rs-regex,
Implementation

Interface

The program is implemented as a command line application and is written in Rust. For normal usage, the program takes a regular expression as the first and only argument like so:

cargo run "a(b|cc)*a"

After that, the program will ask strings as inputs one by one and tell you whether they belong to the language (defined by the regex) or not. An empty string will exit the program.

Alternatively, if you want to see how the regular expression is tokenized, you can give the option -t as on optional argument along with the regex. The program will print all of the tokens along with their values and exit. The output looks like this:

cargo run "a(b|c)*" -t

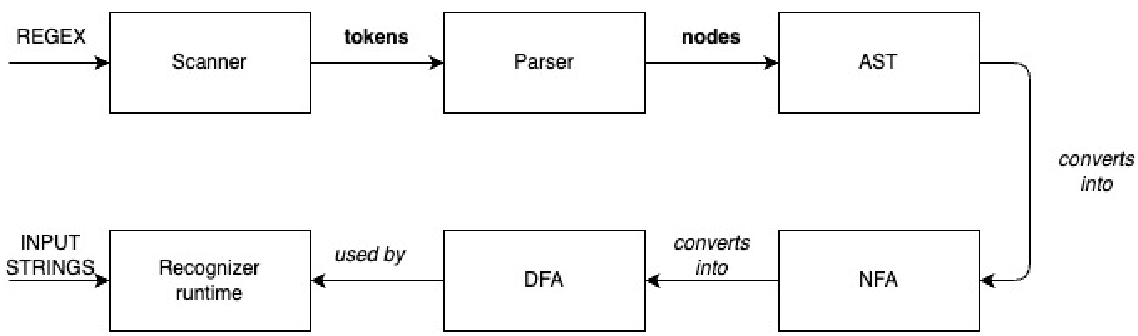
Token(Char, a)
Token(LeftParen, ()
Token(Char, b)
Token(Union, |)
Token(Char, c)
Token(RightParen,))
Token(Star, *)

The regular expression (regex) should be given inside double quotes. The set of supported characters are ASCII (8-bit), from which some special ones, such as the operator symbols, need to be escaped using backslash, e.g. *. The supported operators are:

Operator	Syntax	Matches
Union	A | B	"A" or "B"
Star	a*	0 or more "a"
Concatenation	01	"0" followed by "1"
Group	(a|bb)*	0 or more "a" or "bb"

Internals

The overview of the implementation looks like this:



Scanner

The purpose of the scanner (also known as lexer) is to split the the regex into tokens (**src/tokens.rs**) which will then be consumed by the parser. This stage is very simple for this particular program, since there are only 7 distinct token patterns which are mostly single characters. The patterns look like this

$$\begin{array}{lll}
\textbf{char} & \longrightarrow & \langle\text{ascii_non_special}\rangle \quad | \quad \text{"} \backslash \text{"}\langle\text{ascii}\rangle \\
\textbf{union} & \longrightarrow & \text{"} \mid \text{"} \\
\textbf{star} & \longrightarrow & \text{"} * \text{"} \\
\textbf{lparen} & \longrightarrow & \text{"} (\text{"} \\
\textbf{rparen} & \longrightarrow & \text{"}) \text{"} \\
\textbf{EOF} & \longrightarrow & \epsilon
\end{array}$$

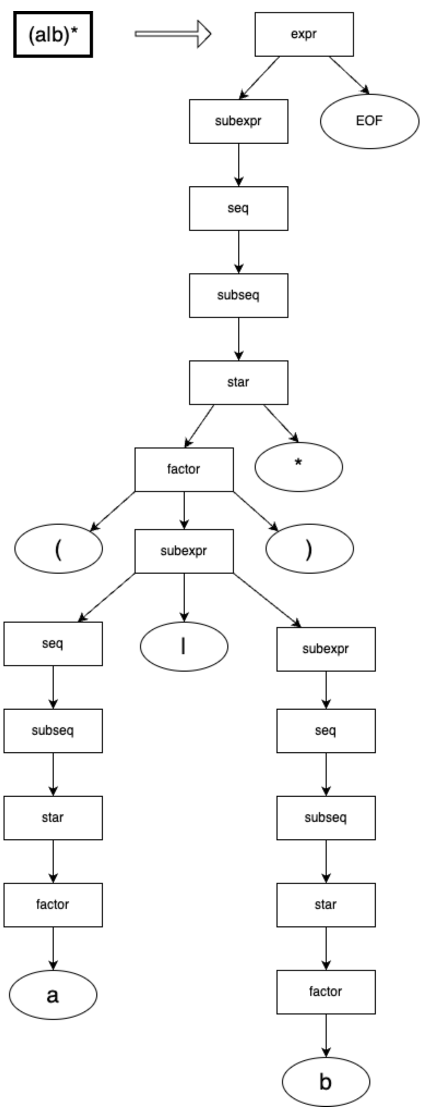
The scanner can be found in **src/scanner.rs**

Parser

The parser (**src/parser.rs**) request tokens from the scanner one by one during the parsing process. The job is to take the incoming tokens and put them in some kind of meaningful context. Often times, there are a specific set of tokens that the parser expects to encounter. If the incoming token does not match a single member of this sets, a parsing error has occured and the parsing process will be terminated. A succesfull series of matches will eventually result into the parser recogizing some notable syntactical structure. These structures can be described as productions in context-free grammar (CFG). For this program, the CFG looks like this:

<expr>	$\longrightarrow$	<subexpr> **EOF**
<subexpr	$\longrightarrow$	<seq> **union** <subexpr> | <seq>
<seq>	$\longrightarrow$	<subseq> | ϵ
<subseq>	$\longrightarrow$	<star> <subseq> | <star>
<star>	$\longrightarrow$	<factor> **star** | <factor>
<factor>	$\longrightarrow$	**lparen** <subexpr> **rparen | char**

Above, tokens are represented in bolded text. The specific techique implemented for parsing is LL(1). The first 'L' refers to "**L**eft-to-right", meaning that the input (RE) is read from left-to-right. The second 'L' refers to "**L**eftmost derivarion", meaning that from the right-hand-side of the productions, we are going to expand the leftmost derivation first. This results to so called top-down (or recursive descent) parsing, which can be visualized in the parse tree example shown below. The number one in LL(1) means that we-re making parsing decisions based on only one look-ahead symbol.



After a succesfull production, a node (**src/ast.rs**) will be created. The information that is relevent for any further stages of the program is stored in the node and it will be inserted as a part of an abstract syntax tree (AST). The AST is a striped-down version of the parse tree. It will only contain the information that is relevant for execution in a form that guarantees syntactical correctness and enables easy translation into a corresponding automaton.

NFA

Nondeterministic finite automaton (**srsc/nfa.rs**) is finite-state machine that can be used to recognize regular languages. This representaion will be constructed from the AST by converting the nodes of the AST into building blocks called NFA fragments (**src/nfa_fragment.rs)** and connecting them to each other in recursive routine (**src/ast.rs : fn to_fragment**). This will result into a single NFA fragment that will then be converted into NFA (**src/nfa_fragment.rs : fn to_nfa**) by reprocessing the state transitions. The process of converting the regular expression into an equivalent NFA is based on techniques discribed in the book "Introduction to the Theory of Computation, Third Edition, Michael Sipser" on pages 66-69. Here are a few examples of the illustrated conversions: