Parsing and Evaluating Arithmetic Expressions

In *infix notation* an arithmetic operator ($+$, $-$, $*$, $/$) appears between two operands (numbers, variables) to which it is being applied. Infix notation may require parentheses to specify a desired order of operations. EXAMPLE: $A*(B+C)/(D-F)$.

In *postfix notation* (also RPN or Reverse Polish Notation) arithmetic operator is placed directly after two operands to which it applies. It does not require parentheses. EXAMPLE: $ABC+*DF-/$.

In *prefix notation* arithmetic operator is placed directly before two operands to which it applies. It does not require parentheses. EXAMPLE: $/*A+BC-DF$.

An algorithm of converting infix notation to postfix notation:

1. Completely parenthesize the infix expression to specify the order of all operations.
2. Move each operator to the space held by its corresponding right (closing) parenthesis.
3. Remove all parentheses.

To find a value of an arithmetic expression, a compiler must

1. Convert the expression from infix form into postfix form.
2. Apply an evaluation algorithm to the postfix form.

Infix to Postfix Conversion

We need the following class members.

1. `str` is an object of class `string` containing the infix expression divided into tokens. It consists of individual tokens:
   (a) operator tokens ($+, -, *, /, \wedge, \sim$) where $\wedge$ denotes the exponentiation operator, and $\sim$ is a unary minus
   (b) operand tokens (variables and constants)
   (c) left parenthesis token (`LPAREN`) and right parenthesis token (`RPAREN`)
   (d) end token (`END`) which delimits the end of the infix expression
2. `tokens` is an array (vector) containing tokens from the input string `str`.
3. `opStack` is a stack (of integers) containing operator tokens, `LPAREN`, and `END`.

4. `postfix` is a queue (of characters or strings) containing the expression in postfix notation.
5. The function `toPostfix` converts a given expression (string) in infix form to postfix form.
6. The function `printPostfix` prints out the expression in postfix form converted by `toPostfix`. Note that `printPostfix` removes tokens from the postfix queue, so after printing the postfix expression the postfix queue is empty. How do you think it can be corrected?

Description of The Algorithm

1. Define a table `precTable` whose index is a token enumeration (for operators, operands, parentheses, or `END` token), and which returns an integer (priority). EXAMPLE:

Token:	$+$	$-$	$*$	$/$	$\wedge$	(	)	operand	`END`
input priority:	1	1	3	3	6	100	0	0	0
stack priority:	2	2	4	4	5	0	99	0	-1

2. Initialize `opStack` by pushing the `END` token.

3. Get the next token from the infix expression (from `tokens`).

4. Test a token and

   (a) if the token is an operand, enqueue it to `postfix`.

   (b) if the token is `RPAREN`, pop entries from `opStack` and enqueue them on `postfix` until a matching `LPAREN` is popped. Discard both left and right parentheses.

   (c) if the token is the `END` token, pop all entries that remain on `opStack` and enqueue them on `postfix`.

   (d) if the token is `LPAREN`, push it on `opStack`.

   (e) otherwise the token is an operator, therefore pop from `opStack` and enqueue on the `postfix` queue those operators whose stack priority is greater than or equal to the input priority of the current operator (corresponding to the current token). After popping these operators, the current token is pushed on the stack.

   (f) repeat the steps 3 and 4 until the token is equal to `END`.

Parsing of Infix Expression: An Example

Consider the expression in infix form: $A * B + (C - D/E)$.

```
token  opStack    postfix     Commentary
--------------------------------------------------------
       #                      push END token
A                             dequeue token
                  A           enqueue token on postfix
*                             dequeue token
       #,*                    push token on stack
B                             dequeue token
                  AB          enqueue token on postfix
+                             dequeue token
       #,+        AB*         pop and enqueue *, push +
(                             dequeue token
       #,+,(                  push token
C                             dequeue token
                  AB*C        enqueue token on postfix
-                             dequeue token
```

token	opStack	postfix	Commentary
	#,+,(,-		push token
D			dequeue token
		AB*CD	enqueue token on postfix
/			dequeue token
	#,+,(,-,/		push token
E			dequeue token
		AB*CDE	enqueue token on postfix
)			dequeue token
	#,+	AB*CDE/-	pop until (reached
#			dequeue token
		AB*CDE/-+	pop and enqueue the rest of stack on postfix

Class Parser

```
class Parser {
private: // enum constants
   enum EnumTokens { END, VALUE, LPAREN, RPAREN, EXP,
          MULT, DIV, PLUS, MINUS, UNARY_MINUS };
   static const char* delimiters;
   LinkedStack opStack; // operator stack
   LinkedQueue postfix; // postfix queue
   char curVal; // current value
   const char *tokens;   // array of tokens
   class Precedence {
       public: int input, stack;
       Precedence(int i=0, int s=0) : input(i), stack(s) {}
   };
   static const Precedence precTable[]; //precedence table
   static const char opTable[]; // conversion table
// class Parser continued
private: // utility functions
   EnumTokens getToken(int c, EnumTokens prevToken = VALUE);
public:
   Parser(string s) : opStack(), postfix() { // constructor
      tokens = s.c_str(); // convert to a C string
      opStack.push(END); // initialize opStack
   }
   // operations
   LinkedQueue getPostfix() {  return postfix;  }
   void printPostfix();
   void toPostfix();
};
```

Class Parser Implementation

```
// initialize constants, delimiters and precedence table
const char * Parser::delimiters = "+-*/^()␣";
const char Parser::opTable[] = {
   '\0', '$', '(', ')', '^', '*', '/', '+', '-', '~' };
const Parser::Precedence Parser::precTable[] = {
   Precedence(0, -1),  //END
   Precedence(0, 0),   //VALUE
   Precedence(100, 0), //LPAREN
   Precedence(0, 99),  //RPAREN
   Precedence(6, 5),   //EXP
   Precedence(3, 4),   //MULT
   Precedence(3, 4),   //DIV
   Precedence(1, 2),   //PLUS
   Precedence(1, 2),   //MINUS
   Precedence(8, 7)    //UNARY_MINUS
};
```

```
Parser::EnumTokens Parser::getToken(int c, EnumTokens prevToken)
{  if (c == '\0') return END;
   // token is a delimiter
   if (c == '^') return EXP;
   else if (c == '/') return DIV;
   else if (c == '*') return MULT;
   else if (c == '(') return LPAREN;
   else if (c == ')') return RPAREN;
   else if (c == '+') return PLUS;
   else if (c == '~') return UNARY_MINUS;
   else  if (c == '-') {
      if (prevToken == VALUE || prevToken == RPAREN)
         return MINUS;
      else return UNARY_MINUS;
   }
   curVal = c; // token is a variable or number
   return VALUE;
}
```

```
void Parser::printPostfix()  {
   LinkedQueue newpost;
   char item;
   while (! postfix.isEmpty()) {
      item = static_cast<char>(postfix.dequeue());
      newpost.enqueue(item);
      cout << item;
   }
   cout << endl;
   postfix = newpost;
}
```

```
void Parser::toPostfix( )  {
   int op; char c;
   EnumTokens newToken = END;
   int tokenIndex = 0;
   do { //do-while loop
      c = tokens[tokenIndex++];
      while (c == '␣') c = tokens[tokenIndex++];
      newToken = getToken(c, newToken);
      switch(newToken) {
      case VALUE:
         postfix.enqueue(curVal);
         break;
```

```
// Parser::toPostfix continued
      case RPAREN: op = opStack.top();
         while (op != LPAREN) {
            postfix.enqueue(opTable[op]);
            opStack.pop();
            op = opStack.top();
         }
         opStack.pop(); //remove LPAREN
         break;
      case END:
         while (! opStack.isEmpty() ) {
            op = opStack.top();
            postfix.enqueue(opTable[op]);
            opStack.pop();
         }
         break;
      case LPAREN:
         opStack.push(newToken); break;
```

```
// Parser::toPostfix continued
      case EXP: case DIV:
      case MULT: case PLUS:
      case MINUS: case UNARY_MINUS:
         op = opStack.top();
         while (precTable[newToken].input <=
                precTable[op].stack) {
            postfix.enqueue(opTable[op]);
            opStack.pop();
            op = opStack.top();
         }
         opStack.push(newToken); break;
      default:
         cerr << "Invalid␣token:␣" << newToken << endl;
         newToken = END;
      }
   } while (newToken != END);
}
```

```
#include "Parser.h"
#include <iostream>
#include <string>
using namespace std;
int main() {
   char line[100]; string str;
   while (true) {
      cout << "Enter␣expression:␣";
      cin.get(line, 100); cin.ignore(100, '\n');
      str = string(line); //convert line to a string
      if (str == "quit") break;
      cout << "Infix␣expression:␣" << str << endl;
      Parser par(str); // new parser
      par.toPostfix(); // convert to postfix form
      par.printPostfix(); // print
   }
   return 0;
}
```

Evaluating Postfix Expressions

If we have an expression in postfix form, we can evaluate it substituting for variables their values (if there are any variables) and after evaluation we get a number (value of this expression). To evaluate any expression (say $1\,2\,3 + * 4\,5 - /$) we repeatedly get tokens from the postfix expression. If the token is an operand, we push the value associated with it onto the stack. If it is a binary operator, we pop two values from the stack, apply the operator to them, and push the result back into the stack. (If it is a unary operator, we pop only one value, apply the operator to it, and push the result back into the stack.)

We assume that first we convert an expression from infix form to postfix form. Then we use the expression in postfix form to evaluate the expression by calling the function `getValue()`.

We need also to initialize the object of type `Evaluator` using an object of type `Parser` (see `main()` below). These are additional functions used in the class `Evaluator`:

1. `valStack` is a stack keeping values of operands.
2. `parser` is a pointer to a parser object we want to evaluate.
3. `isOperator()` receives a token as an argument and returns true if that token is a valid operator.
4. `eval()`—a function that applies a binary operator (third argument) to two numeric operands of type `int` (first and second arguments), and returns the result of this operation. (A similar function with two arguments exists for unary operators.)

The implementation of the class `Evaluator` presented here is based on an integer stack and queue. Therefore, only character tokens are accepted (that is, only digits from 0 to 9 can be used). Also, division of two integers gives an integer (for example, 2/3 is 0, or 6/4 is 1). This restrictions will be removed in your next assignment where characters will be replaced by strings, and division of two numbers will return a number of type `double`.

This is an example how to run the evaluator.

```
Enter expression: (1-2)*3
Expression: (1-2)*3
12-3*
Value = -3
Enter expression: 2*3/4
Expression: 2*3/4
23*4/
Value = 1
Enter expression: quit
```

The Class Evaluator

```
class Evaluator {
private: // exception
   class DivisionByZero : public RuntimeException {
      public: DivisionByZero() :
              RuntimeException("Division␣by␣zero") {}    };
   Parser *parser; // pointer to the current parser
   LinkedStack valStack; // value stack
   int eval(int v1, int v2, Parser::EnumTokens token)
       throw(DivisionByZero); // two-argument evaluator
   int eval(int v1, Parser::EnumTokens token); // one-arg eval
   bool isOperator(Parser::EnumTokens token);
public:     //constructor
   Evaluator (Parser *par) : parser(par), valStack() { }
   int getValue(); // get a value of the expression
};
```

Implementation of the Class Evaluator

```
#include <cmath> // used for pow
#include <cctype>
#include "Evaluator.h"
#include "RuntimeException.h"
int Evaluator::eval(int v1, int v2, Parser::EnumTokens token)
                    throw(DivisionByZero)
{  if (token == Parser::EXP)
      return static_cast<int>(pow(static_cast<double>(v1), v2));
   else if (token == Parser::PLUS) return v1 + v2;
   else if (token == Parser::MINUS) return v1 - v2;
   else if (token == Parser::MULT) return v1 * v2;
   else if (token == Parser::DIV)
      if (v2 != 0) return v1 / v2;
      else throw DivisionByZero();
   return 0;
}
```

```
int Evaluator::eval(int v1, Parser::EnumTokens token) {
   if (token == Parser::UNARY_MINUS) return -v1;
   return 0;
}

bool Evaluator::isOperator(Parser::EnumTokens token) {
   return token == Parser::EXP || token == Parser::DIV ||
      token == Parser::MULT || token == Parser::PLUS ||
      token == Parser::MINUS || token == Parser::UNARY_MINUS;
}
```

```
int Evaluator::getValue() {
   Parser::EnumTokens token;
   int v1, v2;
   LinkedQueue& postfix = parser->postfix; // alias
   while (! postfix.isEmpty()) {
      try { // prevToken is not used here
         token = parser->getToken(postfix.dequeue());
      } catch(...) {
         cerr << "postfix␣queue␣error␣" << endl;
      }
```

```
// Evaluator::getValue continued
     if (isOperator(token)) { // operator
         try {
            if (token == Parser::UNARY_MINUS) {
               v1 = valStack.pop();
               valStack.push(eval(v1, token));
            } else {
               v2 = valStack.pop();
               v1 = valStack.pop();
               valStack.push(eval(v1, v2, token));
            }
         } catch(DivisionByZero& e) {
            cerr << e << endl;
            return 0;
         } catch (...) {
            cerr << "Wrong␣operand(s)␣" << endl;
            return 0;
         }
// Evaluator::getValue continued
      } else if (token != Parser::END) // operand
         try {
            int val = parser->curVal;
            if (! isdigit(val)) // only digits are accepted
              throw RuntimeException("Value␣is␣not␣an␣integer");
            val = val - ’0’; // replace char with an int value
            valStack.push(val);
         } catch(...) {
            cerr << "Wrong␣value:␣" << parser->curVal << endl;
         }
   } // end while loop
   try {
      return valStack.pop(); // return the final value
   } catch (...) { cerr << "Wrong␣expression" << endl; }
   return 0;
}
```