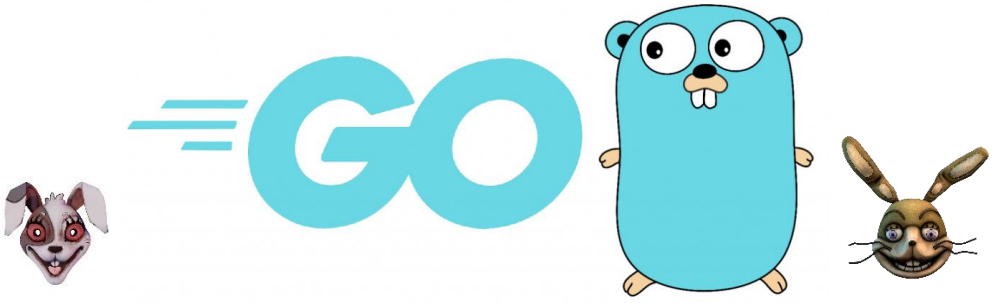
CS315

Programming Languages

HW3: Subprograms in Golang



N. Onur VURAL

21902330

SEC: 01

1- Nested subprogram definition

EXAMPLE->

```
func main() {
	fmt.Println("1) Nested subprogram definitions")
	// First subprogram defininiton
	func1 := func() {
		fmt.Println("HELLO...From func1")
		// Nested subprogram definition
		func2 := func() {
			fmt.Println("HELLO...From func2")
		}
		fmt.Println("func2 (nested inside func1) called: ")
		func2()
	}
	fmt.Println("func1 (nested inside main) called: ")
	func1()
}
```

OUTPUT->

```
1) Nested subprogram definitions
func1 (nested inside main) called:
HELLO...From func1
func2 (nested inside func1) called:
HELLO...From func2
```

The example demonstates that Golang allows to define nested subprograms which basically expresses functions that are defined inside other functions. In Golang, a nested function can only be defined if functions are assigned to local variables such as functionName := func() {}, where calling is done by functionName(). A subprogram may define another subprogram inside and then can call them afterwards which the example above demonstrates by funct1 defining funct2 and callling funct2. In addition to that funct1 is also a nested subprogram inside main as the example demonstrates. In the subprogram definition, it is extremely important to call the nested subprogram **after** its declaration, otherwise it will give an error. If they are declared as local variables, they must be used within as well [1], [2].

2- Scope of local variables

EXAMPLE->

```
func main() {
	fmt.Println("2) Scope of local variables")
	outer := func() { // Scope of myVar1 (72) and myVar2 (27) starts here
		var myVar1, myVar2 int
		myVar1 = 72
		myVar2 = 27
		fmt.Printf("myVar1 accessed from outer is: myVar1 = %d", myVar1)
		fmt.Println(" ")
		fmt.Printf("myVar2 accessed from outer is: myVar2 = %d", myVar2)
		fmt.Println(" ")
		inner := func() { // Scope of  myVar2 (44) and myVar3 (88) starts here
			var myVar2, myVar3 int
			myVar2 = 44
			myVar3 = 88
			fmt.Printf("myVar1 accessed from inner is: myVar1 = %d",     myVar1)
			fmt.Println(" ")
			fmt.Printf("myVar2 accessed from inner is: myVar2 = %d", myVar2)
			fmt.Println(" ")
			fmt.Printf("myVar3 accessed from inner is: myVar3 = %d", myVar3)
			fmt.Println(" ")
		} // Scope of  myVar2 (44) and myVar3 (88) ends here
		inner()
		//fmt.Printf("myVar3 accessed from outer is: myVar3 = %d", myVar3) //
ERROR: undefined: myVar3
```

```
        } // Scope of myVar1 (72) and myVar2 (27) ends here

        outer()

}
```

OUTPUT->

```
2) Scope of local variables
myVar1 accessed from outer is: myVar1 = 72
myVar2 accessed from outer is: myVar2 = 27
myVar1 accessed from inner is: myVar1 = 72
myVar2 accessed from inner is: myVar2 = 44
myVar3 accessed from inner is: myVar3 = 88
```

A local variable exists within the block that the declaration has been made in Golang. Golang does not include any special keyword to distinctiate global and local variables but the initialization place matters. If a variable is declared outside any function it is global, whereas a variable defined inside block/function corresponds to a local one. Local variables can be accessed within the block or inner levels if there is a nested subprogram for example. In the example all myVar1, myVar2 (both in inner and outer) and myVar3 are **local** variables accessible within the block of declaration. Therefore, myVar1 is visible within the outer as well as in inner since inner is is nested in outer. Inside inner, myVar2 prints 44 instead of 27 as myVar2 is also declared inside and therefore accessed prior in the block. When it comes to myVar3, it is only visible in inner, trying to access it outside the block will result in error (undefined: myVar3) [3], [4], [5], [6], [7], [8].

3- Parameter passing methods

EXAMPLE->

```
func main() {
	fmt.Println("3) Parameter passing methods")
	fmt.Println("-->Pass by value example: ")
	var x int
	x = 4
	fmt.Printf("x before accessPassByValue is : x = %d\n", x)
	accessPassByValue := func(x int) {
		x = x + 1
		fmt.Printf("x inside accessPassByValue is : x = %d\n", x)
	}
	// passing x by value
	accessPassByValue(x)
	fmt.Printf("x after accessPassByValue is : x = %d\n", x)
	fmt.Println(" ")

	fmt.Println("-->Pass by reference example: ")
	fmt.Printf("x before accessPassByReference is : x = %d\n", x)
	accessPassByReference := func(x *int) {
		*x = *x + 1
		fmt.Printf("x inside accessPassByReference is : x = %d\n", *x)
	}
	// passing x by reference
	accessPassByReference(&x)
```

```
fmt.Printf("x after accessPassByReferencee is : x = %d\n", x)
fmt.Println(" ")

fmt.Println("-->Swap trial: ")
var a, b int
a = 2
b = 7
fmt.Printf("a is: a = %d / b is: b = %d", a, b)
fmt.Println(" ")
swapPassByValue := func(a int, b int) {
        var temp int
        temp = a
        a = b
        b = temp
}
fmt.Println("After passing the parameters by value: ")
swapPassByValue(a, b)
fmt.Printf("a is: a = %d / b is: b = %d", a, b)
fmt.Println(" ")

swapPassByRef := func(a *int, b *int) {
        var temp int
        temp = *a
        *a = *b
        *b = temp
```

```
        }

        fmt.Println("After passing the parameters by reference: ")

        swapPassByRef(&a, &b)

        fmt.Printf("a is: a = %d / b is: b = %d", a, b)

        fmt.Println(" ")

}
```

OUTPUT->

```
3) Parameter passing methods
-->Pass by value example:
x before accessPassByValue is : x = 4
x inside accessPassByValue is : x = 5
x after accessPassByValue is : x = 4

-->Pass by reference example:
x before accessPassByReference is : x = 4
x inside accessPassByReference is : x = 5
x after accessPassByReferencee is : x = 5

-->Swap trial:
a is: a = 2 / b is: b = 7
After passing the parameters by value:
a is: a = 2 / b is: b = 7
After passing the parameters by reference:
a is: a = 7 / b is: b = 2
```

The default passing method is pass-by-value (in mode) in Golang and therefore only the values are used inside parameters of subprograms without modifying the actual variable. Although Golang does not support pass-by-reference (inout mode) in default, this can be simulated by using values so that the actual parameter can be modified instead of just copying its value. The example above demonstrates this by first placing x as an actual parameter as **value**. As expected, after the operations, the value x of it remains same as 4. By placing the **address** of x to the actual parameter, after the operations, the value x becomes modififed to 5. The following example also demonstrates how the simulation of pass-by-reference may be achived by pointers to modify parameters themselves [9], [10].

4- Keyword and default parameters

EXAMPLE->

```
func main() {
	fmt.Println("4) Keyword and default parameters")
	fmt.Println("-->Keyword param simulation: ")
	//plt.Plot(xs, ys, "r", &plt.PlotKwargs{Lw: 3, Ls: "--", Antialiased: true})
	//plt.Plot(xs, ys, "r", plt.Lw, 3, plt.Ls, "--", plt.Antialiased, true)

	car := func(brand string, price float64, model string) {
		fmt.Printf("brand is : %s / price is : %f/ model  is :  %s\n", brand, price, model)
	}
	type namedParamsOfCar struct {
		price float64
		model string
	}
	fmt.Println(" By using namedParamsOfCar struct as namedParamsOfCar{price: 148.000, model: windsor 1948 saloon} ")
	fmt.Println(" car(Chrysler, namedP.price, namedP.model) gives: ")
	var namedP = namedParamsOfCar{price: 148.000, model: "Windsor 1948 Salon"}
	car("Chrysler", namedP.price, namedP.model)
	fmt.Println(" ")
	fmt.Println("-->Default param simulation: ")
	defParamFunc := func(i int, args ...int) (int, int, int) {
		j := 1
		k := 2
		if len(args) == 2 {
```

```
                j = args[0]
                k = args[1]
        }
        if len(args) == 1 {
                j = args[0]
        }
        fmt.Printf("i is: i = %d / j is: j = %d / k is: k = %d\n", i, j, k)
        return i, j, k
    }
    fmt.Println("defParamFunc(1) gives: ")
    defParamFunc(1)
    fmt.Println("defParamFunc(3, 5) gives: ")
    defParamFunc(3, 5)
    fmt.Println("defParamFunc(3, 5, 7) gives: ")
    defParamFunc(3, 5, 7)
}
```

OUTPUT->

```
4) Keyword and default parameters
-->Keyword param simulation:
 By using namedParamsOfCar struct as namedParamsOfCar{price: 148.000,
model: windsor 1948 saloon}
 car(Chrysler, namedP.price, namedP.model) gives:
brand is : Chrysler / price is : 148.000000/ model  is :  Windsor 1948
Salon

-->Default param simulation:
defParamFunc(1) gives:
i is: i = 1 / j is: j = 1 / k is: k = 2
defParamFunc(3, 5) gives:
i is: i = 3 / j is: j = 5 / k is: k = 2
defParamFunc(3, 5, 7) gives:
i is: i = 3 / j is: j = 5 / k is: k = 7
```

As a result of design choices, Golang itself does not support, in none official way, parameters with keyword or having default parameters inside functions. However this may be achived by a few techniques. Example above shows how to mimic keyword parameters by first creating a struct and then using the attributes of this struct to pass as parameters. When it comes to default parameters, example above shows how to manually check provided arguments and then decide to assign prevalues accordingly. As the results demonstrate those solutions are just some ways to mimic the desired output however the language itself does not provide actual solutions to these issues and rather wants to avoid the usage of such parameters as a defense mechanism [11], [12], [12], [14].

5- Closures

EXAMPLE->

```
func main() {
	fmt.Println("5) Closures")
	caller := func(param string) func() {
		var arg string
		arg = param
		callee := func() {
			fmt.Printf("arg is: %s\n", arg)
		}
		return callee
	}
	// anonymous function is assigned
	myFun := caller("This message demonstrates the usage of closures!!!")
	myFun() // anonymous function is called and it can reach into arg variable although it is defined on outer function!!!
	myFun()
	myFun2 := caller("Hello world!!!")
	myFun2()
```

```
}
```

OUTPUT->

```
5) Closures
arg is: This message demonstrates the usage of closures!!!
arg is: This message demonstrates the usage of closures!!!
arg is: Hello world!!!
```

This example shows how to construct closures by using anonymous functions. Golang allows this feature and therefore allows such anonymous functions as myFun and myFun2 to reach out to variables that are actually declared outside. In the example, myFun and myFun2 succesfully print the desired messages even though the message actually is stored in variable arg that is outside of callee function. Therefore this technique makes it possible to access data in a rather flexible manner [15].

ANALYSIS AND DETAILS

1. Evaluation of Golang in terms of readability and writability of subprogram syntax.

When it comes to Golang, it appears that there is a strong tradeoff between readibility and writability in terms of subprogram syntax. First of all, the design choices of Golang suggest the subprograms are aimed to be easily readable and identifiable. However Golang does not allow to define nested subprograms without assigning them to local variables (In many languages there isn't such an issue) which harms the writablity principles in a great extent. When it comes to scope of local variables inside those subprograms Golang favours writability since a special keyword is not required to distinctiate global and local vars (like my, global) and the place of declaration is the only determining factor. But this can also make it harder to distinctiate global variables and local variables, therefore reducing readablitiy. Parameter passing techniques are very similar to C group languages where the default mode is pass-by-value although pass-by-reference can be simulated as well by using pointers. Golang therefore favors flexibility, writability in this point. Golang does not allow keyword and default parameters not to cause confusion and to keep the declarations as simple/ as possible, thus favoring readability. But this becomes a very crucial issue in terms of writabality since it requires too much effort if one needs to mimic keyword and default parameters (using structs and a combination of other complicated methods). Golang supports closure and therefore allows to flexibly reach out to data under anonymous functions. This is in parallel with writability but can harm readability as the variables reached may be hard to spot. Overall, it can be noticed that the design choices of Golang regarding subprograms try to keep a balance between readability and writability while trying to offer the basic features seen in other programming languages at the same time.

2.Write a separate section about your learning strategy in doing this homework assignment. A learning strategy is an individual's approach to complete a task. In this section, discuss, in detail, the material and tools you used, experiments you performed. Also talk about personal communication, if you had.

To complete this task, I used wide range of sources and research strategies. First of all, I started my research by viewing the official documentation of Golang in order to understand to core functionalities of this language such as how to declare variables, what are the available types, how to get user input, how to output to the console, how to make simple arithmathic operations, how the decision statements are used and how subprograms are created. After reading the official documentation, I wanted to testthose core functionalities by writing a simple test program. Accordingly I created a .go file by using Atom text editor and wrote a simple program that outputs the famous "Hello World" to the screen, then makes some basic operaitons and prints the results to the screen. At this point it was time to answer given questions. Therefore I started by looking question by question at a time. During this process, I used offical documentation, educational websites, articles, tutorials and blogs to obtain the required answers [1-15]. As a support to this, I used my notes that I took during Mr.Güvenir's lectures where they were a guide in my research process and accordingly helped me a lot in terms of verifying that I was on the right path in my research and finding correct and relevant information. With my findings, I completed given questions one by one as my test program slowly evolved from being a simple Hello World program to a program that demonstrates the required answers. In my program, I paid close attention to make sure that the answers are readable and explanatory enough for TA's. In order to check the syntactical correctness of my programs, I used the offical online compliler of Golang [16].

Sources

[1] https://sharbeargle.gitbooks.io/golang-notes/content/nested-functions.html

[2]https://medium.com/swlh/how-to-write-a-nested-function-in-go-3338c56526f3

[3]https://docs.thunderstone.com/site/vortexman/variable_scope_global_vs_local.html

[4]https://appdividend.com/2020/01/29/scope-of-variables-in-golang-go-variables-scope/

[5]https://www.geeksforgeeks.org/scope-of-variables-in-go/

[6]https://www.tutorialspoint.com/go/go_scope_rules.htm

[7]https://golangr.com/scope/

[8]https://golangbyexample.com/scope-of-variable-go/

[9]https://betterprogramming.pub/pass-by-value-and-reference-in-go-94423b6accf1

[10]https://www.geeksforgeeks.org/function-arguments-in-golang/

[11]https://forum.golangbridge.org/t/how-to-give-default-value-to-function-parameters/18099

[12]https://www.reddit.com/r/golang/comments/3gi0pf/different_ways_to_simulate_keyword_arguments/

[13]https://www.golangprograms.com/go-language/struct.html

[14]https://gobyexample.com/structs

[15]https://www.geeksforgeeks.org/closures-in-golang/

Online Compiler

[16] https://go.dev/play/