CS 315 Homework 3

Subprograms in Go Language

Student Name: Esad İsmail Tök

Student ID: 21801679

Section: 3

Design Issue 1: Nested Subprogram Definitions

In Go language nested functions are allowed. Namely it is possible to define functions inside other functions. It can be done in different ways [1].

First of all, a function can be defined and then automatically executed without any call. It is an anonymous function since the function has no name.

```
// Anonymous function that is executed within the definition phase
func(x int, y int) {
  fmt.Println("The addition of the parameters ", x, "and ", y, " is ", x+y)
}(5, 7)
```

Figure 1: First method of declaring nested functions in Go language

```
The addition of the parameters  5 and  7  is  12
------------------
```

Figure 2: Output of the first method of declaring nested functions in Go language

The function in *Figure 1* is defined inside the main function so it is a nested function. It is directly called when the program execution comes to the parameter list after the function definition. *Figure 2* is the output of the print statement.

As the second nested subprogram definition, a nested function can be defined inside the outer function and then it can be assigned to a variable that holds the function. It is also an anonymous function [1]. This function variable can then be called with the parameter list and executed. Unlike the first way of defining a nested function, it does not automatically executed at the same time of definition instead it is called by the programmer.

```
// Anonymous function that is assigned to a variable and then used when it is called
subtractFunct := func(first int, second int) {
  fmt.Println("This is a subtraction function and the result with the parameters ", first, "and ", second, " is ", first-second)
}

subtractFunct(17, 7)
```

Figure 3: Second method of declaring nested functions in Go language

```
This is a subtraction function and the result with the parameters  17 and  7  is  10
------------------
```

Figure 4: Output of the second method of declaring nested functions in Go language

The function in *Figure 3* is defined inside the main function so it is a nested function. It is assigned to a variable that holds the function and then the function can be called with the parameters using that variable. *Figure 4* is the output of the print statement.

Design Issue 2: Scope of Local Variables

Local variables are the variables that are defined inside a function body in the program and it cannot be accessed outside of that function body [3]. Those variables can also be accessed by the other blocks inside the function body. Nested functions are also included in those blocks namely the scope of the local variables include the nested functions inside the function that the variable is defined.

```
var locVar int = 1;
fmt.Println("The value of the local variable inside the main function is ", locVar)

for i := 0; i < 3; i++ {
  fmt.Println("The value of the local variable inside the for block is ", locVar)
}

func(temp int) {
  fmt.Println("The value of the local variable inside a nested funtion is ", locVar)
}(0)
```

Figure 5: Scope of local variables in Go language

Figure 5 demonstrates that the local variable locVar inside the main function can be accessed by the codes inside the main function as well as the other code blocks inside the main function and also the nested functions that has the main function as the outer function. All print statements print 1 in *Figure 5*.

Design Issue 3: Parameter Passing Methods

There are two different ways of passing parameters to functions in Go language. The first way pass by value (it can also be called as call by value). The second way of passing parameters in a function is pass by reference, namely call by reference. The parameters that are showed in the function definition are called the formal parameters and those are the ones that are received by the functions. The original parameters that are passed to the functions are called the actual parameters [2].

In pass by value, the values of the actual parameters are copied to the values of the formal parameters so both the formal and actual parameters have the same values as it should be. However they are stored at separate memory locations therefore the change in one of the variables cannot affect the other.

```go
var number int = 2;

// Pass by value example
fmt.Println("The value of the parameter is ", number, " in the main function before the pass by value")
showPassByValue(number) // Pass by value does not affect the parameter
fmt.Println("The value of the parameter is ", number, " in the main function after the pass by value")
fmt.Println("--------------------")
```

Figure 6: Parameter passing by pass by value in Go language

```
The value of the parameter is  2  in the main function before the pass by value
The value of the parameter is  2  before the change in the function
The value of the parameter is  120  after the change in the function
The value of the parameter is  2  in the main function after the pass by value
```

Figure 7: Output of parameter passing by pass by value in Go language

Figure 6 represents parameter passing by pass by value and it can be seen in *Figure 7* that the value of the variable “number” after the function call is not changed.

In pass by reference, the logic of pointers are used as it is done in the C language. When defining the function, we are taking the addresses of the parameters and then dereference those parameters inside the function. So only the memory location of the actual parameters are used and the changes made in the parameters are reflected to all the others that are placed in that location and pointed by the pointers.

```go
// Pass by reference example
fmt.Println("The value of the parameter is ", number, " in the main function before the pass by reference")
showPassByReference(&number) // Pass by reference takes the address of the variable
fmt.Println("The value of the parameter is ", number, " in the main function after the pass by reference")
```

Figure 8: Parameter passing by pass by reference in Go language

```
The value of the parameter is  2  in the main function before the pass by reference
The value of the parameter is  2  before the change in the function
The value of the parameter is  120  after the change in the function
The value of the parameter is  120  in the main function after the pass by reference
```

Figure 9: Output of parameter passing by pass by reference in Go language

Figure 8 represents parameter passing by pass by reference and it can be seen in *Figure 9* that the changes made in the function is reflected to the variable “number”.

Design Issue 4: Keyword and Default Parameters

As a design choice, go does not support optional parameters and default parameter values [5]. However, the concept of variadic arguments can be used to mimic those functionalities. Variadic arguments allows the function to take different numbers of parameters.

```
func showVariadicArgs(vals ...int) {
  for i := 0; i < len(vals); i++ {
    fmt.Println("The value of the ", (i + 1), "th parameter is ", vals[i])
  }
}
```

Figure 10: Variadic arguments in function definition in Go language

```
// Question 4: Keyword and default parameters
showVariadicArgs()
showVariadicArgs(5)
fmt.Println("--------------------")
showVariadicArgs(12, 23, 14, 54, 6)
fmt.Println("--------------------")
showVariadicArgs(1, 2, 3)
fmt.Println("--------------------")
```

Figure 11: Variadic arguments in function calls in Go language

Figure 10 has the variadic argument declaration in the parameter list of the function. vals …int means that there will be a list of parameters in which the size of the list is not known. So the program is ready to take as much parameters as possible. *Figure 11* shows how the function can take any number of parameters. Variadic parameters are useful since Go language does not support default parameters of optional parameters by design.

Design Issue 5: Closures

Go language supports anonymous functions and anonymous functions can be used to create closures [6]. So a function can have access to a variable that is not in its scope even if the variable is not passed through parameter. This function is usually an inner function that has the access to the variables in the outer function.

```
// Question 5: Closures
var closureVar int = 0

incrementFunct := func() int {
  closureVar = closureVar + 1
  return closureVar
}

 fmt.Println(incrementFunct())
 fmt.Println(incrementFunct())
 fmt.Println(incrementFunct())
 fmt.Println(incrementFunct())
 fmt.Println(incrementFunct())
```

Figure 12: Closures in Go language

Figure 13: Output of Closures in Go language

Figure 12 represents that closures are supported by Go language and the variable closureVar can be accessed by the incrementFunct even if it is out of its scope. *Figure 13* shows the output of the program in *Figure 12*.

Discussion 1:

Readability and writability are 2 of the important evaluation criteria for programming languages. Therefore in order to evaluate the subprogram syntax of the Go language, it is important to consider Readability and writability.
First of all It should be said that the syntax of Go is quite different than the syntax of C based languages in general. For example semicolon (;) is not used and the assignment operator ":=" is used. However in terms of function syntaxes it is not that different. "funct" keyword is meaningful and except that the parameter name and type is reversed in the function arguments, the definition is similar to C based languages. It is also readable since any programmer that has experience in any other programming language before can easily understand those function syntaxes.
In terms of writability, Go is a really nice language, since it has many features that can help the programmers while writing the code. Nested functions are one of the good features that helps the programmer. Also variables in the outer functions are accessible in the inner functions. Two different parameter passing methods also eases the work of the programmer. Similarly, variadic arguments are so flexible and time saving when programming in Go language. The support for closures is also a good factor that increases writability of the function syntax in Go.
Therefore, in general Go is a well designed modern language in terms of function syntax. It is both easy to understand if a programmer is used to use another programing language and it is also easy to write since it has many features that eases the job of the programmer.

Discussion 2:

This homework was a considerably easier one compared to the previous ones since it has only one language to evaluate. As the first step I started by studying the functions chapter in the textbook. For some topics, textbook had a lot of details therefore sometimes I only used the lecture slides since those are shorter and easier to understand.
After I had a good understanding on the relevant topics, I started to learn the basic syntaxes of the Go language in the official website of the Go. It was a little bit hard for me to learn that language since its syntax is quite different than the C based languages. After I learned the basic syntax I started to investigate the design issues related to function. I tried to use the official website of the Go language but its documentation was a bit confusing for me so I used some other resources that I found online and I put all the resources I used to the references section.
For this homework it is said that we are allowed to use online compilers and I found a good online Go language compiler in the official website of Go which has the URL https://go.dev/play/. For each design issue I made a research using my resources and after I had an enough knowledge on the design issue I wrote the relevant information to the relevant section on the report and then I tested all the information that I learned on the online Go compiler. After I became sure that the code is working and the code segments that I have written are enough to demonstrate the design issues, then I took the screenshots of the code segments and I added those to my report.
One of the hard points in Go language in this project was that it has a quite different syntax than the C based languages. For example there is no parentheses in the loops. I spent some time to get familiar with those details before I evaluated the function syntaxes. Also Go language was a bit new for me since we worked on different

programming languages in the first two homework and Go is a complete new programming language for me. But one of the good parts of this homework was that we are allowed to use online compilers and the official online Go playground is a quite good one in terms of testing the codes but one problem about it was it does not auto complete the variables and auto close the parentheses or the braces. Thats why I used my own text editor to write my Go codes and then I copied the codes from my text editor and pasted them to the online compiler. However it was so much easier to test all of the codes in the Dijkstra server since if there happens an error in the Dijkstra server, it is so hard to make changes in the code and re-upload the new code into the server.

References

[1] Sharbeargle. (n.d.). *Nested functions*. Nested Functions · GoLang Notes. Retrieved December 23, 2021, from https://sharbeargle.gitbooks.io/golang-notes/content/nested-functions.html

[2] *Function arguments in Golang*. GeeksforGeeks. (2019, August 13). Retrieved December 23, 2021, from https://www.geeksforgeeks.org/function-arguments-in-golang/

[3] *Scopes in go (examples)*. Scopes in Go (Examples) | Learn Go Programming. (n.d.). Retrieved December 23, 2021, from https://golangr.com/scope/

[4] Krunal. (2021, August 23). *Golang variables scope: What is the scope of variables in go*. AppDividend. Retrieved December 23, 2021, from https://appdividend.com/2020/01/29/scope-of-variables-in-golang-go-variables-scope/

[5] Nilsson, S. (n.d.). *Optional parameters, default parameter values and method overloading*. · YourBasic Go. Retrieved December 23, 2021, from https://yourbasic.org/golang/overload-overwrite-optional-parameter/

[6] *Closures in Golang*. GeeksforGeeks. (2020, March 23). Retrieved December 23, 2021, from https://www.geeksforgeeks.org/closures-in-golang/

[7] Go by example: Closures. (n.d.). Retrieved December 23, 2021, from https://gobyexample.com/closures

[8] *Build fast, reliable, and efficient software at scale*. Go. (n.d.). Retrieved December 23, 2021, from https://go.dev/

[9] *The go playground*. Go. (n.d.). Retrieved December 23, 2021, from https://go.dev/play/