .
<code>web.sendGzip(\$ctx, status, body)</code>	<code>null</code>	Respond, gzip-compressed when the client accepts it.
<code>web.redirect(\$ctx, status, location)</code>	<code>null</code>	Redirect (301/302/303/307/308) with a <code>Location</code> header.
<code>web.serveFile(\$ctx, path)</code>	<code>null</code>	Respond with a file from disk.

Call	Returns	
<code>web.serveDir(\$ctx, root)</code>	<code>null</code>	Serve static files from a directory root.

Cookies

Pure HTTP-header work over `httpd`, no extra dependency:

Call	Returns	
<code>web.cookie(\$ctx, name)</code>	<code>string</code>	The request cookie's value ("" if absent).
<code>web.setCookie(\$ctx, name, value, opts)</code>	<code>null</code>	Emit a <code>Set-Cookie</code> 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	
<code>web.sessionId(\$ctx, cookieName)</code>	<code>string</code>	The request's session id, minting a new UUID + <code>HttpOnly</code> / <code>SameSite=Lax</code> / path- / cookie on first use.

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	
<code>web.new()</code>	An empty app.
<code>web.get(\$app, pattern, handler)</code>	Register a GET route.
<code>web.post</code> / <code>web.put</code> / <code>web.patch</code> / <code>web.delete</code>	The other verbs.
<code>web.route(\$app, method, pattern, handler)</code>	The general form.
<code>web.before(\$app, handler)</code>	Add a middleware (runs before each route handler).
<code>web.notFound(\$app, handler)</code>	A custom handler for unmatched requests (default: a plain 404).

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.

Authentication

`web` parses the incoming `Authorization` header; **checking** the credentials (against your user store) and sending the `401` challenge stay app code.

Call	Returns	
<code>web.basicAuth(\$ctx)</code>	<code>BasicCredentials</code>	Decode <code>Authorization: Basic base64(user:pass)</code> .
<code>web.bearerToken(\$ctx)</code>	<code>string</code>	The token from <code>Authorization: Bearer <token></code> ("" 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	
<code>web.csrfToken(\$ctx, secret)</code>	<code>string</code>	Mint a token, set the <code>csrf</code> cookie, return it for the form / an <code>X-CSRF-Token</code> header.
<code>web.csrfCheck(\$ctx, secret)</code>	<code>bool</code>	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	
<code>web.run(\$app, addr)</code>	Listen on <code>addr</code> and serve forever (blocks; interrupt to stop).
<code>web.serveOn(\$app, srv)</code>	Serve on an already-listening <code>httpd.Server</code> you hold - so you can shut it down (or serve from a <code>spawn</code>).

`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.

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](#).

Functions

Call	Returns	Notes
<code>webhook.sign(payload, secret)</code>	<code>string</code>	<code>sha256=</code> + hex HMAC-SHA256 of <code>payload</code> keyed by <code>secret</code> .
<code>webhook.verify(payload, signature, secret)</code>	<code>bool</code>	True if <code>signature</code> matches (constant-time compare).
<code>webhook.send(url, payload, secret)</code>	<code>http.Response</code>	POST <code>payload</code> to <code>url</code> 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.

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`.

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	
<code>websocket.connect(url)</code>	<code>Conn</code>	handshake with the default receive timeout (30 s)
<code>websocket.connectWith(url, timeoutMs)</code>	<code>Conn</code>	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	
<code>websocket.send(c, text)</code>		send a text message
<code>websocket.sendBytes(c, data)</code>		send a binary message
<code>websocket.ping(c)</code>		send a ping
<code>websocket.receive(c)</code>	Message	read the next message
<code>websocket.close(c)</code>		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).
- **Non-crypto masking key / nonce.** The 4-byte mask and the handshake nonce come from `math`'s non-crypto RNG - neither is a security boundary (masking defeats proxy cache poisoning; the nonce only needs to rarely repeat), so this is correct, not a `crypto` gap.

- **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`).

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.

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.

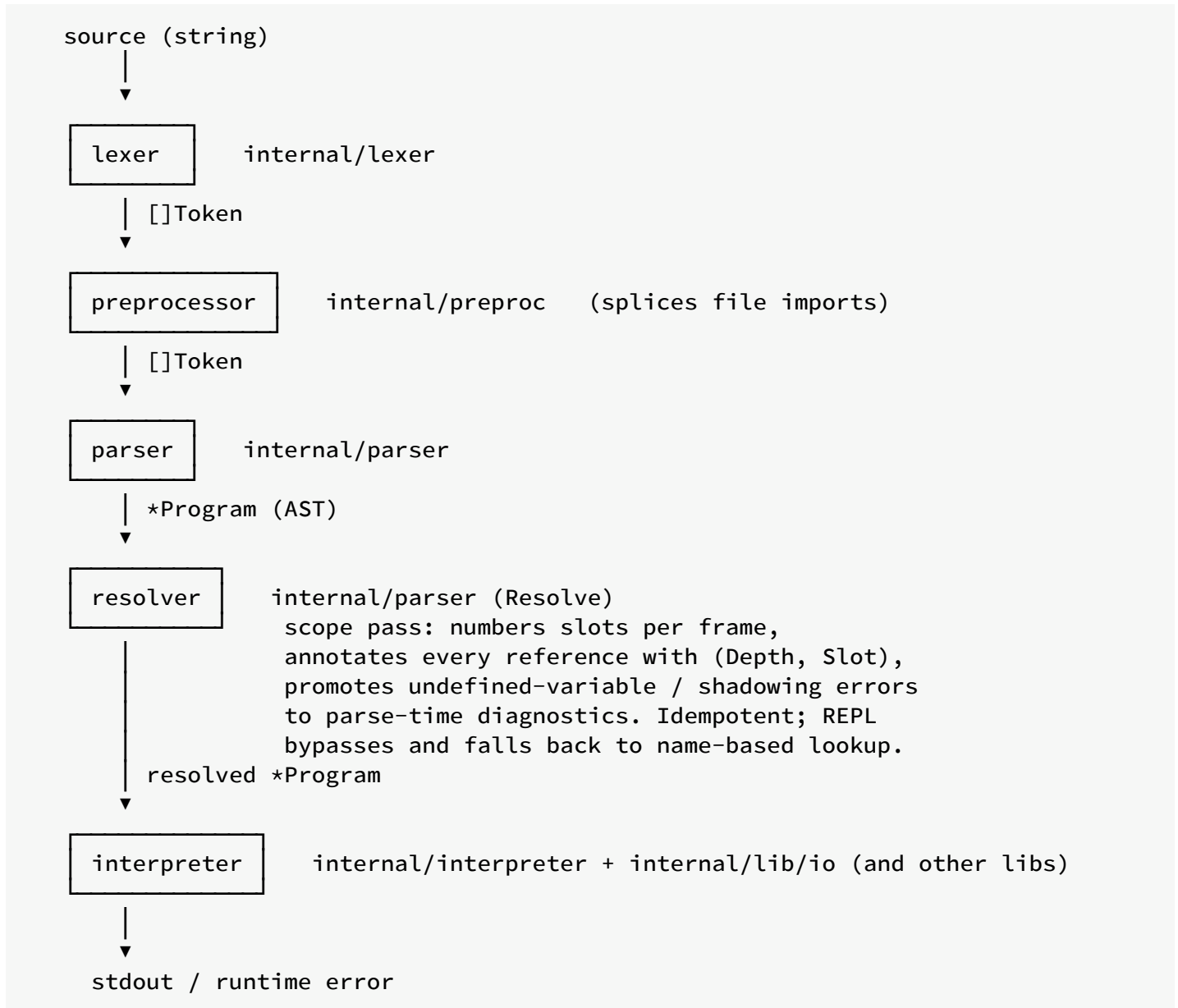
See <docs/design-stances.md> for the canonical table.

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.
- [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):



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 , 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 (==)
Assignment	ASSIGN (=)
Grouping and punctuation	LBRACE ({ }, RBRACE (}), LPAREN ((), RPAREN ()), LBRACKET ([], RBRACKET (]), SEMI (;), COMMA (,), COLON (:), DOT (.)

`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` are two-character (`<=` , `>=` , `==`) and are recognized by a one-character lookahead from `<` , `>` , `=` . `RETURN` is the keyword behind `return [EXPR];` (see [grammar.md](#)).

`VARREF` carries the variable name *without* the leading `$`. `STRING` carries the value *with* escape sequences already processed and *without* surrounding quotes.

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. Escape sequences (`\n`, `\t`, ...) are the conventional spelling; raw multi-line literals work too but aren't the canonical form `fmt` produces.

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`).

Identifier rule

Variable, method, parameter and library names use `[A-Za-z]{1,64}` only - no digits, no underscores. Constants use a looser form: uppercase chunks separated by single `_` characters - `[A-Z]+(_[A-Z]+)*`. Every `_` must be immediately followed by `[A-Z]`, so leading, trailing and consecutive underscores are all rejected.

The lexer reflects this by accepting `_` as a continuation character for bare IDENT tokens (so `MAX_RETRIES` is 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`. `$var` references go through a separate lexer path (`readVarRef`) that still uses the strict letters-only `isIdentPart`, so `$foo_bar` lex-errors directly.

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.

Grammar - EBNF

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.

[illegible]

```

                                in v1. *)
throwStmt    = "throw" expr ";" ;
                                (* expr may produce any
                                value; convention is an
                                `Error` struct. *)

returnStmt   = "return" [ expr ] ";" ;

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 } "." IDENT "=" expr ";" ;
                                (* l-value chain ending in `.field`.
                                Root is a VARREF; tail
                                may mix `[index]` and `.field`. *)

lvalueTail   = "[" expr "]" | "." IDENT ;

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 *)

exprStmt      = expr ";" ;

type          = primType | listType | mapType | taskType | structType ;
primType      = "int" | "float" | "string" | "bool" | "null" | "bytes" ;
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 type (bare
IDENT) or library-provided
namespaced struct type
(`IDENT.IDENT`). Resolved
at runtime against the
user-struct table or the
NSStructs table respectively;
unknown names are positioned
errors. *)

expr          = orExpr ;
orExpr        = andExpr { "or" andExpr } ;
andExpr       = notExpr { "and" notExpr } ;
notExpr       = "not" notExpr | compExpr ;
compExpr      = addExpr { "<" | ">" | "<=" | ">=" | "==" } addExpr ;
addExpr       = mulExpr { "+" | "-" } mulExpr ;
mulExpr       = unaryExpr { "*" | "/" | "/" | "%" } unaryExpr ;
unaryExpr     = "-" unaryExpr | primary ;
primary       = ( INT | FLOAT | STRING | "true" | "false" | "null"
| VARREF | qualifiedCall | qualifiedConstRef
| call | typeCall | structLit | constRef | "(" expr ")"
| listLit | mapLit | lenExpr | spawnExpr )
{ "[" expr "]" | "." IDENT } ;
                (* any primary can be index- or
                field-chained, including the
                `.field` form *)
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 [ "." IDENT ] "{" structLitField { "," structLitField } [ ","
] "}" ;

(* struct literal.
Bare IDENT names a user-defined
struct; `IDENT.IDENT` names a
library-provided namespaced
struct. The recogniser must
decide before the constant-name
check because struct names are
PascalCase / camelCase, not
uppercase.
The `{` after IDENT in
expression position is the
tie-breaker against `constRef`;
empty struct literals are
rejected (every field required). *)

structLitField = IDENT ":" expr ;
call           = IDENT "(" [ expr { "," expr } ] ")" ;
qualifiedCall  = IDENT "." IDENT "(" [ expr { "," expr } ] ")" ;
qualifiedConstRef = IDENT "." IDENT ;

(* qualifiedCall / qualifiedConstRef:
IDENT "." IDENT, then `( ` decides which.
Resolved against the namespaced-builtin
/ constant registry, gated by `use lib;`
(or alias-aware equivalent). *)

typeCall      = ("int" | "float" | "string" | "bool") "(" [ expr { "," expr } ] ")"
;

(* type keywords usable as calls only when
immediately followed by `( `; resolved as
convert-library builtins at runtime *)

constRef      = IDENT ;

(* bare-IDENT: constant reference; the
parser disambiguates `call` vs
`qualifiedCall` vs `constRef` by
peeking for "." / "(" . *)

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 *)

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 `<` `>` `<=` `>=` `==`, additive `+` `-`, multiplicative `*` `/` `%`, unary `-`. Binary operators are left-associative; `not` and unary `-` are right-associative (`not not x` and `--x` are both valid).
- `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).
- 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 must resolve to a constant in scope; a name that resolves to a variable produces a helpful error ("use `$name`").
- **Lists are array-backed sequences, not Lisp-style linked lists:** `def xs as list of int` `init [1, 2, 3]`. Element access is `$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 the EBNF above.

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, 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)
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
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

Node	Kind	Fields
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)
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
StructLit	expr	NS, Name, Fields []StructLitField - Point{...} bare or lib.Point{...} namespaced
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.

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.

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](#).

Aliasing correctness is exercised by [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 `ElemType`. 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.

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 $\rightarrow$ 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 for the same set of bindings:

- `vars map[string]Binding` - the name-keyed view. Present in every frame; the only view the REPL exercises because each REPL turn is a fresh parse with no resolver context linking it to prior-turn globals.
- `slots []Binding` - the slot-indexed view. Sized from `Block.NumSlots` at frame construction (`NewEnvironmentSized`) or grown on demand from `DefineAt`. Empty when the resolver didn't run.
- `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}` carries an extra `Slot` field so name-based writes can mirror into slot storage. `Slot` is `-1` on bindings installed by name-only `Define` (REPL, ad-hoc AST); non-negative when the resolver's `DefineAt` created it.

Storing the declared type lets `Assign` reject type-mismatching writes (you cannot assign a string to a variable declared as int).

Name-based API (fallback path):

- `Define(name, val, declType, isConst)` - adds to the current frame; errors if the name exists *anywhere in the chain* (the spec forbids shadowing). Sets `Binding.Slot = -1`.
- `Assign(name, val)` - walks up the chain to find the binding; errors if the binding is a constant, the value's kind doesn't match its declared type, or the name is undefined. When the target's `Binding.Slot >= 0`, the write also lands in `cur.slots[slot]` so the two views stay in sync.
- `Get(name)` and `GetBinding(name)` - walks up the chain.

Slot-based API (fast path):

- `DefineAt(slot, name, val, declType, isConst)` - installs the binding at `slots[slot]` (growing the slice if needed) and mirrors into `vars[name]` with `Binding.Slot = slot`.
- `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 both `slots[slot]` and `vars[name]` .

`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.

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`. Right after that, `resolveQualifiedRefs(prog)` walks the AST once and pre-fills every `QualifiedCallExpr.Fn / QualifiedConstRefExpr.Const` against the now-populated namespace tables. This pass has to run AFTER import processing (the tables didn't exist during Resolve) and is skipped by the REPL, which builds its namespaces incrementally.
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 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 `@claudio/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`.)

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`: parser side - the `typeCall` production lets `int(...)`, `float(...)`, `string(...)`, `bool(...)` parse despite their names being type keywords. `typeOf` is a normal IDENT call. `bool(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
<code>task.wait</code>	block on <code><-state.Done ; MarkObserved ; return Result</code> or wrap <code>Err</code> as <code>runtimeError</code>
<code>task.poll</code>	<code>BoolVal(state.IsDone())</code> via the non-blocking select on <code>state.Done</code>
<code>task.discard</code>	<code>MarkObserved ; return Null()</code> immediately (does not block)
<code>task.waitAll</code>	iterate list, wait each, mark all observed; return list-of-results or first error in order
<code>task.waitAny</code>	<code>[]reflect.SelectCase</code> over the list, <code>reflect.Select</code> , 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 <file.j>                   format source per docs/user-guide/style-guide.md
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
<code>run <file.j></code>	Run a Jennifer program (<code>-</code> reads source from stdin).	Module resolution flags
<code>repl</code>	Interactive read-eval-print loop, with a line editor and history.	REPL
<code>tokens <file.j></code>	Dump the lexer's token stream.	Inspection
<code>ast <file.j></code>	Dump the preprocessed AST as JSON.	Inspection
<code>fmt <file.j></code>	Format source per the style guide (stdout only).	Formatter
<code>lint <file.j></code>	Report compile-legal but suspect patterns.	Linter
<code>profile <file.j></code>	Per-position hit counts and wall-clock timings.	Profiler

Subcommand	What it does	Details
<code>test <file.j></code>	Discover and run the file's test methods.	Test runner
<code>serve <file.j></code>	Run a program; <code>--watch</code> re-runs it on change (a web-app reloader, or an autorun loop for any script).	The <code>serve</code> command
<code>version [-v]</code>	Print the build version (<code>-v</code> adds module-path layers).	Version injection
<code>help</code>	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).

`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
- 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/metilib.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.

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)

Key	Action
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 <file.j>` rewrites Jennifer source into the one canonical style defined in [../user-guide/style-guide.md](https://github.com/jennifer-lang/jennifer/blob/master/user-guide/style-guide.md) and prints the result to **stdout**. It never edits the file in place, so apply it by redirecting:

```
jennifer fmt prog.j           # preview the formatted source
jennifer fmt prog.j > out.j    # write it out (or pipe to `sponge`, an editor,
...)
```

There are no style options: like `gofmt`, the formatter is opinionated by design, so a whole codebase reads the same and diffs stay minimal.

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 <code>;</code> (<code>use io; import ...;</code> splits onto two lines).
Indentation	4 spaces per block level; a <code>}</code> dedents <i>before</i> it is written, so it lands at the outer level.
Operator spacing	a single space around binary operators (<code>\$a + \$b</code> , <code>\$x > 0</code>); none around a unary <code>-</code> (<code>-\$x</code>).
Punctuation	one space after each <code>,</code> and after the <code>;</code> s in a <code>for</code> header; no space <i>before</i> a <code>;</code> .
Blocks	<code>{</code> follows its header with a space (<code>if (\$c) {</code>); the body is indented; <code>}</code> sits on its own line.
<code>else / elseif</code>	cuddle the preceding brace on one line (<code>} else {</code>).
<code>for</code> header	the two <code>;</code> stay on the header line (<code>for (init; cond; step)</code>), not split across lines.
Calls / params	arguments and parameters get one space after each comma (<code>add(a as int, b as int)</code> , <code>f(a, b)</code>).
Strings	re-quoted with double quotes and standard escapes (<code>quoteJenniferString</code> , mirroring the lexer's <code>readString</code>).
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
<code>import "file.j";</code> statements	<code>fmt</code> works on the token stream <i>before</i> preprocessing, so imports are re-emitted, not inlined - the opposite of a splice.
User-written parentheses	<code>(\$x + 1) * 2</code> keeps its grouping; an AST-based formatter would erase parens the grammar makes redundant.
Comments (and blank lines)	<code>#</code> and nesting <code>/* */</code> 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
<code>indent</code>	current block depth; bumps on <code>{</code> , drops on <code>}</code> (the closing brace dedents before it is written).
<code>prevIs0operand</code>	answers “is the next <code>-</code> binary or unary?” - flipped by <code>is0operandToken</code> after every emit.
<code>prevIsUnaryMinus</code>	suppresses the right-side space after a <code>-</code> that was ruled unary, so <code>-\$x</code> stays tight.
<code>insideForHeader</code>	a small backward scan that lets the two <code>;</code> s inside <code>for (...; ...; ...)</code> 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 <code>include</code> could not be spliced
L004	invalid-directive	error	a malformed or unknown-ID <code># lint-disable</code> comment
L101	unused-local	warning	a local <code>def</code> binding never read (skips spawn-body declarations)
L102	dead-code-after-terminator	warning	a statement after <code>return / throw / exit / break / continue</code>
L103	empty-catch	warning	a <code>catch</code> block with no body
L104	throw-non-error	warning	a <code>throw</code> whose value isn't statically an <code>Error</code>
L105	constant-condition	warning	<code>if (true), while (true)</code> with no escape, <code>if (\$x == \$x), ...</code>
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)
L301	deprecation	warning	reserved family, empty until an API is deprecated

ID	Check	Severity	Flags
L302	removed-api	warning	use of a removed API (e.g. <code>use core;</code>)

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/L201/L202/L105), 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.

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 <i>in this statement</i> , excluding nested statements it called.
CUM	Cumulative wall-clock: this statement plus everything it called.
POSITION	The <code>file:line:col</code> of the statement. Rows are sorted by <code>SELF</code> 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.

Modes and formats

Flag	Effect
<i>(default)</i>	Statement profile: hit counts + self / cumulative time per position.
<code>--allocs</code>	Value-semantics profile instead: where compound values are copied (see below).
<code>--format=table</code>	Human-readable text (default).
<code>--format=pprof</code>	Gzipped protobuf, hand-encoded to keep the zero-dependency stance; <code>go tool pprof</code> and <code>speedscope.app</code> read it.
<code>--format=trace</code>	Chrome-trace JSON of the method-call timeline (open in <code>chrome://tracing</code> / Perfetto).

Unknown `--format` and the unsupported `--allocs --format=trace` combination (allocation events have no timeline) are rejected at argument parse, not deferred to output.

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 <code>def</code> / assignment / parameter binding that deep-copies a compound value (<code>Value.Copy()</code>). 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 <code>spawn</code> launches (<code>snapshotForSpawn</code>).

`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 assertEquals: 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
<code>--filter=REGEX</code>	Run only tests whose name matches <code>REGEX</code> (e.g. <code>--filter=testGreeting</code>).
<code>--format=text</code>	Human-readable, one line per test (default).
<code>--format=tap</code>	TAP version 14 - CI-friendly, with a YAML diagnostic block per failure.
<code>--format=junit</code>	JUnit XML - for CI dashboards that consume it.
<code>--isolated</code>	Run each test in a fresh interpreter subprocess: clean state per test, coarser reporting (a pass/fail line, no in-process kind / position).
<code>--coverage[=FMT]</code>	Also report statement coverage: which executable positions in the tested file(s) ran. <code>text</code> (default) prints a per-file and total percentage plus the never-executed positions; <code>json</code> emits a machine-readable form. Runs in-process (so it overrides <code>--isolated</code>).

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: assertEquals: 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
<code>testing.assertEqual(a, b)</code>	<code>a</code> equals <code>b</code> (value and kind).
<code>testing.assertNotEqual(a, b)</code>	<code>a</code> differs from <code>b</code> .
<code>testing.assertTrue(cond)</code>	<code>cond</code> (a <code>bool</code>) is <code>true</code> .
<code>testing.assertFalse(cond)</code>	<code>cond</code> is <code>false</code> .
<code>testing.assertContains(container, x)</code>	<code>x</code> occurs in the string / list <code>container</code> .
<code>testing.assertThrows(name, kind)</code>	calling the zero-arg method <code>name</code> throws an <code>Error</code> 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
<code>internal/lexer</code>	Token-by-token output for fixed inputs; trivia (comments, blank lines); error cases
<code>internal/preproc</code>	<code>include</code> file splicing and path resolution; circular-include detection; trivia handling
<code>internal/parser</code>	AST shape via <code>Sprint</code> ; operator precedence; the resolver's scope / slot pass with shadowing + undefined-variable errors; constant folding; parse error cases
<code>internal/interpreter</code>	Full programs in-memory with stdout captured; value-semantics aliasing (eager deep copies) under <code>-race</code> ; <code>CallByName</code> / <code>CallByNameWith</code> dispatch and <code>RaiseError</code> classification
<code>internal/lib/crc</code>	CRC-32 / CRC-64 checksums over <code>bytes</code> ; codec-table lookups and aliases
<code>internal/lib/encoding</code>	<code>toText</code> / <code>fromText</code> (hex, base64); charset <code>encode</code> / <code>decode</code> ; <code>isAscii</code> and length introspection
<code>internal/lib/fs</code>	Whole-file read / write / append; metadata (<code>stat</code>); directory ops; buffered <code>File</code> handles, against temp dirs
<code>internal/lib/hash</code>	MD5 / SHA-1 / SHA-256 one-shot <code>compute</code> and streaming <code>update</code> / <code>finalize</code>
<code>internal/lib/io</code>	<code>Install</code> registers <code>printf</code> / <code>sprintf</code> ; format verbs and the modifier grammar; arity and format errors
<code>internal/lib/lists</code>	<code>push</code> / <code>pop</code> / <code>sort</code> / <code>reverse</code> / <code>slice</code> / <code>concat</code> / <code>range</code> ; non-mutating (value) semantics
<code>internal/lib/maps</code>	<code>keys</code> / <code>values</code> / <code>has</code> / <code>delete</code> / <code>merge</code> ; insertion order; missing-key errors
<code>internal/lib/meta</code>	<code>meta.VERSION</code> matches <code>version.Version</code> ; <code>meta.BUILD</code> matches the compiler tag
<code>internal/lib/net</code>	TCP / UDP loopback round-trips and DNS lookups (build-tag gated; the TinyGo stub returns friendly errors)
<code>internal/lib/os</code>	<code>env</code> / <code>args</code> / flag helpers; external-process <code>run</code> / <code>spawn</code> / <code>wait</code> / <code>poll</code> / <code>kill</code>

Package	What it tests
<code>internal/lib/regex</code>	RE2 <code>matches</code> / <code>find</code> / <code>findAll</code> / <code>replace</code> / <code>split</code> ; positional + named captures; the pattern cache
<code>internal/lib/task</code>	<code>spawn</code> <code>observation</code> - <code>wait</code> / <code>poll</code> / <code>discard</code> / <code>waitAll</code> / <code>waitAny</code> - <code>under</code> - <code>race</code>
<code>internal/lib/testing</code>	Assertion vocabulary (<code>assert*</code>) throwing <code>Error{kind:"assertion"}; run</code> / <code>runWith</code> failure classification into <code>testing.Result</code> ; <code>text</code> / TAP / JUnit report rendering
<code>internal/lib/time</code>	Constructors / accessors / arithmetic round-trip; ISO weekday remapping; deterministic via a <code>nowFunc</code> package-var override
<code>internal/lint</code>	The <code>L001</code> - <code>L010</code> checks; <code># lint-disable</code> suppression; <code>--checks</code> / <code>.jennifer-lint</code> selection and unknown-ID rejection
<code>internal/profile</code>	Collector aggregation (self / cumulative, hit counts); table, Chrome-trace, and pprof rendering (gzip + string-table checks)
<code>cmd/jennifer</code>	Golden test that runs every <code>examples/*.j</code> and compares stdout to <code>examples/expected/*.txt</code> ; REPL <code>inputComplete</code> 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
<code>main.go</code>	CLI + source-context error formatting; argv forwarding to the user program.
<code>repl.go</code>	Interactive REPL loop (TTY + bufio fallback).
<code>lineedit.go</code>	Raw-mode line editor (cursor keys, word ops).
<code>history.go</code>	In-memory REPL history ring buffer.
<code>dump.go</code>	<code>tokens</code> and <code>ast</code> subcommands.
<code>astjson.go</code>	Hand-rolled AST -> JSON emitter.
<code>fmt.go</code>	Token-level source formatter; comment + blank-line preservation.
<code>lint.go</code>	<code>lint</code> subcommand: pipeline + config + suppression + JSON / human / GitHub rendering.
<code>profile.go</code>	<code>profile</code> subcommand: table / pprof / trace output.
<code>test.go</code>	<code>test</code> subcommand: <code>test*</code> discovery, <code>setUp</code> / <code>tearDown</code> , <code>--isolated</code> subprocess mode, text / TAP / JUnit reports.
<code>devtools_tinygo.go</code>	TinyGo build: run-only stubs for <code>tokens</code> / <code>ast</code> / <code>fmt</code> / <code>lint</code> / <code>profile</code> / <code>test</code> .

CLI tests

File	Description
<code>examples_test.go</code>	Golden-file integration test over top-level <code>examples/*.j</code> (skips files without an <code>expected/</code>).
<code>module_overlay_test.go</code>	Runs every <code>modules/*_test.j</code> overlay under <code>jennifer test</code> .

File	Description
<code>repl_test.go</code>	REPL <code>inputComplete</code> unit tests.
<code>lineedit_test.go</code>	Line-editor state-machine tests.
<code>dump_test.go</code>	AST-JSON validity and token-name tests.
<code>fmt_test.go</code>	Formatter idempotence + behaviour + comment-preservation tests.
<code>cross_file_error_test.go</code>	Cross-file error reporting tests.
<code>highlight_test.go</code>	Docs-site highlight-def generation test.
<code>smtp_send_test.go</code> / <code>pop_recv_test.go</code> / <code>imap_recv_test.go</code> / <code>mail_xoauth2_test.go</code>	Mail-module integration tests: <code>smtp</code> / <code>pop</code> / <code>imap</code> / XOAUTH2 against in-process fake servers.
<code>redis_test.go</code> / <code>resque_test.go</code>	<code>redis</code> / <code>resque</code> integration tests against an in-process RESP server.
<code>memcache_test.go</code> / <code>session_test.go</code> / <code>ratelimit_test.go</code>	<code>memcache</code> and its reference modules against an in-process memcached fake.
<code>http_test.go</code> / <code>gotify_test.go</code> / <code>rest_test.go</code> / <code>oauth_test.go</code>	<code>http</code> client + <code>gotify</code> / <code>rest</code> / <code>oauth</code> against an in-process <code>net/http</code> server.

Lexer, preprocessor, module resolution (internal/)

File	Description
<code>lexer/token.go</code>	Token type definitions (incl. trivia tokens).
<code>lexer/lexer.go</code>	Scanner with optional file tagging; nested block comments.
<code>lexer/lexer_test.go</code>	Lexer tests.
<code>preproc/preproc.go</code>	File-import preprocessor; trivia stripping.
<code>preproc/preproc_test.go</code>	Preprocessor tests.
<code>module/resolve.go</code>	Module path classification + resolution.
<code>module/sysmoddir.go</code>	System module dir: CLI / env / compile precedence + validation.
<code>module/resolve_test.go</code>	Resolver + <code>sysmoddir</code> tests.

Parser (internal/parser/)

File	Description
<code>ast.go</code>	AST node types + <code>Sprint</code> (incl. namespaced struct types).
<code>parser.go</code>	Recursive-descent parser; namespaced struct literals.
<code>resolver.go</code>	Parse-time scope / slot resolution; undefined-variable + shadowing errors.
<code>fold.go</code>	Parse-time constant folding of literal-only subtrees.
<code>parser_test.go</code> / <code>fold_test.go</code>	Parser and constant-folding tests (incl. “runtime error stays runtime”).

Interpreter (internal/interpreter/)

File	Description
<code>value.go</code>	Runtime <code>Value</code> tagged union (incl. struct + namespaced-struct + <code>KindObject</code> tags).
<code>environment.go</code>	Scoped symbol table (name map + slot slice).
<code>interpreter.go</code>	Tree-walking evaluator; namespaced-struct + object registration and resolution.
<code>interpreter_test.go</code>	End-to-end interpreter tests.
<code>interactive_test.go</code>	REPL <code>EvalInteractive</code> tests.
<code>namespace_test.go</code>	Namespaced builtins + constants tests.
<code>namespaced_struct_test.go</code>	Namespaced struct tests (synthetic <code>widgets</code> lib).
<code>structs_test.go</code>	User-defined struct tests.
<code>trycatch_test.go</code>	<code>try</code> / <code>catch</code> / <code>throw</code> tests.
<code>spawn_test.go</code>	<code>spawn</code> / <code>task of T</code> runtime + registry tests.
<code>control_flow_test.go</code>	<code>break</code> / <code>continue</code> / <code>repeat</code> / <code>exit</code> tests.
<code>collections_test.go</code>	Lists, maps, iteration tests.
<code>compound_test.go</code>	Nested list-of-list / map-of-list / chained-index tests.
<code>bytes_test.go</code>	Bytes type + non-decimal literals + bit-op tests.
<code>callbyname_test.go</code>	<code>CallByName</code> / <code>CallByNameWith</code> dispatch + <code>RaiseError</code> classification tests.
<code>value_alias_test.go</code>	Shared-marker COW alias-stress tests.

File	Description
<code>collection_typing_test.go</code>	Generic collection validated entry-by-entry against the declared element type at def-init + assign.

Standard libraries (`internal/lib/`)

File	Description
<code>io/iolib.go</code> , <code>io/format.go</code> , <code>io/input.go</code>	<code>io</code> : entryptoint; <code>printf</code> / <code>sprintf</code> + verb-modifier mini-language; <code>readLine</code> / <code>readBytes</code> / <code>readChars</code> / <code>eof</code> .
<code>convert/convert.go</code>	<code>convert</code> : <code>toInt</code> / <code>toFloat</code> / <code>toString</code> / <code>toBool</code> / <code>typeOf</code> / <code>objectType</code> + UTF-8 codecs + codepoint bridges.
<code>math/mathlib.go</code>	<code>math</code> : <code>abs</code> / <code>min</code> / <code>max</code> / <code>sqrt</code> / <code>pow</code> / <code>floor</code> / <code>ceil</code> / <code>round</code> / <code>rand*</code> ; <code>PI</code> / <code>E</code> .
<code>strings/stringslib.go</code>	<code>strings</code> : case, predicates, slicing, <code>split</code> / <code>chars</code> / <code>join</code> .
<code>lists/listslib.go</code>	<code>lists</code> : <code>push</code> / <code>pop</code> / <code>first</code> / <code>last</code> / <code>head</code> / <code>tail</code> / <code>reverse</code> / <code>sort</code> / <code>contains</code> / <code>concat</code> / <code>slice</code> / <code>shuffle</code> / <code>range</code> .
<code>maps/mapslib.go</code>	<code>maps</code> : <code>keys</code> / <code>values</code> / <code>has</code> / <code>delete</code> / <code>merge</code> .
<code>os/oslib.go</code> , <code>os/exec.go</code>	<code>os</code> : <code>PLATFORM</code> / <code>ARCH</code> / <code>EOL</code> / <code>DIRSEP</code> / <code>PATHSEP</code> / <code>ARGS</code> + <code>getEnv</code> / <code>hasFlag</code> / <code>flag</code> / <code>isTerminal</code> ; external-program execution (<code>run</code> / <code>spawn</code> / <code>wait</code> / <code>poll</code> / <code>kill</code>).
<code>meta/metallib.go</code>	<code>meta</code> : <code>VERSION</code> / <code>BUILD</code> / <code>SYSMODDIR</code> (interpreter self-identity).
<code>time/timelib.go</code> , <code>time/zone.go</code> , <code>time/format.go</code>	<code>time</code> : <code>Time</code> / <code>Duration</code> , Unix + calendar + arithmetic; fixed-offset <code>Zone</code> , <code>UTC</code> / <code>local</code> / <code>inZone</code> ; <code>strftime</code> <code>format</code> / <code>parse</code> + ISO round-trip.
<code>hash/hashlib.go</code>	<code>hash</code> : MD5 / SHA-1 / SHA-256 one-shot + streaming via a codec table.
<code>crc/crclib.go</code>	<code>crc</code> : CRC-32 / CRC-64 (big-endian) one-shot + streaming.
<code>compress/compresslib.go</code>	<code>compress</code> : <code>pack</code> / <code>unpack</code> (<code>gzip</code> / <code>zlib</code> / <code>deflate</code>) + streaming (<code>compress.Stream</code>); optional level.

File	Description
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).
*/*_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).

File	Description
<code>internal/lint/lint.go</code>	<code>lint</code> checks: <code>Diagnostic</code> + grouped-ID registry (L0nn source / L1nn correctness / L2nn style / L3nn lifecycle) + <code>KnownIDs</code> / <code>selectable</code> / <code>Catalog</code> .
<code>internal/lint/run.go</code>	Check driver + threshold <code>Config</code> + source-error diagnostic builder.
<code>internal/lint/checks.go</code>	The individual check implementations.
<code>internal/lint/scope.go</code>	Scope-aware traversal for binding-visibility checks (L101 unused-local, L104 throw-non-error).
<code>internal/lint/walk.go</code>	Flat AST walker for node-shape checks (L102 / L103 / L105 / L201 / L202).
<code>internal/lint/config.go</code>	<code>--checks</code> / <code>.jennifer-lint</code> selection parsing (selectable IDs only).
<code>internal/lint/suppress.go</code>	<code># lint-disable</code> directive parsing + application; a malformed / unknown-ID directive becomes an L004 finding.
<code>internal/profile/profile.go</code>	<code>profile</code> collector: per-position hit counts + wall-clock (+ optional alloc counting).
<code>internal/profile/render.go</code>	Table + Chrome-trace renderers.
<code>internal/profile/pprof.go</code>	Gzipped-pprof output (hand-encoded, zero-dependency).

Version & build

File	Description
<code>internal/version/version.go</code>	Default <code>Version</code> = "dev".
<code>internal/version/version_gen.go</code>	GENERATED by <code>scripts/gen-version.sh</code> (gitignored).
<code>scripts/version.sh</code>	Computes the version string from git state.
<code>scripts/gen-version.sh</code>	Writes <code>version_gen.go</code> before each build.
<code>Makefile</code>	<code>build</code> (both binaries) / <code>build-tinygo</code> / <code>build-go</code> / <code>test</code> / <code>clean</code> / <code>version</code> .

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
<code>ansi.j</code>	Terminal styling as string wrappers (color / style / rgb / strip); TTY-aware.
<code>csv.j</code>	RFC 4180 CSV parse / format (+ any delimiter) + header-keyed records.
<code>semver.j</code>	Strict SemVer 2.0.0 parse / compare / sort / increment.
<code>htmlwriter.j</code>	Build an HTML element tree and render escaped HTML5.
<code>markdown.j</code>	CommonMark subset -> HTML / ANSI, plus Markdown authoring helpers.
<code>mime.j</code>	Build + parse MIME messages (RFC 5322 / 2045, RFC 2047 encoded-words).
<code>smtp.j</code> / <code>pop.j</code> / <code>imap.j</code>	Mail clients over <code>net</code> : send (SMTP), receive (POP3 / IMAP4rev1).
<code>sasl.j</code>	SASL auth encoders (<code>plain</code> / <code>login</code> / <code>bearer</code> XOAUTH2), pure base64.
<code>idna.j</code>	Internationalized domain names (Punycode <code>toAscii</code> / <code>toUnicode</code>).
<code>redis.j</code>	Redis RESP2 client over <code>net</code> .
<code>resque.j</code>	Resque-wire-compatible background jobs (on <code>redis</code>).
<code>memcache.j</code>	memcached text-protocol client over <code>net</code> .
<code>session.j</code>	Server-side sessions (on <code>memcache</code> + <code>uuid</code> + <code>json</code>).
<code>ratelimit.j</code>	Fixed-window rate limiter (on <code>memcache</code>).
<code>http.j</code>	HTTP/1.1 client over <code>net</code> (<code>https</code> via TLS).
<code>gotify.j</code>	Gotify push notifications (on <code>http</code>).
<code>rest.j</code>	Ergonomic REST layer (on <code>http</code> + <code>json</code>).
<code>oauth.j</code>	OAuth2 client - get-a-token grants (on <code>http</code> + <code>json</code> ; feeds <code>sasl</code>).

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=2mb` 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 <code>jennifer-tiny</code>
<code>os</code>	<code>os.run</code> , <code>os.spawn</code> , <code>os.wait</code> , <code>os.poll</code> , <code>os.kill</code>	Runtime error pointing at the default <code>jennifer</code> binary. The <code>os/exec</code> subprocess surface: unimplemented in TinyGo on host targets, and absent by nature on embedded / WASM. Not the same “recompile” story as <code>net</code> - see the note below.
<code>net</code>	Every entry point (TCP, UDP, DNS)	Runtime error pointing at the default <code>jennifer</code> binary. Our stock <code>jennifer-tiny</code> registers no netdev driver, so <code>net</code> is stubbed. Build-tag split: <code>netlib_tinygo.go</code> returns friendly errors. Not a hard TinyGo limit - see the note below.

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 other shipped library (`io`, `convert`, `math`, `strings`, `lists`, `maps`, `meta`, `time`, `hash`, `crc`, `encoding`, `task`, `fs`, `regex`, `testing`) has full TinyGo support.

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=2mb` 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.

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=2mb` 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 1.26.3, TinyGo 0.41.1, unstripped). The absolute numbers move with toolchain version and platform; the ratio is the stable part.

Binary	Size
<code>jennifer</code>	~7.0 MB (7,250,750 bytes)
<code>jennifer-tiny</code>	~4.5 MB (4,621,624 bytes)

`jennifer-tiny` comes in at ~64% of the default binary (about a third smaller). Most of that gap is TinyGo's smaller runtime versus the standard Go runtime; the run-only trim (excluding the `tokens` / `ast` / `fmt` / `lint` / `profile` / `test` development subcommands) shaves an incremental slice on top.

These are unstripped `make build` (dev) sizes. Release builds strip: the Go binary adds `-trimpath -ldflags "-s -w"` (down to ~5.0 MB, a third smaller) and the TinyGo binary adds `-no-debug` (down to ~1.8 MB, a ~60% cut). Shipped artifacts are therefore well under the dev numbers above; dev builds keep symbols for debugging.

Single-binary benchmark results

Reference numbers from `examples/benchmark.j` (version `0.16.0-dev+7.de20da4`) on an **AMD Ryzen 5 7600X3D** (6 cores, 12 threads; desktop active, load low) - the machine the suite now prints in its own header, from `os.NCPU` plus a `/proc/cpuinfo` read done in Jennifer. 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), so append-in-a-loop is amortised O(N) with in-place growth.

jennifer (standard-Go binary, default)

```
=== Jennifer benchmark suite ===
build:    go
version:  0.16.0-dev+7.de20da4
cpu:      AMD Ryzen 5 7600X3D 6-Core Processor (12 cores)

-----
Workload                                base      iters      time_ms
-----
fib(N) recursive                        23         1         74
primes up to LIMIT                     100000      1       16407
newton sqrt batch                       10000     10000        298
monte carlo pi                         500000    500000        839
list sort/reverse/slice                 10000        500       1449
struct list build+read                   10000     10000         30
string join                             10000     10000         13
map insert+read                         10000     10000       1206
-----
total                                   20316

Parallel comparison (workers = 4, scheduler = go)
-----
Workload                                serial_ms  par_ms      speedup
-----
primes up to LIMIT                     16407     6298        2.61
newton sqrt batch                       298        78         3.82
monte carlo pi                         839        290         2.89
fib(N) x workers                       296        95         3.12
-----

user    40.96s
sys      0.44s
real    27.08s      (152% CPU)
```

`user + sys` (41.4s) exceeds `real` (27.1s) by ~14.3s - that gap is Go's concurrent GC running on a second CPU during the serial section, plus the four spawn workers running in parallel during

the parallel section. Sys time is tiny (0.4s) because Go's runtime coordinates goroutines with cheap in-process sync primitives.

jennifer-tiny (TinyGo binary)

```
=== Jennifer benchmark suite ===
build:   tinygo
version: 0.16.0-dev+7.de20da4
cpu:     AMD Ryzen 5 7600X3D 6-Core Processor (1 cores)
```

Workload	base	iters	time_ms
fib(N) recursive	23	1	76
primes up to LIMIT	100000	1	13287
newton sqrt batch	10000	10000	242
monte carlo pi	500000	500000	896
list sort/reverse/slice	10000	500	1275
struct list build+read	10000	10000	34
string join	10000	10000	21
map insert+read	10000	10000	1775
total			17606

Parallel comparison (workers = 4, scheduler = tinygo)

Workload	serial_ms	par_ms	speedup
primes up to LIMIT	13287	13181	1.01
newton sqrt batch	242	243	1.00
monte carlo pi	896	821	1.09
fib(N) x workers	304	260	1.17

```
user   31.93s
sys     0.03s
real   32.12s    (99% CPU)
```

This is the pinned build (`-scheduler=tasks -stack-size=2mb`): the whole suite completes, serial and parallel. `os.NCPU` reports `1` here - honest about the cooperative single-thread scheduler's usable parallelism, not the 12 threads the machine has. The parallel column hovers at ~1.0 by design - `spawn` under that scheduler is concurrency, not multi-core throughput - and `user ~= real` at 99% CPU confirms the single-thread execution (contrast Go's 152% above). The 2MB stack (up from 1MB) is what lets the serial recursive `fib` run at all: at 1MB it fit bare but overflowed nested one call frame deeper, inside `benchFib`.

Per-workload comparison (serial section)

Ratios are `tiny_ms / go_ms`; > 1.0 means `jennifer-tiny` is slower, < 1.0 means it is faster.

Workload	tiny (ms)	go (ms)	Ratio	Where the time goes
<code>fib(N) recursive</code>	76	74	1.0x	Tight interpreter dispatch loop; effectively tied.
<code>primes up to LIMIT</code>	13287	16407	0.8x	Long numeric dispatch loop; TinyGo pulls ahead at scale - and this row carries the whole aggregate lead.
<code>newton sqrt batch</code>	242	298	0.8x	Float arithmetic + dispatch; TinyGo ahead.
<code>monte carlo pi</code>	896	839	1.1x	Float arithmetic + RNG calls; now marginally Go's.
<code>list</code> <code>sort/reverse/slice</code>	1275	1449	0.9x	Allocation-heavy; TinyGo's simpler GC beats Go's concurrent GC at this scale.
<code>struct list</code> <code>build+read</code>	34	30	1.1x	Append hot loop is O(1). Both binaries are effectively free (sub-40ms).
<code>string join</code>	21	13	1.6x	Build-up-a-string pattern is O(1); both free in absolute terms (sub-30ms), Go's runtime a step ahead.
<code>map insert+read</code>	1775	1206	1.5x	Go's runtime map implementation outperforms TinyGo's at this churn rate.
total	17606	20316	0.9x	TinyGo still posts the lower serial total, but the entire margin is the <code>primes</code> row.

The two binaries have **converged** from the earlier build: TinyGo still posts the lower serial total (17.6s vs 20.3s, ~13% faster), but the gap has closed from the old ~32% - Go's serial total fell (~23.6s -> 20.3s) while TinyGo's rose slightly (~16.0s -> 17.6s), a normal cross-build drift as the interpreter grew.

More telling: TinyGo's entire aggregate lead is **one workload**. The `primes` row alone is 3120 ms in TinyGo's favour (13287 vs 16407), which is *larger* than the whole 2710 ms total gap - so on

the sum of every other workload Go is actually ~410 ms faster. TinyGo wins the long tight numeric loops (`primes` , `newton`) and stays ahead on allocation-heavy `list` ; Go wins the stdlib-churn workloads (`map insert+read` , `string join`), the tiny structural rows, and now edges `monte carlo` . The “compute-bound favours TinyGo” rule of thumb is now really “the longest dispatch loop favours TinyGo, and it’s big enough to tip the total.”

Parallel section

Speedup is `serial_ms / par_ms` ; > 1.0 means the four-worker version beat serial. Both columns are the pinned build now (no more crashed `fib` row): 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
<code>primes up to LIMIT</code>	16407	6298	2.61	13287	13181	1.01
<code>newton sqrt batch</code>	298	78	3.82	242	243	1.00
<code>monte carlo pi</code>	839	290	2.89	896	821	1.09
<code>fib(N) x workers</code>	296	95	3.12	304	260	1.17

Go reaches real multi-core speedup (2.61x-3.82x on four workers). `jennifer-tiny` pins the cooperative scheduler, so `spawn` there is concurrency without multi-core throughput (~1.0 by design); use the default binary when parallel throughput matters.

This is where the serial-total lead reverses. TinyGo has the lower *serial* total (17.6s vs 20.3s), but Go finishes the *whole suite* in less wall-clock time: **real is 27.1s for Go vs 32.1s for TinyGo.** The parallel section is why - Go crunches it in ~6.8s (four workers, real speedup) where TinyGo takes ~14.5s (no parallelism), and that ~7.7s swing more than erases TinyGo’s ~2.7s serial edge. Lower single-thread compute time does not mean a faster end-to-end run once any `spawn` parallelism is in play.

Picking a binary, in short: TinyGo leads single-thread dispatch on the long numeric loops and posts the lower serial total, but that lead lives almost entirely in `primes` ; both are essentially free on the small structural workloads; Go leads on string/map churn, wins the end-to-end wall clock whenever `spawn` parallelism is involved, and is the only choice for real multi-core throughput; TinyGo trades all of this for a larger resident-memory footprint (below).

Memory and page faults

Same machine, `/bin/time` (GNU time) on an equivalent run (per-workload timings within noise of the tables above):

Metric	jennifer (Go)	jennifer-tiny (TinyGo)
peak resident (RSS)	~35 MB	~68 MB
minor page faults	~213,000	~13,100
CPU	152%	99%

The two runtimes trade opposite resources. **TinyGo uses ~2x the peak RSS** - its cooperative scheduler reserves each goroutine's full `-stack-size` up front (the four parallel `spawn` workers each hold a 2MB stack, ~8MB before the interpreter's own data), where Go grows goroutine stacks on demand from ~8KB. Bumping the stack from 1MB to 2MB (to fit the recursive `fib` body) doubled that reservation, and it shows here. **Go, in turn, churns ~16x the page faults** (~213k vs ~13k): its concurrent GC allocates and reclaims pages aggressively across cores - the same activity behind its 152% CPU and the `user >> real` gap. TinyGo's simpler GC touches far fewer pages and runs single-threaded at 99% CPU. So TinyGo buys lower single-thread compute and low GC churn with a larger, flatter memory footprint; Go buys a smaller footprint and real multi-core parallelism with heavy GC activity.

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).

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.

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.

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 > M15.0](#).

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` / `fprintf`) 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 F00 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 i18n)

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 `i18n`.

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 `i18n`, 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(i18n.tr("Hello, %s"), name)`. Translation and formatting already compose cleanly, so there is no gap for a `printf` extension to fill.

So translation stays `i18n.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.

Jennifer - Milestones

Development is split into milestones. Each milestone produces a *working* interpreter that runs a strictly larger subset of the language.

M1 - End-to-end MVP

Status: done.

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

Status: done.

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

Status: done.

- `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

Status: done.

- 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

Status: done.

- **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

Status: done.

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

Status: done.

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

Status: done.

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

Status: done.

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`.

The next phase splits into four arcs after two architectural prerequisites: M10 lands the namespace-first library architecture that the rest of the standard library will be built on; Phase

A (M11-M13) finishes the language so libraries have something to stand on; M14 closes the lexer-side gap (`fmt` losing comments and shebangs) so the first wave of struct-using libraries can ship with doc-comments intact; Phase B (M15.x) ships the foundational libraries that every Jennifer program needs, finishing with **M15.8 - the first public release** (CI, prebuilt binaries, .deb / pacman / AUR packaging); Phase C (M16.x) ships I/O libraries on top of the now-released foundation; Phase D (M17-M20) ships the higher-level ecosystem (Jennifer-coded libraries, the module system that unblocks them, crypto, a server). Everything beyond a 1.0.0 - embedding, WASM, and specialised domains - lives in the [beyond-1.0.0 idea collection](#).

The library milestones use sub-numbering (M15.1, M15.2, ...) so each library ships and is reviewed independently. This is the first time we use sub-milestones; the practice is justified because each library is small enough to land in a single sitting once the language foundation is in place.

M10 - Namespace-first library architecture

Status: done.

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 `used`):
 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.

Phase A: language completion (M11-M13). These three milestones close the biggest daily-use gaps and add the foundational types every later library needs.

M11 - Control-flow completion

Status: done.

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

Status: done.

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 - Structs and catchable errors

Status: done.

The composite-data milestone, batched into two sub-milestones in dependency order: M13.1 ships the struct mechanism, M13.2 ships the error-handling design that uses it. Together they

unblock every library that wants composite returns and give the language a recoverable-error story.

M13.1 - Structs / records

Status: done.

- `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;` (no init) zero-fills, recursing through 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;`); index-assign and field-assign share one walker.
- Value semantics like lists/maps; `const` deep at any depth.
- Strict at boundaries: unknown struct type, missing / unknown field at literal, field-type mismatch on write, field access on a non-struct value are all positioned errors.

See:

- [user-guide/types-and-values.md](#) - language angle.
- [technical/interpreter.md](#) - runtime details (`KindStruct`, hoisting, unified `Lvalue` walker).
- [technical/grammar.md](#) - `structDef`, `structLit`, `fieldAssign`, mixed-tail `Lvalues`.
- `examples/structs.j` standalone; `examples/showcase.j` `=== M13.1 structs ===` section.

M13.2 - try / catch / throw

Status: done.

Catchable errors. New keywords `try`, `catch`, `throw`. Depends on M13.1 because the canonical error value is a struct.

- `try { body } catch (NAME) { handler }` runs the body and, on a catchable error, binds the thrown value to `$NAME` in a fresh per-handler scope.
- `throw EXPR;` raises any value; convention is the auto-hoisted `Error{kind, message, file, line, col}` struct.
- Runtime errors (out-of-bounds, missing key, type mismatch, etc.) are wrapped into the canonical `Error` shape on entry to the catch (`kind` defaults to `"runtime"` until sites opt

in to specific tags); user code catches both kinds uniformly. `throw $err;` inside a catch re-raises to the enclosing `try`.

- **Not** catchable: `exit` (program-level escape, propagates through `try`); `return` / `break` / `continue` (control flow, flow through `try` unchanged).
- No `finally` and no typed catch in v1.
- Internals: `ErrorSignal` sentinel parallels `ExitSignal`; `runtimeError.Kind` field threads the symbolic tag; user code may not redefine the auto-hoisted `Error` struct.

See:

- [user-guide/control-flow.md](#) - language angle.
 - [technical/interpreter.md](#) - runtime details (`ErrorSignal` , wrapping, flow passthrough).
 - [technical/grammar.md](#) - `tryStmt` , `throwStmt` .
 - `examples/trycatch.j` standalone; `examples/showcase.j` `=== M13.2 try/catch ===` section.
-

M14 - Lexer comment + blank-line preservation

Status: done.

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.
-

Phase B: foundational libraries (M15.x). Small, frequently-used libraries grouped under M15 with sub-numbering. The leading M15.0 slot is the “wrap-up of existing libraries” (extensions to M8 / M9 / M10 libraries that depend on language features added since); later slots ship a new library each. M15.8 closes the phase by making the result installable before Phase C starts adding I/O on top.

M15 - foundational libraries + first public release

Status: done. All nine sub-milestones shipped. Three are language work (M15.2, M15.4), the rest are library / tooling / release work. Two recurring patterns surfaced in the shipped APIs and are worth remembering:

- **Codec-table shape** (algorithm/format/codec passed as a string argument). Used by `hash.compute(b, algo)`, `crc.compute(b, algo)`, `encoding.encode(s, codec)`, `encoding.toText(b, format)`. Originally adopted because Jennifer's letters-only identifier rule rejects digits in method names (so `hash.md5(...)` won't parse), but it also honours stance #1 by collapsing parallel verbs into one.
- **Integer-handle struct for opaque resources** (M15.2's namespaced-struct mechanism + a single `id as int` field indexing into a Go-side map). Used by `os.Process`, `hash.Stream`, `crc.Stream`.

M15.0 - existing-library extensions

Done. Two extensions to the M9 `lists` library that needed post-M14 language features: `lists.shuffle(xs)` (Fisher-Yates, respects `math.randSeed`) and `lists.range(start, end[, step])` (half-open, deliberate single-arg omission per stance #2). See [libraries/lists.md](#) and [technical/design-decisions.md > Half-open ranges](#) for the half-vs-closed-range rationale.

M15.1 - os + meta (process metadata)

Done. Reshapes the M8-era `os` surface around one rule: immutable per-run host facts are uppercase constants (`PLATFORM`, `ARCH`, `EOL`, `DIRSEP`, `PATHSEP`, `ARGS`), operations are functions (`getEnv`, `hasFlag`, `flag`). Drops the `JENNIFER_` prefix that only made sense for bare-global use, and introduces a new `meta` library for interpreter-self-identity constants

(`VERSION`, `BUILD`). CLI forwards trailing args to `os.ARGS` (script path at index 0). Breaking renames (`JENNIFER_VERSION` -> `meta.VERSION`, `os.platform()` -> `os.PLATFORM`, `os.JENNIFER_OS` -> `os.PLATFORM`, `os.JENNIFER_LF` -> `os.EOL`); old names now produce plain “undefined” errors with no rename-hint. See [libraries/os.md](#) and [libraries/meta.md](#).

M15.2 - Language: library-provided namespaced struct types

Done. Language work slotted inside Phase B because the next wave of libraries (M15.3 `os`, `{Result, Process}`, M15.5 `time.*`, M15.6 `hash.Stream / crc.Stream`, future M16.1 `fs`, M16.2 `net`) all need their own struct types and M13.1 only handled bare-IDENT names. Adds `def x as lib.Name`; type syntax, `lib.Name{field: ...}` literals, and the Go-side `Interpreter.RegisterNamespacedStruct` API. Reuses M13.1’s value semantics, deep-`const`, and strict-boundary machinery; only the resolution path differs. User code can’t register structs (Go-side only); methods on structs and inheritance stay out of scope. See [technical/interpreter.md > Library-provided namespaced structs](#).

M15.3 - os external-program execution

Done. First library to consume the M15.2 namespaced-struct mechanism. Surface: `os.Result {exitCode, stdout, stderr} + os.Process {pid}` as the public types; `os.run(argv) -> Result` blocking, `os.spawn(argv) -> Process` non-blocking, `os.wait/poll/kill(p)` for handle ops. `argv` is always `list of string` (no shell parsing; explicit `["sh", "-c", $cmd]` for that hop). Non-zero exit codes are values, not errors. **TinyGo limitation:** TinyGo’s runtime doesn’t implement `os/exec`, so the constrained `jennifer-tiny` binary returns a friendly “use the default `jennifer` binary” error instead of panicking - first place the two-binary story becomes user-visible. See [libraries/os.md > External programs](#) and `examples/exec.j`.

M15.4 - Language: len built-in, core removed

Done. Promoted `len(EXPR)` from the auto-loaded `core` library to a reserved keyword + language primary expression (polymorphic over `string / list / map / bytes`). Deleted `internal/lib/core/` entirely; `use core`; now returns a friendly migration error pointing at the built-in and at `meta.VERSION / meta.BUILD`. Stance #2 (“explicit over implicit”) now applies uniformly: every library name lives behind a `use NAME` prefix, no exceptions. See [technical/design-decisions.md > len is a language built-in](#).

M15.5 - time

Done. One opt-in library spanning instants, durations, fixed-offset zones, strftime format/parse, and ISO 8601 round-trip. Three namespaced structs: `time.Time {nanos, offset}` (fields private API), `time.Duration {nanos}`, and `time.Zone {offset, name}` (fields public, so an IANA / DST companion can build them). Granularity (date-only vs time-of-day-only) is a property of formatting, not the value type. Unix timestamps are constructor / accessor pairs, not a separate type. IANA names and DST are out of the fixed-offset core - a Go-backed `time`-library extension, not a `.j` data map (see the Long-horizon “`time`: IANA / DST zones” entry). Three sub-milestones: **M15.5.1** core type + Unix + calendar + 1-based ISO weekday + arithmetic / comparison; **M15.5.2** strftime format/parse (chosen over Go’s reference-time style for familiarity; v1 verbs `%Y %m %d %H %M %S %z %a %A %b %B %j %u %%`) + `time.zone(offset, name)` + `time.inZone` + the `time.UTC` constant coexisting with the `time.utc()` function via case-sensitive lookup + `time.iso` / `time.fromIso` RFC 3339 round-trip; **M15.5.3** the `examples/benchmark.j` side-by-side TinyGo-vs-Go workload suite (eight workloads; the original spec’s “Sieve of Eratosthenes” became trial-division because Jennifer’s value-semantic list mutation turns the sieve into $O(N^2)$). See [libraries/time.md](#), `examples/time{,-format,benchmark}.j`.

M15.6 - hash and crc

Done. Two opt-in libraries with parallel surfaces: `hash` for cryptographic-style digests (`"md5"` , `"sha1"` , `"sha256"`), `crc` for non-cryptographic checksums (`"crc32"` IEEE, `"crc64"` with Go’s `crc64.ECMA` polynomial). Output is raw `bytes` (big-endian 4 / 8 bytes for CRC, natural width for hash). The split keeps “transport integrity” vs “content addressing” visible at the import line and matches Go’s `crypto/*` vs `hash/crc*` arrangement. Both libraries ship the codec-table shape: `compute(b, algo)` one-shot, `stream(algo)` + `update($s, $b)` + `finalize($s)` -> `bytes` for chunked input. Streaming reuses the [integer-handle pattern from os.Process](#). No convenience wrappers like `hash.md5String` or `hash.computeHex` (stance #1; users compose `convert.bytesFromString` and `encoding.toText`). Struct hashing deferred (needs stable byte serialization, its own design problem). See [libraries/hash.md](#), [libraries/crc.md](#), `examples/hash.j`.

M15.7 - encoding

Done. Three-part surface: introspection (`isAscii` / `lenBytes` / `lenRunes`), binary-to-text (`toText` / `fromText` for `"hex"` / `"base64"` / `"base64-url"`), and character codecs (`encode` / `decode` / `codecs`). The cross-kind UTF-8 pair stays in `convert` (M12); `encoding` owns the table-based codec proliferation. Four codecs ship: `"ascii"` , `"iso-8859-1"` , `"windows-`

1252", "ebcdic". The spec's per-format verbs (`encoding.hex`, `encoding.base64`, ...) consolidated into the codec-table pair to dodge the same digit-in-identifier rule M15.6 hit. Codec names and format strings are exact-match (the one canonical spelling only; the original alias / case normalisation was later dropped as a strictness lift, stance #2). Windows-1252's five canonically-undefined positions (0x81, 0x8D, 0x8F, 0x90, 0x9D) reject symmetrically on encode and decode. Long-tail single-byte codecs (ISO-8859-{2..16}, Windows-{1250,1251,1253..1258}) later shipped in [M16.15](#), generated from the Unicode mapping files. See [libraries/encoding.md](#), [examples/encoding.j](#).

M15.8 - distribution + first public release

Done. Packaging / CI / release-engineering only; no `.j` -source language change. Four sub-phases:

- **CI** (`.github/workflows/test.yml`, `release.yml`). PR gate runs `go vet + gofmt + go test ./... + make build` + per-binary smoke run + repo-wide em-dash scan. Release triggers on bare-semver tags (`[0-9]*.[0-9]*.[0-9]*`, no `v` prefix per project convention), cross-compiles both binaries for `linux/{amd64,arm64}` from one runner, QEMU-smoke-tests the non-native arch, runs the benchmark on amd64 so release notes carry fresh numbers, publishes a draft Release.
- **Packaging** under `packaging/{debian,arch,mime,man}/.scripts/build-deb.sh` produces the `.deb` (binaries + gzipped man pages + shared `text/x-jennifer` MIME definition
 - `update-mime-database` hooks). AUR ships `PKGBUILD-bin` (release tarball) and `PKGBUILD-git` (source-tracking) with a shared `.install` hook. Release pipeline auto-fills `PKGBUILD-bin` (real `pkgver` + real `sha256sums_*`) as a release asset so the AUR push is a one-step `curl`. The `.pacman` standalone artefact from the original spec was dropped - `PKGBUILD-bin` + `makepkg` covers the same need.
- **Docs site** via mdBook 0.5.3 (pinned, fetched via direct curl from `rust-lang/mdBook` releases - no third-party action). `book.toml` at repo root, `src = "docs"`, `docs/SUMMARY.md` maps the existing tree into five parts, `docs/introduction.md` is the docs-site landing (README stays GitHub-repo-focused). `.github/workflows/docs.yml` publishes to GitHub Pages on every push to `main`.
- **User-facing install docs**. README gained "Which binary?" + "Install" sections with one command per path. `installing.md` restructured to put package paths first; build-from-source positioned as the developer path. `RELEASE.md` at the repo root documents the steps CI can't do (AUR SSH push, draft-publish review, pre-tag readiness checks).

Conventions established (worth keeping):

- Bare semver tags (`0.14.1`, no `v` prefix); all pipelines pass the tag straight through.

- No top-level `LICENSE` file in v1 - the LGPL-3.0 text ships inline in `packaging/debian/copyright` (the form distros actually consume) + README links to gnu.org's canonical URL.

One-time manual setup before the first push to `main`: in GitHub repo Settings -> Pages, set "Source" to "GitHub Actions" so `docs.yml` can deploy.

Deferred out of this milestone (not gated on it): the cross-build for macOS / Windows and a real apt repository stay in [Requirements for 1.0.0 stable](#); the extra distribution formats (Homebrew, Snap, Nix, Flatpak, AppImage) moved to the [horizon idea collection](#).

M16 - I/O libraries and developer tooling

Status: done. Phase C: system libraries that touch the OS or do significant compute, opened by a concurrency primitive (M16.0) that the I/O libraries build on, then a developer-tooling trio (lint / profile / test) and a run of self-contained data libraries. All sub-milestones shipped; git history holds the full per-milestone specs.

M16.0 - Lightweight concurrency

Done. `spawn { ... }` (block primary expression), `task of T` (new compound kind), and the `task library (wait / poll / discard / waitAll / waitAny)`. Goroutine-backed but data-race-free by construction: `snapshotForSpawn` deep-copies a two-frame globals+locals snapshot at launch (tasks are the one carve-out from value semantics - copies share the `TaskState` pointer). A per-run registry loud-fails unobserved task errors at exit and bumps the exit code (a non-terminating undiscarded spawn hangs at exit - the documented trade-off). `task.wait` re-raises a body error at the wait site for `try / catch`; `waitAny` is the runtime's only `reflect.Select`. The Makefile passes `-stack-size=2mb -scheduler=tasks` to TinyGo. See [concurrency.md](#), [task.md](#).

M16.1 - fs

Done. Blocking filesystem I/O composed with `spawn` (no `*Async` variants): whole-file `read / write / append` (String/Bytes), metadata (`exists / isFile / isDir / stat -> fs.Stat`), directory ops with the two-verb recursion split (`mkdir / mkdirAll`, `remove / removeAll`, plus `rename / list / walk`), and buffered `fs.File` handles (`open / readLine / ... / close`; `eof` peeks

one byte). Path- vs handle-form verbs dispatch on first-arg kind; `fs.File` shares state across copies (the handle carve-out from value semantics). See [fs.md](#).

M16.2 - net

Done. TCP (`connect` / `listen` / `accept` / `readBytes` / `writeBytes`), UDP (`listenUDP` / `sendTo` / `recvFrom`), and 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). See [net.md](#).

M16.3 - regex

Done. RE2 (Go `regexp`, linear-time) over strings: `matches` / `find` / `findAll` / `replace` / `split` / `escape`; `regex.Match{text,start,end,groups,groupsNamed}` with rune indices and a `start=-1` no-match sentinel; an implicit 128-entry LRU pattern cache. Full surface in both binaries. See [regex.md](#).

M16.4 - testing (system-library primitives)

Done. The irreducible system-side surface a `.j` test framework needs: `testing.run(name)` invokes a zero-arg user method via the new `Interpreter.CallByName`, times it, and classifies every failure mode into a `testing.Result`; `results` / `reset` manage a mutex-guarded accumulator; `report` renders text / TAP / JUnit. The one place `exit` is caught (at the Go level, so language `try` / `catch` still cannot). See [testing.md](#).

M16.5 - Interpreter performance pass

Done. Five sub-milestones, user-visible behaviour unchanged: **.1** shared-marker copy-on-write 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 plus a `Share()` scalar fast path. Numbers in [tinygo.md](#).

M16.6 - Developer tooling: linting

Done. `jennifer lint` flags compile-legal-but-suspect patterns: grouped IDs (**L0nn** source errors, always on / **L1nn** correctness / **L2nn** style / **L3nn** lifecycle), `# lint-disable[-file]:` IDs suppression, `--checks` / `.jennifer-lint` config, and human / JSON / GitHub output (source errors render in the chosen format, so a JSON pipeline stays valid); exit 0/1/2. `!tinygo`. See [cli_lint.md](#).

M16.7 - Developer tooling: profiling

Done. `jennifer profile` attributes work to `.j` source positions (what `go tool pprof` cannot): 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 goes to `stderr` so the profile owns `stdout`. `!tinygo`. See [cli_profile.md](#).

M16.8 - Testing framework consolidation

Done. An assertion vocabulary on M16.4's primitives (`assertEqual ... assertThrows`, throwing `Error{kind:"assertion"}` at the call site), `CallByNameWith` / `runWith` argument 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 Jennifer error via `interpreter.RaiseError`. See [cli_test.md](#).

M16.9 - json

Done. 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` map to objects, `bytes` to base64, numbers to `int` when integral 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. See [json.md](#).

M16.10 - uuid

Done. RFC 9562: `uuid.generate("v4")` (random) / `generate("v7")` (time-ordered), the version tag a string argument (identifiers are letters-only), plus `parse` / `isValid` / `version` and constant `NIL`. Randomness draws from `math`'s shared seedable RNG (documented non-crypto; swaps when M20.1 `crypto` lands). See [uuid.md](#).

M16.11 - compress

Done. 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` handle. Go `compress/*`, TinyGo-clean. See [compress.md](#).

M16.12 - archive

Done. `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`. See [archive.md](#).

M16.13 - os.isTerminal

Done. `os.isTerminal(stream) ("stdout" / "stderr" / "stdin") -> bool`, the gate for ANSI colour, via the character-device mode bit (`os.ModeCharDevice`) - pure `stdlib` (keeps `x/term` CLI-scoped), TinyGo-clean; an unstatable stream reports `false`. See [os.md](#).

M16.14 - net TLS

Done. `net.connectTLS(address)` (implicit TLS) and `net.startTLS(conn)` (in-place STARTTLS upgrade), both yielding the transport-agnostic `net.Conn`. Certificate verification is on by default, with a `net.TLSOptions{skipVerify as bool, caCert as bytes}` opt-out. Go `crypto/tls` on the `!tinygo` build (stubbed on `jennifer-tiny`). See [net.md](#).

M16.15 - encoding completion

Done. Filled out `encoding: toText / fromText` gained `quoted-printable`, `base32 / base32-hex`, `ascii85`, and `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. Format and codec names are exact-match (the normalisation layer was dropped as a strictness lift, stance #2). See [encoding.md](#).

M16.16 - json.Value

Done. 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` reports `"object"`, `convert.objectType` the specific `"json.Value"`. Reads and non-mutating writes share JSON Pointer (RFC 6901) addressing - `typeOf` `/ get` `/ has` `/ keys` `/ length` `/ as*` `/ isNull` and `map` `/ list` `/ set` `/ insert` `/ append` `/ remove` `/ move` (strict `/ no-vivify`, `- end-marker`), with node types in `list` `/ map` vocabulary - and a displayer hook renders a handle as its JSON. `json.decode`'s return type changed (a pre-1.0 break) and the decoder's number grammar was tightened to `json.org`. No `any` keyword (rationale in [rejected.md](#)). See [json.md](#).

Phase D: higher-level and Jennifer-coded libraries (M17-M20).

M17 - Module system for Jennifer-coded libraries

Status: done. All six sub-milestones shipped. Jennifer-coded libraries now get their own namespace, scope, and explicit `export s` via a real **module boundary** (`import "x.j" as x;`, parser + interpreter) that lives beside the textual `include "x.j"`; `splice` (preprocessor, for composing one module out of several files). The settled, cross-cutting decisions (turned-down alternatives in [rejected.md](#)): `import` is a parser + interpreter feature, not a preprocessor splice; each module is its own resolution context (own `use` set, namespace + export tables); the module top level is **declarations-only** - no mutable module 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 a name (no `public` `/ private` keyword); multi-file modules assemble via `include` behind one entry file (no directory-as-module, no cross-file re-export); modules need a filesystem (an FS-less `jennifer-tiny` host fails with the ordinary search-path error). User-facing reference: [imports.md](#) + [modules/index.md](#). M18.x builds on top.

M17.1 - Source tree and resolution

Done. Path resolution in `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 a name found in two search dirs and a not-found. The system module directory resolves `--sysmoddir > JENNIFER_SYSMODDIR >` compile-time default (surfaced as `meta.SYSMODDIR`; a named CLI / env dir that is missing or not a directory refuses to start, the

compile default is best-effort), and `-I DIR` (repeatable) appends to the search path after it. `jennifer version -v` reports the resolved layers. See [cli.md](#).

M17.2 - Import statement and loader

Done. `import "path.j" [as NAME];` is a real statement (`ModuleImportStmt`) - the preprocessor passes it through, the parser builds the node. The loader (`internal/interpreter/module.go`) runs each module in a fresh sub-interpreter sharing one `moduleReg` (run-once cache by canonical path, in-progress load stack, search path), so **run-once, 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` initializer) fail the program and are not catchable - an `import` is a declaration, not an expression, so it cannot sit in a `try` / block. `jennifer fmt / ast / tokens` round-trip an `import` line. See [imports.md](#).

M17.3 - Module scope and namespacing

Done. A module top level is **declarations-only** (`checkModuleDeclarationsOnly`: only `def const` / `def struct` / `func` / `use` / `import`; a mutable `def` or free-standing statement is a positioned load-time error - 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` (arguments evaluated in the consumer, body run against the module's globals + methods) and read `alias.CONST` from its scope. `use` non-transitivity, run-once sharing, and `-race` safety all follow from the fresh-sub-interpreter-per-module model - a module holds only immutable constants and read-only methods. See [interpreter.md](#).

M17.4 - Exports and visibility

Done. `export` (a keyword) publishes a top-level `func` / `def struct` / `def const`; unmarked names stay private (reaching `mod.helper` from outside is a positioned "not exported from module" error), and `export` in a `jennifer run` script is rejected (module vs script by entry, via the `isModule` flag). `checkReferentialClosure` rejects an exported struct field or exported function parameter typed as a *private* module struct; library / namespaced types cross freely. **Cross-module struct identity** is 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 on the way in, so `def p as mod.Point, mod.Point{...}`, field reads, and pass-back all type-check while `a.Point` and `b.Point` stay distinct. The `retag` walks values *and* the collection

type tags a `list / map` carries (`retagType`), so a `list of mod.Point` handed back into a module reads as its bare `list of Point` parameter. 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

Done. `modules/ansi.j` - terminal styling as explicit string wrappers, the first module built on the system (pure Jennifer, one `use os`; dependency across the boundary; a real end-to-end 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` (no string-literal escape for it); `strip` uses `regex`; unknown colour / style names `throw`. Stateless and TTY-aware: `enabled()` re-reads `NO_COLOR / FORCE_COLOR / os.isTerminal("stdout")` per call, so redirected output stays clean and no toggle state is stored (honours the no-mutable-state rule); degrades to always-on when `os.isTerminal` is absent. Colour is a string wrapper, so a `%s|color=` `printf` modifier is rejected in its favour ([rejected.md](#)). Demo `examples/modules/ansi_demo.j`; coverage `internal/interpreter/module_ansi_test.go` + a white-box `modules/ansi_test.j` overlay reading ansi's private tables; reference doc [modules/ansi.md](#).

M17.6 - semver module

Done. `modules/semver.j` - strict [SemVer 2.0.0](#) as a second pure-Jennifer reference module (no system dependency beyond `use strings`; `use convert`; `use regex`; , so it runs on both binaries), and the foundation the future `jvc` package manager ([Long horizon](#)) needs since `install gotify>=1.0.0` is semver comparison at its core. Exports `Version { major, minor, patch, prerelease, build }` plus `parse` (throws on invalid) / `isValid` / `toString` (round-trips `parse`); `compare` / `lt` / `eq` / `gt` (SemVer precedence: numeric core, a prerelease ranks *below* its release, prerelease fields compared numeric-below-alphanumeric, build metadata ignored); `isStable` (no prerelease and `major >= 1`; `0.y.z` is unstable by convention) / `isPrerelease`; `incMajor` / `incMinor` / `incPatch` (reset the lower parts, clear pre + build); and `sort` (own pass over `compare`, since `lists.sort` is scalar-only). Strict, not a loose parser: a looser N-segment form (`1.2.3.4`) has no defined ordering and is rejected. `parse` matches the canonical anchored SemVer RE2 pattern with named groups (`regex.find` + `groupsNamed`); the precedence comparison and sort are hand-written Jennifer - the algorithmic dogfood. `meta.VERSION` is valid strict semver, so the module parses Jennifer's own version. Range / constraint matching (`^1.2.0`, `>=1.0.0`, `~1.2.3`) is the harder parser and is deferred to (or just before) `jvc`. Building the demo surfaced and fixed a latent M17.4 boundary gap: passing a consumer-typed `list of semver.Version` back into a module `list of Version` parameter

(`semver.sort`) failed because the `retag` re-tagged the struct element *values* but not the list's element-*type* metadata - now covered by `retagType` (regression `TestModuleListOfStructCrossesBoundary`). Demo `examples/modules/semver_demo.j` ; white-box `modules/semver_test.j` overlay (12 tests); reference doc [modules/semver.md](#).

M18.x - 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.

M18.1-M18.40 - shipped modules (compactd)

All done. 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	<code>csv</code>	RFC 4180 parse / format (+ <code>*With</code> for any delimiter), header-keyed <code>toRecords</code> / <code>fromRecords</code> .
M18.2	<code>htmlwriter</code>	build an HTML element tree and render escaped HTML5 (<code>element</code> / <code>text</code> / <code>raw</code> / <code>render</code>).
M18.3	<code>markdown</code>	Markdown -> HTML.
M18.4.1/.7	<code>mime</code>	RFC 5322 / 2045 message build + parse, incl. RFC 2047 encoded-words.
M18.4.2/.4	<code>smtp</code> / <code>pop</code> / <code>imap</code>	mail send + POP3 / IMAP receive over <code>net</code> (plaintext / STARTTLS / implicit TLS).
M18.4.5/.6	<code>sasl</code> / <code>idna</code>	SASL auth encoders (incl. XOAUTH2); Punycode / IDNA domains.
M18.5	<code>redis</code>	RESP2 client over <code>net</code> .
M18.5.1	<code>resque</code>	Resque-wire-compatible background jobs on <code>redis</code> .
M18.6	<code>memcache</code>	memcached text-protocol client over <code>net</code> .
M18.6.1/.2	<code>session</code> / <code>ratelimit</code>	server-side sessions + fixed-window rate limiting on <code>memcache</code> .

M#	Module(s)	Surface
M18.7	<code>http</code>	HTTP/1.1 client over <code>net</code> (<code>https://</code> via TLS).
M18.7.1/.3	<code>gotify / rest / oauth</code>	push notifications; ergonomic REST layer; OAuth2 get-a-token - all on <code>http</code> .
M18.8	<code>toml</code> (library)	RFC TOML 1.0 encode / decode; opaque <code>toml.Value</code> , JSON-Pointer walk. TinyGo-clean.
M18.9.1	<code>httpd</code> (library)	HTTP/1.1 server engine over <code>net/http</code> ; pull-loop <code>accept / respond</code> .
M18.9.2	<code>web + jennifer serve</code>	<code>.j</code> routing framework over <code>httpd</code> (routes by handler name, <code>:param</code> , middleware, <code>web.Context</code>), dispatched by <code>meta.callMain</code> ; <code>serve</code> runs / <code>--watch</code> -reloads a program.
M18.10	<code>flatdb</code>	file-backed JSON document store over <code>json + fs</code> ; JSON-Pointer query / edit; crash-atomic save.
M18.11	<code>gpio</code>	Linux GPIO over the <code>sysfs</code> / character-device interface.
M18.12	<code>docblock</code>	the Jennifer doc-comment format + parser (<code>FileDoc tree</code> , drift diagnostics).
M18.13	<code>mqtt</code>	MQTT 3.1.1 pub / sub client over <code>net</code> .
M18.14	<code>prometheus</code>	metrics exposition (produce) + retrieval (query the HTTP API).
M18.15	<code>label</code>	industrial label printing: build / render (ZPL + cab JScript) / emit pipeline.
M18.16	<code>web cookies + sessions</code>	cookie helpers + cookie-keyed sessions on the <code>web</code> framework.
M18.17	<code>totp</code>	RFC 6238 TOTP: <code>generate / verify / uri</code> . Over <code>hash.hmac + encoding + time</code> .
M18.18	<code>webhook</code>	GitHub X-Hub-Signature-256 HMAC <code>sign / verify</code> (pure) + <code>send</code> (over <code>http</code>).
M18.19	<code>bucket</code>	S3-compatible object storage over <code>http</code> (AWS SigV4): <code>connect / get / put / delete / listObjects</code> . One module for AWS S3 + MinIO / R2 / B2.
M18.20	<code>dotenv</code>	<code>.env</code> config: <code>parse / read / load</code> (into env via <code>os.setEnv</code>). Over <code>fs + strings + os</code> .
M18.21	<code>cron</code>	parse cron expressions; <code>next(schedule, after) / matches</code> . A calculator over <code>time</code> .
M18.22	<code>log</code>	leveled structured logging (<code>debug .. error ; text / logfmt / json</code>) to <code>stdout / stderr / file / RFC 5424</code>

M#	Module(s)	Surface
		syslog.
M18.23	ical	iCalendar (RFC 5545) build + parse: a Calendar of VEVENT s, escaped + line-folded, dates through time .
M18.24	vcard	vCard (RFC 6350) contacts build + parse; shares the content-line codec (ical_vcard_shared.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).

M#	Module(s)	Surface
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 milestones that improve the interpreter or its tooling but belong to neither the Jennifer-coded modules of M18 nor the Go system libraries of M20. Numbered sub-entries land here as needs arise. M19.1-M19.5 are a correctness / performance hardening pass over the interpreter core and libraries (races, dead code, algorithmic complexity, resource bounds, module identity); M19.6 is the coverage tool; M19.7 lands the @scope/package vendored-module resolver; M19.8 is the one-time relocation to the jennifer-language org and a host-independent vanity module path. None of this work needs reflect or breaks TinyGo-cleanliness. The smallest, localized crash / correctness fixes (out-of-range httpd.respond status, truncated-toml-date-time panic, io.sprintf("%d", MinInt64) , math.randInt span overflow, floorDiv large-quotient garbage, the constant-folder's above-

2^53 comparison divergence, the missing stream-registry mutexes, the `TaskState.Observed` atomic, the numeric-conversion and read-length caps, the cross-module struct-definition lookup so `def x as alias.Struct`; zero-value construction and `$x.field = ...` writes resolve an imported module struct) land as they surface rather than waiting on a milestone; each ships with a regression test.

M19.1 - Interpreter concurrency-safety

Done. Both interpreter data races fixed, each pinned by a `-race` stress test (nested-spawn global mutation; eight spawn workers each declaring an aliased module/library struct in a loop). `snapshotForSpawn` now snapshots the launching goroutine's own root frame via `effectiveGlobal(env)` instead of 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 runs (`resolveDeclaredTypesOnce`, after `loadModuleImports`) and carry a `parser.Type.Resolved` marker, so the per-execution re-resolve in `execDefine` is a read-only no-op: a `def x as alias.Struct` reached from concurrent goroutines never re-stamps the shared AST node. Error timing is unchanged (the stamp pass is best-effort; an unresolved type still errors at execution at its original position), and the marker also fixes a latent bug where an aliased library struct re-resolved in a loop hit the "canonical is aliased" rejection.

Two data races in the interpreter core that the race detector catches and that can crash a program using nested `spawn`. Both are small, localized fixes plus a `go test -race` nested-spawn stress test.

- **Nested-spawn global snapshot race.** `snapshotForSpawn` (`internal/interpreter/interpreter.go`) always iterates `i.global.vars` - the live top-level frame - regardless of which goroutine launches the `spawn`. A `spawn` nested inside a `spawn` body snapshots on a background goroutine while the main goroutine keeps defining / assigning globals: Go's fatal "concurrent map iteration and map write" (uncatchable, takes the interpreter down). Snapshot from `effectiveGlobal(env)` (the launching goroutine's own root frame - inside a task, the outer spawn's snapshot) instead of the live `i.global`. This is also more correct: a nested spawn captures its enclosing scope, not the main goroutine's live globals.
- **Runtime AST mutation.** `resolveDeclaredStructNS` (`internal/interpreter/module.go`) writes `t.StructNS = <canonical>` into the shared AST node every time a `def x as alias.Struct`; statement executes (once per loop iteration, say). The same `def` run concurrently from a spawn body and the main goroutine is a write-write race on the AST node. Resolve declared module-struct types once, at load / resolve time, not per execution.

Acceptance. A nested-spawn stress program that mutates globals on the main goroutine while inner spawns launch runs clean under `go test -race; a def alias.Struct` in a loop body inside a spawn is race-free.

M19.2 - Value representation cleanup

Done. The inert copy-on-write machinery is gone: `Value.shared`, `Share()`, `Ensure()`, `ensureCOW`, and the per-`VarExpr`-read `Share()` call are removed; the four mutation sites now grow the binding's own backing in place, and reads return the binding value directly. Value semantics rest (as they already did) on eager deep copies at every store site, documented on the `Value` type and in `docs/technical/interpreter.md`; the write-through alternative is recorded in `docs/technical/rejected.md`. The now-dead COW-detachment reporting was stripped from the `--allocs` profiler (interface, collector, table, pprof) so it no longer advertises a section that can never fire. A fresh list / map / struct literal RHS is already private, so `execDefine` / `execAssign` skip the redundant whole-value copy (`rhsFreshLiteral`) - proven by a profiler-backed test (literal binding records zero eager copies; an aliasing `def b init $a` records one) alongside a value-independence test; `value_alias_test.go` and the full suite (incl. `-race`) stay green. `Value` shrinking stays deferred.

The copy-on-write marker protocol added for the append-in-a-loop optimization is **inert**: `Value.Share()` has a value receiver, and `Environment.Get` / `GetAt` return the binding's `Value` by value, so the `shared` flag is set on a throwaway copy and never reaches the stored binding. `Ensure()` / `ensureCOW` therefore never detach (the whole `Share` / `Ensure` machinery is dead code), and every read of a compound variable heap-allocates a fresh `*bool` that goes nowhere - pure overhead in hot loops. Correctness never depended on the protocol: it rests entirely on the eager deep copies at every binding site (`def` / assignment / parameter bind) and on builtins copying before they mutate.

- **Delete the dead protocol.** Remove `Share()` / `Ensure()` / `ensureCOW`, the `Value.shared` field, and the per-read `Share()` call, and document the eager-copy invariant that actually provides value semantics. The write-through alternative (store `*Binding` so the marker propagates and mutation sites genuinely detach) was considered and rejected - aliasing-heavy code is rare precisely because we eager-copy, so the complexity buys little; record it in `docs/technical/rejected.md`.
- **Stop double-copying literals.** `evalListLit` / `evalMapLit` `Copy()` every element of a freshly-built literal, then `execDefine` / `execAssign` eager-copy the whole result again. A fresh literal (or a call result) cannot alias a binding - only `Var` / `Index` / `Field` reads can - so the binding site can skip the copy for non-aliasing RHS shapes, removing two full deep copies per literal binding.

Shrinking `value` itself (moving the compound payload behind one pointer so the scalar case stays small) is **deferred**: a large cross-cutting churn, and the map hash index (M19.3) is the bigger algorithmic win. Revisit only if benchmarks still show `value` copying dominating after M19.3.

Acceptance. The Share / Ensure machinery and the `shared` field are gone, the alias-stress tests (`value_alias_test.go`) still pass (value semantics intact), and a compound-var read in a hot loop no longer allocates; a literal `def` binding does one deep copy, not three.

M19.3 - Runtime performance: maps and the call / loop hot path

Done. 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 and value semantics are untouched - any stale / duplicate-key / non-hashable-key case fails the stamp and falls back to the (correct) linear scan. A 100k-key build plus a 100k for-each of indexed reads runs in under a second where the quadratic path took minutes; pinned by `map_index_test.go` (order, updates, misses, duplicate and non-hashable keys, value-semantics independence, 5k-key consistency). The call/loop batch landed too: `execForEach` and `execFor` borrow their frames from `envPool` instead of allocating per iteration / per loop; `DefineAt` skips the enclosing-scope shadow walk on the resolver-verified slot path; `Run` pre-sizes `i.global`'s slots from `prog.NumGlobals` (no one-at-a-time O(n^2) growth); the three mutation sites (`execIndexAssign` / `execAppend` / `execFieldAssign`) fetch and write the root binding through (Depth, Slot) (`getBindingRoot` / `assignRoot` , guarded to keep the REPL name path chain-walking); and `lists.reverse` / `head` / `tail` / `slice` / `concat` take a shallow struct copy instead of deep-copying the whole argument they immediately overwrite. Full suite (incl. `value_alias_test.go` , the map ordering tests, and `-race`) stays green on both toolchains.

The biggest algorithmic issue in the runtime plus the call- and loop-overhead batch. None need `reflect` or break TinyGo-cleanliness.

- **Maps as association lists (biggest win).** `Value.Map` is a `[]MapEntry` ; index reads (`indexInto`) and writes (`writeIndexedSlot`) linear-scan with a recursive `Value.Equal` per entry, and map equality is O(n*m) - building a map with `$m[$k] = $v` in a loop is quadratic. Maintain a side hash index (`map[string]int` over a canonical scalar-key encoding) alongside the ordered slice (insertion order stays a language guarantee), falling back to the linear scan for the rare non-hashable key.
- **Call / loop hot path.** `execForEach` allocates a fresh `NewEnvironmentSized` (new name map) per iteration and `execFor` an unpooled header env - both should borrow from `envPool` like `execBlock` . `DefineAt` re-runs `existsInChain` and mirrors every binding into `e.vars` on the resolver-verified slot fast path (skip the chain walk when `slot >= 0` ,

defer the name-map mirror to rare name lookups). `Run` builds `i.global` without pre-sizing from `prog.NumGlobals`, growing the slot slice one-at-a-time ($O(n^2)$ for n globals). `execIndexAssign` / `execAppend` / `execFieldAssign` re-fetch the root binding by name though the root `VarExpr` already carries `(Depth, Slot)`. `lists.reverse` / `head` / `tail` / `slice` / `concat` deep-copy the whole argument then immediately replace the copied slice - the first copy is pure waste.

Acceptance. A map-heavy build (`$m[$k] = $v` over N keys) is near-linear, not quadratic; a call-heavy and a loop-heavy benchmark improve measurably on the reference machine; every existing test (incl. `value_alias_test.go` and the map ordering tests) still passes.

M19.4 - Resource lifecycle and numeric strictness

Done. `os.spawn` handles are now keyed by a monotonic internal id, not the OS pid, so a recycled pid can never make a later spawn overwrite an earlier handle or make `os.wait` / `poll` / `kill` hit the wrong process (pinned by a distinct-handles test); the reaper also drains the captured buffers into strings and drops the live `*bytes.Buffer`s so a terminated handle stops pinning them for the program's life (idempotent `os.wait` and `poll`-after-`wait` are preserved - literal delete-on-reap would break both, so the persistent-handle contract stays and an explicit release stays a possible future add). Numeric strictness: `convert.toInt` and `math.floor` / `ceil` / `round` reject NaN, +/-Inf, and out-of-int64-range floats with positioned errors instead of an unchecked `int64(f)` cast, `math.abs(MinInt64)` errors (its magnitude does not fit), and the `toml` decoder makes an integer past int64 a decode error rather than a lossy-float downgrade (`json` keeps its deliberate fallback). The most-negative int literal `-9223372036854775808` (and `-0x8000000000000000`) now parses to `MinInt64` - folded at the unary-minus site with `ParseUint` + a 2^{63} range check - while the bare magnitude stays a range error. The uncapped-allocation sinks were already capped ahead of this milestone (`net/fs` `maxReadBytes` / `maxHandleRead`, `compress` / `archive` `maxDecompressed`). All fixes carry regression tests; full suite green on both toolchains.

- **os.spawn handle lifecycle.** `internal/lib/os/exec.go` keys process handles by OS PID and never deletes them: every `processState` (with buffered stdout / stderr) is retained for the interpreter's life, and after the reaper `Wait()`s a process the freed PID can be reused, so a later `os.spawn` overwrites the entry and `os.wait` / `poll` / `kill` on the old handle hits the wrong process. Key handles by a monotonic internal id (like `net` / `fs` / `httpd`) and delete on reap.
- **Numeric strictness.** `convert.toInt` and `math.floor` / `ceil` / `round` do an unchecked `int64(v.Float)`, platform-defined garbage for NaN, Inf, and out-of-int64-range values - contradicting the `math` library's documented strict stance (mathematically-undefined results error, never yield garbage). Reject NaN / Inf / out-of-range with positioned errors;

special-case `math.abs(MinInt64)` (currently returns a negative). The toml decoder degrades an integer literal past int64 to a lossy float - TOML 1.0 says integer overflow is an error, so make it one (json keeps its deliberate lossy-float fallback).

- **Most-negative int literal.** `-9223372036854775808` (and `0x8000000000000000`) is a parse error because the magnitude is parsed with `ParseInt` before the unary minus applies. Parse magnitudes with `ParseUint` and range-check against 2^{63} under a leading minus.

The uncapped-allocation issues (caller-supplied `make([]byte, n)` in `net / fs` reads, and unbounded `io.ReadAll` of decompressed `compress / archive` streams - a zip-bomb sink) are fixed **immediately**, with fixed sensible defaults mirroring httpd's `maxBodyBytes`, ahead of this milestone.

Acceptance. A recycled-PID scenario signals the right process (or errors) and the handle table does not grow without bound; `convert.toInt(NaN) / math.floor(1e300)` error; `math.abs(MinInt64)` errors; `-9223372036854775808` parses; a too-large toml integer is a decode error.

M19.5 - Module struct identity: keyed by canonical path

Done. Module struct values were tagged only by the imported file's **stem** (`moduleStem`), so two modules whose files share a basename (`import "a/util.j" as u1; import "b/util.j" as u2;`, or the M19.7-era `import "@mplx/benchmark/" / "@claude/benchmark/"`) produced values with an identical `(namespace, name)` identity, and `moduleByNS` resolved the stem nondeterministically - a same-named struct from an unrelated module passed the other's type check.

The original plan here was to **fail loud** (reject a stem collision at import time), but that is far too restrictive for real projects (one import would claim a basename project-wide) and would make the `@scope/package` case impossible, so the design was changed to do it properly: **struct identity is keyed by the module's canonical (resolved) path, not the basename.** `Value` and `parser.Type` gained a `ModPath` field (the module's canonical path, empty for library / user structs) that `Value.Equal` and `MatchesDeclared` compare alongside `StructNS`; `StructNS` stays the file **stem** purely for display, so error messages and `%v` still read `benchmark.Point`. The boundary `retag (retagStructs / retagType)` threads the path so a foreign struct that merely shares a stem is never mis-tagged, and method **parameter** types are now stamped (`resolveDeclaredTypesOnce`) so a `func f(s as mod.Struct)` param carries the identity the passed value does. Two imports that resolve to the *same* file stay the same type (path-resolved before comparison, so `./x.j` and `../y/x.j` collapse); different files are different types. No import errors, no collision to reject. Pinned by a same-stem-modules-coexist test (each module's struct round-trips through its own methods; the two types do not cross-satisfy) plus

distinct-stem / re-import tests; full suite, all 53 overlays, the `web` cross-boundary integration, and `-race` stay green on both toolchains.

Acceptance. Two module files sharing a basename import cleanly under distinct aliases and are distinct types that never cross-satisfy; re-importing the same file is the same type; single-stem programs and the module test suite are unaffected.

M19.6 - .j code coverage

Done. `jennifer test --coverage[=text|json]` reports statement coverage by reusing the profiler's per-position hit data (no second counting path): `loadForTestProg` installs a statement profiler before running so hits are captured from top-level init through every test method, and `renderCoverage` intersects those hits with the executable positions statically walked from the AST (`statementPositions`, which mirrors `execStmt`'s per-statement recording so the sets line up). It is scoped to the tested program's files - an imported module that merely ran does not skew it - and reports per-file plus a total; a module overlay shows the module and test files separately. `text` (default) prints percentages and names the never-executed positions; `json` emits a parseable report that owns stdout (the human test report moves to stderr, the `profile --format=pprof` rule). The plain `jennifer test` path is unchanged (`loadForTest` is now a thin wrapper passing a nil collector). Pinned by render-logic unit tests and end-to-end CLI tests (partial coverage names the uncovered lines; json parses on stdout; the plain run emits no coverage), and an HTML view is left as a later `htmlwriter` consumer.

Teach `jennifer test` to report which lines of the code under test actually ran. The profiler (`jennifer profile`) already records per-position hit counts, so the raw signal exists; this milestone surfaces it as coverage: a per-file and total percentage, the list of never-executed positions, and a machine-readable form a CI job or an editor can consume. It reuses the profiler's instrumentation rather than adding a second counting path, and is independent of any renderer: a plain-text summary is the baseline output. Educationally it closes the loop the REPL / linter / profiler / test-runner set opened - a learner sees not just that tests pass but what the tests miss.

- **Surface.** `jennifer test --coverage[=FORMAT]` runs the suite with the profiler's counters live over the tested file(s) and emits a coverage report next to the test report. Default `text` (per-file and total percent); plus a machine-readable form for tooling.
- **Reuse, do not duplicate.** Coverage is a second consumer of the profiler's per-position hit data, not a new instrumentation pass.
- **No renderer dependency.** Text is the baseline; an HTML coverage view would be a later consumer built on `htmlwriter` (M18.2), not part of this milestone.

Acceptance. The coverage report over a file whose tests exercise some but not all of its methods shows below 100 percent and names the unexecuted positions; a suite that touches

every position shows 100 percent; the machine-readable form parses; the plain `jennifer test` path (no `--coverage`) is byte-for-byte unchanged.

M19.7 - `@scope/package` module resolution (vendored packages)

Done. A leading `@` is a vendored-deck reference, expanded by one function (`resolveVendor` in `internal/module`): the `@` swaps in the vendor root and a reference not ending in `.j` gets the package-named entry appended, so `@claude/bitcoin`, `@claude/bitcoin/`, and an explicit `@claude/bitcoin/utils.j` all reduce to a plain absolute path - after which resolution, the run-once cache, and M19.5 path identity are untouched. Because the entry is `<package>.j` (named after the deck), `moduleStem` gives the package name, so the display namespace and default alias fall out with zero special-casing (`import "@claude/bitcoin/"` binds `bitcoin.`); two same-package decks across scopes (`@a/bitcoin`, `@b/bitcoin`) are distinct types (M19.5) and collide only on the default alias, resolved with `as`. The vendor root comes from `module.FindVendorRoot` with the `sysmoddir`-style layering (`--vendor` flag on `run`, else `JENNIFER_VENDOR`, else the nearest `vendor/` above the program; wired into `run / repl / test` via `SetVendorRoot`). Path safety: `@` only at the front, no `./` `../` segments, and the resolved file must stay inside the deck directory; a missing vendor root is a guided error, not a crash. The parser exempts `@` deck references from the `.j` requirement. Pinned by module-package unit tests (expansion, error cases, vendor-root discovery) and end-to-end CLI tests (entry / explicit-file / default-alias imports, cross-deck type distinctness, missing-root error); full suite, all 53 overlays, and both toolchains stay green. The `jvc` manager layered over this stays DRAFT#12.

Today the module resolver keys on the import path's leading token: `./` (or `../`) is local (relative to the importing file), `/` is absolute, and a bare name resolves through the search path (system module dir, then each `-I DIR`). None of those reach a package installed **beside the app** rather than system-wide, and a bare `import "util.j"`; deliberately never consults the importing file's own tree, so it cannot address a vendored package and would collide with a system module of the same stem. A package manager (the planned `jvc`, DRAFT#12 in [horizon.md](#)) installs packages into a project-local `vendor/SCOPE/PACKAGE/` tree; this milestone teaches the resolver to address that tree, so a vendored package imports unambiguously and independently of the system search path.

- **A fourth leading-token kind, expanded by one function.** `import "@scope/package/"` [`as alias`]; addresses a **deck** under the vendor root. Both ends are pure path expansion, done once in `expandModule()`: a leading `@` swaps in the vendor root (`@` is the `<vendorRoot>/` shortcut), and a reference that does not end in `.j` gets the deck's entry file appended, so `@claude/bitcoin`, `@claude/bitcoin/`, and an explicit `@claude/bitcoin/utils.j` all reduce to a plain absolute path. After that expansion every

downstream step (resolution, run-once cache, M19.5 path identity) is unchanged - it is just a path import, and the vendor variable only affects this expansion. It stays an ordinary string-literal path, so there is no new grammar (the inline version selectors `@pkg=1.2.3 / @pkg#commit` are `jvc`'s concern, DRAFT#12).

- **Package-named entry, so the namespace is free.** A deck's entry file is `<package>.j` (`vendor/claude/bitcoin/bitcoin.j`), named after the deck directory. Because a module's display namespace and default alias both come from `moduleStem` (the basename), that convention makes them fall out with **zero special-casing**: `import "@claude/bitcoin/"` binds the `bitcoin.` namespace and `def x as bitcoin.Wallet` type-checks, all from the existing stem logic. Without an alias the default namespace is thus the package name (never the useless `main` a fixed entry name would give); two decks with the same package name (`@a/bitcoin`, `@b/bitcoin`) default to the same alias, collide, and take an explicit `as` (the two-`util.j` rule) - typical use is aliased anyway.
- **Vendor-root discovery.** Walk up from the entry program's directory to the nearest `vendor/` (the `node_modules` / Composer convention), overridable with `--vendor DIR` and `JENNIFER_VENDOR` - the same override layering as `--sysmoddir / JENNIFER_SYSMODDIR`. A missing vendor root, or a missing `@scope/package`, is a positioned error naming the resolved path and pointing at the installer.
- **Identity already handled by M19.5.** Because struct identity is keyed by the resolved canonical path (M19.5), two decks that share a stem (`@a/util`, `@b/util`) resolve to different vendor paths and are already distinct types - this resolver just has to produce the right canonical path per scope; no extra identity work.
- **Path safety.** `@` is legal only as the very first character of the path; a `..` segment may not escape the package root; the resolved path must stay inside `vendor/scope/package/`.

The full `jvc` package manager (registry, `deck.toml` manifest, lockfile, version constraint solving, fetching) stays a beyond-1.0.0 draft (DRAFT#12); this milestone lands only the interpreter-side resolution `jvc` builds on, so a vendored tree populated by hand (or by an early `jvc`) imports today.

Acceptance. `import "@acme/widgets/" as w;` in an app with `vendor/acme/widgets/widgets.j` resolves and runs (the trailing `/` expands to the package-named entry); an explicit `@acme/widgets/util.j` also resolves; without an alias the namespace defaults to `widgets`; the vendor root is found by the upward walk and is overridable via flag / env; a missing package is a positioned error; `./`, `/`, and bare-name imports are byte-for-byte unchanged; `@a/bitcoin` and `@b/bitcoin` structs do not cross-satisfy each other's type checks; a `..`-escape path is rejected.

M19.8 - Relocation: jennifer-language org + vanity module path

A one-time project relocation, no language or interpreter behavior change. The repository moves from the personal `github.com/mplx/jennifer-lang` to a `jennifer-language` GitHub org, and - separately - the Go module path moves **off** GitHub to a vanity import path (`jennifer-lang.dev/jennifer`) served by a `go-import` meta page, so the module identity no longer depends on where the code is hosted. Names are settled: the domain `jennifer-lang.dev` is registered, and the org is `jennifer-language` because `jennifer-lang` and `jenniferlang` were already taken on GitHub. The domain (`-lang`) and org (`-language`) spelling differing is deliberate and invisible in use - the canonical module path is the domain, and the org is only the git host the meta page points at. Purely mechanical, but it touches nearly every file (112 `.go` imports, ~60 docs, CI, packaging), so it gets a milestone to keep the sweep complete and reviewable, and it lands **before the first post-org release** so downstream references (AUR, doc links) migrate exactly once. Two distinct targets - keep them separate:

- **Go module path -> the vanity domain.** `go.mod` becomes `module jennifer-lang.dev/jennifer`; every internal import (`github.com/mplx/jennifer-lang/...` -> `jennifer-lang.dev/jennifer/...`, 112 `.go` files) is rewritten. A one-file `go-import` meta page at `jennifer-lang.dev/jennifer` maps the vanity path to the org repo (`<meta name="go-import" content="jennifer-lang.dev/jennifer git https://github.com/jennifer-language/jennifer">`, plus a `go-source` tag for pkg.go.dev deep links), served from the same site that hosts the mdBook docs. The path is now host-independent: a future repo move never touches `go.mod` again.
- **Human-facing URLs -> the org repo.** Clone URLs, issue links, and every `github.com/mplx/jennifer-lang/blob/main/...` deep link move to `github.com/jennifer-language/jennifer` (flagship repo named `jennifer`, not doubled - the `rust-lang/rust` shape). These point at GitHub, **not** the vanity domain (the `.dev` host only serves the `go-import` page and the docs). GitHub's transfer redirect keeps old links working, but every canonical URL in-tree is updated rather than left to the redirect. Sibling repos (`jvc`, the deck registry) are created empty under the org as their own milestones land, not here.
- **Metadata / CI / packaging sweep.** `README.md` badges and links, `docs/**` (installing, tooling, user-guide, technical), `CLAUDE.md`, `JENNIFER.md`, `modules/README.md`, `book.toml`, the one example that hardcodes the URL (`examples/modules/barcode_demo.j`), the workflows (`.github/workflows/{test,docs,release}.yml`), and the packaging manifests (`packaging/arch/PKGBUILD-{bin,git}`, `packaging/arch/publish-{bin,git}.sh`, `packaging/debian/copyright`, `scripts/build-deb.sh`) all move to the new host. `grep -rn 'mplx/jennifer-lang'` comes back empty at the end.
- **Jennifer deck scope.** The first-party package scope in docs / examples flips from the placeholder `@mplx/...` to `@jennifer/...` (DRAFT#12 and the M19.7 examples),

matching the org and the registry identity the vanity domain anchors. Doc-only until `jvc` and the registry exist.

Acceptance. `go get jennifer-lang.dev/jennifer/cmd/jennifer` resolves through the vanity meta page to the org repo; `go build ./...` and `go test ./...` pass under the new module path; `make build` (both toolchains) still produces `jennifer / jennifer-tiny`; `grep -rn 'mplx/jennifer-lang'` is empty; the old GitHub URL redirects to the org; the AUR package builds from the new source; `pkg.go.dev` serves the module under `jennifer-lang.dev/jennifer`.

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](#)). Members below; more land as M20.x as needs arise.

M20.1 - crypto

Symmetric and asymmetric primitives, key derivation, crypto-grade random. System library; TinyGo-safe primitives only. Hashes already shipped in M15.6.

- **Swap `uuid`'s random source.** `uuid` (M16.10) draws its v4 / v7 randomness from `math`'s shared non-crypto RNG (seedable, predictable - documented). When crypto-grade random lands here, repoint `uuidlib.randByte` at it so `uuid.v4` is unguessable; the change is one function, no surface change. Until then `uuid` must not be used for security tokens.
- **Message authentication (HMAC) already shipped as `hash.hmac`.** HMAC is a hash construction, so it lives in the `hash` library (RFC 2104, Go `crypto/hmac`, TinyGo-clean) rather than here - it unblocks request signing, webhook verification, JWT HS256, and TOTP without a crypto library. What can still land here is a constant-time `crypto.hmacEqual` for MAC comparison (verification today recomputes and compares the full digest). The KDFs below build on HMAC (PBKDF2 is iterated HMAC; HKDF and SASL SCRAM are HMAC-based).
- **Key derivation (stdlib, no dependency).** `HKDF` - derive keys from a high-entropy secret - and `PBKDF2-HMAC-SHA256` - derive a key from a password (salt + iteration count). Both come from the Go standard library (`crypto/hkdf`, `crypto/pbkdf2`, stdlib since Go 1.24), so they add no dependency and stay TinyGo-clean. Shape: `crypto.hkdf(...)` / `crypto.pbkdf2(password, salt, iterations, keyLen) -> bytes`.

- **Password hashing is out of scope here.** Argon2id (and bcrypt / scrypt) moved to the Long-horizon list: they need the `x/crypto` dependency and their own `hashPassword` / `verifyPassword` surface.

M20.2 - xml

Hand-rolled like `json` (Go's `encoding/xml` is reflect-heavy, so TinyGo-hostile). A genuinely complex tree - attributes + ordered, possibly-duplicated children + mixed text + namespaces + entities - whose byte-level parsing is too slow in `.j`. Also the natural mirror target for the `json.Value` read / write vocabulary (M16.16): the same opaque-handle plus path-addressed accessor shape (an XPath-style path dialect in place of JSON Pointer).

M20.3 - yaml

A system library because full YAML - anchors / aliases, flow *and* block styles, implicit typing, multi-document streams - is impractical in `.j` and has no Go stdlib. Unlike `xml`, that means a **Go dependency** (e.g. `gopkg.in/yaml.v3`): the one place a config parser earns one, since a hand-rolled full YAML is a project of its own. Verify TinyGo-cleanliness of the dependency, and fall back to a documented subset if it won't build there.

M20.4 - i18n

Message catalogs and locale-aware translation. A **system library**, not a `.j` module, for two independent reasons: it needs **global mutable state** (the current locale plus loaded catalogs), which a declarations-only module cannot hold; and it needs **performance** - Jennifer's `map` is a linear-scan `[]MapEntry`, so a large catalog looked up per call would be $O(n)$, whereas the library holds catalogs in a Go `map[string]string` ($O(1)$). The `map of string to string` a caller passes is fine as the *load* interface (a one-time ingest); the per-lookup scan is what the library avoids.

Surface: `i18n.load(lang, catalog)` (a `map of string to string`, built from `json` today or `yaml` here at M20.3, or a literal), `i18n.setLocale(lang)` / `i18n.locale()`, `i18n.tr(key)` (translate in the current locale; fallback locale -> default lang -> the key itself, so a missing key is visible), and `i18n.tr(key, params)` for named interpolation ("Hello, {name}"). Pluralization (CLDR per-language plural rules) and an `i18n.loadFile(path)` convenience are follow-ons.

No gettext-style `_()`: `_` is not a valid Jennifer method name (letters-only; `_` is reserved for constant-name separators), and a bare `tr()` builtin does not clear the `len` promotion bar (translation is not useful to nearly every program). Ambient-global `_()` is also exactly what

stances 2 (explicit) and 7 (namespaced, no globals) rule out - so the call is `i18n.tr("key")` (or `use i18n as t; t.tr("key")`). Extending `printf` for translation is rejected (see [technical/rejected.md](#)): translation is content substitution from stateful external data, not presentation of the value in hand. Locale-aware *value* formatting (number / date grouping) is a separate, open `printf` question.

M20.5 - term

A `term` system library exposing the terminal host capabilities a TUI needs and pure `.j` cannot reach: **raw mode** (`makeRaw` / `restore` - unbuffered, no-echo input), **terminal size** (`size` -> `rows`, `cols`), and raw single-key reads from stdin. It reuses `golang.org/x/term` - already a repository dependency scoped to the REPL's line editor (`cmd/jennifer/lineedit.go`) - so it largely exposes a capability the interpreter already exercises. Build-tag split like `net` / `os`: a friendly-error stub on `jennifer-tiny` (embedded / minimal targets may have no controlling TTY). This is the **enabler for interactive TUIs**; the pure-ANSI screen control, key decoding, and rendering sit in the [M21.1](#) `screen` / `tui` module on top. Output-only TUIs (dashboards, progress bars) need neither this library nor raw mode - just `ansi` + `os.IsTerminal`.

M20.6 - hardware buses (serial / spi / iic)

Device-bus I/O libraries for embedded / single-board-computer hosts - the syscall-backed siblings of the sysfs-backed `gpio` module ([M18.11](#)). Each needs `ioctl` / `termios`, which `.j` and plain `fs` cannot reach, so they are Go **system libraries**, not modules:

- **serial** - open a serial port (`/dev/ttyUSB0`, `/dev/ttyAMA0`), configure it (baud rate, data bits, parity, stop bits via `termios`), then `read` / `write` / `close`. Setting the baud rate is a `termios` `ioctl`, unreachable from `fs` - which is what forces a library.
- **spi** - open a SPI device (`/dev/spidev0.0`), set mode / speed, and `transfer(bytes)` -> `bytes` (full-duplex), via the `SPI_IOC_MESSAGE` `ioctl`.
- **iic** - the I2C bus (`/dev/i2c-1`): select a slave address and `read` / `write` register bytes via the `I2C_SLAVE` `ioctl`. Named **iic** (Inter-IC) rather than `i2c` because a library namespace is letters-only (no digit, like `bucket` not `s3`); candidates `iic` / `twi` / `wire`, settled at build time.

Build-tag split like `net` / `os`: the real implementation on Linux (the supported platform), a friendly-error stub elsewhere. Default binary; a `jennifer-tiny` rebuilt for a specific board could include them (TinyGo's `machine` package is a different, microcontroller-level API - these target the Linux `/dev` + `ioctl` interface). Together with `gpio` they complete the SBC I/O story.

M20.7 - sql (MySQL / MariaDB + PostgreSQL)

A relational-database client library over Go's `database/sql`, shipping the two **client-server** engines: MySQL / MariaDB (`go-sql-driver/mysql`) and PostgreSQL (`jackc/pgx`), both **pure-Go** drivers (no cgo, so cross-compile and the best-effort macOS / Windows artifacts stay clean). SQLite - the one *embedded* engine - is deliberately **not** here; it needs a multi-MB dependency and cannot build under TinyGo, so it stays a build-tag opt-in parked in [horizon](#) (the `jennifer-full` variant).

Why a Go library and not a `.j` module over `net`. MySQL and Postgres are open TCP wire protocols, so a client *is* writable in pure Jennifer - the same shape as `redis` / `memcache` / `imap`, and the auth crypto it needs (SHA-1, SHA-256, `hash.hmac`, PBKDF2 as iterated HMAC) already ships. The deciding factor is not performance: a database client is **latency-bound** (network round-trip + server execution dominate; client-side decode only becomes the cost center when streaming 10^5+ rows, a bulk workload Jennifer is not the right tool for regardless of driver), and the COW shared-marker protocol already makes materializing a large result set amortized $O(N)$. The deciding factor is **correctness maturity**: the mature drivers have absorbed a decade of protocol long-tail - every auth plugin (MySQL `caching_sha2_password` full-auth / the RSA path, Postgres 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. Going Go also makes the auth crypto the driver's problem, not the language's, so this needs nothing from [M20.1 crypto](#).

The deliberate dependency break. These are the **first heavyweight dependencies in the library layer** - a conscious exception to the dependency-free discipline [CLAUDE.md](#) states for the library layer (the only third-party dependency today is CLI-scoped [golang.org/x/term](#)). Both drivers are pure-Go, but they are real dependency trees. The exception gets a reasoning record in [technical/design-decisions.md](#) when it lands - the sanctioned home for a feature that ships despite appearing to cut against project doctrine - justified as above, not slid into.

Build / TinyGo. Build-tag split exactly like `net / httpd: sqllib_std.go (//go:build !tinygo)` imports `database/sql + drivers; sqllib_tiny.go (//go:build tinygo)` registers `use sql`; and returns a friendly positioned "not available on this build" error. TinyGo never compiles the driver imports, so the language stays TinyGo-clean and the interpreter core is untouched. On stock `jennifer-tiny` the engines are unavailable anyway (no net driver), consistent with every net-backed module.

Surface. Integer-handle-into-a-registry like `fs / net: sql.open(driver, dsn) -> Connection, sql.query(conn, sql, params...) -> Rows, sql.exec(conn, sql, params...) -> Result` (affected-rows / last-insert-id), prepared statements, `sql.begin / commit / rollback` transactions, `sql.close`. **Values bind only through placeholders** (injection safety) -

`database/sql` abstracts the per-driver spelling (`?` for MySQL, `$1` for Postgres). The result-row shape is an opaque `sql.Row KindObject` mirroring `json.Value` (foreshadowed in interpreter note 18), walked by accessors; a typed-struct path waits on the deferred map-to-struct conversion. A `.j postgres.j` module (Postgres has the clean protocol, no RSA gap) remains a possible later *optional* dependency-free alternative and language stress-test - not the primary path.

Builds on. The `M21.5 orm module` is layered on this library (its hard prerequisite) and inherits the `sql.Row` caveat above - its typed-row ergonomics wait on the same deferred map-to-struct conversion, so it graduates in stages.

M21 - general backlog (catch-all)

The general holding area for milestones that fit no other bucket - 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 (embedding, WASM, and the rest live in the [horizon collection](#)). It is the top-level counterpart to M19's tooling bucket: anything worth recording that has no natural home lands here as a numbered sub-entry, and graduates out into its own bucket once a cluster grows enough to deserve one.

M21.1 - screen / tui module

Jennifer's terminal-UI answer - a `.j` module for terminal user interfaces, since there is no GUI medium-term, so the terminal is the interactive surface. Layered so the output-only subset ships with no host dependency:

- **Stage 1 (no prerequisite).** Pure-ANSI screen control over `ansi`: cursor movement, clear, alternate-screen buffer, hide / show cursor, box-drawing, and a screen buffer + render / diff loop. Enables **output-only TUIs** - live dashboards, progress bars, spinners, self-updating tables (the `rich`-style subset). Pure strings; TinyGo-clean.
- **Stage 2 (needs M20.5).** A key-event decoder (parse the raw byte stream - `ESC [ A` -> Up, and so on) plus an event loop over `term`'s raw mode / size / key reads. Enables **interactive TUIs** - menus, forms, key navigation (the `curses` / `bubbletea` subset).

Positioned as an *explicit* terminal UI, not a GUI framework. Named `screen` or `tui` (settled at build time). Parked in the general backlog for now; it can graduate into the M18 module track when built.

M21.2 - feed module (RSS + Atom)

A `feed` module for web syndication feeds: **build and parse** both RSS 2.0 and Atom 1.0. **One** module, format selected on build and detected on parse (design stance 1 - not separate `rss` / `atom` modules); a feed is a value- semantic `Feed` (title, link, updated, entries) of `Entry` (title, link, id, published / updated, summary, content). It composes across the stack: `http` fetches a feed, `feed` parses it, `time` handles the RFC 3339 / RFC 822 dates - a feed reader, podcast client, news aggregator, or changelog-to-feed generator in a few lines.

Parked here because it is **gated on M20.2 `xml`**: RSS / Atom parsing needs a real XML parser (entities, CDATA, and Atom's XML namespaces), which is exactly why `xml` is a system library rather than a hand-rolled `.j` scanner - so feed *parsing* rides it. Feed *building* (emitting escaped XML) could predate `xml`, but build and parse ship together on the same layer. Also uses `http` (fetch) and `time` (dates); pure `.j` otherwise. It graduates into the M18 module track once `xml` lands. Discipline as usual: a `modules/feed_test.j` overlay (build / parse round-trips and date handling for both formats as pure helpers; a networked fetch-and-parse against an in-process server in a Go test), `docs/modules/feed.md`, a catalog row, a `SUMMARY.md` entry, a `modules/README.md` entry, a `JENNIFER.md` bullet, and a runnable `examples/modules/feed_demo.j`.

M21.3 - jwt module (JSON Web Tokens)

A JWT (RFC 7519) module. The HMAC algorithms (HS256 / HS384 / HS512) need only the shipped `hash.hmac` and could stand alone, but the module targets the full common surface - including the asymmetric **RS256 / ES256** that OAuth / OIDC rely on - so it is parked here **gated on M20.1 `crypto`** for the public-key signing / verification, and graduates into the M18 module track when `crypto` lands. Surface: `jwt.sign(claims, key, alg)`, `jwt.verify(token, key) -> claims` (checks the signature and the `exp` / `nbf` time claims), and `jwt.decode(token) -> claims` (read without verifying). Over `hash.hmac` (+ `crypto` for RS / ES), `encoding` (base64url), `json`, and `time`. `jwt_auth` is not a separate module: it is this module used as a `web.before` middleware that pulls the bearer token from the `Authorization` header, `jwt.verify`s it, and rejects on failure (`func requireJwt(ctx) { ...; return true; }`) - shipped as a snippet in the demo / docs, not its own surface. Discipline as usual: a `modules/jwt_test.j` overlay (sign / verify round-trips and tampered-token rejection), docs, catalog, and demo.

M21.4 - acme module (Let's Encrypt / ACME)

An ACME (RFC 8555) client - obtain and renew TLS certificates from Let's Encrypt and compatible CAs: account registration, an order plus HTTP-01 / DNS-01 challenge, CSR

submission, and certificate download, over `http + json`. Parked here **gated on M20.1**

crypto: ACME requests are JWS-signed with an account key (RS256 / ES256) and the flow needs CSR generation - both asymmetric-crypto operations. Composes with `web / httpd` to serve the HTTP-01 challenge. Graduates into the M18 module track when `crypto` lands. Needs the default binary. Discipline as usual.

M21.5 - orm module

A relational mapper layered over the [M20.7 sql](#) library (its **hard prerequisite** - no `sql`, no `orm`), and a good stress-test of how far the module system stretches. Jennifer's semantics dictate the shape, and it is **Data Mapper, not Active Record**: structs are value-semantic and carry no methods, and a module is declarations-only with no mutable state, so a row object cannot know how to `save()` / `delete()` itself and there is no place to hold identity-map or dirty-tracking state. The pattern is therefore a **repository / table-gateway** - the caller passes a record and a schema to module functions: `orm.insert(conn, schema, record)`, `orm.find(conn, schema, id) -> record`, `orm.update / orm.delete`, `orm.all(conn, query)`.

Three constraints shape the surface:

- **Explicit schema descriptor (no reflection).** `.j` cannot introspect a struct's fields, so the caller declares the mapping once - an `orm.Schema` (table name, column-to-field list, primary key, per-column type) built with a small constructor. This is the module's central object; everything keys off it.
- **Non-mutating functional query builder**, mirroring the `json.Value` write surface (`set / insert / append` returning fresh handles) rather than method chaining (values have no methods): `orm.where(orm.from(schema), "age", ">", 18)` returns a new `orm.Query`, composed functionally through `orm.where / orm.orderBy / orm.limit / orm.join`, then rendered to **parameterized** SQL - values bind only through placeholders (injection safety inherited from `sql`). A small **dialect** layer (placeholder spelling `?` vs `$1`, identifier quoting, `LIMIT / OFFSET`, `RETURNING`) is a backend selector on one module, not parallel modules (stance 1 / the one-module rule).
- **Row-to-struct mapping is partly gated.** `sql.query` yields an opaque `sql.Row`; turning it into a *typed* struct wants the deferred **explicit map-to-struct conversion** ([horizon](#)). So `orm` ships in two steps: first a `map of string to V` (or by-hand field extraction via the schema) row form that needs nothing new, then the typed-struct return once map-to-struct lands.

Transactions come straight from `sql.begin / commit / rollback`. Kept out of v1: relations beyond a plain `join` (has-many / belongs-to eager loading wants object identity and lazy proxies Jennifer does not have), and migrations (a thin `orm.createTable(schema)` DDL emitter is a plausible follow-on, full migration tooling is its own module). Needs the default binary.

Discipline as usual: a `modules/orm_test.j` overlay, docs, catalog, and demo - with the overlay split so the **query-builder-to-SQL** surface (pure string generation) is covered 100% offline, and live CRUD sits behind a DB-service-gated integration test rather than the unit overlay.

Requirements for 1.0.0 stable

The core CI + release + packaging items that used to live here were promoted into M15.8 (the last step before Phase C). What stays here are the distribution requirements for a stable 1.0.0 that aren't themselves milestones - they can land any time and don't block any feature milestone:

- **Cross-build for macOS / Windows.** Waits on the platform-portability work in the [horizon ideas](#); ships as soon as that lands.
- **Real apt repository** (replacing the "GitHub Release artifact" install of the M15.8 `.deb`) if user demand warrants the maintenance.

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 (embedding, WASM, specialised-domain libraries). Each has a design; it just has no schedule.
- **Loose ideas** - a grab-bag of smaller or vaguer possibilities, 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

Embedding, WASM, and specialised domains. Recorded so the design doesn't foreclose them, roughly ordered by the size of the structural change each is: the embedding restructure is a single-repo refactor; the WASM runtime brings in a whole new dependency; the specialised-domain libraries are indefinite-in-count families.

Each carries a stable **DRAFT#** handle so it can be referenced and later graduated into a numbered milestone (e.g. "shift **DRAFT#9** to **M20.6**"). 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. **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.

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).

Specialised domains

Each domain its own effort with sub-pieces as needed:

- **ML.**
 - **DRAFT#5 stats** - descriptive statistics over `list of int|float`: mean, median, mode, variance, stddev, percentile, min / max / sum, correlation. Pure-value, TinyGo-clean; the highest-value, simplest piece, so first. **Requires:** none.
 - **DRAFT#6 linalg** - vectors as `list of float` (dot, norm, cross, scale, add / sub) and matrices as `list of list of float` (matmul, transpose, determinant, inverse, solve, identity). Algorithms implemented directly - no `gonum`, too large a dependency. Matrices stay `list of list of float` for v1 (idiomatic and value-semantic); a Go-backed matrix handle is the noted future escape hatch when big-matrix performance demands it. **Requires:** none.
 - **DRAFT#7 ML primitives** - atop `stats` / `linalg`, when demand surfaces. **Requires:** DRAFT#5 + DRAFT#6.
- **DRAFT#8 Bioinformatics.** Sequence alignment (Smith-Waterman, Needleman-Wunsch), FASTA/FASTQ parsers, molecule structures. **Requires:** none.
- **Encoding / binary protocols.**
 - **DRAFT#9 asn1** - ASN.1 BER/DER encode/decode, as a **Go system library**. Byte-level binary parsing belongs in Go, not `.j` (the `json` lesson: a char-by-char parser in the interpreter pays overhead per byte). This is the *enabler* for a family of binary protocols and PKI formats - LDAP, SNMP, X.509, PKCS. Go's stdlib `encoding/asn1` is DER-only, so the full BER that LDAP / SNMP use (indefinite lengths, alternative encodings) needs either a BER dependency (e.g. `go-asn1-ber`) or a hand-rolled BER codec in Go. **Requires:** none (it is the enabler for the rest of this group).

- **DRAFT#10 ldap / snmp (layered on asn1 + net)**. With `asn1` doing the byte crunching in Go and `net` providing TCP/UDP + TLS (`connectTLS` / `startTLS` already cover LDAPS / StartTLS), the protocol orchestration (bind, build request, iterate results) is not per-byte hot and can live in a `.j` module or a thin Go library. SNMP is the natural first client (simpler PDUs, UDP, no SASL); LDAP adds controls + SASL (SCRAM builds on the `crypto` library). A pure- `.j` implementation of the BER layer itself is explicitly *not* the plan. Existing pure implementations (e.g. PHP FreeDSx) are a protocol reference, not a port target - their heavy OO shape does not map to Jennifer's value-semantic structs. **Requires:** **DRAFT#9** (`asn1`) and the shipped `net` library; LDAP's SASL / SCRAM path additionally needs `crypto` (`M20.1`).
- **DRAFT#11 Sandbox**. Restricted-capability execution. **Requires:** none hard; relates to **DRAFT#1** (embedding) and **DRAFT#3** (WASM isolation).

Ordered when demand surfaces. The WASM libraries idea (**DRAFT#3**) may cover some of this space first.

DRAFT#12 - jvc package manager (decks)

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.
- **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**.

A whole track of its own. **Requires:** [M19.7](#) (the `@scope/package` resolver + vendor root the scheme rests on); the `toml` library (M18.8, for `deck.toml`); the shipped module system and the `semver` module (constraint solving - its `satisfies` / `maxSatisfying` / `minSatisfying` / `validRange` range surface is the resolver primitive); and `http` / `git` (fetching). The public deck **registry** is separate infrastructure, provided later.

DRAFT#13 - Higher-level PDF: font metrics, layout, and Markdown -> PDF

A three-phase build on top of the shipped `pdfwriter` module (M18.35), taking it from “place text and shapes at coordinates” to “flow a document”. The phases are strictly ordered because each depends on the one before; they land as separate milestones when graduated, but share one draft handle because they are one arc.

Phase 1 - font metrics (the keystone). Today `pdfwriter` can place text but cannot **measure** it, so there is no way to wrap a paragraph, auto-size a table column, or align text - all of which need the rendered width of a string. Add the **standard-14 AFM width tables** (public Adobe Font Metrics: per-glyph advance widths, in 1/1000 em, for WinAnsiEncoding) and a `pdfwriter.stringWidth(font, size, text) -> int` that sums them. Courier is monospace (600 units/glyph, trivial); Helvetica and Times need the real per-character tables, generated into

an included `.j` data file the way the `encoding` codecs were generated from the Unicode mapping files (`gen_*.go` -> `*_gen.j`). First payoff: `textAligned(pg, x, y, font, size, align, str)` for left / center / right placement. Nothing in the later phases is expressible without this.

Phase 2 - table and flow layout. The layer that removes the manual cell-positioning pain (the ergonomic win people reach for TCPDF's `writeHTML` tables to get - but as a typed API, not an HTML subset). On top of `stringWidth: table(columns, rows, options)` (auto column sizing, in-cell word wrap, borders, a header row, page-break across rows); `paragraph(pg, text, x, y, width, font, size) -> int` (wrapped flowing text that returns the y it ended at, so callers stack blocks); and `heading` / list helpers.

Phase 3 - Markdown -> PDF. The markup-driven document story, and the reason it is Markdown rather than HTML: Jennifer ships a `markdown` parser (GFM tables, headings, lists, emphasis) but no HTML *parser* (`htmlwriter` only builds HTML), so the cheap, ergonomic path to “write markup, get a PDF” is to drive the Phase-2 layout layer from the existing Markdown parse rather than write a quirky HTML-subset parser (Markdown tables are also easier to author than HTML tables). A prerequisite to verify first: whether the `markdown` module exposes a reusable parse tree or only renders straight to a string - if the latter, it needs a small refactor to surface the intermediate document model. An HTML-subset front-end (TCPDF-style) stays a *later* option, only worth it for consuming pre-existing HTML, and it would sit on the same layout foundation.

Stays pure `.j` (static data + lookups + layout math), both binaries. **Requires:** none hard for Phase 1 (builds on the shipped `pdfwriter` module, M18.35); Phase 2 requires Phase 1; Phase 3 requires Phase 2 and the shipped `markdown` module (plus possibly a parse-tree surface on it).

DRAFT#14 - Project governance, licensing, and contribution policy

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`

`<developer@mplx.eu>` , `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 draft is about the *process and ownership* around it, not a license change.

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. Not legal advice - the chosen model should get a real legal review before it is published.

Loose ideas

A grab-bag, recorded when it comes up.

- **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.
- **Extra distribution packaging.** Beyond the Linux `.deb` (shipped in M15.8) and the two distribution *requirements* for 1.0.0 stable (cross-build for macOS / Windows, a real apt repository), the additional package formats are nice-to-haves, each shipped only when a user asks and a maintainer will keep it green: a **Homebrew tap** (macOS), a **Snap** package, a **Nix flake** / Nix package, and **Flatpak** / **AppImage** or any other Linux distribution format. None blocks a release; none is a 1.0.0 requirement.
- **Cross-platform support.** Linux is the only *supported* platform, but best-effort **unsupported** macOS and Windows binaries (the standard-Go `jennifer`, via cross-compile) already ship with each release - so the work here is not "add the ports" but *promoting them to supported* (real CI coverage, per-OS golden strategy, the platform-specific edge cases). When touching filesystem, paths, line endings, or process behavior, prefer portable stdlib helpers (`path/filepath`, not hardcoded `/`); avoid Linux-only assumptions so those binaries stay genuinely portable, not just compile-clean.
- **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)` $\rightarrow$ `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.
- **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)  $\rightarrow$  string ( $argon2id$... )` plus a constant-time `crypto.VerifyPassword(pw, hash)  $\rightarrow$  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.

- **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).
- **FCGI.** use `FCGI` as web; library when `net` and `httpd` mature. Lets Jennifer host CGI / FastCGI workloads end-to-end.
- **Inline assembler.**
- **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.
- **Profiler: max-call-depth metric.** Have `jennifer profile` track Jennifer call depth (bump in `evalCall` , drop on return) and report the max reached, per source position and overall. Names stack-limit problems directly - the recursion-depth-vs- `-stack-size` headroom that the recursive `fib` in `examples/benchmark.j` exercises on `jennifer-tiny` . Small and additive to the existing hit-count / wall-clock / `--allocs` collector; deferred because stack limits are diagnosable by hand today. Heap-per-position stays out of scope (`--allocs` already proxies copy churn; true RSS needs `runtime.ReadMemStats` sampling, coarse under TinyGo).
- **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.

- **SQLite (`sql` engine backend).** The client-server half of relational support - MySQL / MariaDB + PostgreSQL - is committed as `M20.7` (a `sql` system library over Go’s `database/sql` , pure-Go drivers). 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 M20.7’s two 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 reimplementaion) - multiple MB of generated code, versus the hand-written, few-hundred-KB protocol clients M20.7 ships. 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) $\subset$ `jennifer` (MySQL + Postgres) $\subset$ `jennifer-full` (+ SQLite). The dependency break from “libraries stay dependency-free” is already accepted at M20.7; 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. Because SQLite is really just another driver, the only open call is timing: fold it into M20.7 behind the `-tags sqlite` gate, or keep it deferred here until the `jennifer-full` variant earns its place in the release / CI / packaging matrix. Contrast the text-protocol stores `redis` / `memcache` , pure Jennifer over `net` , which need none of this.
- **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.

- **io.lines() -> list of string.** Slurp the whole stdin into a list. Additive on top of the streaming `readLine() + eof()` idiom; nice-to-have for tiny scripts, not blocking.
- **i18n.** Locale-aware case folding, collation, number / date formatting, BiDi. Gated on the CLDR-data binary-size question (likely an optional library shipped after the WASM runtime, so locale tables aren't baked into every build).
- **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).
- **Performance & memory.** Interpreter-internal optimizations that preserve stance #5 (value semantics) at the user level: copy-on-write for lists / maps / bytes / structs (share underlying storage until a write splits it), per-frame arena allocation, and read-only slice views (`xs[1..5]` as a non-owning window that errors on assignment). Strictly optimizations - no user-visible aliasing or mutation rules change. Stance-breaking variants (mutable references, interior mutability, shared mutable state) are turned down in [technical/rejected.md](#). Best landed once the language is settled and the interpreter doesn't churn under it.

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 <code>++ / --</code> , no <code>+=</code> , no two <code>printf</code> 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 (<code>\$x</code>), <code>def</code> carries the type, libraries are imported per topic (<code>use io;</code> ; nothing auto-loads), conditions must be <code>bool</code> (no truthiness), conversions are spelled out (<code>convert.toInt(v)</code> , <code>convert.toFloat(v)</code>). Nothing important hides.
3	Presentation, not transformation, in format strings.	<code>printf</code> verb modifiers shape how a value is rendered (<code>%d base=2</code> , <code>%f prec=4</code>). Transforming the value itself (<code>upper</code> , <code>substring</code> , markdown rendering) is a library call. Keeps <code>printf</code> 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. <code>const</code> 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 <code>use NAME;</code> - 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: <code>true</code> , <code>false</code> . Comparison operators produce one.
break	language	The <code>break</code> ; statement; exits the innermost enclosing loop. Misuse outside a loop is a positioned runtime error. <i>Not</i> “leave”, “stop”.
builtin / builtins	implementation	A function or constant registered by a library via the Go-side <code>Register*</code> API. Distinguishes “ships with the interpreter” from user-defined <code>func</code> .
bytes	type	Mutable byte sequence. Indexing yields <code>int</code> in <code>[0, 255]</code> ; writes accept the same range and reject anything else. Value-typed (deep-copy on assignment / parameter binding). Built from a string via <code>convert.bytesFromString(s, "utf-8")</code> or grown from <code>def b as bytes; via \$b[] = byte; .</code> <i>Not</i> “buffer”, “bytestring”, “blob”.
constant / constants	language	A name bound once at declaration with <code>def const NAME ...</code> and never reassigned. Constants are deep-immutable for lists, maps, and structs.
continue	language	The <code>continue</code> ; statement; skips to the next iteration of the innermost loop. In C-style <code>for</code> , the step expression still runs before re-checking the condition. <i>Not</i> “next”, “skip”.
<code>core</code> library	historical	Removed. Formerly the single auto-loaded library that shipped <code>len</code> as a bare-name global; <code>len</code> is now a language built-in and the version constants moved to <code>meta</code> . Writing <code>use core;</code> produces a friendly migration error.

Term (singular / plural)	Kind	Meaning
discard	language	The <code>task.discard(\$t)</code> 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 <code>task.wait</code> and <code>task.waitAll</code>). <i>Not</i> “drop”, “detach”.
Error	type / value	The auto-hoisted struct <code>Error { kind, message, file, line, col }</code> that <code>throw</code> conventionally raises and <code>try / catch</code> binds; a runtime error is wrapped into it on entry to a catch. User code may not redefine it. <i>Not</i> “exception”, “panic”.
exit	language	The <code>exit;</code> (code 0) or <code>exit EXPR;</code> (int code) statement. Terminates the whole program; skips every caller frame and remaining top-level statement. <i>Distinct from</i> <code>return</code> (method-scoped) and <i>not</i> catchable by <code>try</code> .
float / floats	type	64-bit floating-point primitive. Mixed <code>int / float</code> arithmetic promotes to <code>float</code> .
function / functions	language	A named callable defined with <code>func NAME(...) { ... }</code> . <i>Not</i> “method” or “procedure”. The <code>func</code> keyword is the canonical source.
global	implementation	A builtin registered via <code>RegisterGlobal / RegisterGlobalConst</code> ; reachable as a bare name (no <code>lib.</code> prefix). The API is still on the interpreter but no shipping library uses it - every library is namespaced, and the polymorphic <code>len</code> is a language keyword. Kept for tests until a cleanup pass removes it.
import	language	The <code>import "path.j" [as ALIAS];</code> statement that loads a <code>.j</code> file as a module: a real module boundary with run-once init and <code>ALIAS.member</code> access. <i>Not</i> <code>include</code> (textual splice) or <code>use</code> (system library).
include	language	The <code>include "path.j";</code> statement that textually splices a file’s tokens at the include site. <i>Not</i> “import” - <code>import</code> 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

Term (singular / plural)	Kind	Meaning
		topic via <code>use NAME; .</code> <i>Not</i> “package” or “module”.
list / lists	type	Ordered, 0-indexed, mutable sequence. Element type fixed at declaration. <i>Not</i> “array”, “vector”, “sequence”.
map / maps	type	Insertion-ordered key → value container, mutable. Key and value types fixed at declaration. <i>Not</i> “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 <code>.j</code> source and loaded via <code>import "modules/foo.j" as foo; .</code> Lives under <code>modules/</code> in the source tree and ships separately from the interpreter binary. <i>Not</i> “package”, “bundle”, “plugin”.
namespace / namespaces	language	The prefix a library introduces at call sites (<code>os.getEnv()</code> , <code>lists.push(...)</code>). Activated by <code>use NAME;</code> or remapped by <code>use NAME as ALIAS; .</code>
null	type / value	A type with a single value (<code>null</code>). Returned by bodyless <code>return;</code> and by a method that runs to the end.
object	type / kind	An opaque, library-owned value (e.g. <code>json.Value</code>) that carries data reachable only through the owning library’s accessors, not through operators or <code>[i] / .field</code> . <code>convert.typeOf</code> reports the generic “object”; <code>convert.objectType</code> the specific registered name. <i>Not</i> “map”, “struct”.
observed	project	A task whose outcome (value or error) has been claimed via <code>task.wait</code> , <code>task.waitAll</code> , or <code>task.discard</code> . Only unobserved error-producing tasks trigger the exit-time loud-fail. <i>Not</i> “seen”, “consumed”.
overlay	project	A white-box test file <code>MODULE_test.j</code> that <code>jennifer test</code> splices <i>after</i> its module, so the tests reach the module’s private functions by

Term (singular / plural)	Kind	Meaning
		bare name. Every shipped module carries one that must pass 100%. <i>Not</i> “fixture”, “harness”.
parameter / parameters	language	A function’s input slot, declared <code>name as type</code> in the function header. <i>Not</i> “argument” (which is the value passed at the call site).
repeat ... until	language	Post-test loop: <code>repeat { ... } until (cond);</code> . Body runs at least once, then <code>until</code> is checked after each pass; loop stops when <code>cond</code> is true. Chosen over <code>do { } while !cond;</code> so the condition reads as “loop until done.”
sigil	language	The <code>\$</code> prefix on a variable use site. Marks “read or assign a mutable binding”; constants and functions do <i>not</i> take the sigil.
spawn	language	The <code>spawn { ... }</code> block expression. Runs its body concurrently with the rest of the program and returns immediately with a <code>task of T</code> handle. Captures its enclosing scope by deep copy (value-semantics capture). <i>Not</i> “async”, “go”, “launch”.
stance / stances	project	One of the seven design rules. Tie-breakers for ambiguous design choices.
statement / statements	language	A <code>;</code> -terminated unit of source: <code>def</code> , assignment, control flow, expression statement, <code>include</code> , <code>use</code> .
string / strings	type	Immutable UTF-8 text primitive. Both <code>"..."</code> and <code>'...'</code> are valid delimiters.
struct / structs	type	A composite value type with named, typed fields, declared <code>def struct Name { field as type, ... };</code> . Field access via <code>\$s.field</code> . Value-typed and deep- <code>const</code> . No methods (see docs/technical/rejected.md).
task / tasks	type	A <code>task of T</code> 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 <code>task</code> library.
throw	language	The <code>throw EXPR;</code> statement that raises a catchable error (convention: the <code>Error</code> struct).

Term (singular / plural)	Kind	Meaning
		Caught by <code>try / catch</code> . <i>Not</i> control flow - <code>exit / return / break / continue</code> are not catchable.
<code>try ... catch</code>	language	<code>try { body } catch (NAME) { handler }</code> runs <code>body</code> , and on a <code>throw</code> (or a runtime error, wrapped into <code>Error</code>) binds the value to <code>NAME</code> and runs <code>handler</code> . No <code>finally</code> , no typed catch. <i>Not</i> “rescue”, “except”.
<code>use</code>	language	The <code>use NAME [as ALIAS];</code> statement that activates a system library’s namespace at the call site. <i>Not</i> the same as <code>include</code> (textual splice) or <code>import</code> (module load).
variable / variables	language	A mutable, typed name bound by <code>def NAME as TYPE [init EXPR];</code> . Read and written through the <code>\$</code> 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.