Turgut Alp Edis

CS315-2

21702587

CS315 Homework #3

In this report, the Golang will be investigated in terms of following design issues.

- Nested subprogram definitions
- Scope of local variables
- Parameter passing methods
- Keyword and default parameters
- Closures

Nested subprogram definitions:

Input:

```
//Rec_fibo function is used to explain the design issue of nested subprogram definitions.
func rec_fibo(n int) int{
    var fibo func(int) int
    fibo = func(x int) int{
        if x == 0 {
            return 0
        }
        if x == 1 {
            return 1
        }
        return fibo(x-1) + fibo(x-2)
    }
    return fibo(n)
}
```

```
fmt.Println("rec_fibo is the recursive fibonnacci method ")
fmt.Println("that explains the design issue of the nested subprogram")
fmt.Println("The rec_fibo is returning the function fibo.")
fmt.Println("The fibo is a function that calculate the fibonacci result of the given number.")
fmt.Print("The fibonacci of 10 is ")
fmt.Println(rec_fibo(10))
fmt.Println("------------------------------------------------------------")
```

Result:

```
rec_fibo is the recursive fibonnacci method that explains the design issue of the nested subprogram
The rec_fibo is returning the function fibo.
The fibo is a function that calculate the fibonacci result of the given number.
The fibonacci of 10 is 55
------------------------------------------------------------
```

In the code above, the recursive Fibonacci function is created to answer the nested subprogram definition design problem. It simply takes the integer value and returns the inner function with same parameter. Then, the inner function returns itself to calculate rest of the value until the parameter becomes zero. This code indicates that the nested functions are allowed in the Golang.

Scope of local variables:

Input:

```
//Below x initialization local and mult functions are used to explain the design issue of scope of local variables.
//Global x variable
var x int = 56

func local() {
    //Local variable x
    var x int = 15
    fmt.Print("Local X inside the local function is ")
    fmt.Println(x)
}

func mult(y int, z int) int {
    fmt.Printf("The y inside the mult function is %d\n", y)
    fmt.Printf("The z inside the mult function is %d\n", z)
    return y * z
}
```

```
local()
fmt.Print("Global X is ")
fmt.Println(x)
var a int = 11
//Local variable x
var x int = 15
fmt.Println("When the function is called, the variable in the local scope is going to be used")
fmt.Println("This is called formal scope in golang")
fmt.Printf("So, the result of x * a will be %d\n", mult(x,a))
fmt.Println("------------------------------------------------------------")
```

Result:

```
Local X inside the local function is 15
Global X is 56
When the function is called, the variable in the local scope is going to be used
This is called formal scope in golang
The y inside the mult function is 15
The z inside the mult function is 11
So, the result of x * a will be 165
```

In the code above, the x is created as global variable in the first place and the function local is created just to show the local scope of the variable. Also, the formal scope is available, which is when the function is called, variable that is in more local scope is going to be used. Thus, there are three scopes in Golang.

Parameter passing methods:

Input:

```
//setXPbV and setXPbR functions are used to explain the design issue of parameter passing methods.
func setXPbV(x int) {
    x = 66
}

func setXPbR(x *int) {
    *x = 45
}
```

```
fmt.Println("Pass by value and Pass by reference is possible in golang")
fmt.Println("The pass by value to the function cannot be modified since it is in formal scope")
fmt.Printf("The value before the function call with pass by value is %d\n", x)
setXPbV(x)
fmt.Printf("The value after the function call with pass by value is %d\n", x)
fmt.Println("However, when the function takes the reference, the value can be modified")
fmt.Printf("The value before the function call with pass by reference is %d\n", x)
setXPbR(&x)
fmt.Printf("The value after the function call with pass by reference is %d\n", x)
fmt.Println("------------------------------------------------------------")
```

Result:

```
Pass by value and Pass by reference is possible in golang
The pass by value to the function cannot be modified since it is in formal scope
The value before the function call with pass by value is 15
The value after the function call with pass by value is 15
However, when the function takes the reference, the value can be modified
The value before the function call with pass by reference is 15
The value after the function call with pass by reference is 45
```

In the above code, there are two methods called setXPbV and setXPbR which are the short form of pass by value and pass by reference respectively. There are two types of parameter passing methods in Golang, which are pass by value and pass by reference. Pass by value does not implement any change of the parameter value, while pass by reference can change the parameter value. Pass by value cannot change the parameter since the parameter is the formal scope inside the function.

Keyword and default parameters:

Input:

```go
//sumList function is used to explain the design issue of keyword and default parameters.
func sumList(numbers ...int) int {
    n := 0
    for _, i := range numbers{
        n += i
    }
    return n
}
```

```go
fmt.Println("Golang does not support default and optional parameters and method overloading")
fmt.Print("However it accepts variadic function which is basically ")
fmt.Println("unlimited variables with the same type and the type is indicated at the end")
fmt.Println("sumList function can take unlimited variables")
fmt.Printf("sumList() is %d\n", sumList())
fmt.Printf("sumList(56,89) is %d\n", sumList(56,89))
fmt.Printf("sumList(1,25,46,77) is %d\n", sumList(1,25,46,77))
```

Result:

```
Golang does not support default and optional parameters and method overloading
However it accepts variadic function which is basically unlimited variables with the same type and the type is indicated at the end
sumList function can take unlimited variables
sumList() is 0
sumList(56,89) is 145
sumList(1,25,46,77) is 149
```

In the above code, there is a method called sumList which takes infinite number of integers and calculate their sums. The keyword and default parameter is not accepted in Golang, but the function with infinite parameters can be accepted if and only if the last parameter is the type.

Closures:

Input:

```
//anon_func function is used to explain the design issue of closures.
func anon_func() func() int{
    x := 1
    return func() int {
        x += 2
        return x
    }

}
```

```
fmt.Println("The golang supports the closure with the anonymous function")
fmt.Print("Anonymous function is the function returns another function with no variable name ")
fmt.Println("so the name of the function will be the name of the variable")
anon := anon_func()
fmt.Printf("The result of the anon_func which returns anonymous function is %d\n", anon())
```

Result:

```
The golang supports the closure with the anonymous function
Anonymous function is the function returns another function with no variable name so the name of the function will be the name of the variable
The result of the anon_func which returns anonymous function is 3
```

In the above code, the function called anon_func creates the anonymous function and returns it as anonymous. Anonymous means that it has not any name, therefore it takes the name of the variable. So, the closure is supported in Golang.

Evaluation of Golang:

Golang is not good enough in terms of writability of the subprogram syntax since the logic of the functions is so complex that they result in error easily when the purpose of the subprogram is not well defined. Besides, the parameter passing methods are not common in different languages, therefore everybody can miscalculate the work when they use the subprograms. Also, the absence of default parameters and keywords can reduce the writability since they make easier the writability of the code.

In addition, Golang is not good enough in terms of readability of the subprogram syntax. The reason is that when other people read the code, it is possible that they understand nothing but not because of the absence of the knowledge about Golang, because the code could be too complex in some places that the complexity makes impossible to read the code.

Learning Strategy:

When I started the homework, initially I read the first, which aims to introduce the language documentation of Golang, since unfortunately I have not any knowledge about Golang. For learning strategy, I read the documentation from the official website for documentation such as Go.dev, then I took a look at example codes from this site which can be found on go.dev/docs and found on the educational websites such as W3Schools to learn how to code the specific language. After reading the documentation, I tried to implement the code to answer the design issues. Since I had to learn just small part of the language, which is subprograms and how to compile and run the code, the learning time is relatively short. However, before writing the code, I needed to learn how to run the compile and run the code, also what packages I need since without that knowledge, I cannot compile and run the code. After gaining this knowledge, I started to write the code for explain the design problem questions. I used Visual Studio Code to write and run the code. Besides, I used terminal to run the code because the extensions sometimes occurred with an error. Since I used Ubuntu as Operating System, working with terminal is easier than Windows since Ubuntu allows users to work more freely. While I was writing the code, I run the code regularly to test answers of the design problem questions. I tried to run possible answers for each design issue, and I tried to find correct answer. When the code was compiled and run truly, I adjusted the comment sections to indicate the which line belongs to which question. Then, the code part of the homework finished.
To run the code, first the go is needed. It can be downloaded from here After the go is installed, the path of the file is opened via terminal and it is enough to write “go run Edis_Turgut.go” is enough to run the code.