Let's Go!

Mansi Sharma

January 20, 2021
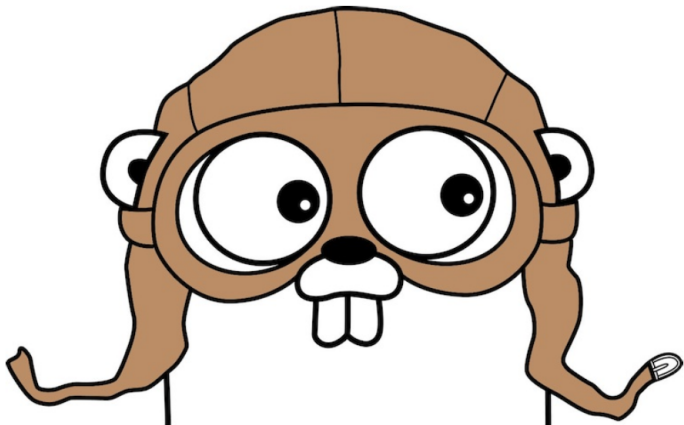
What is Go?

Go is a compiled, concurrent, garbage-collected, statically typed language developed at

see a bit of `code`!

Hello, World!

- Programs start running in package main.
- It is good style to use the factored import statement.
- A name is exported if it begins with a capital letter.

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "math"
 6  )
 7
 8  // import "fmt"
 9  // import "math"
10
11  func main() {
12      fmt.Printf("Now you have %v problems.\n", math.Sqrt(7))
13  }
14
```

Variables

- The var instruction declares a list of variables
- The type is informed at the end
- The var instruction could includes initializers, 1 per variable. In this case, the type could be ommited

```
$ go run variables.go
0 false false false
```

variables.go

```go
1  package main
2
3  import "fmt"
4
5  var c, python, java bool
6
7  func main() {
8      var i int
9      fmt.Println(i, c, python, java)
10 }
```

```
$ go run variables-with-initiali
1 2 true false no!
```

variables-with-initializers.go

```go
1  package main
2
3  import "fmt"
4
5  var i, j int = 1, 2
6
7  func main() {
8      var c, python, java = true, false, "no!"
9      fmt.Println(i, j, c, python, java)
10 }
```

Short Variable Declarations

- Inside a function, the short attribution instruction := can be used instead of a var declaration

```
$ go run short-variable-declarations.go
1 2 3 true false no!
```

```go
short-variable-declarations.go

1  package main
2
3  import "fmt"
4
5  i, j := 1, 2
6  func main() {
7      k := 3
8      c, python, java := true, false, "no!"
9
10     fmt.Println(i, j, k, c, python, java)
11 }
12
```

Constants

- Constants are declared like variables but with keyword const
- Can not use the syntx :=

```
$ go run constants.go
Hello world! Happy 3.14 Day! Go rules?

constants.go

1  package main
2
3  import "fmt"
4  const Pi = 3.14
5
6  func main() {
7      const World = "world! "
8      fmt.Print("Hello ", World)
9      fmt.Print("Happy ", Pi, " Day! ")
10
11     const Truth = true
12     fmt.Print("Go rules? ", Truth)
13 }
```

Functions

- Type comes after the parameter name, like variables
- Shorten (x int, y int) to (x, y int)

func Hello(name string) string

```
Function        Parameter        Return
name              type            type
```

```
functions.go
1  package main
2
3  import "fmt"
4
5  func add(x int, y int) int {
6      return x + y
7  }
8
9  func main() {
10      fmt.Println(add(42, 13))
11  }
12
```

```
$ go run functions.go
55
```

Multiple Return Values

- A function can have multiple return values

```
$ go run multiple-results.go
world hello
```

multiple-results.go

```go
1  package main
2
3  import "fmt"
4
5  func swap(x, y string) (string, string) {
6      return y, x
7  }
8
9  func main() {
10     a, b := swap("hello", "world")
11     fmt.Println(a, b)
12 }
13
```

Looping For

- Go has only one looping construct, the for loop
- No parentheses required, braces are always required
- The init and post statements are optional

```
$ go run for.go
45
```

for.go

```go
1  package main
2
3  import "fmt"
4
5  func main() {
6      sum := 0
7      for i := 0; i < 10; i++ {
8          sum += i
9      }
10     fmt.Println(sum)
11 }
```

```
$ go run for-continu
1024
```

for-continued.go

```go
1  package main
2
3  import "fmt"
4
5  func main() {
6      sum := 1
7      for ; sum < 1000; {
8          sum += sum
9      }
10     fmt.Println(sum)
11 }
```

For is Go's "while" and forever

- Semicolon can be removed and you will have while
- for can run forever

```
$ go run for-is-go-while.go
1024
```

```
for-is-gos-while.go
1  package main
2
3  import "fmt"
4
5  func main() {
6      sum := 1
7      for sum < 1000 {
8          sum += sum
9      }
10     fmt.Println(sum)
11 }
```

```
$ go run forever.go
process took too long
```

```
forever.go
1  package main
2
3  func main() {
4      for {
5      }
6  }
```

if Condition

- No parentheses required, braces are always required

```go
if.go
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  func sqrt(x float64) string {
9      if x < 0 {
10         return sqrt(-x) + "i"
11     }
12     return fmt.Sprint(math.Sqrt(x))
13 }
14
15 func main() {
16     fmt.Println(sqrt(2), sqrt(-4))
17 }
18
```

```go
if-with-a-short-statement.go
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  func pow(x, n, lim float64) float64 {
9      if v := math.Pow(x, n); v < lim {
10         return v
11     }
12     return lim
13 }
14
15 func main() {
16     fmt.Println(
17         pow(3, 2, 10),
18         pow(3, 3, 20),
19     )
20 }
21
```

Switch

- only runs the selected case, not all the cases that follow
- break statement is not required

```go
switch-evaluation-order.go
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func main() {
9      fmt.Println("When's Saturday?")
10     today := time.Now().Weekday()
11     switch time.Saturday {
12     case today + 0:
13         fmt.Println("Today.")
14     case today + 1:
15         fmt.Println("Tomorrow.")
16     case today + 2:
17         fmt.Println("In two days.")
18     default:
19         fmt.Println("Too far away.")
20     }
21 }
22
```

```go
switch-with-no-condition.go
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func main() {
9      t := time.Now()
10     switch {
11     case t.Hour() < 12:
12         fmt.Println("Good morning!")
13     case t.Hour() < 17:
14         fmt.Println("Good afternoon.")
15     default:
16         fmt.Println("Good evening.")
17     }
18 }
19
```

Defer

- Semicolon can be removed and you will have while
- for can run forever

```
$ go run for-is-go-while.go
1024
```

```
for-is-gos-while.go
1  package main
2
3  import "fmt"
4
5  func main() {
6      sum := 1
7      for sum < 1000 {
8          sum += sum
9      }
10     fmt.Println(sum)
11 }
```

```
$ go run forever.go
process took too long
```

```
forever.go
1  package main
2
3  func main() {
4      for {
5      }
6  }
```

Stacking Defer

- Deferred function calls are pushed onto a stack

```go
defer-multi.go

 1  package main
 2
 3  import "fmt"
 4
 5  func main() {
 6      fmt.Println("counting")
 7
 8      for i := 0; i < 10; i++ {
 9          defer fmt.Println(i)
10      }
11
12      fmt.Println("done")
13  }
14
```

```
counting
done
9
8
7
6
5
4
3
2
1
0
```

Pointers

- Synatx for pointers is the same as in C or C++
- But unlike C, Go has no pointer arithmetic.

```go
pointers.go

1  package main
2
3  import "fmt"
4
5  func main() {
6      i, j := 42, 2701
7
8      p := &i          // point to i
9      fmt.Println(*p)  // read i through the pointer
10     *p = 21          // set i through the pointer
11     fmt.Println(i)   // see the new value of i
12
13     p = &j           // point to j
14     *p = *p / 37     // divide j through the pointer
15     fmt.Println(j)   // see the new value of j
16 }
17
```

Structs

- A struct is a collection of fields
- Struct fields are accessed using a dot
- Struct fields can be accessed through a struct pointer
- You can list just a subset of fields by using the Name: syntax
- The special prefix & returns a pointer to the struct value

Structs

```go
struct-literals.go
1  package main
2
3  import "fmt"
4
5  type Vertex struct {
6      X, Y int
7  }
8
9  var (
10     v1 = Vertex{1, 2}  // has type Vertex
11     v2 = Vertex{X: 1}  // Y:0 is implicit
12     v3 = Vertex{}      // X:0 and Y:0
13     p  = &Vertex{1, 2} // has type *Vertex
14 )
15
16 func main() {
17     v1.X = 3
18     q := &v1
19     q.Y = 5
20     fmt.Println(v1, p, v2, v3)
21 }
22
```

```
{3 5} &{1 2} {1 0} {0 0}

Program exited.
```

Arrays

- The type [n]T is an array of n values of type T
- An array's length is part of its type, so arrays cannot be resized

```go
array.go
1  package main
2
3  import "fmt"
4
5  func main() {
6      var a [2]string
7      a[0] = "Hello"
8      a[1] = "World"
9      fmt.Println(a[0], a[1])
10     fmt.Println(a)
11
12     primes := [6]int{2, 3, 5, 7, 11, 13}
13     fmt.Println(primes)
14 }
15
```

```
Hello World
[Hello World]
[2 3 5 7 11 13]


Program exited.
```

Slicing

- Slicing of arrays is done the same way as in Python
- Slices are like references to arrays

```go
slices-pointers.go
1  package main
2
3  import "fmt"
4
5  func main() {
6      names := [4]string{
7          "John",
8          "Paul",
9          "George",
10         "Ringo",
11     }
12     fmt.Println(names)
13
14     a := names[0:2]
15     b := names[1:3]
16     fmt.Println(a, b)
17
18     b[0] = "XXX"
19     fmt.Println(a, b)
20     fmt.Println(names)
21 }
```

```
[John Paul George Ringo]
[John Paul] [Paul George]
[John XXX] [XXX George]
[John XXX George Ringo]

Program exited.
```

Slice Literals Length and Capacity

- A slice literal is like an array literal without the length
- The length and capacity of a slice s can be obtained using the expressions len(s) and cap(s)
- There are also many more built-in functions like make and append that can be used with slices

Slice Literals Length and Capacity

```go
slice-len-cap.go
1  package main
2
3  import "fmt"
4
5  func main() {
6      s := []int{2, 3, 5, 7, 11, 13}
7      printSlice(s)
8
9      // Slice the slice to give it zero length.
10     s = s[:0]
11     printSlice(s)
12
13     // Extend its length.
14     s = s[:4]
15     printSlice(s)
16
17     // Drop its first two values.
18     s = s[2:]
19     printSlice(s)
20  }
21
22  func printSlice(s []int) {
23      fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
24  }
```

```
len=6 cap=6 [2 3 5 7 11 13]
len=0 cap=6 []
len=4 cap=6 [2 3 5 7]
len=2 cap=4 [5 7]

Program exited.
```

Range

- The range form of the for loop iterates over a slice or map
- two values are returned for each iteration, the index and the element at that index
- If you only want the index, you can omit the second variable

Range

range-continued.go

```go
package main

import "fmt"

func main() {
    pow := make([]int, 10)
    for i := range pow {
        pow[i] = 1 << uint(i) // == 2**i
    }
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

```
2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
2**6 = 64
2**7 = 128
2**8 = 256
2**9 = 512

Program exited.
```

Maps

- A map maps keys to values
- The zero value of a map is nil. A nil map has no keys, nor can keys be added
- The make function returns a map of the given type, initialized and ready for use
- We can also initialize maps using map literals

Maps

```
maps.go
1  package main
2
3  import "fmt"
4
5  type Vertex struct {
6      Lat, Long float64
7  }
8
9  func main() {
10     m := make(map[string]Vertex)
11     m["Bell Labs"] = Vertex{40.68433, -74.39967}
12     fmt.Println(m["Bell Labs"])
13 }
14
```

```
map-literals-continued.go
1  package main
2
3  import "fmt"
4
5  type Vertex struct {
6      Lat, Long float64
7  }
8
9  var m = map[string]Vertex{
10     "Bell Labs": {40.68433, -74.39967},
11     "Google":    {37.42202, -122.08408},
12 }
13
14 func main() {
15     fmt.Println(m)
16 }
17
```

Maps

Insert or update an element in map `m`:

```
m[key] = elem
```

Retrieve an element:

```
elem = m[key]
```

Delete an element:

```
delete(m, key)
```

Test that a key is present with a two-value assignment:

```
elem, ok = m[key]
```

If `key` is in `m`, `ok` is `true`. If not, `ok` is `false`.

Function Values

- Function values may be used as function arguments and return values

```go
function-values.go

 1  package main
 2
 3  import (
 4      "fmt"
 5      "math"
 6  )
 7
 8  func compute(fn func(float64, float64) float64) float64 {
 9      return fn(3, 4)
10  }
11
12  func main() {
13      hypot := func(x, y float64) float64 {
14          return math.Sqrt(x*x + y*y)
15      }
16      fmt.Println(hypot(5, 12))
17
18      fmt.Println(compute(hypot))
19      fmt.Println(compute(math.Pow))
20  }
```

```
13

5

81


Program exited.
```

Function Closures

- A closure is a function value that references variables from outside its body

```go
function-closures.go
1  package main
2
3  import "fmt"
4
5  func adder() func(int) int {
6      sum := 0
7      return func(x int) int {
8          sum += x
9          return sum
10     }
11 }
12
13 func main() {
14     pos, neg := adder(), adder()
15     for i := 0; i < 10; i++ {
16         fmt.Println(
17             pos(i),
18             neg(-2*i),
19         )
20     }
21 }
```

```
0 0
1 -2
3 -6
6 -12
10 -20
15 -30
21 -42
28 -56
36 -72
45 -90

Program exited.
```

Questions?