CS315- Programming Languages

Homework 03

Topic: Subprograms in Golang

Ali Eren Günaltılı

21801897

Section 01

Go language subprograms

Introduction to subprograms in Go Language

Subprograms are defined with func keyword in go language. Main is also built-in keyword for main function. Simple function calls in go language is like this.

```
package main
import "fmt"


func test(){
    fmt.Println("test")
}

func main(){
    test()
}
//output
test
```

Generalized function definition in go language:

```
FUNC_KEYWORD FUNC_NAME(PARAMETER-LIST)(RETURN-TYPE){
    //FUNC BODY
}
```

Parameter list takes actual parameters to be used in function body. It takes actual parameters from caller and put the values of actual parameters into formal parameters.

```
func add(x int, y int){
//instead of writing (x int, y int) if types are same it can be written as
// (x,y int)
    fmt.Println(x + y)
}
func main(){
    var a = 11
    var b = 9
    add(a,b)
}
//In this part x and y variables inside of the parameter list of add function
//are formal parameters that use the value of actual parameters a and b.
```

Function can returns value and values in Go language.

Function with return one value

```
func addWithReturn(x,y int)int {
    return x+y
}
func main(){
    var a = 11
    var b = 9
    ans := addWithReturn(a,b)
    fmt.Println(ans)
}
```

To return multiple things parantheses had to be used like (int, int)

```
func addWithReturn(x,y int) (int,int) {
    return x+y, x-y
}
func main(){
    var a = 11
    var b = 9
    ans,ans2 := addWithReturn(a,b) //ans = x+y, ans2 = x-y
    fmt.Println(ans) //ans = 20
    fmt.Println(ans2) //ans2 = 2
}
```

There are different ways to return value in go language, you can predefined a variable name inside of the return-type then assigned desired values to that variable.

```
func addWithReturn(x,y int) (int,int) {
    return x+y, x-y
}
//same thing with addWithReturn
func returnWithVariable(x,y int)(z1, z2 int){
    z1 = x + y
    z2 = x - y
    return //it automatically returns z1 and z2 respect to their position
}
func main(){
    var a = 11
    var b = 9
    ans,ans2 := addWithReturn(a,b)
    b1,b2 := returnWithVariable(a,b)
    fmt.Println(ans)
    fmt.Println(ans2)
    fmt.Println(b1)
    fmt.Println(b2)
}
```

Just like MIPS Assembly language defined the register that holds return value, in this go subprogram variables that is holding the return values are already defined in function definition. There is also one unique attribute of go language's subprograms which is defer. In Go language, defer statements delay the execution of the function or method or an anonymous method until the nearby functions returns. In other words, defer function or method call arguments evaluate instantly, but they don't execute until the nearby functions returns.

```
func addWithReturn(x,y int) (int,int) {
    fmt.Println("Inside of add function")
    return x+ y, x- y
}

//same thing with addWithReturn
func returnWithVariable(x,y int)(z1, z2 int){
    defer addWithReturn(z1,z2)
    z1 = x + y
    z2 = x - y
    fmt.Println("Before return")
    return
}
func main(){
    var a = 11
    var b = 9
    b1,b2 := returnWithVariable(a,b)
    fmt.Println(b1) // b1 = 20
    fmt.Println(b2) // b2 = 2
}
//output
Before return
Inside of add function
20
2
```

As it seenable we don't execute the defer where we see, we wait till we see the return statement. Just before the return statement "defer" is executed.

Go language also provides recursion for functions.

```
func recursion(x int) int{
    if(x == 0){
        return 1
    }
    defer fmt.Println("x is ", x)
    return (x) * recursion(x - 1)
}
func main(){
    fmt.Println("----------------------------")
```

```
    rec := recursion(5)
    fmt.Println("Result: ",  rec)
}
//output
x is  1
x is  2
x is  3
x is  4
x is  5
Result:  120
```

Nested Subprograms In Go Language

As we learned the basic of go language functions we can move into the more advance topics. Nested functions are allowed in go language and it can be reached by using the property of every function is a datatype in go language. What I mean is that,

```
func test(){
    fmt.Println("test")
}
func main(){
    x := test
    x()
}
//output
test
```

Variables can be assigned to function and then called. In this way we can build nested functions.

```
func main(){
    fmt.Println("Beginning")
    y := func(){
        fmt.Println("Inside of nested")
    }
    fmt.Println("Outside of func")
    y()
}
//output
Beginning
Outside of func
Inside of nested
```

Program won't print "inside of nested" before "outside of func", because it considers that sentence as an assigment to y variable. It can only execute what y is, after it is called with y().

Defer which is a predefined statement for go language also can be used in nested function definitions. It is a different way of calling some methods inside of a method. In nested functions inner functions can reach parent's variable but it is not valid for defer statement calls.

```
func test(){
    fmt.Println("test")
}
func main(){
    fmt.Println("Beginning")
    y := func(){
        fmt.Println("Inside of nested")
        defer test()
    }
    fmt.Println("Outside of func")
    y()
    fmt.Println("After function call")
    fmt.Println("--------------------")
}
//output
Beginning
Outside of func
Inside of nested
test
After function call
--------------------
```

Right before returning from y() call, we go to the test() function as defer statement responsible for that call.

```
func nested2(y int){
    x = 45
    fmt.Println(y)
}
func nested1(x int){
    x = 15
    defer nested2(36)
}
```

But defer statement cannot means nested functions actually since it hasn't got all properties of nested function in go language. For example, we get error for this call, because x is not reachable inside of the nested2 function. 'x' is not reachable because they are not really nested functions. Defer only means execute another function or something right before returning. This concept of reaching the parent's variable is related with the scope of a local variable thus I am going to talk this in the other section with nested functions.

Important notes about defer statement and nested functions

```
//nested1 and nested2 created for trying defer statement in multiple functions
//defer can call other function or statement but it is not directly same thing
with
//nested functions.
func nested2(y int){
    //when we try to reach nested1's local variable x inside of nested2
    //we face with error since we reach nested2 with defer statement.
    fmt.Println(y)
}
func nested1(x int){
    x = 15
    defer nested2(36)
}
func nested3(y int) int{
    var x = 15
    nested4 := func(y int){
        x = x * 2
    }
    nested4(y) //now we can reach x and can change it's value
    return x
}
func main(){
    res := nested3(12)
    fmt.Println("Res:", res)
}
//output
Res: 30
```

If we use defer we can't reach parent's variables. But if we directly call the function and properly using the nested function properties of go language we can reach and change parent's variables.

```
func nested3(y int) int{
    var x = 15
    nested4 := func(y int){
        x = x * 2
        fmt.Println("x:",x)
    }
    defer nested4(y)
    return x
}
func main(){
    res := nested3(12)
    fmt.Println("Res:", res)
}//output
X:30
Res: 15
```

Interesting thing is that as we don't go the what defer says till we found a return statement, once we reach the return statement of nested3 function before executing return statement, we already pass the old x value 15 as a return value. Even if we change the x after executing defer statement, it won't affect the value to be returned to main. Thus, defer statement is dangerous.

Scope of a Local Variable and Nested Functions

Before investigating the scope of a local variable we have to know what is a scope in go language. In go language there are three types of scope;

a.Inside a function or a block (local variable)

b.Outside of all functions (global variable)

c.In the definition of function parameter (formal parameters)

```
package main
import "fmt"

//g is global value
var g int = 20

func test(){
    fmt.Println("test")
    fmt.Println("g in test:", g)
}
func main(){
    fmt.Println("--------------------------------")
    test()
    fmt.Println("global g:", g)

}
//output
--------------------------------
test
g in test: 20
global g: 20
```

Go language as I introduced above have 3 type of scoping. Formal parameters also are reachable inside of the function or a block. Thus, we can say that they are similar with local variables. Go language firstly searches local one and then goes for the global. In this program there is no local g or formal parameter g inside of test function, thus it directly reaches the global g and prints it.

1-Local Variable for main but not for test1

```
package main
import "fmt"

//g is global value
var g int = 20
func test(){
    fmt.Println("test")
    fmt.Println("g in test:", g)
}
func main(){
    fmt.Println("--------------------------------")
    var g = 30
    test()
    fmt.Println("g:", g)
}
//output
--------------------------------
test
g in test: 20
g: 30
```

In this program, Go cannot find any local variable inside of the test function. Therefore, it still use the global g above the functions. But in main function we have local g defined with a value 30. Therefore, the program sees the local one first and print 30.

2-Local variable for both test and main functions

```
//g is global value
var g int = 20
func test(){
    var g = 40
    fmt.Println("test")
    fmt.Println("g in test:", g)
}

func main(){
    fmt.Println("--------------------------------")
    var g = 30
    test()
    fmt.Println("g:", g)
}
//output
--------------------------------
test
g in test: 40
g: 30
```

3-Local variable for main and formal parameter for test function

```
package main
import "fmt"

//g is global value
var g int = 20
func test(g int){
    //var g = 40 //if we defined formal parameters with same name, it gives
error
    //by saying g redeclared in this block
    //go allows one unique definition inside of one block
    //but we have a same name with global and local variable for one block
    fmt.Println("test")
    fmt.Println("g in test:", g)
}
func main(){
    fmt.Println("------------------------------")
    var g = 30
    test(60)
    fmt.Println("g:", g)
}
//output
------------------------------
test
g in test: 60 //sees the formal paramater as it is the local one
g: 30 //locally created variable for main function
```

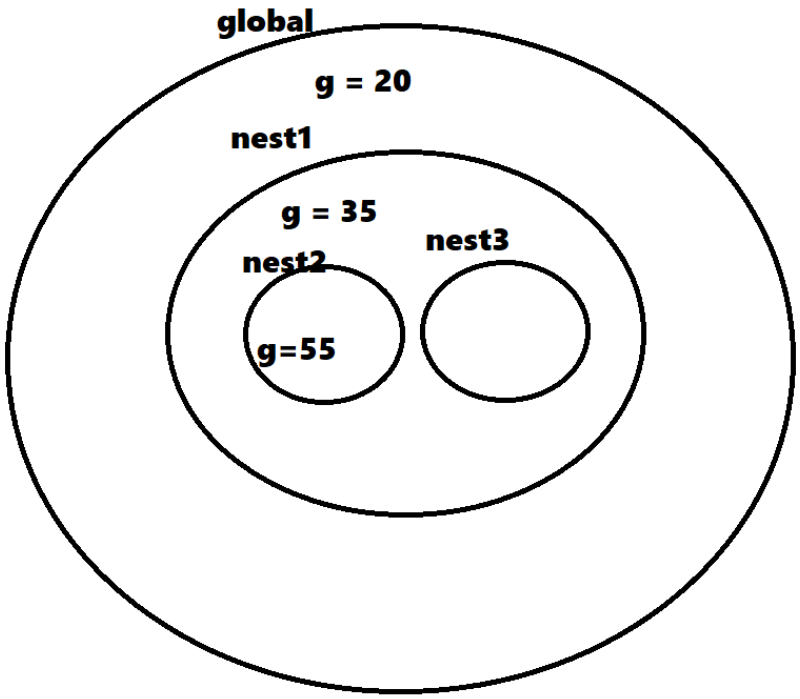
More complex examples on scoping

Example 1:

```
package main
import "fmt"
//g is global value
var g int = 20
func nest1(x int){
    var g = 35
    nest2 := func(x int){
        var g = 55
        fmt.Println("Inside nest2 g:", g)
    }
    nest3 := func(x int){
        fmt.Println("Inside nest3 g:", g)
    }
    nest2(x)
    nest3(x)
```

```
}
func main(){
    nest1(88)
}
//output
Inside nest2 g: 55
Inside nest3 g: 35
```

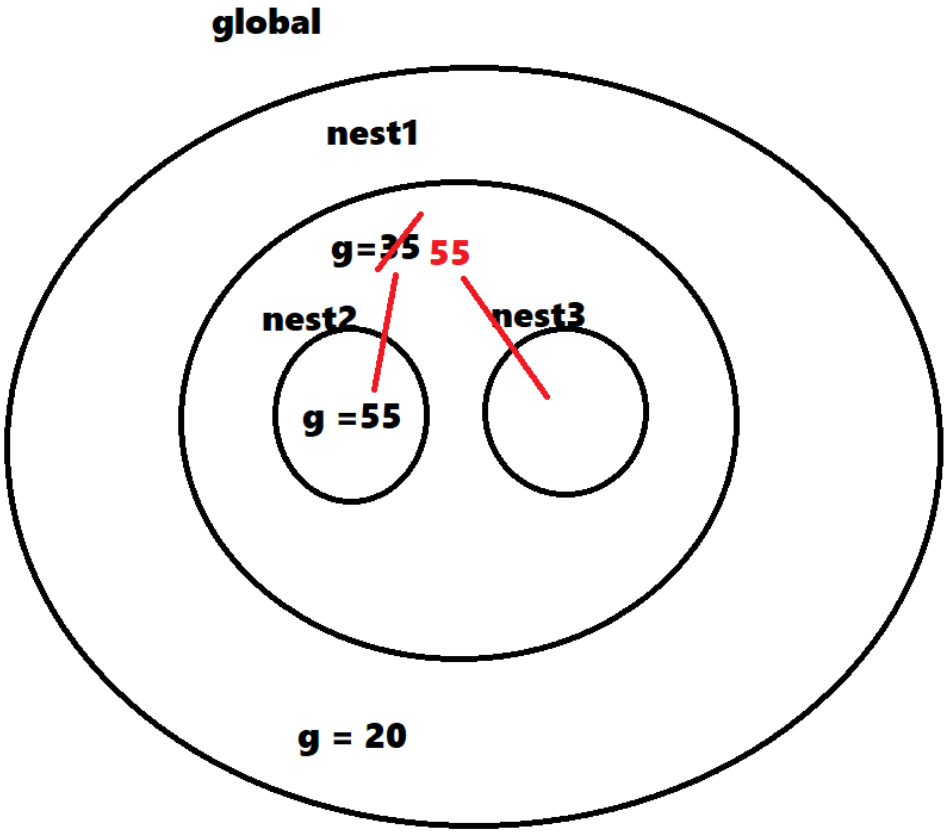
Inside of nest2 it directly reaches the local g but inside in the nest3 as there is no local g it searches inside of the parent which is nest1.



Example 2: Not declaring new variable inside of nest2. Nest2 g operation change parent's value so the result of nest3 too.

```
package main
import "fmt"
//g is global value
var g int = 20
func nest1(x int){
    var g = 35
    nest2 := func(x int){
        g = 55
        fmt.Println("Inside nest2 g:", g)
    }
```

```
    nest3 := func(x int){
        fmt.Println("Inside nest3 g:", g)
    }
    nest2(x)
    nest3(x)
}
func main(){
    nest1(88)
}
//output
Inside nest2 g: 55
Inside nest3 g: 55
```



As there is no new variable definition inside of nest2 unlike previous example, the one in nest2 reaches the g variable inside of the nest1 one. Then, it changes the value to 55. Therefore, nest3 also prints 55 as when nest3 tries to reach g variable, it's value is updated as 55. If we call nest3 before nest2 we reach the 35 as a g value inside of nest3 as we haven't change it yet.

Example 3: Formal Parameter Included

```
package main
import "fmt"

//g is global value
var g int = 20
func test(g int){
    //var g = 40 //if we defined formal parameters with same name, it gives
error
    //by saying g redeclared in this block
    //go allows one unique definition inside of one block
    //but we have a same name with global and local variable for one block
    fmt.Println("test")
    fmt.Println("g in test:", g)
}

func nest1(x int){
    g = 77 //updating the global g
    var g = 35 //creating local g in nest1
    nest2 := func(x int){
        g = 55 //update the g value defined as var g= 35
        fmt.Println("Inside nest2 g:", g)
    }
    nest3 := func(g int){
        fmt.Println("Inside nest3 g:", g) //reaching formal parameter g
    }
    nest2(x)
    nest3(x)
}
func main(){
    fmt.Println("Global g before method", g)
    fmt.Println("---------------------------")
    nest1(88)
    fmt.Println("Global g after method", g)
}
//output
Global g before method 20
---------------------------
Inside nest2 g: 55
Inside nest3 g: 88
Global g after method 77
```

In nest3 function, we can reach the formal parameter g as it is the locally presented for function. To sum up, scope of a local variable is restricted for it's block or function. Local variables cannot be reachable outside of their block or function. Local variable of a nest1 however will be available for nest2 and nest3

because it is their parent. In another word, in nested functions, children can see parent's local variables and even change it just like nest2 did it in our example. Another important issue about scoping in go language is that, Go starts its search local one to global one. It search locally if it couldn't find anything then search in parent and so on.

Important Note: We can't present a formal parameter and a local variable inside of the same function.

```
14  func error(x int){
15      var x = 15
16  }
```

Error Message : .\scope.go:15:6: x redeclared in this block

Parameter Passing Methods

Go language provide two different parameter passing methods as call by value and call by reference. Default one is call by value actually. In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

```
func swap(x,y int){
    var tmp int
    tmp = x
    x = y
    y = tmp
    fmt.Println("Inside call-by-value swap x",x)
    fmt.Println("Inside call-by-value swap y", y)
}
func main(){
    fmt.Println("-------------------------")
    fmt.Println("BELOW PART IS FOR PARAMETER PASSING METHOD")
    fmt.Println("Call by value part")
    var par1 = 88
    var par2 = 44
    fmt.Println("Before call-by-value swap x", par1)
    fmt.Println("Before call-by-value swap y", par2)
    swap(par1,par2)
    fmt.Println("After call-by-value swap x", par1)
    fmt.Println("After call-by-value swap y", par2)
}
//output
-------------------------
```

```
BELOW PART IS FOR PARAMETER PASSING METHOD
Call by value part
Before call-by-value swap x 88
Before call-by-value swap y 44
Inside call-by-value swap x 44
Inside call-by-value swap y 88
After call-by-value swap x 88
After call-by-value swap y 44
```

As it is obvious value of x and y don't change after the swap function. Change only takes place inside of the swap method because go language uses call by value parameter passing method if it is not specificly indicated with * operator.

```
func swapRef(x,y *int){
    var tmp int
    tmp = *x
    *x = *y
    *y = tmp
    fmt.Println("Inside call-by-reference swap x",x)
    fmt.Println("Inside call-by-reference swap y", y)
}
func main(){
    fmt.Println("Call by reference part")
    par1 = 100
    par2 = 200
    fmt.Println("Before call-by-reference swap x", par1)
    fmt.Println("Before call-by-reference swap y", par2)
    swapRef(&par1,&par2)
    fmt.Println("After call-by-reference swap x",par1)
    fmt.Println("After call-by-reference swap y", par2)
}
//output
Call by reference part
Before call-by-reference swap x 100
Before call-by-reference swap y 200
Inside call-by-reference swap x 200
Inside call-by-reference swap y 100
After call-by-reference swap x 200
After call-by-reference swap y 100
```

When pointer value created as formal parameter and able to hold address of an actual parameter, it means program can change actual parameter's value permanently.

```
func swap(x,y int){
    var tmp int
    tmp = x
    x = y
    y = tmp
```

```
    fmt.Println("Address of x in swap method call by value", &x)
    fmt.Println("Address of y in swap method call by value", &y)
    fmt.Println("Inside call-by-value swap x",x)
    fmt.Println("Inside call-by-value swap y", y)
}

func swapRef(x,y *int){
    var tmp int
    tmp = *x
    *x = *y
    *y = tmp
    fmt.Println("Address of pointer x in reference method", x)
    fmt.Println("Address of pointer y in reference method", y)
    fmt.Println("Inside call-by-reference swap x",*x)
    fmt.Println("Inside call-by-reference swap y", *y)
}
func main(){
    fmt.Println("Call by value part")
    var par1 = 88
    var par2 = 44
    fmt.Println("Before call-by-value swap x", par1)
    fmt.Println("Before call-by-value swap y", par2)
    fmt.Println("Address of x in main", &par1)
    fmt.Println("Address of y in main", &par2)
    swap(par1,par2)
    fmt.Println("After call-by-value swap x", par1)
    fmt.Println("After call-by-value swap y", par2)
    fmt.Println("Call by reference part")
    par1 = 100
    par2 = 200
    fmt.Println("Address of x in main", &par1)
    fmt.Println("Address of y in main", &par2)
    fmt.Println("Before call-by-reference swap x", par1)
    fmt.Println("Before call-by-reference swap y", par2)
    swapRef(&par1,&par2)
    fmt.Println("After call-by-reference swap x",par1)
    fmt.Println("After call-by-reference swap y", par2)
}
//output
Call by value part
Before call-by-value swap x 88
Before call-by-value swap y 44
Address of x in main 0xc000014180
Address of y in main 0xc000014188
Address of x in swap method call by value 0xc000014190
Address of y in swap method call by value 0xc000014198
Inside call-by-value swap x 44
Inside call-by-value swap y 88
After call-by-value swap x 88
```

```
After call-by-value swap y 44
Call by reference part
Address of x in main 0xc000014180
Address of y in main 0xc000014188
Before call-by-reference swap x 100
Before call-by-reference swap y 200
Address of pointer x in reference method 0xc000014180
Address of pointer y in reference method 0xc000014188
Inside call-by-reference swap x 200
Inside call-by-reference swap y 100
After call-by-reference swap x 200
After call-by-reference swap y 100
```

Important thing about this example program is that the address value of variables in the program and address values of formal parameters in functions.

In call by value swap method: New formal parameters are created with different address value as the output proves.

```
Address of x in main 0xc000014180
Address of y in main 0xc000014188
Address of x in swap method call by value 0xc000014190
Address of y in swap method call by value 0xc000014198
```

That means, we only copy the value of x from main and put that value inside of formal parameter. Formal parameter is also created with different address value and as we discussed previous Go searches for the most local one. When it sees the formal parameter it stop its search and change only formal parameter value. We never have a chance to reach actual x and change its value by using call by value.

However in call by reference swap method pointer directly holds the address of actual parameters to reach them.

```
Call by reference part
Address of x in main 0xc000014180
Address of y in main 0xc000014188
Before call-by-reference swap x 100
Before call-by-reference swap y 200
Address of pointer x in reference method 0xc000014180
Address of pointer y in reference method 0xc000014188
```

Keyword and Default Parameters

Go language doesn’t support default parameters passing. There are few way to provide a similar thing to default parameters but that always depends on the user. The language itself won’t provide directly unlike python or most modern languages. Go language also doesn’t support keyword to be used in parameter passing. There is no thing in go language like,

```
 def bar(a,b,c):
        print(a,b,c)
args={c: 1, b: 2, a: 3}
bar(**args)
```

```
func default(var y = 1){
//default parameters are not supported in Go language
}
//Error
.\main.go:99:6: syntax error: unexpected default, expecting name or (
.\main.go:160:2: syntax error: unexpected default, expecting }
```

But user can built similar system for itself like this.

```
func defaultMethod(y int){
    if y == -99 {
        fmt.Println("Succesfull")
    } else{
        fmt.Println("Unsuccesfull")
    }
}

func main(){
    var def = -99
    defaultMethod(def)
}
//output
Succesfull
```

Programmer can manipulate provided functionalities of go language to provide keyword or default parameters for language. But as I said before, language itself is not supporting these two things.

Subprograms as a Parameter and DataType

Go language considers the function as a datatype. Functions can be assigned to some variable or passed as a parameter to function.

```
package main
import "fmt"
func nested3(y int) int{
    var x = 15
    nested4 := func(y int){
        x = x * 2
        fmt.Println("x:",x) // x: 30
    }
    defer nested4(y)
```

```
    return x //returning 15 because defer is executed after
//seeing the return statement. Thus, x value returns as 15 not 30.
}
func funcParam(myFunc func(int)int){
    fmt.Println(myFunc(7)) //prints 15 as myFunc(7) returns 15
}
func main(){
    fmt.Println("BELOW PART IS CLOSURES AND FUNCTIONS AS A SUBPROGRAM")
    funcParam(nested3)
} //functions as a function parameter
//output
BELOW PART IS CLOSURES AND FUNCTIONS AS A SUBPROGRAM
x: 30
15
//-----Another example to show different ways of calling function------//
func main(){
    func(x int){
        fmt.Println("Function is called after its creation", x )
    }(16) //this parantheses and parameters calls the anonymous func.
}
//output
Function is called after its creation 16
```

We can call a function without using its name. Moreover in go language there can be anonymous function definition just like this example. Our function doesn't have a name but we can call it after its creation.

Closures

```
//closure part
func closure(x string) func(){
    return func(){ fmt.Println(x)}
}
func main(){
    closure("HELLO")()
}
//output
HELLO
//we can also called closure by storing into variable like this
func main(){

    funcTmp := closure("HELLO")

    funcTmp()

}
//Output will be the same since these two ways are same.
```

This output is reasonable because when we say closure(“HELLO”) we expect to get some return type. Our return type is some function with a empty parameter list and void return type. Thus, writing closure(“HELLO”)() and writing return func(){ fmt.Println(x)}(“HELLO”) inside closure method is same thing.

```
package main
import "fmt"
var i int = 40
func returnFunc() func() int{
    return func() int{
        i++
        return i
    }
} //returnFunc is a function that takes no parameters and returns
//some function which returns int
func main(){
    nextInt := returnFunc() //this means that nextInt = func() int
//if you fmtPrintln(func() int) => you get some integer
//as the function is going to return some integer.
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    updated := nextInt
    fromBeginning := returnFunc
    fmt.Println("Already defined", updated())
    fmt.Println(fromBeginning()())
}
//output
41
42
43
Already defined 44
45
```

As we are dealing with global value i, it keeps increasing.

```
package main
import "fmt"

//g is global value
var g int = 20
var i int = 40


func returnFunc() func() int{
    i := 0
    return func() int{
        i++
        return i
```

```
    }
}

func main(){
    nextInt := returnFunc()
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    updated := nextInt
    fromBeginning := returnFunc
    fmt.Println("Already defined", updated())
    fmt.Println(fromBeginning()())
}
//fromBeginning := returnFunc then print(fromBeginning()())
//fromBeginning := returnFunc() print(fromBeginning()) are same things.

//output
1
2
3
Already defined 4
1
```

If we don’t use global variable and make i local inside of the function, output will change according to that. We start to count from 0 this time as i initially sets up to 0. More dramatic change occurres on the

fromBeginning := returnFunc part. When we say this, it automatically calls the function again and assigned to variable on the left side. Thus, i is 0 again.

Evalution of Go Language

Go language does not seem so strange at first look. The way it uses func as a built-in keyword or using parentheses for function declaration is a widely chosen way of defining a function. Although it has lots of edges on writing, it also has problems with it. Some errors that I faced when I tried to write my program were a headache for me. For example, if and else syntax is too confusing to be understood by a beginner. Why else needs to be in the same line with closing brackets of if. I wouldn't expect this from a language like Go which can be categorized as new maybe. I am also have disappointed in the default parameters issue. Go language unlike most modern languages doesn't support default parameters or keyword parameter passing. One of my favorite things about goes language that I learned in this assignment was probably the defer statement. It is useful and it comes built-in. Go language also has some problems with the variable declaration in the case of readability and writability. Sometimes, I use:=, sometimes var, or sometimes I directly declared my variables. This probably happened because I don't have any experience with it before. It is flexible in case of variable declarations or subprogram use. You can use a subprogram as a parameter or variable. One of the small things that I liked about Go was the easiness of importing some libraries. I only import one library as fmt to use println method but the syntax for importing new libraries seems too easy in comparison to any other language.

I consider the Go language as hard to write. The function definition is confusing especially that return type part. Why doesn't it use similar syntax like C-group languages for return type? But I liked that the Go language can return multiple values and it can do that in different ways. It is very flexible as I said multiple times, there is more than one syntax to do one job in the Go language. One of my favorites was returning multiple variables like this func returnWithVariable(x,y int)(z1, z2 int).

Using comma for multiple returns or multiple assignments is also confusing but this property is really good and eases the programmer's job. To sum up, Go is very flexible but I wouldn't choose it if I had a chance as it has a confusing syntax and is not readable due to function definitions, if-else statements or assignment statements, etc.

My Learning Strategy

I started my homework from youtube by watching some basics of the Go language as I had no idea what it is. I only heard that it is useful for backend web development from some of my friends. After I watched the required things for my homework, I tried to write codes on my own. I faced lots of errors in this part and it is very frustrating. Even the IDE that I downloaded for writing Go language GoLand was not working. Then I used an online compiler for writing basic stuff. After I started to understand how to write and run my code on VsCode, I started to learn subprograms, particularly in the go language. As I already took some notes from youtube tutorials, I didn't have to read lots of documents from web pages. I mostly use the web for solving the error that I faced. I solved my problems by checking StackOverflow. Stackoverflow was one of the helpful sources in Go Language for me. The official page of Go Language and some sites that directly share code examples on the Internet were also helpful.

When I started to work on my homework I firstly added some introduction parts about the Go language and properties of subprograms. Because I also learned what to do by learning the basics. After I added my introduction, I tried to solve the question of the homework by reading in the first place. After I read and learn the solution for those questions, I started to try myself on VsCode. After I successfully ran my example programs on VsCode, I started to answer questions on the report. I did the same thing for all of the questions because I felt like this homework was code dominant homework unlike the other two. I felt like I had to write the code for understanding the mechanism then put the results into the report. My learning process was this. Firstly I check official documents or youtube for what is Go, how should I write it, what is the syntax or what is the answers to questions in homework. Then, I wrote code according to the concepts of questions. Finally, I transmitted my results, codes, and answers to my report.

References

Code Examples and Documentation: https://gobyexample.com

Official web site: https://go.dev/

Website for tutorials of Go Language: https://www.tutorialspoint.com/go/index.htm

YouTube tutorials and slides about Go Language: https://www.techwithtim.net/

Helpful site for errors and many problems: https://stackoverflow.com/

Site for documentation and information about various programming languages: https://www.w3schools.com/default.asp

Shell scripting information about Unix: https://www.tutorialspoint.com/unix/shell_scripting.htm