* Due: Friday, November 1st, 11:59pm
* Directory name for this homework (case sensitive): hw4
  + This directory should be in your hw\_username repository
  + This directory needs its own README.md file
  + You should provide a Makefile to compile the code.
* Warning: Start early!

**Overview**

For this (second) part of the project, we will build an interpreter for the subset of BASIC we parsed on HW2, and analyze the running time for some key operations that you need. If you built your parser and class structure very carefully, this part of the project may be comparatively painless. If not, this is your chance to improve and redevelop your class hierarchies.

This part of the project is where you will need to make a lot of good decisions about which data structures to use for different parts of your implementation.

**The semantics of BASIC commands**

In HW2, you learned the **syntax** of BASIC (or at least our simplified version of it), i.e., what is and isn't a correct command that you need to parse. We had alluded to, but not really described in detail, what the commands actually do. We will do so now, because you now need to run programs, not just print them.

**Variables**

1. integer variables: each variable stores a (signed) 32-bit integer. If a variable is read without having been initialized, its value is 0.
2. one-dimensional arrays of integers: We imagine the arrays to have "infinite" size, running from negative infinity to positive infinity. (For practical purposes, they have size equal to the range of a (signed) integer.) All (infinitely many) locations are 0 unless overwritten otherwise. Because the arrays are infinite, there can never be segfaults. If a user tries to read a location they have not written before, the value is 0.

**Expressions**

When evaluating an expression, these are the rules to apply.

* a constant: that value is always its numerical value.
* an integer variable name <VAR>: the value is the value currently stored in that variable.
* an array evaluated at an integer position: the value is the array's value at the position, as described in the rules above.
* (<AEXPR1> + <AEXPR2>): the value is the sum of the values of <AEXPR1> and <AEXPR2>.
* (<AEXPR1> - <AEXPR2>): the value is the value of <AEXPR1> minus the value of <AEXPR2>.
* (<AEXPR1> \* <AEXPR2>): the value is the value of <AEXPR1> multiplied with the value of <AEXPR2>.
* (<AEXPR1> / <AEXPR2>): the value is the value of <AEXPR1> divided by the value of <AEXPR2>, and rounded down to the nearest integer. (That is, the standard integer division.) If the value of <AEXPR2> is 0, then this is a division-by-0 error. The program should output an error message, including the line number where the error happened, the strings corresponding to <AEXPR1> and <AEXPR2>, and the values when the error happened. Then, the program should terminate. See below for exact format for error messages.

For all arithmetic expressions, we guarantee that the calculations will always result in integers that fit into a (signed) 32-bit integer, so you do not have to worry about overflows.

* <AEXPR1> = <AEXPR2>: evaluates to true if <AEXPR1> and <AEXPR2> evaluate to the same number.
* <AEXPR1> < <AEXPR2>: evaluates to true if <AEXPR1> evaluates to a (strictly) smaller number than <AEXPR2>.
* <AEXPR1> > <AEXPR2>: evaluates to true if <AEXPR1> evaluates to a (strictly) larger number than <AEXPR2>.

**Commands**

We next provide some more detail on the commands.

* PRINT <AEXPR>: prints the value of the arithmetic expression, evaluated according to the rules above. Always prints a newline after the value.
* LET <VAR> <AEXPR>: evaluates <AEXPR> and stores the resulting value into the variable <VAR>, overwriting the previous value if there was one.
* LET <VAR> [<AEXPR1>] <AEXPR2>: evaluates <AEXPR2> and stores the resulting value into the array <VAR>, at the position obtained from evaluating <AEXPR1>.
* GOTO <LINEJ>: Jump to the line <LINEJ>. Forgets which line it came from. Just resumes processing at the given line <LINEJ>. If there is no line numbered <LINEJ> in the program, it should print an error and terminate. See error messages below.
* IF <BEXPR> THEN <LINEJ>: evaluates <BEXPR> according to the rules given above. If it comes out true, then jumps to the line <LINEJ> (as in the case of GOTO). Otherwise just advances to the next line. As for GOTO, if the target line does not exist, it should output an error. (See error messages below.) If the condition comes out false (so there's no jump), and the target line does not exist, then no error is reported, as the program can safely continue running.
* GOSUB <LINEJ>: Jump to the line <LINEJ>, and remember where you came from. We will give a more detailed example of GOSUB/RETURN below. As for GOTO/IF, if the line <LINEJ> does not exist, then an error is printed (including the current line number and non-existing target line), and the program terminates. See below for details.
* RETURN: Go back to the line immediately after the line whence the most recent GOSUB jump started. See the more detailed example below. If there has been no applicable GOSUB jump, then an error message is printed; see below for the exact message and format.
* END: Terminate the execution of the program. Nothing can go wrong here. :)

**Some simple examples of BASIC programs**

Here are some simple examples of BASIC programs, and what they would output, so that you can familiarize yourself with the meanings of commands.

5 LET HAN 42

10 LET SOLO -9

20 PRINT HAN

25 PRINT SOLO

You will need to be able to parse programs where line numbers are no longer consecutive. We only guarantee they will be positive, and in strictly increasing order (such as the above example), with execution starting at the lowest-numbered line. The above program would output:

42

-9

1 LET YODA 1

2 GOTO 4

3 LET YODA 2

4 PRINT YODA

This program would output 1, because by skipping line 3 (with the command GOTO 4), we prevented YODA from being assigned the value 2.

1 LET CHEWY 3

2 LET VADER 6

3 IF CHEWY < 4 THEN 5

4 PRINT CHEWY

5 PRINT VADER

This will print out only 6, since line 4 is skipped over.

1 LET ANAKIN 1

2 GOSUB 6

3 PRINT ANAKIN

4 PRINT PADME

5 END

6 LET ANAKIN 2

7 LET PADME 3

8 RETURN

In the above program, ANAKIN will get set to 1, then the program will jump to line 6, overwriting ANAKIN with 2, setting PADME to 3, and then returning to the point it jumped from. It will then output ANAKIN, then PADME, then terminate. So the output will be

2

3

**A detailed example of GOSUB and RETURN**

To understand GOSUB and RETURN a bit better, consider the following piece of code:

10 GOSUB 50

20 PRINT 20

30 RETURN

40 END

50 GOSUB 80

60 PRINT 60

70 RETURN

80 LET N 0

90 IF N > 1 THEN 130

100 PRINT N

110 LET N (N+1)

120 GOSUB 90

130 RETURN

Let's see what happens here.

1. In line 10, the program jumps to line 50, remembering that it came from 10.
2. From there, it jumps to 80, remembering that it came from 50.
3. In 80, it initializes N to 0.
4. Because N is not larger than 1, in line 90, it does not jump to 130, and instead prints N (which is 0) in line 100.
5. It increments N in line 110, and then jumps to line 90, remembering that it came from line 120.
6. Because N is still not greater than 1, it still does not jump, instead printing N (which is now 1) in line 100.
7. It increments N to 2, and jumps from line 120 to line 90, remembering that it came from line 120.
8. Now, in line 90, N **is** greater than 1, so it jumps to line 130.
9. This is the first RETURN statement, and the most recent GOSUB was in step 7 of this description. It originated from line 120, so that's where the program returns (more precisely, to the next line, which is 130).
10. That's another RETURN statement, and now, the most recent GOSUB that **wasn't** already accounted for is the one in step 5 of this description. It originated from line 120, so the program returns to line 130.
11. That's **yet** another RETURN statement. At this point, the most recent unaccounted-for GOSUB is the one from step 2 of this description. It originated from line 50, so the RETURN statement jumps to the next line after 50, which is line 60.
12. Here, it prints the number 60.
13. Line 70 is a RETURN statement. Now, the next unaccounted-for GOSUB is the one from step 1, which originated from line 10. So the program next returns to line 20.
14. Here, it prints the number 20.
15. In line 30, we have another RETURN, but there are no more unmatched GOSUB statements left. At this point, an error occurs.

All in all, this would be the output:

0

1

60

20

Error in Line 30: No matching GOSUB for RETURN.

**Error messages**

Whenever an error happens, the program prints an error message and terminates. Each error message begins with "Error in Line <LINE>: ", where <LINE> is the number of the line where the error occurred. After that, it prints an error-specific message, which will be one of the following (which you should match **exactly** if you want to earn points from our automated tests):

* "Division by 0: <AEXP1> = <VAL1>, <AEXP2> = <VAL2>." Here, <AEXP1> and <AEXP2> are the pretty-printed text strings for the expressions, and <VAL1> and <VAL2> are the current value. (Presumably, when an error happened, <VAL2> has a value of 0.)
* "GOTO to non-existent line <LINEJ>." Here, <LINEJ> is the line that you were supposed to jump to, but which didn't exist.
* "IF jump to non-existent line <LINEJ>." Again, <LINEJ> is the line that you were supposed to jump to, but which didn't exist.
* "GOSUB to non-existent line <LINEJ>." Here, <LINEJ> is the line that you were supposed to jump to with GOSUB, but which didn't exist.
* "No matching GOSUB for RETURN."

In all cases, after the error message is printed, the program terminates.

**What your program needs to do (85%)**

You will use your parser and pointer-based representation of a program to build an interpreter for BASIC. When we type make or make hw4, it should produce an executable named hw4 in your hw4 directory.

As before, your program hw4 must take a single command-line argument: the name of the file that will be parsed and interpreted. If that file does not exist or cannot be opened, your program should print the error "File <FILENAME> cannot be opened." and exit. Otherwise, it should parse the file into your parse tree structure (see HW2). It should then interpret/run it according to the rules above. When the run terminates, your program should also exit. All output (pretty-printing, output from PRINT statements, and error messages) should go to cout.

**Thoughts on Implementation**

You will most likely expand the functionality of the classes you designed on HW2 so that expressions (arithmetic and Boolean) can now also be evaluated. Almost certainly, to do so, you will add more functions and/or variables to them. Perhaps the most natural way to do this to add to your ArithmeticExpression base class a function

virtual int getValue () const = 0;

and to your BooleanExpression base class a function

virtual bool getValue() const = 0;

However, if your BooleanExpression inherits from ArithmeticExpression, then this implementation will cause problems, so you may need to do something slightly different. The implementations of these functions for all subclasses should be fairly straightforward; they will obviously be recursive. The "only" interesting cases are likely going to be the getValue() functions for Variables and ArrayVariables. Here, you need to figure out what data structure to use to be able to realize that all occurrences of X refer to the same variable. See below for a bit more on this.

Next, you will need to think about data structures. Here are some of the more central data structures and design thoughts you should consider:

**Arrays**

How should you store arrays? They are very big (infinitely big?). Do you need to store all positions? Is there a better way?

**Variables**

What data structure will quickly let you find a variable (array or non-array) given the variable name? You may already have had to implement this for HW2, but we understand that many students created new Variable objects for each occurrence of even the same variable, say, X. Now, you need to make sure that you somehow connect all those occurrences, since otherwise, when you change X in one line, it will not affect other lines.

**GOSUB/RETURN**

How will you keep track of all those nested GOSUB calls, and know where to return to next? How will you handle errors relating to GOSUB/RETURN?

**Error handling**

Some parts of your program can cause errors (such as division by 0, or a non-existent line number). Can they handle the errors themselves? If not, who handles them? How do you pass around the responsibility for dealing with errors?

**Overall**

Who keeps track of variables, arrays, GOSUB destinations, program code, etc.? Should they all be variables in main()? Would you have a class Interpreter? Might you have additional classes, and if so, how would they interact with Interpreter?

**Analyzing your Code (15%)**

In writing your interpreter, you made a number of data structure choices, in particular regarding how you keep track of the values of variables and array entries, and how you figure out where to jump (GOTO, GOSUB, IF) and where to return (RETURN). As the second part of this assignment, you should analyze how long these key operations take your interpreter to execute.

Notice that there are now a number of different quantities that may affect how long things take, such as the number of lines of code, the number of different variables, the size of arrays (what is the right measure of the size anyway, given that arrays are infinite in size?), etc. In your analysis, don't just write O(n) or Ω(log n) or something like that. Make sure that you tell us exactly what all the variables are, such as "n is the number of lines of code, and m is the number of variables, and ..." You can pick the notation, but make sure to tell us. Ideally, your writeup should start with a complete description/definitions of all the variables you use, so that your grader can easily follow along.

Also, your grader will not want to read 500 lines of code just to figure out how long one particular piece takes. So while you should point to the code (e.g., "this part is implemented in lines 63-98 of my file interpreter.cpp"), your answer here should also include a high-level description of your approach. Such as for example:

* My high-level approach is for each of The Rani's experiments to be a set. You can see in Lines 63-70 where I initialize the set. The important part of the ADD operation is to create a new set, which is done in lines 75-79, and takes constant time. When I move n subjects from one experiment to another, say from i to j, my implementation (in lines 90-110) runs an iterator over the set i. So even though sets are efficient, the iterator will take time Θ(m(i)), which - as I defined above - is the number of subjects in experiment i at the time. And so on ...

Since your runtime analysis may depend on STL data structures, here are some runtimes you might want to refer to:

Iterators

* iterating from start() to end() using operator++ takes Θ(n), regardless of data structure
* start() and operator++ for maps and sets take Θ(log n), while end() takes Θ(1).

Stacks/Queues/Lists

* assume these are implemented efficiently with a linked list in your analysis

Maps/Sets

* operator[], insert, erase, and find all take Θ(log n)

If you are unsure about a certain function, feel free to ask on Piazza!

**What you need to do**

1. Analyze the running time for figuring out which line a GOTO, GOSUB, IF jumps to. (That is, going from having the line number to actually knowing what command is on that line.) You can analyze them together if they take the same amount of time; otherwise, explain and analyze the differences. For the analysis of IF, you can exclude the part of figuring out if the Boolean Expression is true.
2. Analyze the running time for figuring out the command that a RETURN jumps to. That is, how long does it take to go from seeing RETURN until you know the command to go to.
3. Analyze the running time for figuring out the value of a variable or overwriting the value of a variable.
4. Analyze the running time for figuring out the value of an array entry, assuming you have already figured out the numerical value of the index. (That is, you have already calculated <AEXP> in VAR [<AEXP>].)

Your analysis should go in a file hw4analysis.txt in your main directory. If you choose to use a different way (e.g., you produce JPG by scanning/photographing your solution or produce a PDF from a Word or LaTeX document), your hw4analysis.txt file should contain clear instructions on where and how to read your actual analysis.

Your high-level description and analysis should mostly match your code. If there are small mistakes in your code that make it not actually work like you thought it did, we will consider the analysis of your high-level idea the correct one. If there is massive discrepancy, there may be deductions.

If your code is slightly inefficient in solving a problem (e.g., you spend time Θ(log n) to do something that could be done in O(1)), there will usually only be small deductions. If your implementation is highly inefficient (e.g., you spend time Θ(n) for something that should be done in O(log n) or O(1)), you may lose a significant number of points.

Your analysis should be as tight as possible, but obviously still correct. For instance, if your code runs in time O(1), and you say it takes O(log n), you will likely get decent partial credit. If you say it takes O(n), that will be worth much less credit. On the other hand, if your code takes Ω(n), and you say it takes O(1), then you will probably get close to nothing, because this would just be false.

**Chocolate Problem (2 Chocolate Bars)**

A more limited "Programming Language" than C++, Java, etc., can be obtained using just a single Stack for all data storage. This "computer" has a finite set of "states" (numbered from 0 to S). It starts in state 0, and at any given time, it's in one of those states. It reads a string one character at a time. In addition, it is allowed to push characters on the stack, pop characters off the stack, and read the top element of the stack. In each step, based on the state, the current character, and the character on top of the stack, it decides (1) whether to pop a character off the stack or not, (2) whether to push a character onto the stack or not (and if so, which one), and (3) which new state to enter. Two of the states (let's say S-1 and S) are special, in that when you reach S-1, you output "Yes", and when you reach S, you output "No". (Otherwise, nothing is ever output.)

Here would be an example program:

(0, 'a', 'b') -> (-, 'c', 2)

(1, 'b', 'b') -> (+, 'a', 1)

(1, 'b', -) -> (-, 'b', 3)

The first line says that if the computer is in state 0, reads an 'a' from input, and sees a 'b' on top of the stack, then it should not pop from the stack, push a 'c' onto the stack, and enter state 2. The second line says that if the computer is in state 1, reads a 'b' from input and also sees a 'b' on top of the stack, it should pop from the stack, then push an 'a' onto the stack, and reenter state 1. The final line says that if the computer is in state 1, reads a 'b' from input, and the stack is empty, then it should not pop from the stack (good idea!), push a 'b' on the stack, and enter state 3. This type of computer is not as powerful as, say, a C++ compiler. But it can do some interesting things.

**(a)**

Write a simulator for this type of program. Feel free to alter the format from the one given above, so long as what you design is readable and your program can parse it. You should output what happens to your program on a given input string when the computer starts in state 0.

**(b)**

As a warmup, write a program in this language that will output "Yes" exactly when the string is correctly parenthesized (see early recursive class definitions). So the only legal characters will be '(', '[', ')', ']', and you need to check that all parentheses match.

**(c)**

Write a program in this language that will output "Yes" exactly when the input string is a correct arithmetic expression. The legal symbols are variable names made up only of the letters 'a'-'z' (let's say only lowercase), digits '0'-'9' forming integer numbers, and the symbols '+', '-', '\*', '/', '(', and ')'. No other symbols will occur. As some examples, your program should output "Yes" for the following:

(hello+world+531)\*(the-answer-is+42)

(a\*(b+(c\*(d+(e\*f)))))

a/b/c/d/e

and "No" for

(hello2+world1)

)a+b(

(a+(b+c)