EECS 211 Program 6
Due: Monday, May 21, 2012

We are now prepared to implement our infix-to-postfix translator for basic expressions (i.e., expressions that do not involve branching).

Background:

Recall from the Project Overview document the two notations for expressions – infix and postfix. Infix expressions have the operators in between the operands, while postfix expressions have the operator after the operands. Compare, for example,

infix: (A - B) / (C + D)

and postfix: A B - C D + /

Recall that infix notation uses parentheses to indicate that the operators inside the parentheses are to be done before the neighboring operators outside. This idea of ordering the operands applies even when there are no parentheses. For example, in the statement

X = A + B * C;

the multiplication is done before the addition, and the addition is done before the assignment, even though this is the reverse order of their occurrence in the statement. Multiplication and division have higher "natural precedence" than addition and subtraction in the absence of parentheses. When there is a sequence of operators of the same precedence, in most cases the operations are done left-to-right. (Assignment in C/C++ is one of the exceptions to the left-to-right rule.)

Consider a slightly more complicated example:

X = (A+B)*C + D + E - F/(G+H);

The first addition, adding A and B, is inside parentheses, so even though multiplication has a higher natural precedence, A and B are added first. The second addition operator is not inside any parentheses, so in the above statement that operator has lower total precedence than the multiplication. The multiplication should be done before adding D. Adding D should be done before adding E. Finally, the subtraction has to wait until the division is performed because division has a higher natural precedence than subtraction and neither of these operators is inside parentheses. Of course, the division must wait until the addition of G and H, because that plus operator <u>is</u> inside parentheses.

Note, many textbooks describe the order of computation by saying that <u>all</u> operators inside parentheses are done before any operators outside parentheses. As seen in the above example, this is not quite true. If an expression has multiple parenthesized sub-expressions but there are lower precedence operators in between, the operators inside the later parenthesized sub-expressions are performed <u>after</u> all but the last intervening lower precedence operator. The left-to-right rule overrides the precedence of the operators in the later parenthesized sub-expressions. In the above example, the plus operator in the second parenthesized sub-expression has higher total precedence than the plus operators

in between the two parenthesized sub-expressions, but those intervening additions (addition of D followed by addition of E) are done before the addition of G and H. Only the subtraction operator has to wait for the addition of G and H and the division.

We can formalize the notion of precedence as follows:

- The operators are all assigned natural precedence values that represent their relative strength in binding neighboring operands.
- We define a "precedence boost" as the amount to be added to natural precedences to account for (nested) parentheses. For each level of parentheses, the precedence boost is increased by an amount greater than the largest natural precedence of any operator. This condition ensures that the total precedence defined in the next bullet will be greater for operators inside parentheses than for operators outside parentheses.
- Each occurrence of an operator in an expression has a total precedence which is the sum of its natural precedence and the precedence boost. Note, the same operator symbol may occur several times in an expression and have different total precedences for each occurrence.

In our project, the natural precedence will be 1 for assignment, 5 for addition/subtraction, and 6 for multiplication/division. The precedence boost for parentheses will 25. With these values, the total precedence of the various occurrences of operators in the above statement are:

```
X  =  (A  +  B) * C  +  D  +  E   -  F / (G  +  H);
   1      30     6     5     5      5   6      30
```

It is possible to translate infix expressions to postfix notation in a single left-to-right scan. In this method, when an operator symbol is identified it is impossible to tell if that operator can be done immediately because that depends on what lies further to the right in the expression. For example, if we have read X = (A + , we don't know if the plus is the next operation to be done until we see more of the expression. On the other hand, we can tell the relative order of the operator just read vs. earlier operators. In the case in the previous sentence, it is clear that the assignment operator can NOT be performed until at least after the addition because their two precedences are 1 and 30. Even after reading the B it is impossible to tell if the addition of A and B is the next operation. It is only when we read the closing parenthesis that we can tell for sure that the addition of A and B is the first operation to be performed in this statement. Now consider the situation after reading more of the expression: X = (A + B) * C + . We can tell at this point that the multiplication operator is the next operation to be performed (after adding A and B) because the occurrence of that operator has total precedence 6 while the occurrence of the second plus operator has total precedence only 5. Note that the assignment operator, which was the first operator read in the statement still has to wait. So, at this point we know that the first operation is to add A and B and the second operation is to multiply that result by C; in postfix notation this would be A B + C *. We also know that the assignment operator definitely has to wait until after the second addition. We cannot tell anything more about the second addition until we read further into the statement, in particular until we have read past the D and seen the third plus operator. In general, earlier operators with lower total precedence wait and are performed later in the

evaluation, i.e., occur later in the postfix expression. This suggests the use of a stack to hold operators as the infix expression is read left-to-right. If an operator symbol is read and has lower total precedence than the operator on the top of the stack, the operators on the stack are moved to the postfix expression until the stack is empty or the operator on top has lower precedence. Then the new operator is pushed onto the stack. If a closing parenthesis is read, all operators on the stack that were inside the parentheses (i.e., which have total precedence higher than the reduced parentheses boost) are moved to the postfix. Note, operands (i.e., variables and constants) occur in the same order in the infix and corresponding postfix expressions. So, when an operand is read, it is appended directly to the postfix.

We can summarize these comments as the following algorithm. In this algorithm, PB is the amount to be added to natural precedences for each level of nested parentheses.

1. Set precedenceBoost to 0. Set the postfix expression to empty. Set the operator stack to empty.
2. Read the next token in the infix expression.
3. If the token is a variable or constant, append it to the postfix expression. Return to step 2.
4. If the token is "(", increase precedenceBoost by PB. Return to step 2.
5. If the token is ")", first append operators from the operator stack to the postfix expression until either the stack is empty or the top operator has total precedence less than precedenceBoost. Then decrease precedenceBoost by PB. Return to step 2.
6. If the token is an operator, first compute the total precedence of that occurrence of the operator symbol. Then append operators from the operator stack to the postfix expression until either the stack is empty or the top operator has total precedence less than the total precedence of the new token. Then push the new token and its total precedence onto the operator stack. Return to step 2.
7. If the token is ";" (i.e., end of the expression or statement), append operators from the operator stack to the postfix expression until the stack is empty. Terminate.

The Appendix shows this algorithm applied to the expression listed earlier in this section.

Assignment:

(1) In definitions.h add the following new constants:

PARENBOOST	25
NUMBER_OF_OPERATORS	12
ASSIGNPREC	1
ADDPREC	5
SUBPREC	5
MULTPREC	6

DIVPREC	6
LESSPREC	4
GREATERPREC	4
EQUALPREC	4
IFPREC	1
WHILEPREC	1
VARPREC	1
COUTPREC	1
UNBALANCED_PARENTHESES	26
SYMBOL_LIST_FULL	29
SYMBOL_NOT_FOUND	30

(2) In system_utilities.cpp add a new class, **operatorPrecedence** with two data members – an **int** representing an operator number and an **int** representing the operator's natural precedence. Then declare an array of pointers to **operatorPrecedence** of length NUMBER_OF_OPERATORS. Write a function **fillPrecedenceList** to fill this array with the correct information using the various constants for operator numbers and operator precedences. Call this function once at the beginning of main. (NOTE: This is very similar to **specialTokenList** from Program 5. You should be able to use that as a pattern and possibly even copy and paste some of that code with minor modification.)

(3) Declare the prototype for the following function in system_utilities.h and implement the function in system_utilities.cpp.

int getOperatorPrecedence(int op);

Search your array of precedences for an element with operator number op. If one is found, return that operator's precedence. Otherwise, return UNRECOGNIZED_SYMBOL.

(4) In the function **getNextToken**, print out each new line from the input file as you read it. Later parts of this assignment will be checking for certain types of errors, for example unbalanced parentheses. By printing each file line on the screen we will be able to print error messages immediately below the line where the error occurred.

(5) Declare a new class **symbolStack** in symbol_list.h and implement the member functions in symbol_list.cpp. Although in this assignment the stack will contain only operators, in program 7 we will use a second stack for evaluating postfix expressions, and that stack may contain variables and constants. Therefore, the stack we implement here will be able to hold any of variables, constants or operators.

Class **symbolStack**:

- Data members:
    - An array of **SYMBOL** pointers of length LISTLEN. This will be the stack of operators used during the translation.
    - An **int** telling how many elements are currently in the stack.
- Function members:
    - A constructor. The constructor function has no arguments. It sets the number of elements in the stack to 0 indicating that the stack is empty when first created.
    - A print function. The print function has no arguments. It prints a suitable message if the stack is empty. Otherwise it prints the elements of the stack one per line, starting at the top level of the stack.
    - A push function. The push function has one argument – a pointer to **SYMBOL**. If the stack is full, do nothing and return SYMBOL_LIST_FULL. Otherwise, push the argument onto the stack and return 0.
    - A function to return a copy of the top level of the stack. The argument of this function is of type pointer to pointer to **SYMBOL**, i.e., **SYMBOL** **s. If the stack is empty, do nothing and return SYMBOL_NOT_FOUND. Otherwise, make a copy of the top level element, assign the argument to point to the copy, and return 0. Because we are implementing a general stack that can hold variables, constants and operators, this function must first retrieve the name of the top-level element and test to see what kind of object the top-level element is. Only then can this function malloc the right amount of memory to hold the new object. Therefore, this function will have three cases. Each case will have a malloc statement which assigns new memory to the argument pointer and a memcpy statement to fill that new memory with the data occurring in the top level.
    - A pop function. This function has no arguments. If the stack is empty, this function returns SYMBOL_NOT_FOUND. Otherwise, decrement the number of elements in the stack and return 0.

(6) In the main function declare a variable of type **symbolList** with duplicates to hold the postfix version of the input file, a variable of type **symbolStack** to hold operators during the translation process, and a variable of type **int** to hold the precedence boost for parentheses. Replace the code in the four cases of the switch as follows:

- For the variable case, create a new VARIABLE object using the token as the name and add to the postfix expression.
- For the constant case, convert the token to **int** format, create a new CONSTANT object with name "0CONST", and add to the postfix expression.
- For the operator case, get the operator number and its natural precedence. Compute the total precedence (natural precedence plus precedence boost). Call the function **popOperatorStack** described in item (7) using the total precedence. Then create a new **OP** object with the token, operator number and total precedence as data members and push this new object onto the operator stack.

- For the punctuation case, first get the token number. Then switch to one of three cases – left parenthesis, right parenthesis, or semicolon. Handle the three sub-cases as follows:
    - Semicolon: Call **popOperatorStack** with total precedence 0 to clear out the operator stack. Check if the parenthesis boost is 0. If not print an error message (i.e., call **printError**) and reset the parenthesis boost to 0. The error message should say " ***ERROR: Unbalanced parentheses".
    - Left parenthesis: Add PARENBOOST to the parenthesis boost.
    - Right parenthesis: If the parenthesis boost is not greater than or equal to PARENBOOST, print the same error message as above. Otherwise, decrease the parenthesis boost by PARENBOOST. In either case call **popOperatorStack** with the current parenthesis boost.

The main loop should process tokens until the end of the file is reached. After that, the main function should print out the postfix expression with a suitable labeling.

(7) Declare the prototype for the following function in system_utilities.h and then implement the function in system_utilities.cpp.

int popOperatorStack(symbolStack *stk, symbolList *postfix, int prec);

The first argument will always be the operator stack from the main function. The second argument will always be the postfix from the main function. The third argument will be either the total precedence of an individual operator or the current precedence boost or 0 as described in item (6). This function moves symbols from the operator stack to the postfix until either the stack is empty or the top level of the stack has precedence less than the third argument. Remember, the stack class has functions to make a copy of the top element and to pop the top element. Thus, the loop in this function first gets a copy of the current top level. It can then compare the precedences. If the stack element has greater than or equal precedence, then the copy is added to the postfix and this function pops the stack.

Comments, suggestions, and hints:

Implement and test the **operatorPrecedence** class and **getOperatorPrecedence** function with a throw-away main function. Implement and test the **symbolStack** class with another throw-away main function. You could then implement and test the variable and constant cases in the main switch using a test file that just had variables and constants. Then perhaps add operators and semicolon without parentheses next. Finally add parentheses and complete the assignment.

Test file p6input.txt:

```
x15 = y*42 + w/(a+b);
a=(b+c)*(d-e);
x  =  (a * ( b - c / (d + e) * (f - g) ) ) + 27;
z = (a-42;
j = x/3);
VAR a;
VAR b;
cout x15;
```

Appendix:

We show the algorithm applied to the statement

X = (A + B) * C + D + E - F / (G + H);

using the natural precedences and parenthesis boost given in the **Background** section.

Step 1: Initialize precedenceBoost, postfix expression and operator stack:

precedenceBoost: 0
operator stack:
postfix:

The next token is X, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 0
operator stack:
postfix: X

The next token is =, an operator. Step 6 applies. The total precedence is 1. The operator stack is already empty, so **popOperatorStack** doesn't transfer any operators. This symbol and its total precedence are pushed onto the operator stack.

precedenceBoost: 0
operator stack: (=,1)
postfix: X

The next token is (. Step 4 applies. Increment precedenceBoost.

precedenceBoost: 25
operator stack: (=,1)
postfix: X

The next token is A, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 25
operator stack: (=,1)
postfix: X A

The next token is +, an operator. Step 6 applies. The total precedence is 30. The symbol on the top of the operator stack already has total precedence less than 30, so **popOperatorStack** again does not transfer any symbols. The new symbol and its total precedence are pushed.

precedenceBoost: 25
operator stack: (=,1) (+,30)
postfix: X A

The next token is B, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 25
operator stack: (=,1) (+,30)
postfix: X A B

The next symbol is). Step 5 applies. First, call **popOperatorStack** with the current value of PrecedenceBoost - 25. In this case, only one operator gets moved. Then reduce precedenceBoost.

precedenceBoost: 0
operator stack: (=,1)
postfix: X A B +

The next token is *, an operator. Step 6 applies. The total precedence is 6. The symbol on the top of the operator stack already has total precedence less than 6, so **popOperatorStack** does not transfer any symbols. The new symbol and its total precedence are pushed.

precedenceBoost: 0
operator stack: (=,1) (*,6)
postfix: X A B +

The next token is C, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 0
operator stack: (=,1) (*,6)
postfix: X A B + C

The next token is +, an operator. Step 6 applies. The total precedence is 5. **popOperatorStack** moves operators one-by-one from the operator stack to the postfix expression until either the stack is empty or the top element has total precedence less than 5. In this case only one operator gets moved. Then push the + and its total precedence, 5, onto the stack.

precedenceBoost: 0
operator stack: (=,1) (+,5)
postfix: X A B + C *

The next token is D, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 0
operator stack: (=,1) (+,5)
postfix: X A B + C * D

The next token is +, an operator. Step 6 applies. The total precedence is 5. **popOperatorStack** moves operators one-by-one from the operator stack to the postfix expression until either the stack is empty or the top element has total precedence less than 5. In this case only one operator gets moved. Then push the + and its total precedence, 5, onto the stack.

precedenceBoost: 0
operator stack: (=,1) (+,5)
postfix: X A B + C * D +

The next token is E, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 0
operator stack: (=,1) (+,5)

postfix: X A B + C * D + E

The next token is -, an operator. Step 6 applies. The total precedence is 5. **popOperatorStack** moves one operator. Then the - and its total precedence, 5, aer pushed onto the stack.

precedenceBoost: 0
operator stack: (=,1) (-,5)
postfix: X A B + C * D + E +

The next token is F, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 0
operator stack: (=,1) (-,5)
postfix: X A B + C * D + E + F

The next token is /, an operator. Step 6 applies. The total precedence is 6. In this case the top stack element has precedence lower than 6, so no operators are moved. Just push the / and its total precedence, 6, onto the stack.

precedenceBoost: 0
operator stack: (=,1) (-,5) (/,6)
postfix: X A B + C * D + E + F

The next token is (. Step 4 applies. Increment precedenceBoost.

precedenceBoost: 25
operator stack: (=,1) (-,5) (/,6)
postfix: X A B + C * D + E + F

The next token is G, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 25
operator stack: (=,1) (-,5) (/,6)
postfix: X A B + C * D + E + F G

The next token is +, an operator. Step 6 applies. The total precedence is 30. Move operators one-by-one from the operator stack to the postfix expression until either the stack is empty or the top element has total precedence less than 30. In this case the top stack element has precedence lower than 30, so no operators are moved. Just push the + and its total precedence, 30, onto the stack.

precedenceBoost: 25
operator stack: (=,1) (-,5) (/,6) (+,30)
postfix: X A B + C * D + E + F G

The next token is H, a variable. Step 3 applies. Append this to the postfix list.

precedenceBoost: 25
operator stack: (=,1) (-,5) (/,6) (+,30)
postfix: X A B + C * D + E + F G H

The next symbol is). Step 5 applies. **popOperatorStack** is called with the current value of precedenceBoost (25) as the precedence. This moves operators one-by-one from the operator stack to the postfix expression until the stack is empty or the top operator has total precedence smaller than 25. In this case, only one operator gets moved. Then reduce precedenceBoost.

precedenceBoost: 0
operator stack: (=,1) (-,5) (/,6)
postfix: X A B + C * D + E + F G H +

The next token is ;. Step 7 applies. Call **popOperatorStack** to move all the operators in the stack one by one to the postfix expression.

precedenceBoost: 0
operator stack: (=,1) (-,5) (/,6)
postfix: X A B + C * D + E + F G H + / - =