CS315 – Homework Assignment 3

Burak Ozturk - Section 3 - 21901841

Nested Subprogram Definitions

Code segment:

```
// 1 Nested subprogram definitions
// Source: https://sharbeargle.gitbooks.io/golang-notes/content/nested-functions.html

// 1.1 Defining a function globally first
func nestedFunctions() {
   // 1.2 Defifining a function under another function is as follows

   // 1.2.1 Direct use without binding to a variable
   func(str string) {
      fmt.Println(str)
   }("Single-use function")

   // 1.2.2 Binding to a variable to being able to use multiple times
   var ordinarySubFunc = func(str string) {
      fmt.Println(str)
   }

   ordinarySubFunc("use1")
   ordinarySubFunc("use2\n")
}
func main() {
   nestedFunctions()
}
Output:
Single-use function
use1
use2
```

Explanation:

Nested subprograms, functions as they called in Go, can be nested in one another. As seen in the example above, there are two ways to define a function. They are defined as shown in 1.1 in the global scope and accessed by fName variable but in another function's scope, they are defined as shown in 1.2. After defined as in 1.2, they can be used directly as in 1.2.1 or can be bound to a variable as in 1.2.2.

Ease of use:

Global function declaration is very similar to C-like languages which causes familiarity for C user base and on top of that, since returns are declared in the function signature, it becomes easier to see inputs and outputs of the function clearly and apply black box model.

On the other hand, nested function declaration is odd. It is no problem once the user gets familiar with it but I had to read documentation as someone who is familiar with C family, Python, Java and some JavaScript.

Writability-wise, Go is between C-likes and Python. It is more complex than Python since there are more regulation characters like parentheses, brackets, etc. and Python is comparable to English thanks to the keywords such as and, or, not, etc. But C-like languages are more complex because of the same reasons. For example, Go doesn't need semicolons, library usage is easier and type-checking is not as strict.

Learning strategy and sources:

My learning strategy was to learn basics about nested functions and expand upon that base knowledge.

Scope of Local Variables

Code segment:

```
// Global variable
var a = 3

// 2 Scope of local variables
// I experimented for this one, no source used for knowledge.
func func1() {

   // 2.1 Global variable
   fmt.Println("a in func1 before closure: ", a)

   // 2.2 Closure
   {
      fmt.Println("a in closure before local a: ", a)
      var a = 4
      fmt.Println("a in closure after local a = 4: ", a)
   }

   // 2.3 Local variable
   fmt.Println("a in func1 after closure a = 4, before local a: ", a)

   var a = 5

   fmt.Println("a in func1 after local a = 5, before func2: ", a)

   // 2.4 Inside function
   var func2 = func(i int) {
      fmt.Println("a in func2 before local a: ", a)
      var a = 6
      fmt.Println("a in func2 after local a = 6: ", a)

      fmt.Println("i in func2: ", i)
   }

   func2(a)

   fmt.Println("a in func1 before for loop: ", a)

   // 2.5 For loop closure
   fmt.Println("In for loop:")
   for a := 7; a < 10; a++ {
      fmt.Print(a, " ")
   }
   fmt.Println()

   fmt.Println("a in func1 after for loop (a=10): ", a)

}
func main() {
   // 2 Scope of local variables
   fmt.Println("a in main before func1: ", a)
   func1()
   fmt.Println("a in main after func1: ", a, "\n")
}
```

```
Output:
a in main before func1:  3
a in func1 before closure:  3
a in closure before local a:  3
a in closure after local a = 4:  4
a in func1 after closure a = 4, before local a:  3
a in func1 after local a = 5, before func2:  5
a in func2 before local a:  5
a in func2 after local a = 6:  6
i in func2:  5
a in func1 before for loop:  5
In for loop:
7 8 9
a in func1 after for loop (a=10):  5
a in main after func1:  3
```

Explanation:

Scopes in Go are very predictable. There are global scope and main scope. Globally declared functions have their own scopes and they can access global scope. Nested functions have their own scopes as well and can access their super-functions' scopes. Closures such as "{}" blocks, for loops create new scopes with access to upper scopes. Code segment and output show these situations.

Ease of use:

Predictability makes Go's scoping easy for other large languages' users. I didn't come across any edge cases that could bug a beginner Go programmer's experimental codes.

Learning strategy and sources:

As I said, scopes in Go are predictable and because of that, I didn't need to research about this topic. I did experiments and tried every potential scope creator that I could think of and they worked.

Parameter Passing Methods

Code segment:

```go
// 3 Parameter passing methods
// I experimented for this one, no source used for knowledge.

// 3.1 Trivial types
func paramFunc(str string, i int) {
   fmt.Println("str in paramfunc before changing: ", str)
   fmt.Println("i in paramfunc before changing: ", i)
   str = "paramFunc'ed"
   i = 12346789
   fmt.Println("str in paramfunc after changing: ", str)
   fmt.Println("i in paramfunc after changing: ", i)

}

// 3.2 Array type
func arrayFunc(arr []int) {
   fmt.Println("arr[0] in arrayFunc before changing: ", arr[0])
   arr[0] = 343434
   fmt.Println("arr[0] in arrayFunc after changing: ", arr[0])
}

// 3.3 Structs
type ArgStr struct {
   str string
}

func structFunc(str ArgStr) {
   fmt.Println("str in structFunc before changing: ", str)
   str.str = "changed"
   fmt.Println("str in structFunc after changing: ", str)
}
```

```
func main() {
    // 3 Parameter passing methods

    // 3.1 For trivial types
    var str string = "abc"
    var i int = 72

    fmt.Println("str in main before changing: ", str)
    fmt.Println("i in main before changing: ", i)

    paramFunc(str, i)

    fmt.Println("str in main after changing: ", str)
    fmt.Println("i in main after changing: ", i, "\n")

    // 3.2 Array type
    arr := []int{1, 2, 3}
    fmt.Println("arr[0] in main before changing: ", arr[0])
    arrayFunc(arr)
    fmt.Println("arr[0] in main after changing: ", arr[0], "\n")

    // 3.3 Struct type
    var struc ArgStr = ArgStr{str: "not changed"}
    fmt.Println("struc in main before changing: ", struc)
    structFunc(struc)
    fmt.Println("struc in main after changing: ", struc, "\n")
}
Output:
str in main before changing:  abc
i in main before changing:  72
str in paramfunc before changing:  abc
i in paramfunc before changing:  72
str in paramfunc after changing:  paramFunc'ed
i in paramfunc after changing:  12346789
str in main after changing:  abc
i in main after changing:  72

arr[0] in main before changing:  1
arr[0] in arrayFunc before changing:  1
arr[0] in arrayFunc after changing:  343434
arr[0] in main after changing:  343434

struc in main before changing:  {not changed}
str in structFunc before changing:  {not changed}
str in structFunc after changing:  {changed}
struc in main after changing:  {not changed}
```

Explanation:

Go doesn't give the option to choose between pass-by-reference and pass-by-value like C does. By experimenting, I found out for trivial types like int and string and for custom structs (which are closest thing to a class in Go) pass-by-value is used but for arrays pass-by-pointer (or reference but unlikely) is used.

I didn't include in the code segment but Go has pointers and that means they can be used to imitate pass-by-reference just like C.

Ease of use:

In the first glance, it looks like no option to choose between parameter passing methods reduces ease but I think the opposite. In C, pass-by-reference most commonly used as an output method since C only supports one return value but Go supports multiple return values. Therefore, most of the use cases for pass-by-reference in Go is already covered. Plus, if someone really needs pass-by-reference, pointers are always available.

Learning strategy and sources:

I didn't research for this topic either. I already knew function signature for Go and extended on top of that by experimenting.

Keyword and Default Parameters

Code segment:

```
// 4.1 Keyword parameters (named arguments)
// Source: https://groups.google.com/g/golang-nuts/c/oWeeqCJfRzk?pli=1

type fArgs struct{ a, b int }

func structArgsFunc(args fArgs) (m int) {
   m = args.a - args.b
   return
}

// 4.2 Default parameters (Closest to that we can get)
// Source: https://stackoverflow.com/a/19813113
func defParamFunc(a ...int) {
   if len(a) > 0 {
      fmt.Println("a in defParamFunc: ", a[0])
   } else {
      fmt.Println("a in defParamFunc: ", 45)
   }

}

func main() {
   // 4.1 Named arguments
   var args1 = fArgs{3, 4}
   var args2 = fArgs{4, 3}
   fmt.Println("a = 3, b = 4 => a-b = ", structArgsFunc(args1))
   fmt.Println("a = 4, b = 3 => a-b = ", structArgsFunc(args2), "\n")

   // 4.2 Default parameters
   fmt.Println("Default 45:")
   defParamFunc()

   fmt.Println("a[0] = 4:")
   defParamFunc(4)
}
Output:
a = 3, b = 4 => a-b =  -1
a = 4, b = 3 => a-b =  1

Default 45:
a in defParamFunc:  45
a[0] = 4:
a in defParamFunc:  4
```

Explanation:

Go doesn't have named keyword arguments but structs can be used to imitate a structure like Python's **kwargs. I showed an example in 4.1. Go doesn't have default parameters but variadic arguments can be used to imitate default arguments and example for that is in 4.2.

Ease of use:

Most of the big cons about Go are these little features that most of the users are familiar with. List of the features that Go doesn't have but most of the modern languages have is surprisingly long: Exit-controlled loop, ternary operator, default parameters, keyword arguments, etc. None of these makes writing Go code significantly harder but it can be annoying to use a if-else structure instead of, for example, "user ? user.getName : "NoUser" " as in JavaScript.

Learning strategy and sources:

My research experience was learning Go doesn't have practically any quality of life features and searching for work-arounds around them. Then I prepared examples to explain them.

Research Sources and Tools

Tools

GoLand 2021.3.1 as IDE

Go 1.17.3 as SDK.

Sources

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

Keyword parameters
https://groups.google.com/g/golang-nuts/c/oWeeqCJfRzk?pli=1

Default parameters
https://stackoverflow.com/a/19813113

I had some prior knowledge of the general language as well.