**Homework 2**

* Assigned: September 14, 2019 PST
* Due: September 27, 2019 at 23:59 PST
* Directory name in your github repository for this homework (case sensitive): hw2
  + Once you have cloned your https://github.com/usc-csci104-fall2019/hw-username repo, create this hw2 folder underneath it (i.e. hw-username/hw2).
  + Skeleton code for this assignment is available in homework\_resources/hw2.

Warning: this assignment is the hardest yet. Start early!

**1. Basic Interpreter (100%)**

When you compile your C++ code, the compiler turns your code into a bunch of 0s and 1s (machine code). There is another way this could be done however: an *interpreter* doesn't compile the code, but rather reads through your program line by line, interpreting what needs to be done, and doing it. It keeps track of the *program state* as it runs (that is, the values of variables, what line it is on, etc). Implementing an effective interpreter for C++ would take years.

For your class project this semester, you will be writing an interpreter for a very simple programming language (much simpler than C++). In the current part of the project, you will write the parser, which will lay the foundation for the interpreter that you write in homework 4. This will primarily consist of setting up a class inheritance hierarchy, and building something called a parse tree

**The Language**

The language you will be working with is a very limited subset of the well-known programming language BASIC. BASIC was widely used a few decades ago, and is considered easy to learn.

Like any other language, BASIC has variables, expressions, and commands. Each line is numbered, in increasing order. For this part of the project, you can assume the line numbers are 1 through n, in consecutive order. In the second part of the project, however, you will only know that the line numbers are in increasing order (this fact may influence your design decisions).

One of the main control flow commands in BASIC is GOTO, which jumps to a given line number. While you will need to handle GOTO statements in BASIC, don't try to use them in C++: this is considered very bad style. The GOTO statement produces code which is very difficult to understand, which is why most languages nowadays strongly discourage it, or not make it available at all. See the article [Go To Statement Considered Harmful](https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf), one of the most influential computer science articles of all time, and also check out the following [XKCD comic](https://xkcd.com/292/).

In this part of the project, you will only be worried about syntax. That is, you will need to parse a line of code such as:

4 IF XYZ > 5 THEN 22

and store it in a parse tree (which we will describe later). You don't need to worry (yet) about what this line of code actually means. That is, you do not need to worry about the semantics. For now, you just need to parse and print programs. We'll give a few basic explanations about the meanings of commands now, but details will follow later in homework 4.

**Variables**

We will only consider two types of variables:

1. Integer variables: the names of integer variables will be strings of all uppercase letters - no other characters allowed. The strings will have length at most 8. No variable can have a name that is a command in the language.
2. One-dimensional arrays of integers: exactly the same rules apply as for integer variables. An array is accessed by using the [] operator. You can have an array variable of the same name as an integer variable; your program should be able to tell them apart by whether there are [] after the name to access an element.

**Expressions**

In order to assign values into variables, or compute array indices, or print values, we will frequently need numeric expressions, or expressions that evaluate to numbers. For IF statements, we will need to evaluate Boolean expressions. Here is a recursive definition of these two constructs.

A numeric expression is one of the following:

* A constant: this is simply an integer number, like 42, written out normally.
* An integer variable name <VAR>: see the variables section
* An array evaluated at an integer position: this will be of the form <VAR>[<NEXP>]. Here, <VAR> denotes any legal variable name, and <NEXP> is any legal numeric expression.
* One of the following four arithmetic numeric expressions:

(<NEXP1> + <NEXP2>)

(<NEXP1> - <NEXP2>)

(<NEXP1> \* <NEXP2>)

(<NEXP1> / <NEXP2>)

In all of these cases, <NEXP1> and <NEXP2> denote legal numeric expressions themselves. We are putting a lot of extra parentheses in here to make your parsing task a lot easier; with these requirements, operator precedence will not be a problem. It means that some expressions (such as 5 \* (3 + x)) that should really be legal are not, and you don't have to worry about them. Some things that are legal numeric expressions would be the following:

42

X

Y[4]

(3+X)

(5\*(3+X))

Y[(3+X)]

A Boolean expression is one of the following. Notice that equality is compared with only a single =, which is different than for C++.

<NEXP1> = <NEXP2>

<NEXP1> > <NEXP2>

<NEXP1> < <NEXP2>

Here, <NEXP1> and <NEXP2> are legal numeric expressions. Notice that this is very restricted, to keep your parsing simple. You do not have to worry about logical and, or, or not. You also do not have to worry about >= or <=.

**Commands**

BASIC only puts one command per line, so you do not need semicolons between commands. All commands are in upper-case letters. In all of the following, <LINE> is a positive integer line number. All our programs for this homework will number lines in sequential increasing order, starting at 1 (though this will change in homework 4). Here, we will consider the following subset of BASIC commands only:

* <LINE> PRINT <NEXP> prints the value of the numeric expression
* <LINE> LET <VAR> <NEXP> writes the value of the numeric expression into the variable <VAR>
* <LINE> LET <VAR> [<NEXP1>] <NEXP2> writes the value of <NEXP2> into the array <VAR> at the position <NEXP1>
* <LINE> GOTO <JLINE> jumps to the line <JLINE>
* <LINE> IF <BEXP> THEN <JLINE> checks if the Boolean expression <BEXP> is true and if so jumps to the line <JLINE>
* <LINE> GOSUB <JLINE> jumps to the line <JLINE> and remembers where you came from
* <LINE> RETURN goes back to the line immediately after the line from which the most recent GOSUB jump started
* <LINE> END terminates the execution of the program

In all of these cases, <NEXP>, <NEXP1>, <NEXP2> are numeric expressions, <BEXP> is a Boolean expression, <VAR> is a correct variable name, and <JLINE> is a valid line number in the program.

Remember, we are only telling you what these lines mean so you can gain a general understanding of the language. You could implement everything for the current homework assignment without knowing anything about what each line of code actually means.

**Other Things to Note**

There can be spaces (single or multiple spaces, or tabs, but not newlines) between parameters or in other places. When referring to an array variable (such as A[4]) there may or may not be spaces before and after the [, and there may or may not be spaces before the ]. However, we will not give you any empty lines, and you will never get spaces in the middle of a variable name, integer, or command name (RET URN is invalid).

When writing your parser, at least for now, you can assume that you will only get correct BASIC programs, and do not have to deal with any errors. Feel free to check for these errors: this may prove useful in homework 4.

You do not need to use the following fact until homework 4, but when a variable is used for the first time and has not been assigned a value, its value will be 0. The same applies if an array is read at a position where it has not been written. Similarly if you access an array outside what you have set, the array will be expanded and the values will be set to 0.

**Your Task**

You are to write a program that parses a BASIC program and produces an object-oriented data structure that contains everything you need to know about the program to further process it easily. In this homework, you will **only** need to pretty-print the program, but on homework 4, you will need to run/simulate the program. The hard part here is not the printing, but producing the right class structure. Consequently, if you write code that implements pretty-printing correctly by scanning the program, but you do not internally build a correct parse tree, you will get at most 25/100 points. More importantly, you will have put yourself at a huge disadvantage for homework 4, because you will need to implement all of the parsing then.

As an example, reconsider this line of code:

4 IF XYZ > 5 THEN 22

You might store this line of code in an IfStatement class, which has pointers both to a BooleanExpression class, and maybe a LineNumber class (or just an int or array index).

By typing make (or make basic\_interpreter), we should be able to produce an executable basic\_interpreter. This executable basic\_interpreter should receive a single parameter at the command line, the name of a file that contains a correct BASIC program. It should then output the program pretty-printed to cout, as described below.

**Considerations for Implementation**

In this project, you will need to design a complex class hierarchy using inheritance. We will discuss some basic thoughts that you may want to consider.

**Class Structure**

There are several basic types of entities that could potentially be used to build a parse tree for this project. They are listed below. Note that it is not necessarily required that you implement any particular one of these.

* Variables
* Numeric expressions
* Boolean expressions
* Commands

We suggest to have some kind of abstract base class tree for these elements, or nodes, of the parse tree. In fact, one possible implementation might involve a common ancestor for all four. Line number was not included in the above list, but whether you want to treat line numbers as just integers or another new data type is up to you - one can make arguments for either approach. For numeric expressions and commands, our starter code provides abstract class definitions. You don't have to use it, but it's a suggestion and will likely lead to good design, so think carefully if you want to discard it.

The only functionality that your objects need (for the time being) is printing themselves. The format() function is pure virtual in the base classes we provide: make sure to think about why this is the case before you design your class hierarchy.

**Variables**

Variables always have a name and sometimes have an index if they point to an array element. Therefore, two implementation options include having an optional pointer to a numeric expression array index or having two subclasses, NumericVariable and ArrayVariable.

**Numeric Expressions**

For each type of numeric expression, you probably want a separate class that inherits from the base class (one for variables, one for constants, one for '+', one for '-', etc.). We are giving you an example with a class Addition that inherits from NumericExpression. As an alternative to having Addition directly inherit from NumericExpression, it would also make sense to have another abstract class in the middle, say, called BinaryExpression. This one could capture the common parts for all binary operators, since they might share some code. Or maybe, instead of separate classes for Addition, Division, etc., you'd just want one class for all binary operators, and the objects remember which type of operator they use. All of these can work, and it's up to you to pick. Each object of a binary operator type will likely want to store pointers to the two numeric expressions it combines - our example shows what we mean.

**Boolean Expressions**

Similarly, for Boolean expressions, you might want to have classes inheriting from the base class BooleanExpression. Do you want an intermediate class for all comparisons of integers? It should probably store pointers to the two numeric expressions it compares. You don't need to deal with AND, OR, NOT, etc., but we may want to add those in the future, so consider this when designing your classes.

**Commands**

For commands, you will want to do something similar. We are providing you with a base class Command as a suggestion. You probably want to define a class for each type of command, which has pointers to all objects that are part of the command (like numeric or Boolean expressions). You might want each command to remember what line number it is on. For GOTO, IF, and GOSUB statements, you will need to think about how to store the destination. Do you want to store an integer, or an object of type LineNumber, or a pointer to the command that is on that line? In the latter case, you might have to wait until you have parsed the later line.

**Overall Approach**

Overall, you'll find that you have a lot of classes. You are **strongly** encouraged to plan out your class hierarchy before you start to code anything. Determine which classes inherit from each other, publicly or privately, and which contain instances of (or pointers to instances of) other classes as subcomponents. Use your understanding of "A has a B", "A is a B", or "A as a B" to help you with this design. Good class design will make your code much easier to edit and maintain in later parts of the project.

Given that you will have lots of classes, you probably want to break them up across multiple files. We suggest two files (a .h and a .cpp) for all classes related to numeric expressions, two for all classes related to Boolean expressions, and two related to all commands. Alternatively, you could have two files per class. Notice that Makefiles will really shine during compilation, so spend some time developing a good Makefile for this project. We **strongly** recommend that you develop and test each class individually. Don't try to write 20 classes all at once, and then compile for the first time. If you made a serious mistake, you might have replicated it many times.

**Data Structures**

For now, you do not have to simulate the program, so you don't need to worry about how to store values for variables or arrays, or how to remember all those GOSUB jumps (which could be nested). That will be on homework 4.

However, there may still be some structures you need to think about. For instance, how do you look up the command corresponding to a particular line? Remember that line numbers may not be consecutive in project part 2, so you might want to put some infrastructure into your program with this in mind. You might want to define a single overall class that encapsulates the entire BASIC program, and gives you all the functions that you need to interact with it. This design is reflected in the provided skeleton code.

You may freely use the vector and other containers from the STL in this assignment. However, do not use any parsing algorithms from the STL: you are expected to implement these yourself.

**Pretty Printing**

Each of the classes (for numeric and Boolean expressions and commands) must provide a public std::string format() const function which returns itself formatted in a string. When printing, obey the following rules about where to put spaces. "Space" means "exactly one space".

* Space between line number and command
* Space between command and its parameters
* Space between any two parameters of command
* No space between variable name and array brackets []
* Space before and after binary operators (+, -, \*, /, =, <, >)
* No space before or after opening or closing parentheses/brackets (except as required by the rule for commands)

Furthermore, you should do the following to demonstrate that your program has a strong enough representation of the BASIC input:

* For the IF <BEXP> THEN <JLINE> command, you should put square brackets [] around the <BEXP> when you print it.
* For each GOTO <JLINE>, GOSUB <JLINE>, and IF <BEXP> THEN <JLINE> command, you should put angled brackets <> around the destination line numbers.
* For Boolean expressions of the form <NEXP1> > <NEXP2>, they should be replaced by the equivalent expression <NEXP2> < <NEXP1>.

Your formatting functions will almost certainly want to call the format functions of the classes that they contain. Returning again to our example line of code:

4 IF XYZ > 5 THEN 22

Our IfStatement class's format function will itself call the format functions for the BooleanExpression class (and the LineNumber class, if you chose to define it), and should ultimately produce the following output:

4 IF [5 < XYZ] THEN <22>

Some sample input and output can be found in the examples folder of the skeleton code. README.md explains how different rule are applied to each input file to produce the output. It is suggested that you run your program on the sample input, and do a comparison between your output and the sample output. You can do it by using diff file1 file2. Make sure your output matches the sample output exactly.

**Parsing**

Besides good class design, most of your work will probably go into writing the parser. Parsing the line number and command name should be straightforward enough, since we promise there will be no bad inputs. We are providing you some simple code to do so (which you are welcome to change or completely rewrite). The difficult part will be parsing numeric and Boolean expressions.

It is highly recommended that this be the last thing you implement, and that you think carefully about how you might parse numeric expressions before reading the following advice. It will make much more sense once you understand the magnitude of the problem. For the purpose of the subsequent discussion, assume you've already parsed to a point where you know a numeric expression begins, and you have a string that contains it and possibly more stuff afterwards.

You'll first want to skip all whitespace, and then distinguish the following cases. Because we know that we are parsing a numeric expression, the first character must be one of ( (indicating a binary operator), a letter (indicating a variable or array), or a - or digit (indicating a numerical constant). Correspondingly, the function does the following:

* If the first character is a - or a digit, then we know that the expression is a constant. So we can simply return parse\_constant(line, position);.
* If the first character is a letter, then we must have a variable or array name. We could first get the full name with something like string name = parse\_variable\_name(line, position);. After this, we can skip over all whitespace until we see the first non-whitespace. There are now two cases:
  + If the character is anything but a [, then the variable was just a simple variable. position is already pointing to the correct place, and we have the variable name ready in name, and we return new Variable(name);.
  + If the character is a [, then we have an array. So we skip over the [ and all subsequent whitespace. At this point, we know that a new numeric expression must start, so we call NumericExpression\* index = parse\_numeric\_expression(line, position);. This recursive call will return the pointer for the expression inside the array indices, and also update the position to point to the place after. We again skip whitespace, and the next character of the string has to be the ]. We skip it, and now position points to the end of this numeric expression. All that's left to do is to return new ArrayVariable(name, index);.
* If the first character is a (, then the expression must be a binary operator. Therefore, it must be of the form (<NEXP1> operator <NEXP2>). After skipping all whitespace that may follow the (, it is safe to try and parse the expression <NEXP1>, using a recursive call NumericExpression \*left = parse\_numeric\_expression(line, position);. This will return the pointer to the correctly parsed first expression, and position will be the character right after that expression. By skipping all whitespace, we can now find the operator (such as + or -), and remember it in a local variable. We again skip all whitespace to find the beginning of <NEXP2>. That can be parsed using NumericExpression \*right = parse\_numeric\_expression(line, position);. Now, we again skip over all remaining whitespace and the closing parenthesis which we must find. At this point, position should point to the position right after the closing parenthesis. Depending on which operator we saw (for example, say we saw +), we can now just write return new Addition(left, right);

**Technical note**: because our BASIC subset has no ambiguous structures or precedence, it can be parsed from left to right with no backtracking. Furthermore, it is possible to implement the interpreter such that there are no cases where more than one character must be looked ahead. As such, instead of using a string and position, a string stream can also potentially be used.