Project 1: Regular Expression Matcher

Formal Languages

HAUTEFAYE Corentin

April 7, 2025

Contents

1 Introduction

1.1 Problem Statement

Within the scope of this class on *Formal Languages*, the first assignment I had to work on was to implement a regular expression matcher in Java, C or Python. The goal of this project was to understand how such a program works without using any kind of extern library.

Our program must take two strings as parameters, in other words one for the regular expression and another for the word to check. At the end, this has to return whether such word is spanned by that regular expression or not. This program should handle basic operations on regular expressions ; the latter is described in the following subsection.

1.2 Notations

Let r and s be two regular expressions. Then, we have the following:

- r and s are called *atomic expressions*.
- (r) is a regular expression.
- rs denotes the *concatenation* of r and s.
- $r|s$[1] denotes an *alternative* (or union) between r and s.
- $r*$ denotes the *transitive closure* of r.
- $r+$ denotes the *positive closure* of r.
- $r?$ denotes a *choice* to put r or not.

From those operators, we can draw the following properties:

- $r+ = rr*$.
- $r? = r|\epsilon$.
- A single character has the highest precedence.
- Closure (such as *, + or ?) has a lower precedence than a character.
- Concatenation has a lower precedence than closure.
- Union has a lower precedence than concatenation.
- Union has the lowest precedence.
- Grouping regular expressions with parenthesis overrides precedence.

2 Algorithms & Implementation

2.1 Choices

I have chosen to implement this project in C99 as I am much more comfortable in using that language. Moreover, I only needed structures and not classes to program, thus the choice was quite quick.

[1] Please see section 2.1

I have also decided to replace the notation used in the instructions for union (*i.e for all r and s two regular expressions, $r+s$ was replaced by $r|s$*) for several reasons. First of all, I did not like the idea of having the same character '+' used for different purposes as building the syntax tree would have been more difficult. Then, that rule implied that even writing a very simple regular expression using an operator with two different precedences could lead to difficulties and could be ambiguous. For instance, let r, s and t be three regular expressions. The following $r + s + t$ is not entirely clear, except if we put parenthesis but this operation is costly to write and to parse (*especially for top-down recursive algorithms*). Thus, we could interpret it as $(r + s + t)$, $(r+)(s+)(t+)$, $(r+)(s + (t)+)$ and so on... Whereas, $r|s|t$ is not ambiguous at all.

The design of this project can be described in three steps. First, we need to parse a regular expression so that it can be read properly by the program afterwards. I decided to parse it into an abstract syntax tree. Then, I had several choices I could make to use that tree. Thus, I decided to reproduce what I would have done, that is turn the previous tree into an automaton. Once that step is done, matching the given word is pretty easy as the program just has to follow each state in the automaton so that it would arrive in an accepting state or not.

Finally, this project has been compiled on Windows with CLion. It should be able to compile also on any Linux distribution with GCC installed.

2.2 Parser

Let $\mathcal{A}$ be a finite set of characters that we will call alphabet. Using the properties in section 1.2, we can formally define a grammar $G = (\mathcal{A} \cup \{(,), |, *, +, ?\}, \{X, C, R, A\}, \rightarrow, X)$ such that we have the following axioms : $\begin{cases} X & \rightarrow C\ ('|'\ C)^* \\ C & \rightarrow R^+ \\ R & \rightarrow A\ ('*' \mid\ '+' \mid '?')^* \\ A & \rightarrow\ '('X')' \mid c \in \mathcal{A} \end{cases}$ where characters between simple quotes are terminal.

It is now easy to write a parser that follows these rules. However, we do need a structure to hold the data parsed by this automaton. Therefore, we will use an abstract syntax tree defined as follows: `Node` : $\begin{cases} \texttt{type} & : \texttt{unsigned integer} \\ \texttt{value} & : \texttt{character} \\ \texttt{left} & : \texttt{Node} \\ \texttt{right} & : \texttt{Node} \end{cases}$.

Furthermore, we assume that a node can have one of the following types:

- Character
- Concatenation
- Union
- Star (*transitive closure*)
- Star (*transitive closure*)
- Optional (*choice*)

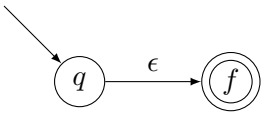
2.3 Non-Deterministic Finite Automaton

We use the same notations as in the previous subsection. For any abstract syntax tree, it is possible to create a non-deterministic finite automaton step by step. This can be proven with a mathematical induction.

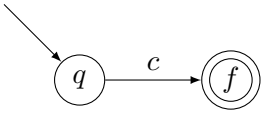
Thus, we need to define trivial cases as well as induction cases. Let q be the initial state and f be the final state.

Trivial cases

For the empty word ϵ, we have:



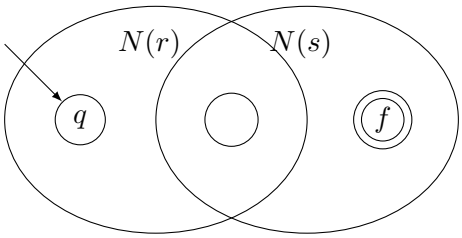
Let $c \in \mathcal{A}$. For the regular expression c, we have:



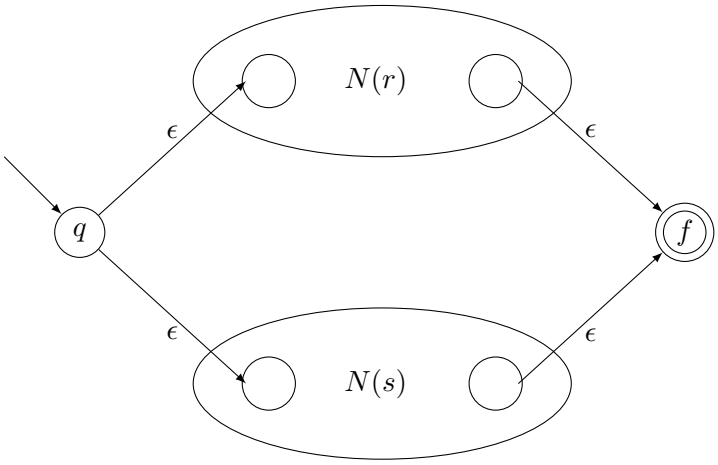
Induction

Let r and s be two regular expressions. Without loss of generality, we denote an automaton of r by $N(r)$.

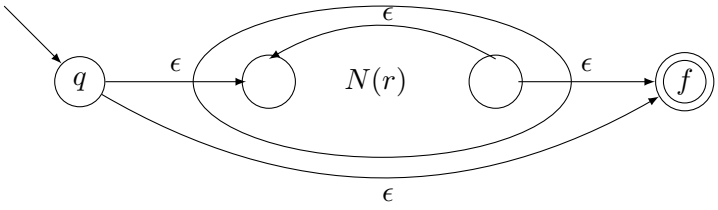
For the regular expression rs, we put $N(r)$ and $N(s)$ right after the other, whence:



For the regular expression $r|s$, we have two ϵ-transitions towards $N(r)$ and $N(s)$, whence:



For the regular expression r^*, we have:



Using the properties in section 1.2, automata for r^+ and $r^?$ can be built easily using the previous rules.

Thus these rules yield that an automaton in this context has both starting and accepting states ; the latter are unique. Moreover, a state allows no more than two transitions. By convention, we choose to represent the ϵ-transition by using the null-character, denoted by \0 or 0. If a state does not have ϵ as a transition, then there is only one transition which is not empty.

Thus, we can define the following structures : `State` : $\begin{cases} \texttt{id} & : \texttt{unsigned integer} \\ \texttt{transition} & : \texttt{character} \\ \texttt{out1} & : \texttt{State} \\ \texttt{out2} & : \texttt{State} \end{cases}$

and `Fragment` : $\begin{cases} \texttt{start} & : \texttt{State} \\ \texttt{final} & : \texttt{State} \end{cases}$.

Furthermore, for an easier implementation, we assume that there will be no more than a certain amount of states in our automaton. Here, we have chosen 1024.

2.4 Matching

The algorithm used to match a word with a given non-deterministic finite automaton as defined above is not very complex. However, we do need a structure to hold all states from a given state, so that if a chosen "path" does not succeed, the program can go back and try another state. Thus, we have: `StateSet` : $\begin{cases} \texttt{states} & : \texttt{array<State>[1024]} \\ \texttt{count} & : \texttt{unsigned integer} \end{cases}$. This structure would be equivalent to a stack of which `count` is the top.

Algorithm 1: Add a state to look at if it has not already been visited

Input: E: StateSet, Q: State, v: Array<boolean>
Output: $\emptyset$
Procedure *add_state(E: StateSet, Q: State, v: Array<boolean>)*
Begin
  If $E \neq \emptyset$ **AND** $Q \neq \emptyset$ **AND NOT** $v[id(Q)]$ **Then**
    $v[\text{id}(Q)] \leftarrow$ True;
    $(\text{states}(E))\,[\text{count}(E)] \leftarrow Q$;;
    $\text{count}(E) \leftarrow \text{count}(E) + 1$;
    If *transition*$(Q) = {}'\backslash 0'$ **Then** `/* ϵ-transitions */`
      add_state$(E, \text{out1}(Q), v)$;
      add_state$(E, \text{out2}(Q), v)$;
    End If
  End If
End

Algorithm 2: Look at each state from the current state set and try to move to another state

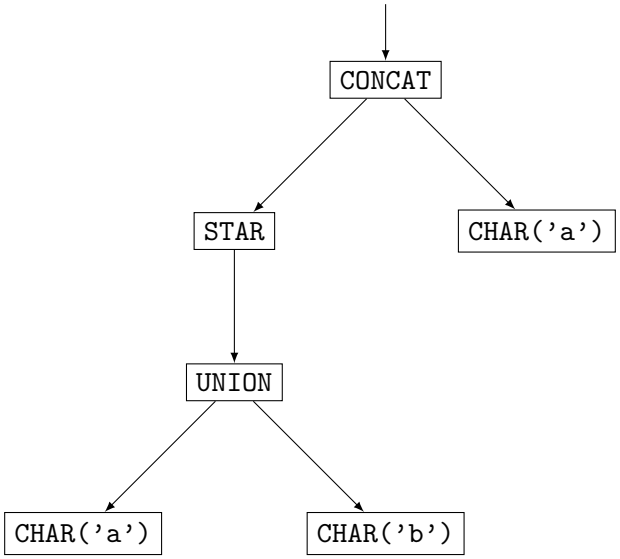
Input: *Current*: StateSet, *c*: character, *Next*: StateSet
Output: $\emptyset$
Procedure *step(Current: StateSet, c: character, Next: StateSet)*
Begin
    count(*Next*) $\leftarrow 0$;
    v : Array<boolean>[1024] $\leftarrow$ {False};
    i : unsigned integer $\leftarrow 0$;
    n : unsigned integer $\leftarrow$ count(*Current*);
    For i **from** *0* **to** n **Do**
        s : State $\leftarrow$ states(*Current*)[i];
        If *transition*(s) $= c$ **AND** *out1*(s) $\neq \emptyset$ **Then**
            add_state(*Next*, out1(s), v);
        End If
    End For
End

Algorithm 3: Matching a string with a given automaton

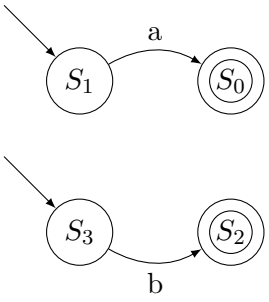
Input: Q: State, F: State, S: string
Output: Boolean
Function *match(Q: State, F: State, S: string) : boolean*
Begin
    Current : StateSet;
    Next : StateSet;
    count(*Current*) $\leftarrow 0$;
    v : Array<boolean>[1024] $\leftarrow$ {False};
    add_state(*Current*, Q, v);
    i : unsigned integer $\leftarrow 0$;
    While $S[i] \neq$ '\0' **Do**
        step(*Current*, $s[i]$, *Next*);
        `// Shallow copy`
        Current $\leftarrow$ *Next*;
        $i \leftarrow i + 1$;
    End While
    n : unsigned integer $\leftarrow$ count(*Current*);
    For i **from** *0* **to** n **Do**
        If *states*(*Current*)[i] $= F$ **Then**
            match $\leftarrow$ True;
        End If
    End For
    match $\leftarrow$ False;
End

3 Example

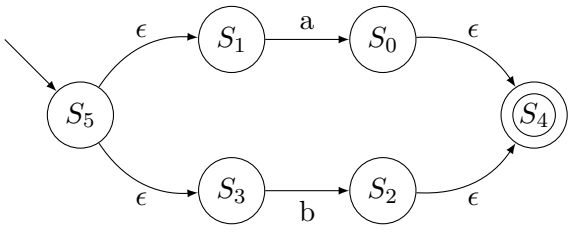
Let's consider the following regular expression $r = (a|b)^*a$. Thus, here $\mathcal{A} = \{a, b\}$. Let's first build the abstract syntax tree. Using the rules described in section 1.2, we get:



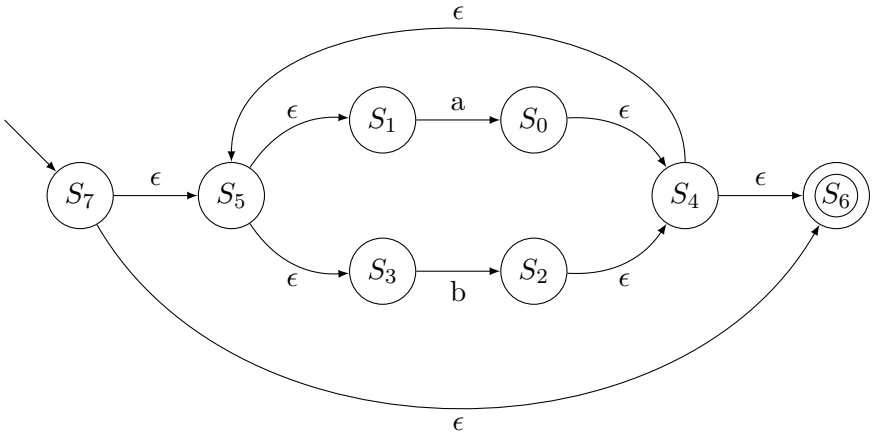
We can construct our non-deterministic finite automaton step by step. We will denote each state by $(S_i)_{i\in\mathbb{N}}$. First, we have two trivial cases:



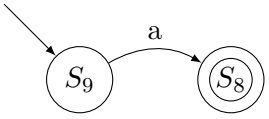
Afterwards, we have an union of the two previous expressions. Thus, we get:



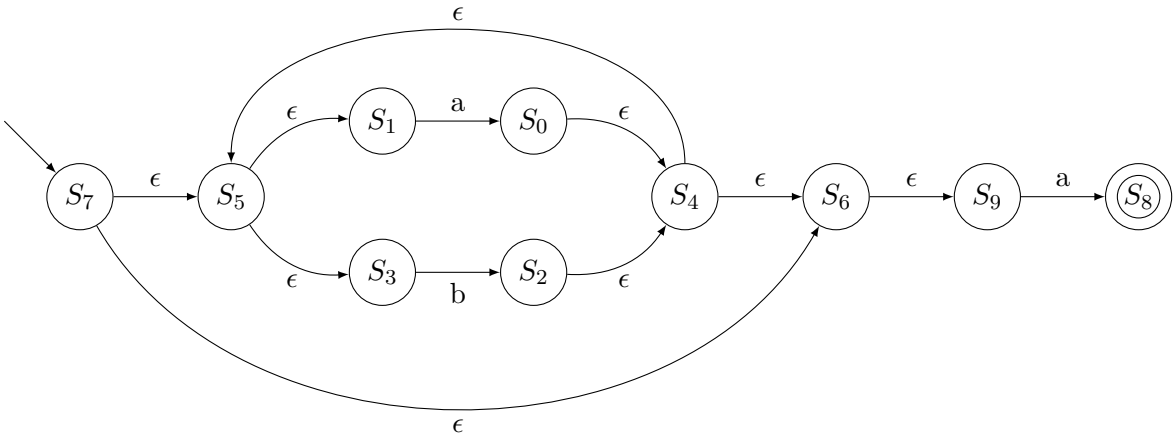
Then, we need to apply a transitive closure onto the previous automaton. Thus:



Next, we have another atomic expression:



Finally, we need to concatenate the last two automata to get r, whence:



Let $w_1 = aaab$, $w_2 = ccabacc$ and $w_3 = a$. Let's find whether there exists an accepting configuration for these words.

Study of w_1 Going to S_6 does not yield an accepting configuration as we would have $(w_1, S_7) \vdash (w_1, S_6) \vdash (w_1, S_9) \vdash (aab, S_8) \nvdash (\epsilon, S_8)$. Thus, we must have $(w_1, S_7) \vdash (w_1, S_5)$. We can easily apply each transition to get $(\epsilon, S_4) \vdash (\epsilon, S_6) \vdash (\epsilon, S_9)$. However, we cannot go from S_9 to S_8 as it is not an ϵ-transition. Moreover, all states have been studied. Therefore, $\boxed{w_1 \text{ is not accepted.}}$

Study of w_2 First, going to S_5 is useless since both S_1 and S_3 do not have a transition for character 'c'. Thus, we have $(w_2, S_7) \vdash (w_2, S_6) \vdash (w_2, S_9)$. Yet, S_9 does not have a transition for character 'c'. Since S_9 is not terminal, $\boxed{w_2 \text{ is not accepted.}}$

Study of w_3 Going to state S_5 would again lead to (ϵ, S_9) which is not accepting. Thus, $(w_3, S_7) \vdash (w_3, S_6) \vdash (w_3, S_9) \vdash (\epsilon, S_8)$. Since S_8 is final and there is no letter left to proceed, we get that $\boxed{w_3 \text{ is accepted.}}$

Let's now try to run the program with r and w_1, whence:

```
======================
Syntax tree for regex: "(a|b)*a"
----------------------
ROOT
|_ CONCAT
    |- STAR
    |   |_ UNION
    |       |- CHAR('a')
    |       |_ CHAR('b')
    |_ CHAR('a')
======================
Non-Deterministic Finite Automaton:
----------------------
State 7: \0 -> 5, 6
State 6: \0 -> 9
State 9: 'a' -> 8
State 8: \0
State 5: \0 -> 1, 3
State 3: 'b' -> 2
State 2: \0 -> 4
State 4: \0 -> 5, 6
State 1: 'a' -> 0
State 0: \0 -> 4
======================
Matching word "aaab":
----------------------
Current character: 'a'
----------------------
|  Peeking State 7
|  Peeking State 5
|  Peeking State 1
|->Transition State 1 -> State 0
|  Peeking State 3
|  Peeking State 6
|  Peeking State 9
|->Transition State 9 -> State 8
----------------------
Current character: 'a'
----------------------
|  Peeking State 0
|  Peeking State 4
|  Peeking State 5
|  Peeking State 1
|->Transition State 1 -> State 0
|  Peeking State 3
|  Peeking State 6
```

```
|  Peeking State 9
|->Transition State 9 -> State 8
|  Peeking State 8
----------------------
Current character: 'a'
----------------------
|  Peeking State 0
|  Peeking State 4
|  Peeking State 5
|  Peeking State 1
|->Transition State 1 -> State 0
|  Peeking State 3
|  Peeking State 6
|  Peeking State 9
|->Transition State 9 -> State 8
|  Peeking State 8
----------------------
Current character: 'b'
----------------------
|  Peeking State 0
|  Peeking State 4
|  Peeking State 5
|  Peeking State 1
|  Peeking State 3
|->Transition State 3 -> State 2
|  Peeking State 6
|  Peeking State 9
|  Peeking State 8
----------------------

Word "aaab" is not accepted!
```

N.B: The procedure `print_tree` first prints the left, then the right nodes for any node in a given tree.

Here, we can see how the previous algorithms processed both the regular expression as well as the word. The final result is correct.

4 Feedback

4.1 Difficulties

At first, I had some difficulties because I tried to work only with deterministic finite automata. My functions were indeed very hectic, so I tried another approach.

I wanted to convert directly the string into a graph ; thus matching the word would have been very easy as we just had to run through the graph. This method worked but did not account for grouping with parenthesis.

Finally, I had the idea to use an abstract syntax tree, yet I did not know how to turn it into an automaton. So I started to sketch different "fragments" of automata that I could combine to represent a given regular expression. At the end, I realized that those rules were the same seen during the lecture on non-deterministic automata ; that was my clue. From there, I was able to implement everything as intended, except for the notation of the union which I decided to change.

4.2 Suggestions

This program is not perfect at all. For instance, I used two global variables whereas it is not recommended in C. The way parameters are passed to the program is not very versatile. For example, I could have chosen to use "short" parameters such as "-r" or "-s", but I did not think at that time that it really mattered. Moreover, as of now, due to the way arguments are passed to a program through the command prompt (*or terminal*), it is not possible to recognize words with whitespace within, unless it is done directly during the execution. Therefore, it may have been better to have a menu with the following choices:

1. Enter regular expression
2. Match a word
3. Quit

5 Conclusion

In fine, working onto this project was very interesting as I had never tried it before. I was able to define properly objects and properties on the theoretical plane, and to have a concrete application of the latter. The way non-deterministic finite automata are built was both intriguing and fun to understand and to discover by myself.