# Lab Sheet 3

"Results! Why, man, I have gotten a lot of results!
I know several thousand things that won't work."
-- Thomas Edison

1. *Brace matching* or *parentheses matching*, is a syntax highlighting feature of certain text editors and IDEs that highlights matching sets of braces in languages, such as Java and C++ that uses them.

   Brace matching is one of the applications of *stack* data structure. Make use of the template Stack class (that you have already created) to create a *parentheses matcher*. Accept expressions of the form:

   ```
   (14+(20 * 31)/42)
   1+2
   ()
   2+3*(2^3)    etc.
   ```

   Accept expressions contains positive integers separated by white spaces and the operators +, -, *, ^ and /. Presence of any other characters should be reported as *parse error*.

   Create a *template* `Token` and `Tokenizer` classes to tokenize the input expressions. The `Tokenizer` class should have a function by the name `get_token()` that returns the *next token* from the input stream. The type of tokens returned should be:

   ```
   enum Token_Type {
         EOL,         /* End of Line.          */
         VALUE,       /* Value.                */
         LPAREN,      /* Left parenthesis.     */
         RPAREN,      /* Right parenthesis.    */
         PLUS,        /* Addition              */
         MINUS,       /* Subtraction.          */
         DIV,         /* Division.             */
         MULT,        /* Multiplication.       */
         EXP,         /* Exponentiation.       */
         UNKNOWN      /* Unknown token.        */
   };
   ```

   The `Token` template class should be designed as follows:

   Data members:

   | Name of data type | Description |
   |---|---|
   | `Token_Type token_type;` | For storing the type of the token (LPAREN, RPAREN, PLUS, etc.) |
   | `Typename token_value;` | The value of the token.<br><br>In our case, only the VALUE token possesses value. VALUE token represents integer numbers. For example, the input 10 is represented by a combination of (VALUE, 10).<br><br>Other tokens like left-parenthesis, plus, minus, etc. do not have a value. The token alone is required for them. |

   Constructors:

   | Constructor | Description |
   |---|---|
   | `Token(Token_Type tt = EOL, const T& val = 0);` | Default constructor. |

   Member functions:

   | Function name | Description |
   |---|---|
   | `Token_Type get_type()const ;` | To get the type of token. |
   | `const T& get_value()const ;` | To get the value of token. |

The `Tokenizer` template class should be designed as follows:

Data members:

| Name of data type | Description |
|---|---|
| `std::istream& in;` | The input stream to which the Tokenizer is connected to. |

Constructors:

| Constructor | Description |
|---|---|
| `Tokenizer(std::istream& is);` | Default constructor. |

Member functions:

| Function name | Description |
|---|---|
| `Token<T> get_token();` | To get the next token. |
| `void clear();` | To clear the input stream connected to the tokenizer. |

**Sample run of program:**

```
Enter expression: (
Unbalanced left parenthesis

Enter expression: )
Unbalanced right parenthesis

Enter expression: ()
Equal number of LPAREN & RPAREN

Enter expression: 1
Equal number of LPAREN & RPAREN

Enter expression: 1+2
Equal number of LPAREN & RPAREN

Enter expression: (1+2)*
Equal number of LPAREN & RPAREN

Enter expression: (1)+(2+3))
Unbalanced right parenthesis

Enter expression: ((1)+(2+3)
Unbalanced left parenthesis

Enter expression: $
Parse Error

Enter expression: 1/2^4
Equal number of LPAREN & RPAREN
```

2. Modify the main program to accept *real numbers* instead of *integer* numbers alone. For example, `(1.2 + 2.3) /1.0` etc. Do you have to make changes to the `Tokenizer` or `Token` classes?

3. Write a program to evaluate *postfix* expressions (*Reverse Polish notations*). Make use of the `Tokenizer` and the `Stack` template classes that you have already made. Use *white spaces* to separate the individual elements of the expression. The expression can contain real numbers and not just integers alone.

   **Sample run of the program:**

```
1 2 +
3
1.2 2.3 +
3.5
1
1
```

```
1 + 2
Error
*
Error
$
Parse error
1 2 + 3 4 * /
0.25
1 2 * *
Error
```