ICSI 311 Assignment 10 – Interpreter Part 4

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

We are almost done! We have two minor pieces left to do (and, of course, a lot of testing).

The first thing we need to do is bridge the gap from ProgramNode → InterpretListOfStatement.

The second thing that we need to do is implement function calls. That's the more interesting part, so let's take a look at that in depth.

Consider this function:

```
doSomething(int a, int c)
```

Now, if we call it:

```
int c=6
doSomething(c, 2+2)
```

We should notice a couple of things – the caller's variable names may be different or not even exist (what is the name of 2+2?). Function calls happen positionally – the first parameter passed (c in our case) is mapped to the first parameter variable (a). The second parameter (2+2) is mapped to the second parameter variable (c).

We have a very convenient place to do this mapping – the locals variable hashmap. The algorithm looks like this for non-variadic:

```
RunFunctionCall(locals, functionCallNode)
        func = find the function definition
        if (func.parameterCount != functionCallNode.parameterCount) throw error
        map = new hashmap<string, IDT>
        for each parameter in func
                map[parameter.name] = getIDT(functionCallNode.parameters[i++])
        if (func is builtin)
```

```
            func.execute(map)
        else
            InterpretListOfStatements(func, map)
```

Variadic are a little different (but only a little). We follow the same mapping process for all but the last variable. For the last variable, we make an array and put all of the remaining values from the functionCallNode into that array. All variadic functions are built-in in our implementation, so we can then directly call Execute.

Details

Create a method called “InterpretProgram”. It should run each of the “BEGIN” blocks (calling a new method called “InterpretBlock” for each one). It should then call LineManager’s SplitAndAssign, and for every record, call InterpretBlock() on every one of the blocks that are not BEGIN or END. Outside of that loop, it should call InterpretBlock() on each of the END blocks.

Create a method InterpretBlock, which takes a BlockNode as a parameter. If the block has a condition, we will test it to see if it is true. If there is no condition OR the test is true, then for each statement, we will call ProcessStatement.

Replace our non-functional RunFunctionCall with the implementation discussed in the introduction. Make sure to throw exceptions for errors (function doesn’t exist, parameter counts don’t match up on non-variadics, etc).

The implementation isn’t very large here, but we didn’t do testing last time, so expect more bugs than usual. For testing, you can supply a whole AWK file and a whole text file. You should be able to see your program run from beginning to end.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
InterpretProgram	None (0)			Runs start and end blocks, calls SplitAndAssign, runs other blocks for each line of input (10)

InterpretBlock	None (0)			Evaluates condition, calls InterpretListOfStatements() if condition is true (10)
RunFunctionCall – non-variadic	None (0)	One of: Checks parameter count, maps parameters into new hash map, calls Execute or InterpretListOfStatements (3)	Two of: Checks parameter count, maps parameters into new hash map, calls Execute or InterpretListOfStatements (6)	Checks parameter count, maps parameters into new hash map, calls Execute or InterpretListOfStatements (10)
RunFunctionCall – variadic	None (0)	One of: Maps non-variadic parameters, makes array for remaining parameters, calls Execute (3)	Two of: Maps non-variadic parameters, makes array for remaining parameters, calls Execute (6)	Maps non-variadic parameters, makes array for remaining parameters, calls Execute (10)
Unit Tests Conditional blocks	None (0)			Shown to work (10)
Unit tests – user created function w/ calls	None (0)			Shown to work (10)
Unit tests – input processing	None (0)			Shown to work (10)
Unit tests – math and logic	None (0)			Shown to work (10)
Unit tests - loops	None (0)			Shown to work (10)