CS315 Homework 3

Rüzgar Ayan 21801984

1 Nested subprogram definitions

First, to understand how functions work in Golang, I've tried the following:

```
    /* It is not possible to define subprograms in this way
    func sayHi (name string) {
        fmt.Println("Hi", name)
    }
    */

    //We need to assign a function into a variable.
    sayHi := func(name string) {
        fmt.Println("Hi", name)
    }
    sayHi("Rüzgar")
    fmt.Println("Type of sayHi:", reflect.TypeOf(sayHi))

    square := func(num int) int {
        return num * num
    }
    fmt.Println("square(5):", square(5))
    fmt.Println("Type of square:", reflect.TypeOf(square))
```

We cannot define nested subprograms as in Javascript or PHP directly. **Instead, in a slightly different way, we create a function object and assign it to a variable as seen above [1]**. I've also tried to output the types of these function variables[2]. That part has the output

```
Hi Rüzgar
Type of sayHi: func(string)
square(5): 25
Type of square: func(int) int
```

It is also possible to directly call a subprogram after its definition as below. I cannot see where this might be helpful other than creating closures.

```
    //Can directly define a function and call it, but not very useful
    func(message string) {
        fmt.Println("Inside the directly called function. Message:",
message)
    }("a special message")
```

As expected, we can create nested subprograms inside of other nested subprograms. Since these functions can be assigned to variables too, we can return functions from functions:

```
//This function returns another function to be used later
generateHiFunction := func(name string) func() {
    return func() {
        fmt.Println("Hi", name, ",from the generated function")
    }
}

sayHiRuzgar := generateHiFunction("Rüzgar")
sayHiRuzgar() //Prints "Hi Rüzgar ,from the generated function"
```

Finally, nested subprograms can change the values of variables in the upper levels:

```
//Subprograms with side-effects
num := 2
squareAndPrint := func() {
    num = num * num
    fmt.Println("After squaring, num:", num)
}

squareAndPrint() //4
squareAndPrint() //16
squareAndPrint() //256
```

2 Scope of local variables

According to the official documentation on scope [3], "Go is lexically scoped using blocks". For local variables, "their scope is only in the body of the function" [4]. The example below shows how nested subprograms can use the local variables of the higher level subprograms:

```
foo := "Main"
subprogram1 := func() {
    foo = "SP1"
    subprogram2 := func() {
        foo = "SP2"
        fmt.Println("At SP2, foo:", foo)
    }
    fmt.Println("At SP1, foo:", foo)
    subprogram2()
    fmt.Println("At SP1 after calling SP2, foo:", foo)
}
fmt.Println("At Main, foo:", foo)
subprogram1()
fmt.Println("At Main after calling SP1, foo:", foo, "\n\n")
```

The output for this is:

```
At SP1, foo: SP1
At SP2, foo: SP2
At SP1 after calling SP2, foo: SP2
At Main after calling SP1, foo: SP2
```

The modifications in the inner subprograms can affect the outer local variables. Also, the code snippet below tests whether dynamic scoping is used in Golang.

```
//Checking if static or dynamic scoping is used
    foo = "Main"
    subprogram1 = func() {
        fmt.Println("At SP1, foo:", foo)

        //Gives undefined: specialVariable at compilation
        //If it were dynamic, would not be able decide this until runtime
        //fmt.Println("At SP1, specialVariable :", specialVariable)
    }
    subprogram2 := func() {
        foo := "SP2" //Define new local variable foo
        specialVariable := "specialVariable"
        fmt.Println("At SP2, foo:", foo)
        fmt.Println("At SP2, specialVariable:", specialVariable)
        subprogram1()
    }
    subprogram2()
```

If there was dynamic scoping, the foo would have the value of “SP2” at subprogram1 when called by subprogram2. Also, it is not possible to access the local variables created by the caller function, “specialVariable” in this case cannot be accessed from subprogram1. We directly get a compilation error, which suggests dynamic scoping is not used. The output is:

```
At SP2, foo: SP2
At SP2, specialVariable: specialVariable
At SP1, foo: Main
```

3 Parameter passing methods

In the official documentation about function calls [5], it is told that “In a function call, the function value and arguments are evaluated in the usual order. After they are evaluated, the parameters of the call are passed by value to the function and the called function begins execution. The return parameters of the function are passed by value back to the caller when the function returns”. **This suggests that only pass by value is used in Golang.** We test this with the following programs.
In the first examples below, we see that the value of the variable does not change after the function call.

```
    num = 10
    modifyNum := func(numIn int) { numIn = 100 }
    fmt.Println("num before modification:", num)
    modifyNum(num)
    fmt.Println("num after modification:", num)
    fmt.Println()

    nums := [5]int{1, 2, 3, 4, 5}
    modifyArray := func(arr [5]int) { arr[0] = 100 }
    fmt.Println("nums before modification:", nums)
    modifyArray(nums)
    fmt.Println("nums after modification:", nums)
    fmt.Println()
```

The output is:

```
num before modification: 10
num after modification: 10

nums before modification: [1 2 3 4 5]
nums after modification: [1 2 3 4 5]
```

Just like in C, Golang has pointers. We can pass pointers to functions as parameters. But this is not to be confused with pass by reference, we are still copying that pointer value while passing it through.

```
   //Different than the array above, slice holds a pointer internally,
    //so when passed into a function by a value, we can still modify the
actual elements through the pointer
    numsSlice := []int{1, 2, 3, 4, 5}
    modifySlice := func(slice []int) { slice[0] = 100 }
    fmt.Println("nums before modification:", numsSlice)
    modifySlice(numsSlice)
    fmt.Println("nums after modification:", numsSlice)
    fmt.Println()
```

This time we can modify the array's content as seen in the output below. This is not because of pass by reference, but because we are passing the actual address of that array as a parameter.

```
nums before modification: [1 2 3 4 5]
nums after modification: [100 2 3 4 5]
```

Similarly, we can use the pointers more explicitly as in the example below

```
   //This is still not pass by reference since pointers are also copied
with pass by value.
    num = 10
    modifyNumByPointer := func(numIn *int) { *numIn = 100 }
    fmt.Println("num before modification by pointer:", num) //10
```

```
    modifyNumByPointer(&num)
    fmt.Println("num after modification by pointer:", num) //100
    fmt.Println()
```

4 Keyword and default parameters

Golang does not support keyword input parameters, however, it supports a similar feature for the return parameters [6]. This makes sense because Golang can have multiple return parameters. In the example below, instead of writing “return square, cube” at the end of the function, we name those output parameters and assign their values inside the functions in any order we want.

```
   //Keyword parameters are not supported, but there is a similar feature
for output parameters
    squareAndCube := func(num int) (square int, cube int) {
        square = num * num
        cube = square * num

        //Return empty and the current values of square and cube will be
returned
        return
    }
    squareOf5, cubeOf5 := squareAndCube(5)
    fmt.Println("Square and cube of 5:", squareOf5, cubeOf5) //25 125
```

For the default parameters, there is no support at all [7]. A similar result can be obtained by checking the parameters in the beginning and giving their default values manually. But this same thing can be done for any programming language and is special or easy to us at all.

```
   //Default parameters are not supported
   //We can only simulate the behaviour by passing empty inputs such as
empty string or nil
    simulateDefault := func(name string) {
        if name == "" {
            name = "Rüzgar"
        }
        fmt.Println("name is:", name)
    }

    simulateDefault("Ali")
    simulateDefault("") //Prints Rüzgar
```

5 Closures

In Golang, closures are implemented as usual. In the official tutorial of Golang [8], it is told that "**Go functions may be closures.** A closure is a function value that references variables from outside its body. The function may access and assign to the referenced variables; in this sense the function is 'bound' to the variables". We try this in the following code snippet:

```
sequenceGenerator := func() (func(), func()) {
    i := 0

    increaseBy1 := func() {
        i = i + 1
        fmt.Println("Increased by 1, current value:", i)
    }

    increaseBy2 := func() {
        i = i + 2
        fmt.Println("Increased by 2, current value:", i)
    }

    return increaseBy1, increaseBy2
}

increaseBy1, increaseBy2 := sequenceGenerator()

increaseBy1() //1
increaseBy1() //2
increaseBy1() //3
increaseBy2() //5
increaseBy2() //7
increaseBy1() //8
```

Even though the sequenceGenerator() function ends, its local variable "i" continues to live because the returned functions depend on it.

6 Evaluation of Readability and Writability

I have found a lot of nice features from different languages in Golang. Firstly, the usage of ":=" which can do declaration and assignment at a single line highly increases writability. Similarly, multiple assignments at a single line used together with the multiple return parameters of the functions increase both the readability and writability. About the nested subprogram definitions in the first design issue, I was surprised that we cannot define them as normal functions in PHP or Javascript but we must assign these defined functions to variables as the only way.

Having keywords would be a plus, I think it is a huge plus for both readability and writability and I don't know for what reasons Golang choose not to have them. The feature of named

return parameters that I explained in the design issues is not very useful too. Instead, I think it might even be confusing in long functions because we may forget the last assignment of one of these named outputs. Another feature that Golang didn't have was default parameters. From my perspective, default parameters actually harm the readability and writability because we might need to change the order of these parameters to give them default values and also function calls become confusing with more than one way to give parameters. So, I am glad that there is no default parameters feature.

Also, usage of pointers in this language of course makes it harder to read and write but it also increases the capabilities of the language.

7 Learning Strategies

I noticed that the official documentation for Golang was much more explanatory compared to the other programming languages. I found most of the answers very clearly stated in these documentation pages. Also, the tutorial of the Golang language (https://go.dev/tour/) was very informative and there was even a question answer for this homework. This tutorial was using the language's official online compiler, which I also used while doing this tutorial.

After learning the programming language through the tutorial, I looked into the textbook to understand what kind of different design choices there are for each of the questions. That way, I could try with small programs to see which of these design choices were made. I often found the correct answers through documentation or some external tutorials, but I still verified these with my own code. However, in section 2 "Parameter passing methods", I initially thought both pass by value and pass by reference were used. Then I found out that this was a little bit tricky and the usage of pointers did not mean there were pass by reference. For that, I actually looked into the properties of C programming language which also used pointers.

For lacking features such as keyword and default parameters, I tried to find out how these features can be simulated with normal functionalities. This way, I tried to understand why the designers of the language made these design choices.

For the question about the usage of keywords, I couldn't find an answer from official documentation. First time in these homeworks, I've found a discussion page of the Golang developers where they stated the answer to my question. I guess this is because the programming languages in the other homeworks were much older than Golang.

References

[1] "Nested Functions · GoLang Notes - sharbeargle",
https://sharbeargle.gitbooks.io/golang-notes/content/nested-functions.html

[2] "How to find the type of an object in Go? - Stack Overflow",
https://stackoverflow.com/questions/20170275/how-to-find-the-type-of-an-object-in-go

[3] “The Go Programming Language Specification - Golang”, https://go.dev/ref/spec#Declarations_and_scope

[4] “Scopes in Go (Examples) | Learn Go Programming”, https://golangr.com/scope/

[5] “The Go Programming Language Specification - Golang”, https://go.dev/ref/spec#Calls

[6] “Named Return Parameters in Golang - GeeksforGeeks”, https://www.geeksforgeeks.org/named-return-parameters-in-golang/

[7] “Default function arguments”, https://groups.google.com/g/golang-nuts/c/3Go3gPNIPaw/m/Pv2B5udjDgMJ

[8] “Function closures - A Tour of Go”, https://go.dev/tour/moretypes/25