Module Four

Function Basics

I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000. Even though Go does not have the power of abbreviation, the flexible type system seems to out-run Scala when the programs start getting longer. Hence, Go produces much shorter code asymptotically.
Petar Maymounko

Go isn't functional, it's pragmatical. Why pure paradigms like FP or OOP are always a must? (sigh)
Frank Mueller

Introduction to Programming in Go

4.1 Introduction

Functions are one of the fundamental building blocks of any programming language whether they are defined as C style procedures, or methods that belong to a class as in Java, or as program objects like in functional programming languages like Scala.

The initial syntax of functions in Go can easily mislead you into thinking that as far as functions are concerned, Go is just a 21st century rewrite of how C "does functions" – which it most definitely is not. Go is not a programming language that implements a particular programming paradigm like Ruby implements object oriented programming or Scala implements functional programming. Instead it has taken ideas from a variety of programming models and created a very powerful implementation of the function concept.

This is the first of four modules on functions. The focus of this module is on the basics of Go functions with an emphasis on how Go functions are different that those in other C-style programming languages.

Module seven on advanced functions looks at functions as first class objects, which is programming jargon that means we can do almost anything with a function that we can do with a variable. We also look at using anonymous functions in Go, which is the way that Go implements lambda functions, as well as examining function literals and closures.

Module nine introduces Go methods which are a mechanism for associating Go functions with user defined types to deliver the same sort of capabilities we find in object oriented languages.

Module eleven introduces one of the more unique features of Go functions which is using functions as concurrent goroutines. This is Go's very powerful and elegant model for implementing concurrency.

4.2 Function Definitions

Functions in Go are defined using the following syntax:

func fname(var$_1$ type$_1$,..var$_n$ type$_n$) type { ...}

However functions in Go can also have named return values, in which case the syntax looks like this

func fname(var$_1$ type$_1$,..var$_n$ type$_n$) (var type) { ... }

Named return values are something that other C-style languages do not have. A named return value is a local variable defined in the function declaration which can be used like any other local variable but which is returned when the function is finished executing.

The func keyword is mandatory for a couple of reasons. The first is that it conforms to the rule that all package level constructs begin with an identifying token to improve the efficiency of parsing of Go code. The other reason is that as we get into more advanced use of functions, it will be critical to know if a chunk of code is just an ordinary chunk of code or if it is in fact a function literal. We will return to that topic in module seven.

The order of the parameters and types, and the return values and their types is the same as for Go variable declarations, which is reversed from most of the other languages. This takes a bit of getting used to in you are a long time programmer in C or Java.

Go, like most other C-style languages, passes parameters and returns by value, which means that the function is working with a local copy of the arguments that were passed by the calling function. I am assuming that since you are experienced programmers, that the concepts of call by value and call by reference are quite familiar to you so we won't go into the mechanics of it in this material.

4.2.1 Basic function definition

Example 4.1 shows the most C-style function definition we can have in Go. If you are a C-style programmer, aside from the order of the parameters and types, this looks like pretty standard code.

Example 4.2 demonstrates the same function but using a named return value. A couple of points to note:

1. If a named return value is used, then the parentheses around the return values are not optional. Similarly, if multiple returns are used, the parentheses around the list of return values are not optional.
2. When the function is called, the named return variables are created and initialized to their zero values. After that, they are treated just like any other local variable in the function body.

```go
// Example 04-01 Basic function sytnax

package main

import "fmt"

func square(x int) int {
  y := x * x
  return y

}
func main() {
  x := 4
  retval := square(x)
  fmt.Println("x = ", x, "square = ", retval)
}
```

```
[Module04]$ go run ex04-01.go
x =  4 square =  16
```

```go
// Example 04-02 Named return value

package main

import "fmt"

func square(x int) (result int) {
  result = x * x
  return
}
func main() {
  x := 4
  retval := square(x)
  fmt.Println("x = ", x, "square = ", retval)
}
```

```
[Module04]$ go run ex04-02.go
x =  4 square =  16
```

3. When using named return values, we don't need to specify what is returned and can use a naked return statement (just the word "return" alone) since Go knows which variables to return.
4. Mixing the two styles is not allowed, you either use a naked return with named return values or return the value explicitly like in C or Java.

Just for completeness, example 04-03 shows an alternate way of writing a parameter list which is in keeping with the Go way of declaring variables.

```go
// Example 04-03 Multiple parameters

package main

import "fmt"

func divides(x, y int) (div bool) {
  div = (y % x) == 0
  return true
}
func main() {
  x, y := 2, 21
  fmt.Println(x, "|", y, " is ", divides(x, y))
}
```

```
[Module04]$ go run ex04-03.go
2 | 21  is  true
```

4.2.2 Multiple Return Values

One of the more innovative features of Go is that functions can return multiple values. We can emulate this other languages by bundling up a collection of values into some return object, but we still wind up returning a single object. One of the reasons Go allows multiple returns is to accommodate the error handling mechanism it uses instead of exceptions. We will look at errors a bit later in this module.

The syntax for returning multiple values is a logical extension of the single return value case is shown below. Multiple return values can be named or unnamed but may not be a mixture of the two.

func fname(v_1 t_1...) ($type_1$, $type_2$, ..) { ... }
func fname(v_1 t_1...) (var_1 $type_1$, var_2 $type_2$) { ... }

Example 04-04 shows the use of multiple unnamed return values.

```
// Example 04-04 Multiple return values

package main

import "fmt"

func divides(x, y int) (int, int) {
  return (y / x), (y % x)
}
func main() {
  x, y := 3, 23
  quot, rem := divides(x, y)
  fmt.Println(x, "/", y, " is ", quot, "R", rem)
}
```

```
[Module04]$ go run ex04-04.go
3 / 23  is  7 R 2
```

Example 04-05 shows the use of multiple named return values.

```
// Example 04-05 Multiple named return values

package main

import "fmt"

func divides(x, y int) (q int, r int) {
  q = y / x
  r = y % x
  return
}
func main() {
  x, y := 3, 23
  quot, rem := divides(x, y)
  fmt.Println(x, "/", y, " is ", quot, "R", rem)
}
```

```
[Module04]$ go run ex04-05.go
3 / 23  is  7 R 2
```

There are some basic rules about multiple return values:

1. The return values either all named or all unnamed – you cannot mix the two types.
2. Unnamed return values have to be returned in a list following the keyword "return."
3. The named return value form uses a naked return. You don't have to tell Go what to return, when it encounters a return, it will return the current values of the return variables at the moment the return is executed.
4. The function passes the return values to the calling function in the same order they listed in the function declaration.
5. It's all or nothing, all of the values have to be returned, you can't return some of them. Notice though for named return values this is automatic.

4.3 Deferred Execution

This concept illustrates the first of several Go operators that change the behavior of a function when it is called, sort of an operator on a function. What Go does with the defer operator is allow you to call a function and have it run later rather than at the time you called actually called it.

We defer a function's execution by using the word “defer” in front of it when we call it. This has the effect of waiting until the calling function is about to exit before actually running the deferred function. The result is that we are calling the function, evaluating the arguments to the call, then waiting until the calling function executes before running the deferred function.

There is an important aspect about deferring a function which makes them very useful – the arguments to the deferred function are evaluated when the function is called, not when it is executed.

Example 4.6 shows that the deferred function f() clearly runs after the main is finished executing and just before the main function exits because the message printed by f() occurs after the message printed by main.

Also notice that the value of the message is what the value of “m” was when f() was called, not when f started to execute.

```go
 // Example 04-06 Deferred execution

package main

import "fmt"

func f(message string) {
  fmt.Println("Value of param =", message)
  return
}
func main() {
  m := "before defer"
  defer f(m)
  m = "after defer"
  fmt.Println("Value of m when main() exits =", m)
}
```

```
[Module04]$ go run ex04-06.go
Value of m when main() exits = after defer
Value of param = before defer
```

A deferred function is almost like a closure in the sense that it is closed over its arguments when it is called. If you don't know what a closure is, don't worry, we will be seeing them in module seven.

Two of the uses of defer are to recover from panics, which we will look at later, and to do some sort of cleanup before a function exits. For example, suppose we have a program that opens some sort of connection or has a lock on some resource; if the code that closes the connection is put into a deferred function, then no matter when the original function executes a return, the deferred function will execute and the cleanup will be done.

4.3.1 Stacked Defers

We can defer as many functions calls as we want. Each time we make a deferred call, the function arguments and evaluated and the function put into a stack of deferred function calls. When the calling function exits, the deferred functions are then executed in a first in last out order (like a stack).

```go
// Example 04-07 Stacked defer

package main

import "fmt"

func f(k int) {
  fmt.Printf("executed f(%d)\n", k)
  return
}

func main() {
  for i := 1; i < 4; i++ {
       fmt.Printf("called f(%d)\n", i)
       defer f(i)
  }
  fmt.Println("end main")
}
```

```
[Module04]$ go run ex04-07.go
called f(1)
called f(2)
called f(3)
end main
executed f(3)
executed f(2)
executed f(1)
```

Example 04-07 demonstrates a stacked defer. Notice that each function call remembers the value of “i” when it was called, not when it executed. Also notice that the function calls execute in the reverse order they were called.

4.4 Recursion

Go supports recursion in the same way most programming languages do. I am assuming that you know what recursion is and how it works so we will not be going into the mechanics of recursion in this class. Example 04-08 just demonstrates what recursion looks like in Go. The function sum() recursively adds up the a series of integers from 1 to k for a given k.

```go
// Example 04-08 Recursion

package main

import "fmt"

// sums numbers from 1 - k

func sum(k int) int {
  fmt.Printf("executed sum(%d)\n", k)
  if k > 0 {
       return (k + sum(k-1))
  } else {
       return 0
  }
}

func main() {
  fmt.Println(sum(3))
}
```

```
[Module04]$ go run ex04-08.go
executed sum(3)
executed sum(2)
executed sum(1)
executed sum(0)
6
```

4.5 Varadic Functions

A Varadic function is one that can be called with a varadic parameter which is one that can be passed a variable number of arguments by the calling function. Varadic functions are incredibly useful – for example the fmt.Println() function is varadic since we can pass it a variable number of arguments when we call it.

Go does not allow optional arguments or keyword named parameters like some other programming languages, and the rules for varadic functions tend to be a little bit on the strict side.

The format for a varadic function is:

```
func fname (p1 t1, v ... type)
```

for example:

```
func v(message string, x ... int)
```

In the example above, message is a non varadic parameter since there is only one, but x is a varadic parameter so there may be any number of ints passed as arguments to x. All of the following would be valid calls to v()

```
v(“hi”)
v(“hi,1)
v(“hi”,1,2,3,4)
```

Varadic functions must obey the following two rules:

1. There may be only one varadic parameter although there may be other non-varadic parameters. The varadic parameter must be the last parameter in the parameter list.
2. Ellipsis “...” is used to identify the parameter as varadic.

4.5.1 Processing varadic parameters

In the next module we will be looking at arrays in Go but for now all you have to know is what an array is, which you is something that should be quite familiar to you, and also from the lab in the last module.

The arguments passed to the varadic parameter are made available as an array of the parameter type.

This is demonstrated in example 04-09.

```go
// Example 04-09 Varadic functions

package main

import "fmt"

func addup(nums ...int) (sum int) {
  for _, val := range nums {
       sum += val
  }
  return
}

func main() {
  fmt.Println(addup(1, 2, 3, 4, 5))
}
```

```
[Module04]$ go run ex04-09.go
15
```

4.6 Error Handling

Almost everyone who programs in an object oriented language uses exceptions. Go does not use exceptions for a variety or reasons.

First, is the fact that exceptions are cumbersome to compile and produce a lot of code bloat in binaries which has an impact on build time. In fact, Bjarne Stroustrup recommends that if we are looking to improve the performance of C++, then not using exceptions makes the resulting code smaller and faster by a significant amount, but at the cost of being somewhat riskier.

Second, throwing exceptions that might be caught literally anywhere, or even worse they might never be caught, can produce problems in managing large code bases. While exceptions have some nice properties for certain kinds of designs, the developers of Go felt that it was a effective optimization to not have them in the language for the kinds of large scale code bases Go is intended to be used with.

Third, exceptions are often used to get around the issue of how we handle error conditions. The usual explanation is that because we were confined to looking at a single return value, trying to determine if the value returned is an error code or valid value, the code in the calling function can become very complex. This is very true unless we can return multiple values, and because Go supports multiple return values we can return an additional error object that we can test locally to see whether or not a function call succeeded. We do want to avoid a discussion on which is better theoretically – exceptions or the Go error system – but there are certainly good arguments made by the language developers for this way of handling errors.

Not every function needs to deal with errors. The only time that we normally use a Go error is when there is a possibility that a function may fail for the same types of conditions that would raise an exception in a language like Java. For example, a simple computation would probably never need an error but a read from a file would because it is possible that a number of exceptional conditions could occur that would prevent the read from happening.

Some basics about using errors:

1. The error is always the last return value in the return list.
2. The error is created using the New operator – this construct will be covered a later module so we defer the discussion of what that one line of code means until then.
3. A nil can be returned as the error if no error occurs. We discuss nil in a future module but for now just think of this as being like what nil means in other C-Style languages, or a null pointer.
4. If the error is nil, then we can proceed with execution normally
5. If the error is non-nil, we should ignore the other results returned because they are now suspect, although in some cases we may be able to fix things up.

```go
// Example 04-10 Error Handling

package main

import "fmt"
import "errors"

func division(num, denom int) (int, error) {
  if denom == 0 {
       return 0, errors.New("Divide by zero")
  }
  return (num / denom), nil
}

func main() {
  res, e := division(56, 0)
  if e != nil {
       fmt.Println(e)
  } else {
       fmt.Println(res)
  }
}
```

```
[Module04]$ go run ex04-10.go
Divide by zero
```

6. The error is printable. We will look more into the error construct in a later module, but for now just note that there is a string output we can get by printing the error.

Example 4-11 and 4-10 show variations on the handling of an error where we don't care about what error occurred but are testing to see if we should proceed with processing or should abort processing because of an error.

```go
// Example 04-11 Error Handling

package main

import "fmt"
import "errors"

func division(num, denom int) (int, error) {
  if denom == 0 {
       return 0, errors.New("Divide by zero")
  }
  return (num / denom), nil
}

func main() {
  res, e := division(56, 2)
  if e != nil {
       fmt.Println(e)
  } else {
       fmt.Println(res)
  }
}
```

```
[Module04]$ go run ex04-11.go
28
```

4.6.1 Comma ok Idiom

In some cases, such as in example 4-11, all we want to know is whether an error occurred or not. In this case we often return a bool instead of an error object to provide a test value which by convention we assign the to the variable "ok". Generally we use a bool when all we care about is if an error occurred or not, but we really don't care what kind of error nor are we interested in any details about the error.

We will see this idiom later when we look at maps and in other places where we just want to see if some data value exists or some processing took place. We tend to use this idiom when the condition is not so much an error as some condition that could happen that we have to respond to or take into account.

For example, we may want to delete a file but the file might not be there. Is it really an error if the file is not there or just a possibility we have to handle? In this case the ok variable may just tell us whether or not the file is there so that we know if we should proceed with the delete operation.

```go
// Example 04-12 Comma OK Idiom

package main

import "fmt"

func division(num, denom int) (int, bool) {
  if denom == 0 {
       return 0, false
  }
  return (num / denom), true
}

func main() {
  res, ok := division(56, 2)
  if ok {
       fmt.Println(res)
  } else {
       fmt.Println("division failed")
  }
}
```

```
[Module04]$ go run ex04-12.go
28
```

4.7 Panics and Recoveries

Those who are used to UNIX environments will know what a panic is, but if you don't, then think of a panic as being like a run time error or run time exception that is thrown in Java.

Generally, the best practice if a panic occurs is to terminate the application gracefully.

However is some cases we may be able to recover from the panic. It is advisable to only recover from a panic into some fail-safe state where the application can clean up its environment and then perform a controlled shutdown. The problem is that we can get sloppy and use panics to deal with situations that are not really serious enough to be panics but should be handled instead with the use of errors in the program logic. This is the same sort of trap that OO programmers fall into with using exceptions for things that are not really exceptional.

Summarizing panics and recoveries

1. A panic is generated by Go when a runtime error occurs
2. During a panic, all deferred functions are called, then execution halts
3. Panics can be generated by calling the panic() function
4. Panics can be recovered from by executing the recover() function
5. The recover() function only works in deferred function
6. Panics should be used only for critical problems, use errors otherwise

```go
// Example 04-13 Runtime panic

package main

import "fmt"

func main() {
  x, y := 1, 0
  fmt.Println(x / y)
}
```

```
[Module04]$ go run ex04-13.go
panic: runtime error: integer divide by zero
[signal 0x8 code=0x1 addr=0x40102c pc=0x40102c]
```

```go
// Example 04-14 Runtime panic

package main

import "fmt"

func division(num, denom int) int {
  if denom == 0 {
       panic("Dividing by zero?!?")
  }
  return (num / denom)
}

func main() {
  res := division(56, 0)
  fmt.Println(res)
}
```

```
[Module04]$ go run ex04-14.go
panic: Dividing by zero?!?
goroutine 1 [running]:
panic(0x4b8e00, 0xc82000a3c0)
```

Example 4-15 shows how we can generate a panic by a call to the panic() function. Generally this should be one of those programming strategies of last resort, meaning the situation you have encountered cannot be reasonably resolved and some sort of system shutdown is necessary. I wouldn't recommend it to be used the way it is in the example.

Example 4-16 shows the use of the recover() function to try and recover from a panic.

Since a panic causes an immediate exit from the function where the panic occurred, any recovery code has to go into a deferred function that has already been called. No more function calls can take place after a panic so only deferred functions can execute.

If there is no panic, executing the recover() function does nothing, there are no effects from its execution and it returns a nil. However, if a panic has occurred, then recover() prevents the program from shutting down and returns the panic item that caused the panic so that any actual recovery code can be executed. In this example, the recovery code just prints out a message before exiting.

The best practical use of recovery is to move the system in to a fail-safe state before shutting down so that we do not leave the environment or other connected resources in an unstable or invalid state.

```go
// Example 04-15 Runtime panic and recovery

package main

import "fmt"

func rec() {
  r := recover()
  if r != nil {
       fmt.Println("recovered value = ", r)
  }
}
func main() {
  x, y := 1, 0
  defer rec()
  fmt.Println(x / y)
}
```

```
[Module04]$ go run ex04-15.go
recovered value =  runtime error: integer divide by zero
```