CS 315

Homework 3

Erhan Er

21801809

Section 3

1. Nested Subprogram Definitions in Go

In Go, nested subprograms are available. In order to use them, the function declaration must be assigned to a variable. Then, the function can be reached by using the variable with a closed parenthesis at the end.

```go
fmt.Println("********** Nested subprogram definitions **********")

/* In Go, nested functions are allowed.
 * In order to create a nested function
 * the function declaration must be assigned to a variable.
 **/
sub1 := func(a int) int { // nested in main
   sub2 := func() int { return a * 5 } // nested in sub1
   return sub2()
   // total return 5a
}

sub3 := func(a int) int { // nested in main
   sub4 := func(a int) int { // nested in sub3
      sub5 := func(a int) int { // nested in sub4
         return a + 5
      }
      a = a * 5
      return sub5(a)
   }
   a = a * 10
   return sub4(a)
   // total return 50a + 5
}

fmt.Println(sub1(10)) // It will print 50
fmt.Println(sub3(20)) // It will print 1005
fmt.Println("************************************************")
```

2. Scope of Local Variables in Go

Go has a static scope. Therefore, the functions will look for variables in their parents. In the given example, the scope will look for the parent functions of scope5. The parent function is main. Therefore, the result is a - 1, b - 2, c – 3. Other functions explanations are in the code and in the photos.

```
fmt.Println("********** Scope of local variables **********")

a := 1
b := 2
c := 3

scope5 := func() {
   fmt.Println("In scope5: a -", a, " b -", b, " c -", c)
   /* Go has static scope, thus it will look for the parent function of scope5.
    * The parent function is main.
    * Therefore the result is a - 1, b - 2, c - 3
    **/
}

scope1 := func() {
   b := 5
   scope2 := func() {
      a := 10

      scope4 := func() {
         fmt.Println("In scope 4: a -", a, " b -", b, " c -", c)
         /* Go has static scope, thus it will look for the parent functions of scope4.
          * The parent function is scope2 and scope 2 has the variable a.
          * Therefore, a comes from scope2. However, scope2 does not have other variables.
          * Therefore, it will look the parent of scope2 for the other variables.
          * Scope2's parent is scope1 and it has the variable b.
          * Thus, b comes from scope1. However, it does not variable c
          * Therefore, it will look the parent of scope1 for the variable c.
          * Main is the parent of scope1 and main has variable c.
          * Thus, c comes from main.
          * Therefore the result is a - 10, b - 5, c - 3
          **/
      }

      scope4()
      fmt.Println("In scope 2: a -", a, " b -", b, " c -", c)
      /* Go has static scope, thus it will look for the parent functions of scope2.
       * Therefore, a comes from scope2, b comes from scope 1 and c comes from main
       * Thus, the result is a - 10, b - 5, c - 3
       **/
   }
```

```
    scope3 := func() {
        c := 20
        fmt.Println("In scope 3: a -", a, " b -", b, " c -", c) // a - 1, b - 5, c - 20
        /* Go has static scope, thus it will look for the parent functions of scope3.
         * Therefore, b comes from scope1, a and c come from main.
         * Thus, the result is a - 1, b - 5, c - 3
         **/
    }

    scope2()
    scope3()
    scope5()
    fmt.Println("In scope 1: a -", a, " b -", b, " c -", c) // a - 1, b - 5, c - 3
    /* Go has static scope, thus it will look for the parent functions of scope1.
    * Therefore, b comes from scope1, a comes from main and c comes from scope3
    * Thus, the result is a - 1, b - 5, c - 20
    **/
}

fmt.Println("In main: a -", a, " b -", b, " c -", c) // a - 1, b - 2, c - 3
scope1()
scope5()
fmt.Println("In main: a -", a, " b -", b, " c -", c)
// a - 1, b - 2, c - 3
// Global variables did not change. That means, local variables with the same name
// of a global variables do not change the value of the global variables.
fmt.Println("*********************************************")
```

3. Parameter Passing Methods in Go

Go uses call by value by default. However, parameters also can be passed with call by reference with using pointers.

```
fmt.Println("********** Parameter passing methods **********")
// Call by Value
var1 := 10
modify1 := func(x int) { x = x * 5 }            // It multiplies the given integer with 5
fmt.Println("Before modify1 call var1: ", var1) // 10
modify1(var1)                                   // It is call by value because var1's value did not change.
fmt.Println("After modify1 call var1: ", var1)  // 10

// Call by Reference
modify2 := func(x *int) { *x = *x * 5 }         // It multiplies the given with 5
fmt.Println("Before modify2 call var1: ", var1) // 10
modify2(&var1)                                  // It is call by reference because var1's value changed
fmt.Println("After modify2 var1: ", var1)       // 50
fmt.Println("*********************************************")
```

4. Keyword and Default Parameters in Go

Keywords and default parameters are not allowed in Go.

```
fmt.Println("********** Keyword and default parameters **********")
keyword1 := func(a int, b int) int { return a + b }
//keyword2 := func(a int, b = 10 int, c = 5 int) {fmt.Println("keyword2:", a + b + c)} It does not work
//fmt.Println(keyword1( b = 30, a = 40)) It does not work
fmt.Println(keyword1(10, 20)) // It only work in this form
//keyword2(10) keyword2 is not working so we can not call it
fmt.Println("*************************************************")
```

5. Closures in Go

Closures are allowed in Go. In the given example, the function is assigned to a variable named “closure1”. Then the “closure1” is called and assigned to a variable named “closure2”. When the “closure2” is called, it will print “You have reached the returned function” in the console. That means “closure2” is equal to the returned function in “closure1”. The second example is also the same. In the third example, “closure5” is equal to the whole function that takes an int variable. Then, the “closure5” is called with integer “100” and assigned to a variable named “closure6”. If “closure6” is called with integer “101”, it will return “-1”. If closure6 is called with integer “102”, it will return “-2”. That means closure6 stored the integer “100” and it will use “100” in the all operations.

```
fmt.Println("********** Closures **********")
// A simple closure
// It return a func with a message
closure1 := func() func() {
    return func() { fmt.Println("You have reached the returned function") }
}

closure2 := closure1()
// Now closure2 is equal to func() { fmt.Println("You have reached the returned function") }
// if we call closure2, it will print "You have reached the returned function"
closure2()

// Another closure example
// The return type must be a func with 2 int paramters and an int.
// Because Go adds the all return types to the parent function if parent function has a func in its return.
closure3 := func() func(int, int) int {
    inner := func(a int, b int) int { return a + b }
    // inner function takes 2 int and sum them up and return it.
    return inner
}

closure4 := closure3()
// Now closure 4 is equal to inner.
// Therefore, if we call it in println, it will print a + b.
fmt.Println(closure4(10, 20)) // it prints 30 because 10 + 20 = 30

// Another closure example
// Outer function takes an int value
// The inner function takes another int value
// And subtract the inner’s int from outer’s int
closure5 := func(a int) func(int) int {
    return func(b int) int { return a - b }
}

closure6 := closure5(100)
// closure6 is now equal to inner function
fmt.Println(closure6(101))
// It will print -1 because 100 - 101 = -1.
fmt.Println(closure6(102))
// It will print -2 because 100 - 102 = -2
// Therefore, closure stores the first value and uses it in other operations.

fmt.Println("*****************************")
```

Readability and Writability

Go has the “func” keyword to create a function. Therefore, it increases the readability because it is similar to other languages. However, in the function declarations, the developer must assign the subprogram to a variable and it decreases the writability. In addition to that, parameter types come after the parameter name and that decreases the readability and writability. In addition to that, return types are also in between parameters and function body and it decreases readability. In addition to that, having to assign subprograms to a variable decreases the writability.

Learning Strategy

I mostly experimented with design issues. However, to do the homework, I had to read the documentation for function declarations and data types [1]. After that, I started to try the design issues. The first design issue was very straightforward. Therefore, writing code for it was not hard. The second design issue was also straightforward. Therefore, I directly wrote a code segment for it and evaluated the results to decide whether it is static or dynamic. For the third design issue, I tried C type parameter passings because in the documentation, examples are given with C comparison. Therefore, I thought Go can have similar passing methods with C. In the fourth design issue, I tried C++ keyword type and default parameters. However, it did not work. Then I searched it online and found out Go does not have default parameters [2]. For the final design issue, I tried closures similar to Javascript’s closures. I used an online Go compiler [3].

References

[1] “Functions,” *Go.* [Online]. Available: https://go.dev/doc/effective_go#functions. [Accessed Dec. 24, 2021].

[1] “How to give default value to function parameters,” *Go Forum.* [Online]. Available: https://forum.golangbridge.org/t/how-to-give-default-value-to-function-parameters/18099. [Accessed Dec. 24, 2021].

[1] “The Go Playground,” *Go.* [Online]. Available: https://go.dev/play/. [Accessed Dec. 24, 2021].