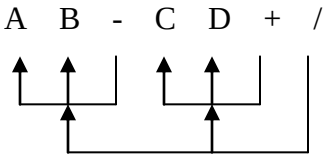
EECS 211 Program 7
Due: Monday, May 28, 2012

In this assignment we will implement a postfix evaluator for basic expressions.

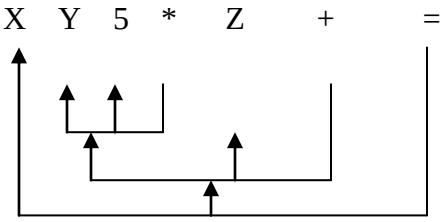
Background:

Evaluating postfix expressions is accomplished with a stack and a list of the variables. In our program we will have a **symbolList** object to hold each variable declared by a VAR operator. When the variable is first declared, its value is undefined. When an assignment operator is found in the postfix expression, a new value will be assigned to the variable. When an operation that requires the value of a variable is encountered, the program searches for the variable in the variable list and retrieves its current value. Thus, the variable list operates much like a stack frame in a program.

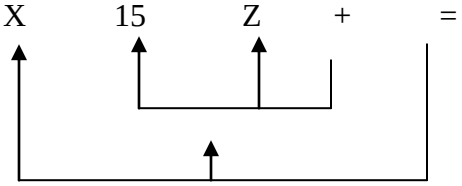
The actual evaluation is carried out using a stack. Remember, a postfix expression is evaluated left-to-right. Each time an operator is encountered, that operation is applied to the most recent values in the expression, and the result becomes, in effect, a new single item. Recall the following diagram from the Overview document:



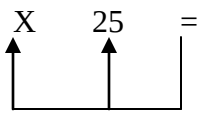
Now consider a simple assignment statement in postfix notation, with the links showing the connection between operators and sub-expressions:



This statement says to compute Y*5 first, then to add that to Z, and finally to store the result in X. Suppose Y had the value 3 and Z had the value 10. After performing the multiplication, the effect is to have a new postfix expression



The addition operator applies to the result of the multiplication and Z. After performing the addition, we have the equivalent of



Performing the assignment operation means to take the most recent value, in this case 25, and assign it to the variable indicated by the second most recent element, in this case X.

Using "the most recent" elements suggests the use of a stack. The top levels of a stack are the "most recent" ones pushed into the stack. The algorithm for postfix evaluation is quite simple. Process each element left-to-right. Each variable or constant is pushed onto the stack. When an operator is encountered, remove (pop) the correct number of elements from the stack, look up the values of any variables, perform the operation, and push the result onto the stack. Consider the postfix for the above example:

X Y 5 * Z + =

After processing the first three symbols, the evaluation stack will look like:

5
Y
X
stack

The multiplication operator takes two arguments, so the top two elements are popped from the stack. The current value of Y is retrieved and multiplied by 5. The result, 15, is pushed back onto the stack

15
X
stack

Next, the variable Z is pushed onto the stack.

Z
15
X
stack

When the addition operator is encountered, the two arguments to which it applies are indeed the top two levels of the stack and in the proper order. These are popped, the current value of Z retrieved, and the result, 25, pushed back onto the stack.

25
X
stack

When the assignment operator is encountered, again the appropriate two arguments are the top two levels of the stack in proper order – the value to be assigned is on the top level and the variable to which the value is to be assigned is on the second level. The assignment operator removes these two elements, makes the assignment, but does not push anything back onto the stack. The stack is empty, and the assignment has been accomplished.

Note that the evaluation stack has both constants and variables. Some of the constants are from the original postfix expression; others are ones generated by operations. This means that our program will have to ascertain the types of the evaluation stack elements when it performs an operation in order to properly handle the operands – variables or constants.

Assignment:

(1) Add the following member function to the **VARIABLE** class:

int amIUndefined();

This function returns 1 if the variable is undefined and 0 if the variable has a valid value.

(2) Add the following functions to the **symbolList** class:

SYMBOL *retrieveElement(int j);

If j is a valid index (i.e., between 0 and the number of elements-1) return the j^{th} pointer in the list; otherwise, return NULL.

void setVariableValue(char *n, int v);

The first argument points to a variable name. If that variable occurs in the list, set its value to v; otherwise, do nothing.

void getVariableValue(char *n, int *v, int *u);

The first argument points to a variable name. If that variable occurs in the list, set v to be its value and u to be its undefined flag. If the variable does not occur in the list, set u to indicate undefined value.

(3) Add the following function to the **symbolStack** class:

int getTopLevelValue(int *v);

This function assigns v to be the value of the variable or constant on the top level of the stack. It returns an indicator whether the value was defined or not. If the top level of the stack is a constant, set v to the value and return the indicator for defined. If the top level is a variable retrieve the value and defined flag from the list of variables (see next item).

(4) Make the following additions and modifications to main.

(a) Move the declaration of the postfix symbol list out of the main function and up to the file level.

(b) Add a new file-level symbol list to hold the variable symbols encountered by VAR declarations in the postfix expression. This symbol list should NOT allow duplicates.

(c) Add a new variable of type **symbolStack** for the evaluation stack.

(d) Add code to evaluate the postfix expression generated by your infix-to-postfix translation. Thus, after your main function translates the program in the input file and prints the postfix, it will then execute the postfix. Use an integer variable to index the positions in the postfix expression. Of course, this index starts at 0 and increments by 1 after each postfix symbol is processed. Continue evaluating until the end of the postfix expression is reached (the retrieveElement function returns NULL). For each postfix element that is a variable or constant, simply push that element onto the evaluation stack. For an operator, perform the correct action for that operator as follows:

- VAR operator. Copy the top level of the evaluation stack to the variable list, then pop the element off of the evaluation stack.
- cout operator. The top level of the evaluation stack will be a variable. Print the name and value of that variable on a new line on the screen, then pop that element. If the value is undefined, print the variable name and a suitable error message.
- = operator. The second level of the evaluation stack will be a variable. If the top level is undefined, print a suitable error message. Otherwise, assign the value of the top level to the variable. In both cases, pop both elements from the evaluation stack.
- Arithmetic and comparison operators. Perform the operation on the top two levels. Then create a new **CONSTANT** object to hold the value of the result and replace the top two levels with this new **CONSTANT**. In the case of the comparison operators, the result is 1 if the test is true and 0 if the test is false. For division, if the top level of the evaluation stack is 0, print a suitable error message and use 0 as the value of the operation. If either level of the evaluation stack is undefined, print a suitable error message and use the value 0 as the value of the operation.

Comments, suggestions, and hints:

Implement and test the five new class function members first. Then implement the outer loop and constant/variable and VAR cases of the postfix evaluation code using a file that only has variables, constants, and the VAR operation and semicolons.. Then add assignment and cout, and test with a file having only those operations. Finally, add in the remaining operators. Use the print function for the evaluation stack and the variable list so that you can see how your program is handling these symbols, but be sure to remove these before submitting the assignment.

The postfix expression and the evaluation stack have symbols of various kinds – **VARIABLE** objects, **CONSTANT** objects, and **OP** objects. The elements of the lists and stacks are all of type **SYMBOL *** even though they point to objects of different kinds. Therefore, once you determine which type one of these elements is, you will need to use a cast in order to access the member functions in that derived class. For example, in the **getTopLevelValue** member function of the **symbolStack** class, if s is the top-level element of the stack and you determine that s points to a constant, you can retrieve the value of that CONSTANT object by a statement such as

```
val = ((CONSTANT *) s)->getValue();
```

The cast ensures that the **getValue** function in the class **CONSTANT** will be called. You will need to use such casts in many places in the new code for the evaluator.

Use a switch in the main loop of the evaluation code to branch to the cases for VARTYPE, CONSTTYPE, and OPTYPE. In the case for OPTYPE, again use a switch to branch to the cases for the various operators. This inner switch should have separate cases for VAR, ASSIGNMENT, and OUTPUT and a **default** case for the remaining operators. Finally, the default case should have a switch that branches to the individual arithmetic/comparison operators.

You need to be careful about the order for using the **getTopLevelValue** and **pop** functions. The algorithm, for example, simply says for a binary operator to get the value of the top two elements and replace them by the result. The functions we have must be applied in proper order to accomplish such an operation. Your case for arithmetic/comparison operators, for example, might look like the following:

1. Retrieve the value and defined flag for the top level.
2. Pop the top level.
3. Retrieve the value of defined flag for the new top level (original second level before step 2 above).
4. Pop the top level.
5. If either was undefined, create a new constant 0 and push onto the stack. Otherwise, compute the new result, create a constant to hold it, and push the constant onto the stack.

Test file p7input.txt:

```
VAR x15;  VAR a;  VAR b;  VAR c;
cout x15;
x15 = a + b;
cout x15;
a = 42;  b = 5;  c = 2;
cout a;
x15 = a*(b-c+10);
cout a; cout b; cout c;
cout x15;
c = b/(c-2);
cout c;
c = a<b;  cout c;
c = a>b;  cout c;
c = a==b; cout c;
a = ((c-2)==0);  cout a;
```