10/1/2020

CS 317 - Automata and Formal Languages
Project – Regular Expression to Finite Automata
Due Sunday, October 18

150 points

Introduction

For this programming project, you will construct a regular expression engine using C (preferable) or Java to implement your program. If you use Java, your program must run without any external libraries (default java libraries only). Graders must be able to easily compile and run your code. The "engine" involves creating Nondeterministic Finite Automata (NFA) from user supplied regular expressions in postfix notation (see below).

Regular Expressions

Regular expression semantics:

- **Σ = {a, b, c, d, e}** and expressions are over **Σ^*** this includes **ε**. For ease of processing we will write E for ε.
- **r_1|r_2** is union (this is **$r_1 \cup r_2$**, either expression r_1 or r_2, the | is easier to type/process)
- **r_1r_2** is concatenation (expression r_1 followed by r_2 also written **$r_1 \circ r_2$**). For ease of processing we will write **r_1&r_2** for concatenation.
- **r*** Kleene closure (expression r zero or more times)
- **(r)** parenthesized expression (postfix notation will not use parenthesizes)

Operators are listed in increasing order of precedence, lowest is |. Your program should check for well formed regular expressions and give appropriate error messages if the input is incorrect.

Postfix Regular Expressions

Most people who use HP calculators are used to postfix notation for arithmetic expressions. For example, the infix expression

(3 + (4 * 8)) + ((6 + 7)/5)

is expressed as

3 4 8 * + 6 7 + 5 / +

in postfix notation. In prefix form, this expression would be

+ + 3 * 4 8 / + 6 7 5

Both prefix and postfix notation are nice because parentheses are not needed since they do not have the operator-operand ambiguity inherent to infix expressions. Postfix expressions are also easy to parse (which is why we are discussing them here). Later in the course, when we discuss context-free languages, we will revisit the problem of parsing infix expressions. Using the same idea for regular expressions, the infix regular expression

$((a\cup b)^*aba^*)^*(a\cup b)(a\cup b)$

can be represented as

$((a|b)^*\circ a\circ b\circ a^*)^*\circ(a|b)\circ(a|b)$

where $\circ$ is added for concatenation. This can be changed to

ab|*a&b&a*&*ab|&ab|&

in postfix notation. Notice we removed the need for parentheses but added the & operator for concatenation. Your program will work with regular expressions in their postfix form.

10/1/2020

Postfix Regular Expression to NFA

We will convert a postfix regular expression into an NFA using a stack, where each element on the stack is an NFA. The input expression is scanned from left to right. When this process is completed, the stack should contain exactly one NFA. We construct NFAs based on the inductive rules below.

Converting regular expressions into FAs

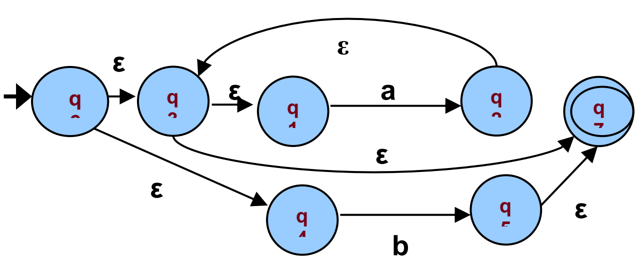
- **Rule 1:** There is a FA that accepts any symbol of Σ and there is a FA that accepts ε.
  - If x is in Σ then give a FA that accepts x
  - Give a FA that accepts ε.
- **Rule 2:** Given FA_1 that accepts regular expression r_1 and FA_2 that accepts regular expression r_2 then make FA_3 that accepts $r_1 \cup r_2$. Add a new start state s and make a ε-transition from this state to the start states of FA_1 and FA_2. Add a new final state f and make a ε-transition to this state from each of the final states of FA_1 and FA_2.
- **Rule 3:** Given FA_1 that accepts regular expression r_1 and FA_2 that accepts regular expression r_2 then make FA_3 that accepts $r_1 \circ r_2$. Add a ε-transition from the final state of r_1 to the start state of r_2. The start state of FA_3 is the start state of FA_1 and the final state of FA_3 is the final state of FA_2. You will have to think about it, but I do not think you will have multiple final states in FA_1.
- **Rule 4:** Given FA_1 that accepts regular expression r then make a FA_2 that accepts r*. Add a new start state ***s*** and make a ε-transition from this state to the start state of FA_1. Make a ε-transition from the final state of F_1 to the new start state ***s***. The final states of FA_1 are no longer final and ***s*** is the final state of FA_2.

Input/Output Conventions

Input: Your program should read from a text file that contains a list of regular expression in postfix form (the infix form will not be included). The file will be setup so that there is one regular expression per line. See the posted test file.
Input symbols: a, b, c, d, e, |, &, * and E (for ε). We will skip the empty set.
If you want to include capital S for Σ you may but I will not test your program on this since $\Sigma = (a \cup b \cup c \cup d \cup e)$.

Output: You can use any internal data structures that you want to represent the NFA you are building. When your program is done please print the NFA out as a table or as a list of transitions. **Make sure the grader knows which you have picked.** For example, input **a*b|** (which is postfix for $a^* \cup b$ might produce the NFA below. Only print a table or a list of transitions as output.

(q1, a) → q2
(q2, **ε)** → q3
(q3, **ε)** → q1, q7
(q4, **b)** → q5
(q5, **ε)** → q7
S (q6, **ε)** → q3, q4
F (q7, **ε)**



	a	b	ε
q1	q2		
q2			q3
q3			q1, q7
q4		q5	
q5			q7
q6 S			q3, q4
q7 F			

10/1/2020

Postfix Regular Expressions

These are samples of regular expression in both infix and postfix form. Only the postfix list would be in an input file for your program.

<table>
<tr><td>1.</td><td>a∪b</td><td>a|b</td><td>ab|</td></tr>
<tr><td>2.</td><td>a∪ε</td><td>a|E</td><td>aE|</td></tr>
<tr><td>3.</td><td>ab</td><td>ab</td><td>ab&</td></tr>
<tr><td>4.</td><td>a*</td><td>a*</td><td>a*</td></tr>
<tr><td>5.</td><td>(a∪b)*</td><td>(a|b)*</td><td>ab|*</td></tr>
<tr><td>6.</td><td>a*∪b</td><td>a*|b</td><td>a*b|</td></tr>
<tr><td>7.</td><td>(a∪b)*b</td><td>(a|b)*b</td><td>ab|*b&</td></tr>
<tr><td>8.</td><td>aba*(a∪b)b</td><td>aba*(a|b)b</td><td>ab&a*&ab|&b&</td></tr>
<tr><td>9.</td><td>((a∪b)*aba*)*(a∪b)(a∪b)</td><td>((a|b)*aba*)*(a|b)(a|b)</td><td>ab|*a&b&a*&*ab|&ab|&</td></tr>
<tr><td>10.</td><td>(aba*(a∪b))*b</td><td>(aba*(a|b))*b</td><td>ab&a*&ab|&*b&</td></tr>
<tr><td>11.</td><td>((aba*(a∪b))*b)*</td><td>((aba*(a|b))*b)*</td><td>ab&a*&ab|&*b&*</td></tr>
<tr><td>12.</td><td>((aba*(a∪b))*b)*(a∪b)*b</td><td>((aba*(a|b))*b)*(a|b)*b</td><td>ab&a*&ab|&*b&*ab|*&b&</td></tr>
</table>

Basic Code Outline

```
while (not end of postfix expression) {
        c = next character in postfix expression;
        if (c == '&') {
                nFA2 = pop();
                nFA1 = pop();
                push(NFA that accepts the concatenation of L(nFA1) followed by L(nFA2));
        } else if (c == '|') {
                nFA2 = pop();
                nFA1 = pop();
                push(NFA that accepts L(nFA1) | L(nFA2));
        } else if (c == '*') {
                nFA = pop();
                push(NFA that accepts L(nFA) star);
        } else
                push(NFA that accepts a single character c);
}
```

Data Structure Idea

Below is one suggestion on how to implement your code, but you do not have to use this.
For example, in C you might have

1. A structure that represents an NFA containing the number for the start state, the number for the accept state and a pointer to a list of transitions
   NFA

start state	integer
accept state	integer
transition list	pointer to a list of transitions for the NFA

2. A second structure that represents transitions containing a number for the first state, a number for the second state, the symbol that goes from the first state to the second and last, and a pointer to a list of transitions. This could represent (q1, a) → q2.
   Transition

state 1	integer
state 2	integer
symbol	integer too if a = 1, b = 2, c = 3, etc.
transition list	pointer to a list of transitions for the NFA

With this type of data structure what you need to push and pop from the stack is the NFA structure because the list of transitions for the NFA is connected to it and can have any length including null.

Submitting your solution

Archive all of your source code, documentation, makefile, README, etc. in a single compressed file and submit it electronically via Blackboard. Makefile is a must for compiled languages like C, C++, Java, etc. Name your compressed file: **firstname.lastname.zip**.

Your project archival should include a text file named **README** with the following information:

1. Your name and email address

2. List of all the files in your archive with a one line description of each file

3. Compile instructions – a precise description of how to build your code

4. Run instructions – a precise description of how to run your code

Graders will deduct points if they need to communicate with you in order to test your code. The regular late policy applies – 10% off for each late day.