Module Seven

Advanced Functions in Go

As someone who has written a fair bit of code in functional languages and a fair bit of Go, I find that the more Go I write the less I care about the language features (or lack thereof) that I was horrified by at first, and the more I see most other languages as overcomplicated. Go isn't a very good language in theory, but it's a great language in practice, and practice is all I care about, so I like it quite a bit
supersillyus in hackernews

Go is like a better C, from the guys that didn't bring you C++
Ikai Lan

7.1 Introduction

In module four, we looked the basics properties and uses of functions in Go but most of our discussion was essentially confined to how Go does certain things, like handle errors, in ways that are different than other programming languages while still more or less treating Go functions as a different flavor of the same sort of functions we find typically in C-style languages.

However, we can do a lot more with functions in Go than just defining and calling them. In this module we look at some more advanced concepts involving using functions which may be new to you if you have been programming in a language that does not allow this sort of functionality, or does allow it but in very abstruse and convoluted ways. If you have been programming in a functional language like Scala or Haskell, many of these concepts will be quite familiar to you.

7.2 Functions as First Class Objects

Functions in Go are “first class citizens” which in programming language jargon means that you can do the same things to and with functions that you can do to or with anything else in the language. This means assigning a function to variable, passing it as an argument to another function and using a function as a return value from other functions.

7.2.1 Function variables

In example 07-01, two functions f1 and f2 are defined. The variable f is a variable of type 'func() string' which means that is able to be assigned as a value "a function of no arguments that has a string return value". In the lab, you will change f1 and f2 in different ways to verify this.

The “type” of a function then is made up to two parts:

1. The number and type of arguments to the function.
2. The return value type.

```
// Example 07-01 Function variables

package main

import "fmt"

func f1() string {
  return "I'm f1"
}
func f2() string {
  return "and I'm f2"
}

func main() {
  f := f1
  fmt.Printf("f is of type '%T' \n", f)
  fmt.Println(f())
  f = f2
  fmt.Println(f())
}
```

```
[Module07]$ go run ex07-01.go
f is of type 'func() string'
I'm f1
and I'm f2
```

Example 07-01 has a number of points that need to be understood.

The variable f is a function variable which means that it holds a reference to a function. However because we have defined f with a short form assignment, the type of function f can hold is determined when the variable is initialized. Since f is a variable, it can be used in the same way as any other variable.

However because f holds a function, we can execute whatever function is currently referenced as the value of f. If you look at the listing carefully you will see that f is initially set to f1 which means that the functional call f() is in fact the function call to f1(). Then when we assign the function f2 to f, the function call f() is now a function call to f2().

7.2.2 Passing function as parameters

Functions can also be passed as parameters, but we need to be able to specify what type of function we are passing as a parameter in the same way that we have to specify the data type of something we are passing.

Example 07-02 illustrates this

```
// Example 07-02  Functions as parameters

package main

import "fmt"

func f1() string {
  return "I'm f1"
}
func f2(fparam func() string) {
  fmt.Println(fparam())
}

func main() {
  f := f1
  f2(f)
}
```

```
[Module07]$ go run ex07-02.go
I'm f1
```

The function f1 is the function that is going to be passed as a parameter. We already know from the last example that the function type of the function f1 is 'func() string'

The function f2 is the one we will be passing to f1 as a parameter. You should have noticed by now that when we use the syntax f() we are making a function call but when we use the function name alone it means that we are not calling the function but are using the function definition as a variable value.

The parameter fparam is a function variable and has the appropriate type defined.

This example now also answers one of the points we deferred on in the module on basic functions –why do we need the func keyword? One of the reasons is that it makes this sort of syntax using functions as first class object possible.

The main function calls f2() and passes the function f1 as a parameter. The executable line in f2 now uses the () notation to execute the function in the variable fparam.

7.2.3 Functions as return arguments

If we can use functions as parameters, then we should be able to use them as return values as well, which is exactly what we can do.

```
// Example 07-03  Function as return value

package main

import "fmt"

func f1() string {
  return "I'm f1"
}
func f2() func() string {
  return f1
}

func main() {
  f := f2()
  fmt.Println(f())
}
```

```
[Module07]$ go run ex07-03.go
I'm f1
```

Example 07-03 Demonstrates this although you may have to read through the code a few times to follow what is happening.

The function f2 is going to return the function f1 as a return value. Since we declare return types in the same way that we declare parameters, we use the same type expression we did when we used f1 as a parameter.

The part that may be difficult to get on the first reading is decoding what is happening in the main function. We are calling f2 and get back a function that we assign to f and then we execute the function that now the value of function variable f. There is another way to do this without using f by just doing f2()() which means execute the result of executing f2. We will be working with this syntax in one of the labs.

7.3. Function literals

The way we have been working with functions so far is a bit clunky and not something we normally do, but it has illustrated the idea of functions as first class objects.

Continuing the analogy with variables, the use of the function definitions that we have used in previous modules are the equivalent to the var form of defining a variable – we are defining the function and then using the name of the function to pass it around. As we have seen,with variables, this form is a little more formal and awkward so we tend to use the short form definition of variables with the := operator. That leads to the obvious question as to whether there is some way that we could use the := operator for functions the same way we do for variables.

The answer is yes and is demonstrated in example 07-04 where we create a function variable f and initialize it with what is called a function literal. A function literal is a function definition without a name that we can then use as a regular function once it is assigned to a variable.

```
// Example 07-04  Function literal

package main

import "fmt"

func main() {
  f := func(i int) bool { return i == 0 }
  fmt.Println("2 == 0 is", f(2))
  fmt.Println("0 == 0 is", f(0))
}
```

```
[Module07]$ go run ex07-04.go
2 == 0 is false
0 == 0 is true
```

However the scoping rules still apply to f – it is a still a local variable so we can't call it directly from outside the scope in which it was defined. This sort of dynamic defining of a function using function literals is usually not found in compiled languages but is more characteristic of interpreted languages like Ruby and JavaScript.

But what if we wanted to use our function outside of the scope in which it is defined? We can pass it as a parameter into that other scope. This is demonstrated in example 07-05 where we pass the function literal directly to f2 and execute it there.

Until you get used to it, the code is a bit harder to read but after you work with it for a while, it becomes second nature.

```go
 // Example 07-05  Function parameter

package main

import "fmt"

func f2(p func(int) bool) {
  fmt.Println("2 == 0 is", p(2))
  fmt.Println("0 == 0 is", p(0))
}

func main() {
  f2(func(i int) bool { return i == 0 })
}
```

```
[Module07]$ go run ex07-05.go
2 == 0 is false
0 == 0 is true
```

7.4 Anonymous functions.

So finally we ask the question of whether we even need to use function variables at all given that we can use function literals. The answer is no we don't, we can use these function literals on their own in a form we call anonymous inner functions.

In example 07-06 the function literal is executed directly without needing to assign it to a variable. There is an important notational point here that needs to be clarified

```
func() { ...}  is a function literal
func() {...}()      executes the function literal
```

this means that

```
f: = func() {...}
```

assigns the function literal to the variable f while

```
g : = func() {...} ()
```

assigns the result of evaluating the function literal to g.

```go
// Example 07-06  Executing a literal directly

package main

import "fmt"

func main() {
  z := func(x int) (y int) {
       y = x + 1
       return
  }(0)
  fmt.Println(z)
}
```

```
[Module07]$ go run ex07-06.go
1
```

Another very simplistic form of an anonymous inner function is depicted in example 07-07.

Unlike example 07-06 where we execute the anonymous function and assign the result to Z, in example 07-07 we don't have to assign the function or the result to anything. The function is inner since it is defined inside another function and it is anonymous since there is no way to reference the function: it just executes. The only real difference between 07-07 and 07-06 is that we don't care about a return value in 07-07: all we are doing is encapsulating a task or some functionality in a function literal.

While this all seems a bit complex and unnecessary, in a future module, we will find that these anonymous functions are very useful in implementing concurrency.

```
// Example 07-07  Anonymous inner function

package main

import "fmt"

func main() {
  defer func(name string) {
       fmt.Println("Hello ", name)
       return
  }("World")
fmt.Println("Main Function")
}
```

```
[Module07]$ go run ex07-07.go
Main Function
Hello  World
```

7.5 Closures

Closures are often confused with anonymous functions, but while they are related to them, closures are not the same thing. For example, the anonymous function in example 07-07 is not a closure.

Closures occur when a function has what are called free variables in the function body. A free variable is a variable used in a function which is not a local variable defined in the function body or one that is passed as a parameter, although in Go we should also add variables that are used as named return values are not free variables either.

In example 07-08, the variable “a” is a free variable. It is defined in the same scope as the anonymous function but is not part of the function. When we assign the anonymous function to z, we create an instance of the function, but because there is a free variable, the instance also makes a copy of the free variable. The function instance along with the copy of the free variable it makes is called the closure of the function.

We know this is a closure because when we pass the anonymous function as a parameter to f, we create an instance of the function including a copy of the variable “a”. Since f is out of the scope of the original definition of “a” it can't access the original variable so instead it uses the copy that was sent along with the function instance in the closure.

A closure can be thought of as an instance of the function that also has copies of all the necessary information from the scope in which it was defined (ie. free variables) so that it can execute in another scope.

```
// Example 07-08  Simple Closure

package main

import "fmt"

func f(p func(string)) {
  p("Mars") // "a" is out of scope here
}

func main() {
  a := "again "
  z := func(name string) {
       fmt.Println("Hello ", a, name)
       return
  }
  f(z)
}
```

```
[Module07]$ go run ex07-08.go
Hello  again  Mars
```

A more complex example of a closure is shown in example 07-09 which is taken from the Go Tour on the Golang website.

```
// Example 07-09 Closure

package main

import "fmt"

func intSeq() func() int {
  i := 0
  return func() int {
       i += 1
       return i
  }
}

func main() {
  nextInt := intSeq()
  fmt.Println(nextInt())
  fmt.Println(nextInt())
  newInts := intSeq()
  fmt.Println(newInts())
}
```

```
[Module07]$ go run ex07-09.go
1
2
1
```

In this case the closure is taking place in the function intSeq. When the line

```
nextInt := intSeq()
```

is executed, an instance of the inner function is created with a closure that contains a copy of the variable i. Every time this instance is executed, the copy of "i" in the closure is incremented.

The statement

```
newInts := intSeq()
```

creates a new instance of the inner function and a new closure copy of the variable i which is independent of the other copy in the other closure. This can be seen in the output where each time a closure is created, the value of “i” is set to 0.