Parallel Programming

Introduction to Go

Morgan Ericsson

Today

- Go
  - we go through gobyexample.com

Hello World! (again)

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello world")
7  }
```

Building and running

```
❯ go run hello.go
Hello world
❯ go build hello.go
❯ ./hello
Hello world
❯ file hello
hello: Mach-O 64-bit executable arm64
```

Values / Literals

```
 1  func main() {
 2      fmt.Println("go" + "lang")
 3
 4      fmt.Println("1+1 =", 1+1)
 5      fmt.Println("7.0/3.0 =", 7.0/3)
 6
 7      fmt.Println(true && false)
 8      fmt.Println(true || false)
 9      fmt.Println(!true)
10  }
```

Values / Literals

- Literals are not typed
  - but there are still rules
- Ok
  - `5.45 * 4`
  - `'a' + 10` (but probably not what you want)
- Not ok
  - “Hello” * 5
  - Overflow (!!!)

Variables

```
 1  func main() {
 2      var a = "initial"
 3      fmt.Println(a)
 4
 5      var b, c int = 1, 2
 6      fmt.Println(b, c)
 7
 8      var d = true
 9      fmt.Println(d)
10  }
```

Predeclared types

- `bool`
- Many different integers, e.g., `int8`, `uint16`, etc.
  - `u` for unsigned, number for number of bits
  - Also `byte` (`uint8`), `int` (`int32` or `int64`), and `uint`
  - `rune` (`int32`)
- `float32` and `float64` for floating point numbers
- `complex64` and `complex128` for complex numbers
- `string`

`var` vs `:=`

- `var <name> <type> = <value>`
  - `var x int`
  - `var x = 10`
  - Type is required if no value
- `<name> := <value>`
  - `x := 10`
  - At least one variable on the left-hand side must be “new”

`var` vs `:=`

- `:=` is generally used, but…
- `:=` can only be used inside functions
  - but you should generally avoid declaring (mutable) variables outside of functions
- `:=` assumes default type, must use `var` if you want another type
  - You can convert, but… (`x := byte(20)`)
- `:=` can shadow variables rather than reuse them
  - Blocks

Zero value

- Any variable that is declared but not assigned a value defaults to the zero value
- Different for different types, e.g., 0 for int, “” for strings, and nil for many other things
- Use `var` if you want the zero value rather than, e.g., `m := 0`

Variables (cont'd)

```
1  func main() {
2      var e int
3      fmt.Println(e)
4
5      f := "apple"
6      fmt.Println(f)
7  }
```

Constants

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "math"
 6  )
 7
 8  const s string = "constant"
 9
10  func main() {
11      fmt.Println(s)
12
13      const n = 500000000
14
15      const d = 3e20 / n
16      fmt.Println(d)
17
18      fmt.Println(int64(d))
19
20      fmt.Println(math.Sin(n))
21  }
```

Constants

- Very limited, basically a way to name values
- Can only hold things that can be determined at compile time, e.g.,
  - Values of predeclared types
  - Expressions with operators and known values
- Can be typed

Groups

- Things, e.g., variable and constant declarations can be grouped with ( )
- Good practice to group related things

Loops

```
1  func main() {
2      for j := 7; j <= 9; j++ {
3          fmt.Println(j)
4      }
5  }
```

No while, just for

```
1  func main() {
2      i := 1
3      for i <= 3 {
4          fmt.Println(i)
5          i = i + 1
6      }
7  }
```

Break and continue

```
 1  func main() {
 2      for {
 3          fmt.Println("loop")
 4          break
 5      }
 6
 7      for n := 0; n <= 5; n++ {
 8          if n%2 == 0 {
 9              continue
10          }
11          fmt.Println(n)
12      }
13  }
```

If

```
1  func main() {
2      if 8%4 == 0 {
3          fmt.Println("8 is divisible by 4")
4      }
5
6      if 8%2 == 0 || 7%2 == 0 {
7          fmt.Println("either 8 or 7 are even")
8      }
9  }
```

Else

```
1  func main() {
2      if 7%2 == 0 {
3          fmt.Println("7 is even")
4      } else {
5          fmt.Println("7 is odd")
6      }
7  }
```

If again

```
1  func main() {
2      if num := 9; num < 0 {
3          fmt.Println(num, "is negative")
4      } else if num < 10 {
5          fmt.Println(num, "has 1 digit")
6      } else {
7          fmt.Println(num, "has multiple digits")
8      }
9  }
```

Switch

```
 1  func main() {
 2      i := 2
 3      fmt.Print("Write ", i, " as ")
 4      switch i {
 5      case 1:
 6          fmt.Println("one")
 7      case 2:
 8          fmt.Println("two")
 9      case 3:
10          fmt.Println("three")
11      }
12  }
```

Default

```
1  func main() {
2      switch time.Now().Weekday() {
3      case time.Saturday, time.Sunday:
4          fmt.Println("It's the weekend")
5      default:
6          fmt.Println("It's a weekday")
7      }
8  }
```

“Empty” switch

```
1  func main() {
2      t := time.Now()
3      switch {
4      case t.Hour() < 12:
5          fmt.Println("It's before noon")
6      default:
7          fmt.Println("It's after noon")
8      }
9  }
```

Arrays

```
 1  func main() {
 2      var a [5]int
 3      fmt.Println("emp:", a)
 4
 5      a[4] = 100
 6      fmt.Println("set:", a)
 7      fmt.Println("get:", a[4])
 8
 9      fmt.Println("len:", len(a))
10  }
```

Arrays

- Note that both type and length are part of the type of the array
- So, fixed in length, just like, e.g., Java
- Can be annoying
  - E.g., requires copying to grow/shrink

Declaring values

```
1  func main() {
2      b := [5]int{1, 2, 3, 4, 5}
3      fmt.Println("dcl:", b)
4  }
```

More dimensions

```
 1  func main() {
 2      var twoD [2][3]int
 3
 4      for i := 0; i < 2; i++ {
 5          for j := 0; j < 3; j++ {
 6              twoD[i][j] = i + j
 7          }
 8      }
 9      fmt.Println("2d: ", twoD)
10  }
```

Slices (lists)

```
1 func main() {
2     var s []string
3     fmt.Println("uninit:", s, s == nil, len(s) ==
4
5     s = make([]string, 3)
6     fmt.Println("emp:", s, "len:", len(s), "cap:",
7 }
```

Make, length, and capacity

- We use make to make “things,” e.g., slices and channels
  - Allocates memory, does setup, and sets default values
- A slice has a length and a capacity
  - Length is the number of values in the slices
  - Capacity is the number of values the slice can hold
  - This is an important distinction!

Make, length, and capacity

```
1 func main() {
2     var a []int
3     fmt.Println("a:", a, "len:", len(a), "cap:", c
4     a = append(a, 10)
5     fmt.Println("a:", a, "len:", len(a), "cap:", c
6 }
```

Make, length, and capacity

```
1  func main() {
2      a := make([]int, 5)
3      fmt.Println("a:", a, "len:", len(a), "cap:", c
4      a = append(a, 10)
5      fmt.Println("a:", a, "len:", len(a), "cap:", c
6  }
```

Make, length, and capacity

```
1  func main() {
2      a := make([]int, 5, 10)
3      fmt.Println("a:", a, "len:", len(a), "cap:", c
4      a = append(a, 10)
5      fmt.Println("a:", a, "len:", len(a), "cap:", c
6  }
```

Slices

```
1  func main() {
2      s := make([]string, 3)
3
4      c := make([]string, len(s))
5      copy(c, s)
6      fmt.Println("cpy:", c)
7  }
```

Slices

```
 1  func main() {
 2      s := []string{"a", "b", "c", "d", "e", "f"}
 3
 4      l := s[2:5]
 5      fmt.Println("sl1:", l)
 6
 7      l = s[:5]
 8      fmt.Println("sl2:", l)
 9
10      l = s[2:]
11      fmt.Println("sl3:", l)
12  }
```

Slices

- There are many useful functions on slices in the `slices` package
  - equals, max, min, …
  - sort, search, …
  - replace, reverse, …

Slices

```
1  func main() {
2      t := []string{"g", "h", "i"}
3      fmt.Println("dcl:", t)
4
5      t2 := []string{"g", "h", "i"}
6      if slices.Equal(t, t2) {
7          fmt.Println("t == t2")
8      }
9  }
```

Multiple dimensions

```
 1  func main() {
 2      twoD := make([][]int, 3)
 3      for i := 0; i < 3; i++ {
 4          innerLen := i + 1
 5          twoD[i] = make([]int, innerLen)
 6          for j := 0; j < innerLen; j++ {
 7              twoD[i][j] = i + j
 8          }
 9      }
10      fmt.Println("2d: ", twoD)
11  }
```

Maps

```
1  func main() {
2      m := make(map[string]int)
3
4      m["k1"] = 7
5      m["k2"] = 13
6
7      fmt.Println("map:", m, "len:", len(m))
8  }
```

Maps

```
1  func main() {
2      m := make(map[string]int)
3      v1 := m["k1"]
4      fmt.Println("v1:", v1)
5
6      v2, here := m["k1"]
7      fmt.Println("v2:", v1, "here:", here)
8  }
```

Maps

```go
func main() {
    m := make(map[string]int)

    m["k1"] = 7
    m["k2"] = 13
    delete(m, "k2")
    fmt.Println("map:", m)

    clear(m)
    fmt.Println("map:", m)
}
```

Maps

```
1  func main() {
2      n := map[string]int{"foo": 1, "bar": 2}
3      fmt.Println("map:", n)
4
5      n2 := map[string]int{"foo": 1, "bar": 2}
6      if maps.Equal(n, n2) {
7          fmt.Println("n == n2")
8      }
9  }
```

Range (like Python)

```
1  func main() {
2      nums := []int{2, 3, 4}
3      sum := 0
4      for _, num := range nums {
5          sum += num
6      }
7      fmt.Println("sum:", sum)
8  }
```

Range (enumerate)

```
1  func main() {
2      for i, num := range nums {
3          if num == 3 {
4              fmt.Println("index:", i)
5          }
6      }
7  }
```

Range

- Note that you can omit later values
  - `for i := range nums {}`
  - Omits the copy of the value
- But only later values, so
  - `for _, num := range nums {}`
  - to get a copy of the value but not the index

Range (items)

```
1 func main() {
2     kvs := map[string]string{"a": "apple", "b": "b
3     for k, v := range kvs {
4         fmt.Printf("%s -> %s\n", k, v)
5     }
6 }
```

Range (keys)

```
1  func main() {
2      for k := range kvs {
3          fmt.Println("key:", k)
4      }
5  }
```

Range on strings

```
1  func main() {
2      for i, c := range "go" {
3          fmt.Println(i, c)
4      }
5  }
```

Functions

```
1  func plus(a int, b int) int {
2      return a + b
3  }
4
5  func main() {
6      res := plus(1, 2)
7      fmt.Println("1+2 =", res)
8  }
```

Functions

```
1 func plusPlus(a, b, c int) int {
2     return a + b + c
3 }
```

Multiple return values

```
1  func vals() (int, int) {
2      return 3, 7
3  }
4
5  func main() {
6      a, b := vals()
7      _, c := vals()
8  }
```

Return values can be named

```
1 func vals() (x, y int) {
2     x = 3
3     y = 7
4     return x, y
5 }
```

Return values can be named

- The return variables are declared automatically
- It is possible to use a blank return
  - Latest assigned value is returned
- but considered bad practice since it can be confusing

Variadic functions

```
 1  func sum(nums ...int) int {
 2      total := 0
 3
 4      for _, num := range nums {
 5          total += num
 6      }
 7
 8      return total
 9  }
10
11  func main() {
12      _ = sum(1, 2)
13      _ = sum(1, 2, 3)
14  }
```

Variadic functions

- Allows us to accept any number of parameters to a function
  - Used by, e.g., `Println`
- Accessible as a slice
- Similar to passing a slice

Splat (unpacking)

```
1  func main() {
2      nums := []int{1, 2, 3, 4}
3      _ = sum(nums...)
4  }
```

Anonymous functions

```
1  func main() {
2      func() { fmt.Println("Anon!") }()
3  }
```

Anonymous functions

```
1  func main() {
2      a := 20
3      f := func() {
4          fmt.Println("a =", a)
5          a = 30
6      }
7      f()
8      fmt.Println("a =", a)
9  }
```

Passing functions

```
 1  func runme(f func(int, int) int) {
 2      fmt.Println(f(1, 2))
 3  }
 4
 5  func main() {
 6      add := func(a, b int) int {
 7          return a + b
 8      }
 9      runme(add)
10  }
```

Closures

- A closure is a function together with its environment
  - Variables used by the function but declared in its enclosing scope
- The captured variables can be used even when the function is called outside their scope
- Feature of (instantiated) anonymous functions

Closures

```
 1  func intSeq(start int) func() int {
 2      return func() int {
 3          start++
 4          return start
 5      }
 6  }
 7
 8  func main() {
 9      nextInt := intSeq(0)
10
11      fmt.Println(nextInt())
12      fmt.Println(nextInt())
13      fmt.Println(nextInt())
14
15      newInts := intSeq(10)
16      fmt.Println(newInts())
17  }
```

Recursion

```
 1  func fact(n int) int {
 2      if n == 0 {
 3          return 1
 4      }
 5      return n * fact(n-1)
 6  }
 7
 8  func main() {
 9      fmt.Println(fact(8) == 40_320)
10  }
```

Pointers

```
 1  func setToZero(ival int) {
 2      ival = 0
 3  }
 4
 5  func main() {
 6      i := 1
 7      fmt.Println("i =", i)
 8      setToZero(i)
 9      fmt.Println("i =", i)
10  }
```

Call by?

- When you call a function, parameters can be passed in different ways
  - By reference
  - By value
- Go uses by value (copies)
  - That is why `i` is not modified; the function gets a copy

Pointers

```
 1  func setToZero(iptr *int) {
 2      ival = 0
 3  }
 4
 5  func main() {
 6      i := 1
 7      fmt.Println("i =", i)
 8      setToZero(&i)
 9      fmt.Println("i =", i)
10  }
```

Call by value

- The pointer's value is copied to the function
- But we use the value to find the memory location of the variable
- So, we can change it inside the function
  - `&` gets the address
  - `*` is used to define and dereference a pointer
- Explicit dereferencing is seldom needed in Go

Pointers

```
1  func main() {
2      i := 1
3      fmt.Println("Value =", i, ", Pointer =", &i)
4  }
```

Strings and runes

```
1  func main() {
2      const s = "สวัสดี"
3      fmt.Println("Len:", len(s))
4  }
```

Strings and runes

```
1  func main() {
2      const s = "สวัสดี"
3
4      for i := 0; i < len(s); i++ {
5          fmt.Printf("%x ", s[i])
6      }
7      fmt.Println()
8  }
```

Strings and runes

```
1  func main() {
2      const s = "สวัสดี"
3
4      for _, rv := range s {
5          fmt.Printf("%#U\n", rv)
6      }
7      fmt.Println()
8  }
```

Strings and runes

```
1  func isT(r rune) bool {
2      return r == 'T'
3  }
4
5  func main() {
6      _ = isT('A') != isT('T')
7  }
```

Structs

```
 1  type person struct {
 2      name string
 3      age int
 4  }
 5
 6  func main() {
 7      p1 := person{"Alice", 10}
 8      fmt.Println(p1)
 9      p2 := person{name: "Bob"}
10      p2.age = 20
11      fmt.Println(p2)
12  }
```

Structs

```
 1  func newPerson(name string, age int) *person {
 2      p := person{name: name, age: age}
 3      return &p
 4  }
 5
 6  func main() {
 7      var p3 *person
 8      p3 = newPerson("Carol", 30)
 9      fmt.Println(p3.name)
10  }
```

Structs

```
1  func newPerson(name string, age int) person {
2      return person{name: name, age: age}
3  }
```

Anonymous structs

```
 1  func main() {
 2      dog := struct {
 3          name string
 4          isGood bool
 5      }{"Rex", true}
 6
 7      if dog.isGood {
 8          fmt.Println(dog.name)
 9      }
10  }
```

Methods

```
 1  type rect struct {
 2      width, height float64
 3  }
 4
 5  func (r *rect) area() float64 {
 6      return r.width * r.height
 7  }
 8
 9  func (r rect) perim() float64 {
10      return 2 * r.width + 2 * r.height
11  }
```

Methods

```
1  func main() {
2      r := rect{width: 10.0, height: 5.0}
3      fmt.Println("Area:", r.area())
4      fmt.Println("Area:", r.perim())
5  }
```

Interfaces

```
1 type geometry interface {
2     area() float64
3     perim() float64
4 }
```

Interfaces

```
 1  func measure(g geometry) {
 2      fmt.Println(g)
 3      fmt.Println("Area:", g.area())
 4      fmt.Println("Perim:", g.perim())
 5  }
 6
 7  func main() {
 8      r := rect{width: 10.0, height: 5.0}
 9      measure(&r)
10  }
```

Interfaces

```
 1 type circle struct {
 2     radius float64
 3 }
 4
 5 func (c circle) area() float64 {
 6     return math.Pi * c.radius * c.radius
 7 }
 8
 9 func (c circle) perim() float64 {
10     return 2 * math.Pi * c.radius
11 }
12
13 func main() {
14     c := cicle{5}
15     measure(c)
16 }
```

Struct embedding

```
1 type base struct {
2     num int
3 }
4
5 func (b base) describe() string {
6     return fmt.Sprintf("base with num=%v", b.num)
7 }
```

Struct embedding

```
 1 type container struct {
 2     base
 3     str string
 4 }
 5
 6 func main() {
 7     co := container{base: base{1}, str:"my string"
 8     fmt.Println(co.base.num)
 9     fmt.Println(co.num)
10     fmt.Println(co.describe())
11 }
```

Generics

```go
func MapKeys[K comparable, V any](m map[K]V) []K {
    r := make([]K, 0, len(m))
    for k := range m {
        r = append(r, k)
    }
    return r
}

func main() {
    var m = map[int]string{1: "2", 2: "4", 4: "8"}
    fmt.Println("keys:", MapKeys(m))
    _ = MapKeys[int, string](m)
}
```

A generic linked list

```
1  type List[T any] struct {
2      head, tail *element[T]
3  }
4
5  type element[T any] struct {
6      next *element[T]
7      val T
8  }
```

A generic linked list

```
1  func (lst *List[T]) Push(v T) {
2      if lst.tail == nil {
3          lst.head = &element[T]{val: v}
4          lst.tail = lst.head
5      } else {
6          lst.tail.next = &element[T]{val: v}
7          lst.tail = lst.tail.next
8      }
9  }
```

A generic linked list

```
1  func (lst *List[T]) GetAll() []T {
2      var elems []T
3      for e:=lst.head;e!=nil;e = e.next {
4          elems = append(elems, e.val)
5      }
6      return elems
7  }
```

A generic linked list

```
1  func main() {
2      lst := List[int]{}
3      lst.Push(10)
4      lst.Push(11)
5      lst.Push(12)
6      fmt.Println("List:", lst.GetAll())
7  }
```

Errors

```
 1  package main
 2
 3  import ( "errors"; "fmt" )
 4
 5  func f1(arg int) (int, error) {
 6      if arg == 42 {
 7          return -1, errors.New("42 does not work")
 8      }
 9      return arg + 3, nil
10  }
```

Errors

```
1  func main() {
2      if r, e := f1(42); e != nil {
3          fmt.Println("f1 failed:", e)
4      } else {
5          fmt.Println("f1 worked:", r)
6      }
7  }
```

Errors

- Error is an interface so we can define custom error types
- We simply define the `Error()` method

Panic

```
1  func main() {
2      if r, e := f1(42); e != nil {
3          panic(e)
4      }
5  }
```

Defer

- `defer` is used to ensure that a function call is performed later in a program's execution
  - Often used to clean up
  - Similar to `finally` in Java
- Called when the surrounding function exits
  - If multiple calls to `defer`, LIFO is used

Defer

```
 1  func myfunc() {
 2      defer fmt.Println("Exiting")
 3      fmt.Println("Entering")
 4      fmt.Println("Executing")
 5  }
 6
 7  func main() {
 8      defer fmt.Println("Done")
 9      myfunc()
10  }
```

Recover from a panic

```
1  func myfunc() int {
2      a := 0
3      return 1 / a
4  }
5
6  func main() {
7      myfunc()
8      fmt.Println("Survived function call")
9  }
```

Recover from a panic

```
 1  func myfunc() int {
 2      defer func() {
 3          if r := recover(); r != nil {
 4              fmt.Println("Recovered error:", r)
 5          }
 6      }()
 7
 8      a := 0
 9      return 1 / a
10  }
11
12  func main() {
13      myfunc()
14      fmt.Println("Survived function call")
15  }
```

Regex

- Works as expected, e.g.,
  - `regexp.MatchString()`
  - `regexp.Compile()`
- Anything other than `MatchString()` requires a compiled expression, e.g.,
  - `FindAll()`
  - `ReplaceAll()`

Regex

```
1  func main() {
2      r, _ := regexp.MustCompile("p([a-z]+)ch")
3      fmt.Println(r.MatchString("peach"))
4      fmt.Println(r.FindAllString("peach punch pinch
5  }
```

Regex

```
1  func main() {
2      qs := regexp.MustCompile(`"(.+?)"`)
3      fmt.Println(r.MatchString("peach"))
4      fmt.Println(r.FindAllString("peach punch pinch
5  }
```

Capture groups and raw strings

```
1  func main() {
2      qs := regexp.MustCompile(`"(.+?)"`)
3      fmt.Println(qs.FindStringSubmatch("A quoted string: \"dog\"")[1])
4
5  }
```

Futher readings

- Learn Go
  - A tour of Go
  - Go by example
- Documentation
- Standard library

Next time

» Computer and operating system