Advanced Testing Concepts for Go 1.7

Marcel van Lohuizen
Go Team @ Google
@mpvl_
github.com/mpvl

Go 1.7 is Released!

IBM z Systems port

SSA

context

hierarchical tests and benchmarks

garbage collector improvements

...

What's New in 1.7?

```
func (t *T) Run(name string, f func(t *T)) bool
```

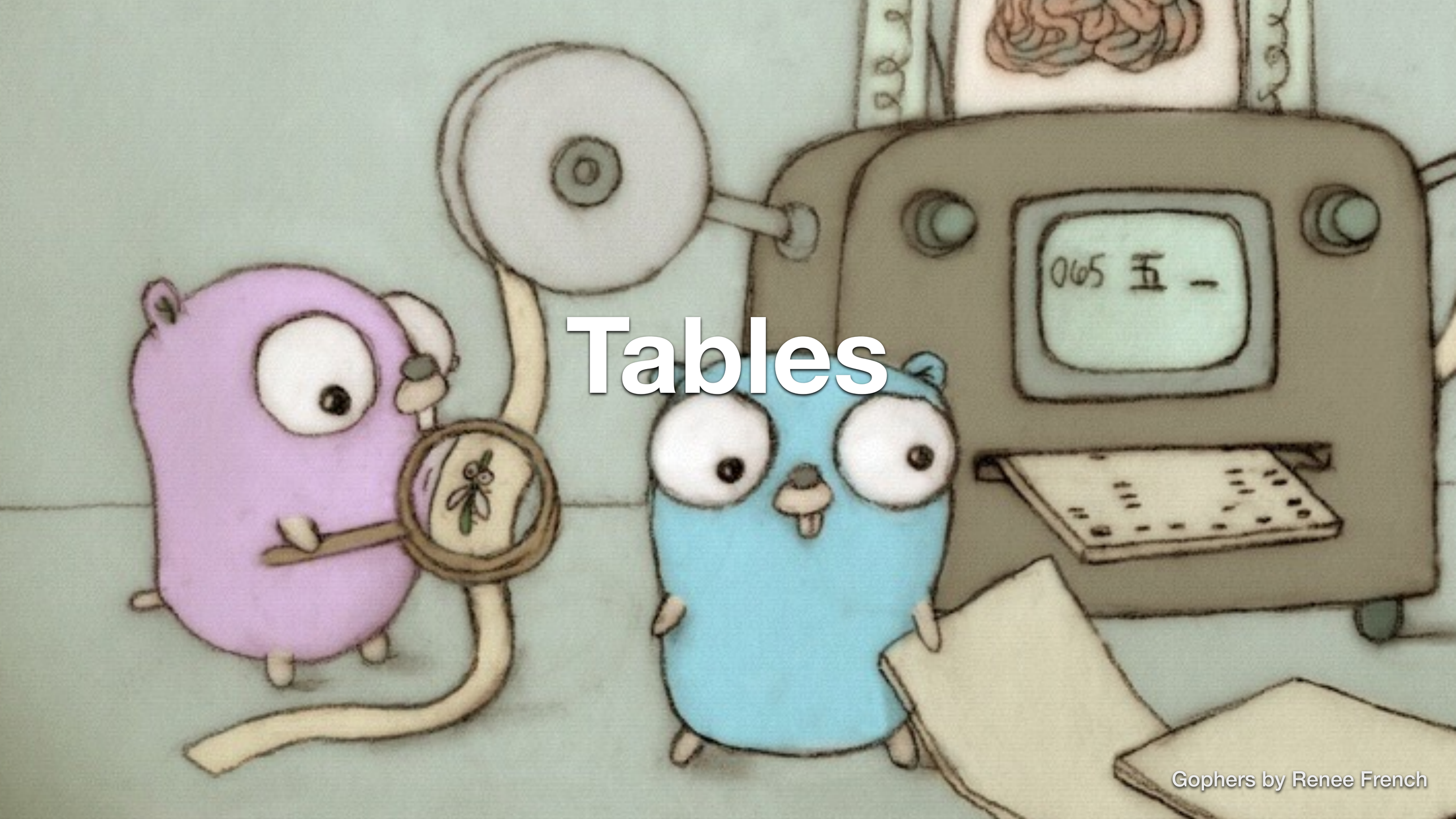
Run runs f as a subtest of t called name. It reports whether f succeeded. Run will block until all its parallel subtests have completed.

```
func (b *B) Run(name string, f func(b *B)) bool
```

Run benchmarks f as a subbenchmark with the given name. It reports whether there were any failures.

A subbenchmark is like any other benchmark. A benchmark that calls Run at least once will not be measured itself and will be called once with N=1.

Tables



Gophers by Renee French

Table-Driven Tests pre 1.7

```
func TestUpper(t *testing.T) {
   testCases := []struct { in, upper string }{
      {“Foo", "FOO"},
      {“fuß”, "FUSS"},
   }
   for _, tc := range testCases {
      if got := strings.ToUpper(tc.in); got != tc.upper {
         t.Errorf("ToUpper(%q) = %q; want %q", tc.in, got, tc.upper)
      }
   }
}
```

Benchmarks pre 1.7

```
func benchmarkAppendFloat(b *testing.B, f float64, fmt byte, prec, bitSize int) {
    dst := make([]byte, 30)
    b.ResetTimer() // Overkill here, but for illustrative purposes.
    for i := 0; i < b.N; i++ {
        AppendFloat(dst[:0], f, fmt, prec, bitSize)
    }
}

func BenchmarkAppendFloatDecimal(b *testing.B) { benchmarkAppendFloat(b, 33909, 'g', -1, 64) }
func BenchmarkAppendFloat(b *testing.B)        { benchmarkAppendFloat(b, 339.7784, 'g', -1, 64) }
func BenchmarkAppendFloatExp(b *testing.B)     { benchmarkAppendFloat(b, -5.09e75, 'g', -1, 64) }
func BenchmarkAppendFloatNegExp(b *testing.B)  { benchmarkAppendFloat(b, -5.11e-95, 'g', -1, 64) }
func BenchmarkAppendFloatBig(b *testing.B)     { benchmarkAppendFloat(b, 12345678912, 'g', -1, 64) }
    ...
```

Table-Driven Benchmarks

```
func BenchmarkAppendFloat(b *testing.B) {
    benchmarks := []struct{
        name           string
        float          float64
        fmt            byte
        prec, bitSize  int
    }{
        {"Decimal", 33909, 'g', -1, 64},
        {"Float", 339.7784, 'g', -1, 64},
        {"Exp", -5.09e75, 'g', -1, 64},
        {"NegExp", -5.11e-95, 'g', -1, 64},
        {"Big", 123456789123456789123456789, 'g', -1, 64}, …
    }
    dst := make([]byte, 30)
    for _, bm := range benchmarks {
        b.Run(bm.name, func(b *testing.B) {
            for i := 0; i < b.N; i++ {
                AppendFloat(dst[:0], bm.float, bm.fmt, bm.prec, bm.bitSize)
            }
        })
    }
}
```

Tests

Source https://commons.wikimedia.org/wiki/File:Three_point_flexural_test.jpg

Error Messages pre 1.7

```
func TestUpper(t *testing.T) {
   testCases := [][2]string{{"foo", "FOO"}, {"fuß", "FUSS"}}
   for _, tc := range testCases {
      if got := strings.ToUpper(tc[0]); got != tc[1] {
         t.Errorf("ToUpper(%q) = %q; want %q", tc[0], got, tc[1])
      }
   }
}
```

```
--- FAIL: TestUpper (0.00s)
    case_test.go:21: ToUpper("fuß") = "FUß"; want "FUSS"
```

Error Messages with Subtests

```
func TestUpper(t *testing.T) {
   testCases := [][2]string{{"foo", "FOO"}, {"fuß", "FUSS"}}
   for _, tc := range testCases {
      t.Run(tc[0], func(t *testing.T) {
         if got := strings.ToUpper(tc[0]); got != tc[1] {
            t.Errorf("got %q; want %q", got, tc[1])
         }
      })
   }
}
```

```
--- FAIL: TestUpper (0.00s)
    --- FAIL: TestUpper/fuß (0.00s)
        case_test.go:44: got "FUß"; want "FUSS"
```

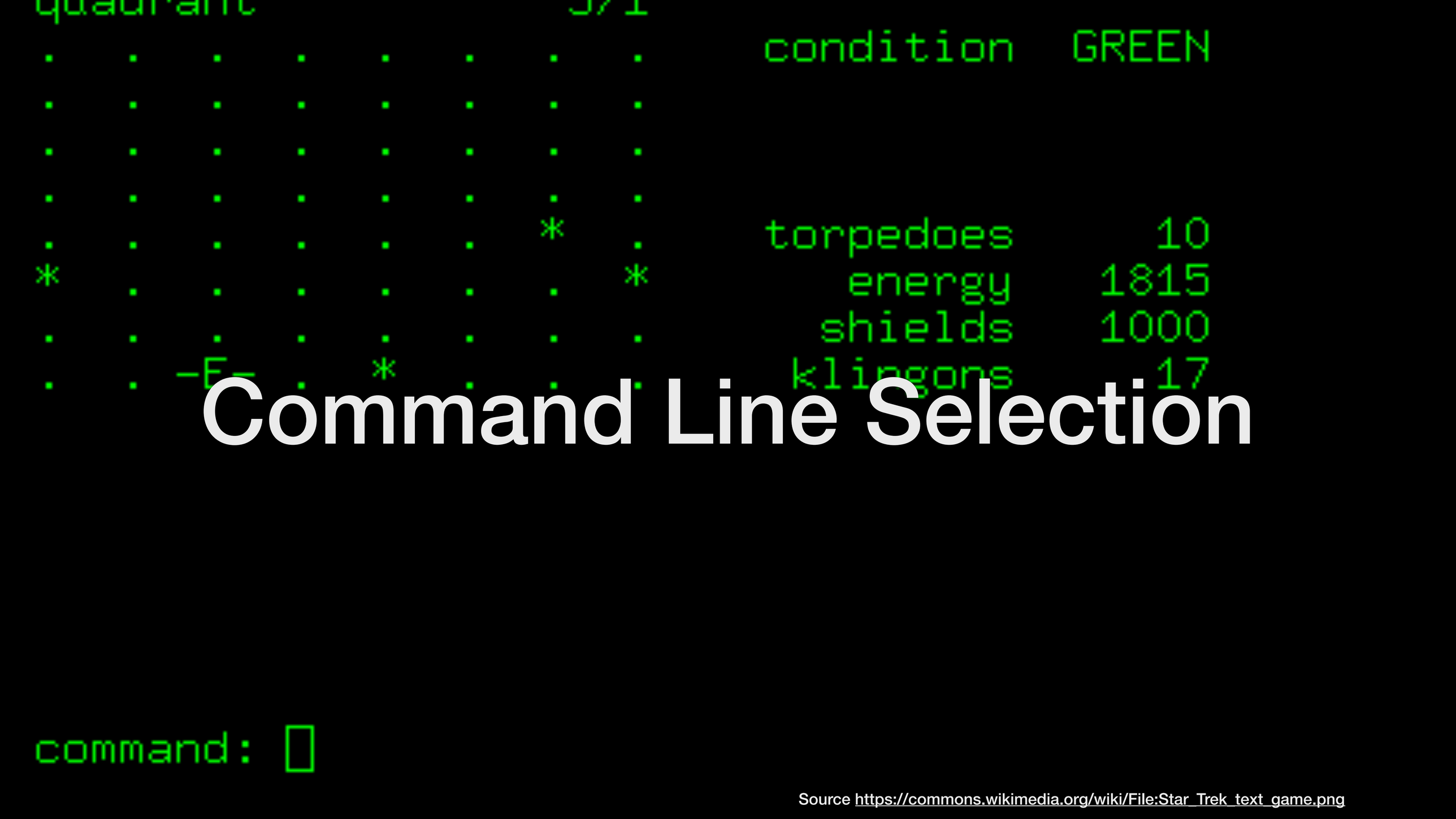
Use of Fatal and Skip pre 1.7

```
var testCases = []struct { gmt, loc, want string }{
   {"12:31", "Europe/Zuri", "13:31"},     // error: invalid Location
   {"12:31", "America/New_York", "7:31"}, // error: missing “0”
}
func TestTime(t *testing.T) {
   for _, tc := range testCases {
      loc, err := time.LoadLocation(tc.loc)
      if err != nil {
         t.Fatalf("could not load location %q", tc.loc)
      }
      gmt, _ := time.Parse("15:04", tc.gmt)
      if got := gmt.In(loc).Format("15:04"); got != tc.want {
         t.Errorf("In(%s, %s) = got %s; want %s", tc.gmt, tc.loc, got, tc.want)
      }
   }
}
```

```
--- FAIL: TestTime (0.00s)
    time_test.go:62: could not load location "Europe/Zuri"
```

```
func TestTime(t *testing.T) {
   for _, tc := range testCases {
      t.Run(fmt.Sprintf("%s in %s", tc.gmt, tc.loc), func(t *testing.T) {
         loc, err := time.LoadLocation(tc.loc)
         if err != nil {
            t.Fatal("could not load location")
         }
         gmt, _ := time.Parse("15:04", tc.gmt)
         if got := gmt.In(loc).Format("15:04"); got != tc.want {
            t.Errorf("got %s; want %s", got, tc.want)
         }
      })
   }
}
```

```
--- FAIL: TestTime (0.00s)
    --- FAIL: TestTime/12:31_in_Europe/Zuri (0.00s)
        time_test.go:84: could not load location
    --- FAIL: TestTime/12:31_in_America/New_York (0.00s)
        time_test.go:88: got 07:31; want 7:31
```



Command Line Selection

Source https://commons.wikimedia.org/wiki/File:Star_Trek_text_game.png

Selecting subtests

```
var testCases = []struct { gmt, loc, want string }{
   {"12:31", "Europe/Zuri", "13:31"},     // error: invalid Location
   {"12:31", "America/New_York", "7:31"}, // error: missing “0”
   {"08:08", "Australia/Sydney", "18:08"},
}

func TestTime(t *testing.T) {
   for _, tc := range testCases {
      t.Run(fmt.Sprintf("%s in %s", tc.gmt, tc.loc), func(t *testing.T) {
```

TestTime/12:31_in_Europe/Zuri
TestTime/12:31_in_America/New_York
TestTime/08:08_in_Australia/Sydney

Selecting subtests

`TestTime/12:31_in_Europe/Zuri`
`TestTime/12:31_in_America/New_York`
`TestTime/08:08_in_Australia/Sydney`

Run tests that use a timezone in Europe:

```
$ go test --run=TestTime/"in Europe"
--- FAIL: TestTime (0.00s)
    --- FAIL: TestTime/12:31_in_Europe/Zuri (0.00s)
        time_test.go:85: could not load location
```

Selecting subtests

TestTime/12:31_in_Europe/Zuri
TestTime/12:31_in_America/New_York
TestTime/08:08_in_Australia/Sydney

Run only tests for input times after noon:

```
$ go test --run=Time/12:[0-9] -v
=== RUN   TestTime
=== RUN   TestTime/12:31_in_Europe/Zuri
=== RUN   TestTime/12:31_in_America/New_York
--- FAIL: TestTime (0.00s)
    --- FAIL: TestTime/12:31_in_Europe/Zuri (0.00s)
        time_test.go:85: could not load location
    --- FAIL: TestTime/12:31_in_America/New_York (0.00s)
        time_test.go:89: got 07:31; want 7:31
```

Selecting subtests

TestTime/12:31_in_Europe/Zuri
TestTime/12:31_in_America/New_York
TestTime/08:08_in_Australia/Sydney

Run tests for New York time:

```
$ go test --run=Time//New_York
--- FAIL: TestTime (0.00s)
    --- FAIL: TestTime/12:31_in_America/New_York (0.00s)
        time_test.go:88: got 07:31; want 7:31
```

Selecting subtests

```
t.Run(path.Join(tc.gmt, tc.loc), func(t *testing.T) {
```

TestTime/12:31/Europe/Zuri
TestTime/12:31/America/New_York
TestTime/08:08/Australia/Sydney

Test Names are Uniqued

```
func TestDouble(t *testing.T) {
    testCases := []string{
        "a", "a", "b", "b",
    }
    for _, tc := range testCases {
        t.Run(tc, func(t *testing.T) {
            t.Fail()
        })
    }
}
```

```
--- FAIL: TestDouble (0.00s)
    --- FAIL: TestDouble/a (0.00s)
    --- FAIL: TestDouble/a#01 (0.00s)
    --- FAIL: TestDouble/b (0.00s)
    --- FAIL: TestDouble/b#01 (0.00s)
```

Test Names are Unique

```
func TestFew(t *testing.T) {
   for i := 0; i < 10; i++ {
      t.Run("", func(t *testing.T) {
          t.Fail()
      })
   }
}
```

```
--- FAIL: TestFew (0.00s)
    --- FAIL: TestFew/#00 (0.00s)
    --- FAIL: TestFew/#01 (0.00s)
    --- FAIL: TestFew/#02 (0.00s)
    --- FAIL: TestFew/#03 (0.00s)
    --- FAIL: TestFew/#04 (0.00s)
    --- FAIL: TestFew/#05 (0.00s)
    --- FAIL: TestFew/#06 (0.00s)
    --- FAIL: TestFew/#07 (0.00s)
    --- FAIL: TestFew/#08 (0.00s)
    --- FAIL: TestFew/#09 (0.00s)
```

Set Up and Tear Down

Source https://commons.wikimedia.org/wiki/File:Great_Blue_Heron_with_Gopher.jpg

Common Set Up and Tear Down

```
func TestFoo(b *testing.B) {
    // <common set-up code>
    b.Run("A=1", func(t *testing.T) { ... })
    b.Run("A=2", func(t *testing.T) { ... })
    b.Run("B=1", func(t *testing.T) {
        if … { t.Fail() }
        …
    })
    // <common tear-down code>
}
```

Per-Test Set Up and Tear Down

```
func TestFoo(b *testing.B) {
    // <common set-up code>
    for _, tc := range testCases {
        b.Run(tc.name(), func(t *testing.T) {
            data := tc.setUp()
            defer tc.tearDown()

        })
    }
    // <common tear-down code>
}
```

Source https://pixabay.com/en/road-long-road-empty-asphalt-1444397/

Semantics

- Each test has a test function
- Calling t.Parallel() marks it as parallel
- Parallel tests don’t run concurrently with sequential tests
- A test does not complete until all of its subtests complete

Top-level tests are subtests of a hidden “main” test.

Limit which Tests Run in Parallel Together

```
func TestGroupedParallel(t *testing.T) {
    for _, tc := range testCases {
        tc := tc // capture range variable
        t.Run(tc.Name, func(t *testing.T) {
            t.Parallel()
            if got := foo(tc.in); got != tc.out {
                t.Errorf("got %v; want %v", got, tc.out)
            }
            ...
        })
    }
}
```

Clean up after Group of Tests

```
func TestTeardownParallel(t *testing.T) {
    // <set-up code>
    // This Run will not return until its parallel subtests complete.
    t.Run("group", func(t *testing.T) {
        t.Run("Test1", parallelTest1)
        t.Run("Test2", parallelTest2)
        t.Run("Test3", parallelTest3)
    })
    // <tear-down code>
}
```

Backwards compatibility

```
import "github.com/mpvl/subtest"

func TestFoo(t *testing.T) {
    subtest.Run(t, "", func(t *testing.T) {
        …
    })
}
```

Thank you

@mpvl_
github.com/mpvl

```
$ go test
PASS
ok      talk    1799.89s
```