CS315 HW3 Report

Efe Beydoğan

21901548

Section 1

1) Nested subprogram definitions

The Go language allows nested subprogram definitions to be written. Functions that are nested can be "instantiated, assigned to variables, and have access to its parent function's variables" (https://sharbeargle.gitbooks.io/golang-notes/content/nested-functions.html). A nested function inside another function must be assigned to a local variable, after which it can be called and can access and alter the static parents' variables. Nested subprograms are useful when a function is suppposed to be only called inside a specific function, and nowhere else. **However, the nesting of functions require them to be assigned to a local variable, as mentioned earlier, and these functions do not support recursion, for example. So, nesting cannot be done with regular functions like this:**

```
func main() {
  func inc(x int) int { return x+1; }
}
```

The above code snippet is taken from this StackOverflow discussion: https://stackoverflow.com/questions/21961615/what-are-the-problems-that-are-mitigated-by-not-allowing-nested-function-declara. As stated here, lambdas are supported by Go, although nested functions aren't allowed unless the function is assigned to a variable.

Trying to define a recursive nested function inside main results in an error:

```
1.  func main() {
2.
3.      recursive := func(num int) int {
4.          if num < 0 {
5.              return 0
6.          }
7.          fmt.Println(num)
8.          num--
9.          return recursive(num)
10.     }
11.
12.     recursive(5)
13. }
```

Recursion can be done with nested functions like this:

```
1.  func main() {
2.      var recursive func(num int) int
3.
4.      recursive = func(num int) int {
5.          if num < 0 {
```

```
            return 0
        }
        fmt.Println(num)
        num--
        return recursive(num)
    }

    recursive(5)
}
```

Here we can see that first declaring a variable and then defining the function allows for recursion.

The above recursive code outputs:

```
5
4
3
2
1
0
```

Code sample for nested functions:

```
fmt.Println("NESTED SUBPROGRAM DEFINITIONS")

fmt.Println("An outer function that contains an inner function, which in turn contains another function, is defined")

out := func( message string ) { // out is nested inside main
  fmt.Println( "The message in the outer function, before inner function is called: " + message)

  inner := func( innerMessage string) { // inner is deeply nested inside out
    deepNested := func( deepest string) { // deepNested is deeply nested inside inner
      fmt.Println( "inside the deepest function, message is: " + deepest)
    }
    fmt.Println( "The message in the inner function is: " + innerMessage)
    fmt.Println( "Even a more deeply nested function is called")
    deepNested( "inner called the function inside itself") // "inner" calls the "deepNested" function that is nested inside of itself
    fmt.Println( "inner function changes the message in the outer function")
    message = "inner function has altered the message in the outer function" // the inner function alters the message variable, which was passed to the outer function as a parameter
  }
  fmt.Println( "inner function is called inside the outer function")
  inner( "inner is called from outer") // inner is called inside outer
  fmt.Println( "message in outer after being changed by inner: " + message) // message is altered by the inner function
}

fmt.Println( "outer function is called")
out( "outer is called from main") // outer function is called in main
// inner( "inner is called from main") -> an error, cannot be called
```

The above program produces the following output:

```
NESTED SUBPROGRAM DEFINITIONS
An outer function that contains an inner function, which in turn
contains another function, is defined
outer function is called
The message in the outer function, before inner function is called:
outer is called from main
inner function is called inside the outer function
The message in the inner function is: inner is called from outer
Even a more deeply nested function is called
inside the deepest function, message is: inner called the function
inside itself
inner function changes the message in the outer function
message in outer after being changed by inner: inner function has
altered the message in the outer function
```

As can be seen from the above code and its output, nested subprograms are allowed in the Go language. The code displays that nested subprograms have access to their static parents' variables and can also change them. The nested functions cannot be called from an outer scope from where they are declared, meaning for example the main function cannot call the "inner" function declared inside the "out" function or the "out" function cannot call the "deepNested" function that is defined inside "inner". In the code sample, the "inner" function alters the value of the "message" parameter that was passed to the "out" function from main, and the "out" function calls the inner function, after which outputting the altered value. The inner function also calls another function nested inside of it, before returning control to "out", and the example illustrates the usage of nested subprograms in Go. Subprograms can be deeply nested inside of each other and can be used to do operations on their parents' variables, before returning control.

2) Scope of local variables

In Go, local variables are variables which are declared inside functions or blocks**. Local variables can only be accessed by statements that are inside the specific function or block of code where they were first declared, hence Go uses static scoping for all identifiers and the local variables are block scoped** (https://appdividend.com/2020/01/29/scope-of-variables-in-golang-go-variables-scope/). Subprograms that are nested inside a block can access the variables declared in their enclosing block, or static parents. A compile-time error will be thrown if a variable with the same name is declared twice. Moreover, these variables' lifetimes are over once the function's or block's execution is finished (https://www.geeksforgeeks.org/scope-of-variables-in-go/).

```
1.  fmt.Println("SCOPE OF LOCAL VARIABLES")
2.
3.  fmt.Println("two functions are defined, one is nested inside the other")
4.
5.  var y = 20
6.  fmt.Printf("a variable y in main is declared, y = %d \n", y)
7.  func1 := func() { // func1 is nested inside main
8.    var x = 10; // x is a local variable, only visible inside the block of func1
```

```
9.    fmt.Printf("a variable x is defined inside the outer function, x = %d \n", x)
10.   func2 := func() { // func2 is nested inside func1
11.     fmt.Printf("inside the inner function, variable x = %d \n", x)
12.     x += 10 // inner function can alter the value of x
13.     fmt.Println("inner function changes the value of x to x + 10")
14.
15.     fmt.Printf("inside the inner function, variable y = %d \n", y)
16.     y += 10 // inner function can alter the value of y which was declared in main
17.     fmt.Println("inner function changes the value of y to y + 10")
18.   }
19.   fmt.Println("inner function is called inside outer function")
20.   func2()
21.   fmt.Printf("after the execution of inner function is over, variable x = %d \n", x)
22.   // x's value has been changed by the inner function which has access to it
23. }
24.
25. func1()
26. fmt.Println("outer function's execution has finished, and the main function doesn't
    have access to the variable x that was defined locally inside the outer function")
27. fmt.Printf("new value of variable y, which was altered in the deeply nested func2(),
    y = %d \n", y)
28. // fmt.Println( x) -> gives an error, x is undefined here
```

The above program produces the following output:

```
SCOPE OF LOCAL VARIABLES
two functions are defined, one is nested inside the other
a variable y in main is declared, y = 20
a variable x is defined inside the outer function, x = 10
inner function is called inside outer function
inside the inner function, variable x = 10
inner function changes the value of x to x + 10
inside the inner function, variable y = 20
inner function changes the value of y to y + 10
after the execution of inner function is over, variable x = 20
outer function's execution has finished, and the main function doesn't
have access to the variable x that was defined locally inside the outer
function
new value of variable y, which was altered in the deeply nested
func2(), y = 30
```

Above code segment and its output denote that the scope of local variables are limited to their enclosing blocks, and these variables cannot be accessed outside. It can be observed from the example code that if two functions are nested, then the inner function can access and change the value of the local variables of the outer function, however the main function, which encloses the outer function, cannot access the variables that were declared inside the outer function's block. Also observed is that the deeply nested func2() inside func1() is able to reach a variable declared in main and change its value as well. The local variables are statically scoped, so only the variables of the static parents can be reached.

3) Parameter passing methods

Golang features support for two ways of passing parameters to functions: **pass by value and pass by reference**. The default is pass by value. When passing by value, the values of the actual parameters (parameters passed to the function) are copied to the formal parameters (parameters received by the function) and the parameters and the actual values are stored in different places in the memory. As such, changes made to the parameters inside the function aren't reflected in the actual parameters.

Pass by reference on the other hand, allows to change variables inside a function, when they are passed as arguments since both the actual and formal parameters point to the same locations. "The dereference operator * is used to access the value at an address. The address operator & is used to get the address of a variable of any data type" (https://www.geeksforgeeks.org/function-arguments-in-golang/).

```
1.  package main
2.
3.  import "fmt"
4.
5.  func change(param1 int, param2 int) {
6.      fmt.Println("inside change function that uses pass by value")
7.      var temp = param1
8.      param1 = param2
9.      param2 = temp
10. }
11.
12. func changePointer(param1 *int, param2 *int) {
13.     fmt.Println("inside change function that uses pass by reference")
14.     var temp = *param1
15.     *param1 = *param2
16.     *param2 = temp
17. }
18. func main() {
19.     fmt.Println("PARAMETER PASSING METHODS")
20.
21.     fmt.Println("two methods are defined to swap integers, one uses pass by value
    while the other uses pass by reference")
22.
23.     var a = 5
24.     var b = 31
25.     fmt.Printf("before calling the swap method that uses pass by value: a = %d, b =
    %d \n", a, b)
26.     change(a, b)
27.     fmt.Printf("after calling the swap method that uses pass by value: a = %d, b =
    %d \n", a, b)
28.     fmt.Println("as can be seen, the values are not swapped")
29.
30.     fmt.Printf("before calling the swap method that uses pass by reference: a = %d,
    b = %d \n", a, b)
31.     changePointer(&a, &b) // receiving the addresses of the parameters
32.     fmt.Printf("after calling the swap method that uses pass by reference: a = %d, b
    = %d \n", a, b)
33.     fmt.Println("the values are swapped as a result of using pass by reference
    (pointers)")
34. }
```

The above program produces the following output:

```
PARAMETER PASSING METHODS
two methods are defined to swap integers, one uses pass by value while the
other uses pass by reference
before calling the swap method that uses pass by value: a = 5, b = 31
inside change function that uses pass by value
after calling the swap method that uses pass by value: a = 5, b = 31
as can be seen, the values are not swapped
before calling the swap method that uses pass by reference: a = 5, b = 31
inside change function that uses pass by reference
after calling the swap method that uses pass by reference: a = 31, b = 5
the values are swapped as a result of using pass by reference (pointers)
```

It can be observed from this example program that pass by value cannot be used to alter the original value of a parameter while pass by reference could be used to change the original value of the parameter that is passed to the subprogram. Also, when a variable is passed by pointer, a new copy of the pointer pointing to the same address is created. Accordingly, passing by value could be more efficient than passing by reference in the Go language, as opposed to other languages that feature pointers, such as C or C++. It is explained that this happens because "Go uses escape analysis to determine if variable can be safely allocated on function's stack frame, which could be much cheaper then allocating variable on the heap. Passing by value simplifies escape analysis in Go and gives variable a better chance to be allocated on the stack" (https://goinbigdata.com/golang-pass-by-pointer-vs-pass-by-value/).

4) Keyword and default parameters

Default parameters in functions are used to assign default values to parameters of the function, so even if the user calling the functions doesn't provide values for those parameters, they can be used with the default values. **However, Go does not support default parameters** (https://yourbasic.org/golang/overload-overwrite-optional-parameter/). The Go language **does support variadic functions** (https://yourbasic.org/golang/variadic-function/), variadic functions are methods that take a variable number of arguments. From https://yourbasic.org/golang/variadic-function/:

"If the **last parameter** of a function has type ...T it can be called with **any number** of trailing arguments of type T."

Variadic function example:

```
1.  package main
2.
3.  import "fmt"
4.
5.  func variadic(numbers ...int) int { // this function takes an undetermined number of
    integers and sums them
6.      fmt.Println("inside the variadic function that takes a variable number of
    integers and returns their sum")
7.      sum := 0
```

```
8.
9.      for _, i := range numbers {
10.         sum = sum + i
11.     }
12.     return sum
13. }
14. func main() {
15.     fmt.Println("a variadic function is defined")
16.
17.     fmt.Printf("variadic function is called with no arguments: %d", variadic())
18.     fmt.Printf("variadic function is called with the arguments 31, 32, 33: %d",
    variadic(31, 32, 33))
19. }
```

Output of the above code:

```
a variadic function is defined
inside the variadic function that takes a variable number of integers and
returns their sum
variadic function is called with no arguments: 0
inside the variadic function that takes a variable number of integers and
returns their sum
variadic function is called with the arguments 31, 32, 33: 96
```

It can be observed from this example that variadic functions can take a variable number of arguments, but default parameters are not supported in Go by design.

On the other hand, keyword parameters also aren't supported in Go, as discussed in this StackOverflow post: https://stackoverflow.com/questions/23447217/go-named-arguments. Based on the discussion from there, keyword parameters can be remotely simulated with the use of structs but they aren't supported directly as in Python, for instance.

5) Closures

The Go language has support for creating anonymous functions, which in turn can be used to make closures. "In Golang, a closure is a function that references variables outside of its scope" (https://betterprogramming.pub/closures-made-simple-with-golang-69db3017cd7b). A closure can be used to access variables within its scope, even if that scope has been destroyed.

```
1.  package main
2.
3.  import (
4.      "fmt"
5.      "strconv"
6.  )
7.
8.  func closure(message string) func() string { // this function returns a closure
    which returns a string
9.      counter := 0
10.     fmt.Println("Inside the function that returns a closure, counter = 0 at first")
11.
```

```
12.     return func() string {
13.         counter++ // increments the counter every time
14.         fmt.Printf("inside closure, counter = %d \n", counter)
15.         message = message + " " + strconv.Itoa(counter) + " " // closure appends
    counter to the message received by the original function every time it is called
16.         return message
17.     }
18. }
19. func main() {
20.     fmt.Println("a function which returns a closure returning a string is defined")
21.     fmt.Println("function returning the closure receives a string message initially,
    and then that message is altered by the closure every time it is called")
22.     fmt.Println("the closure increments and appends the counter variable to the
    initial message every time it is called")
23.     closureFunc := closure("closure is called and appends the counter to the
    message: ")
24.
25.     fmt.Println("closure is called 3 times")
26.     fmt.Println(closureFunc())
27.     fmt.Println(closureFunc())
28.     fmt.Println(closureFunc())
29. }
```

The above code produces the following output:

```
a function which returns a closure returning a string is defined
function returning the closure receives a string message initially, and
then that message is altered by the closure every time it is called
the closure increments and appends the counter variable to the initial
message every time it is called
Inside the function that returns a closure, counter = 0 at first
closure is called 3 times
inside closure, counter = 1
closure is called and appends the counter to the message:  1
inside closure, counter = 2
closure is called and appends the counter to the message:  1  2
inside closure, counter = 3
closure is called and appends the counter to the message:  1  2  3
```

As it can be seen from the output of the code sample, closures can be used to access variables even after their scopes were destroyed. The closure is able to increment the counter variable that was defined in the body of the function which returns the closure, and the closure can also append the counter variable to the same message that the parent function had received as a parameter. Potential uses of closures include isolating data, such as isolating the counter variable here so the closure has access to it but no one can accidentally change it. Additionally, closures can be used to imitate callback functions, like the ones in JavaScript, and produce more readable code (https://www.calhoun.io/5-useful-ways-to-use-closures-in-go/).

Evaluation of Golang

In general, I believe the subprogram syntax of the Go language can usually be considered readable and writable since the subprogram syntax of Go resembles the syntaxes of popular languages such as Java or C++. Go doesn't support nested functions in the literal sense, since functions cannot be

nested unless they are assigned to variables and these functions cannot be recursively written. This feature of Go decreases writability since nested recursive functions cannot be defined, however I believe readability is increased as a result since it could be hard to comprehend nested and recursive functions when looking at the code, if they were allowed. Scope of local variables is not different from other popular languages such as Java, as local variables are statically scoped and they can only be accessed inside the block where they were defined. This feature makes Go more readable and writable since the scoping is easily understandable and usable when writing Go code. Go supports only two parameter passing methods; pass by value and pass by reference. Since the number of parameter passing methods is 2, Go is more readable due to this, since a person reading the code only has to be aware of two different methods for passing parameters. Also, writability is also increased as a result of this since pass by value is the default parameter passing method for Go, and pass by reference is done with pointers, just as in C or C++. A person that is familiar with these fundamental languages won't have problems using the two different parameter passing methods offered by Go. Unlike Python, Go does not support keyword or default parameters. I believe this is a hit for writability since default parameters usually provide an easy way to call functions without having to provide every parameter. However, the lack of keyword parameters could be good for both writability and readability since neither the coder nor the reader of the program has to remember the names of the parameters of the function. Finally, Go supports closures and these closures can be used to isolate data or imitate callbacks in the Go language, thus they increase writability. Although, the presence of closures in a program decreases readability since closures may complicate the code and make it harder to understand.

Learning Strategy

To write Go code, I used the compilers at https://go.dev/play/ and https://onecompiler.com/go. For the first question, I did some research on the internet about nested functions in Go, and realized that nested functions aren't supported in Go as expected. To use nested functions, they must be assigned to a variable and to make them recursive even an additional step needs to be taken, as I found out. I used the information found on these links https://sharbeargle.gitbooks.io/golang-notes/content/nested-functions.html and https://stackoverflow.com/questions/21961615/what-are-the-problems-that-are-mitigated-by-not-allowing-nested-function-declara to answer this question. Following this, I tried to find information on the scope of local variables in Go, and learned that local variables are always statically scoped and variables can only be accessed inside the blocks where they were declared, and not anywhere else. These links helped me come up with this information: https://appdividend.com/2020/01/29/scope-of-variables-in-golang-go-variables-scope/ and https://www.geeksforgeeks.org/scope-of-variables-in-go/. When I looked at the internet for the answer of the 3rd question, I found these links https://goinbigdata.com/golang-pass-by-pointer-vs-pass-by-value/,https://www.geeksforgeeks.org/function-arguments-in-golang/ which have taught me that Go only supports two types of parameter passing methods, pass by value and pass by reference. Since I was familiar with these methods from previous languages I had learned such as Java or C++, it was easy to adapt and write code to display Go's parameter passing methods. For the fourth question, when I did research, I realized that neither keyword nor default parameters are supported in Go. This link yielded this result: https://yourbasic.org/golang/overload-overwrite-optional-parameter/. Also from this link, https://yourbasic.org/golang/variadic-function/ I learnt

about variadic functions in Go which are functions that take a variable of number of parameters of a specific type, so they kind of simulate optional parameters but there are no default or keyword parameters in Go. Finally, for the last question I searched for closures in Go and could find a lot of resources. The two links https://gobyexample.com/closures and https://www.geeksforgeeks.org/closures-in-golang/ showed me how to use closures in Go, while this link https://www.calhoun.io/5-useful-ways-to-use-closures-in-go/ presented ways where closures could be useful.

Resources

https://onecompiler.com/go

https://go.dev/play/

https://golangr.com/closure/

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

https://stackoverflow.com/questions/21961615/what-are-the-problems-that-are-mitigated-by-not-allowing-nested-function-declara

https://quasilyte.dev/blog/post/go-nested-functions-and-static-locals/

https://gobyexample.com/variables

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

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

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

https://goinbigdata.com/golang-pass-by-pointer-vs-pass-by-value/

https://yourbasic.org/golang/overload-overwrite-optional-parameter/

https://yourbasic.org/golang/variadic-function/

https://www.geeksforgeeks.org/closures-in-golang/

https://gobyexample.com/closures

https://www.calhoun.io/5-useful-ways-to-use-closures-in-go/

https://betterprogramming.pub/closures-made-simple-with-golang-69db3017cd7b