Coms 262 Project 4 - Infix Evaluator Spring 2018

Introduction

Infix notation places the binary operator between the left and right operands. Postfix notation places the operator *after* the two operands. Infix notation is the traditional notation, and it has several advantages, but it also requires the extensive precedence and associativity rules regarding the order of operation and the use of parentheses to override the precedence and associativity rules when necessary. On the other hand, in postfix notation, the operators appear in the order in which they are to be executed, so there is no need for parentheses or any rules of precedence or associativity.

In this project, we will write a program that will read an expression in infix notation, including parentheses, and will display that expression in postfix notation and display the value of the expression.

The Program Interface

The program begins by prompting the user for a numerical expression written in infix notation followed by an equal sign `=`. The numbers may be positive or negative, integer or floating point. After this expression has been entered, the program will display the expression in postfix notation and then display its numerical value.

Classes

This program will involve a number of classes. The expression can contain up to four types of *token*: numbers, operators, left parentheses, and right parentheses. Combinations of these types of token will be stored in a stack and a queue. Thus, we will have the `Token` class as a base class for the `Number` class, the `Operator` class, the `LParen` class, and the `RParen` class. The `Token` class and its subclasses are described in the documents `Token Class`, `Number Class`, `Operator Class`, `LParen Class`, and `RParen Class`.

The Functions

The heart of the program will be the three functions `infixToPostifx()`, `evaluatePostfix()`, and `getToken()`. I have written `getToken()`. You should read it over and understand it. It is not very complicated, but it does use polymorphism. The `getToken()` function will return a pointer to a token. Thanks to polymorphism, the pointer can point to any type of token. Exactly how the token is processed after that will depend on the type of token it *really* is. That will be handled by the `typeid` operator, described below. When `getToken()` reads the `'='` character terminating the infix expression, it will return a `NULL` pointer.

The `infixToPostifx()` function will call `getToken()` repeatedly to read all of the tokens of the infix expression. It will use the first algorithm described below to

create a queue of pointers to the `Operator` and `Number` tokens in postfix order. It will then return that queue.

The `evaluatePostfix()` function will dequeue the tokens and use a stack of numbers to evaluate the expression, according to the second algorithm described below.

The `typeid` Operator

C++ provides an operator that can be used to determine the type of object that a pointer points to. If `p` is a pointer to an object, then the expression `typeid(*p)` will return the type of object that `p` points to. For example, if we want to know whether `p` points to an `int`, we would write the following `if` statement.

```
if (typeid(*p) == typeid(int))
```

You should use the `typeid` operator as you process the tokens returned by `getToken()`.

To use the `typeid` function, you need to include the header file `<typeinfo>`.

The Algorithm to Convert Infix to Postfix

To convert the expression from infix to postfix, use the following algorithm.

1. Begin with an empty stack and an empty queue.
2. Process the tokens from left to right, as returned by `getToken()` according to the following rules.
   (a) If the token is a number, enqueue the token.
   (b) If the token is a left parenthesis, push the token onto the stack.
   (c) If the token is a right parenthesis,
      i. Pop tokens off the stack and enqueue them until a left parenthesis is popped.
      ii. Discard the right and left parentheses.
   (d) If the token is an operator,
      i. Pop tokens off the stack and enqueue them until one of the following occurs.
         A. An operator of lower precedence is on top of the stack, or
         B. A left parenthesis is on top of the stack, or
         C. The stack is empty.
      ii. Push the operator onto the stack.
3. After processing the last input token, pop all remaining tokens off the stack and enqueue them in the order in which they are popped.

The algorithm assumes that the data type stored in the stack and the queue are pointers to tokens, not the tokens themselves. In this way, polymorphism will allow us to store pointers to different types of token in the same structure.

The Algorithm to Evaluate a Postfix Expression

To evaluate the postfix expression, use the following algorithm.

1. Begin with an empty stack.
2. Process the tokens as they are dequeued according to the following rules until the queue is empty.
   (a) If the token is a number, push it onto the stack.
   (b) If the token is an operator,
      i. Pop two numbers off the stack.
      ii. Combine them according to the operator (add, subtract, etc.)
      iii. Push the result onto the stack.
3. When the queue is empty, there should be exactly one value on the stack. It is the value of the expression.

When you are finished, turn in *all* of the files necessary to compile your program. You work is due by 5:00 pm, Monday, April 8.