NPRG041 – 2025/26 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 extend the currently provided functionality 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 had 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). As we will describe later, however, we will detect and handle certain incorrect situations.

Input strings will logically be perceived as a sequence of tokens, which can be numbers, operator symbols, or symbols of opening or closing parentheses. To process these tokens (i.e., parse the input string and construct the resulting expression tree), we will use the shunting-yard algorithm described in the following pseudocode. It is based on working with two stacks, one for operators (including opening parentheses) and another for operands (already created tree nodes). For both these stacks, we will use the standard `std::stack` container from the `<stack>` library. Specifically, `std::stack<char>` for the former and `std::stack<Node*>` for the latter.

```
1  foreach token t in the input string in infix notation do
2      if t is a number then
3          create a new leaf node for t and add it onto the stack of operands
4      else if t is an opening parenthesis ( then
5          put ( onto the stack of operators
6      else if t is a closing parenthesis ) then
7          while there is an operator o on top of the stack of operators do
8              remove o from the stack of operators
9              remove two nodes r and l from the stack of operands
10             create a new inner node for o based on l and r and add it onto the stack of operands
11         remove ( from the stack of operators
12     else t is an operator n
13         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
14             remove o from the stack of operators
15             remove two nodes r and l from the stack of operands
16             create a new inner node for o based on l and r and add it onto the stack of operands
17         add n onto the stack of operators
18 while the stack of operators is not empty do
19     remove o from the stack of operators
20     remove two nodes r and l from the stack of operands
21     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. 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.

The only required interface we must adhere to is the following new constructor of the existing `Expression` class. Since we neither need to store nor modify the input string, we will use the `std::string_view` type

from the `<string_view>` library to pass it. This type enables efficient encapsulation and unified processing of both C-style strings and `std::string` strings, moreover, without copying its content in the latter case.

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

Since the entire parsing algorithm is apparently non-trivial, it would not be a good idea to implement it directly inside this constructor. The core of our will therefore be a parser class that will encapsulate the entire parsing process. Although its exact form, interface, and even name are not prescribed at all, the overall concept of the solution must be followed.

One might understandably consider implementing all functions of this class as static, i.e., without the intention of creating its instances (similarly to the counter class from the third assignment). However, requiring a suitable decomposition at least at the level of processing individual tokens (at least because of decoupling the high-level logic of the entire parsing algorithm from the individual low-level situations to be handled), there will be quite a few such functions. Moreover, we would constantly have to pass the stacks between them, which would certainly be possible, but not particularly convenient.

We will hence adopt the following solution. Once again, we will offer the users a public static function, e.g., in the form `[[nodiscard]] static Node* parse(std::string_view input)`. Its task will be to carry out the entire parsing process and return the resulting pointer to the root node. Inside this function, we will use a private constructor to create an instance of the parser, maintaining both the stacks as its data members. All other helper methods can then be regular (i.e., non-static), and so the stacks will be accessible directly. However, since the constructor will be private, the users will still not have access to it.

Let us also explain the purpose of the (voluntarily) used `[[nodiscard]]` attribute. It is a general mechanism that allows additional information to be provided to the compiler. Specifically, this attribute enforces checking that the return value of a function is not entirely ignored by the caller. Of course, this cannot automatically guarantee program correctness, but it can reduce the risk of an accidental oversight. In our specific case, this is particularly appropriate due to the use of dynamic allocation.

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 propose our own custom hierarchy of exception classes. Specifically, we start with a base class `Exception` and derive from it variants `ParseException`, `MemoryException`, and `EvaluationException`. We will offer one constructor `Exception(const char* message)`. Since we will work only with fixed text messages (defined as constant expressions) further describing given error situations, we will pass them via C-style strings (and we do not become their owners). 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 begin 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, 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 (the 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 (including handling error situations because of preventing memory leaks). This means, it is again necessary to use the traditional C-style pointers to our nodes rather than unique smart pointers `std::unique_ptr`. 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.

Let us note that our intention is not to create a universal code that would be capable of parsing and processing expressions of arbitrary forms. 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.

While the manual deallocation of the already created nodes and so expressions as a whole is relatively straightforward and was already addressed through 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 constructors or even 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.

To fulfill both the aspects, it is necessary to carefully place the required `try` / `catch` blocks. The objective is not only to ensure correctness, but also avoid unnecessary repetition of code that handles our compensation actions. An approach where we allow certain issues to intentionally bubble up to more appropriate places rather than addressing them immediately could help. In any case, we must ensure that the parsing method fully handles all issues internally, exposing only our custom exception types from the `Exception` hierarchy. This specifically means that no other exceptions (such as `std::bad_alloc`) may propagate even into the constructor of the `Expression` class.

Technically, we will need to formulate `catch` blocks where we want to catch any exception from our 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 via `throw e`, as this would cut its specialization. Instead, we rethrow it using a standalone `throw`.

Finally, let us emphasize that within the ReCodEx tests, it is unfortunately not possible to cover all the cases where the discussed dynamic allocation might fail. Therefore, before submitting the final version, think everything through carefully and check your code very, really very thoroughly.