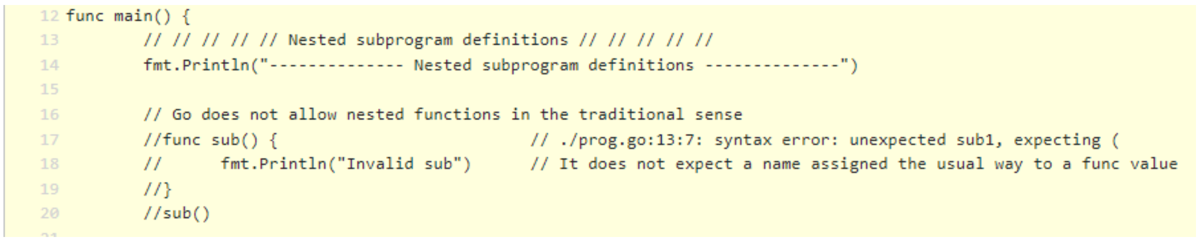
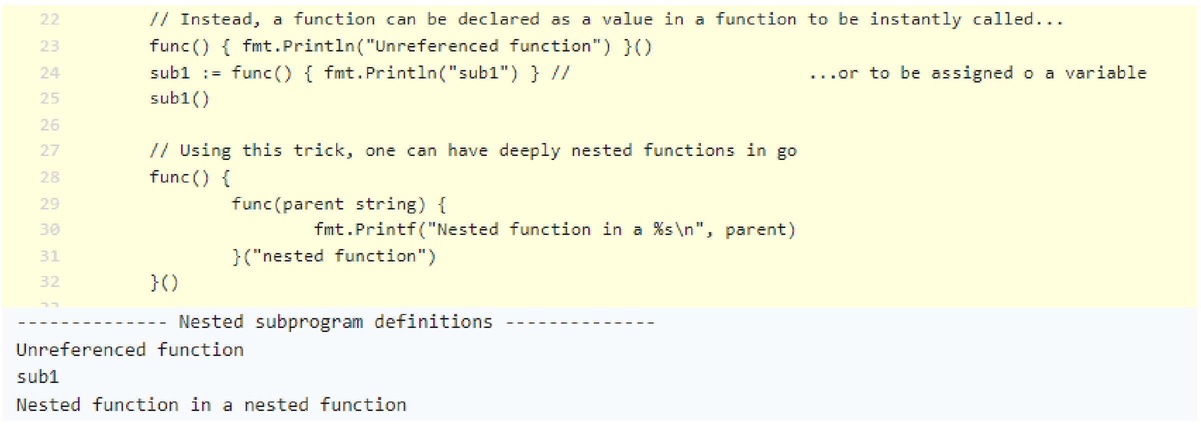
1. Nested subprogram definitions

Go does not allow nested functions in the traditional sense. When one tries to declare a function in a function using the usual syntax as the following, the error message “syntax error: unexpected sub1, expecting (” is got.



```go
func main() {
        // // // // // Nested subprogram definitions // // // // //
        fmt.Println("------------- Nested subprogram definitions -------------")

        // Go does not allow nested functions in the traditional sense
        //func sub() {                      // ./prog.go:13:7: syntax error: unexpected sub1, expecting (
        //      fmt.Println("Invalid sub")      // It does not expect a name assigned the usual way to a func value
        //}
        //sub()
```

The error is like this because, functions are regarded as values in go. So the token func is fine until one tries to assign a name to it. Instead however, they can be assigned into variables. This allows the programmers to have nested and deeply nested functions.



```go
        // Instead, a function can be declared as a value in a function to be instantly called...
        func() { fmt.Println("Unreferenced function") }()
        sub1 := func() { fmt.Println("sub1") } //                        ...or to be assigned o a variable
        sub1()

        // Using this trick, one can have deeply nested functions in go
        func() {
                func(parent string) {
                        fmt.Printf("Nested function in a %s\n", parent)
                }("nested function")
        }()
```

```
------------- Nested subprogram definitions -------------
Unreferenced function
sub1
Nested function in a nested function
```

2. Scope of local variables

In Go, variables are statically scoped by default. Furthermore, the language does not support dynamic scoping [1].

In Go, functions can access and manipulate variables from a broader scope, meaning they can be closures [2]. This will be demonstrated in the next code snippet. In functions, local variables can have the same name with a variable from parent-scope. When nested subprograms call these variables, they might access either one depending on whether the variable is re-declared prior to the function. If the variable was re-declared prior to the function declaration, they will access the narrower parent scope’s variable. Else, they will access the broader scope’s variable as in the following code:

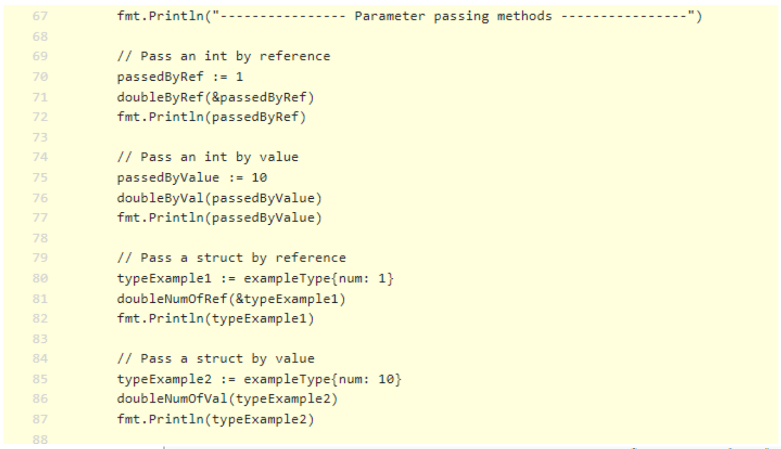
```go
var globalVariable = "global variable"
var scopeTestVar = "global"
var scopeTestVar1 = "global1"
var scopeTestVar2 = "global2"

```

```go
	fmt.Println("---------------- Scope of local variables ---------------")

	fmt.Println("In main")
	fmt.Println(scopeTestVar)
	scopeTestVar := "main" // Declaring a new variable with same name as the global

	subB := func(callerScopeName string) {
		// static scoped by default
		if callerScopeName == scopeTestVar {
			fmt.Println("Dynamic scoping")
		} else {
			fmt.Println("Static scoping")
		}
		fmt.Println(scopeTestVar) // static scoped by default
	}

	subA := func() {
		fmt.Println("In subA")
		fmt.Println(scopeTestVar)  // Is affected by the name reassignment
		fmt.Println(scopeTestVar1) // Is affected by the value change
		fmt.Println(scopeTestVar2) // Is not affected by the name reassignment

		scopeTestVar := "subA" // Test Dynamic scoping
		subB(scopeTestVar)
	}

	scopeTestVar1 := "main1" // Declaring a new variable with same name as the global
	scopeTestVar2 = "main2"  // Changing value of the global variable

	fmt.Println(scopeTestVar1)
	fmt.Println(scopeTestVar2)
	subA()
```

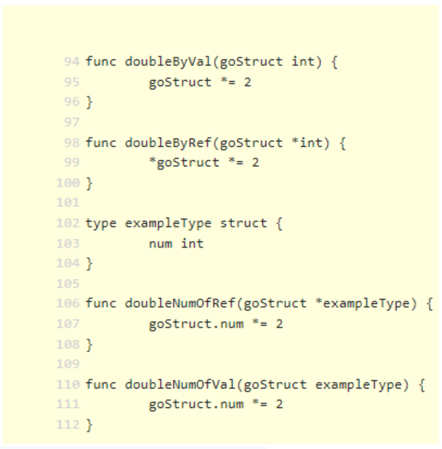
```
----------------- Scope of local variables ----------------
In main
global
main1
main2
In subA
main
global1
main2
Static scoping
main
```

3. Parameter passing methods

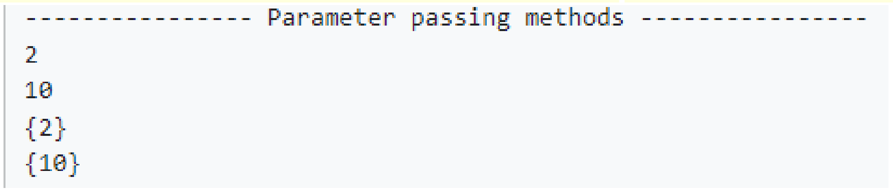
The language Go supports pass by value and pass by reference in function parameters. Parameters are passed by values by default in go. One can implement passing by reference by taking pointers as parameters while defining a function with “*”, and passing a reference to the variable and to the function call with “&”.



```go
	fmt.Println("--------------- Parameter passing methods ---------------")

	// Pass an int by reference
	passedByRef := 1
	doubleByRef(&passedByRef)
	fmt.Println(passedByRef)

	// Pass an int by value
	passedByValue := 10
	doubleByVal(passedByValue)
	fmt.Println(passedByValue)

	// Pass a struct by reference
	typeExample1 := exampleType{num: 1}
	doubleNumOfRef(&typeExample1)
	fmt.Println(typeExample1)

	// Pass a struct by value
	typeExample2 := exampleType{num: 10}
	doubleNumOfVal(typeExample2)
	fmt.Println(typeExample2)

```



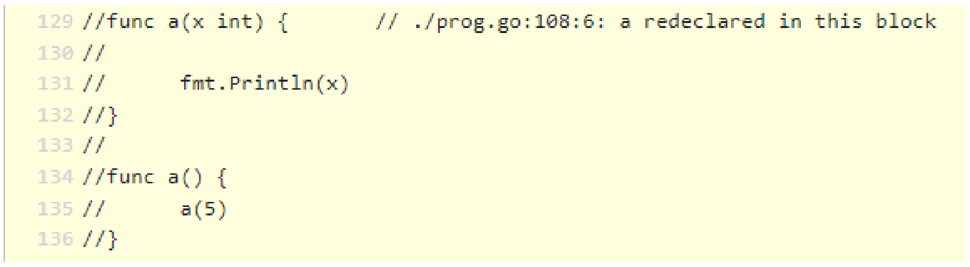
```go
func doubleByVal(goStruct int) {
	goStruct *= 2
}

func doubleByRef(goStruct *int) {
	*goStruct *= 2
}

type exampleType struct {
	num int
}

func doubleNumOfRef(goStruct *exampleType) {
	goStruct.num *= 2
}

func doubleNumOfVal(goStruct exampleType) {
	goStruct.num *= 2
}
```



```
---------------- Parameter passing methods ----------------
2
10
{2}
{10}
```

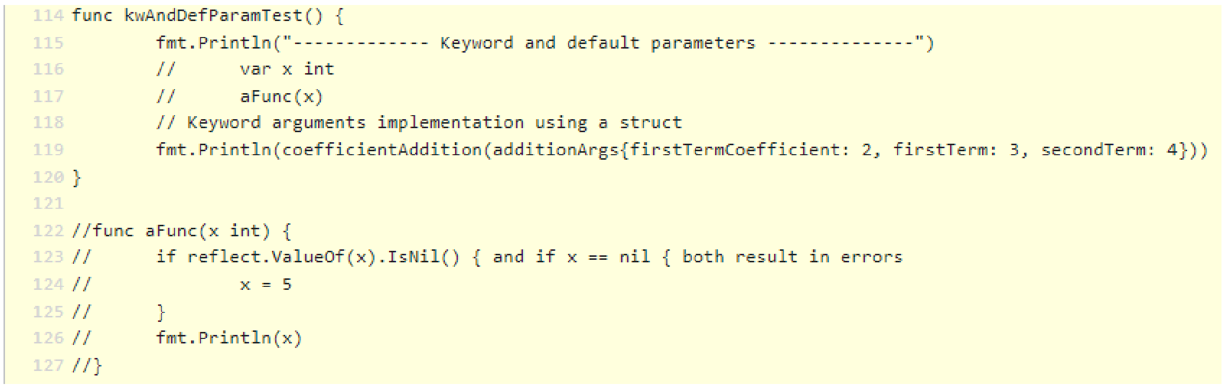
4. Keyword and default parameters

Go does not support default or optional parameters [3]. Because that the parameters are not a part of the method signature unlike in java, one cannot simulate optional parameters as the following:



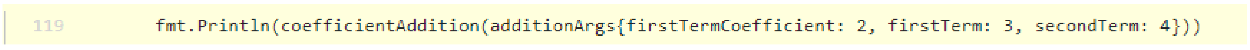
```go
//func a(x int) {       // ./prog.go:108:6: a redeclared in this block
//
//	fmt.Println(x)
//}
//
//func a() {
//	a(5)
//}
```

I also tried passing null variables to functions and null checking them in the function to simulate the behavior.

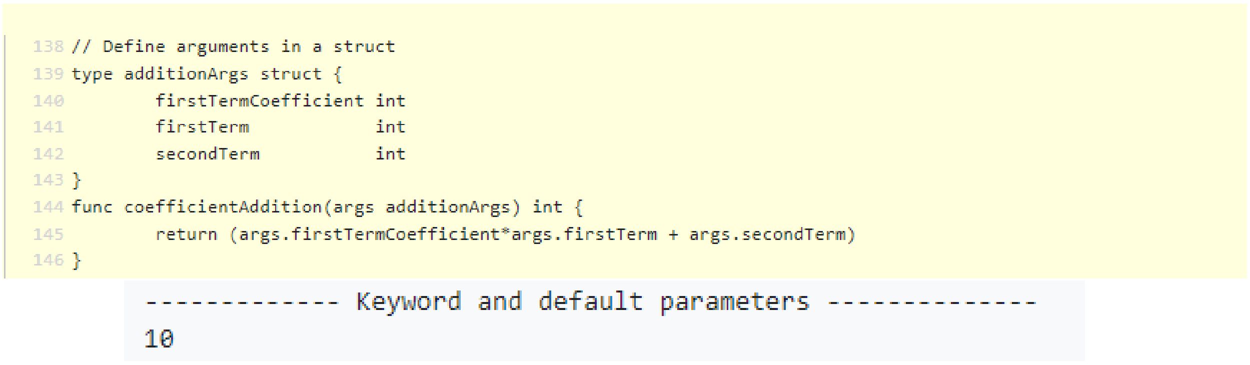


```go
func kwAndDefParamTest() {
	fmt.Println("------------- Keyword and default parameters --------------")
	//	var x int
	//	aFunc(x)
	// Keyword arguments implementation using a struct
	fmt.Println(coefficientAddition(additionArgs{firstTermCoefficient: 2, firstTerm: 3, secondTerm: 4}))
}

//func aFunc(x int) {
//	if reflect.ValueOf(x).IsNil() { and if x == nil { both result in errors
//		x = 5
//	}
//	fmt.Println(x)
//}
```

Go does not support keyword arguments either. However, this behavior can be simulated with use of structs and proper documentation. However, it still decreases readability and is able to cause side effects [4] [5].



```go
        fmt.Println(coefficientAddition(additionArgs{firstTermCoefficient: 2, firstTerm: 3, secondTerm: 4}))
```

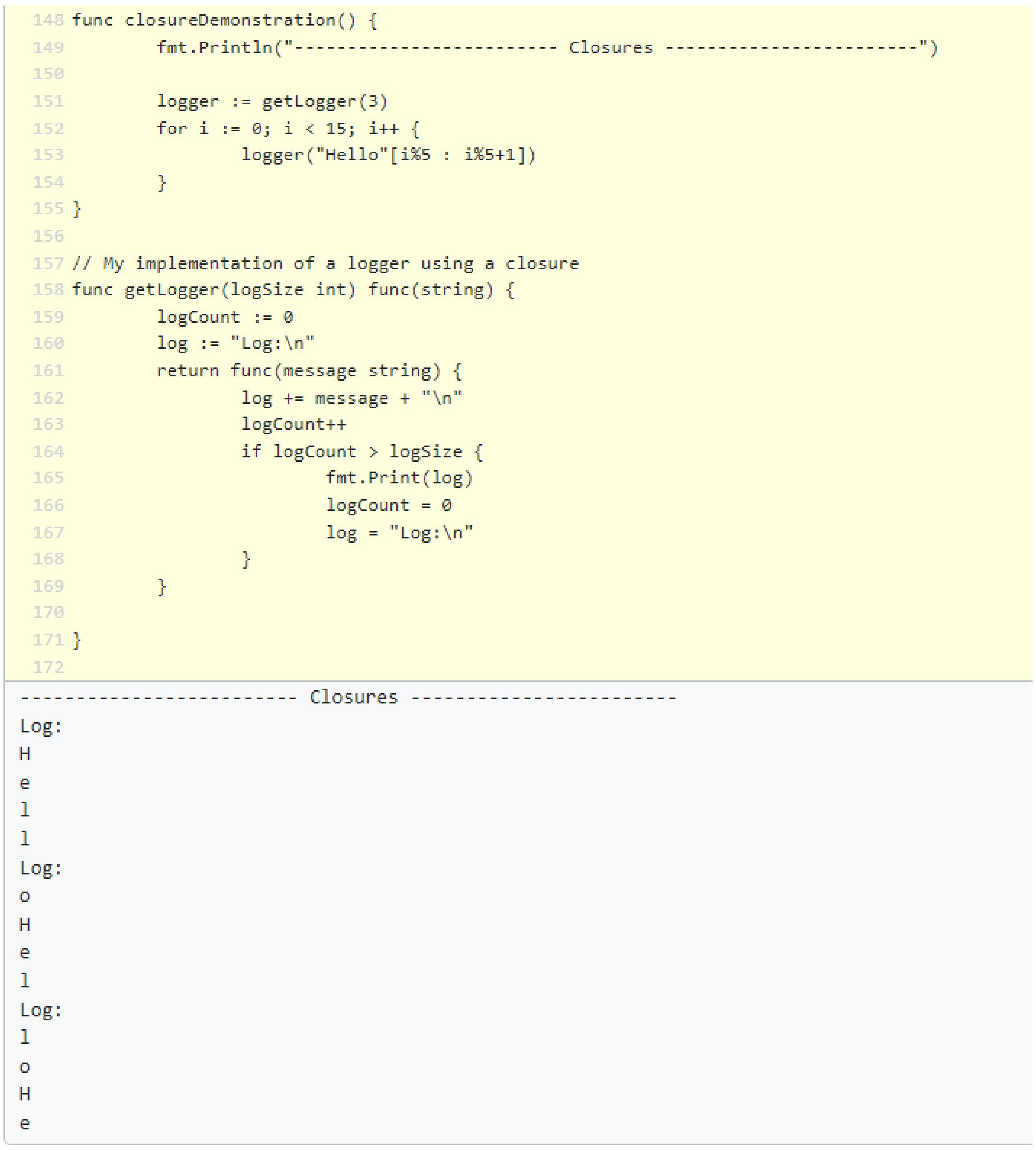


```go
// Define arguments in a struct
type additionArgs struct {
        firstTermCoefficient int
        firstTerm            int
        secondTerm           int
}
func coefficientAddition(args additionArgs) int {
        return (args.firstTermCoefficient*args.firstTerm + args.secondTerm)
}
```

```
------------- Keyword and default parameters --------------
10
```

5. Closures

In the “Nested subprogram definitions” section, we have briefly mentioned closures. Using nested functions the way previously discussed, one can have closures in their Go programs.



```go
func closureDemonstration() {
        fmt.Println("------------------------ Closures -----------------------")

        logger := getLogger(3)
        for i := 0; i < 15; i++ {
                logger("Hello"[i%5 : i%5+1])
        }
}

// My implementation of a logger using a closure
func getLogger(logSize int) func(string) {
        logCount := 0
        log := "Log:\n"
        return func(message string) {
                log += message + "\n"
                logCount++
                if logCount > logSize {
                        fmt.Print(log)
                        logCount = 0
                        log = "Log:\n"
                }
        }

}

```

```
------------------------ Closures -----------------------
Log:
H
e
l
l
Log:
o
H
e
l
Log:
l
o
H
e
```

- **Language Evaluation**

To implement nested subprograms in Go, a programmer has to choose slightly unusual ways. This decreases readability and writability very little. But once one gets the grasp of the syntax, it is very straightforward and there is no loss of readability and writability. Nested subprograms are useful and powerful tools. But once they are implemented, it can become hard to trace variables in them. For these reasons, nested subprograms not being directly supported in Go can increase readability but decrease writability.

Variables in Go subprograms are statically scoped. Subprograms also can access the variables in higher scopes unless the variable names are overridden. These expected behaviors make the language easily readable and writable.

In Go, parameters are passed by value by default. The programmer can explicitly state to take the reference as a parameter. This is the expected behavior and is great for readability and writability.

Go not supporting keyword parameters and default parameters makes the language less writable. To replicate the behavior of keyword arguments, a programmer has to spend effort. And the resulting code is less readable.

Go is pretty straightforward and acts as expected when it comes to nested subprograms. This makes closures simpler. The language is great in terms of writability and readability for closures.

- **Learning Strategy**

I started by learning the basic syntax of the language Go. For this, I did what I usually do while learning a new programming Language, framework or library. I briefly looked at tutorials and blog posts, then dived right into it. I used the playground they provide in their own website: https://go.dev/play/. The text editor in the playground did not have any syntax highlighting but it felt comfortable so I did not feel the urge to look for another.

I started experimenting with language in the directions the given design issues led me towards. The error messages I got while learning Go are pretty clear and explanatory. Hence, the googling part was easier. Some of the design issues I googled were about concepts, some were about specific what and how-to's. Those searches led me to forum threads, and go.dev documentations.

First and the design issues were about how to do a common thing in languages. So, I did not have to do much experiment-oriented coding but more demonstration-oriented coding. Instead, I had to do some research. I scanned through results, read some of them and noted them.

For second design issue, first I did some googling. The research I had to do was about the existence of some concepts in the language. After learning what Go supports, I did quite a bit of experiments to grasp the behavior of the language and took notes as comments.

Fourth design issue was similar to the second in the sense that first thing I had to do was to get the information about if some concepts exist in the language. After brief searching, I saw it did not, and tried to find ways to simulate it. I both experimented and surfed forums to find ways to catch those behavior. I then Demonstrated my findings in the code and took notes.

I had already found the information about the fifth design issue, closures while researching for the second design issue. So I only had to write a clear closure for demonstration.

While doing these researches, I noted down some of the URLs for future use. While writing the report, I picked the ones that I felt that I had better to display as reference, and referred to them.

References:

1. dave.cheney.net/2019/12/08/dynamically-scoped-variables-in-go
2. go.dev/tour/moretypes/25
3. yourbasic.org/golang/overload-overwrite-optional-parameter/
4. stackoverflow.com/questions/23447217/go-named-arguments
5. www.reddit.com/r/golang/comments/3gi0pf/different_ways_to_simulate_keyword_arguments/