CS131 Compilers: Writing Assignment 1
Due Saturday, March 13, 2021 at 23:59pm

Feiran Qin - 2019533161

This assignment asks you to prepare written answers to questions on regular languages, finite automata, and lexical analysis. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work and you should indicate in your submission who you worked with, if applicable. Written assignments are turned in at the start of lecture. You should use the Latex template provided at the course web site to write your solution and use the *tikz* package to draw automata.

I worked with: (Haotian Jing, 2018533233), (Yining Zhang,2019533103), (Songhui Cao,2018533156)

1. ($1 \times 3 = 3$ **pts**) For each of the follow prompts, write any non-empty sentence:

   (a) Name one reason why you would like to learn in this class.

   Understanding Compilers can serve for the programming of Underlying Code, which is important in learning computer science.

   (b) Write a question you would like the professor to answer 1 on any topic, from personal opinions to the class material and the teaching assistant to answer 1 in the discussion part.

   I want to know more about the lowring optimization of the IR code.

   (c) What do you expect from this class.

   The ability of system programming and the knowledge of Programming Language.

2. ($1 \times 3 = 3$ **pts**) Briefly explain why the following language do not have regular expression.

   (a) The set of all finite strings over the alphabet $\Sigma = \{a, b\}$ that the number of a is bigger than b.

   Regex is used for match strings. It can't be used for counting numbers of specific character and do logical calculations.
   Suppose $s = a^{p+k}B^p$, which is in $s = uv^nw$ where $uv^n = a^{p+k}$. When $n = 1$, the number of a can be smaller than b. Which is wrong.

   (b) The set of all finite palindromic strings over the alphabet $\Sigma = \{a, b\}$ that remains the same when its digits are reversed.

   Regex expression can't determine the Symmetry point of the palindromic expression.
   Suppose $s = abcddcba$, which in in $s = uv^nw$ where $u = a$, $v = bcd$, $n = 1$, $w = dcba$. It's obvious that when $n <> 1$. Which is wrong.

(c) Grammar correct C++ program.

Regex expression can't do the jobs of Syntax Analyser and Parser.

3. ($2 \times 2 = 4$ **pts**) Write regular expressions for the following languages over the alphabet $\Sigma = \{a, b, c\}$:

(a) L_1: The set of all finite strings that the first appearing a is ahead of the first appearing b.

$(c)^*(a)(a+c)^*(a+b+c)^*$

(b) L_2: The set of all finite strings containing $a's$ for even times.

$(b+c)^*(a(b+c)^*a)^*(b+c)^*$

4. ($2 \times 2 = 4$ **pts**) Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$:

(a) L_3: The set of all finite strings that is Binary number and bigger than 101001.

$(0^*)1|(0|1)|(0|1)|(0|1)|(0|1)|(0|1)|(0|1)^+)$
$|(0^*)10101(0|1)$
$|0^*1011(0|1)(0|1)$
$|0^*11(0|1)(0|1)(0|1)(0|1)$

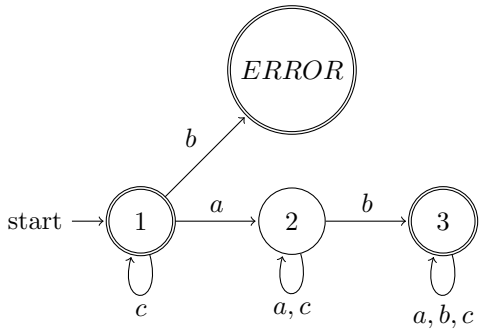
(b) L_4: The set of all finite strings containing at least two 0's and at most three 1's

$(0^+0^+)|$
$(0^+0^+10*)|(0^+10^+)|(0*10^+0^+)|$
$(0^+0^+10*10*)|(0*10^+0^+10*)|(0*10*10^+0^+)|(0^+10^+10*)|(0^+10*10^+)|(0*10^+10^+)|$
$0^+0^+10*10*10*|0*10^+0^+10*10*|0*10*10^+0^+10*|0*10*10*10^+0^+|0^+10^+10*10*|0^+10*$
$10^+10*|0^+10*10*10^+|0*10^+10^+10*|0*10^+10*10^+|0*10*10^+10^+$

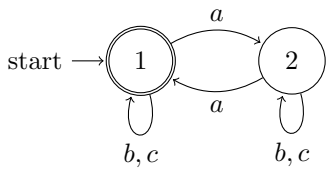
This example illustrates that regular languages are closed under intersection. Note that $L_3 = L_1 \cap L_2$.

5. ($2 \times 4 = 8$ **pts**) Draw DFA's for each of the languages L_1, L_2, L_3 and L_4 from Question 3 and 4.

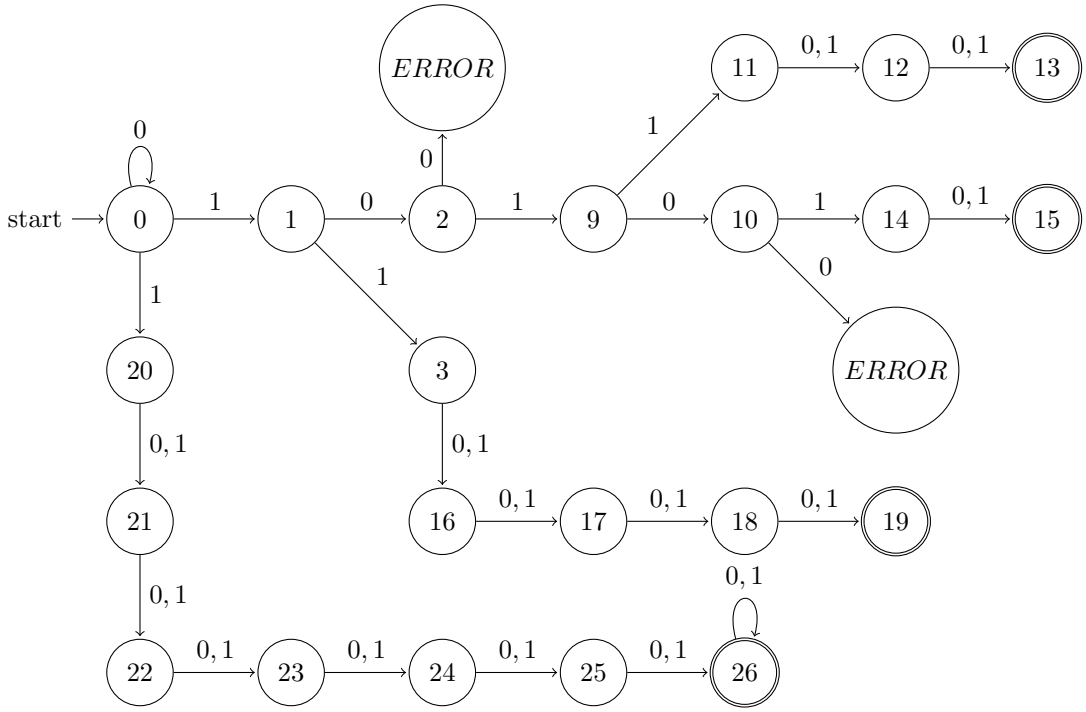
(a) L_1.



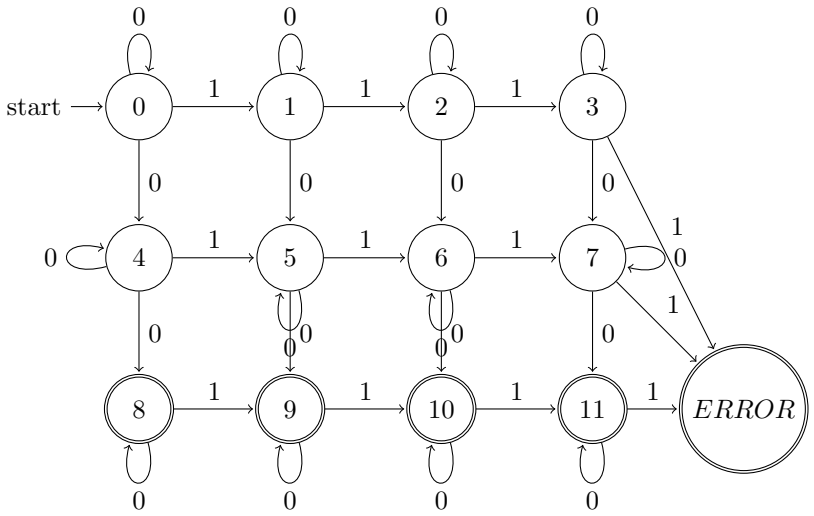
(b) L_2.



(c) L_3.

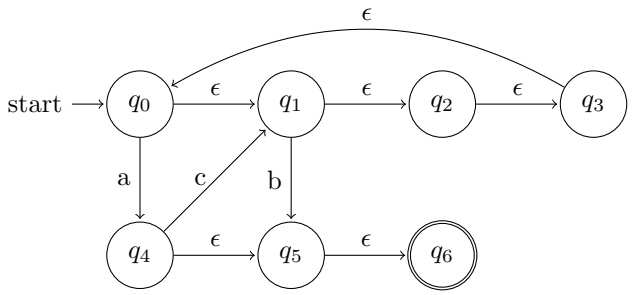


(d) L_4.

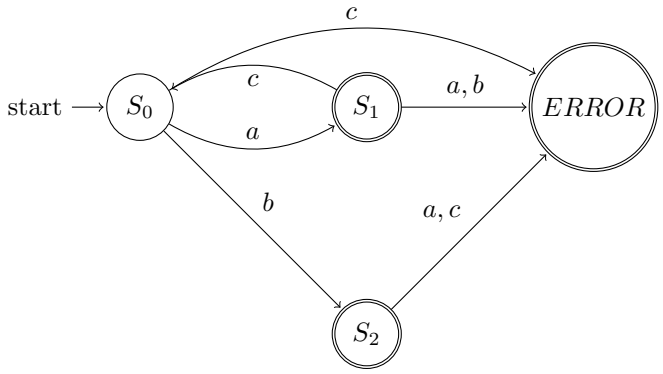

start
0
1
2
3
4
5
6
7
8
9
10
11
ERROR

6. ($5 \times 3 = 15$ **pts**) Using the techniques covered in class, transform the following NFAs with ϵ-transitions over the given alphabet Σ into minimized DFAs. Note that a DFA must have a transition defined for every state and symbol pair, whereas a NFA need not. You must take this fact into account for your transformations. If you can't make it and write down the intermediate process, you will gain some of the score.
Hint: Is there a subset of states the NFA transitions to when fed a symbol for which the set of current states has no explicit transition?

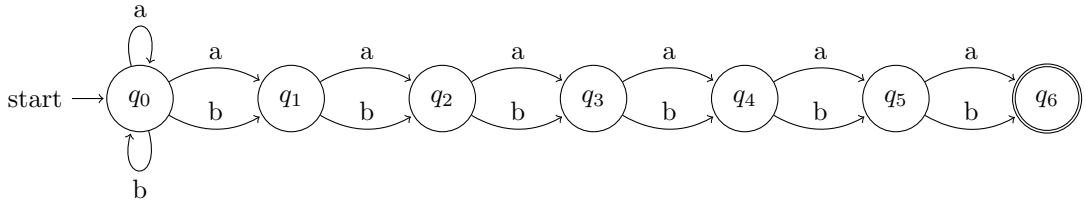
(a) Original NFA, $\Sigma = \{a, b, c\}$:



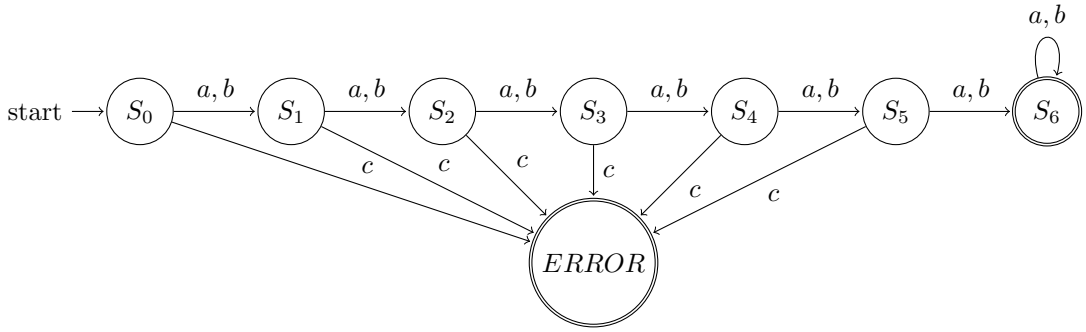
Minimized DFA:



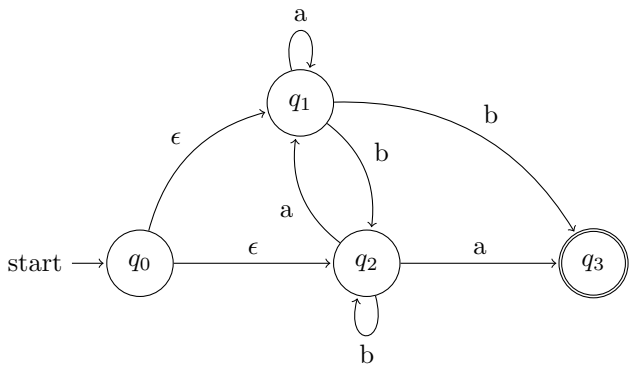
(b) Original NFA, $\Sigma = \{a, b, c\}$:



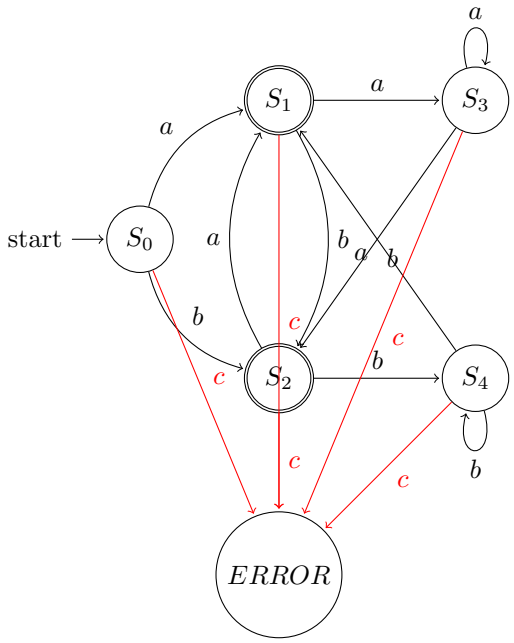
Minimized DFA:



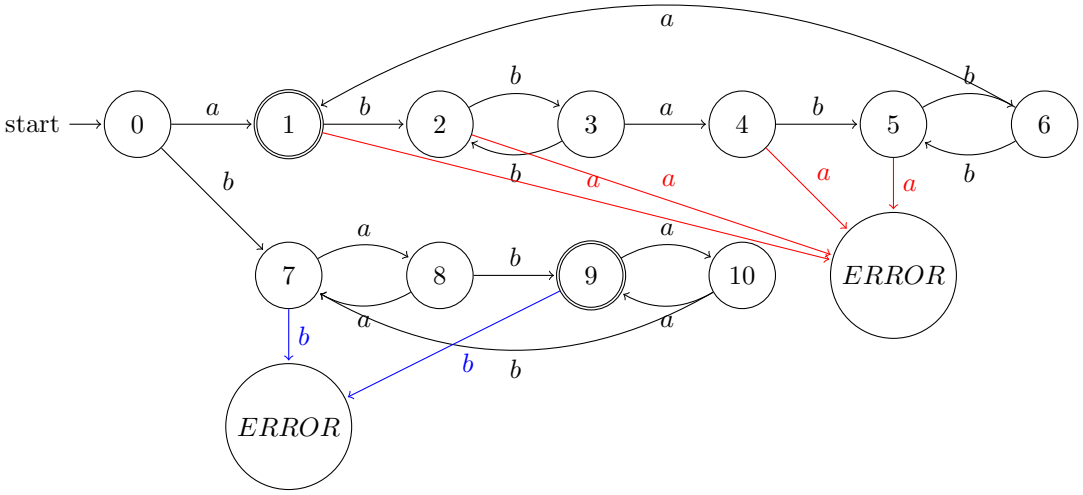
(c) Original NFA, $\Sigma = \{a, b, c\}$:



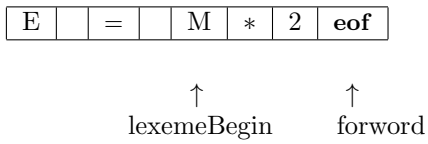
Minimized DFA:



7. (13 **pts**) Draw the NFA for the set of all strings over the alphabet $\Sigma = \{a, b\}$, where either a occurs an odd number of times and each of pair of a's is separated by exactly $2n+2$ consecutive b's (for some $n \geq 0$), or b occurs an even number of times and each of pair of relative consecutive b's is separated by exactly $2m+1$ consecutive a's (for some $m \geq 0$).
Examples of strings that should be accepted by this NFA: abbabbbba, babaaabaaaaab. Examples of strings that should **not** be accepted: ababb, abbbabba.



8. (10 **pts**) **Flex** scanner is very good tools to use, it can easily solve the problem listed above and do good error handling. The teaching assistant is recently facing a big bug when dealing with the indentation of a python parser. Please help him.
As we discussed in the lecture, the input buffer is the size of 4096 Bytes by default and we have 2 of them to take care of the lookahead safely. However, the both buffer will take in the **eof** and emit the **eof** twice. **eof** means the character is the end of the file.



In the **Flex**, we have something like below, the forword will touch the eof twice, and if you write something in the **EOF** block, it will be called twice. In python, we have to call the dedent when end of file but only once.

```
switch ( *forward++ ) {
case eof:
    if (forward is at end of first buffer ) {
        reload second buffer;
        forward = beginning of second buffer;
    }else if (forward is at end of second buffer ) {
        reload first buffer;
        forward = beginning of first buffer;
    }else /* eof within a buffer marks the end of input */
        terminate lexical analysis;
    break;
Cases for the other characters
}
```

In terms of two buffer design of **Flex**, we can't write the code like below, thus we have to modify it to a right one, please do the modification to avoid the second eof be read so that the **EOF** block will only read once. You can test it in the program yourself.

```
#include <stack>
#include <iostream>
#define LITERAL(type) {  std::cout << #type; return TOKEN_##type; }
std::stack<int> stack_indent;

%%

%{
  /* Init indent stack_indent */
  if (stack_indent.empty()) {
    stack_indent.push(0);
  }
%}
/* Other Lexical rules here. */
<<EOF>>   {
    while(stack_indent.top() != 0) {
          stack_indent.pop();
          LITERAL(DEDENT);
    }
    yyterminate();
}

    %x /* Some state defined here */
    %%
```

```
    <<EOF>> {

    while(stack_indent.top() != 0){
        stack_indent.pop();
        LITERAL(DEDENT);
        }
                yyterminate();
}
```