CISC 220 – 010 Project #1 Due Monday, October 23, 2000

For our first program project, we will apply lists, stacks and queues to write a small calculator program. The following little sample run illustrates how the program should work:

```
% Calculate
A = 2
B = 5
C = 3
A + B * C
17
(A + B) * C + A
27
```

Notice that the calculator can process simple assignment of a value (integers only) and simple arithmetic expressions involving variables (A-Z), operators (+. -. *, %, and /), and parenthesis ((, and)). These are the only symbols the calculator can deal with.

Part A – The Symbol Table) First, we need a way to hold variable names and their values. Do this by defining an object, called `Assignment`, with two fields: symbol (1 char), and value (int). This object can be a simple struct. Next, we'll use a linked list to hold all of the `Assignments`. Do this by creating a new list class, called `SymbolTable`, which privately inherits the list class, and has the following interface:

```
SymbolTable();
~SynbolTable();
empty();
getValue( char, int& );
storeSymbolValue( char, int);
dumpTable();
clearTable();
```

Remember that the derived class can use the base class member function to implement the new interface. Once this class is coded up, make a small driver program to test the class – making sure every function works in all cases. When you have verified the correctness of the `SymbolTable` class, make a script file (`Proj1-A.scr`) in which you cat the class header file, your driver, and show a sample run. Use the following input in your test; after the input is finished, retrieve and display a value, and dump the table (i.e., print the whole thing out in some readable fashion), then clear the table. You should process *only* simple assignments, consisting of *variable = value*. All other input should be flagged as erroneous at this point.

Sample input:

```
D = 10
= 10 ---Error
A = 0
A + B  ---Error
M = -5
Z = 44
N = A + D ---Error
B = 8
C = -13
A = B  ---Error
D = 15
```

Part B – Input of Expressions) Input expressions for our little calculator are simply a sequence of characters; a *legal* expression is one consisting of only the characters mention above, and white space. All others are erroneous. In this part, write a driver program that reads a line of characters, one character at a time, discard any white space, and stores the characters in a queue. If the expression is erroneous, clear the queue, and read another line; otherwise print the contents of the queue by dequeueing each character and printing it. Once this code works, make a script file (`Proj1-B.scr`) containing a cat of the queue class header file, this driver, a compile and a run. Use the linked version of the queue class (without inheritance). Use the following input as a test – at this point don't check for balanced parenthesis.

```
A + B – C * D
A+B-C*D
A*(B + D) – G
A + 10           -----ERROR
AB – CD          -----ERROR
(A / B + D % F *(A + B)) * C
(A – B)
A + B – C.D  + R  -----ERROR
```

Part C – Infix to Postfix) Infix form of expression means the operator is between its two operands – as in `A + B * C`. Postfix form of expression means the operator directly follows its operators (HP calculators used to use this form called reverse polish notation). For example, the postfix expression `A B C * +` is equivalent to the preceding infix expression. There are two nice things about postfix expressions: 1) the lack of parenthesis; for example, `A B + C *` is

equivalent to the infix expression `(A + B) * C`, and 2) their ease of evaluation by programs.

Our next job is to take an infix expression (in a queue from part B) and convert it to a postfix expression, and store that in another queue for further processing. There is a neat little algorithm for doing this conversion, which goes as follows.

```
Assumption: We have a queue representing a valid infix expression
            involving only operators +, -, *, /, %

Algorithm:
      Initialize an operator stack, s

      While there are still characters in the infix expression
            Get a character
            Case
                  An operand        output to result queue

                  '('               push '(' onto s
                  ')'               pop and output all operators
                                    until encountering a '(' then pop
                                    the '('
                  '*', '/', or '%'  pop and output all '*', '/', and
                                    '%' operators from s, down to but
                                    not including the top most '(',
                                    '+', '-', or to the bottom of the
                                    stack

                  '+' or '-'        pop and output all operators down
                                    to but not including the top most
                                    '(', or to the bottom of the
                                    stack

      end of expression pop and output all operators.
```

Notice that you need two queues – the input queue representing the infix expression, the result queue representing the postfix expression and a stack for operators. Write a function that implements this conversion algorithm. Then write a driver to test this function. Once it works, make a script file (`Proj1-C.scr`) in which you cat your conversion function, your driver, and show a sample run.

Part D – Evaluation) Given a valid postfix expression (in a queue) it's a simple matter to evaluate it – but you need a stack again – this time for operands! Here's the algorithm.

```
Assumption:  We have a queue representing a valid postfix
      expression.

Algorithm:
      Initialize an operand stack s

      While there are still characters in the input queue
```

```
            Input a character
            Case
                  Operand convert to its integer value (using
                          your symbol table and push the value
                          on the stack s
                  Operator pop the two operands, compute the
                          result and push it on the stack s

      Pop and display the answer.
```

Implement this algorithm as a function – call it `Evaluate`. Test the function with a small driver. Once the function works, make a script file (`Proj1-D.scr`) in which you cat your `Evaluate` function and your driver, and show a sample run.

Part E – The Calculator) Finally! We can put all these pieces together to make a little program that inputs assignments, and infix expressions, and outputs results. All you have to do is take all the pieces from parts A-D, and add a little "glue" to make the program. There are a few little subtleties, though. For example, you need to deal with both `A = 5` and `A + B + C`, as inputs from the user. I'll let you decide how fancy to make it at this point. Just be sure to use the structure and functions as developed, and don't stray too far form the original problem. When you have a working calculator, make a script file (`Proj1-E.scr`) in which you cat you main program, and any new functions you wrote for this part, and show a sample run.

Extra Credit) Hmmm…. Do we really need this? ☺ Well, ok, for the crazy people out there, here are some additional things we might want our calculator to do. 1) Deal with <u>long</u> expressions by allowing the user to indicate a continuation with a special character (like a tab or backslash); 2) When do we check to see if a symbol is in the symbol table? If we wait until evaluation, we could prompt the user for a value of an undefined symbol (and add it to the symbol table). 3) How about saving the answer of one expression to use later? All we have to do is allow things like this: `A = B + C`. 4) Finally, (for the really crazy people!), we could make a little baby language out of this mess! The keywords could be SET, PRINT, INPUT, etc., and then you could write little programs in this language – stick them in a source file – and feed them to your calculator (now it should be called an interpreter).

What to hand in: Hand in all the script files generated above (and the extra credit if you're one of the crazy people!), stapled together as one set. (Don't forget to print your name on the front page!)