ICSI 311 Assignment 7 – Start 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

Let's talk about interpreters. Remember that our AST is **read only** while our BASIC code is interpreting. It is a definition of our program in memory. But, of course, we know in computer science, our program needs two things to run: storage and input/output. The AST is the program; to build the interpreter, we will build facilities for storage of variables, input/output functionality and finally some code that will look at each Node and "execute it".

First, though, let's think about efficiency a little. Consider a program:

```
READ count
FOR I = 0 to count
      READ F%
      GOSUB Convert
NEXT I
END
DATA 10, 22.3, 33.1, 55.2, 44.2, 17.8, 66.2, 47.1, 33.2, 42.9, 17.2
Convert:
      C% = 5*(F%-32)/9
      PRINT F%, "DEG F = ",C%, "DEG C"
      RETURN
```

When we encounter that first "READ", how can we get that value? We have to walk through the entire program looking for a "DATA". Once, that wouldn't be bad, but not look down to "READ F". We (again) have to walk the entire program to find the "DATA". **And** we have to keep track of what has been read already. That's going to be really slow. What if, before we started running, we did that "walk" once, and pulled all of the DATA items into a Java LinkedList/Queue? Once we do that, READ becomes "pop". Fast and easy!

Now, let's look at the GOSUB. In theory, the subroutine could be above or below the call. So (similar to READ), we would have to start at the top and read all the way through the AST to find it! That's slow! So

we will modify our AST – we will look through the AST before we start running, find all of the labels and make a HashMap<String,LabeledStatementNode>. This will let us run our code quickly.

We will need to have storage for our variables. In other languages, we must be very flexible. BASIC lets us take a couple of shortcuts because there are only 3 data types (int, float, string) and every variable is global. That means no variables ever disappear (go out of scope). We don't have to track what is in scope and what is not; once a variable is created, it is "forever". We will turn, again, to HashMap. We will make 3 – one for each variable type.

The last bit of "prep work" is to implement the built-in functions. There are 8 of them (2 versions of NUM$). Implement these as static methods on your interpreter, using Java data types (for example:

public static String left(String data, int characters) { /* implement me */ }

Details

Create a new class (Interpreter.java).

Since it is possible (and, in fact, good form) to put the DATA at the end of your program, we need to pre-process these. Walk the AST, searching for the DATA statements. Insert their contents into a Java collection that we can use for READ.

We also need to know where the labels are. Create a hash map string-> LabeledStatementNode. Use a visitor pattern to walk the statementsnode searching for labeledstatements.

We will need storage for our variables. Create HashMaps that map name (string) to data type for our variables. We will need three hash maps: String->Integer, String->Float and String->String.

Finally, implement all of the built-in functions.

Testing is (of course) still important. You can test the data optimization and the label hashmap as well as the built-in functions.

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 (15)
Variable Storage	None (0)			Correct (5)
Label optimization	None (0)		Significantly Attempted (10)	Correct (15)
Read optimization	None(0)		Significantly Attempted (10)	Correct (15)
RANDOM()	None (0)			Correct (5)
LEFT$	None (0)			Correct (5)
RIGHT$	None (0)			Correct (5)
MID$	None (0)			Correct (5)
NUM$	None (0)		Works for int OR float (5)	Correct for int and float (10)
VAL	None (0)			Correct (5)
VAL%	None (0)			Correct (5)