NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C07**

Arithmetic Expressions II

In this homework, we will again focus on the topic of arithmetic expressions and working with them. Specifically, we will preserve all the source code from the previous task and expand the functionality currently offered by allowing the construction of our expressions from input strings in infix notation.

The content of these input strings corresponds exactly to the assumptions we made the last time. It means we work with the operations of addition, subtraction, multiplication, and division. The first two listed have a lower precedence than the remaining two, all are left-associative. In addition to numbers (natural numbers or zero without unary signs), parentheses can also appear in the expressions, but only the round ones (i.e., not other types such as square brackets or curly braces). Individual numbers, operators, and parentheses occur immediately after each other, there are no separating spaces or other white characters in the strings.

A sample input string could be, e.g., `10*2+3*((1+14)-18)-10`. In general, we can assume that these input strings are valid (syntactically well-formed). But as we will describe later, we will detect certain incorrect situations anyway.

The core of the implementation will most likely be some kind of a parser class that will offer its interface through static functions. However, their specific form is not prescribed in any way. The only thing that needs to be followed exactly is to add the following constructor to our current `Expression` class:

- `Expression(const std::string& input)`: creates a new instance of an arithmetic expression based on parsing the provided input string in infix notation

Input strings will logically be perceived as sequences of tokens, which can be numbers, operator symbols, or symbols of opening or closing parentheses. We will then use the shunting-yard algorithm to process these tokens, i.e., parse such input strings.

---

foreach token t in the input string in infix notation **do**
  if t is a number **then**
    create a new leaf node for t and add it onto the stack of operands
  else if t is an opening parenthesis `(` **then**
    put `(` onto the stack of operators
  else if t is a closing parenthesis `)` **then**
    while there is an operator o on top of the stack of operators **do**
      remove o from the stack of operators
      remove two nodes r and l from the stack of operands
      create a new inner node for o based on l and r and add it onto the stack of operands
    remove `(` from the stack of operators
  else t is an operator n
    while there is an operator o on top of the stack of operators with precedence higher than n, or the same, but only if n is left-associative at the same time **do**
      remove o from the stack of operators
      remove two nodes r and l from the stack of operands
      create a new inner node for o based on l and r and add it onto the stack of operands
    add n onto the stack of operators
while the stack of operators is not empty **do**
  remove o from the stack of operators
  remove two nodes r and l from the stack of operands
  create a new inner node for o based on l and r and add it onto the stack of operands

---

If the algorithm runs successfully to the very end, exactly one node should remain on the stack of operands we used. This node will constitute the root node of our entire expression. We just need to take it and wrap it by an instance of the expression we were supposed to create.

Let us add that the entire algorithm has the property that it can successfully process certain input expressions that are not actually valid. Moreover, since we are not able to detect all such situations straightforwardly, we will not even attempt to do so. On the other hand, as already indicated, we can come to correct conclusion in certain situations of obvious errors. And that is why we will detect and treat such situations.

Not only because of this intention, we will propose a hierarchy of classes for exceptions. Specifically, we will assume an `Exception` class and its derived variants `ParsingException`, `MemoryException`, and `EvaluationException`. We will offer two constructors: `Exception(const std::string& message)` and `Exception(std::string&& message)`, where, in both cases, we expect a text message further describing a given error situation. As for the remaining interface, we will only offer a method enabling access to this message via `const std::string& what() const`.

When it comes to particular situations when to throw these exceptions and their text messages, we expect the following behavior. Let us start with the `ParsingException` exception, which is supposed to respond to incorrect input strings during their parsing.

- `Unknown token`: we encounter an invalid or unknown token
- `Malformed number token`: we are not able to correctly recognize a numerical value, including its overflow (`int` type is considered)
- `Missing operands`: we fail to construct the current operation node due to missing operands in the stack of operands
- `Unmatched closing parenthesis`: when processing a closing parenthesis, we are not able to find the corresponding opening parenthesis in the stack of operators
- `Unmatched opening parenthesis`: during the final cleaning of the stack of operators, we come across an opening parenthesis that has not yet been closed
- `Unused operands`: at the very end of the algorithm, the stack of operands contains more than just a single node
- `Empty expression`: or in the same situation, none on the contrary

We must also think of the situations when there will not be enough memory to perform dynamic allocation of individual tree nodes. We detect such cases by catching the standard `std::bad_alloc` exception. Instead of it, we just throw our `MemoryException` exception with a text message `Unavailable memory`.

If the parsing fails for any of the reasons mentioned above, we must not forget to correctly deallocate all the already successfully created nodes. We would otherwise not be able to ensure atomicity and consistency of the entire algorithm in terms of correct memory usage, i.e., we could lose our memory uncontrollably and irreversibly.

Finally, we will also start to handle situations where we would be attempting to divide by zero when evaluating the already constructed expressions. In this case, we will throw an `EvaluationException` exception with a message `Division by zero`.

Submit all the created source files (`*.cpp` and `*.h`) except for the main file `Main.cpp` with the `main` function. The predefined one will then contain a directive `#include "Expression.h"`.

The primary goal of this task is to verify the ability to implement a more complex algorithm according to a provided pseudocode. In terms of technical resources, it is necessary to demonstrate the use of the stack container, in the form of a polymorphic container allowing to store instances of different types of nodes. Again, follow the usual assignment requirements. In terms of proper decomposition, be sure to separate a mechanism for comparing operator priorities, regardless of the way you choose to implement it.

As already indicated, the actual implementation of the shunting-yard algorithm is not suitable to be placed directly in the provided constructor of the `Expression` class, we will therefore put it into a separate class. Considering its non-trivial extent, it surely deserves its own module, and so let us not forget to divide our code into appropriate files. Let us also add that all our auxiliary variables such as the stacks of operators or operands are only of temporary nature. In other words, it does not make sense to preserve them even after the parsing process is over, and so it is not appropriate to represent them as data members within the parser class.