Advanced programming with Jennifer



A highly optimized interpreted programming language, and how to get even more speed out of it: concurrency, errors, profiling, memory.

Based on **Jennifer 0.26.0**, November 2026, by @mplx.

spawn: run work in parallel

```
def tasks as list of task of int init [];
for (def n in 1..5) {
    $tasks[] = spawn { return work($n); };
}
def results as list of int init task.waitAll($tasks);

serial 370 ms, spawned 122 ms, same: true
```

`spawn` returns a `task of T` at once; `task.waitAll` keeps list order.

A spawn gets its own copy

```
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];
};
$xs[] = 99;
io.printf("%d %d\n", task.wait($t), len($xs));   # 6 4
```

No shared memory, no locks: data races are impossible by construction.

Channels stream values between tasks

```
def ch as channel of int init channel.make(4);
def producer as task of int init spawn {
    for (def n in 1..6) { channel.send($ch, $n * $n); }
    channel.close($ch);
    return 0;
};
def sum as int init 0;
while (true) {
    # recv throws once the channel is closed and drained
    try { $sum = $sum + channel.recv($ch); } catch (e) { break; }
}
```

Cancellation and deadlines

```
task.cancel($worker);                # worker catches kind "cancelled"
try {
    task.waitTimeout($slow, 200);  # give up after 200 ms
} catch (e) {
    io.printf("gave up: %s\n", $e.kind);   # timeout
    task.cancel($slow);
    task.discard($slow);
}
```

```
worker stopped: cancelled after 3 rounds
gave up: timeout
```

try / catch: decide by kind

```
try {
    return json.decode($text);
} catch (e) {
    if ($e.kind == "limit") {
        io.printf("too big or too deep: %s\n", $e.message);
    } else {
        io.printf("%s: %s\n", $e.kind, $e.message);
    }
}
```

```
runtime: json.decode: unexpected end of input at line 1, column 13
too big or too deep: json.decode: nesting exceeds 1000 levels at line
1, column 1001
```

Error kinds you can rely on

Kind	Means
limit	a size, count or depth cap was reached
timeout	a deadline passed
closed	the other side closed, also mid-read
cancelled	the task was cancelled
net	a socket failed: net.NetError{reason, sent, replied}

Libraries with data to report throw a struct: `rest.HttpError` carries the status, `smtp.SmtpError` the reply code.

defer and errdefer: clean up on the way out

```
func setup(fail as bool) {
    defer io.printf("setup done\n");   # always
    io.printf("open\n");
    errdefer undo("open");             # only on error
    io.printf("reserve\n");
    errdefer undo("reserve");
    if ($fail) { throw Error{kind: "value", message: "no config"}; }
    return "ready";
}
```

- `setup(false)`: open, reserve, setup done, returns `"ready"`
- `setup(true)`: open, reserve, **undo reserve, undo open**, setup done, throws

Measure first: jennifer profile

```
$ jennifer profile concat.j
   HITS           SELF            CUM  POSITION
      1   33.861817ms  1.585271611s  concat.j:8:1
  20000  1.551409794s  1.551409794s  concat.j:9:5
      1    10.91819ms    78.20022ms  concat.j:13:1
  20000    67.28203ms    67.28203ms  concat.j:14:5
```

Two loops, 20,000 rounds each. One line takes 95% of the time.

Why + in a loop is slow

```
def out as string init "";
for (def i in 0..$n) {
    $out = $out + "line\n";    # copies all of $out, every round
}
```

Lines	+ in a loop	append, then join
20,000	976 ms	68 ms
40,000	4,872 ms	101 ms

Twice the lines, five times the time: quadratic, $O(n^2)$.

Append to a list, join once

```
def lines as list of string init [];
for (def i in 0..$n) {
    $lines[] = "line";            # grows in place: amortised O(1)
}
def out as string init strings.join($lines, "\n") + "\n";
```

- `$xs[] = item;` appends in place; `lists.push($xs, item)` builds a copy
- the same for `bytes`: append with `$b[] = byte;`
- must you use +? group the small part: `$out + ($k + ": " + $v + "\n")`

profile --allocs: what was copied

```
func total(xs as list of int) { ... }                    # reads only
func cellOf(rows as list of list of int, r as int, c as int) { ... }
func cell(m as Matrix, r as int, c as int) { ... }       # rows in a
struct

$ jennifer profile --allocs borrow.j
Eager copies - a def / assignment / parameter binding that deep-copied
a compound value:
  COUNT     BYTES  TYPE                    POSITION
    300   3.5 GiB  list of list of int  borrow.j:32:22
```

Each was called 300 times. Only `cellOf` copied anything.

Copied or borrowed?

- **Borrowed:** a parameter the function never writes, for `list`, `map`, `bytes` and structs
- **Updated in place:** `$grid = put($grid, x, y, v);` on a local, outside `try`
- **Copied every call:**
  - nested collections (`list of list`, `map of K to list`): wrap them in a struct
  - functions that use callbacks (`lists.map`, `sortBy`, ...) when the program has mutable globals
  - module calls, for values holding the module's own structs
- **Copied once:** a `spawn` snapshot, a `channel.send`

More tuning, from the docs

- **Hot calls out of the innermost loop:** inline or batch a helper called per element; a highlighter got 1.37x faster this way
- **Bytes work belongs in binary:** one native scan instead of a per-byte loop
- **Lift loop-invariant work** above the loop
- **Everywhere else, keep the small helpers:** call overhead only shows in the hottest loop, and the profiler tells you which one that is

Read more

- Concurrency: `docs/user-guide/concurrency.md`
- Performance: `docs/user-guide/performance.md`
- Profiler: `docs/technical/cli_profile.md`
- Language: jennifer-lang.dev
- This talk: in the Jennifer Academy, made with cantrip

Based on Jennifer 0.26.0, November 2026.