NPRG041 – 2023/24 Winter – Labs MS – **Big Assignment A01**

Regular Expressions

Regular Expressions

Regular expressions are one of the means we can use to describe the so-called regular languages. You will get acquainted not only with them during the summer semester within the Automata and Grammars course. In order to be capable of working with them from the practical point of view, fortunately for us, we can do without any larger theoretical background.

Regular expressions may be encountered in a number of contexts, not only within, e.g., programming languages or query languages in database systems. We usually work with variants that offer a variety of user-friendly simplifications or shortcuts, but we can do without them. In other words, we will consider regular expressions only in the form that is necessary to achieve the expected expressive power.

We define a regular expression v over some alphabet Σ (e.g., symbols of the English alphabet) inductively as follows. First, we introduce the following simple regular expressions:

- $\mathtt{a}$ for each $\mathtt{a} \in \Sigma$ corresponding to a language $h(\mathtt{a}) = \{\mathtt{a}\}$, i.e., language that contains only one single-symbol string $\mathtt{a}$,
- ε (empty string) corresponding to a language $h(\varepsilon) = \{\varepsilon\}$, i.e., language that contains only the empty string, which we denote as ε (sometimes also λ), and
- $\emptyset$ (empty language) corresponding to an empty language $h(\emptyset) = \{\}$, i.e., language that does not contain even a single string.

Given already defined regular expressions r and s (however complicated), using them and the following operations, we are able to construct the following compound expressions as follows:

- $r \cdot s$ (concatenation $\cdot$ operation) corresponding to a language $h(r \cdot s) = h(r) \cdot h(s) = \{uv \mid u \in h(r), v \in h(r)\}$, i.e., language that contains all strings formed such that we can split them to two substrings, the former of which can be generated by the first expression r and the latter by the second s,
- $r + s$ (alternation + operation) corresponding to a language $h(r + s) = h(r) \cup h(s)$, i.e., language that contains all strings that can be generated via the first expression r or the second one s, and
- r^* (iteration * operation) corresponding to a language $h(r^*) = \bigcup_{i \in \mathbb{N}_0, i \geq 0} L^i$, where $L^0 = \{\varepsilon\}$ and $L^i = L \cdot L^{i-1}$ for $\forall\, i \in \mathbb{N}, i \geq 1$, i.e., language that contains all strings formed such that we can split them to an arbitrary number of substrings, each of which can be generated by the expression r.

To ensure the correct evaluation of regular expressions, auxiliary round parentheses need to be used appropriately. However, this is often not necessary. Moreover, we can also afford to involve further simplifications. In particular, we will consider the following two:

- If it is not necessary, we will not write the round parentheses, and so, e.g., $((\mathtt{a} + \mathtt{b}) + \mathtt{c})$ corresponds to $\mathtt{a} + \mathtt{b} + \mathtt{c}$ and $((\mathtt{a} \cdot \mathtt{b}) \cdot \mathtt{c})$ to $\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}$.
- If we concatenate several symbols of our alphabet, we will not even write the $\cdot$ operator as such, and so, e.g., $\mathtt{abba}$ corresponds to $\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{b} \cdot \mathtt{a}$. We can generalize this idea so that any symbol $\mathtt{a} \in \Sigma$ or closing parenthesis) or iteration operator * can be immediately followed by another arbitrary symbol $\mathtt{a} \in \Sigma$ or opening parenthesis (, which again leads to the implicit concatenation operation in all these cases. For example, $(\mathtt{a} + \mathtt{b})^*\mathtt{aa}(\mathtt{a} + \mathtt{b})$ corresponds to $(\mathtt{a} + \mathtt{b})^* \cdot \mathtt{a} \cdot \mathtt{a} \cdot (\mathtt{a} + \mathtt{b})$.

In order to make sure we understand the construction and meaning of the introduced regular expressions, let us have a look at some examples over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$:

- $(\mathtt{a}+\mathtt{b})^*\mathtt{abba}(\mathtt{a}+\mathtt{b})^*$ describes a language $\{w \mid w = u\mathtt{abba}v, u, v \in \{\mathtt{a}, \mathtt{b}\}^*\}$, i.e., strings containing $\mathtt{abba}$
- $((\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b}))^*(\mathtt{a}+\mathtt{b})$ describes a language $\{w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*, |w| = 3k+1, k \in \mathbb{N}_0\}$, i.e., strings with length divisible by 3 with a remainder of 1
- $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{a} + \mathtt{b}(\mathtt{a}+\mathtt{b})^*\mathtt{b} + \mathtt{a} + \mathtt{b}$ describes a language $\{w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*, w$ starts and ends with the same symbol$\}$

Neighbors Method

A common problem we need to solve when working with regular expressions is the situation where we have to decide whether a given string corresponds to a given regular expression, i.e., whether such an expression can generate it. For example, string `aba` matches the last mentioned regular expression (strings starting and ending with the same symbol), while `abb` does not.

The mentioned problem is solved with the help of the so-called finite automata in practice. However, we must first be able to construct such an automaton for a given expression. Several methods can actually be applied, e.g., the method of neighbors proposed by Victor Mikhailovich Glushkov. In order to be able to use it, we first need to inductively calculate four auxiliary functions for the provided regular expression, based on which we can then construct such an automaton. However, this is not our goal (you will learn it during the summer semester), it is only the enumeration of these functions.

First, we need to assign an auxiliary number to each occurrence of an alphabet symbol $\mathtt{a} \in \Sigma$ in the specified regular expression (let us denote it as an index), using which we will then be able to distinguish individual occurrences of the same symbol from each other. This can be done in various ways, e.g., by sequentially assigning natural numbers starting with `1`. And that is exactly what we will do. For example, having a regular expression $(\mathtt{a} + \mathtt{b})^*\mathtt{ab}$, we get its indexed version as $(\mathtt{a}_1 + \mathtt{b}_2)^*\mathtt{a}_3\mathtt{b}_4$. Notice, however, that neither ε nor $\emptyset$ are symbols of the alphabet, they are therefore never assigned with these indices.

We are now ready to describe what auxiliary functions we need and how they can be calculated for an arbitrarily complicated indexed regular expression r':

- $\mathtt{Starting}(r')$ denotes the set of all indexed symbols of the alphabet by which some string corresponding to r' may begin.
- $\mathtt{Neighbors}(r')$ denotes the set of all pairs of indexed symbols that can occur immediately after each other in some string corresponding to r'.
- $\mathtt{Ending}(r')$ denotes the set of all indexed symbols of the alphabet by which some string corresponding to r' may end.
- $\mathtt{Epsilon}(r') \in \{\mathtt{true}, \mathtt{false}\}$ indicates a flag whether r' can generate the empty string ε.

Having our sample regular expression $r = (\mathtt{a} + \mathtt{b})^*\mathtt{ab}$ and its indexed version $r' = (\mathtt{a}_1 + \mathtt{b}_2)^*\mathtt{a}_3\mathtt{b}_4$, we specifically get the following values:

- $\mathtt{Starting}(r') = \{\mathtt{a}_1, \mathtt{b}_2, \mathtt{a}_3\}$
- $\mathtt{Neighbors}(r') = \{\mathtt{a}_1\mathtt{a}_1, \mathtt{a}_1\mathtt{b}_2, \mathtt{a}_1\mathtt{a}_3, \mathtt{b}_2\mathtt{a}_1, \mathtt{b}_2\mathtt{b}_2, \mathtt{b}_2\mathtt{a}_3, \mathtt{a}_3\mathtt{b}_4\}$
- $\mathtt{Ending}(r') = \{\mathtt{b}_4\}$
- $\mathtt{Epsilon}(r') = \mathtt{false}$

If we want to be capable of calculating these four functions in general, it is sufficient to follow the inductive structure of our regular expressions. In other words, for simple expressions, we express everything trivially, for expressions created by individual operations, we exploit the knowledge of these functions for individual operands. These are simpler, and, therefore, we really can calculate them before. Let us start with the simple expressions:

- Indexed symbol $\mathtt{a}_i$, $\mathtt{a} \in \Sigma$, $i \in \mathbb{N}$
  - $\mathtt{Starting}(\mathtt{a}_i) = \{\mathtt{a}_i\}$, $\mathtt{Neighbors}(\mathtt{a}_i) = \{\}$, $\mathtt{Ending}(\mathtt{a}_i) = \{\mathtt{a}_i\}$, $\mathtt{Epsilon}(\mathtt{a}_i) = \mathtt{false}$
- Empty string ε
  - $\mathtt{Starting}(\varepsilon) = \{\}$, $\mathtt{Neighbors}(\varepsilon) = \{\}$, $\mathtt{Ending}(\varepsilon) = \{\}$, $\mathtt{Epsilon}(\varepsilon) = \mathtt{true}$
- Empty language $\emptyset$
  - $\mathtt{Starting}(\emptyset) = \{\}$, $\mathtt{Neighbors}(\emptyset) = \{\}$, $\mathtt{Ending}(\emptyset) = \{\}$, $\mathtt{Epsilon}(\emptyset) = \mathtt{false}$

If r and s are arbitrarily complex indexed expressions, we then calculate our auxiliary functions for the individual operations as follows:

- Concatenation $r \cdot s$

  - $\texttt{Starting}(r \cdot s) = \begin{cases} \texttt{Starting}(r) & \text{if } \texttt{Epsilon}(r) = \texttt{false} \\ \texttt{Starting}(r) \cup \texttt{Starting}(s) & \text{otherwise} \end{cases}$
  - $\texttt{Neighbors}(r \cdot s) = \texttt{Neighbors}(r) \cup \texttt{Neighbors}(s) \cup \{xy \mid x \in \texttt{Ending}(r), y \in \texttt{Starting}(s)\}$
  - $\texttt{Ending}(r \cdot s) = \begin{cases} \texttt{Ending}(s) & \text{if } \texttt{Epsilon}(s) = \texttt{false} \\ \texttt{Ending}(s) \cup \texttt{Ending}(r) & \text{otherwise} \end{cases}$
  - $\texttt{Epsilon}(r \cdot s) = \texttt{Epsilon}(r) \wedge \texttt{Epsilon}(s)$
- Alternation $r + s$
  - $\texttt{Starting}(r + s) = \texttt{Starting}(r) \cup \texttt{Starting}(s)$
  - $\texttt{Neighbors}(r + s) = \texttt{Neighbors}(r) \cup \texttt{Neighbors}(s)$
  - $\texttt{Ending}(r + s) = \texttt{Ending}(r) \cup \texttt{Ending}(s)$
  - $\texttt{Epsilon}(r + s) = \texttt{Epsilon}(r) \vee \texttt{Epsilon}(s)$
- Iteration r^*
  - $\texttt{Starting}(r^*) = \texttt{Starting}(r)$
  - $\texttt{Neighbors}(r^*) = \texttt{Neighbors}(r) \cup \{xy \mid x \in \texttt{Ending}(r), y \in \texttt{Starting}(r)\}$
  - $\texttt{Ending}(r^*) = \texttt{Ending}(r)$
  - $\texttt{Epsilon}(r^*) = \texttt{true}$

Task Assignment

The goal of this task is to implement a simple application that calculates the described auxiliary functions used in the neighbors method for each regular expression from the set of expressions specified on the input.

Input regular expressions can be provided in two ways: either we find them directly as individual arguments passed from the command line, or we find them stored in text files whose names are again passed as arguments. To distinguish between the two situations, we will use two options, `-a` for the first behavior (arguments), and `-f` for the second (files).

We will process the passed arguments one after the other, from left to right. If we find an expected option, we process all the following arguments (none or more) accordingly (as an expression or a file) until we find another option or until all arguments have already been processed. If the first argument is not an option, we assume the default mode `-a`.

In order to process the arguments and find all input regular expressions, we propose a `Reader` class. Its declaration will be placed in a header file `Reader.h`. All functionality of this class is expected to be implemented through static methods. In addition to any other internal methods, only the following single method will be mandatory:

- `static void process_arguments(const std::vector<std::string>& arguments, std::vector <std::string>& expressions)`: processes the input program arguments and saves all the found input regular expressions (from arguments and files) in the form of ordinary strings `std::string`, preserving their order, in the already created (not necessarily empty) output container

Input files with regular expressions will have one regular expression per line (and nothing else), empty lines will be skipped.

Let us now look at sample input arguments: `(a+b) aa*+\epsilon+\emptyset* -a ab* -f file1.txt file2.txt -a (aa)*bb`. If the first listed file is completely empty and the second contains two expressions `(aaa)*` and `a*+b*`, all the regular expressions found will be as follows:

```
(a+b)
aa*+\epsilon+\emptyset*
ab*
(aaa)*
a*+b*
(aa)*bb
```

All regular expressions considered will be over the alphabet of English letters (lowercase and uppercase). For operations, we will use symbols `.` (concatenation), `+` (alternation), and `*` (iteration). Empty string expression will be denoted by `\epsilon` instead of ε, and, analogously, empty language expression will be denoted by `\emptyset` instead of $\emptyset$. We can also use auxiliary round parentheses `(` and `)`. In accordance with the explained rules, these parentheses can be omitted, as can the concatenation operation. Let us now assume that all regular expressions are correct, i.e., syntactically well-formed. Later, however, we will also add the ability to detect and treat certain error situations.

For the actual parsing of the regular expressions, we will use the extended shunting-yard algorithm and directly create a syntactic tree corresponding to the inductive structure of our regular expression with its help. Note that the algorithm pseudocode below does not handle implicit (missing) concatenation operators. As for the operators in general, we expect the following properties:

- Operator `*` is unary postfix and has the highest precedence
- Operator `.` is binary infix left-associative and has a middle precedence
- Operator `+` is binary infix left-associative and has the lowest precedence

```
1  foreach token t in the input expression do
2      if t is an alphabet symbol 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 if t is a unary postfix operator n then
13         remove one node p from the stack of operands
14         create a new inner node for n based on p and add it onto the stack of operands
15     else t is a binary infix operator n
16         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
17             remove o from the stack of operators
18             remove two nodes r and l from the stack of operands
19             create a new inner node for o based on l and r and add it onto the stack of operands
20         add n onto the stack of operators
21 while the stack of operators is not empty do
22     remove o from the stack of operators
23     remove two nodes r and l from the stack of operands
24     create a new inner node for o based on l and r and add it onto the stack of operands
```

To encapsulate the representation of a parsed regular expression, we will propose an `Expression` class. It is expected that its declaration will be placed in a header file `Expression.h`. In terms of the public interface of this class, we will provide the following constructor and method:

- `Expression(const std::string& input)`: creates a new instance of a regular expression by parsing the provided input string
- `Result evaluate() const:` based on the internal tree traversal, calculates all the functions from the neighbors method, and returns their values in the form of a `Result` class instance

The purpose of this `Result` class is to encapsulate all values in just one place, using the public data members `std::set<Symbol> starting`, `std::set<Neighbors> neighbors`, `std::set<Symbol> ending`, and `bool epsilon` for individual functions `Starting`, `Neighbors`, `Ending`, and `Epsilon` in that order.

As expected, the `Symbol` class represents one indexed symbol, the `Neighbors` class represents a pair of neighbors, i.e., a pair of such indexed symbols. Both of these classes must implement public methods `void print(std::ostream& stream = std::cout) const` through which we will be able to print them. Symbol `a` with index `1` will be printed as `a1`, pair of neighbors `a1` and `b2` then as `(a1, b2)`.

For the first three data items of the result class, we used the standard set container `std::set`, available in the `<set>` library. In order to do this, it is necessary to implement a comparison mechanism for the inserted objects, namely the `<` operator. We can easily achieve this through a global function `bool operator<(const Item& left, const Item& right)`. The function itself returns `true` if and only if the first operand is less than the second. Specifically, we will sort the indexed symbols in ascending order according to their indices, pairs of neighbors primarily according to the index of the first symbol and secondarily according to the index of the second.

In order to handle various error situations correctly, we will propose our own hierarchy of exception classes. Specifically, we will consider four exception types `ArgumentException`, `FileException`, `ParsingException`, and `MemoryException`, all derived from a common ancestor `Exception`. Two constructors will be offered: `Exception(const std::string& message)` and `Exception(std::string&& message)`, where in both cases we expect a text message describing the error situation in more detail. As for the additional interface, we will only offer a method to get this message using `const std::string& what() const`. Regarding particular situations when to throw these exceptions and their text messages, we assume the following behavior:

- Exceptions of type `ArgumentException`
  - `Invalid option`: we find an invalid option (argument starting with `-` other than any expected option) during the processing of input arguments
- Exceptions of type `FileException`
  - `Unable to open input file`: we are not able to open the specified input file with expressions
- Exceptions of type `ParsingException`
  - `Unknown token`: we encounter an invalid or unknown token
  - `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
- Exceptions of type `MemoryException`
  - `Unavailable memory`: dynamic allocation fails due to lack of memory

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

The goal of the task is to demonstrate the ability to work with constructs that we have encountered since the beginning of the semester. In addition to the basic skills, it involves working with program arguments, text files and streams in general, functions, parameter passing, design of classes, use of constructors and destructors, inheritance, virtual methods, and dynamic allocation.

The submitted implementation must, of course, be correct, stable, and without compilation warnings. The overall quality of the code will also be evaluated. It means especially, but not exclusively, the organization of the code into individual files, classes and functions, use of header files, naming of files, functions or variables, the overall visual style of the code and indentation, passing arguments by value or reference, quality of class design and use of inheritance and virtual methods, unnecessary repetition of the same code, use of named constants, handling of error situations, as well as use of standard libraries, containers or functions.