ICSI 311 Assignment 8 – Build on the Interpreter

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

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

The idea behind an interpreter is the same as a "to do list" –

1) Find the next thing to do
2) Do it
3) Are we done? If not, goto 1.

For this assignment, we will focus on a few statement types and implement "do it". There is a sub-section of "do it" that we need to think about, though. Consider:

```
a = a+2
PRINT "Answer:", a
```

In the assignment, we have to evaluate the MathOpNode, then set a variable. By hand, we could do this:

```
Integer a = myIntegerVariables.get("a");
a = a + 2;
myIntegerVariables.put("a",a);
System.out.print("Answer:", myIntegerVariables.get("a"));
```

A few things about that might make you pause:

Why did we have to put the value back into the hashmap? Because that's the definition of assignment! The next reference to "a" must be the new value.

Why did we have to get from the hashmap when we already had a local Java variable? Because that will be "forgotten" when we leave the assignment statement; it will be a local variable.

How do we make this? Well, one step at a time!

Let's consider "evaluate()" – a function that returns a value given a Node; this is the right side of our assignment statement, among other things. The implementation pattern is to consider the possible data types and use "instanceof".

Evaluate() could get a variable (x=y), a number (x=4) or a math expression (x=2+2). But notice that the math expression is recursive – either side of the "+" could be another expression (x = (1+1) + (1 + 1)). Fortunately, since we already dealt with order of operations in the parser, this is a pretty straightforward situation; it's just recursive:

If (node is variable) return lookupVariable()

If (node is number) return number

If (node is mathOP) return expression(left) <<<OP>>> expression(right) { where OP is +, -, *, / }

The only case missing is function calls. We could have a=3+RANDOM()

That's just another case – we loop over the parameters to the function, calling evaluate() on them. Then we call the function and return its return value.

One more "tricky bit" – int vs. float. The easiest way to go about this is to make 2 different versions of evaluate (evaluateInt and evaluateFloat). Be careful – if you copy/paste this code, and you fix a bug in one, you have to fix the same bug in the other 99% of the time. I would debug integer completely, then implement float.

Most of the time, we know which evaluate() to call – we can tell by the return type we need (a=2+2 as compared to a% = a% + 2). There are two cases where we can't – PRINT and NUM$. The short answer is that you have to be prepared to try both.

Details

Create the two evaluate calls discussed above.

Create an "void Interpret(StatementNode)" method. We won't do all of them, but some:

ReadNode – for each variable, get the InterpreterDataType from the map (create it if necessary), get the next value from the collection of data values. if the types don't match, throw an exception. Set the InterpreterDataType's value to the data value.

AssignNode – process the expression. Lookup/create the variable and set its value in the map.

InputNode – use Java to print the string. Use Java appropriate input processing for input. Assign the value as in AssignNode.

PrintNode – for each child of Print, use Java to print the values. Don't forget to Evaluate() if necessary.

Testing

One thing that will be helpful here is a way to do testing on INPUT and PRINT that supports testing. Add a new mode to "INPUT" and "PRINT" – a testing mode. If in test mode, INPUT takes strings from a List<String>. If in test mode, PRINT outputs strings to a List<String> (you may find String.format() useful). With this, you can test INPUT and PRINT in an automated way.

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)
Unit Tests	Don't exist (0)	At least one (3)	Missing tests (6)	All functionality tested (20)
Interpret()	None (0)			Determines the type and executes the appropriate functionality (5)
Evaluate	None (0)			Determines the type and executes the appropriate functionality (5)
Evaluate Numbers	None (0)		Returns in EITHER integer OR float variants (2)	Returns in both integer and float variants (5)
Evaluate Variables	None (0)		Returns in EITHER integer OR float variants (2)	Returns in both integer and float variants (5)
Evaluate MathOP	None (0)		Returns in EITHER integer OR float variants (4)	Returns in both integer and float variants (10)
Evaluate Function Calls	None (0)		Returns in EITHER integer OR float variants (4)	Returns in both integer and float variants (10)
Read	None (0)			Reads and removes from internal collection. Updates variable(s) (5)
Assign	None (0)			Calls appropriate evaluate and sets the value (5)
Input	None (0)			Prints the prompt. Reads data and sets the variable(s). (5)
Print	None (0)			Prints each data item (5)
Input "test mode"	None (0)			Can be turned on and off. Reads from a collection instead of "terminal" (5)
Print "test mode"	None (0)			Can be turned on and off. Writes to a collection instead of "terminal" (5)