Functions as Return Variables

This lesson discusses how to use a function as a return variable or value.

WE'LL COVER THE FOLLOWING

- Returning a function using closures
- Factory Function

Returning a function using closures

Just like we return a variable or a value from a function, we can return a function too. The following function `genInc` returns a function:

```
// genInc creates an "increment n" function
func genInc(n int) func(x int) int {
    return func(x int) int {
        return x+n
    }
}
```

It's obvious from the header of `genInc`, that it's returning a function that takes a parameter `x` of type `int`, and that function returns an `int` value. The function returned by `genInc` returns `x+n`. The following program is an implementation of returning a function.

```
package main
import "fmt"

func main() {
    // make an Add2 function, give it a name p2, and call it:
    p2 := Add2()
    fmt.Printf("Call Add2 for 3 gives: %v\n", p2(3))
    // make a special Adder function, a gets value 3:
    TwoAdder := Adder(2)
    fmt.Printf("The result is: %v\n", TwoAdder(3))
}

func Add2() (func(b int) int) {           // return a function
```

```
    return func(b int) int {
        return b + 2
    }
}
func Adder(a int) (func(b int) int) {   // return a function
    return func(b int) int {
        return a + b
    }
}
```

Return Function

In the above program, we implement *two* functions `Add2` and `Adder`, which return another *lambda* function. There is a header of `Add2` at **line 13**. You can notice that it returns an *anonymous* function that takes `b` (of type `int`) as a parameter and returns an `int` value. That *anonymous* function is returning `b+2`. Similarly, there is a header `Adder` at **line 18**. You can notice that it takes a parameter `a` (of type `int`), and it also returns a *closure* that takes `b` (of type `int`) as a parameter and returns an `int` value. That *closure* is returning `a+b`.

Now, look at **line 6** in `main`. We are calling `Add2` and setting it equal to `p2`. Then, we call `p2(3)` at **line 7**, which calls the closure returned by `Add2`. So `b` of `Add2` is equal to **3**. On returning `b+2`, **5** will be printed. Similarly, look at **line 9** in `main`. We are calling `Adder(2)` and setting it equal to `TwoAdder`. Then, we call `TwoAdder(3)` at **line 10**, which calls the closure returned by `Adder`. So `b` of `Adder` is equal to **3**, and `a` of `Adder` is equal to **2**. On returning `a+b`, **5** will be printed.

Here is (nearly) the same function used in a slightly different way:

```
package main
import "fmt"

func main() {
    var f = Adder()
    fmt.Print(f(1), " , ")
    fmt.Print(f(20), " , ")
    fmt.Print(f(300))
}

func Adder() func(int) int {
    var x int
    return func(delta int) int {
        x += delta
        return x
```

```
    }
}
```

Function Closure

In the above program, we implement function `Adder` which returns another *lambda* function. There is a header of `Adder` at **line 11**. You can notice that it returns an *anonymous* function that takes an `int` parameter and returns an `int` value. We declare an `int` variable `x` in `Adder` at **line 12**, and that *anonymous* function returns the modified value of `x`, after adding its parameter `delta` into `x`.

Now, look at **line 5** in `main`. `Adder()` is now assigned to the variable `f` (which is then of type `func(int) int`). In the calls to f, `delta` in Adder() gets the values **1**, **20** and **300** from **line 6**, **line** 7 and **line 8**, respectively. We see that between the calls of `f` the value of `x` is retained, first it is: **0** + **1** = **1**, then it becomes **1** + **20** = **21**, then 21 is added to 300 to give the result **321**. The lambda function stores and accumulates the values of its variables. It still has access to the (local) variables defined in the current function.

Closures help in returning a function as a variable. This approach can also convert a recursive approach to a non-recursive one. How about writing a *non-recursive* version of the **Fibonacci** program using a function as a closure?

```
package main
import "fmt"

// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int {
    a, b := 1, 1
    return func() int {
        a, b = b, a + b
        return b
    }
}

func main() {
    f := fib()
    // Function calls are evaluated left-to-right.
    // fmt.Println(f(), f(), f(), f(), f())
    for i := 0; i <= 9; i++{
        fmt.Println(i + 2, f())
    }
}
```

Fibonacci Closure

In the program above, we implement function `fib`, which returns another *lambda* function. In the header of `fib` at **line 6**, notice that it returns an *anonymous* function that takes nothing as a parameter and returns an `int` value. We declare an `int` variable `a` and `b` in `fib` at **line 7** and initialize both of them with **1**. That *anonymous* function returns the modified value of `b`, after adding `a` into `b`. Now, look at **line 15** in `main`. The function `fib()` is now assigned to the variable `f` (which is then of type `func() int`). At **line 18**, there is a *for* loop that iterates **10** times. In each iteration, we call `f()`. This means the first 10 Fibonacci values will be printed, excluding values of **0** and **1**. Let's follow the first **5** iterations.

- In the *first* iteration, `a` and `b` are declared and initialized with **1**. Now, the lambda function makes `b` equal to `a+b` (which is **2**) and `a` equal to `b` (which is **1**). The value **2** will be returned as `b` holds the Fibonacci value at a position and `a` holds previous Fibonacci value.
- In the *second* iteration, `a` and `b` will hold their values from the previous iteration. Now, the function makes `b` equal to `a+b` (which is **3**) and `a` equal to `b` (which is **2**). The value `b` will be returned which is **3**.
- In the *third* iteration, the function makes `b` equal to `a+b` (which is **5**) and `a` equal to `b` (which is **3**). The value `b` will be returned which is **5**.
- In the *fourth* iteration, the function makes `b` equal to `a+b` (which is **8**) and `a` equal to `b` (which is **5**). The value `b` will be returned which is **8**.
- In the *fifth* iteration, the function makes `b` equal to `a+b` (which is **13**) and `a` equal to `b` (which is **8**). The value `b` will be returned which is **13**.

The same pattern will be followed for the rest four iterations.

Factory Function

A function that returns another function can be used as a **factory function**. This can be useful when you have to create a number of similar functions: write 1 factory function instead of writing them all individually. The following function illustrates this:

```go
func MakeAddSuffix(suffix string) func(string) string {
    return func(name string) string {
        if !strings.HasSuffix(name, suffix) {
            return name + suffix
        }
        return name
    }
}
```

`MakeAddSuffix` is returning functions that add a suffix to a filename when this is not yet present. Now, we can make functions like:

```go
addBmp := MakeAddSuffix(".bmp")
addJpeg := MakeAddSuffix(".jpeg")
```

And you can call them as:

```go
addBmp("file") // returns: file.bmp
addJpeg("file") // returns: file.jpeg
```

Here is an example of a factory function that takes a function, and creates another one of a completely different type.

```go
package main
import "fmt"

type flt func(int) bool
type slice_split func([] int)([] int, [] int)

func isOdd(integer int) bool {  // check if integer is odd
    if integer % 2 == 0 {
        return false
    }

    return true
}

func isBiggerThan4(integer int) bool {  // check if integer is greater than 4
    if integer > 4 {
        return true
    }
    return false
}

func filter_factory(f flt) slice_split {    // split the slice on basis of func
    return func(s[] int)(yes, no[] int) {
        for _, val := range s {
            if f(val) {
                yes = append(yes, val)
            } else {
```

```
                no = append(no, val)
            }
        }

        return
    }
}

func main() {
    s := [] int {1, 2, 3, 4, 5, 7}
    fmt.Println("s = ", s)
    odd_even_function := filter_factory(isOdd)
    odd,even := odd_even_function(s)
    fmt.Println("odd = ", odd)
    fmt.Println("even = ", even)
    //separate those that are bigger than 4 and those that are not.
    bigger,smaller := filter_factory(isBiggerThan4)(s)
    fmt.Println("Bigger than 4: ", bigger)
    fmt.Println("Smaller than or equal to 4: ", smaller)
}
```

Filter Factory

The above program has two basic functions. The first function, `isOdd` takes an integer as a parameter and returns bool value (see its header at **line** 7). If the integer is *odd*, it will return *true*, otherwise *false*. Similarly, the second function `isBiggerThan4` takes an integer as a parameter and returns bool value (see its header at **line 15**). If the integer is greater than **4**, it will return *true*, otherwise *false*.

At **line 4**, we are aliasing a type. A function that takes a single *integer* as a parameter and returns a single *boolean* value is given a type `flt`. Similarly, at **line 5**, we are aliasing a type. A function that takes a slice of *integers* as a parameter and returns two slices of integers is given a type `slice_split`.

Now moving towards a major part of the program that is `filter_factory` function, see its header at **line 22**. It takes a function `f` of type `flt`(either `isOdd` or `isBiggerThan4`). The function `filter_factory` returns a *lambda* function of type split_slices that takes `s` as a parameter and returns two integer slices `yes` and `no` (see **line 23**). From **line 25** to **line 30**, we are evaluating `s` slice on the basis of `f` function given to `filter_factory`. If the `f` function returns **true** for an integer, it becomes part of `yes`, otherwise `no`.

Let's see the `main` function now. At **line 36**, we declare a slice of integers named `s`. Then at **line38**, we call the `filter_factory` function with `isOdd` as the parameter and store the result in `odd_even_function` of type `slice_split`. Now, the `odd_even_function` is called with `s` as a parameter, which returns two slices `odd` and `even` that contain odd and even numbers from `s`, respectively. Similarly at **line 43**, we call the `filter_factory` function with `isBiggerThan4` as the parameter. You may have noticed we pass `s` on the same line as: `bigger, smaller: = filter_factory(isBiggerThan4)(s)`, rather than making a separate `split_slices` variable and then passing `s` to it. The result is stored in the `bigger` and `smaller` slices. Printing `odd` and `even` slices at **line 40** and **line 41**, respectively, verifies the result. Similarly, printing `bigger` and `smaller` slices at **line 44** and **line 45** respectively verifies the result.

Now, you are familiar enough with the use of closures. You’ll study how to debug using closures in the next lesson.