ICSI 311 Assignment 9 – Finish the Interpreter

This is the last assignment in BASIC – no more dependencies!

You must submit .java files. Any other file type will be ignored. <u>Especially</u> ".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.

<u>You must submit buildable .java files for credit.</u>

Introduction

In our last assignment, we interpreted most of the AST nodes. We have a few more to do and we need to tie it altogether to process a whole program.

We have a little bit of a problem with our design. There is no good way to know what the next instruction should be. We could loop over our StatementsNode, but when we have flow control (if, for, while, gosub), we need to go out of order. Consider:

```
FOR A = 0 TO 10 STEP 2
PRINT A
NEXT A
```

As an AST: StatementsList(ForNode, PrintNode, NextNode). We start interpreting the list of statements list. We get to the for node, we set A, then we print A, then we hit NEXT. How do we loop? We need to go back!

How could we build this? When we encounter the "FOR", we push that StatementNode on a stack. That's easy enough. Then we keep looping over our StatementsList. When we get to the NEXT, we pop the FOR node and make that "current". But … there is no connection between FOR and PRINT except that they are next to each other in the StatementsList.

We could search the StatementsList (indexOf() or build it ourselves) for the FOR. But that will be very slow in a bigger program.

What we really want is to be free from the StatementsList – every statement should know the next one in line. Then you could jump around and still know how to get to the next one!

Add a "Next" member to StatementNode, of type StatementNode. This will indicate the statement to run after this statement.

Finally, a visitor pattern needs to be run over the StatementsNode – set the “Next” element for each to the next item in the StatementsNode’s statements member. For example, if you have a program:

```
A=4
PRINT A
```

The assignment node’s next should point to the print node. The print node’s next should be null. Now we have a linked list of StatementNode where we can feel free to jump around!

The last thing we need to do is set up our loop, so we can run our program! Our logic is this:

```
while (not END)
    next = currentStatement.next  (this is our default case)
    do the current statement.
    currentStatement = next (next might have changed in the middle)
```

Details

Add the Next member to the StatementNode in the AST and write the code to build the linked list; put this code in a method by itself.

Add a few members to interpret – a loop Boolean, currentStatement (a StatementNode) and a stack of StatementNodes. The stack, as described above, will be used for cases where we have to return to a previous location in the code (like RETURN, NEXT, etc.).

Finish the interpret() method:
IfNode – Process the Boolean expression. This is actually very easy – call evaluate() on both sides. If true, look up the label in the label hashmap and set currentStatement to be that label. This is a one-way trip and doesn’t use the stack.

GosubNode – Push the gosub’s next node onto the stack. Set the next to the label lookup (from the map) for the gosub’s label.

ReturnNode – Pop a value from the stack. Set the next to be that StatementNode.

ForNode – Create/get the variable from the int variable map. If you created it, set it to the initialization. If not, add the step to the variable (default is 1). If the variable is past the limit, skip over the instructions until we find the matching NEXT. Otherwise, push the FOR statement on the stack.

NextNode – Pop from the stack and set next to be StatementNode (same as return).

EndNode – set the boolean to end the run loop.

Implement the loop as discussed above; when complete, we should run a whole program; we just call interpret() with the first node from the statementList. Change main() to call the interpreter as well as the lexer and parser.

Testing

Remember way back in assignment 0, we wrote some BASIC programs? You should now be able

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)
Next member and visitor	Don’t exist (0)			Next exists and is poplated by the visitor (10)
END	Not handled(0)			Handled correctly (5)
IF	Not handled(0)			Boolean expression processed and “next” conditionally updated (10)
GOSUB	Not handled(0)			Handled correctly (5)
RETURN	Not handled(0)			Handled correctly (5)
FOR	Not handled(0)			Initializes/updates variable, increments, skips past CORRECT “next” at the end (15)
NEXT	Not handled(0)			Handled correctly (5)
Interpreter Loop	Not handled(0)			Handled correctly (10)

to run those to completion and see results! Make sure that you automate that process so that we can tell if any additional (hypothetical) changes to our interpreter break anything.