INFO0004-1 : Project 2 (20%)
Infix to RPN

Submission before December 15

Infix notation for expressions is the most familiar notation for mathematical expressions. For instance, the average of x and y is noted $(x + y)/2$.

Reverse Polish notation (RPN), or postfix notation, on the other hand, is a notation where the operators follow their operands. For instance, the average of x and y is noted $x\ y\ +\ 2\ /$. One way to "visualize" RPN is to consider that operands are pushed on a stack, that (binary) operators operate on the top two positions of the stack, with the result of operations being pushed on the stack (with the operands "consumed" by operators removed from the stack).

In this project, you will implement a C++ "parser" that takes expressions in infix notation and transforms it into a sequence of keystrokes for evaluation on an RPN calculator.

1 Expressions

To simplify, we will only consider expressions built from 6 basic expressions: x, y, $\sin(\pi * \text{expr})$, $\cos(\pi * \text{expr})$, the product of two expressions, and the mean (average) of two expressions.

Of the six basic expressions, two are simple expressions (x et y), the four others are composed expressions (that is, expressions made up of two or more expressions).

Your parser will read (infix) expressions from an input stream, where each basic expressions will be in the following format, where *exp1* and *exp2* represent components in composed expressions:

Expression	**Input**
x	`x`
y	`y`
$\sin(\pi * exp1)$	`sin(pi * exp1)`
$\cos(\pi * exp1)$	`cos(pi * exp1)`
$exp1 * exp2$	`(exp1 * exp2)`
$(exp1 + exp2)/2$	`avg(exp1, exp2)`

2 Lexical Analysis

Lexical analysis is a phase that usually precedes parsing and whose goal is to convert a sequence of characters into a sequence of tokens. Given the language (format) used for our expressions (see "Input" column in the table above), our lexical analyser, or lexer for short, will issue 10 different tokens and will have the following *public* interface:

```
class Lexer {
public:
  enum token {X, Y, SIN, COS, PI, OPEN_PAR, CLOSE_PAR, TIMES, AVG, COMMA};
  explicit Lexer(std::istream& in);
  token next();
```

```
  token peek();
  std::streamoff count() const;
  void reset();
};
```

The `next()` member function returns the next token, removing it from the input.

The `peek()` member function returns the next token, but without removing it from the input. Several consecutive calls to `peek()` must therefore return the same token, until a call to `next()` is interposed.

`count()` returns the number of characters that have been read from the `istream` since the last call to `reset()` that resets the character count to zero.

When your lexer encounters an unknown sequence of characters (for instance "`cas`"), it must throw a `domain_error` whose message is `"BAD TOKEN: "`, followed by the sequence of characters that triggered the exception.

Also, on encountering the end-of-file condition on the input, your lexer must throw a `domain_error("EOI")` exception to indicate the end of the input.

One of the main concerns of your lexer is to deal correctly with white characters in the input. For instance, `(x * sin(pi * y))` should produce the same sequence of tokens as `(x* sin(pi*y) )`, while `(x * si n(pi * y))` should throw an exception as `si` is not a valid character sequence.

3 Parser

Your parser will essentially transform the sequence of tokens issued by the lexer, and generate a sequence of keystrokes on an RPN calculator, so that the corresponding RPN calculation is equivalent to the evaluation of the expression on the input.

It should be noted that to place an operand on the top of the RPN stack, the `enter` key must be pressed, except if the operator key is pressed just after: `(x * y)` can thus be evaluated with the following RPN keystroke sequence

```
x
enter
y
*
```

Remembering that the result of the evaluations of operations (here the product) are automatically pushed onto the stack, your parser will generate a sequence of RPN keystrokes that evaluates the infix expression on the input.

For instance, the expression `(sin(pi * avg(x, y)) * x)` should generate a sequence of keystrokes similar to

```
pi
enter
x
enter
y
+
2
/
*
sin
x
*
```

The `class Parser` will expose the following public interface:

```
explicit Parser(std::istream& in);
bool parse(Exp& exp);
```

where `typedef std::vector<std::string> Exp` is the type of the "return argument" that will contain the RPN keystroke sequence. The `parse` function returns `true` if it successfully parsed a complete expression, and `false` if the expression is incomplete.

On encountering a malformed expression, the parser must immediately throw a `domain_error` whose message is `"PARSE ERROR at "`, followed by the relative character number *from the beginning of the expression* on the input.

Note that, the only valid expressions are those following the format given in the table above. For instance, you must assume that `pi` is *always* the left hand operand of the multiplication within the trigonometric functions.

4 Evaluation

This project counts towards 20% of your final mark. In this project, you will be evaluated on: readability of your code, organization of your code, usage of the STL, correctness, object-oriented design, judicious and correct use of C++ language features, code reusability, and efficiency of the solution.

5 Submission

Projects must be submitted before Friday December 15, 11:59 pm. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^N - 1$ marks (where N is the number of started days after the deadline).

Your submission, on the submission platform, will consist of the two files named `parser.hpp` and `parser.cpp`. Your classes must scrupulously expose the public interfaces described in this document. You are, however, free to define whatever private interface you see fit. You can also define your own function outside classes that you see fit, as long as these are properly hidden from your users and do not interfere with the good operations of the interface defined above.

Your programme must compile on the submission platform, without error or warning. Failure to compile will result in an awarded mark of 0. Warnings systematically result in lost marks.

Have fun...