CSCI 104 - Spring 2016 Data Structures and Object Oriented Design

Data Structures and Object Oriented Design

- Due: Tuesday, March 1, 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!

Skeleton Code

Some skeleton code has been provided for you in the `hw4` folder and has been pushed to the Github repository `homework-resources`. If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

```
$ git clone git@github.com:usc-csci104-spring2016/homework-resources
```

Copy the contents of `hw4` (and its subdirectories) over to a `hw4` folder under your `hw_usc-username` repository.

Overview

For the first part of the project you will be writing an Interpreter for a very simple programming language.

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 goes line by line, interpreting what needs to be done. 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.

You will be writing an Interpreter for a very limited version of BASIC called Facile, which supports only twelve kinds of statements. There is a generic `Statement` class, which serves as the base class for all twelve Facile statements (that is, there is a class for each Facile statement, and it inherits from the generic `Statement` class). You will need to add functionality for each of the twelve types of statements.

You will implement an `interpretProgram` function which will read in a Facile program from an input stream, interpret that program, and output everything to an output stream. In the skeleton code, we are

passing in a file stream as input and cout as output, but your program should work just as well if we pass in different streams to the `interpretProgram` function (such as redirecting output to a file).

A valid Facile program is a sequence of statements, one per line. **You will only receive valid Facile programs as input, so you don't have to do error checking of this type.** Here is an example of a Facile program:

```
LET ABRA 42
PRINT ABRA
GOSUB 7
PRINT ABRA
PRINT CADABRA
GOTO 10
LET ABRA 9001
LET CADABRA 3720
RETURN
PRINT ABRA
END
.
```

Each line contains exactly one statement (there are no blank lines). A line number is assigned to each line, where the first line is numbered 1, the second line is numbered 2, etc. The last line is always a period. Execution of the program always starts at line 1. A Facile program may not exceed 1000 lines.

Variables

A variable in a Facile program consists of a string of upper-case letters. Each variable is capable of storing an integer value. They do not need to be declared: if a variable is referenced before a value is set, the variable should have a value of `0`.

The value of a variable can be changed with a LET statement, and printed with a PRINT statement. The PRINT statement prints the value of a single variable on its own line.

```
LET HAN 42
LET SOLO -9
PRINT HAN
PRINT SOLO
.
```

The above program would output:

```
42
-9
```

The PRINTALL statement should output the value of all variables which have been used: each line should have the name of a variable, followed by a space, followed by the value.

```
LET HAN 42
LET SOLO -9
PRINTALL
.
```

The above program would output:

```
HAN 42
SOLO -9
```

It's a little ambiguous how this should behave:

```
PRINT A
PRINTALL
.
```

Either of the following two outputs would be reasonable:

```
0
```

or

```
0
A 0
```

depending on how you implement your program, you may choose your functionality, just document your choice in your README.

Execution of a Facile program

A program is executed one line at a time, starting at line 1, and proceeding sequentially. The program terminates when an END statement is reached, or the last line of the program is reached (which consists of a single period).

You can cause the program to execute out of sequence via a GOTO statement, which indicates which line to jump to.

```
LET YODA 1
GOTO 4
LET YODA 2
PRINT YODA
.
```

The above program would output 1, since line 3 is skipped over via the GOTO statement. A GOTO statement can jump either forward or backward. If a GOTO statement tries to jump to a line that is not within the boundaries of the program (non-positive integers, or integers larger than the number of lines in the program), the interpreter should terminate with the error message "Illegal jump instruction".

Mathematical operations

You can perform addition, subtraction, multiplication, and division.

```
LET LUKE 4
ADD LUKE 3
PRINT LUKE
LET LEIA 5
SUB LEIA 3
PRINT LEIA
LET REY 6
MULT REY LUKE
PRINT REY
LET FINN 7
DIV FINN 2
PRINT FINN
.
```

The above example will output:

```
7
2
42
3
```

Note that integer division is performed, so we round down. If a statement tries to divide by zero, the interpreter should immediately terminate with the error message "Divide by zero exception". Note that you

could have the statement `DIV PALPATINE 0` in your program as long as that line never gets executed.

IF Statements

An IF statement acts as a conditional GOTO statement. It performs a comparison, and jumps to the specified line number if the comparison is true.

```
LET CHEWBACCA 3
LET VADER 6
IF CHEWBACCA < 4 THEN 5
PRINT CHEWBACCA
PRINT VADER
.
```

This will print out only 6, since line 4 is skipped over. A comparison can use any of the following operators: `<`, `<=`, `>`, `>=`, `=` (equal to), or `<>` (not equal to).

An IF will always be followed by exactly five strings. The first is the name of the variable, the second is the operator, the third is an integer, the fourth is the word THEN, and the fifth is the line number.

As with GOTO statements, your program should terminate with the error message "Illegal jump instruction" if it tries to jump outside the boundaries of the program.

Subroutines

There are no functions in Facile, but a simpler mechanism called a subroutine, which is called with a GOSUB statement. A GOSUB is just like a GOTO, except that it allows you to use the RETURN statement to return to where you jumped from.

```
LET ANAKIN 1
GOSUB 6
PRINT ANAKIN
PRINT PADME
END
LET ANAKIN 2
LET PADME 3
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.

```
2
3
```

A RETURN statement returns from the *most recent* GOSUB statement. As an example:

```
LET KYLO 1
GOSUB 7
PRINT KYLO
END
LET KYLO 3
RETURN
PRINT KYLO
LET KYLO 2
GOSUB 5
PRINT KYLO
RETURN
.
```

The above program will set KYLO to 1, jump to line 7, print KYLO, overwrite KYLO with 2, jump to line 5, overwrite KYLO with 3, return from the most recent GOSUB statement by jumping back to line 10, printing KYLO, then returning from the first GOSUB statement by jumping to line 3, printing KYLO, and terminating.

```
1
3
3
```

Whitespace

There will be no blank lines in your Facile program, but there may be an arbitrary amount of whitespace placed between tokens. The following program would be valid:

```
    LET    Z  5
 GOTO   7
LET WEDGE   4
 PRINT WEDGE
PRINT         Z
  END
PRINT WEDGE
        PRINT  Z
    GOTO      3
  .
```

Reference chart

Here is a list of all Facile statements you will need to support. Syntax is exactly as written (you won't get lower-case 'let' in a valid Facile program).

```
LET *var* *int*  | Change the value of variable *var* to the integer *int*
-------------------------------------------------------------------------------
--------------------
PRINT *var*      | Print the value of variable *var* to output
-------------------------------------------------------------------------------
--------------------
PRINTALL         | Prints the value of all used variables to output, one per line.
-------------------------------------------------------------------------------
--------------------
ADD *var* *p*    | Adds *p* to the value of the variable *var*, where *p* is an int
or variable.
-------------------------------------------------------------------------------
--------------------
SUB *var* *p*    | Subtracts *p* from  the value of the variable *var*, where *p*
is an int or variable.
-------------------------------------------------------------------------------
--------------------
MULT *var* *p*   | Multiplies the value of the variable *var* by the integer or
variable *p*
-------------------------------------------------------------------------------
--------------------
DIV *var* *p*    | Divides the value of the variable *var* by the integer or
variable *p*
-------------------------------------------------------------------------------
--------------------
GOTO *linenum*   | Jumps execution of the program to the line numbered *linenum*
-------------------------------------------------------------------------------
--------------------
IF *var* *op*    | Compares the value of the variable *var* to the integer *int*
*int* THEN       | via the operator *op* (<, <=, >, >=, =, <>), and jumps
*linenum*        | execution of the program to line *linenum* if true.
-------------------------------------------------------------------------------
--------------------
GOSUB *linenum*  | Temporarily jumps to line *linenum*, and jumps back after a
RETURN
```

```
--------------------------------------------------------------------------------
--------------------
RETURN          | Jumps execution of the program back to the most recently
executed GOSUB.
--------------------------------------------------------------------------------
--------------------
END             | Immediately terminates the program.
--------------------------------------------------------------------------------
--------------------
.               | Placed at the end of the program, and behaves like an END
statement.
```

Step 1 (Copy over your StackInt class)

Copy your StackInt code from HW3, as well as everything it is based upon (your LListInt class).

You MUST use your StackInt class to handle GOSUB and RETURN statements. Whenever you reach a GOSUB statement, you should push the line number you want to return to onto your stack. When you reach a RETURN statement, you should pop the line number from your stack and jump to that line. If you hit a RETURN statement and nothing is on your stack, the program should terminate as if it hit an END statement.

If you were unable to finish your StackInt class in the previous homework, you may instead use the STL stack. However, this will result in a 5% deduction from your score.

Step 2 (Add a Map to the Program State)

You will need to keep track of what are the values of each variable at all points in time. We require you to use the STL Map for this purpose. You will want to specifically use a `map<string, int>` so that you can pass in a variable name and quickly get out its value.

To implement the PRINTALL statement, we require you to use the map iterator. Note that this will display your variables in alphabetical order rather than the order they are added to the map.

Step 3 (Implement the LET statement)

You will probably want to start by thinking about how to implement the `LetStatement::execute` function. You will need to use your `ProgramState` map to accomplish this, and you will likely want to add some functions to your `ProgramState` class.

Next you will want to review how the LET statement is parsed in the `parseLine` function. You will need to implement this functionality for the other types of statements.

Step 4 (Implement the interpretProgram function)

The `interpretProgram` function should execute each line in the Facile program in the correct order of execution. You will want to place this inside a loop which breaks when the Facile program is supposed to terminate.

Step 5 (Implement the END statement)

Follow the template above to implement the next statement.

Step 6 (Test your code)

Make some small Facile programs using only LET statements and END statements to test the code you have written so far.

Step 7 (Implement the rest of the Interpreter)

Its your choice where to go from here, but we strongly recommend that you implement one statement-type at a time and thoroughly test it before moving on to the next statement.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your `hw4` directory to your `hw_usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2016/hw_usc-username.git`
5. Go into your hw4 folder `$ cd hw_username/hw4`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.

Acknowledgements

This project is based on "What's Simple is True," by Alex Thornton.