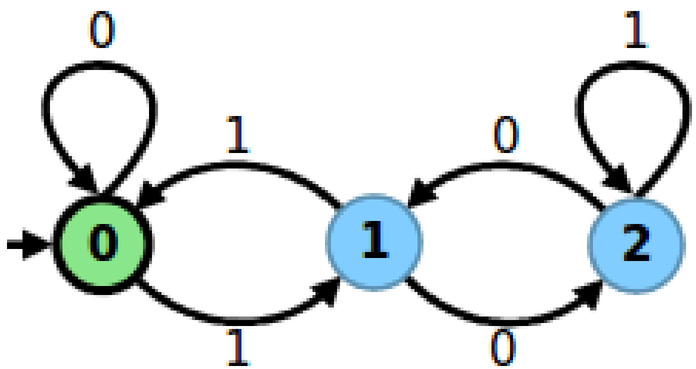
Proving correctness of automata for $\{ x \mid x \,\%\, 3 = 0, x \in \{0,1\}^* \}$

- Show that the following automaton accepts all binary strings w divisible by 3



- Induction over |w|
- Claim: $\forall\, w \in \{0,1\}^*, (w \,\%\, 3 = i) \Rightarrow \hat{\delta}(s_0, w) = s_i$
- Base case, |w| = 0 i.e, w = ϵ,
  - $(\epsilon \,\%\, 3 = 0)$ (by def.), $\hat{\delta}(s_0, \epsilon) = s_0$

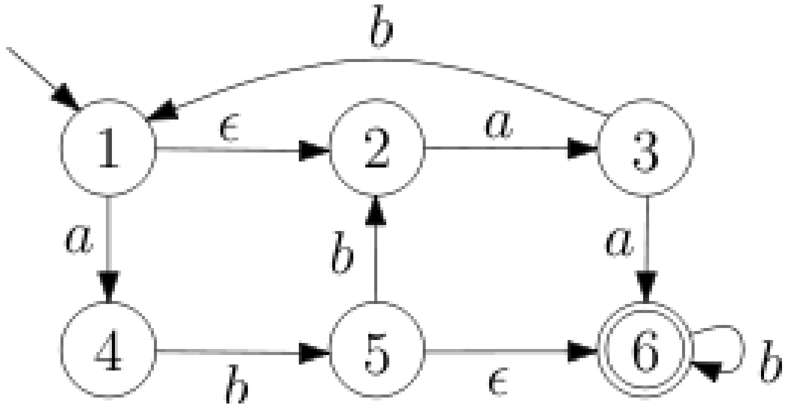
Proving correctness of automata for $\{ x \mid x \% 3 = 0, x \in \{0,1\}^* \}$

- Inductive step |w| = k + 1,
  - Case(a): w = x0
  - $w \% 3$ = x0 % 3 = (2*(x%3)) % 3
    - If x%3 = 0, w % 3 = 0
      - $\hat{\delta}(s_0, w) = \hat{\delta}(s_0, x0) = \delta(\hat{\delta}(s_0, x), 0)$
      - By hypothesis, $x \% 3 = 0 \Rightarrow \hat{\delta}(s_0, x) = s_0$
      - Therefore, $\hat{\delta}(s_0, w) = \delta(s_0, 0) = s_0$
      - Hence, when w % 3 = 0, $\hat{\delta}(s_0, w) = s_0$
    - If x%3 = 1, w % 3 = 2
      - By hypothesis, $x \% 3 = 1 \Rightarrow \hat{\delta}(s_0, x) = s_1$
      - Therefore, $\hat{\delta}(s_0, w) = \delta\left(\hat{\delta}(s_0, x), 0\right) = \delta(s_1, 0) = s_2$
- Similarly for other cases.

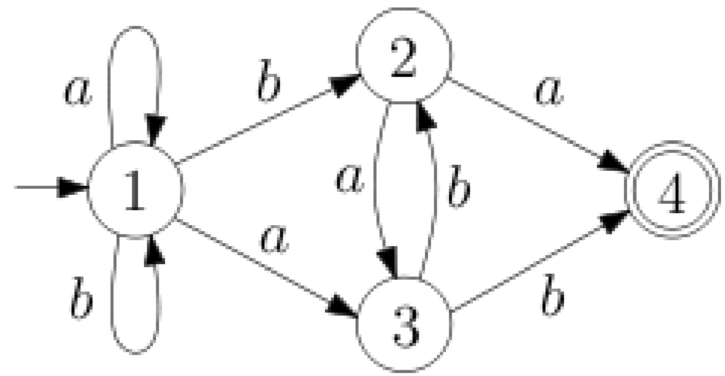
Exercise

Convert the following NFAs to deterministic finite automata.

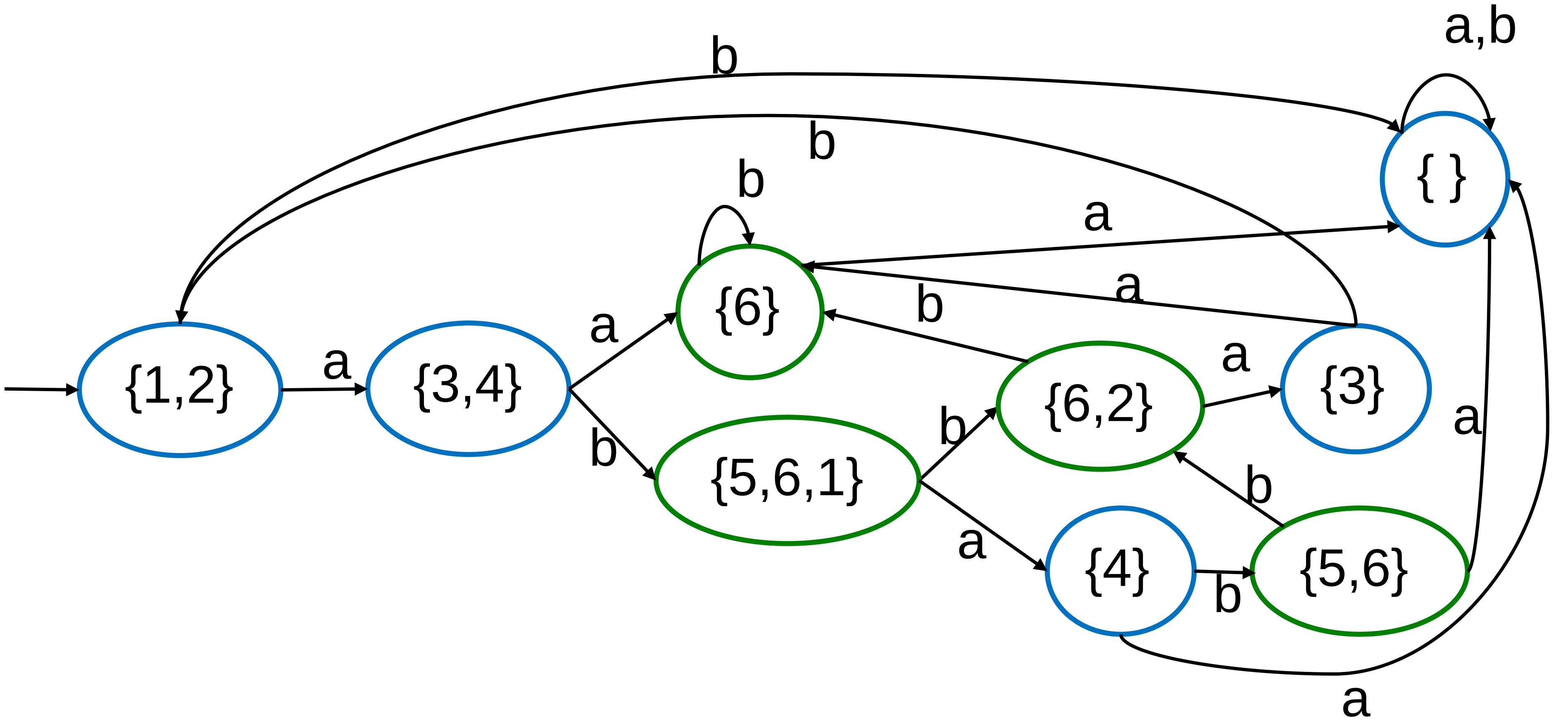
a)



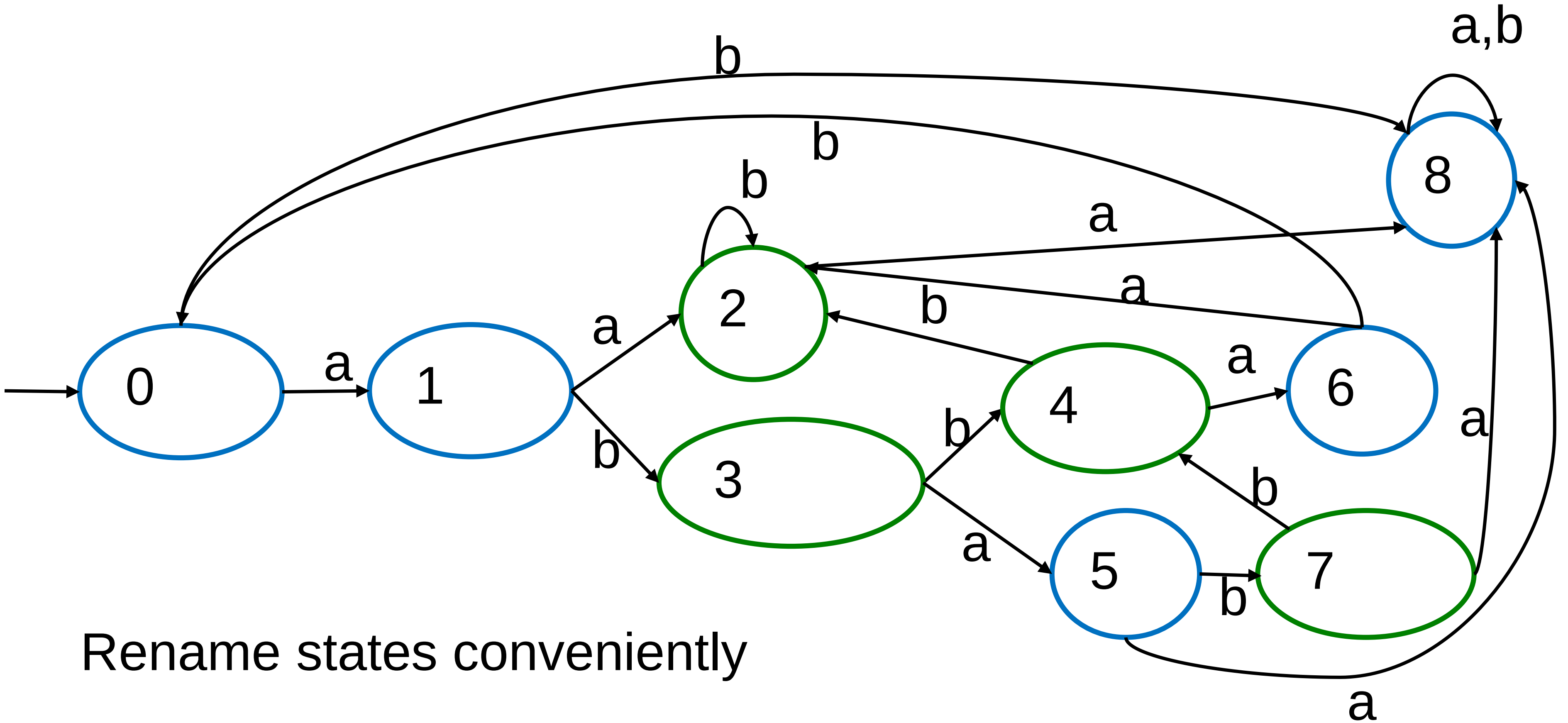
b)



Solution for Exercise (a)

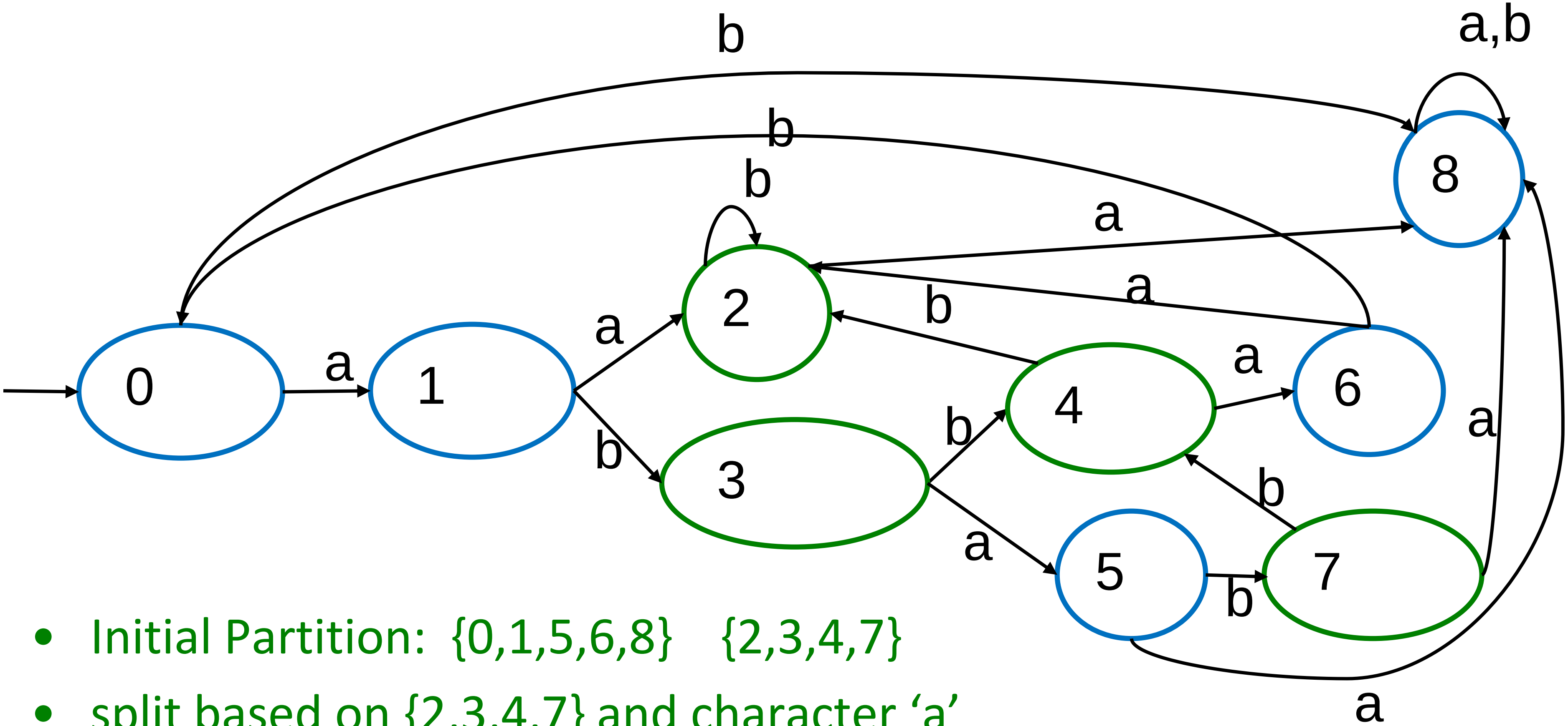


Solution for Exercise (a)



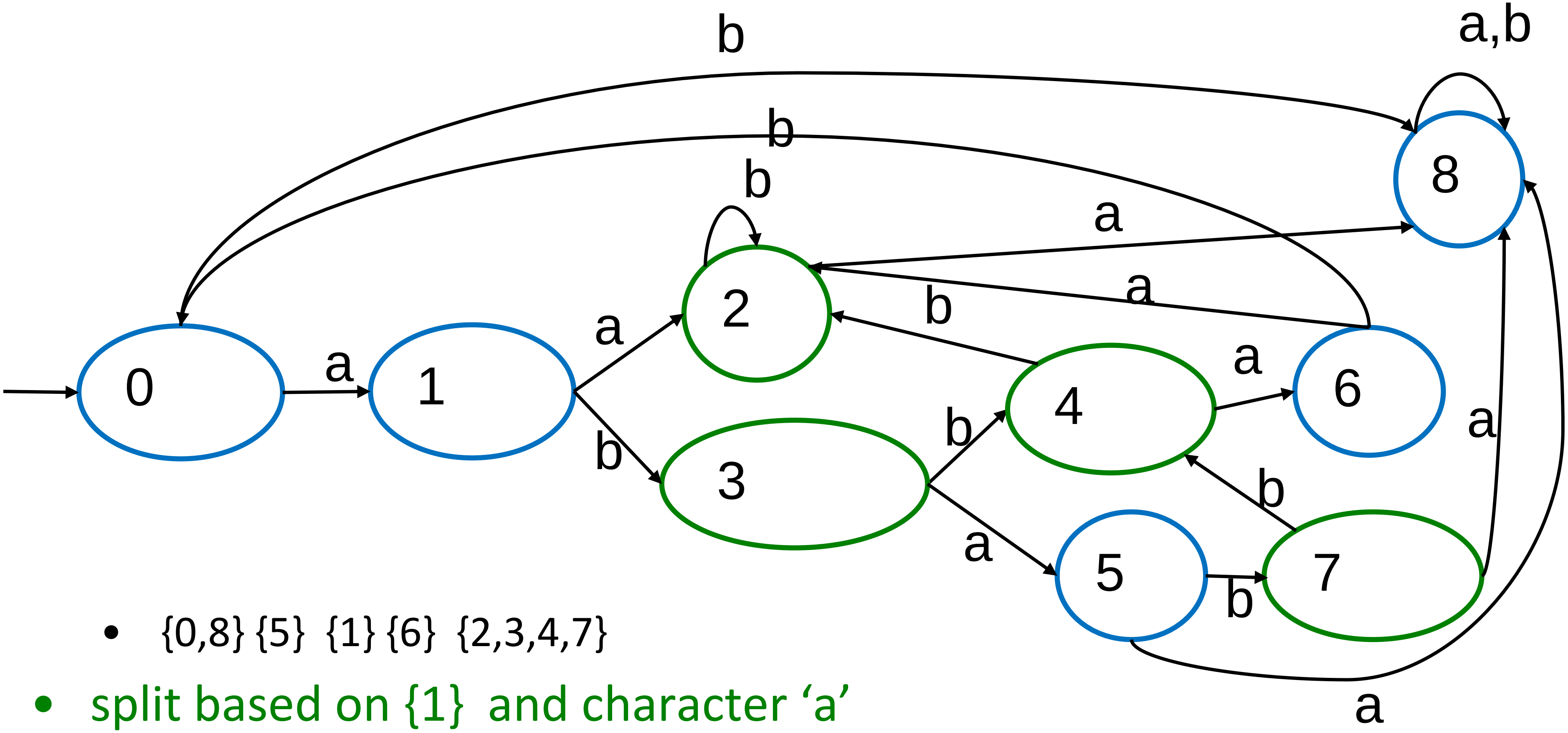
Rename states conveniently

Minimizing solution for (a)



- Initial Partition: {0,1,5,6,8} {2,3,4,7}
- split based on {2,3,4,7} and character ‘a’
  - {0,5,8} {1,6} {2,3,4,7}
- split based on {2,3,4,7} and character ‘b’
  - {0,8} {5} {1} {6} {2,3,4,7}

Minimizing solution for (a) [Cont.]



- {0,8} {5} {1} {6} {2,3,4,7}
- split based on {1} and character ‘a’
  – {0} {8} {5} {1} {6} {2,3,4,7}
- split based on ({8}, ‘a’), followed by ({4}, ‘b’) and ({5}, ‘a’)
  – {0} {8} {5} {1} {6} {2}{7} {3} {4}

Automated Construction of Lexers

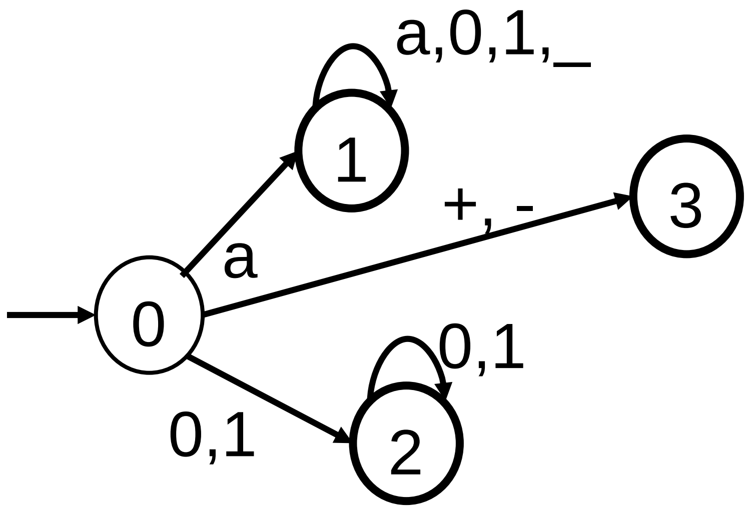
- let r_1, r_2, ..., r_n be regular expressions for token classes
  - <ID: a (a | 0 | 1 | _)*>
  - <INT: (0 | 1) (0 | 1)*>
  - <OP: + | - >

- consider combined regular expression: (r_1 | r_2 | ... | r_n)

  a (a | 0 | 1 | _)* | (0 | 1) (0 | 1)* | (+ | -)

Automated Construction of Lexers

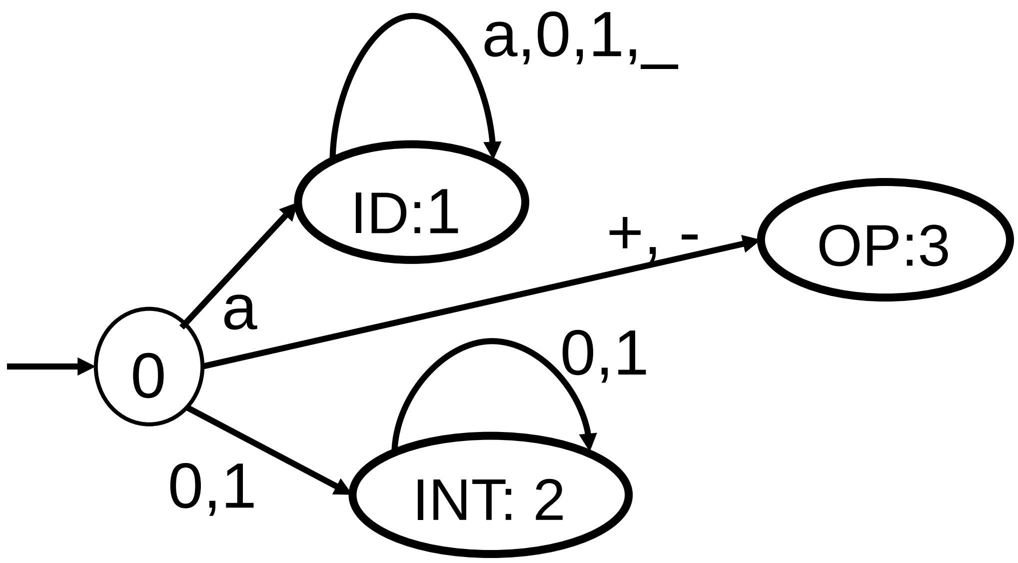
- Convert the regular expression to automaton



- For each accepting state of r_i specify the token class *i* being recognized

Automated Construction of Lexers

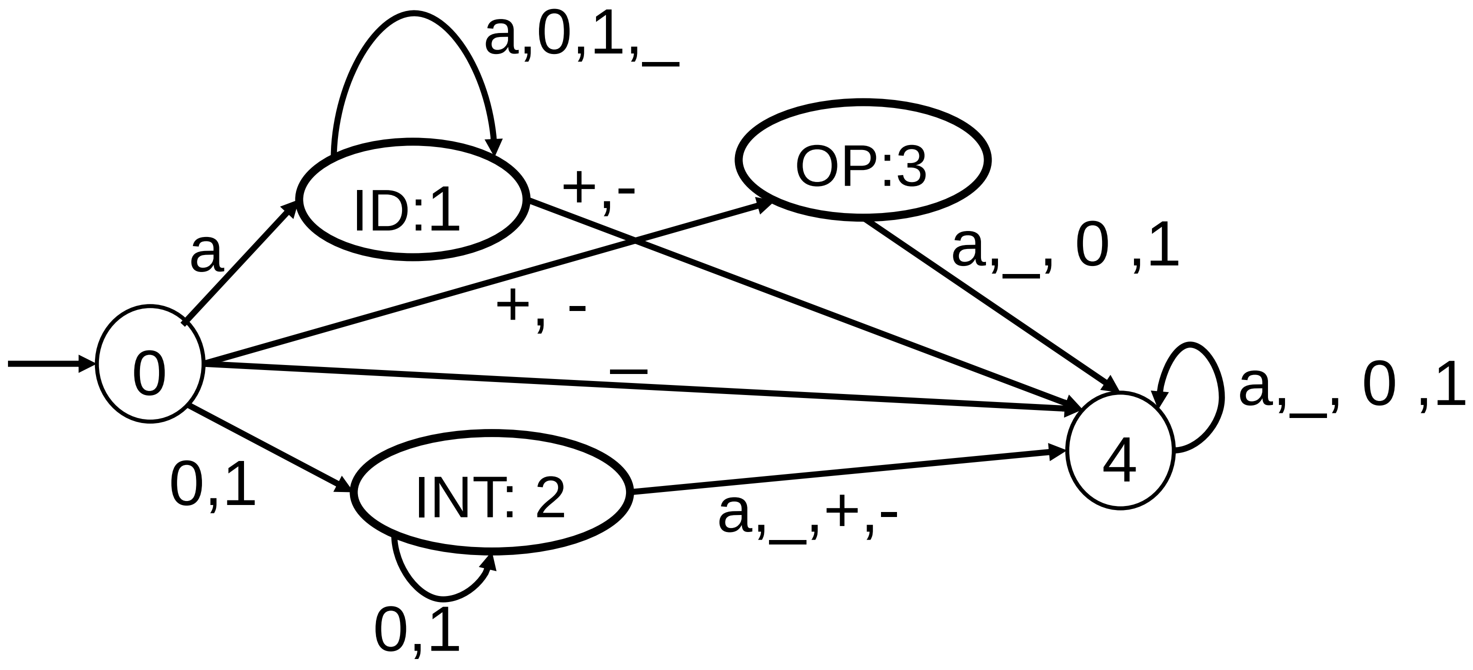
- Convert the regular expression to automaton



- For each accepting state of r_i specify the token class *i* being recognized

Automated Construction of Lexers

- Eliminate epsilon transitions and determinize
- Minimize the resulting automaton to reduce its size

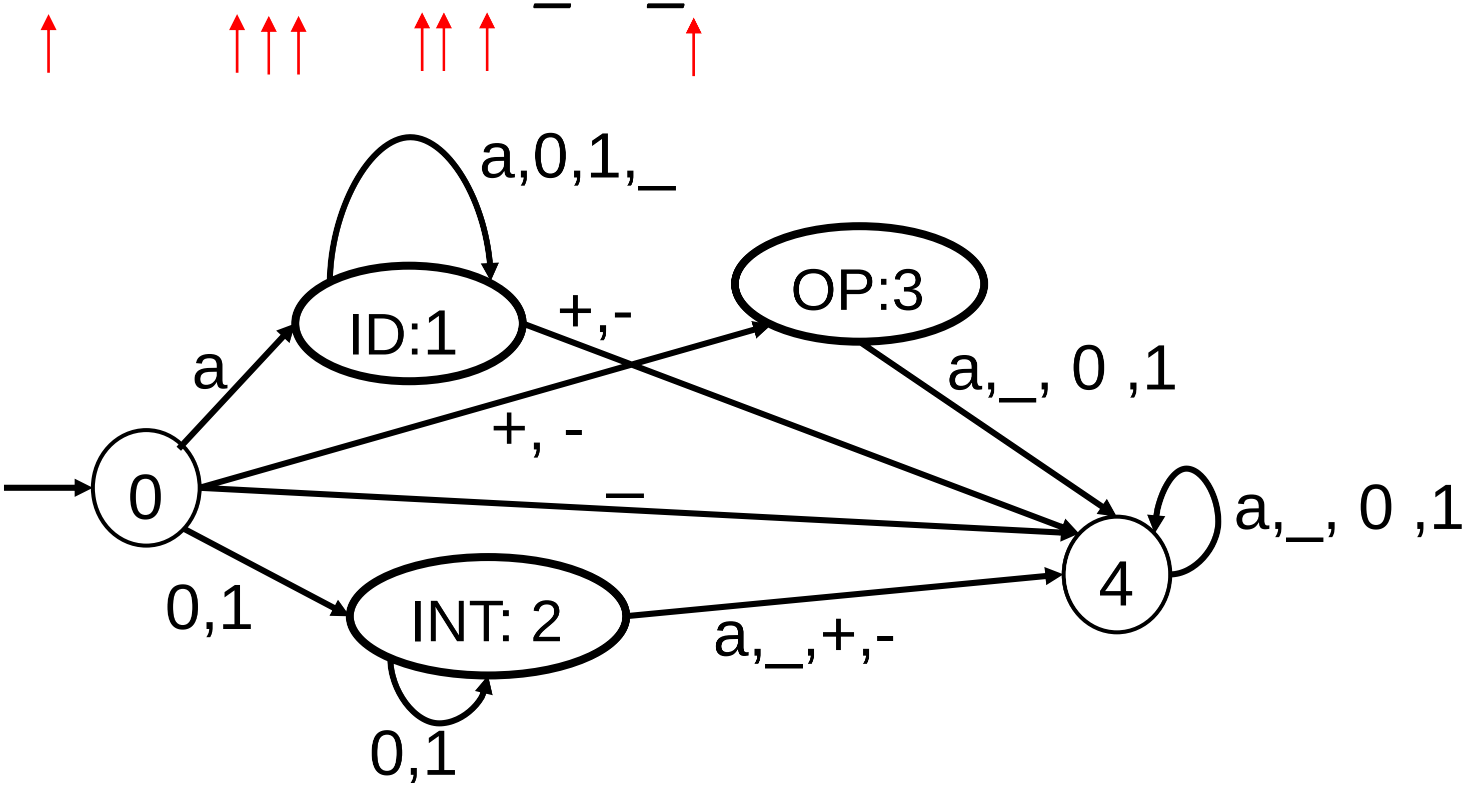


From $(r_1|r_2|\ldots|r_n)$ to a Lexer

- **Longest match rule:** remember last token and input position for a last accepted state
- When no accepting state can be reached (effectively: when we are in a trap state)
  - revert position to last accepted state
  - return last accepted token
- *Why can't we simply use $(r_1|r_2|\ldots|r_n)^*$?*

Example

- *Tokenize the following*
  - *a10110+0110-a0_10_*



Exercise

Build lexical analyzer for the following two tokens using longest match. The first token class has a higher priority:

binaryDigit ::= $(\mathbf{z} | 1)^*$

ternaryDigit ::= $(0|1|2)^*$

1111z1021z1 →

binaryDigit: 1111z1

ternaryDigit: 021

binaryDigit: z1

Realistic Exercise: Integer Literals of Scala

- Integer literals are in three forms in Scala: decimal, hexadecimal and octal. The compiler discriminates different classes from their beginning.
  - Decimal integers are started with a non-zero digit.
  - Hexadecimal numbers begin with 0x or 0X and may contain the digits from 0 through 9 as well as upper or lowercase digits from A to F.
  - If the integer number starts with zero, it is in octal representation so it can contain only digits 0 through 7.
  - l or L at the end of the literal shows the number is Long.
- Draw a single DFA that accepts all the allowable integer literals.
- Write the corresponding regular expression.

Exercise

- Let L be the language of strings over {<, =} defined by regexp (<|=| <====*). That is, L contains <,=, and words $<=^n$ for n >= 3.
- Construct a DFA that accepts L
- Describe how the lexical analyzer will tokenize the following inputs.

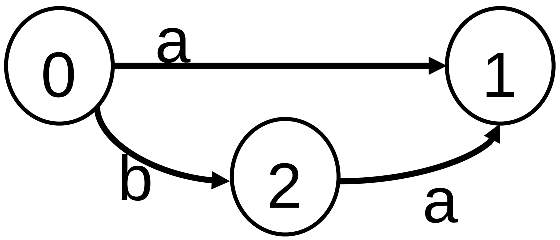
  1) <=====

  2) ==<==<==<==<==

  3) <=====<

Automata to Regular Expressions

- Every path in the automata corresponds to a RE



RE: a

RE: ba

- R_{pq}^{X} : RE corresponding to all paths from state ‘p’ to state ‘q’ that goes through only states in ‘X’
  - $R_{01}^{\emptyset}$ = a
  - $R_{01}^{\{2\}}$ = a + ba

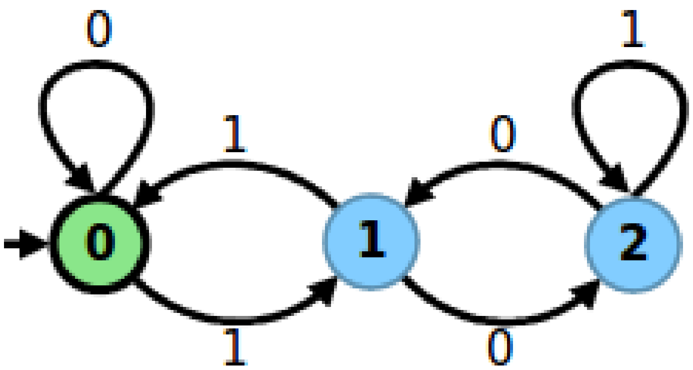
Automata to Regular Expressions

- $R_{pq}^{X} = R_{pq}^{X-\{u\}} + R_{pu}^{X-\{u\}}\left(R_{uu}^{X-\{u\}}\right)^{*} R_{uq}^{X-\{u\}}$

- $R_{pq}^{\emptyset} = a_1 + a_2 + \cdots + a_n\ ,\ \delta(p, a_i) = q$
- $R_{pp}^{\emptyset} = a_1 + a_2 + \cdots + a_n + \epsilon$

- R_{sf}^{Q} is the required regular expression

Automata to Regular Expressions

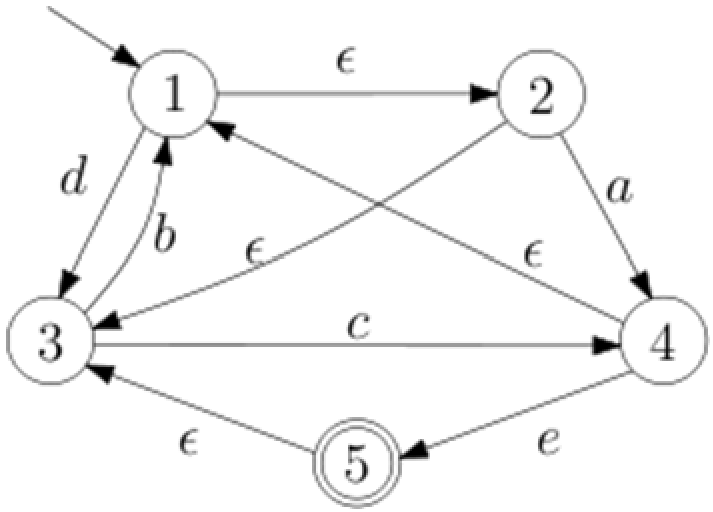


$$0^* + 0^*1(10^*1)^*10^*$$
$$+0^*1(10^*1)^*0\begin{pmatrix} 1^* \\ +0(10^*1)^*0 \end{pmatrix}^*$$
$$0(10^*1)^*10^*$$

- $R_{00}^{\{0,1,2\}} = R_{00}^{\{0,1\}} + R_{02}^{\{0,1\}}\left(R_{22}^{\{0,1\}}\right)^* R_{20}^{\{0,1\}}$
- $R_{00}^{\{0,1\}} = R_{00}^{\{0\}} + R_{01}^{\{0\}}\left(R_{11}^{\{0\}}\right)^* R_{10}^{\{0\}}$ $\quad 0^* + 0^*1(10^*1)^*10^*$
- $R_{00}^{\{0\}} = 0^*$
- ….

Exercise

- Convert the following automaton to RE



First Half of a Regular Language

Let L be a language. Define half(L) to be

{x | for some y such that |x| = |y|, xy is in L} .

That is, half(L) is the set of first halves of strings in L.

Prove that if L is regular then so is half(L).

There are many solutions to the problem.

The following is a tutorial on one good solution to the problem:

www-bcf.usc.edu/~breichar/teaching/2011cs360/half(L)example.pdf

More Questions

- For which of the following languages can you find an automaton or regular expression:
  - Sequence of open or closed parentheses of even length? E.g. (), ((,)),)()))(, ... yes
  - as many digits before as after decimal point? No
  - Sequence of balanced parentheses No

    ((()) ()) - balanced

    ( )) (( ) - not balanced
  - Comments from // until LF Yes
  - Nested comments like /* ... /* */ ... */ No