CS 315 – Homework 3
Subprograms in Julia

Maryam Shahid
21801344
Section 01
December 20, 2020

Design Issues

1. Nested subprogram definitions

Julia allows nested subprograms. They allow function decomposition which offers for a more readable code. The nested function has access to all parameters and variables declared in the function that contains it – which make them quite useful and valuable.

```
function outside()
  function inside()
    number = 2
    println("number inside nested function: ", number)
  end
  inside()
  number += 1
  println("number outside: ", number)
end
outside()
```

```
Output: number inside nested function: 2
        number outside: 3
```

The above code shows a function “outside”. The function contains a nested function namely “inside.” The nested function declares a variable – number and sets its value to 2. It then prints the number and a message for output purposes. Once the nested function ends, the block of the wrapper function continues. It increments the value of number by 1 and prints it as the output. This shows that the outside function has access to variables declared inside the nested function. This can be observed in the output.

2. Scope of local variables

In Julia, a new local scope is created by the declared subprograms. It is local by default, but can also be explicitly declared using the local keyword before new variables.

The following code shows the scope of local variables declared in a function. I have a global variable “name”, declared outside the subprogram. The first statement in the subprogram also has the variable name with a different value. Since the function has local scope, it creates a new variable and assigns it the value “maryam”. The value can be changed using an assignment operator (=). However, accessing the value of name outside the function shows that the value of the global variable remains unchanged due to local scope. This can be observed in the output.

```
name = "-no name-"
function nameFunc()
  name = "maryam"
  println(name)
  name = "maryam shahid"
  println(name)
end

nameFunc()
println(name)
```

```
Output: maryam
        maryam shahid
        -no name-
```

Similarly, Julia also consists of modules which act in a similar way as functions. It also provides a local scope for the variables. The following code shows a module "nameMod" which creates a new local variable – name. It assigns it the value "jem" and prints it to the console. Once the function ends and a print statement is called on name, it still prints the value of the global variable name. This emphasizes the local scope of variables declared inside the module. The value of the variable declared inside the module can be accessed using the nameMod.name statement.

```
module nameMod
  name = "jem"
  println(name)
end

println(name)
println(nameMod.name)
```

```
Output: jem
        -no name-
        jem
```

3. Parameter passing methods

Julia uses the generic "pass-by-sharing" parameter passing method for functions. This means that the values are not copied, instead function arguments act as new variable bindings. [4] This is represented in the following code. Two variables – a and b, are declared and given specific values. Then the function paraPass is called which takes them both as parameters and updates their value. This is printed in the first statement shown by the output. Then the current values of the global variables are printed which remain the same, as noted by the output.

```
function paraPass(a, b)
  a = a + b
  b = a - b
  println("a = ", a ," b = ", b) # values of parameters inside
function
end

a = 10
b = 20
paraPass(a,b)

println("a = ", a ," b = ", b)

Output: a = 30 b = 10
        a = 10 b = 20
```

4. Keyword and default parameters

In Julia, functions can have variables forms of parameters. We can provide a default value to a parameter by using the assignment operator (=). The following code declares a function "student1" which takes in two parameters. Using the assignment operator, the department parameter has the value CS, whereas the other parameter is bounded by position. To test the function, I created called the function using different parameters. In the first example I provided a value for name, but left department empty. This assigned it the default value. In the second example, I provided both parameters which ignored the default. This can be noted in the output.

```
function student1(name, department = "CS")
  println("bilkent student: ", name, ", ", department)
end

student1("maryam")
student1("mert", "PHY")

Output: bilkent student: maryam, CS
        bilkent student: mert, PHY
```

Julia also allows keyword parameters which holds its specified value by name. A semicolon (;) denotes the start of keyword parameters. In the following function, both parameters are keyword parameters. I call the following function twice with keyword parameters. Once in the same order, and once irrespective of the order. This does not affect the output as seen below.

```
function student2(; name, department)
```

```
  println("bilkent student: ", name, ", ", department)
end

student2(name = "shamin", department = "EE")
student2(department = "CS", name = "hammad")

Output: bilkent student: shamin, EE
        bilkent student: hammad, CS
```

Both parameters can also be combined in the following way. The parameters before the semicolon (;) are positional parameters - name. Whereas the parameters after the semicolon – department and uni are keyword parameters. Inevitably, the order matters in the positional parameters but keyword parameters can be written in anyway as observed below.

```
function student3(name, ; department, university)
  println(name, ", ", department, ", ", university)
end

student3("maryam", department = "CS", university = "bilkent")
student3("shamin", university = "fast", department = "EE")

Output: maryam, CS, bilkent
        shamin, EE, fast
```

5. Closures

A closure is simply a callable object with field names corresponding to captured variables. Since every object in Julia is callable, it is not hard to achieve this. This can be done in Julia in the following way. Here I have declared a function addNumber which takes the variable “a” as its parameter (no specific type). It then has another function in it – this defines a closure. In my code, this is an anonymous function but it can also be named. The function takes a parameter – “b”, and adds both the parameters – a and b, together.

```
function addNumber(a)
  function(b)
    return a + b
  end
end
```

The closure can be declared in the following way. Here add20 and add2 are both closures which will add 20 and 2 respectively to the statement that calls them.

```
add20 = addNumber(10);
```

```
add2 = addNumber(2);
```

Here add20 calls the function with 10 as its parameter which adds 20 to 10. The result is printed through the println statement. The same goes for add2 which calls upon the function with 3 as its parameter.

```
println("add 20 to 10: ", add20(10))
println("add 2 to 3: ", add2(3))

Output: add 20 to 10: 20
        add 2 to 3: 5
```

Evaluation of Julia

Julia is a fairly straightforward language and follows the generic syntax for subprograms. It allows the user to declare functions and modules with the respective keywords for each. It also consists of the end keyword to mark the ending of both. This greatly adds to the readability and writability of the language. It does not require types for variables or function arguments to be declared. This enhances writability but might create difficulties for readability. It follows the generic lexical scoping rules for its subprograms and also the local variables. This inevitably, increases readability of the language since it is fairly easy to tell the scope of local variables by simply reading the code. Julia also allows a lot of flexibility such as nester subprograms, a plethora of keyword and default parameters, closures etc. All this greatly adds to the writability of the language.

Learning Strategy

The biggest source that helped my with my research on subprograms in Julia was the documented manual provided by Julia's website. [1,2,4,7,8] The manual was extremely descriptive and thorough which fully helped me understand the language. I also found answers to some confusing issues such as closures in the discussion section provided by Julia's official website. [9] Multiple had used to it ask and answer questions that also came to my mind upon working on the design issues. Moreover, due to the extreme similarities between Python and Julia, I also used some articles and examples given for Python to figure out the same thing in Julia. For example, I studied keyword and default parameters in Python and used the examples and to develop my understanding for the same in Julia. [5] This turned out to be quite helpful. I also installed Julia on my computer. However, since it is in terminal, it does not keep track of the previously written code. Thus, ended up using the online compiler for Julia provided by Repl.it. [10]

References

[1] https://docs.julialang.org/en/v1/manual/variables-and-scoping/

[2] https://docs.julialang.org/en/v1/manual/modules/

[3] http://web.eecs.umich.edu/~fessler/course/598/demo/pass-by-sharing.html

[4]https://docs.julialang.org/en/v1/manual/functions/#Argument-Passing-Behavior

[5] https://www.programiz.com/python-programming/function-argument

[6] https://stackoverflow.com/questions/58132507/julia-how-are-keyword-arguments-used

[7] https://docs.julialang.org/en/v1/manual/functions/#Keyword-Arguments

[8]https://docs.julialang.org/en/v1/devdocs/functions/#Closures-1

[9] https://discourse.julialang.org/t/closures-section-in-documentation-is-not-clear-enough/18717/13

[10] https://repl.it/@maryamshahid16/homework3