CS 315 HW3

BURAK YETİŞTİREN 21802608 SECTION-1

Table of Contents

Nested subprogram definitions

In Julia, programmers can define nested subprograms. The scopes of the variables apply for the subprograms too; the discussion for scope will be done in the next section. The parameter passing methods also apply for nested program definitions; this discussion will be held in the "*Parameter Passing Methods*" section. For now, the following code segment and the provided output for this segment should demonstrate an example behavior of nested subprograms in Julia:

Definition:

```
function sumUpToN(n, name) # outer fuction definition
    sum = 0 # sum is initially zero
    print("Hello ", name) # initially, "Hello " and name is printed

    function calculate(n)  # inner function definition,
                           # parameter n is passed from the outer function

        for i = 1:n # for loop in the inner function iterated n times
            sum = sum + i
        end
    end
    calculate(n) # call to inner function with parameter n
    println(" your result is: ", sum) # result calculated in the inner function
                                      # is printed.
end
```

Input:

```
x = 5
name = "Burak"
sumUpToN(x, name)

x = 10
name = "Alper"
sumUpToN(x, name)
```

Output:

```
Hello Burak your result is: 15
Hello Alper your result is: 55
```

In the function definition, there are two nested functions. The outer function is called "*sumUpToN*" and takes two integers, one for name, and the other for the number, which is the upper limit for the positive integers to be added together. In the outer function, a variable for sum is declared and initialized to 0. Then the first parameter is taken directly, and it is printed. Followingly, a call is made to the inner function with the parameter for the upper limit explained before. In the inner function, there is a for loop to sum the integers up to the limit, so the limit is determined as the limit of the counter-controlled for loop. In the for loop, the variable "*sum*" defined on the outer function is manipulated. After the loop terminates, the inner function terminates, and the value of the variable "*sum*" is printed. The input and output codes above exemplify the explained procedure.

Scope of local variables

In Julia, mostly a new local scope is created for code blocks [1]. There are three different paths in Julia for a newly defined local variable [1]. If the is existent, meaning it is already a local variable, that variable is assigned with the new one [1]. The scopes in Julia, are concerned with the definition of local variables. They are named as "*hard*", and "*soft*" scopes and these scopes are determinant about the scope of the newly defined variables. If the assignment occurs in a "*hard scope*", meaning that, in a "let block, function or macro body, comprehension, or generator", if the variable is not already defined in this scope, a new local is created in the scope of the definition [1]. In the "*soft scope*", meaning that in "*loops, try/catch blocks, or struct blocks*", if the variable is not a local variable, and if the scopes, that the variable is defined are again "*soft scopes*", the behavior changes case to case [1]. If the variable is not defined globally, a new local is created in the scope of the definition; if not, there occurs an ambiguity [1]. There are two distinct behavior to overcome this ambiguity; the first one is that if the program is being run on a file, not in console, etc., a warning message is created, and a new local is created; in the other case, such that if the program is being run on the console, no warning message is given, and the global variable is manipulated [1]. Example code segments below should further explain the discussion.

Definition:

```
result = 2
function area(radius) # the function takes a parameter for radius

    result = pi * radius^2  # variable result is assigned to the
                            # formula for the area of a circle.
                            # Pi is a constant available in the language

    check = "Am I visible?"

    for i = 1:1 # new scope is created by a for loop
        msg = "I love maths!" # new variable msg is created
    end

    return result, @isdefined(msg) # return statement to get the result and check,
                                   # if the variable msg is defined
                                   # in the scope of the function
end
```

Input:

```
println(area(5)) # function call
@show (result) # to see the value of 'result'

println("Variable 'check' is defined?: ", @isdefined(check)) # to see if variable check is defined here
println("Variable 'result' is defined?: ", @isdefined(result)) # to see if variable result is defined here
println("Variable 'msg' is defined?: ", @isdefined(msg)) # to see if variable msg is defined here
```

Output:

```
(78.53981633974483, false)
result = 2
Variable 'check' is defined?: false
Variable 'result' is defined?: true
Variable 'msg' is defined?: false
```

In this code segment, there exists a function, which basically calculates the area of a circle, with a given radius. To test the local scopes there exist some unrelated statements in the function. As stated, functions create "*hard scopes*" in the Julia language. This function also creates a "*hard scope*", therefore the variable called, '*result*' is created in the function, which is different than the variable with the same name defined in the global scope. This can be seen in the output section. The function returns the result, and as it can be seen, the result is approximately 78.54; whereas the result in the global scope is 2, it is unaffected by the operations inside the function. This phenomenon can be further seen by the '*check*' variable inside the function. After returning from the function, as it can be seen above, the "*@isdefined()* " function, which returns Boolean values whether the parameter given to it checks if it is defined in that scope, is called for this variable in the global scope, and it returned false. The variable "*msg*" in the for loop in the function is only defined within the scope of the for loop, as it can be seen from both calls to "*@isdefined()* ", for "*msg*" one in the function, and the other one in the global scope, that they return false.

Input:

```
increment = 1 # globally defined variable
warningCreator = "Hi" # globally defined variable
for i = 1:3 # for loop (soft scope)
    warningCreator = "This statement will create a warning!"

    let increment = increment # here we assign a local variable 'increment'
                              # to the global 'increment'
        increment = "IN LET"
        @show increment
    end

    global increment += 1 # the 'global' keyword is used to
                          # manipulate the global 'increment'
end
@show (increment) # we print the result of the increment
```

Output:

```
┌ Warning: Assignment to `warningCreator` in soft scope is ambiguous because a global v
ariable by the same name exists: `warningCreator` will be treated as a new local. Disam
biguate by using `local warningCreator` to suppress this warning or `global warningCrea
tor` to assign to the existing global variable.
└ @ C:\Users\burak\Desktop\CS315 HW3\2.jl:35
increment = "IN LET"
increment = "IN LET"
increment = "IN LET"
increment = 4
```

This code segment demonstrates the behavior of the variable definitions, and the statements in a "*soft scope*" structure, a for loop. Firstly, two global variables are defined in the global scope, "*increment*", and "*warningCreator*". The for loop is a counter-controlled for loop iterates from 1 to 3. As it can be seen from the example code above, when it is tried to manipulate the variable defined in the global scope, a warning occurs, saying that this variable will be treated as a new local, which was explained in the beginning of this section. To handle this properly, three different approaches can be used. If the variable is wanted to be used in the function without errors, a new scope can be created in the function with the keyword "*let*". When we take a look at the example code above, the local "*increment*" is created by assigning the global "*increment*". The operations done to the variable do not affect the one in the global scope, as it can be seen in the output of the example code. The values printed for the variable in the loop is "IN LET", but the value of the global variable is 4, after the loop terminates. The second way is to use the "*global*" keyword. This way, we make sure that the variable in the global scope is manipulated, which can be seen in the example code. The third way is to use the "*local*" keyword, whose functionality is not shown in this code, but will be demonstrated in the following code segment.

Input:

```
let # hard scope
    course = "Outer: CS315"
    let # hard scope
        local course = "Inner: CS315 Section 1" # local keyword is used to create
                                                # a new variable with the same name
        println(course)
    end
    println(course)
end
```

Output:

```
Inner: CS315 Section 1
Outer: CS315
```

In this code segment, the behavior of the "*let*" scope, and the "*local*" keyword is tested. As it can be interpreted from the output of the given code, the variable "*course*" in the outer "*let*" and the inner "*let*" are different, meaning that the "*local*" keyword created a new variable with the same name in the inner "*let*".

Definition:

```
function tryNormal() # in this function, in the
                     # for loop the 'outer' keyword is not used
    isHere = "I am not here"
    for isHere = 1:1
        isHere = "I am here"
    end
    return isHere
end

function tryEnhanced() # in this function, in the
                       # for loop the 'outer' keyword is used

    isHere = "I am not here"
    for outer isHere = 1:1
        isHere = "I am here"
    end
    return isHere
end
```

Input:

```
println(tryNormal())
println(tryEnhanced())
```

Output:

```
I am not here
I am here
```

In this code segment, the behavior of the keyword "*outer*" is tested, in usage with the for loop. There are two functions defined "*tryNormal()*", and "*tryEnhanced()*" respectively. In the first function "*outer*" keyword is not used in the for-loop definition, but it is used in the second function. When we take a look at the output of the codes, in the first function, the variable "*isHere*" on the outside of the loop was not manipulated, meaningly, the "*isHere*" variable in the for loop was different than the one in the outside. On the other hand, in the second function, the variable on the outside was manipulated by the keyword we have used, meaning that the variable inside the for loop pointed to the one in the outside.

Parameter passing methods

The parameter passing behavior in Julia does not have a special name, but in literature, the method used is usually called "*pass-by-sharing*" [2]. Meaning that the arguments of the functions are "*new variable bindings*" [2]. But for "*mutable*" types like arrays, are passed like the method of "*pass-by-reference*" [3]. The example code below should further demonstrate this phenomenon.

Definition:

```
function parameterPass1(x) # this function tests the parameter-passing
                           # for mutable objects like arrays
    x[1] = 5
    @show x
end
```

Input:

```
x = [1,2] # an array is initialized
@show x # array is printed before the function call
parameterPass1(x) # array is passed to the function as a paramater
@show x # array is printed after the function call
```

Output:

```
x = [1, 2]
x = [5, 2]
x = [5, 2]
```

In this code segment, the behavior of the parameter passing in Julia for the mutable objects like arrays is tested. The function takes an array as a parameter and manipulates its first item (changes it to 5). To test this function, I have created an array, printed the array, and passed it to the function. In the function, and after returning from the function, I have printed the array again. I have seen that the array was changed in and outside of the function. Meaning that the reference of the array was passed to the function.

Definition:

```
function parameterPass2(x) # this function tests the parameter-passing
                           # for other types
    x += 1
    @show x # variable is printed in the function
end
```

Input:

```
x = 4 # a variable is initialized
parameterPass2(x) # variable is passed to the function as a parameter
@show x # variable is printed after the function call
```

Output:

```
x = 5
x = 4
```

In this code segment, the behavior of the parameter passing in Julia for the other types is tested, integers in this example. The function takes a variable as a parameter and increments it by one. To test this function, I have initialized a variable, and passed it to the function. In the function, and after returning from the function, I have printed the variable again. I have seen that the variable was changed in the function, but not outside of the function. This means that the variable inside the function was different from the one on the inside and the reference of the outside variable was not passed, but the variable was copied having a different address.

Keyword and default parameters

In the Julia language, the programmers are allowed to have both keywords, and default parameters for function parameters [4] [5]. The programmer has to separate the parameters without keywords and default values with the ones, with those, with a semicolon [4] [5]. The parameters without keywords and default values, must be defined firstly, the other ones come after the semicolon [4] [5]. If the programmer wants to have variables with default keywords, s/he must give them keywords [4]. The program segment below should demonstrate these phenomena.

Definition:

```
function printInfo(name, surname; age, nationality, pronoun="")
# parameters age, nationality, and pronoun are keyword parameters;
# variable pronoun has also a default value

    println(name, " ", surname," is ", age ," years old and ", pronoun, " is ", nationality, ".")
end
```

Input:

```
# different combinations for variables are tested, in the last two function calls,
# the default parameter is tested
printInfo(age=22, "Burak", nationality="Turkish", pronoun="he", "Yetiştiren")
printInfo("İlayda", "Pekgöz", pronoun="she", age=21, nationality="Turkish")

printInfo(age=22, "Burak", nationality="Turkish", "Yetiştiren")
printInfo("İlayda", "Pekgöz", age=21, nationality="Turkish")
```

Output:

```
Burak Yetiştiren is 22 years old and he is Turkish.
İlayda Pekgöz is 21 years old and she is Turkish.
Burak Yetiştiren is 22 years old and  is Turkish.
İlayda Pekgöz is 21 years old and  is Turkish.
```

In this program segment, there is a function called "*printInfo()*", which takes two parameters without keywords and default values, and three keywords with keywords, one of them with a default value. In the function there exists a print statement which prints information about a person with information given as parameters. To test this function, I have called the function four times in which I have changed the order of the arguments. In the last two calls, I have not provided the argument, which has a default value in the function definition. If we take a look at the outputs, we can see that the order of the arguments does not change the output, but the arguments without keywords, and default values should be called in order, and given values. It should be obvious why we should call them in order, that if we do not, the places of name and surname would be replaced with one another. If we do not provide parameters without default values, with anything an error occurs. For the others, the default value is used, as it can be seen from the output.

Closures

In Julia, the functionality of the closure can be done in three ways. One can use "*function composition*", "*implementing currying*", and the normal way of closure like in the other programming languages [6]. Their functionality will be demonstrated below.

Definition:

```
function square(x) # function calculating the square of the given value
    x = x^2
end

function multiplyByTwo(x) # function calculating the double of the given value
    x = 2x
end

# ways of defining composite functions
composite1 = ∘(square,  multiplyByTwo)
composite2 = square ∘ multiplyByTwo
```

Input:

```
# test
x = 2
x = composite1(x)

y = 2
y = composite2(y)
```

Output:

```
x = 16
y = 16
```

This code segment demonstrates the usage of the "*function composition*" in Julia. The function composition works like the function composition in mathematics. I have defined two functions: one for taking the square of the given value, and one for calculating the double of the value given. In the next part two more functions are defined, they are basically the composite functions. The reason I have defined two composite functions was to show the two ways of definition in Julia. Mathematically speaking, the first function calculates $y = x^2$, and the second $y = 2x$; hence, their composite should be defining $y = 4x^2$. If we further look at the input, and output codes, we can see that the composites are indeed functioning properly and producing valid outputs.

Definition:

```
function powerFunc(x,y) # function calculating the power of given two values
    y^x
end
curry(f, x) = (x2) -> f(x, x2) # implementing currying

# crating closures
const powerBy2 = curry(powerFunc, 2)
const powerBy3 = curry(powerFunc, 3)
const powerBy5 = curry(powerFunc, 5)
```

Input:

```
# test
@show powerBy2(2)
@show powerBy3(2)
@show powerBy5(2)
```

Output:

```
powerBy2(2) = 4
powerBy3(2) = 8
powerBy5(2) = 32
```

In this code segment, the phenomenon of "*currying*" is tested. This means that some part of the parameters is provided initially, the others are provided later [6]. If we take a look at the example program, there is a function defined, which calculates the power of the two given variables. After this definition comes the "*currying*" part. In this part, we define that the curry function will take a function, and a variable as its parameters. Then in the next part, we implement the closures. The closures take the power of 2,3 and 5 for a given value. In the test part I have tested if my implementation was correct. My inputs intended to calculate the given powers of 2. The given powers are provided by the closures, and indeed the outputs calculated the 2^{nd}, 3^{rd}, and 5^{th} power of 2.

Definition:

```
# normal way to impelement closures
function adder(first) # outer function
    function (second) # inner function
        second + first
    end
end

# creating closure
adderBy10 = adder(10)
```

Input:

```
# test
@show adderBy10(5)
```

Output:

```
adderBy10(5) = 15
```

This code segment demonstrates the traditional way of implementing closure. In the function definition part, we have two nested functions. The outer function takes the first parameter, and the inner function takes the second. Then the sum of these two parameters is calculated. The closure is created by calling the outer function with a parameter. In my example, I have called the function with 10, creating an adder which sums a given parameter with 10. To test this, I have called the closure I have created with 5 and printed the result. In the end I have gotten 15, which was correct.

Evaluation of Readability and Writability of Julia in Terms of Subprogram Syntax

In terms of defining functions, creating loops, defining variables, I found Julia readable and writable. The syntax was quite straightforward, it was similar to most of the other programming languages, and close to the natural language English. The statements did not have extraordinary names, for example words like "*function*", "*end*", "*return*", "*local*", *global*", etc. were understandable, and natural names. In terms of leaning concepts about the programming language, Julia had different conventions providing the same functionality, this case reduced the readability, as someone can know some convention, but the written program used the other, such that the reader would have a hard time trying to understand the program. This case had instances in the design issues of scope, and closures more than the other design issues. There were many ways to define closures and some ways were more complex than the others, this issue also introduced problems about writability, additional to readability. The design issues about the scoping introduced so many different conventions, so that I have struggled to understand them all in the first place. The distinction of the scopes like hard, and soft scoping also reduced the readability and writability of the language. There were too many keywords to be able to work with scopes like "*global*", "*local*", etc. such that a newbie programmer for the language struggles so much while trying to understand the conventions by looking at the codes available on the internet. This way, I can say the readability is reduced. As for writability, once a programmer learns the conventions in the language, I think it is high, such that the programmer is given more freedom to play around in the program, unlike the C-group languages. A final negative note about the readability and writability would be, that there were different conventions about the runtime environment of the program, this is also touched upon in the "*Scope of the Local Variables*" of my report. There were differences whether the programmer ran a file or coded them into the console (REPL in Julia) line by line. In the first choice a warning is given, and a set of operations are done, those are different from the second option, in which there appears no warnings.

Learning Strategies That I Have Used

I did not have any prior knowledge to the language. I had to learn most of the concepts from other sources. Because of my knowledge about other programming languages, I have guessed some of the conventions of the language, some of them turned out to be true, some of them false. The sources I have used included the language definition provided by Julia [7], and some other outside sources like forums [3] [4] [5]. For the concepts independent of the Julia language, I have referred to the lecture slides. The slides helped me to understand the design issues in a more general way. But most of the time I have used the language definition provided by Julia, because it was comprehensive, and easily understandable, when compared to other languages.

I have implemented my own code following the conventions that I have learned from the sources. I have written some test codes to see if I was on the right path. I can say that the best teacher of mine was the errors I have made throughout this homework. The errors I have gotten taught me best however long time I have spent on reading the definitions. Also, my errors made the definitions more understandable.

Overall, I have gained a good insight to a language, that I did not know about. I think my leaning strategies were powerful, that I feel like that I have a good knowledge about the language, in the scope of this homework.

IDE & Interpreters I Have Used

In this homework I have used both an online interpreter and a local IDE:,

- **Atom IDE:** https://atom.io/, **Julia-client package:** https://atom.io/packages/julia-client
- https://repl.it/languages/Julia

References

[1] Julia, "Scope of Variables," n.d.. [Online]. Available: https://docs.julialang.org/en/v1/manual/variables-and-scoping/. [Accessed 22 December 2020].

[2] Julia, "Functions," n.d.. [Online]. Available: https://docs.julialang.org/en/v1/manual/functions/#man-functions. [Accessed 22 December 2020].

[3] stackoverflow, "Passing arguments to functions without copying them in JULIA-LANG," 14 February 2018. [Online]. Available: https://stackoverflow.com/questions/48781821/passing-arguments-to-functions-without-copying-them-in-julia-lang. [Accessed 22 December 2020].

[4] M. V. D. Vyver, "Best Practice: Default Values/Optional Arguments/Readable Code," May 2019. [Online]. Available: https://discourse.julialang.org/t/best-practice-default-values-optional-arguments-readable-code/24226. [Accessed 22 December 2020].

[5] stackoverflow, "Julia: How are keyword arguments used?," 27 September 2019. [Online]. Available: https://stackoverflow.com/questions/58132507/julia-how-are-keyword-arguments-used. [Accessed 22 December 2020].

[6] Julia Language Pedia, "Closures," n.d.. [Online]. Available: https://julia-lang.programmingpedia.net/en/tutorial/5724/closures#. [Accessed 22 December 2020].

[7] Julia Language Pedia, "Getting started with Julia Language," n.d.. [Online]. Available: https://julia-lang.programmingpedia.net/. [Accessed 22 December 2020].

[8] Atom, "Atom," n.d.. [Online]. Available: https://atom.io/. [Accessed 22 December 2020].