BILKENT UNIVERSITY

CS315

Homework Assignment 3

Report

Subprograms in Julia

İrem Seven

21704269

NESTED SUBPROGRAM DEFINITIONS

In Julia, function definitions are executable. Thus, we can define a function in runtime depending on the given value of the outer nested function. Below it is given an example of this situation. [1], [2]

Example

In this example, nested_out1 function takes parameter n, and depending on the value of n the function nested1 will be defined accordingly. If n value is not a negative number, then the function will return false.

```
#----------design issue: nested functions -----
function nested_out1(n)
  if (n >= 0)
    function nested1(msg)
      println(msg)
      return "false"
    end
  else
    function nested1(msg)
      println(msg)
      return "true"
    end
  end
end
```

If we call the function defined above what we get first will be reference to the defined function. Thus, when we print is_neg we will get the function name defined in the outer function.

```
is_neg = nested_out1(8) #here we get the function name
println(is_neg)
```

Output:

```
nested1
```

When we call the function by using the name is_neg defined before, we will be able to use the inner function as defined before.

```
is_res = is_neg("hello")   #here we get return value
println(is_res) #we can reach the function even though outer not called
```

Output:

```
hello
true
```

SCOPE OF LOCAL VARİABLES

As most of the other programming languages, Julia provides two types of scoping, global and local. Considering the local scoping within subprograms, Julia uses lexical scoping for them. [3] Below it is given an example to explain this situation.

Example 1

In this example, in the function scope1 x is assigned to a value. On the other hand, there is also x value outside of the function, which is the global x value. Because of the lexical scoping, when we want to print x in the function the nearest x value will be used. This x value is a new local variable within scope1 function, and it does not change the value of the global variable.

```
#----------design issue: scope of local variables-----
#without global prename
x = "name" # global
function scope1()
   x = "id" # new local
   println(x)
end
```

By calling the function as below, we will get the following output.

```
scope1()
println(x)
```

Output:

```
id
name
```

Example 2

Unlike the previous example, if we use a global keyword before the x in the scope2 function, we will modify the global variable.

```
#with global prename
function scope2()
   global x = "surname" # global will change
   println(x)
end

scope2()
println(x)
```

Output:

```
surname
surname
```

PARAMETER PASSING METHODS

Julia uses “pass-by-sharing” as a parameter passing method. Thus, unlike C like languages values are not copied when they are passed to functions. As in Python, the passing will depend on the object being mutable or not. If the object passed is immutable, then when there is an assignment in the function, a new different object will be created. If a mutable object is passed then the parameter and the passed object will be the same. [1]

Example 1

Since 52 is an immutable object, changes to k will not affect my_number.

```
#------design issue: parameter passing methods (pass by assignment)-----
#immutable object, different object when assigned
my_number = 52
```

```
function pass1(k)
   k = k % 3
   return k
end
```

When we call as below, we will get the following output.

```
println(pass1(my_number))
println(my_number)
```

Output:

```
1
52
```

Example 2

In this case, my_numbers is a mutable object. Thus, the changes within the function will affect the original object too.

```
#mutable object
my_numbers = [5, 7, 2, 3]
function pass2(n1)
   n1[1] = 9
   return n1
end
```

When we call as below, we will get the following output.

```
println(pass2(my_numbers))
println(my_numbers) #it is also changed
```

Output:

```
[9, 7, 2, 3]
[9, 7, 2, 3]
```

Example 3

In this example my_numbers2 is again a mutable object. However, because of the language design, the operation in the function will create a new object, so my_numbers2 will not change.

```
my_numbers2 = [12, 3, 0, 5,7]
function pass2(n2)
   n2 *= 2
   return n2
end
```

When we call as below, we will get the following output.

```
println(pass2(my_numbers2))
println(my_numbers2) #it not changed since new list created by rule of language
```

Output:

```
[24, 6, 0, 10, 14]
[12, 3, 0, 5, 7]
```

KEYWORD AND DEFAULT PARAMETERS

Example 1

In Julia, we can use default parameters. Since the second argument is not provided when calling, the default value will be used.

```
#default parameters
p1 = 6
function def(param1, param2 = 1)
   return param1 + param2
end

println(def(p1))
```

Output:

```
7
```

Example 2*

We cannot provide a non-default argument after a default argument. Functions cannot be defined like this. Thus, it will give an error.

```
p1 = 6
function keyword1(param1 = 1, param2)
   return param1
end

println(keyword1(p1))
```

Output:

```
ERROR: LoadError: syntax: optional positional arguments must occur at end
while loading /home/cg/root/648726/main.jl, in expression starting on line 85
```

Example 3

“A semicolon demarks the start of keyword arguments”.[4] In Julia, keyword parameters should be defined, and they should be defined after any of the positional parameters. In this example, param1 is for positional binding and param2 is for keyword binding. For keyword parameters, we should also give default values when defining the function. In this example default value for keyword parameter is used.

```
#keyword parameters 1
function keyword1(param1;param2 = 1)
   return param1 + param2
end

println(keyword1(12,))
```

Output:

```
13
```

Example 4

In this example default value for both keyword parameters is used.

```
#keyword parameters 2
function keyword2(;param1 = 1, param2 = 1)
   return param1 + param2
end

println(keyword2())
```

Output:

```
2
```

Example 5

In this example, we provide values for both keyword parameters.

```
#keyword parameters 3
function keyword3(;param1 = 1, param2 = 1)
   return param1 + param2
end

println(keyword3(param1 = 4, param2 = 6))
```

Output:

```
10
```

Example 6*

As mentioned before keyword parameters need default values. Otherwise we get an error.

```
#keyword parameters
function keyword1(param1;param2)
   return param1 + param2
end

println(keyword1(5, param2 = 4))
```

Output:

```
ERROR: LoadError: syntax: keyword argument "param2" needs a default value
```

```
while loading /home/cg/root/648726/main.jl, in expression starting on line 94
```

Example 7*

In this example since the semicolon is provided after param1 it is defined as a positional parameter. In Julia, we cannot use keyword binding if it is not defined as a keyword parameter. Otherwise we get an error.

```
#keyword parameters
function keyword1(param1;param2 = 1)
   return param1 + param2
end
println(keyword1(12, 2))
```

Output:

```
ERROR: LoadError: MethodError: no method matching keyword1(::Int64, ::Int64)
Closest candidates are:
  keyword1(::Any; param2) at /home/cg/root/648726/main.jl:95
while loading /home/cg/root/648726/main.jl, in expression starting on line 98
```

CLOSURES

As we can use nested subprograms, we can obviously use closure. Below it is given an example for closures in Julia.

Example

In this example, we can see the effects of closures. After calling the num_func_out function, we can reach its inner functions by getting a reference name to them. Although the num_func_out are erased from the stack, we can still reach its local variables.

```
#--------design issue: closure-----
function num_func_out(n)
   num = n;
   function num_func()
      println(num)
```

```
    end
    return num_func
end
```

If we call the function as below, we will be able to use the inner functions even after the execution os num_func_out is terminated.

```
n1_func = num_func_out(100)
n2_func = num_func_out(2000)
n1_func()
n2_func()
```

Output:

```
100
2000
```

LANGUAGE EVALUATION

Regarding subprograms in Julia in terms of readability and writability, it can be said that the language is good enough for these design issues. It provides flexibility of nested subprograms which can reduce readability, however, since the syntax is very simple I do not think it makes the language less readable than the other languages. Most importantly, when Julia allows keywords, it sets a rule of using semicolons for them. This rule makes the language more readable since when we want to analyze a code we will be able to understand the function usage easily. Since writability is affected by readability, writing code would be also easy and simple. The syntax of the language is simple as a whole, so the syntax for subprograms does not add additional complexity to the languages syntax. Thus, in my opinion, the syntax of Julia for subprograms are readable and writable in a good manner.

LEARNING STRATEGY

For this assignment, I mostly used the official website of Julia. [1] In fact the official Julia website and documentation are good enough for my purpose, so I did not need any further sources. When I needed additional information I got help from the stackoverflow website, and watched youtube tutorials. [4], [5]

COMPILER USED

https://www.tutorialspoint.com/execute_julia_online.php

REFERENCES

[1] Docs.julialang.org. 2020. Julia Documentation · The Julia Language. [online] Available at: <https://docs.julialang.org/en/v1/> [Accessed 22 December 2020].

[2] Docs.julialang.org. 2020. Functions · The Julia Language. [online] Available at: <https://docs.julialang.org/en/v1/manual/functions/#:~:text=In%20Julia%2C%20a%20function%20is,global%20state%20of%20the%20program.> [Accessed 22 December 2020].

[3] Docs.julialang.org. 2020. Scope Of Variables · The Julia Language. [online] Available at: <https://docs.julialang.org/en/v1/manual/variables-and-scoping/#:~:text=The%20scope%20of%20a%20variable,referring%20to%20the%20same%20thing.> [Accessed 22 December 2020].

[4] J., Sarnoff, J., Sarnoff, J. and Sarnoff, J., 2020. Julia: How Are Keyword Arguments Used?. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/58132507/julia-how-are-keyword-arguments-used#:~:text=A%20keyword%20argument%20holds%20the,there%20are%20no%20positional%20arguments.> [Accessed 22 December 2020].

[5] Youtube.com. 2020. [online] Available at: <https://www.youtube.com/watch?v=sE67bP2PnOo> [Accessed 22 December 2020].