CS 315 Homework 3

Subprograms in Julia

Nested Subprogram Definitions

```
function f()
    x = 4
    function g(x)
        x = x * 2
        return x
    end
    b = g(x)
    return b
end

a = f()
println("a: $a")


# here, function g is defined inside of the function f.
# function g takes one parameter and the variable x, which is
# defined in f is passed as that parameter. The value x is returned
from
# function g and assigned to variable b, which is returned from
function f.
# From the output it can be seen that, the function returns 8, which
means
# it takes the value from the inner function g, since there is return
statement
# and a function call with assignment.



function h()
    x = "cs315" # x is local variable and cannot go outer h
    function j(x)
        x = 315 # x is local variable and cannot go outer function j
    end
    println("x: $x")
end

b = h()


# Here, function j is defined inside of the function h.
```

```
# Since x, that is defined j is local and not returned with a call, it
cannot
# go to outside j. That's why, what is printed inside h is the h's
local variable
# x, which is "cs315" string.
```

Output:

```
a: 8
x: cs315
```

Explanation is given above in the code as comment.

Scope of Local Variables

In Julia, functions have hard local scope, which means that if an assignment statement occurs in a function, the program checks whether the left hand side value is already locally created. If there is no created local variable before the assignment, a new local variable is created.

Code:

```
function g()
    var2 = 5
end

println(var2)

"""
this would cause an error since var2 inside the function g is locally
created and has no value outside the function (globally). Therefore,
in global scope there is no such variable var2. The error would tell
us that var2 is not defined.

"""
```

Output:

```
ERROR: LoadError: UndefVarError: var2 not defined
while loading /home/cg/root/5902396/main.jl, in expression starting on line
5
```

Explanation:

In this code segment, a function is created with no parameters. The purpose is to show that a locally created variable inside a function, cannot be seen by the program globally.

Code:

```
function f()
    var = 3 #var is locally created here
end


var = "str" #var is globally created
f() # function is called
println(var)

"""
even if var is the name of both global and local variables and function
is
called AFTER the global declaration,when var is printed, it will print
"str"
since function f has local scope and variables cannot go outside from
the
function.

"""
```

Output:

str

Explanation:

Again, a function is created with not parameters. Differently from the previous code segment, a global variable is also created outside the function. The purpose is to show that even the local

variable has the same name with the global variable, program outside the function sees the globally declared variable and prints, therefore, “str”.

Parameter Passing Methods

Code:

```
function f(n::Integer)
    n = n * 5
    return n
end

a = 4
b = f(a)
println("a: $a")
println("f(a): $b")

"""
This is one of the parameter passings in Julia. Here,
the parameter does not mutate globally when the function is called

"""
```

Output:

```
a: 4
f(a): 20
```

Explanation:

Integers are not mutable when passed as parameters to functions. Here, a function f with one integer parameter n is created. The returned value is n * 5. However, value that is passed to the function does not change globally.

Code:

```
function g(x, y)
    x .+= y
    return x
end

x = [4, 5, 6]
y = 1
println("before the function call x was: $x")
out = g(x, y)
println("after the function call x is: $x")
println("y: $y")
println("out: $out")

"""
This is pass-by-sharing with mutation. Arrays are mutable values by using
functions. Here, an array x and an integer y is passed as parameters to the
function g. x .+= y statement is to increment every element of x by the value
of y. Finally, x is returned. Now, in the global scope, the array x has been
changed due to the function call. The elements of x has been
incremented by 1
```

```
(the value of y) in the global scope even if x was only passed as a
parameter.
This is pass-by-sharing.

"""
```

Output:

```
before the function call x was: [4, 5, 6]
after the function call x is: [5, 6, 7]
y: 1
out: [5, 6, 7]
```

Explanation:

Here, an array, x, is passed as a parameter to a function. Arrays are mutable as parameters. Therefore, elements of x changed globally. I added further explanation to the code as comments.

Code:

```
function h(a)
    a[1] .+= 5
    return a
end

a = [3,3,3]
s = h(a)
println(a)
println(s)

"""
This is another pass-by-sharing with mutation method. An array "a" is
created
and passed to the function h. The first element in array a is
incremented by 5
and the array a is returned. When we print a, and the function call, we
would
see both of them are the same, which means a is changed by the function
call
```

```
"""
```

Output:

```
[8, 3, 3]
[8, 3, 3]
```

Explanation:

Again, an array is passed as a parameter and only one element of the array is changed. Since arrays are mutable with function, the element is also globally changed.

Keyword and Default Parameters

```
function keyfunction(x, y; var1=5, var2=10)
    println("this function does this: ")
    println("( $x + $y ) * $var1 * $var2")
    x = (x + y) * var1 * var2
    return x
end


x = 3
y = 2
zero_default = keyfunction(x, y)
println("zero_default: $zero_default")
println()
one_default = keyfunction(x, y, var2=4)
println("one_default: $one_default")
println()
two_defaults = keyfunction(x, y, var1=2, var2=20)
println("two_defaults: $two_defaults")


"""
there are 2 default parameters in keyfunction (var1 and var2)

in Julia, we can call a function using these default parameters.

If no default parameters is passed, the default values are used.
```

```
We can also pass default parameters with different values than the
default values and function uses the values that we have passed rather
than the default values.
"""
```

Output:

```
this function does this:
( 3 + 2 ) * 5 * 10
zero_default: 250

this function does this:
( 3 + 2 ) * 5 * 4
one_default: 100

this function does this:
( 3 + 2 ) * 2 * 20
two_defaults: 200
```

Closures

```
println("----------------------CLOSURES----------------------")
a = 3
function f(b::Int64)
    println("a is global a: $a")
    println("b is parameter b: $b")
    c = 6
    println("c is local c: $c")
    function closure(d)
        println("************************")
        println("a is global a: $a")
        println("d is parameter d: $d")
        e = 4
        println("e is local e: $e")
        println("b is closed value(outer parameter) b: $b")
        println("c is closed value(outer local) c: $c")
    end
end

param = 5
var = f(param)
var(10)

# closure is defined inside the function f.
# Normally f can get global variable a, parameter b and local variable
c.
# Closure can access to global variable a, parameter d, local variable
e,
# outer local c and outer parameter b.
# it is called as I wrote

# we assign the return value of f to var
# when we pass 10 as parameter to var, the closure occurs
```

Output:

```
a is global a: 3
b is parameter b: 5
c is local c: 6
**********
a is global a: 3
d is parameter d: 10
e is local e: 4
b is closed value(outer parameter) b: 5
c is closed value(outer local) c: 6
```

Explanation is given as comment in the code.

Evaluation: Julia is, in my opinion, kind of a mixed version of Java and Python. I encountered with very different type of syntaxes when dealing with functions. Readability is an issue since variables are dynamically declared (like Python). The scoping of local variables in Julia is, I think, not so good since I prefer more accessible variables among subprograms. Hard local scoping may cause disparicies since a new variable/object/array etc. is created with the same name but in a different address which is not good for readability. For writability, it is an advantage since Julia is flexible. Not declaring types, common usage of syntaxes (operators, function keyword etc.) is an advantage in terms of writability. Using parameters' types when defining functions is a good aspect for readability since it helps third person to understand the variables and parameters more accurately. For writability, it is not a good aspect but it can be argued whether it is necessary or not. Also, pass-by-sharing, I think, is a very unique parameter passing method. However, I can say that it is hard to get used to pass-by-sharing and it would be more appropriate if there were other methods such as pass-by-value or pass-by-reference as well. Consequently, in terms of readability and writability, Julia is similar to Python with little differences. I would say it is a more writable language than it is a readable language.

Strategy: I divided every operation and I went one by one. I read the documentation of Julia and I watched a tutorial about Julia language. I, also, looked at some discussions about Julia on StackOverflow. References are given below.

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

Youtube Tutorial: https://www.youtube.com/watch?v=sE67bP2PnOo&t=2047s

Documentation: https://julialang.org

Discussions:
https://stackoverflow.com/questions/58150295/how-to-pass-an-object-by-reference-and-value-in-julia#:~:text=Julia%20function%20arguments%20follow%20a,identical%20to%20the%20passed%20values.

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