Automating Construction of Lexers

Example in javacc

```
TOKEN: {
   <IDENTIFIER: <LETTER> (<LETTER> | <DIGIT> | "_")* >
  | <INTLITERAL: <DIGIT> (<DIGIT>)* >
  | <LETTER: ["a"-"z"] | ["A"-"Z"]>
  | <DIGIT: ["0"-"9"]>
}
SKIP: {
  " "   |  "\n"   |   "\t"
}
```

--> get automatically generated code for lexer!

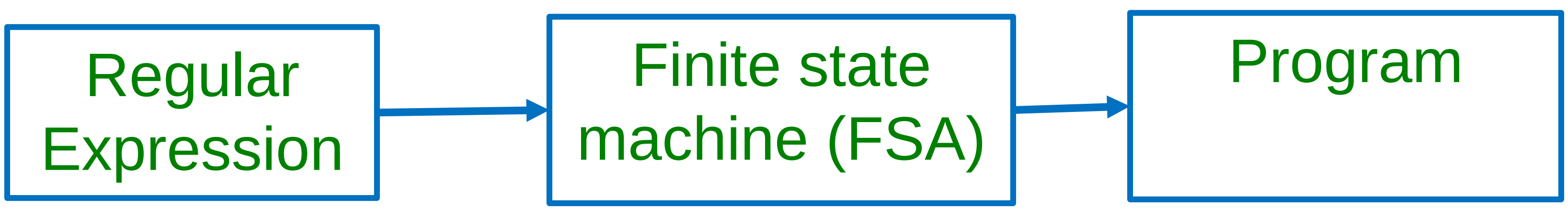
But how does javacc do it?

A Recap:
Simple RE to Programs

Regular Expression	**Code**
• a	• **if** (current=a) next **else** error
• r1 r2	• (code for r1) **;** (code for r2)
• (r1 **|** r2)	• **if** (current in first(r1)) code for r1 **else** code for r2
• r*****	• **while**(current in first(r)) code for r

Regular Expression to Programs

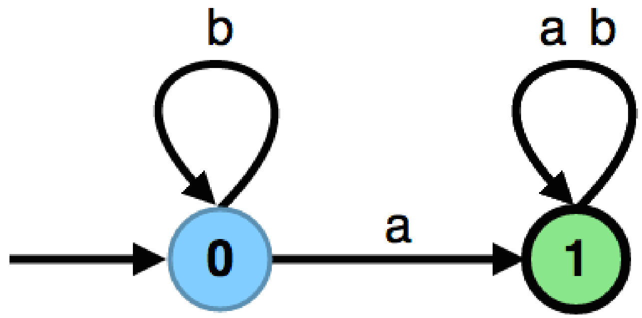
- How can we write a lexer for (a*b | a) ?
- aaaab Vs aaaaa



Finite Automaton (Finite State Machine)

- A = (Σ, Q, q_0, δ, F)

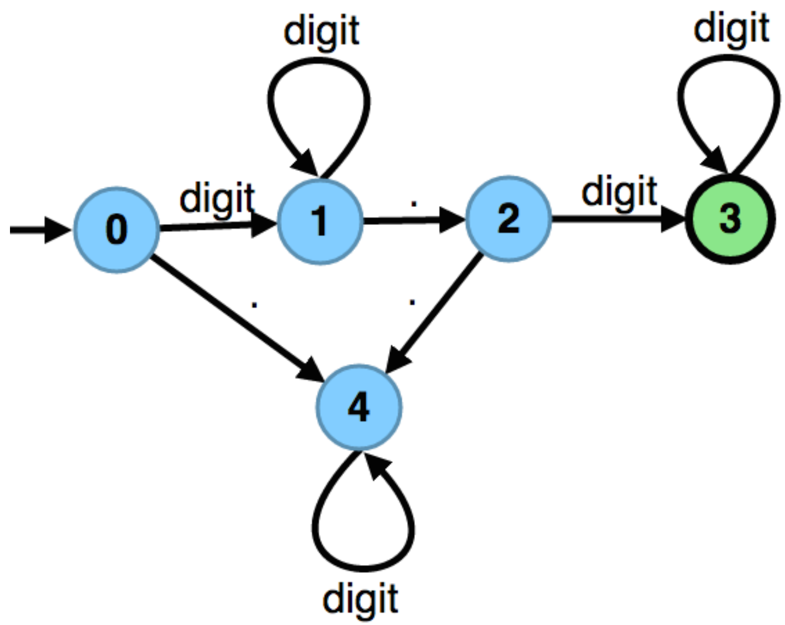
$$\delta \subseteq Q \times \Sigma \times Q,$$
$$q_0 \in Q,$$
$$F \subseteq Q$$



$$q_0 \in Q,$$
$$q_1 \subseteq Q$$
$$\delta = \{\, (q_0, a, q_1), (q_0, a, q_0)$$
$$(q_1, a, q_1), (q_1, b, q_1)\}$$

- Σ - alphabet
- Q - states (nodes in the graph)
- q_0 - initial state (with '->' sign in drawing)
- δ - transitions (labeled edges in the graph)
- F - final states (double circles)

Numbers with Decimal Point



digit digit* . digit digit*

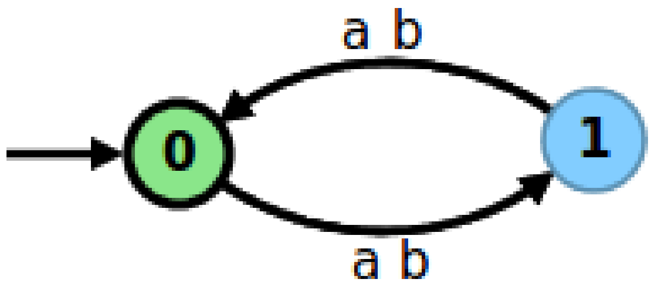
What if the decimal part is optional?

Automata Tutor
www.automatatutor.com

- A website for learning automata
- We have posted some exercises for you to try.
- Create an account for yourself
- Register to the course
  - Course Id: **23EPFL-CL**
  - Password: **GHL2AQ3I**

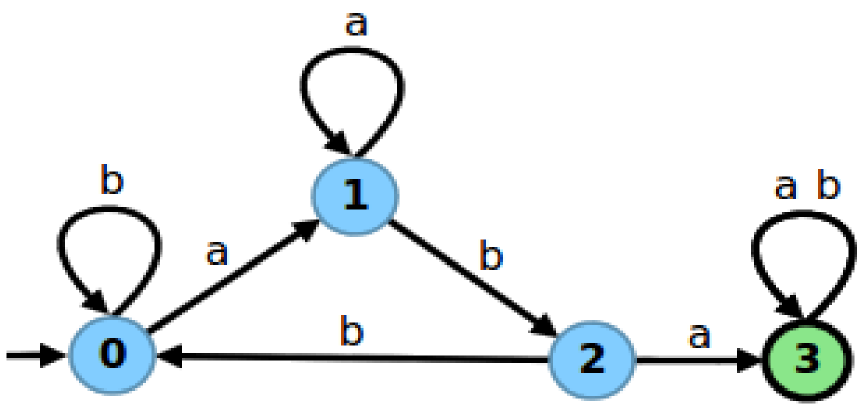
Exercise

- Design a DFA which accepts all strings in {a, b}* that has an even length



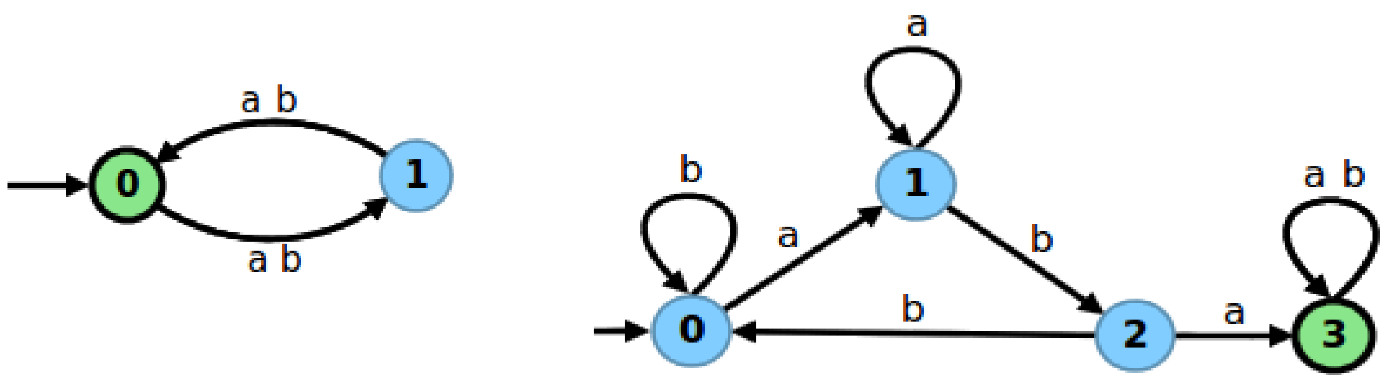
Exercise

- Construct a DFA that recognizes all strings over {a, b} that contain "aba" as a substring



Exercise

- Construct an automaton that recognizes all strings over { a,b} that contain "aba" as a substring and is of even length
  - Construct the product automaton of the following

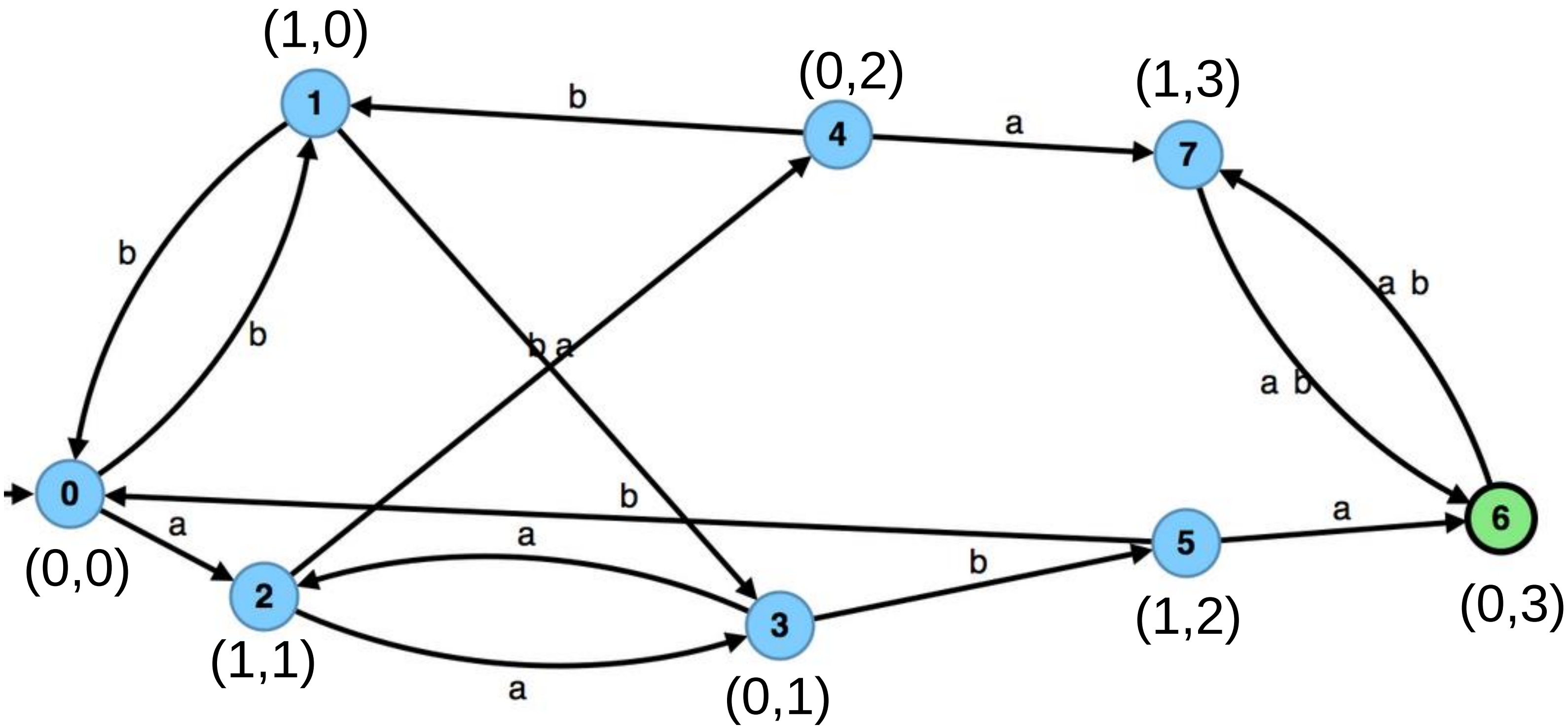


  - States: $Q_1 \times Q_2$ = { (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3) }
  - Transitions: For each $a \in \Sigma$,

    $\delta\big((q_1, q_2), a\big) = \big(\delta(q_1, a), \delta(q_2, a)\big)$

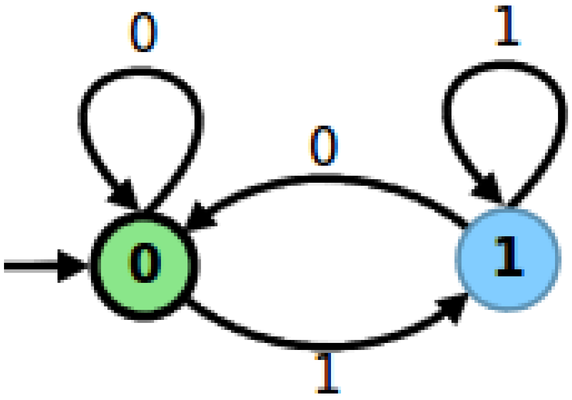
    Eg. $\delta\big((0{,}0), a\big) = (1{,}1)$, $\delta\big((0{,}0), b\big) = (1{,}0), \delta\big((1{,}1), b\big) = (0{,}2)$, ...
  - Start state: $(0{,}0)$, Final state: (0,3)

Solution: the product automaton



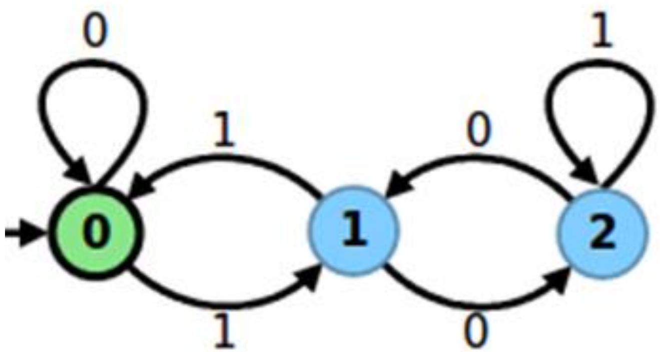
Exercise

- Design a DFA which accepts all the numbers written in binary and divisible by 2. For example, your automaton should accept the words 0, 10, 100, 110…



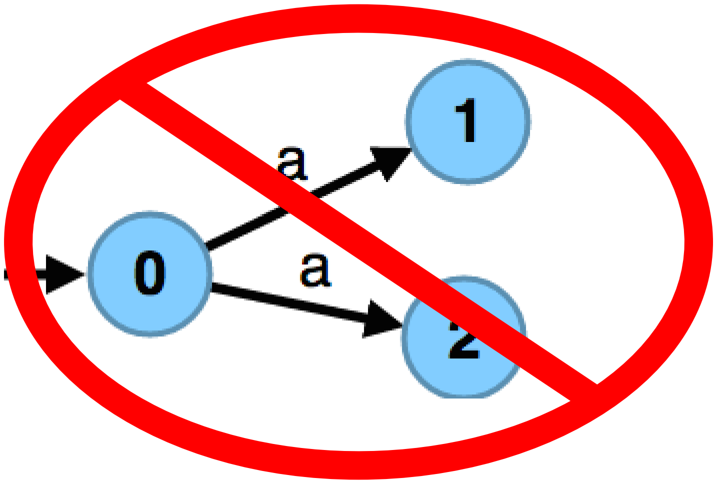
Exercise

- Design a DFA which accepts all the numbers written in binary and divisible by 3. For example your automaton should accept the words 0, 11, 110, 1001, 1100 …



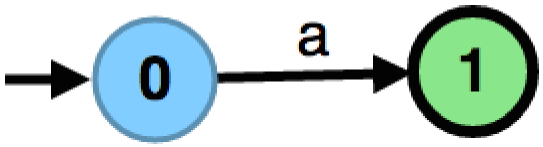
- Can you prove that the automaton accepts language ?
- Can you generalize this to any divisor 'n' and any base 'b' ?
  - Answers are in the next lecture slides

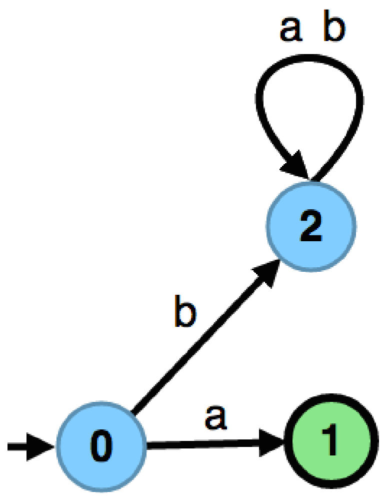
Kinds of Finite State Automata



- Deterministic FA (DFA): δ is a function : $(Q, \Sigma) \mapsto Q$
- Non-deterministic FA (NFA): δ could be a relation
- In NFA there is no unique next state. We have a set of possible next states.

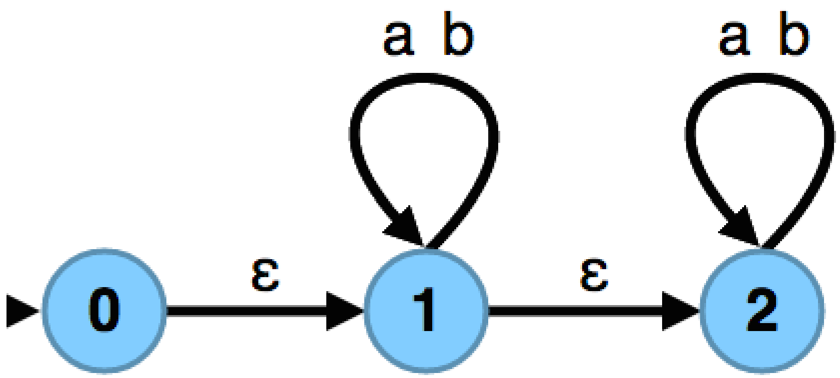
Undefined Transitions





- Undefined transitions lead to a sink state from where no input can be accepted

Epsilon Transitions



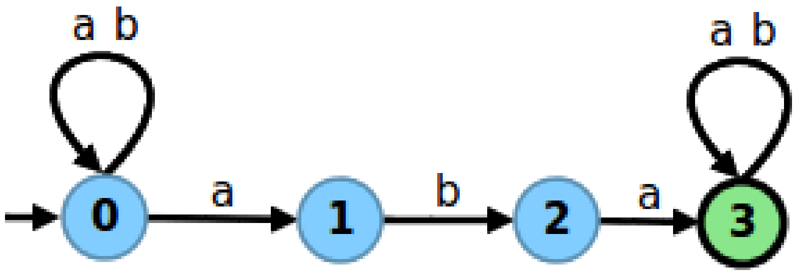
- Epsilon transitions: traversing them does not consume anything (empty word)
- More generally, transitions labeled by a word: traversing such transition consumes that entire word at a time

Interpretation of Non-Determinism

- For a given word (string), a path in automaton lead to accepting, another to a rejecting state
- Does the automaton accept in such case?
  - yes, if there **exists** an accepting path in the automaton graph whose symbols give that word

Exercise

- Construct a NFA that recognizes all strings over {a,b} that contain "aba" as a substring



NFA Vs DFA

- For every NFA there exists an equivalent DFA that accepts the same set of strings
- But, NFAs could be exponentially smaller.
- That is, there are NFAs such that every DFA equivalent to it has exponentially more number of states

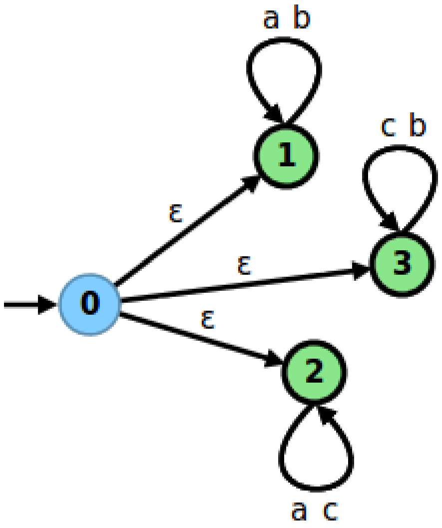
Exercise

- Construct a NFA and a DFA that recognizes all strings over {a,b,c} that do not contain all the alphabets a, b and c.

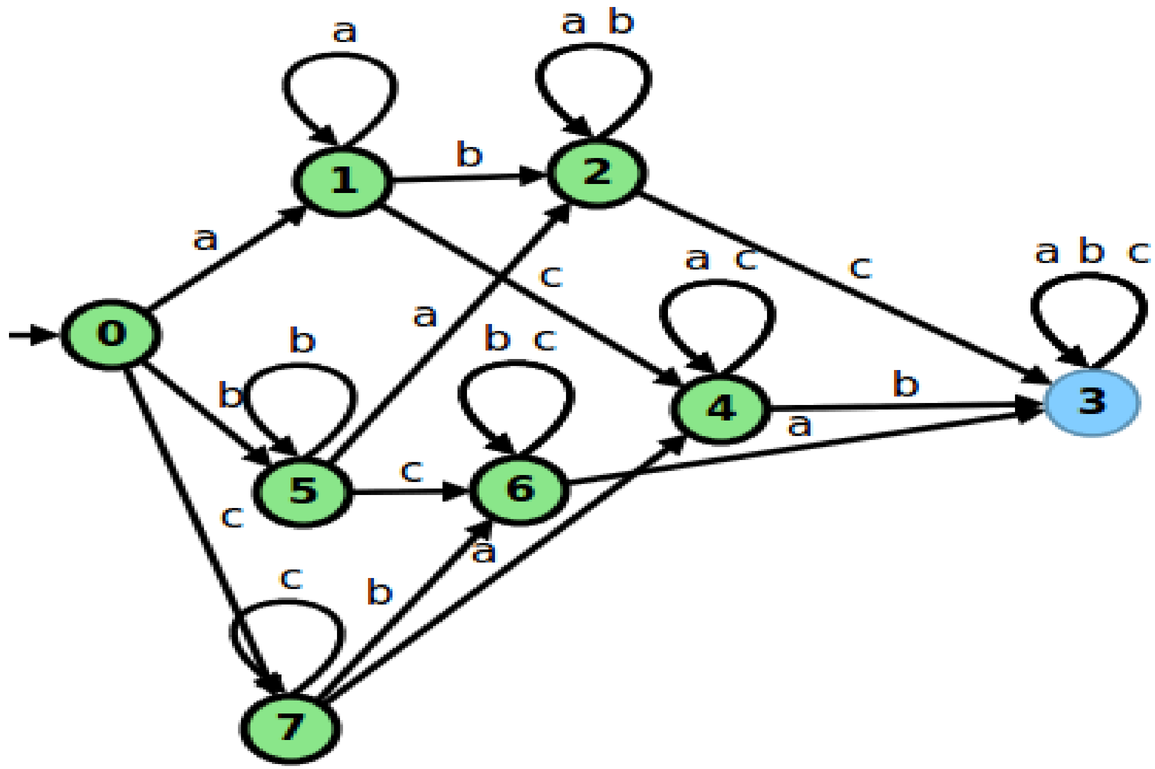
  (let's start with a regular expression)

  – Regular expression: $(a|b)^* \mid (b|c)^* \mid (a|c)^*$

  – NFA:



Solution: DFA



- Can you prove that every DFA for this language will have exponentially more states than the NFA ?
- Hints: Why is every intermediate state necessary ?
- Can you minimize the DFA any further ?

Regular Expressions and Automata

Theorem:

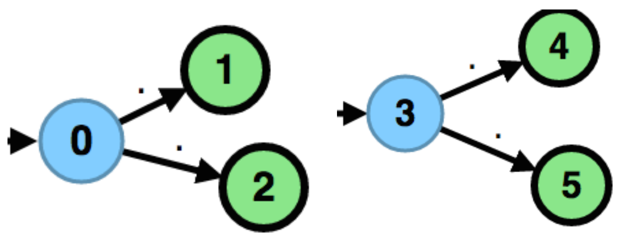
If L is a set of words, it is describable by a regular expression iff (if and only if) it is the set of words accepted by some finite automaton.

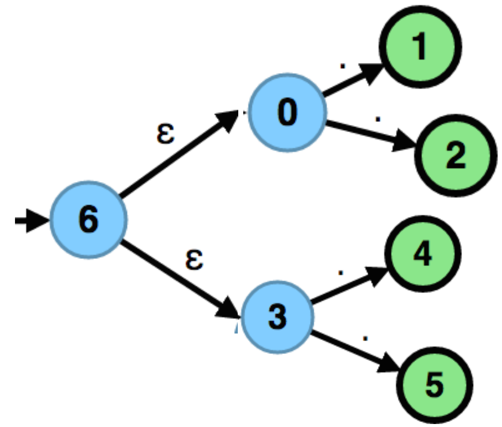
Algorithms:

- regular expression → automaton (important!)
- automaton → regular expression (cool)

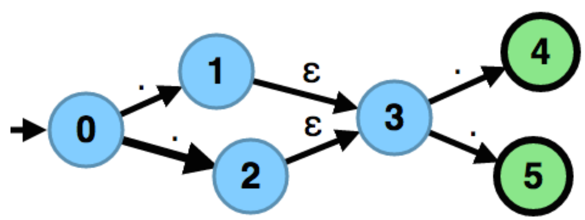
Recursive Constructions

- Union



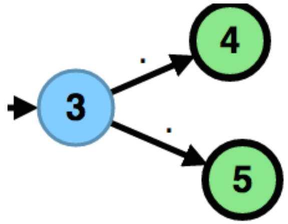


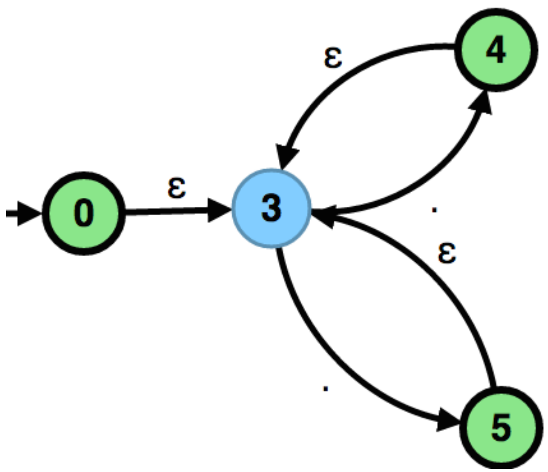
- Concatenation



Recursive Constructions

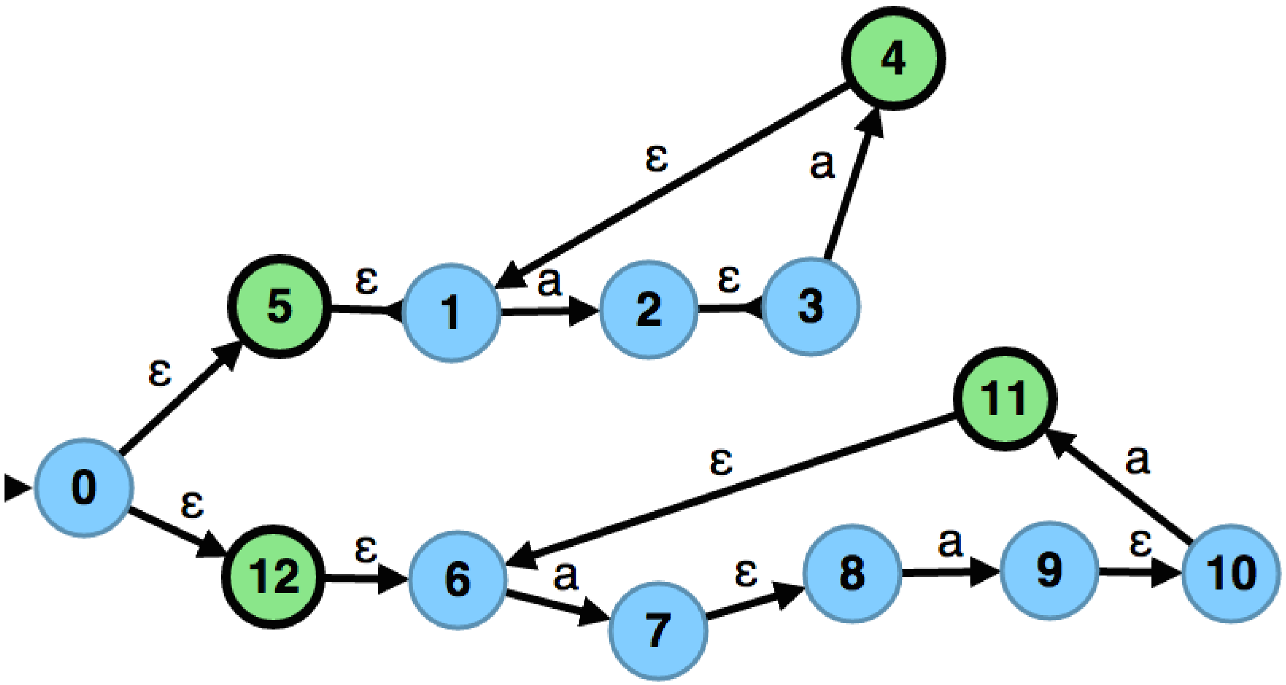
- Star





Exercise: (aa)* | (aaa)*

- Construct an NFA for the regular expression

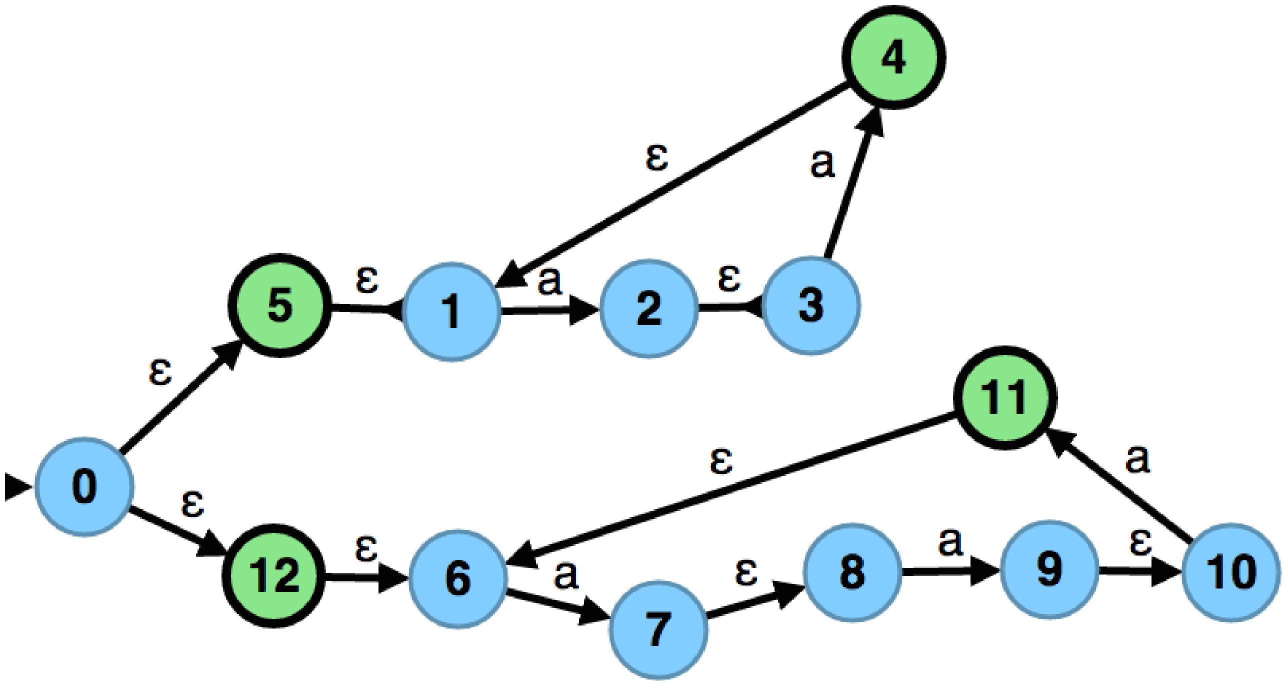


NFAs to DFAs (Determinisation)

- keep track of a set of all possible states in which the automaton could be

- view this finite set as one state of new automaton

NFA to DFA Conversion



Possible states of the DFA: 2^Q

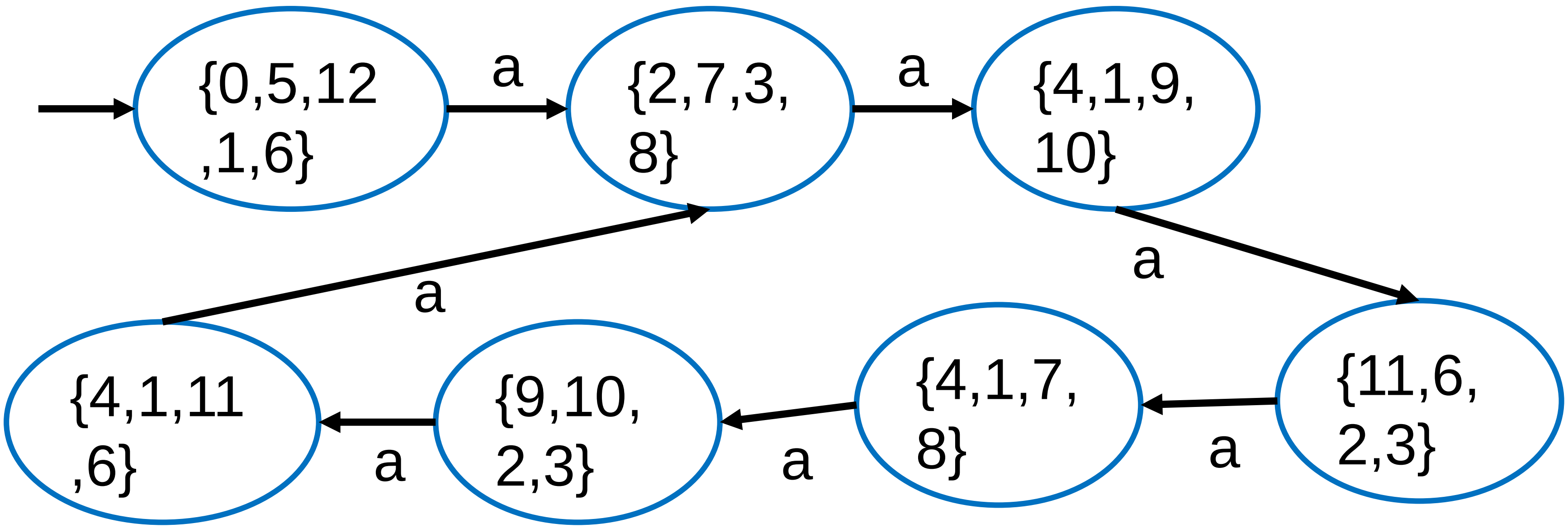
{ { } , { 0},…{12}, {0,1}, …,{0,12}, …{12, 12}, {0,1,2} …, { 0,1,2…,12 } }

NFA to DFA Conversion

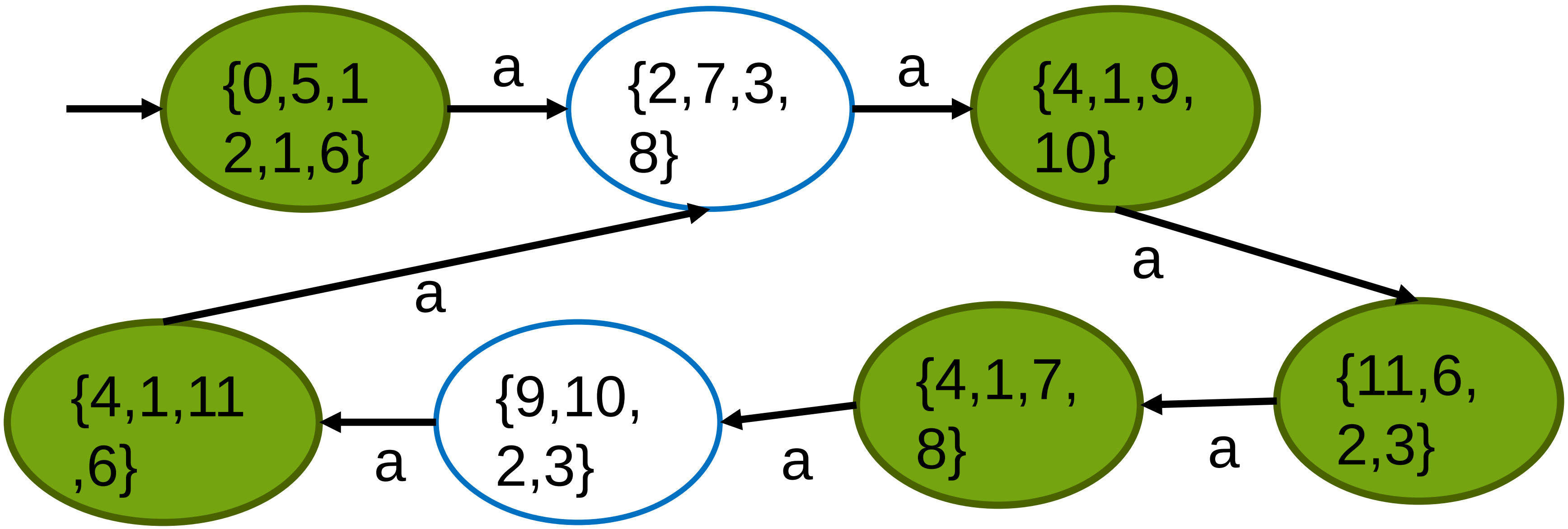
- Epsilon Closure
- E(0) = { 0,5,1,2,6}, E(1) = { 1}, E(2) = {
- $E(q) = \{ q_1 \mid \delta(q, \epsilon, q_1) \}$

- DFA: $(\Sigma, 2^Q, q_0', \delta', F')$
- $q_0' = E(q_0)$
- $\delta'(q', a) = \bigcup_{\{\exists q_1 \in q', \delta(q_1, a, q_2)\}} E(q_2)$
- $F' = \{ q' | q' \in 2^Q, q' \cap F \neq \emptyset\}$

NFA to DFA Conversion



NFA to DFA Example



Remark: Relations and Functions

- Relation $r \subseteq B \times C$

  $r = \{ \ldots, (b,c1) , (b,c2) ,\ldots \}$

- Corresponding function: $f : B \to 2^C$

  $f = \{ \ldots (b,\{c1,c2\}) \ldots \}$

$f(b) = \{ c \mid (b,c) \in r \}$

- Given a state, next-state function returns the set of new states
  - for deterministic automaton, the set has exactly 1 element

Clarifications

- what happens if a transition on an alphabet ‘a’ is not defined for a state ‘q’ ?
- $\delta'(\{q\}, a) = \emptyset$
- $\delta'(\emptyset, a) = \emptyset$

- Empty set represents a state in the NFA
- It is a trap/sink state: a state that has self-loops for all symbols, and is non-accepting.

Running NFA (without epsilons) in Scala

```
def δ(q : State, a : Char) : Set[States] = { ... }
def δ'(S : Set[States], a : Char) : Set[States] = {
  for (q1 <- S, q2 <- δ(q1,a)) yield q2
}

def accepts(input : MyStream[Char]) : Boolean = {
 var S : Set[State] = Set(q0) // current set of states
 while (!input.EOF) {
  val a = input.current
  S = δ'(S,a)             // next set of states
 }
 !(S.intersect(finalStates).isEmpty)
}
```

Running NFA in Scala

- Modify this to handle epsilons transitions.

def δ(q : State, a : Char) : Set[States] = { ... }
def δ'(S : Set[States], a : Char) : Set[States] = {
  for (q1 <- S, q2 <- δ(q1,a))
    for(q <- δ(q2, ϵ)) **yield** q
}

Minimizing DFAs

- Merge equivalent states.
  - q_0 and q_1 are equivalent iff there is no distinguishing string
  - $\hat{\delta}(q_0, z) \in F \Leftrightarrow \hat{\delta}(q_1, z) \in F$
  - Corollary of *Myhill-Nerode Theorem*
- Final and non-final states are not equivalent as ϵ distinguishes them

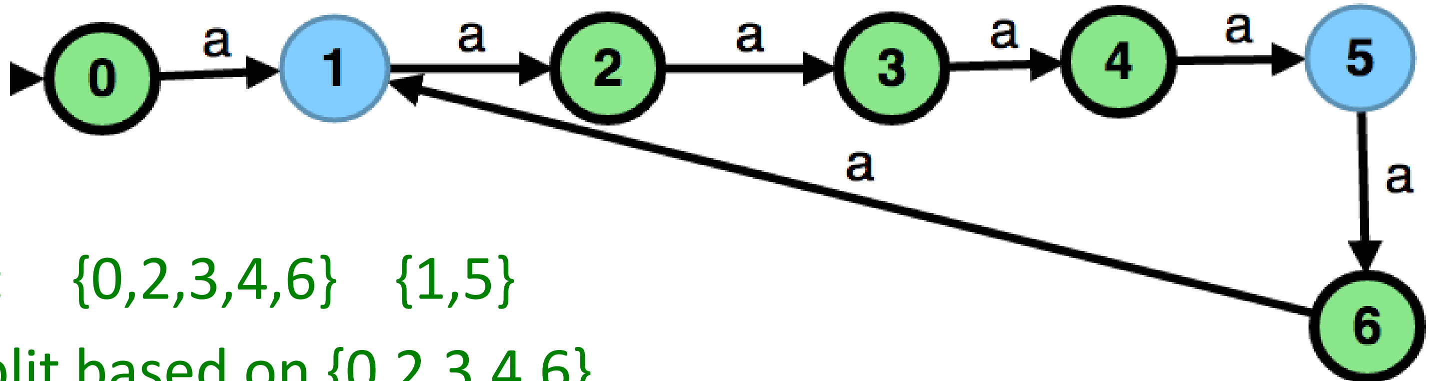
Minimizing DFAs: Procedure

- Maintain a partition A of states
- Every set in the partition has a different behavior i.e, they have a distinguishing string
- States within a partition may or may not be equivalent
- Initially, we have (F, Q - F)

Minimizing DFAs: Procedure [Cont.]

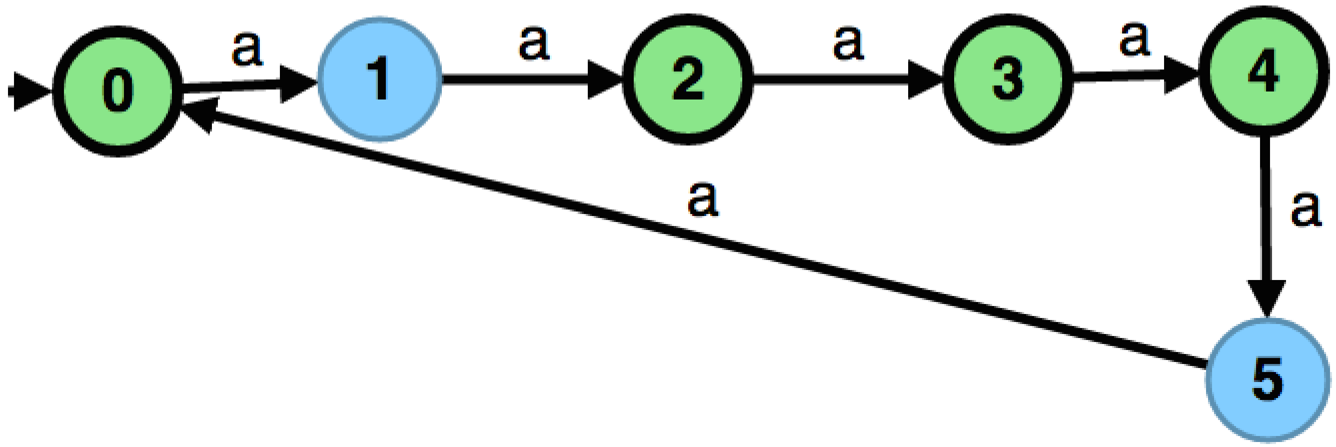
- Pick any partition P, choose some alphabet ‘a’.
- Split every partition (including P) by separating the states that has a transition to a state in P on ‘a’, and those that do not.
- Repeat until no partition can be split. That is, no choice of P and ‘a’ will split any partition

Minimizing DFAs: Procedure



- A: {0,2,3,4,6} {1,5}
- split based on {0,2,3,4,6}
  - A: {0,4,6} {2,3} {1,5}
- split based on {2,3}
  - A: {0,4,6} {2,3} {1} {5}
- split based on {1}
  - A: {0,6} {4} {2,3} {1} {5}
- split based on {4}
  - A: {0,6} {4} {2} {3} {1} {5}

Minimizing DFAs: Procedure



- The minimal DFA is unique (up to isomorphism)
- Implication of *Myhill-Nerode theorem*
- *Food For Thought: Can we minimize NFA ?*

Properties of Automatons

- **Complement:**
  - Given a DFA A, switch accepting and non-accepting states in A to obtain the complement automaton A^c
  - $\mathrm{L}(\mathrm{A}^{\mathrm{c}}) = (\Sigma^* \setminus L(A))$
  - Does not work for NFA
- **Intersection:**
  - Define $A' = (\Sigma, Q_1 \times Q_2, (q_0^1, q_0^2), \delta', F_1 \times F_2)$
  - $\delta'\big((q_1, q_2), a\big) = \delta(q_1, a) \times \delta(q_2, a)$
  - $\mathrm{L}(\mathrm{A}') = L(A_1) \cap L(A_2)$

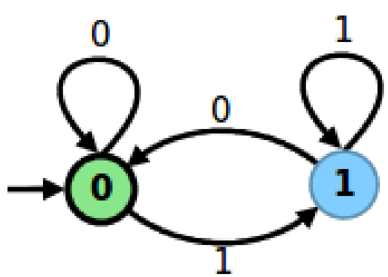
Properties of Automatons

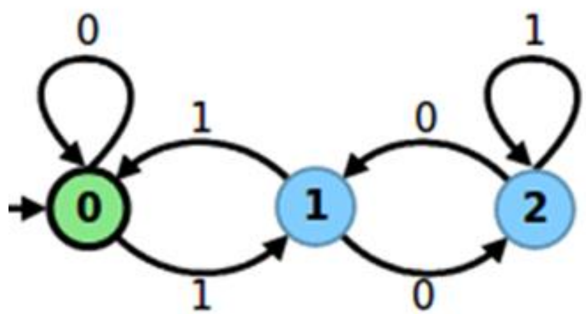
- **Intersection (another approach):**
  - complement union of complements
  - $A_1 \cap A_2 = (A_1^c \cup A_2^c)^c$
- **Set difference**: intersection with complement
  - $A_1 \setminus A_2 = (A_1 \cap A_2^c)$
- **Inclusion:** Is $L(A_1) \subseteq L(A_2)$?
  - emptiness of set difference
  - True iff $A_1 \setminus A_2$ does not accept any string. L($A_1 \setminus A_2$) = ∅
- **Equivalence:** Is $L(A_1) = L(A_2)$?
  - two inclusions

Exercise

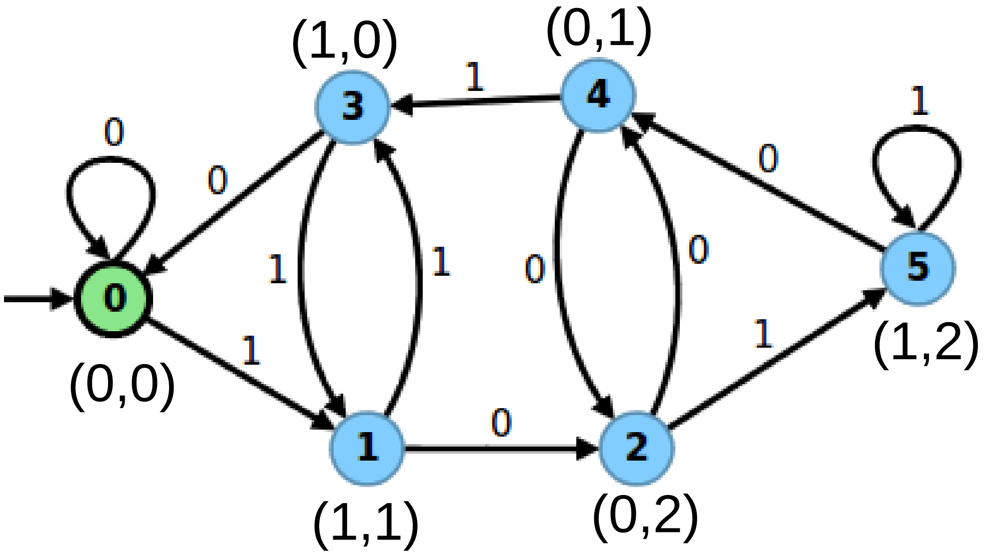
- Design a DFA which accepts all the numbers written in binary and divisible by 6. For example your automaton should accept the words 0, 110 (6 decimal) and 10010 (18 decimal).

  – You can construct the product of the following automatons that accept numbers divisible by 2 and 3





Solution: Product Automaton



Exercise: first, nullable

- For each of the following languages find the *first* set. Determine if the language is *nullable*.
  - (a|b)* (b|d) ((c|a|d)* | a*)

  Answer:
  - First = { a, b, d }
  - not nullabe, the minimal strings belonging to the regex are 'b' and 'd'