Jennifer for the classroom



A modern interpreted language that explains itself.

> It also suits teaching and learning: an interactive REPL, an easy-to-read grammar, and token and AST dumps show what an interpreter does at each stage, 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, errors are positioned) turns a mistake into a clear message instead of silent misbehavior.

by @mplx

An interactive REPL

```
$ jennifer repl
>>> def x as int init 6 * 7;
>>> $x;
42
>>> $x > 40;
true
```

Try a line, see the value, change it, try again.

A grammar you can read aloud

```
use io;

def scores as list of int init [72, 88, 95];
def total as int init 0;
for (def s in $scores) {
    $total = $total + $s;
}
io.printf("average: %f|prec=1\n", $total / len($scores));
```

- `def name as type init value`: every variable says what it is
- `$name` reads a variable, so reading and declaring never look alike
- `/` is true division and gives a float (`85.0`); `//` floors to an int

A linter that says what and why

```
$ jennifer lint average.j
average.j:2:1: warning: unknown library `strngs` - did you mean
`strings`? [L115]
average.j:5:5: warning: local `count` is declared but never used [L101]
average.j:11:5: warning: unreachable code after `return` [L102]
```

Built in, no setup. Every finding has a position, a reason and a rule code.

One format for everyone

```
func grade(points as int){if($points>=90){return "A";}
    return "B";}

$ jennifer fmt grade.j
func grade(points as int) {
    if ($points >= 90) {
        return "A";
    }
    return "B";
}
```

`fmt -w` rewrites the file; `fmt -l` lists unformatted files and fails.

A profiler for "why is it slow?"

```
$ jennifer profile fib.j
   HITS          SELF          CUM  POSITION
      1    558.476µs  78.105093ms  fib.j:8:1
  10945  62.804101ms  77.426766ms  fib.j:6:5
  21891   11.71869ms  14.742516ms  fib.j:3:5
  10946   3.023826ms   3.023826ms  fib.j:4:9
```

Hits and time for every statement: recursion you can count.

Tests are part of the language

```
use testing;

func testBoundary() {
    testing.assertEqual(grade(75), "B");
}
```

```
$ jennifer test grade_test.j
PASS testBoundary (0 ms)
FAIL testFail (0 ms)
     [assertion] grade_test.j:12:5 assertEqual: C != B
2 passed, 1 failed, 3 total
```

Conditions must be bool

```
def count as int init 3;
if ($count) {
    io.printf("non-empty\n");
}
```

```
runtime error at cond.j:3:5: `if` condition must be bool, got int
  if ($count) {
      ^
```

Write the comparison you mean: `if ($count > 0)`.

Conversions are spelled out

```
def total as int init 10;
def label as string init "total: " + $total;

runtime error at conv.j:3:26: operator + requires numeric operands,
got string and int

def label as string init "total: " + convert.toString($total);
```

Formatting you can read

```
C:         printf("%-8s %6d %.1f%%", ...)
Jennifer:  io.printf("%s|pad=8 %d|pad=6 %f|prec=1%%", ...)
Both:      Ada           42 97.2%
```

`%d|pad=5|fill=0|sign=always` prints `+0042`; `%d|base=16|group=4|sep=_` prints `dead_beef`.

Names never shadow

```
def name as string init "Ada";
func greet() {
    def name as string init "Grace";
}
```

```
parse error at shadow.j:4:5: name "name" is already defined in an
enclosing scope
      def name as string init "Grace";
      ^
```

Errors are positioned

Every message on these slides had the same shape:

- **file:line:column**: where it happened
- **what**: the rule that was broken, in plain words
- **the line itself**, with a caret under the spot

A mistake becomes a clear message instead of silent misbehavior.

Tokens: what the lexer sees

```
$ jennifer tokens hello.j
2:1   DEF
2:5   IDENT     "n"
2:7   AS
2:10  INT_TYPE
2:14  INIT
2:19  INT       "2"
2:21  PLUS
2:23  INT       "3"
2:24  SEMI
```

The first stage of every interpreter, with line and column for each token.

The AST: what the parser builds

```
$ jennifer ast hello.j
"init": {
  "type": "BinaryExpr",
  "line": 2, "col": 19,
  "op": "+",
  "left":  { "type": "IntLit", "value": 2 },
  "right": { "type": "IntLit", "value": 3 }
}
```

The same line as a tree, as JSON, ready for any tool to read.

Try it

- Language: jennifer-lang.dev
- Source: github.com/jennifer-language/jennifer
- More talks: academy.jennifer-lang.dev
- This talk: Markdown in the Jennifer Academy, made with cantrip

```
jennifer repl
```

Based on Jennifer 0.26.0, November 2026.