NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C06**

Arithmetic Expressions II

In this homework, we will again focus on the topic of arithmetic expressions and working with them. Specifically, we 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 selected situations.

Not only because of this intention, we will propose our own custom hierarchy of classes for exceptions. Specifically, we will assume an `Exception` class and its derived variants `ParseException`, `MemoryException`, and `EvaluationException`. We will offer one constructor `Exception(const char* message)`. Since we will work only with fixed text messages further describing given error situations, we will pass them via C-style strings. As for the remaining interface, we will only offer a method enabling access to this message via `const char* message() 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 exceptions of the `ParseException` type, which are supposed to deal with incorrect input strings during their parsing.

- `Unknown token`: we encounter an invalid or unknown token (e.g., `3+a`, `3+1a`, or `3+a1`)
- `Overflow number`: we are not able to correctly recognize a numerical value because of its overflow
- `Missing operands`: we fail to construct the current operation node due to missing operands in the stack of operands (e.g., `1+`)
- `Unmatched closing parenthesis`: when processing a closing parenthesis, we are not able to find the corresponding opening parenthesis on the stack of operators (e.g., `1+2)`)
- `Unmatched opening parenthesis`: during the final cleaning of the stack of operators, we come across an opening parenthesis that has not yet been closed (e.g., `(1+2`)
- `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. Exactly that exception is thrown when an attempt at dynamic allocation using the `new` operator fails. 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 use, i.e., we could lose our memory uncontrollably and irreversibly (so called *memory leaks*).

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`. At the same time, we avoid repeated evaluation of the right subtree since it may not be trivial.

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"`. Everything needed must therefore be directly or indirectly available through this header file again.

The primary goal of this task is to verify the ability to implement a more complex algorithm according to a provided pseudocode and correctly work with dynamically allocated memory in connection with objects having a non-trivial life cycle and especially handling error situations with the aim of preventing memory leaks. 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 even 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.

Let us once again note that our goal is not to create a universal code that would be capable of parsing and processing expressions of arbitrary types. Therefore, we can hard-wire all specifics of arithmetic expressions (such as operations, precedence, associativity, etc.) directly into the code in an appropriate way. Specifically, it is not reasonable to design the parsing algorithm as a loop over general tokens, which we would first recognize, temporarily store as instances of some auxiliary classes, and only then process. Technical complications and growth of code volume would not outweigh the benefits. Moreover, such an approach would likely not be general enough anyway.

On the contrary, we should not forget the principles of decomposition here. In other words, we should handle different types of tokens or situations using appropriately designed functions. This approach should at least lead to a separation of the high-level logic of the entire parsing algorithm from handling of individual low-level situations.

Now that we can safely create instances of expressions by parsing them from input strings, it would likely be reasonable to prevent the end users from using our original constructors for expressions and individual nodes directly. Although this restriction is technically relatively easy to achieve, we will keep all the original constructors public because of the testing purposes. We will also continue to assume that any expression instances created through them will still be correct.

As previously mentioned, the main goal of the task is the manual control of the life cycle of dynamically allocated nodes within the internal trees of expressions. For this reason, it is again necessary to use the traditional C-style pointers to these nodes rather than unique smart pointers `std::unique_ptr`.

While the manual deallocation of the already created nodes and so expressions as a whole is relatively straightforward and was already addressed through the design of the respective destructors in the previous assignment, ensuring correct handling of dynamically allocated memory during the parsing algorithm is a significantly more complex challenge. We must prevent memory leaks even in various edge and error situations that may arise during the entire process and which may prevent it from reaching its successful end at all.

This specifically includes all situations in which we throw the previously discussed exceptions as a response to invalid input strings or unavailable memory. In particular, we must realize that memory issues can arise not only from the failure of the `new` operator we explicitly use to allocate new nodes but also when using standard containers that also rely on dynamic memory allocation. This includes the stack of operators as well as the stack of operands, but also operations involving `std::string` strings, even when creating them via methods such as `substr`. Even the creation of empty instances of all these containers may also potentially fail.

The first key aspect, therefore, is being able to identify all places where such failures might occur. The second key aspect is ensuring proper cleanup, i.e., manually deallocating anything that will not be deallocated automatically. This includes all instances of nodes we have successfully created, whether they are still in the stack of operands or have already been removed from it. In case of any failure, we must be able to perform kind of a rollback concerning the existing nodes.

The third key aspect is to carefully consider the particular placement of all necessary `try / catch` blocks. The objective is to avoid unnecessary repetition of code that will handle our compensation actions. An approach where we allow certain issues to bubble up as far as possible rather than addressing them immediately could help. In any case, we must ensure that the parsing method (and, consequently, the constructor of the expression class) fully handles all issues internally, exposing only our custom exception types from the `Exception` hierarchy.

Technically, we will need to formulate `catch` blocks where we want to catch any exception from this hierarchy. In such cases, we will naturally use a constant reference, i.e., `catch (const Exception& e) {...}`. However, the important point is that if we need to rethrow such a caught exception, we cannot do it using the standard `throw e;` approach, as this would cut the specialization. Instead, we rethrow it using a simple `throw;`.