Hammad Khan Musakhel

21801175

Section 3

HW3

JULIA

Nested subprogram definitions

```
println("HW3 Hammad Khan Musakhel 21801175 \n")
#1
function outer1() #first subprogram
  function outer2()#defined second subprogram
    function outer3() #3rd nested subprogram
    z = 3
    println("now we are in function outer ", z)
    end
  outer3() #call for 3rd nested subprogram
  z -= 1
  println("now we are in function outer ", z)
  end

  outer2() #call for second nested subprogram
  z -= 1
  println("now we are in function outer ", z)
end
outer1() #call for the primary subprogram
#println(z), gives error
println("\n")
```

The nested subprograms in Julia are very convenient to use. Julia has the option o f having several nested subprograms. The syntax is fairly similar to that of Python for functions. As we can see **outer1** is the outer most function that is supposed to followed by **outer2** and **outer3.** It can be noted that subprograms in Julia can access the variables defined in scope of the outer most function; as we can see **z** is defined first in the **outer3** function but is accessed in all the outer functions that are beyond the scope of the function **outer3**; it can be evaluated the same for the parameters. This

helps in the making of Julia a relatively simple language to use especially in nested subprograms with usage of variables initialized in different subprograms. Nested subprograms have the same functionality is Julia as other generic languages. It can be evaluated to be a subprogram-friendly language; one can define a subprogram inside another one and then call it in the proceedings. This allows the user to have more freedom in programming subprograms. The result for the above program is:

now we are in function outer 3

now we are in function outer 2

now we are in function outer 1

This proves the concepts stated above.

Scope of local variables

```
#2
glo_v = 5
live = "Turkey"
cash = 100
function visit()
  println(glo_v)
  #variables in the function
  live = "greece"
  live1 = "Mykonos"
  cash = 50
  println(live)
  println(live1)
  println(cash)
  live = "Hungary" #update the local str
  cash = 25
  println(live)
  println(cash)
end

visit()
println(live)
println(cash)
println(glo_v)
#println(str1) gives error
```

The scope of variables is an important concept across programming platforms. It allows the user to have the knowledge of which variable is durable to which extent. The scope pf variables is quite concrete I'm Julia. We can see how the program has initialized variables **live** and **cash**; we then provide a function that has the same names for the variables. However, we can see that the scope of variables in mainly limited to the function they are in at (unless and until passed as parameters where they still cannot be altered but copied). The essence of the core feature is that a variable initialized outside the function (unless global variable which in this case is glo_v = 5, cash = 100, and str = Turkey) cannot be accessed inside the function (with no passing) and the variable initialized inside the function cannot be accessed by programs outside the function. We can see how live retains its value across the function although it is also used inside the function but scoping produces a huge margin; we can see how cash and live are printed as they are in the last and if any variable inside the function is accessed outside produces an error. Live1 is a local variable of the function in the program and thus cannot be accessed beyond the function's body. The result for the above program is:

5

greece

Mykonos

50

Hungary

25

Turkey

100

5

We can see how Turkey and 100 are printed in the end although variables of its name are used and called inside the function. Julia has a restricted local variable space, consequently asserting. Although parameter passing plays a vital role in this regard which is explained in the next section.

Parameter passing methods

```
#3
v = 4
function f5(x,y) #conventional
  x = x + 1
  x + y

end
println(f5(v,2))

f6(x,y) = x * y
println(f6(v,2))

function Date(y::Int64, m::Int64=1, d::Int64=1)
    println(y," ", m," ", d)
end
Date(2000, 12)
```

The Language of Julia has primarily several ways of passing parameters depending on the scenario. Firstly, the conventional way of passing parameter yet with no defined type (a feature that adds to the quality of Julia) is depicted in **function f5(x,y)** with call by copying the variable is done. We pass the variable **v** to the function f5 and increment the value of **x** that has copied the value of v and see the desired result of 5 + 2 = 7. However, if we pass **f5(0.1, 2)** and print it we can see it will give result of 2.1, as the type isn't defined earlier and assigned when the parameter is passed; in essence we use "pass-by-sharing" in Julia where the values are rather shared with the parameters and the parameters rather retain themselves a new value. The second way is a bit new as it is so short: **f6(x,y) = x * y,** f6 doesn't need the word function by it and **=** is used to assign the function and its parameters to a programming behavior, in this case multiplication. The result for **f6(v, 2)** is 8 as **v = 4**.

Keyword and default parameters

```
#4
println()
function Date(y::Int64, m::Int64=1, d::Int64=1)
    println(y," ", m," ", d)
end
```

```
Date(2000, 12)

function f7(x, y)
  return x + y
end

y = f7(6,7)
println(y)

function f8(x, y)
  x * y + x - y
  nothing
end
println(f8(7,8))
u = f8(7,8)
println(u)
function f9(x, y)
  x * y, x + y
end
println(f9(3, 5))
```

We can also have default parameters as depicted in the first set of instructions; the type is defined and if a parameter isn't passed to the set of parameters then the default value is depicted. Thus, when **Date(2000, 12)** is passed, we still get the result of **2000 12 1;** The default parameters could be of String and other types. Secondly, Julia has the keywords "return" and "nothing." We can use them as needed. The return is used to return the value to a variable and nothing is used to allow the user/reader to revise that it is a function with void type. We can also also have multiple return types in a function as depicted in the program above. The result of the program would explain all the concepts mentioned factually:

2000 12 1

13

nothing

nothing

(15, 8)

Closures

```
println()
check = 7
function createAdder(x)
  println(check, " global")
  println(x, " 1st param")
  println("break")
  function(y)
    println(check, " global")
    println(y, " param of closure")
    println(x, " 1st param")
    return  x + y
  end
end

sum1 = createAdder(11)
#println(sum2)
println(sum1(7))
```

Julia also supports Closure, owing to its variable scopes inside subprograms. Thus, we can determine the closure is a subprogram that can be called from any arbitrary place in the program. The function is defined inside the function and doesn't need to be called rather has its own way; **sum1 = createAdder(11)** passes the param of 11 to the function but it isn't executed to the full rather sum1 is the container now as it is assigned. Next, calling **sum1(7)** results in the execution of the entire function to the full and the sum is also produced and also printed. However, in the program above, we have depicted the behavior of closure overall; how and what are called when **createAdder(11)** is called and what is called when **sum1(7)** is called. The result below explains it:

```
7 global
11 1st param
break
7 global
7 param of closure
11 1st param
18
```

The break and content above it is called when **createAdder(11)** is called whereas after break content is called after **sum1(7)** is called.

A section that includes your evaluation of Julia in terms of readability and writability of subprogram syntax.

The program of Julia is quite writable when it comes to the syntax of the subprograms (nested or not); initially the return type isn't necessary to be included by the function and the type of parameters are also not needed as we can see in the case above; this gives the writer greater freedom in terms of writing however these features collectively make the language less readable as one doesn't really know what type of function is it given that a function is hundreds of lines of code. However, the feature of not asking for parameter types is a very new and contemporary feature that makes it very writable. Not only this, the language doesn't have and right brackets to mark the start and end of functions rather it has the word "end" to mark the end of a function. This feature makes the code more readable and more writable along with the feature of not writing the type of variable to be initialed as int or double. We now know what the variable consists of by looking what is on the right side of the assignment operator. It also has the shortcut features to add and subtract variables making it more writable.

A section about your learning strategy

Being particularly flooded with assignments, I found this homework really out of place however thanks to the internet I was able to do it easily. Julia is a fairly new language and it seemed that not many have used it right now if compared to JS and Python or Java. However, to find the particular information for the first tree parts I sought help from: **https://docs.julialang.org/en/v1/manual/functions/** , **https://craftofcoding.wordpress.com/2017/07/18/is-it-a-subroutine-or-a-function-or-a-procedure/** and **https://discourse.julialang.org/t/subroutine-like-functions-in-julia/4626**

This particular helped me in gaining relevant information regarding the syntax and structıre of the programming language of Julia. Next, I also consulted slides from y CS315 class and chapter 9 from the Course Book to clarify my concepts. Lastly, for

closure I sought help from: **https://julia-lang.programmingpedia.net/en/tutorial/5724/closures**. This is it, I was able to do it with help from these websites.

Online Compiler:
https://repl.it/@hammadmacho7/main#main.jl