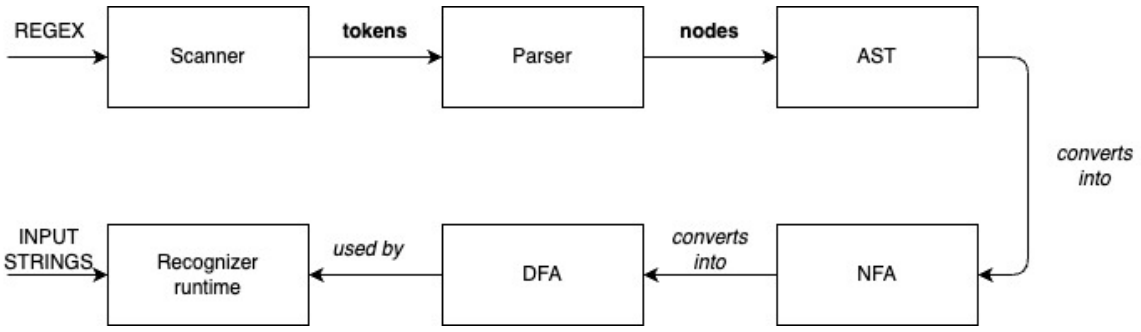
rs-regex,
Implementation

Overview

The basic structure of the program looks like this:



Interface

The program is implemented as a command line application and is written in Rust. For normal usage, the program takes a regular expression (RE) 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.

The RE 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"

Scanner

The purpose of the scanner (**src/scanner.rs**) is to split the the RE 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

char	⟶	<ascii_non_special> | " \ "<ascii>
union	⟶	" | "
star	⟶	" * "
lparen	⟶	" ("
rparen	⟶	") "
EOF	⟶	ϵ

As an example, the RE "a(b|c)*" will be split into following tokens:

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

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

Parser

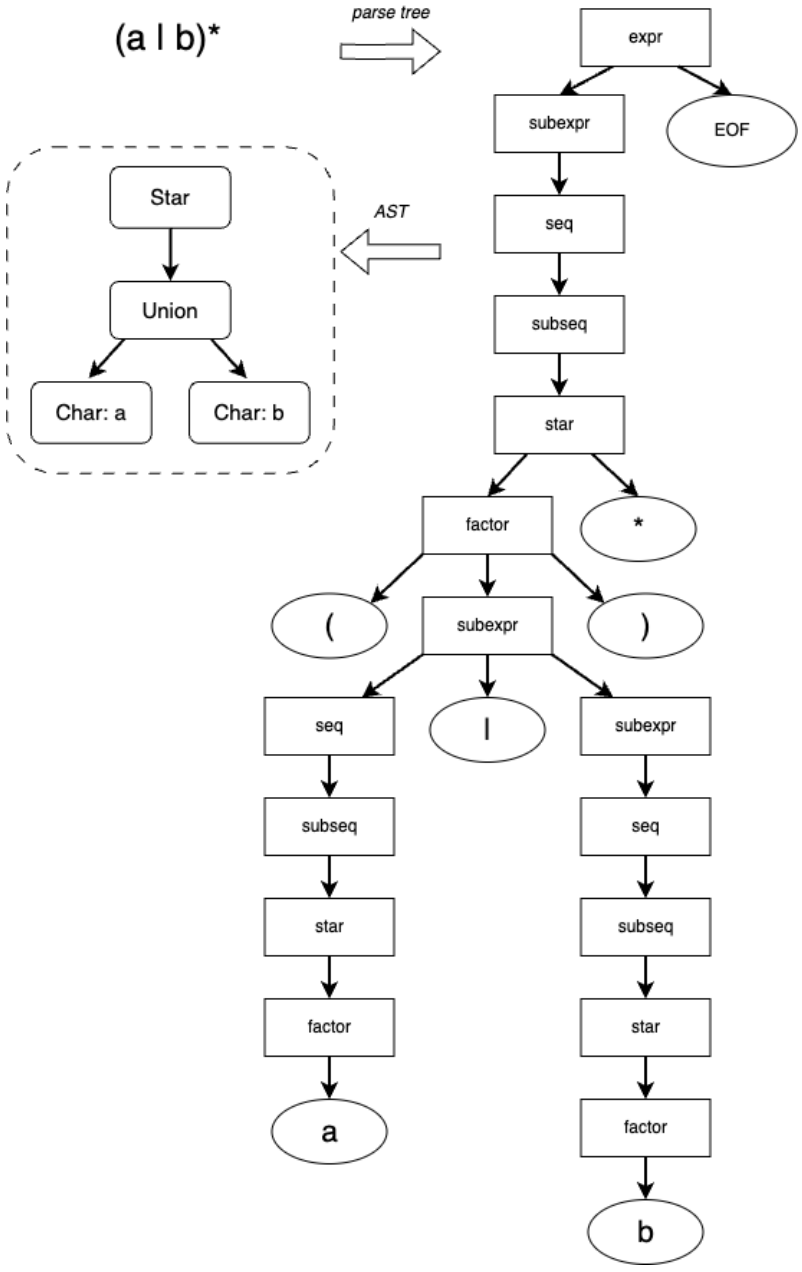
The parser (**src/parser.rs**) request tokens from the scanner one by one during the parsing process. The job is to take the incoming tokens and put them in some kind of meaningful context. Often times, there are a specific set of tokens that the parser expects to encounter. If the incoming token does 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. For the regular expressions, it is specified as follows:

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

Above, the bolded text refers to terminals (tokens) and labels between < and > symbols are non-terminals.. In order to translate this into a working program, each production (a line divided by an arrow) has been made to it's own function. The name of the function comes from the left-hand side of the arrow and a non-terminal on the right-hand side represents a function call.

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. The number one in LL(1) means that we're making parsing decisions based on only one look-ahead symbol. Some of the productions are a result left-recursion elimination. This is necessery in LL(1), since recursive derivation happening on the left essentially means making a recursive function call at the very beginning of a function. This would result in infinite recursion.

Using LL(1) results to so called top-down (or recursive descent) parsing, which can be visualized in the parse tree example shown below. The parsing starts from the root of the tree at non-terminal *expr:*



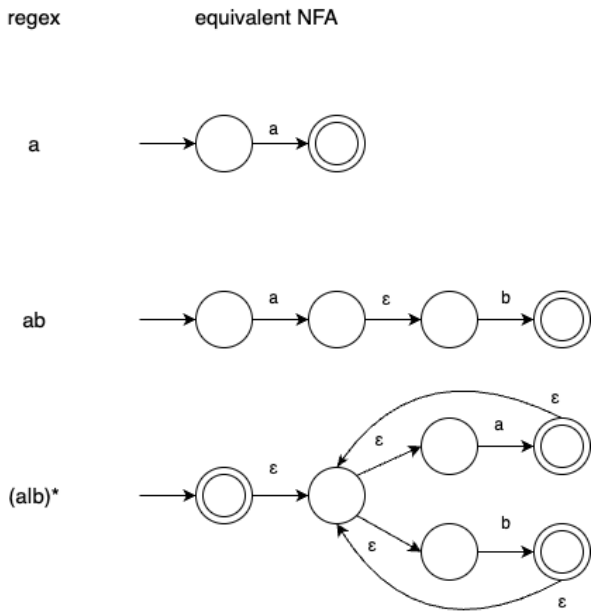
After a succesfull production, a node (**src/ast.rs**) will be created. The information that is relevant for any further stages of the program is stored in the node and it will be inserted as a part of an abstract syntax tree (AST). The AST is serves as a concise representation for the syntactical structure of the RE.

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, where the transitions are implemented as a hash table. Eash state is also uniquely labeled with an 32-bit integer.

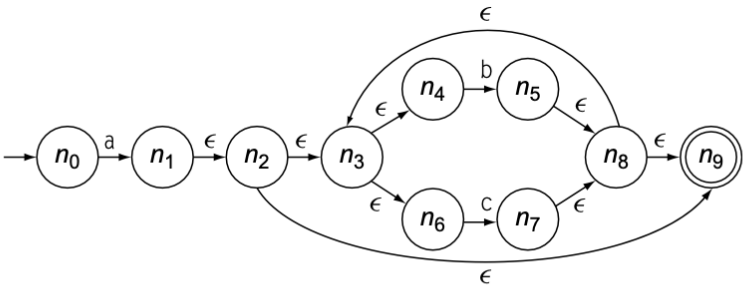
After the process of converting the AST into NFA fragment(s) is done, the fragment will be converted into NFA (**src/nfa_fragment.rs : fn to_nfa**) by reprocessing the state transitions into hash sets.

The process of converting the regular expression into an equivalent NFA is based on techniques discribed in the books *Introduction to the Theory of Computation* and *Engineering a Compiler*. Here are a few examples of the illustrated conversions:



DFA

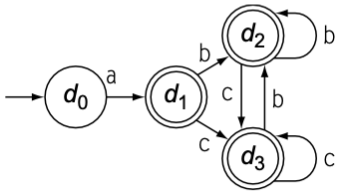
The next step is to convert the NFA into a DFA (**src/dfa.rs**). This is done by using a technique called the *subset construction (**src/nfa.rs : fn to_dfa**)*. The idea is to collect every unique set of states that can exist at the same time and make it into a DFA state. The transitions from that states will be the set of all transitions from all of the states in the set. Here is an illustration from the book *Engineering a Compiler* that describes the process



(a) NFA for "$a(b \mid c)^*$" (With States Renumbered)

Set Name	DFA States	NFA States	ϵ-closure(Delta(q,*))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3

(b) Iterations of the Subset Construction



(a) Resulting DFA

Engineering a Compiler, 2nd Edition

Recognizer

The recognizer (**src/dfa.rs**) will use the DFA to accept or reject input strings. This will happen by reading the charecters of the string one by one and making the corresponding transitions along the DFA if they exist. Lack of an available transition will result to a transition to an "eternal" rejecting state, that is, a rejecting state that only has transitions to itself. Once the recognizer has consumed the whole string, it will check if the final state is an accept state and return the answer as a boolean value.

Sources

1. Introduction to the Theory of Computation, Third Edition by Michael Sipser
2. Engineering a Compiler, 2nd Edition by Keith D. Cooper and Linda Torczon