First Class Functions

Objectives

- Gain an deeper understanding of Go functions
- Understand:
  - Anonymous functions
  - Variadic functions
  - Closures

Go Functions

- A function declaration has
  - The keyword ‘func’
  - A name
  - A list of parameters
  - An optional list of results
  - A body

- A Function’s signature is its type, the list of parameter and return types it takes/produces (ignoring names)

```
func functionName(paramName paramType) returnType {
        //body
}
```

Function Parameters

- The parameter list specifies the names and types of the function's parameters
- These are the local variables whose values or arguments are supplied by the called
- The parameter name is followed by the type
- If two or more consecutive named functions parameters share the same type, the type only needs to follow the last parameter
- The blank identifier (underbar) can be used to accept parameters that will be discarded

```
func add(x int, y int, z int) int {
    return x + y + z
}

func add(x, y, z int) int {
    return x + y + z
}
```

Function Results

- The result list specifies the types of the values that the function returns
- There can be multiple return values
- If the function returns one unnamed result or no results at all, parentheses are optional and usually omitted
- Leaving off the result list entirely declares a function that does not return any value and is called only for its effects
- Results can be named
  - Each name declares a local variable initialized to the zero value for its type
  - Very useful when returning multiple values of the same type

```
1   package main
2
3   import "fmt"
4
5   func sub(x, y int) (z int) {
6       z = x - y
7       return
8   }
9
10  func main() {
11      fmt.Println(sub(8, 4))
12  }
13
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
2017/04/10 14:19:37 server.go:73: Using API v1
2017/04/10 14:19:37 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/10 14:19:37 debugger.go:414: continuing
4
```

Go Functions

- Callers must provide an argument for each parameter
  - Arguments must be in the order the parameters are declared
  - Go has no concept of default parameter values
  - Go offers no way to specify arguments by name
- Parameters are local variables within the body of the function
  - Their initial values are set to the arguments supplied by the caller
  - Function parameters and named results are variables in the function's outermost lexical block
  - Arguments are passed by value
    - Functions receive a copy of each argument
    - Modifications to the copy do not affect the caller
    - If the argument contains a reference (pointer, slice, map, function, or channel) the caller may be affected by modifications the function makes to external objects indirectly
- Functions without a body indicate the function is implemented in a language other than Go
- Functions may be recursive
  - They may call themselves directly or indirectly
  - This can give rise to stack overflows in some languages, for example, when traversing deep data structures
  - Go implementations use variable-size stacks that start small and grow as needed
    - Up to a limit on the order of a gigabyte
    - This makes recursion fairly safe and causes stack overflow event to fall into the clearly erroneous category in most cases
- Named functions can be declared only at the package level
  - Function literals (lamdas) can denote a function value within any expression

First Class Functions

- Functions are first-class values in Go
  - function values have types
  - may be assigned to variables
  - may be passed or returned from functions
  - may be called like any other function
  - are NOT comparable
  - can NOT be used as map keys
- Cause a panic when unset and invoked

```
1   package main
2
3   import "fmt"
4
5   func afun() {
6       fmt.Println("A")
7   }
8
9   func bfun() {
10      fmt.Println("B")
11  }
12
13  func main() {
14      f := afun
15      f()
16      f = bfun
17      f()
18      var g func(int32) float64
19      g(3)
20  }
21
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 17:28:58 server.go:73: Using API v1
2017/04/10 17:28:58 debugger.go:68: launching process with args: [d:\dev
API server listening at: 127.0.0.1:2345
2017/04/10 17:28:58 debugger.go:414: continuing
A
B
2017/04/10 17:29:05 debugger.go:414: continuing
panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xc0000005 code=0x0 addr=0x0 pc=0x488bb0]

goroutine 1 [running]:
main.main()
        d:/dev/go/example/src/example.go:19 +0x60
```

Variadic Functions

- Variadic functions can be called with varying numbers of arguments
  - fmt.Printf requires one fixed argument then accepts any number of subsequent arguments
- To declare a variadic function the type of the final parameter is preceded by an ellipsis: "…int"
  - This implicitly packs the parameters into a slice of that type
    - The caller allocates an array, copies the arguments into it, and passes a slice of the entire array to the function
  - This can also be done explicitly
    - Pass a slice with a trailing ellipsis
    - To unpack a sliced array: a[:]…
- The type of a variadic function is different from the identical function that accepts a slice explicitly
- To create a variadic function that can accept any type in any position make the last argument an interface type
  - More on this in the next module

```go
package main

import "fmt"

func sum(v ...int) int {
    total := 0
    for _, x := range v {
        total += x
    }
    return total
}

func main() {
    fmt.Println(sum(0))
    fmt.Println(sum(1))
    fmt.Println(sum(1, 2))
    fmt.Println(sum(1, 2, 3))
    s := []int{4, 5, 6}
    fmt.Println(sum(s[:]...))
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 20:07:14 server.go:73: Using API v1
2017/04/10 20:07:14 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/10 20:07:14 debugger.go:414: continuing
0
1
3
6
15
```

Library Functions values

- The Go standard library makes extensive use of function values
  - strings.Map()
  - bytes.Map()
- Function literals can be convenient in contexts where a simple function must be supplied as a parameter (line 14)

```
1   package main
2
3   import (
4       "fmt"
5       "strings"
6   )
7
8   func inc(r rune) rune {
9       return r + 1
10  }
11
12  func main() {
13      fmt.Println(strings.Map(inc, "0123456"))
14      fmt.Println(strings.Map(func(r rune) rune { return r + 2 }, "0123456"))
15  }
16
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 17:40:58 server.go:73: Using API v1
2017/04/10 17:40:58 debugger.go:68: launching process with args: [d:\dev\go\example
API server listening at: 127.0.0.1:2345
2017/04/10 17:40:58 debugger.go:414: continuing
1234567
2345678
```

  - A function literal is written like a function declaration, but without a name following the func keyword
  - It is an expression whose value is called an anonymous function
  - Function literals let us define a function at its point of use

Summary

- Go provides
  - Many common function features
    - Parameters, return values, recursion, …
  - Several not so common function features implemented in a Go specific way
    - Anonymous functions/lamdas, Closures, …

Lab: Functions

- Go specific function features