Syntax

Objectives

- Learn basic syntax
- Understand Go variables, functions, conditionals, and loops
- Explain scope and lifetime in Go

Comments

- Comments are ignored by the Go compiler
  - Used by humans to document and delineate
- Go supports two forms of comments
  - //
    - "Line Comments"
    - Begins a comment which terminates at the end of the line
    - Used in most cases
  - /* */
    - "Block comments"
    - Everything between the *s is part of the comment
    - Can include multiple lines
      - Often used to disable large blocks of code
    - Also useful within an expression
    - Used for package comments
      - Every package should have a package comment, a block comment preceding the package clause
      - The package comment should introduce the package and provide information relevant to the package as a whole

```
/*
Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

```
        // At this point, the initialization of etcd is done.
        // The listeners are listening on the TCP ports and ready
        // for accepting connections. The etcd instance should be
        // joined with the cluster and ready to serve incoming
        // connections.
        notifySystemd()

        select {
        case lerr := <-errc:
                // fatal out on listener errors
                plog.Fatal(lerr)
        case <-stopped:
        }

        osutil.Exit(0)
}

// startEtcd runs StartEtcd in addition to hooks needed for standalone etcd.
func startEtcd(cfg *embed.Config) (<-chan struct{}, <-chan error, error) {
        if cfg.Metrics == "extensive" {
                grpc_prometheus.EnableHandlingTimeHistogram()
        }
```

Declarations

- A Go program consists principally of 5 types of declarations:
  - Packages – package
  - Functions – func
  - Variables – var
  - Constants – const
  - Types – type
- Declarations begin with the keyword that specifies the declaration type, are followed by the declaration identifier and end with the specification of the thing
  - var x int = 5
    - Type can be inferred on initialization
  - func x …
  - const x int = 6
    - When type can be inferred idiomatic Go leaves it out
  - package sailing …

```go
package sailing

const x = 3.1415

var y int = 8

func test() (float32) {
    return x
}
```

Functions

- A function declaration consists of:
  - The keyword func
  - The name of the function
  - A parameter list (can be empty)
  - A result list (can be empty)
  - The body of the function
- The statements that define a function's behavior are enclosed in braces
- Execution of the function begins with the first statement and continues until it encounters a return statement or reaches the end of a function that has no results
  - Control and any results are then returned to the caller
- Go requires no semicolons
- Go supports:
  - first class functions
  - higher-order functions
  - user-defined
  - function types
  - function literals
  - closures
  - and multiple return values

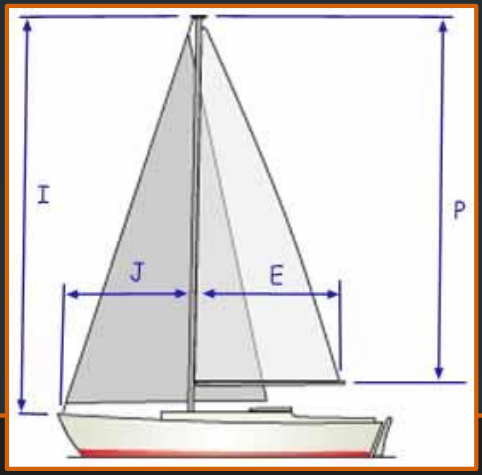
```go
// ensurePath is used to make sure a path exists
func ensurePath(path string, dir bool) error {
	if !dir {
		path = filepath.Dir(path)
	}
	return os.MkdirAll(path, 0755)
}
```

```go
// Returns if a member is a consul node. Returns a bool,
// and the datacenter.
func isConsulNode(m serf.Member) (bool, string) {
	if m.Tags["role"] != "node" {
		return false, ""
	}
	return true, m.Tags["dc"]
}
```

```go
// runtimeStats is used to return various runtime information
func runtimeStats() map[string]string {
	return map[string]string{
		"os":         runtime.GOOS,
		"arch":       runtime.GOARCH,
		"version":    runtime.Version(),
		"max_procs":  strconv.FormatInt(int64(runtime.GOMAXPROCS(0)), 10),
		"goroutines": strconv.FormatInt(int64(runtime.NumGoroutine()), 10),
		"cpu_count":  strconv.FormatInt(int64(runtime.NumCPU()), 10),
	}
}
```

Variables

- var declarations create a variable with a
  - Type
  - Identifier
  - Initial value
- General form: `var name type = expression`
  - Either type or the expression may be omitted
  - If the expression is omitted the identifier is initialized to the "zero value" for that type
    - 0 for numbers
    - False for Booleans
    - "" for strings
    - nil for nilable types
  - Go does not allow uninitialized variables
  - Suggests making the zero value a meaningful marker within your app




```go
package main

import "sailing"
import "fmt"

func main() {
    var i float32 = 77.5
    var j float32 = 23.3
    var e float32 = 21.7
    var p float32 = 74.3
    fmt.Println("Main area", sailing.CalcM(e, p))
    fmt.Println("Foretriangle", sailing.CalcFT(j, i))
    fmt.Println("Sail Area", sailing.CalcSailArea(e, p, j, i))
}
```

```go
package sailing

//CalcM returns the main sail area
func CalcM(e float32, p float32) float32 {
    return e * p / 2
}

//CalcFT returns the fore triangle (jib sail area)
func CalcFT(j float32, i float32) float32 {
    return j * i / 2
}

//CalcSailArea returns the total sail area
func CalcSailArea(e float32, p float32, j float32, i float32) float32 {
    return CalcM(e, p) + CalcFT(j, i)
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:29228
Main area 806.1551
Foretriangle 902.875
Sail Area 1709.03
```

Expressions

- Assignment
  - x = 9
  - Each arithmetic and bitwise binary operator has a corresponding combined assignment operator
    - x += 7, x *=2
  - Tuple assignment allows several variables to be assigned at once by evaluating all of the right side variables before any variables are updated
    - x, y = y, x
  - The blank identifier "_" can absorb unwanted assigned values
    - x, _ = y, x
    - _, y = Components()
- Increment/decrement
  - x++, x--
  - No prefix inc/dec
    - Less need without pointer arithmetic - https://golang.org/doc/faq#inc_dec

Binary Operators

- 5 Levels of precedence (from high to low):
  - * / % << >> & &^
  - + - | ^
  - == != < <= > >=
  - &&
  - ||
- Operators in each category are left associative
  - Parenthesis can be used to force ordering
  - mask & (1 < < 28)
- The sign of the remainder produced by % is always the same as the sign of the dividend (the number to the left of the operator)
- Integer division truncates the result toward 0
- Overflows truncate high order bits

+ and – also have unary forms
(+ has no effect and – negates the value it precedes)

& | ^ &^ << >> are bitwise
(AND, OR, XOR, AND NOT, Left Shift and Right Shift respectively)
<< fill 0, >> of unsigneds fill 0, >> of signeds fill the sign bit
^ is bitwise NOT in unary form

example.go

```go
package main

import "fmt"

func main() {
    var x uint8 = 1<<1 | 1<<5
    var y uint8 = 1<<1 | 1<<2
    fmt.Printf("%08b\n", x)
    fmt.Printf("%08b\n", y)
    fmt.Printf("%08b\n", x&y)
    fmt.Printf("%08b\n", x|y)
    fmt.Printf("%08b\n", x^y)
    fmt.Printf("%08b\n", x&^y)
    for i := uint(0); i < 8; i++ {
        if x&(1<<i) != 0 {
            fmt.Println(i)
        }
    }
    fmt.Printf("%08b\n", x<<1)
    fmt.Printf("%08b\n", x>>1)
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
API server listening at: 127.0.0.1:39274
00100010
00000110
00000010
00100110
00100100
00100000
1
5
01000100
00010001
```

Compact initializers

- Multiple variables can be initialized with a single var statement
  - If the type is omitted it is inferred from the initializer
  - This allows multiple variables of different types to be initialized on a single line
  - Multiple variables can be initialized with a function that returns the correct number of values
  - Multiple variables can also be initialized in a var block within parenthesis
- The := operator can be used to initialize variables within function definitions
  - When it can be used, this is the preferred syntax
  - If some of the left side variables already exists they are assigned to rather than created
    - At least one of the variables must be created in the expression for it to compile (otherwise assignment "=" should be used)
- The default type for floats is float64
  - You can cast these to float using the Go "type()" cast syntax
  - The var keyword is often used when an explicit non-default type is required of when there is no explicit initial value for the identifier

```go
package main

import "fmt"

func main() {
    const (
        x = 3
    )
    var (
        t = 342
        u = 42
    )
    fmt.Println(u, t, x)
}
```

```go
package main

import (
    "fmt"
    "sailing"
)

func ftm(e float32, p float32, j float32, i float32) (float32, float32) {
    return sailing.CalcFT(j, i), sailing.CalcM(e, p)
}

func main() {
    var i, j, e, p float32 = 77.5, 23.3, 21.7, 74.3
    var ft, m = sailing.CalcFT(j, i), sailing.CalcM(e, p)
    fmt.Println("FT, M", ft, m)
    var ft2, m2 = ftm(e, p, j, i)
    fmt.Println("FT, M", ft2, m2)
    fmt.Println("Main area", sailing.CalcM(e, p))
    fmt.Println("Foretriangle", sailing.CalcFT(j, i))
    fmt.Println("Sail Area", sailing.CalcSailArea(e, p, j, i))

    j2 := 24.1
    e2, p2 := 22.4, 75.9
    fmt.Println("Alt Foretriangle", sailing.CalcFT(float32(j2), i))
    fmt.Println("Alt Main", sailing.CalcFT(float32(e2), float32(p2)))
}
```

Lifetime

- The lifetime of a variable is the interval of time during which it exists as the program executes
- Package-level variables live for the entire execution of the program
- Local variables have dynamic lifetimes:
  - Live until they become unreachable
    - At which point its storage may be recycled
  - A new instance is created each time the declaration statement is executed
  - Function parameters and results are local variables
- The compiler (not the programmer) decides whether to allocate a variable on the stack or the heap
  - Locals referenced outside of their function are said to have “escaped” their function/loop and must be heap allocated

Conditional and Switch Statements

- Similar to C syntax
  - Don't require parenthesis
  - Do require curly braces
  - An if statement can appear without an else statement
  - A variable can be created on the fly within an if statement
  - There is no ternary if in Go
    - e.g. a ? b : c

```go
package main

import "fmt"

func main() {
    if 4 % 2 == 0 {
        fmt.Println("4 is even")
    } else {
        fmt.Println("4 is odd")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "is a one digit number")
    } else {
        fmt.Println(num, "has multiple digits")
    }

}
```

```go
i := 2
switch i {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two")
case 3:
    fmt.Println("three")
}
```

- A basic switch statement
  - Can also use 'default' as an alternate way to express if/else logic
  - Can use commas to separate multiple values in one case statement

Loops

- There is no ‘while’ loop in Go
- Compact initialization can (and should) be used
- Break and continue have the same usage as they do in Python

```go
// `for` is Go's only looping construct. Here are
// three basic types of `for` loops.

package main

import "fmt"

func main() {

    // The most basic type, with a single condition.
    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    // A classic initial/condition/after `for` loop.
    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    // `for` without a condition will loop repeatedly
    // until you `break` out of the loop or `return` from
    // the enclosing function.
    for {
        fmt.Println("loop")
        break
    }

    // You can also `continue` to the next iteration of
    // the loop.
    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

Source: https://gobyexample.com/for

Summary

- Go syntax
- Go variables
- Go variable scope and lifetime

Lab: Syntax and Basics

- Gain familiarity with writing Go code