How to write a programming language and shell in Go with 92% test coverage and instant CI/CD

Qi Xiao (xiaq)

2024-11-12 @ GoLab

Intro

- About myself
- The programming language and shell this talk is about
  - Elvish https://elv.sh
- Like bash / zsh / …, but more modern
  - More powerful interactive features
  - Full-fledged programming language
  - Other modern shells: Nushell, Oils, Murex
- Why make a shell?
  - Make my own tool
  - Help others make their own tools

Full-fledged programming language

- Some think advanced programming features and shell scripting are incompatible
- But real programming features are great for shell scripting!

```
# [foo bar] - list
# [&key=value] - map
var hosts = [[&name=a &cmd='apt update']
             [&name=b &cmd='pacman -Syu']]
# peach = "parallel each"
# {|h| ...} - lambda
peach {|h| ssh root@$h[name] $h[cmd] } $hosts
```

- Elvish has all the familiar shell features too

```
vim main.go
cat *.go | wc -l
# Elvish also supports recursive wildcards
cat **.go | wc -l
```

Interactive features

- Great out-of-the-box experience (demo)
  - Syntax highlighting
  - Completion with Tab
  - Directory history with Ctrl-L
  - Command history with Ctrl-R
  - Filesystem navigator with Ctrl-N
- Programmable

  ```
  set edit:prompt = { print (whoami)@(tilde-abbr $pwd)'$ ' }
  ```
- Soon the entire UI will be programmable with a new TUI framework

Implementing the Elvish interpreter

Interpreter basics

- All interpreters are alike
  - Parsing text → parse tree
  - Optionally compiling: parse tree → internal representation
  - Executing parse tree / internal representation
  - Runtime support: builtin data types, standard library
- Shells are *a bit* different:
  - External commands
  - Pipelines
  - Consider:

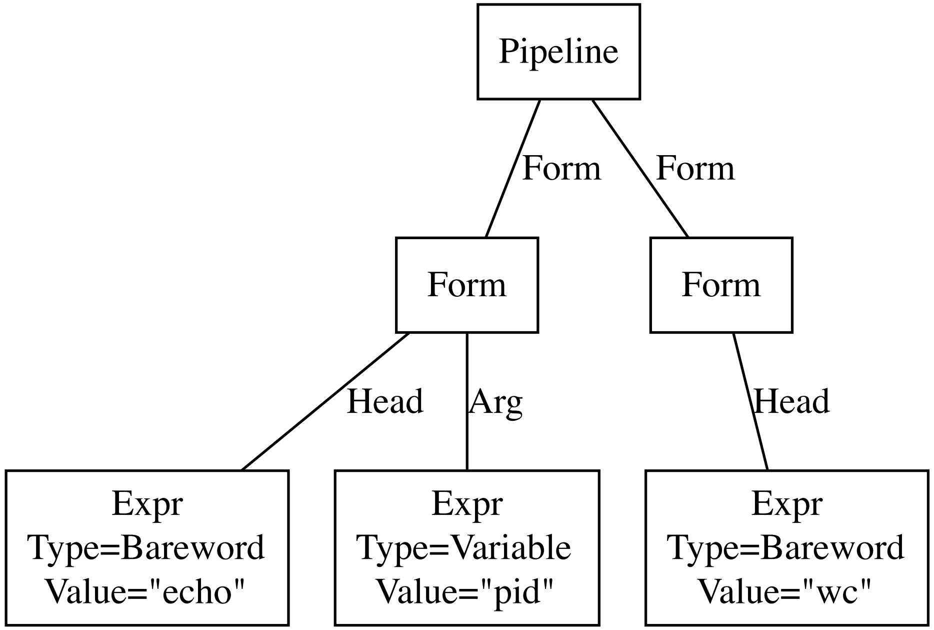
    ```
    echo $pid | wc
    ```

Parsing and “compiling”

- Source code

  ```
  echo $pid | wc
  ```

- Syntax tree:



Pipeline

Form

Form

Form

Form

Head

Arg

Head

Expr
Type=Bareword
Value="echo"

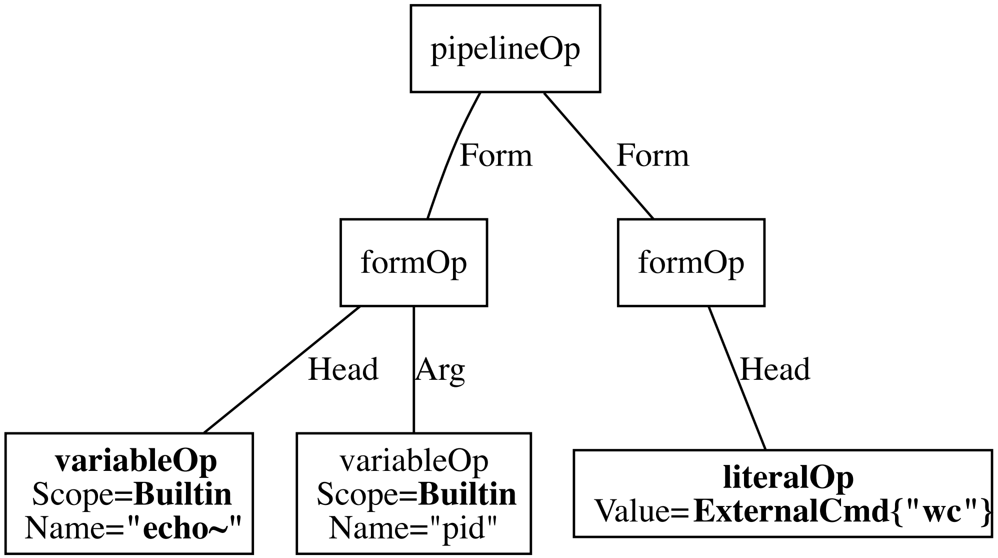
Expr
Type=Variable
Value="pid"

Expr
Type=Bareword
Value="wc"

- Source code

  ```
  echo $pid | wc
  ```

- Op tree:



pipelineOp

Form

Form

formOp

formOp

Head

Arg

Head

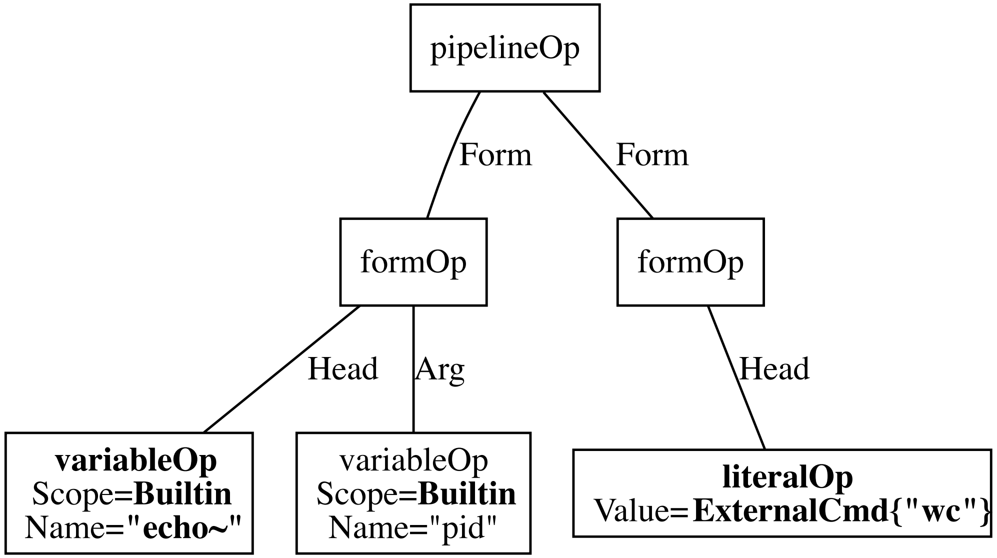
variableOp
Scope=**Builtin**
Name=**"echo~"**

variableOp
Scope=**Builtin**
Name="pid"

literalOp
Value=**ExternalCmd{"wc"}**

Execution

- `echo $pid | wc`



pipelineOp

Form

Form

formOp

formOp

Head

Arg

Head

variableOp
Scope=**Builtin**
Name="**echo~**"

variableOp
Scope=**Builtin**
Name="pid"

literalOp
Value=**ExternalCmd**{"**wc**"}

-
  ```
  type pipelineOp struct { formOps []formOp }
  func (op *pipelineOp) exec() { ... }

  type formOp struct { ... }
  func (op *formOp) exec() { ... }
  ```

- The `echo` command and the `wc` command execute within different **contexts**:

  ```
  type Context struct {
      stdinFile *os.File
      stdinChan <-chan any
      stdoutFile *os.File
      stdoutChan chan<- any
  }

  func (op *pipelineOp) exec(*Context) { ..
  func (op *formOp) exec(*Context) { ... }
  ```

Executing a pipeline

```
type pipelineOp struct { forms []formOp }

func (op *pipelineOp) exec(ctx *Context) {
    form1, form2 := forms[0], forms[1] // Assume 2 forms
    r, w, _ := os.Pipe()               // Byte pipeline
    ch := make(chan any, 1024)         // Channel pipeline
    ctx1 := ctx.cloneWithStdout(w, ch) // Context for form 1
    ctx2 := ctx.cloneWithStdin(r, ch)  // Context for form 2
    var wg sync.WaitGroup              // Now execute them in parallel!
    wg.Add(2)
    go func() { form1.exec(ctx1); wg.Done() }()
    go func() { form2.exec(ctx2); wg.Done() }()
    wg.Wait()
}
```

- Real code

Data types

- Go `bool` and `string`
- Numbers: Go's primitive number types (`int`, `float64`) and big number types (`big.Int`, `big.Rat`):

  ```
  ~> * (range 1 41) # 40!
  ▶ (num 815915283247897734345611269596115894272000000000)
  ~> + 1/10 2/10
  ▶ (num 3/10)
  ```

- Elvish has its own list and map implementations (modelled after Clojure)

Standard library

- Elvish's `math:` ← Go's [`math`]:

  ```
  ~> math:log10 100
  ▶ (num 2.0)
  ```

- Elvish's `str:` ← Go's [`strings`]:

  ```
  ~> str:has-prefix foobar foo
  ▶ $true
  ```

- Elvish's `re:` ← Go's [`regexp`]:

  ```
  ~> re:match '^foo' foobar
  ▶ $true
  ```

Go is great for writing a shell

- Execution semantics
  - Pipeline: `os.Pipe`, channels, goroutines and `sync.WaitGroup`
  - Running external commands: `os.StartProcess`
- Free data types and standard library
- Free garbage collection

Testing the Elvish interpreter

Test strategy

- Testing is important
  - Gives us confidence about the correctness of the code
  - Especially when changing the code
- Most important thing about your test strategy
  - Make it *really* easy to create and maintain tests
  - Easy-to-write tests ⇒ more tests ⇒ higher test coverage
  - Elvish has 92% test coverage
- Interpreters have a super simple API!
  - Input: code
  - Output: text, values

  ```
  ~> echo hello world
  hello world
  ~> put [hello world] [foo bar]
  ▶ [hello world]
  ▶ [foo bar]
  ```

Iteration 1: table-driven tests

```go
// Simplified interpreter API
func Interpret(code string) ([]any, string)

var tests = []struct{
    code string
    wantValues []any
    wantText   string
}{
    {code: "echo foo", wantText: "foo\n"},
}

func TestInterpreter(t *testing.T) {
    for _, test := range tests {
        gotValues, gotText := Interpret(test.code)
        // Compare with test.wantValues and test.wantText
    }
}
```

Adding a test case with table-driven tests

- Steps:
  1. Implement new functionality
  2. Test manually in terminal:

     ```
     ~> str:join , [a b]
     ▶ 'a,b'
     ```

  3. Convert the interaction into a test case:

     ```
     {code: "str:join , [a b]", wantValues: []any{"a,b"}}
     ```

- Step 3 can get repetitive
  - Computers are good at repetitive tasks 🤔

Iteration 2: transcript tests

- Record terminal *transcripts* in `tests.elvts`:

  ```
  ~> str:join , [a b]
  ▶ 'a,b'
  ```

- Generate the table from the terminal transcript:

  ```
  //go:embed tests.elvts
  const transcripts string

  func TestInterpreter(t *testing.T) {
      tests := parseTranscripts(transcripts)
      for _, test := range tests { /* ... */ }
  }
  ```

- Embrace text format
  - We lose strict structure, but it doesn't matter in practice

Adding a test case with transcript tests

- Steps:

1. Implement new functionality
2. Test manually in terminal:

   ```
   ~> str:join , [a b]
   ▶ 'a,b'
   ```

3. Copy the terminal transcript into `tests.elvts`

- Copying is still work
  - What if we don't even need to copy? 🤔

Iteration 2.1: an editor extension for transcript tests

- Editor extension for `.elvts` files
  - Run code under cursor
  - Insert output below cursor
- Steps (demo):

1. Implement new functionality
2. Test manually in `tests.elvts` within the editor:

   ```
   ~> use str
   ~> str:join , [a b]
   ▶ 'a,b'
   ```

- We have eliminated test writing as a separate step during development!

Tangent: a weird dependency injection trick

You're probably familiar with dependency injection tricks like this:

```
// in foo.go
package foo
var stdout = os.Stdout
func Hello() {
    fmt.Fprintln(stdout, "Hello!")
}

// in foo_test.go
package foo
func TestHello(t *testing.T) {
    stdout = ...
    ...
}
```

What if the test is an external test? You can export `stdout`, but that makes it part of the API. Instead:

```
// foo.go is unchanged

// in testexport_test.go
package foo // an internal test file
var Stdout = &stdout

// in foo_test.go
package foo_test // an external test file
func TestHello(t *testing.T) {
    *foo.Stdout = ...
    ...
}
```

Testing the terminal app

Widget abstraction

- Like GUI apps, Elvish’s terminal app is made up of *widgets*, conceptually:

  ```
  type Widget interface {
      Handle(event Event)
      Render(width, height int) *Buffer
  }
  ```

- `Buffer`: stores *rich text* and the cursor position
- `Event`: keyboard events (among others)
- Example: `CodeArea`
  - Stores text content and cursor position
  - `Render`: writes a `Buffer` with current content and cursor
  - `Handle`:
    - `a` → insert `a`
    - `Backspace` → delete character left of cursor
    - `Left` → move cursor left

Widget API is also simple(-ish)

- Input: `Event`
- Output: `Buffer`
- But:
  - Multiple inputs and outputs, often interleaved.

    A typical test:

    1. Press `x`, press `y`, render and check
    2. Press `Left`, render and check
    3. Press `Backspace`, render and check
  - Tests end up verbose and not easy to write 😞

Leveraging Elvish and transcript tests!

- Create Elvish bindings for the widget
- Now just use Elvish transcript tests

  ```
  ~> send [x y]; render
  xy
  ~> send [Left]; render
  xy
  ~> send [Backspace]; render
  y
  ```

- Look a lot like screenshots tests!
  - With “screenshots” embedded directly in test files

Encoding text style and cursor position

Actual `render` output is slightly more sophisticated:

```
~> send [e c o]; render
```

```
eco ̲
RRR ̂
```

```
~> send [Left]; render
```

```
eco̲
RRR̂
```

```
~> send [h]; render
```

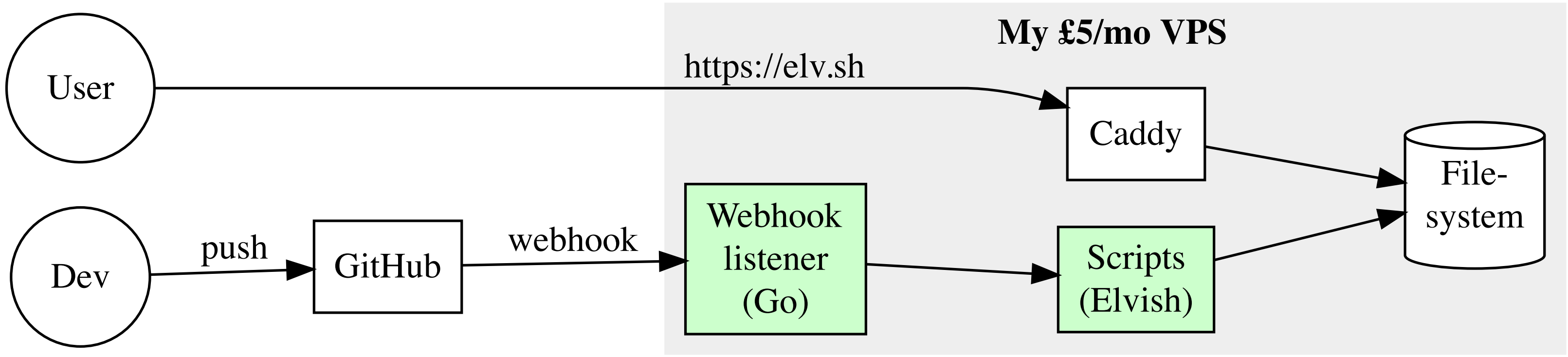
```
echo̲
GGGĜ
```

Testing strategy recap

- Make testing easy
- Embrace DSLs, embrace text
  - If DSLs don’t solve your problem, you’re not using enough of it
- Prior art: Mercurial’s tests

CI/CD

- CI just uses GitHub Actions and Cirrus CI (mostly for BSD runners)
  - You can simulate CPU architectures seamlessly with qemu + binfmt
- CD uses a custom pipeline (https://github.com/elves/up)



  - Go is a great language to write a web server with
  - Elvish is a great language for scripting
- CD builds are reproducible
  - The CI workflows also verify the reproducibility of CD builds

Learn more

- About interpreters
  - Crafting Interpreters
- Use and learn Elvish: https://elv.sh/
  - Get Elvish: https://elv.sh/get/ (one-liner installation script thanks to Go)
  - Adopting a shell is not an “all or nothing” matter
  - Try Elvish in the browser: https://try.elv.sh
- Hack on Elvish: https://github.com/elves/elvish
  - Developer docs: https://github.com/elves/elvish/tree/master/docs

Q&A