Turan Mert Duran

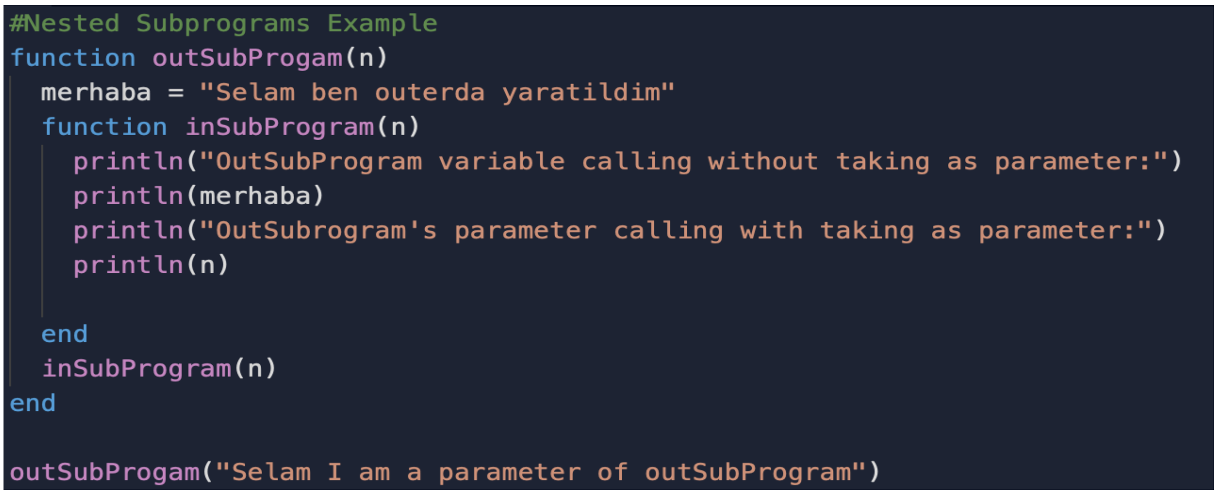
21601418

CS315- Programming Languages

Homework 3

Nested Subprogram Definitions:

Julia programming language allows programmers to code nested subprograms. It provides comfort to programmers in terms of writability of language. Parameters can both come from as parameter inside of inner subprograms or without parameters. Inner subprograms are checking outer subprograms and looking for variable.

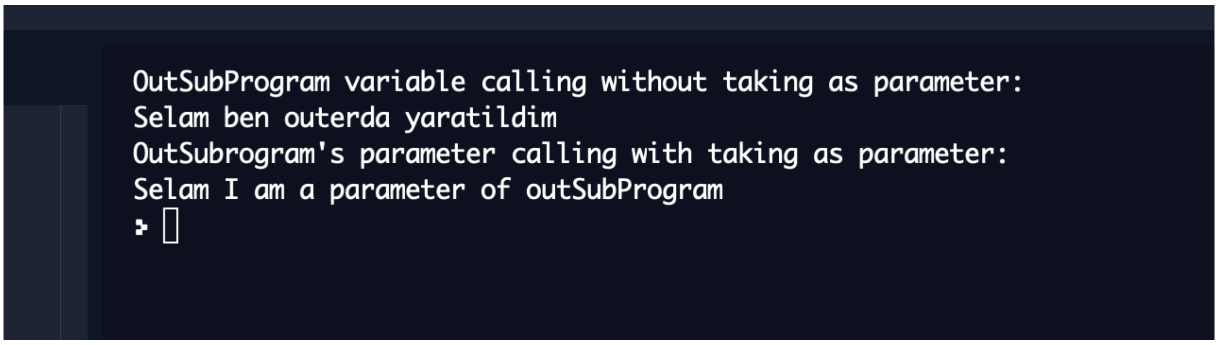


```julia
#Nested Subprograms Example
function outSubProgam(n)
  merhaba = "Selam ben outerda yaratildim"
  function inSubProgram(n)
    println("OutSubProgram variable calling without taking as parameter:")
    println(merhaba)
    println("OutSubrogram's parameter calling with taking as parameter:")
    println(n)

  end
  inSubProgram(n)
end

outSubProgam("Selam I am a parameter of outSubProgram")
```

Figure 1: Nested Subprograms Code Example



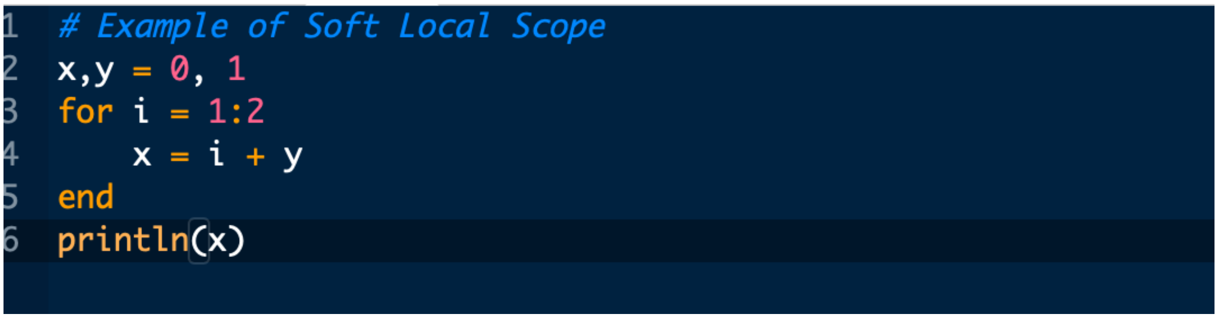
```
OutSubProgram variable calling without taking as parameter:
Selam ben outerda yaratildim
OutSubrogram's parameter calling with taking as parameter:
Selam I am a parameter of outSubProgram
```

Figure 2: Nested Subprograms' Example Code Execution Output

As it can be seen in Figure 1 above, variable which isn't created inside of inSubProgram subprogram, "merhaba" is used inside inSubProgram although it is not taken by parameter or etc. Inner subprograms are looking variables outside of their scopes. As it can be seen in "merhaba" example. On the other hand, nested subprograms are allowed to take parameter that is taken as parameter by their outer subprograms. "n" variable is one of these examples of it. As it can be seen at Figure 2, in both situation inSubProgram (nested subprogram) reached the variables' values and printed them.

Scope of Local Variables:

In Julia programming language there are two types for scopes of local variables [1]. First one is soft local scope. In soft local scoping all variables are taken from parent scopes unless they are introduced as keyword “local”. In the while loops, list comprehensions, let blocks and for loops they are used as soft local scope and variables inherited from parent scopes. Such as:



```julia
# Example of Soft Local Scope
x,y = 0, 1
for i = 1:2
    x = i + y
end
println(x)
```

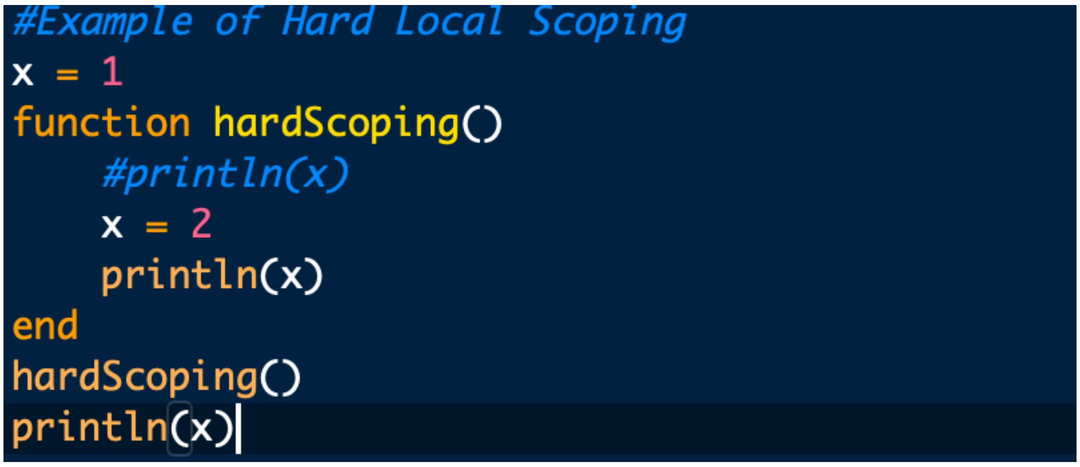
Figure 3: Example of Soft Local Scoping

In figure 3, it can be seen that x is initialized as 0 at the beginning of the code. After calling for loop, in for’s scope it’s value is going to change because in the block of the for there is no x that is initialized again. It goes to parent scope and takes x as parameter and change its value. The result will be x = 2 + 1 = 3 as it is expected.

```
$julia main.jl
3
```

Figure 4: Output of the code that is given on Figure 3

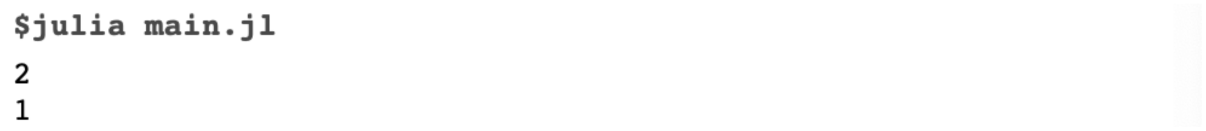
Second type of local scoping in Julia programming is the hard local scope. In function definitions, macro definitions and type immutable blocks Julia’s hard local scoping is used [1]. In hard local scoping all variables are taken from parent scopes as how it was in soft local scope unless variables are defined with keyboard “local” or a result of a variable can be defined inside of the scope as global then it’s values can be changed inside of functions.



```
#Example of Hard Local Scoping
x = 1
function hardScoping()
    #println(x)
    x = 2
    println(x)
end
hardScoping()
println(x)
```

Figure 5: Example of Hard Local Scoping

As it can be seen in Figure 5, variable of x is defined as 1 in the outside of the scope of function definition. Value of x cannot be accessed from the function hardScoping. If the "#" is removed from the line 4 on Figure 5, code gives error and says x not defined. In function a new x is created as 2 locally and printed its' value. However according to hard scoping unless inside of function global has not written this will not affect our outside parent x and after calling function x's value should remain same.

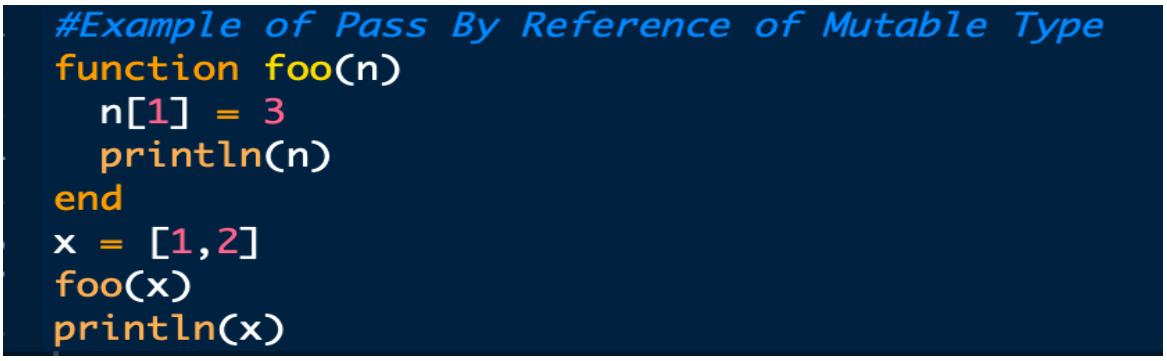


```
$julia main.jl
2
1
```

Figure 6: Output of Hard Local Scoping example given on Figure 5

Parameter Passing Methods:

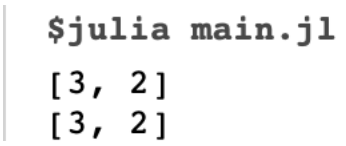
Julia uses pass by sharing convention for passing parameters [2]. In this language there are two types of variables mutable or immutables. Mutables are consist of objects arrays etc. They are pass by reference. On the other side immutables, which are consist of integers, bittypes etc, are passing by value.



```julia
#Example of Pass By Reference of Mutable Type
function foo(n)
  n[1] = 3
  println(n)
end
x = [1,2]
foo(x)
println(x)
```

Figure 7: Example Of Pass By Reference Of Mutable's

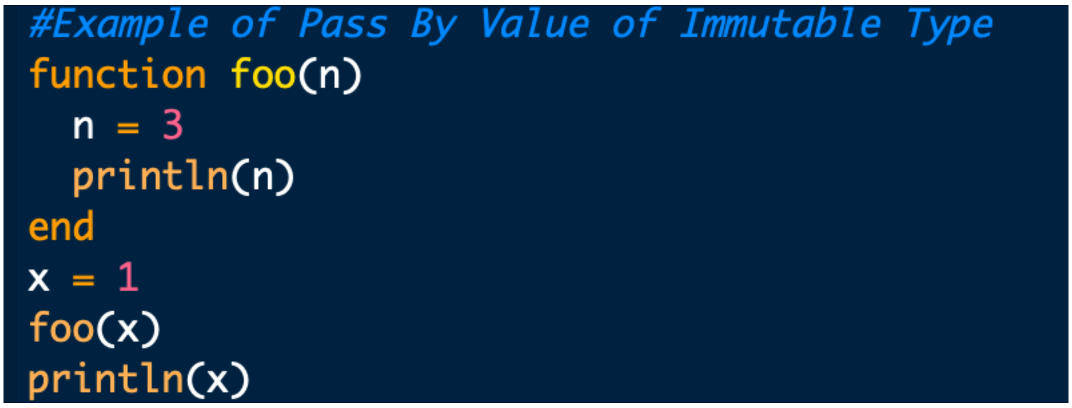
In the example that is given on Figure 7, a mutable (integer array) is passed to the function called foo. Because it's type is mutable it is expected to changing value of first integer of array.



```
$julia main.jl
[3, 2]
[3, 2]
```

Figure 8: Output Result Of Pass By Reference Of Mutable in Figure 7

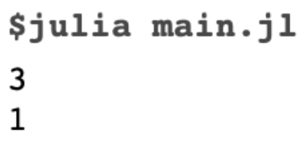
It can be seen that changing of inside of mutable array is expected and changed. If it was immutable type, the value inside would not be changed.



```julia
#Example of Pass By Value of Immutable Type
function foo(n)
  n = 3
  println(n)
end
x = 1
foo(x)
println(x)
```

Figure 9: Example Code Of Pass By Value of Immutable Type

X is an integer that is created in parent scope of function foo. It initially has 1 in it. However, after passing x to function foo, its value is not changing and it remains its initial value although it can be seen as changed its' value inside of foo.



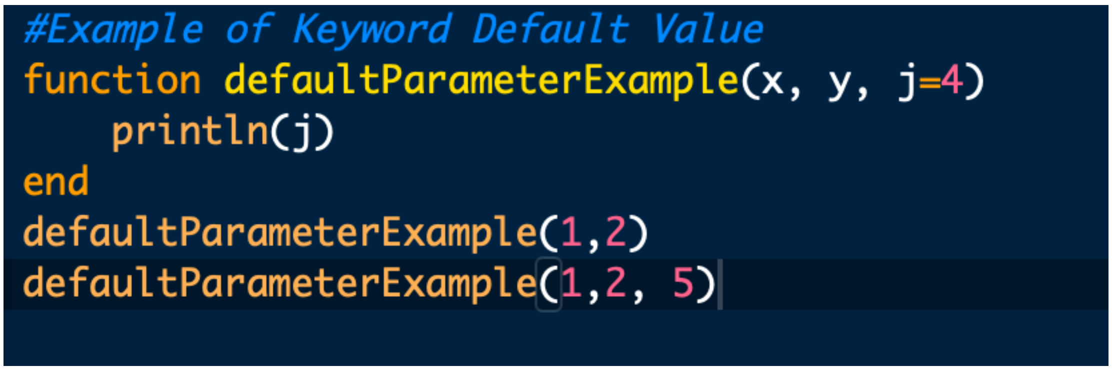
```
$julia main.jl
3
1
```

Figure 10: Output of Pass By Value Immutable Type in Figure 9

Expected result can be seen on Figure 10. Value of X is not changed after execution of foo. Because it is immutable type and immutable types are passing by value on the contrast of mutable types.

<u>Keyword and Default Parameters:</u>

Keyword argument default values are available in Julia. They are used for only corresponding keyword arguments is not passed.[3]



```
#Example of Keyword Default Value
function defaultParameterExample(x, y, j=4)
    println(j)
end
defaultParameterExample(1,2)
defaultParameterExample(1,2, 5)
```

Figure 11: Example of Keyword Argument Default Values

In the example that can be seen on Figure 11, a function called defaultParameterExample is defined as taking 3 different parameter x,y,j. However, parameter j is given 4 as default. It means that if j is not passed through function, function realizes its default value and puts value of j in its code block. Therefore it is expected that code that is available on Figure 11 should print out firstly 4 because j is not given as parameter, and after 5 because 5 is given as value of j on second call of function.

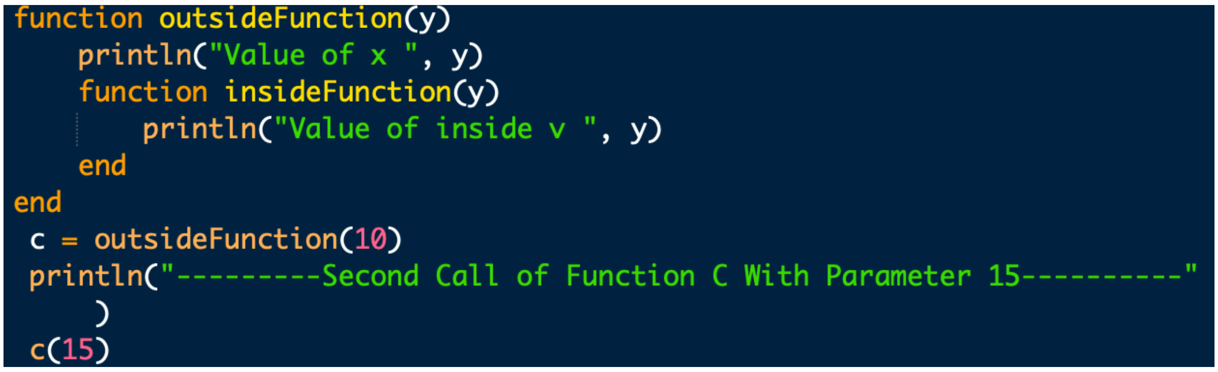
```
$julia main.jl
4
5
```

Figure 12: Output of Keyboard Argument Default Values on Figure 12

Program gave expected results and put j into 4 as default in first call and on the second call j's value is given 5 and value of j is changed as 5 in the second call.

Closures:

Functions that are defined within other functions are called closures [4].



```
function outsideFunction(y)
    println("Value of x ", y)
    function insideFunction(y)
        println("Value of inside v ", y)
    end
end
 c = outsideFunction(10)
 println("---------Second Call of Function C With Parameter 15----------"
     )
 c(15)
```

Figure 13: Closure Coding Example

As it can be seen in Figure 13, there are two functions one of them defined inside of other. In this example insideFunction is a closure function example. By writing c = outsideFunction(10), c is becoming closure function. After calling c again and giving it's parameter it is expected that parameter of inside's function should be taken from calling c(parameter). It should execute inside(closure) function as it can be seen as in Figure 14.

```
$julia main.jl
Value of x 10
---------Second Call of Function C With Parameter 15----------
Value of inside v 15
```

Figure 14: Closure Output Example

<u>Evaluation of Julia in terms of readability and writability of subprogram syntax:</u>

In terms of readability and writability of subprogram, Julia language was easy to write and read. Actually, I liked the way that how it uses pass by reference by understanding it's a mutable type and how it uses pass by value by understanding it's type as immutable like integer. It provides comfort programmers not to thinking about whether arrays, objects are passed by value or not. The language handles when passing. On the other hand, creating nested subprograms and closures are easy to use, their syntax is not much different than other programming languages. It is providing default keyword default values which makes programmers life easier. Scoping of local variables, a little bit complex I think, it reaches outer scope's value and changes if it is soft local scope, and cannot changing the value if it is hard local scope. It can be forgotten and can cause confusions.

<u>My Learning Strategy:</u>

I started firstly by looking what are subprograms generally. I searched them on book of course. Then I started to check how these subprograms are designed in Julia programming language. I used official website of Julia while checking syntax of language. I noticed that syntax of the language Julia was not different than Java that I am familiar. Thus, writing codes and understand its' syntax was not too difficult for me. I was using repl.it (Online compiler tool) on before homeworks but I realized that repl.it had Julia's past versions (I think it should be Julia 0.4 version). Thus, I changed my online compiler and used tutorialspoint's online Julia compiler. For editing and designing my code I used VSCode and download Julia syntax extension for it. Thanks to VSCode's extension of Julia I didn’t face difficulties while getting familiar with syntax of Julia.

Used Tools & Compilers:

- https://www.tutorialspoint.com/execute_julia_online.php (Compiler)
- VSCode (Used as code editor)

Informative Websites That I Used While Learning Julia:

- https://stackoverflow.com
- https://www.w3schools.com
- https://www.tutorialspoint.com

References:

[1] “Scope of Variables.” Scope of Variables - Julia Language 0.4.7 Documentation, matrix.umcs.lublin.pl/DOC/julia-doc/html/manual/variables-and-scoping.html. Access Date: 21.12.2020

[2] Pass-by-Sharing, web.eecs.umich.edu/~fessler/course/598/demo/pass-by-sharing.html. Access Date: 21.12.2020

[3] “Functions.” Functions · The Julia Language, docs.julialang.org/en/v1/manual/functions/. Access Date: 21.12.2020

[4] Community, SO. “Closures.” Julia Language Tutorial, Julia Language Pedia Https://Julia-Lang.programmingpedia.net/Favicon.ico, 18 Sept. 2016, julia-lang.programmingpedia.net/en/tutorial/5724/closures. Access Date: 21.12.2020