ECOTE - final project

Date:

06.2024

Semester:

2024L

Author and group:

Fengxi Zhao, 101

Subject:

Task 1 Aut1: Write a program able to input regular expressions, then constructing NFA using Thompson algorithm, and checking if input strings are generated by this expression (working on NFA).

I. General overview and assumptions

The project is to write a program that takes a regular expression as input, then convert it to NFA using Thompson's construction. After that it will take another input string and check if it works on this NFA.

II. Functional requirements

General algorithm:

Step 1: Get regular expression and check if there are mistakes like unpaired parentheses. Display detailed error message if not correct.

Step 2: With confirmation the regular expression is correct, the program generates NFA according to Thompson's construction. Display the result NFA.

Step 3: Get the string and check if it satisfies the NFA

Regular expression syntax:

1.Parentheses:

Opening Parenthesis:

Must be at the start of the expression or follow an operator ("|" or after a closed parenthesis ")").

Closing Parenthesis:

Cannot follow immediately after an operator, except after a * when it modifies a previous subexpression.

Must correspond to a previously opened parenthesis.

2.Kleene Star:

Can only follow a letter or a closed parenthesis. Note that if you want input “a***”, write it as “(((a)*)*)*”.

Cannot be immediately preceded by another “*” or an operator like “|”.

3.Or Operator:

Cannot be the first or last character in the expression.

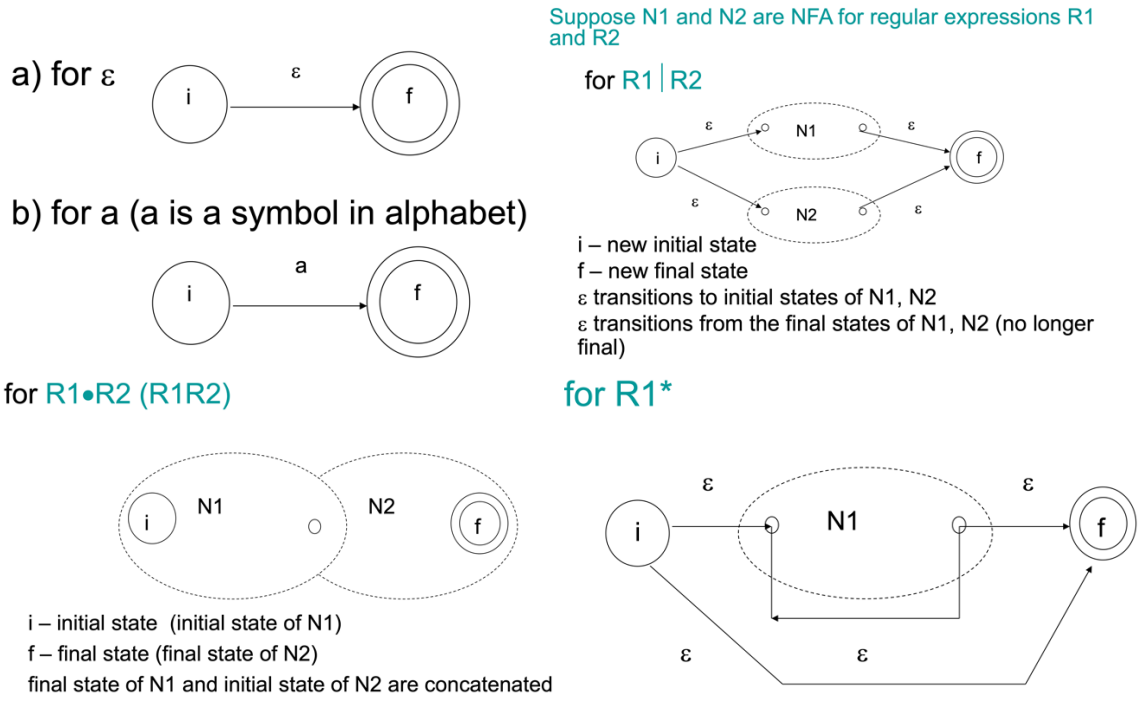
Must not follow another “|” directly.

Can follow a “*” if it is part of a valid subexpression.

4.Characters:

Only lowercase letters “a” to “z” are permitted.

Thompson’s construction:



Source: *https://studia.elka.pw.edu.pl/f-pl/24L/103A-CSCSN-ISA-ECOTE*

III. Implementation

General architecture:

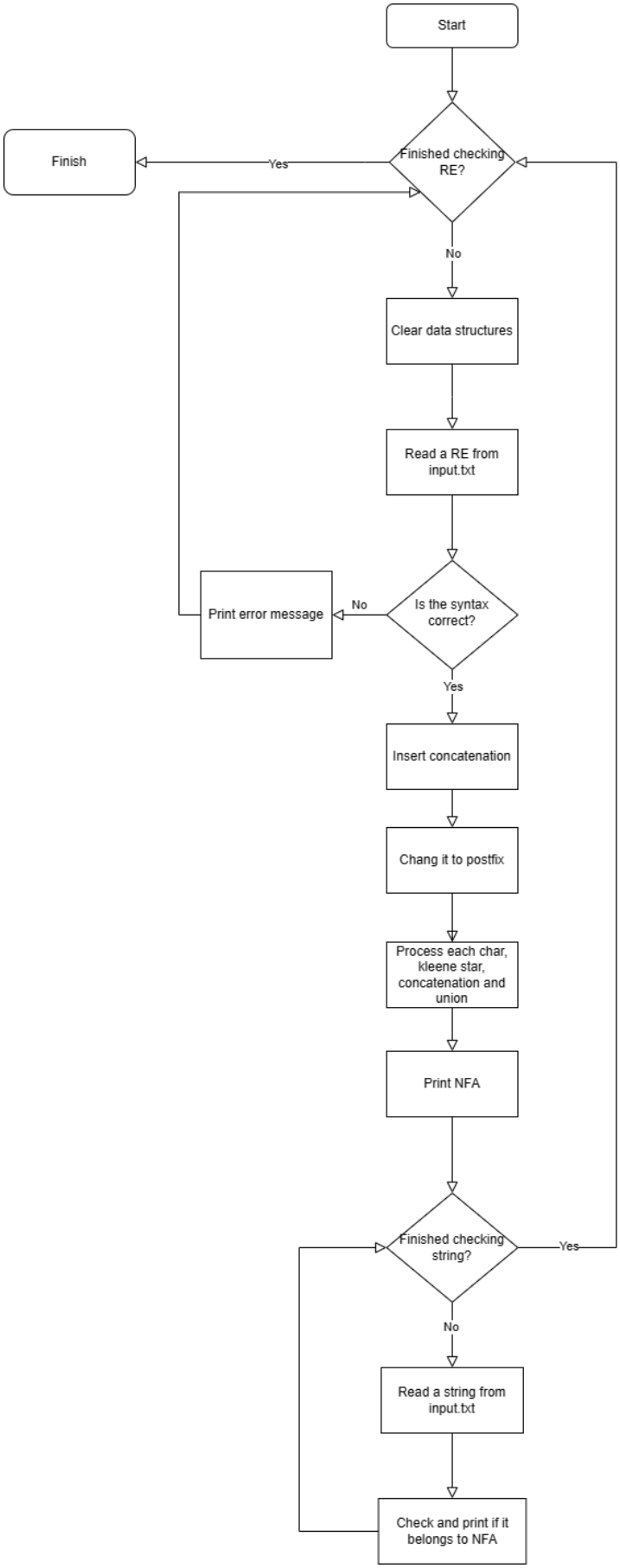
The program will be written in C++ and due to its simplicity, we don't need to specify classes. Struct and STL are enough.

The main function will be used to differ the regular expressions and the strings to be tested by calling different functions.

Solve function is to transfer regular expressions to NFA and re_match_check function is to check the strings.

The rest functions are just intuitively like in flowchart.

Flowchart:


Start
Finished checking RE?
Yes
Finish
No
Clear data structures
Read a RE from input.txt
Is the syntax correct?
No
Print error message
Yes
Insert concatenation
Chang it to postfix
Process each char, kleene star, concatenation and union
Print NFA
Finished checking string?
Yes
No
Read a string from input.txt
Check and print if it belongs to NFA

Module descriptions:

Check RE syntax and print error:

We can use stack to check the parentheses. Display the error message for different types of input errors.

Insert concatenation:

Insert "." into RE to divide characters. It reveals the relationship of concatenation. There are two examples:

b*a => b*.a

(ba)* => (b.a)*

Change RE to postfix notation:

The precedence is: Parentheses > "*" > "." > "|".

Postfix notation is a way to express the order we process the RE. Manually we can write the hidden parentheses like: a.b* => (a.b*) =>ab*. to find the postfix notation.

In program we can use stack to deal with it: Read each character from RE, if there is "(" we start to push them into stack until ")" been read. Then pop all characters inside of parentheses since we need to process them first. Append them to the result. For the rest we simply compare the precedence and append them to the result.

Postfix notation to NFA:

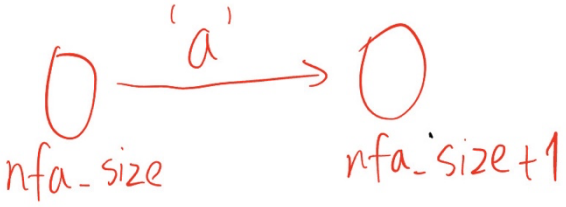
After it becomes postfix notation, all the precedence information has been included. Now we can simply process it one character by one character based on what it is.

Process character:

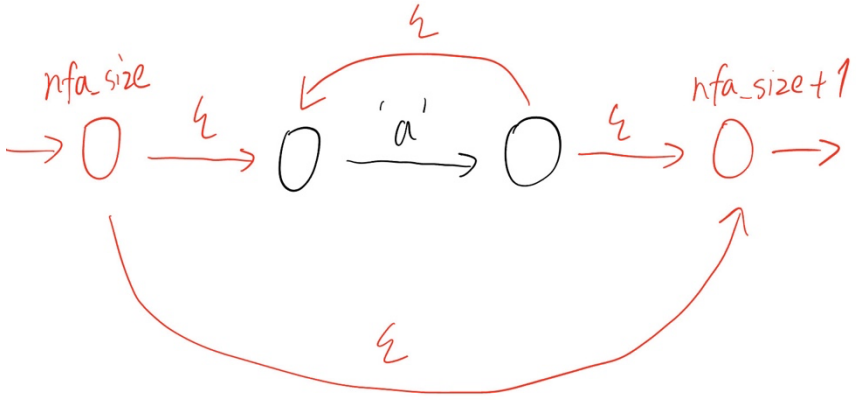
We can use an integer nfa_size as the name of the states, it will expend as the progress goes. We also need a struct to record the edges(transformation between states). Struct of edges should include the transition input(from 'a' to 'z' and epsilon), and whether it goes to the final state. A vector of this struct will be built.

An extra stack is necessary to record the name of the states.

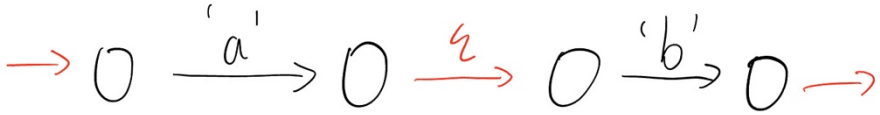
In this case, we just need to add one edge. Then push the states to stack and wait for further process.



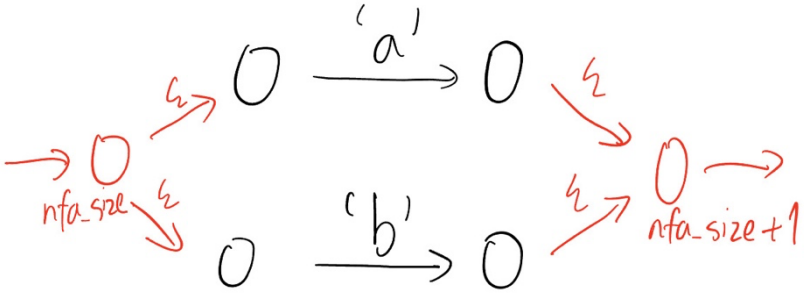
Process Kleene star:
Pop 2 states from the stack, then add 4 edges.



Process concatenation:
Pop 4 states, then add 1 edge.



Process union:
Pop 4 states, then add 4 edges.



Print NFA:

A vector of a new struct of production will be needed for the answer. We traverse the vector of struct of edge to store it into answer.

Check if the string belongs to NFA:

We can check the string based on regular expression.

Clear data structure:

Clear the struct and other information to get ready for next RE.

Input/output description

input.txt file should include the regular expressions and the strings to test.
Regular expressions begin with @. For each regular expression there are following test strings without @. Test strings are not necessary.

output.txt should include postfix notation, the transition functions of NFA, the start state and the final state. It will also include the information about whether the strings matches the previous regular expression. If a regular expression has syntax problem, neither it will be transformed to NFA nor the following strings will be checked. It will only display the syntax error.

IV. Functional test cases

input.txt:

@a*b
aaab
bbaa

@(ab)*ab*
ababa
abababab

@a*b
aaabb
baa

@a*|b*

@a|b*

@(ab)*b

@ba*|(ab)

@((a*)*)*

@(a*|b*)*

@a(ac

@b)
bba

@Aaab*

@a)(ab*

@a***

output.txt:

.....
Regular expression: a*b
Postfix expression: a*b.
Regular expression to NFA:
States = {A, B, C, D, E, F};
Inputs = {a, b};
Transition functions: f(A, a) = B, f(B, ε) = A, f(B, ε) = D, f(C, ε) = A, f(C, ε) = D, f(D, ε) = E, f(E, b) = F;
Initial state = C;
Final state = {F};

String "aaab" matches: Yes
String "bbaa" matches: No
.....
Regular expression: (ab)*ab*
Postfix expression: ab.*a.b*.
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H, I, J, K, L};
Inputs = {a, b};

Transition functions: f(A, a) = B, f(B, ε) = C, f(C, b) = D, f(D, ε) = A, f(D, ε) = F, f(E, ε) = A, f(E, ε) = F, f(F, ε) = G, f(G, a) = H, f(H, ε) = K, f(I, b) = J, f(J, ε) = I, f(J, ε) = L, f(K, ε) = I, f(K, ε) = L;
Initial state = E;
Final state = {L};

String "ababa" matches: Yes
String "ababab" matches: Yes
.....
Regular expression: a*b
Postfix expression: a*b.
Regular expression to NFA:
States = {A, B, C, D, E, F};
Inputs = {a, b};
Transition functions: f(A, a) = B, f(B, ε) = A, f(B, ε) = D, f(C, ε) = A, f(C, ε) = D, f(D, ε) = E, f(E, b) = F;
Initial state = C;
Final state = {F};

String "aaabb" matches: No
String "baa" matches: No
.....
Regular expression: a*|b*
Postfix expression: a*b*|
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H, I, J};
Inputs = {a, b};
Transition functions: f(A, a) = B, f(B, ε) = A, f(B, ε) = D, f(C, ε) = A, f(C, ε) = D, f(D, ε) = J, f(E, b) = F, f(F, ε) = E, f(F, ε) = H, f(G, ε) = E, f(G, ε) = H, f(H, ε) = J, f(I, ε) = C, f(I, ε) = G;
Initial state = I;
Final state = {J};

.....
Regular expression: a|b*
Postfix expression: ab*|
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H};
Inputs = {a, b};
Transition functions: f(A, a) = B, f(B, ε) = H, f(C, b) = D, f(D, ε) = C, f(D, ε) = F, f(E, ε) = C, f(E, ε) = F, f(F, ε) = H, f(G, ε) = A, f(G, ε) = E;
Initial state = G;
Final state = {H};

.....

Regular expression: (ab)*b
Postfix expression: ab.*b.
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H};
Inputs = {a, b};
Transition functions: f(A, a) = B, f(B, ε) = C, f(C, b) = D, f(D, ε) = A, f(D, ε) = F, f(E, ε) = A, f(E, ε) = F, f(F, ε) = G, f(G, b) = H;
Initial state = E;
Final state = {H};

.....
Regular expression: ba*|(ab)
Postfix expression: ba*.ab.|
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H, I, J, K, L};
Inputs = {a, b};
Transition functions: f(A, b) = B, f(B, ε) = E, f(C, a) = D, f(D, ε) = C, f(D, ε) = F, f(E, ε) = C, f(E, ε) = F, f(F, ε) = L, f(G, a) = H, f(H, ε) = I, f(I, b) = J, f(J, ε) = L, f(K, ε) = A, f(K, ε) = G;
Initial state = K;
Final state = {L};

.....
Regular expression: ((a*)*)*
Postfix expression: a***
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H};
Inputs = {a};
Transition functions: f(A, a) = B, f(B, ε) = A, f(B, ε) = D, f(C, ε) = A, f(C, ε) = D, f(D, ε) = C, f(D, ε) = F, f(E, ε) = C, f(E, ε) = F, f(F, ε) = E, f(F, ε) = H, f(G, ε) = E, f(G, ε) = H;
Initial state = G;
Final state = {H};

.....
Regular expression: (a*|b*)*
Postfix expression: a*b*|*
Regular expression to NFA:
States = {A, B, C, D, E, F, G, H, I, J, K, L};
Inputs = {a, b};
Transition functions: f(A, a) = B, f(B, ε) = A, f(B, ε) = D, f(C, ε) = A, f(C, ε) = D, f(D, ε) = J, f(E, b) = F, f(F, ε) = E, f(F, ε) = H, f(G, ε) = E, f(G, ε) = H, f(H, ε) = J, f(I, ε) = C, f(I, ε) = G, f(J, ε) = I, f(J, ε) = L, f(K, ε) = I, f(K, ε) = L;
Initial state = K;
Final state = {L};

.....
Regular expression: a(ac
Unmatched '(': Not all opening parentheses have a matching closing parenthesis.
.....
Regular expression: b)
Unmatched ')': No corresponding opening parenthesis.
.....
Regular expression: Aaab*
Invalid character: Only lowercase letters 'a' to 'z' are allowed.
.....
Regular expression: a)(ab*
Unmatched ')': No corresponding opening parenthesis.
.....
Regular expression: a***
Invalid placement of '*': Consecutive stars are not allowed.