CSE 326: Data Structures
Assignment #2
January 14, 1998
due: Friday, January 23

In this programming assignment, you will use stacks and queues to convert infix arithmetic expressions such as $7-(2*3+53)*(8-12/2)$ into equivalent postfix expressions such as $7\ 2\ 3 * 53 + 8\ 12\ 2\ / - * -$ and then evaluate them, respecting operator precedence and parenthesization. This will give you a flavor of one task a compiler must do to compile just the arithmetic expressions in your programs. You should review the discussion of postfix expressions on page 106.

1. Implement the abstract data types `stack` and `queue` as `C++` classes. Be sure to include all 5 operations on page 74 for each of them. (In place of `MakeEmptyStack` and `MakeEmptyQueue`, just implement constructors that create an empty stack or queue, respectively.) You should also include `Print` members for debugging purposes. The "`info`" field for each stack or queue entry will be either a single character (representing an arithmetic operator or parenthesis) or an integer.

   Use the linked memory implementation for stacks (Algorithm 3.3) and the contiguous memory implementation for queues (Algorithm 3.2). You are responsible for memory leaks in the linked implementation. Make your maximum queue size 200. For each `info` field, the simplest solution is to use a `struct` consisting of three fields, an int, a char, and a bool. The value of the bool indicates whether the appropriate value is an int or a char.

2. Implement a procedure `infix_to_postfix` that reads an infix expression consisting of integers, arithmetic operators $(+, -, *, /)$, and parentheses from the input stream, and returns a pointer to a queue Q consisting of the equivalent postfix expression. The algorithm for doing this is outlined in Figure 1. For the purposes of this algorithm, a "token" is either an integer, an operator, or a parenthesis. The "precedence" of the operators is 2 for "$*$" and "/", 1 for "+" and "$-$", and 0 for "(". You may assume $-$ is always a binary operator; you do not have to handle unary $-$ operators. For ease of parsing, you may assume that tokens are separated by white space (one or more blank, tab, or newline characters). The algorithm also makes use of a local stack S.

   You should hand-simulate this algorithm on an infix expression such as the one given at the beginning of these instructions, to be sure you understand how it works.

3. Implement a procedure `postfix_evaluate` that takes a pointer to a queue Q produced by `infix_to_postfix` and, using a local stack, returns the integer that results from evaluating the postfix expression contained in Q. This procedure simply implements

```
while any tokens remain in the input stream do
  begin
    get a token t from the input stream;
    if t is a number
      then Enqueue(t, Q)
    else if S is empty
      then Push(t, S)
    else if t is "("
      then Push(t, S)
    else if t is ")"
      then while x ← Pop(S) is not "(" do Enqueue(x, Q)
    else begin
      while precedence(t) ≤ precedence(Top(S)) do Enqueue(Pop(S), Q);
      Push(t, S);
    end ;
  end ;
while S is not empty do Enqueue(Pop(S), Q);
```

Figure 1: The body of `infix_to_postfix`

Algorithm 4.3. You should assume that / is integer division, discarding the remainder, and reporting an error if the denominator evaluates to 0.

The reason for using a queue is that you can imagine two parallel processes cooperating to evaluate an infix expression: one converts to postfix, enqueuing as it does so and, in parallel, the other evaluates, dequeuing as it does so.

Don't forget to handle errors in the input. If the input contains an error, it is sufficient to report the error and terminate execution, as long as your program does not dump core. You needn't worry about memory leaks in this case.

Instructions for your driver, makefile, and turn-in will follow by electronic broadcast.