Automating Construction of Lexers by converting Regular Expressions to Automata

Regular Expression to Programs

- How can we write a lexer that has these two classes of tokens: {a, b}
  - a*b
  - aaa
- Consider run of lexer on: **aaaab** and on: **aaaaaa**

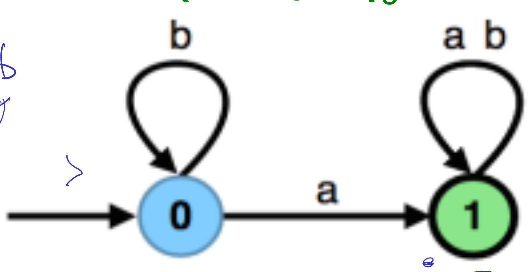
Regular Expression to Programs

- How can we write a lexer that has these two classes of tokens:
  - a*b
  - aaa
- Consider run of lexer on: **aaaab** and on: **aaaaaa**
- A general approach:

Finite Automaton (Finite State Machine)

$A = (\Sigma, Q, q_0, \delta, F)$

abb



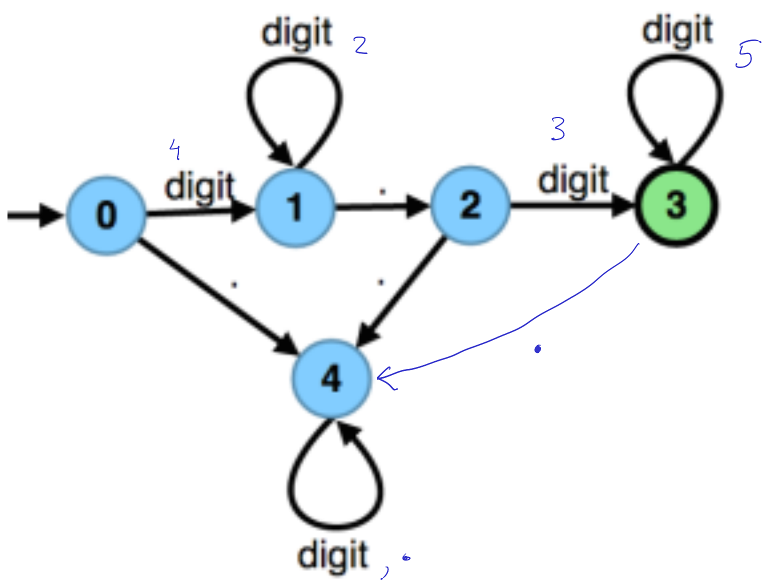
$$\delta \subseteq Q \times \Sigma \times Q,$$
$$q_0 \in Q,$$
$$F \subseteq Q$$
$$q_0 \in Q$$
$$q_1 \subseteq Q$$
$$\delta = \{ (q_0, a, q_1), (q_0, b, q_0), (q_1, a, q_1), (q_1, b, q_1), \}$$

- Σ - alphabet $= \{a, b\}$
- 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*

42.35

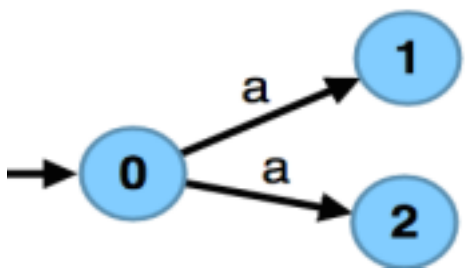


What if the decimal part is optional?

Kinds of Finite State Automata

- DFA: δ is a function : $(Q, \Sigma) \mapsto Q$
- NFA: δ could be a relation

- In NFA there is no unique next state. We have a set of possible next states.



Remark: Relations and Functions

- Relation $r \subseteq B \times C$

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

  $\mathcal{P}(c)$

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

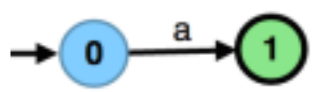
  $f = \{ \ldots (b,\{c1,c2\}) \ldots \}$

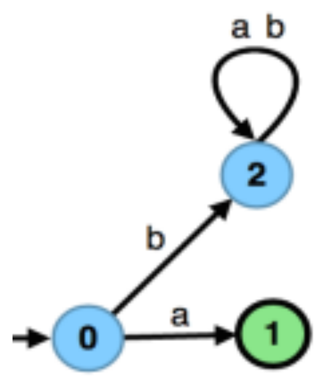
$f(b) = \{ c \mid (b,c) \in r \}$

- Given a state, next-state function returns a **set** of new states

  for *deterministic* automaton, set has *exactly 1* element

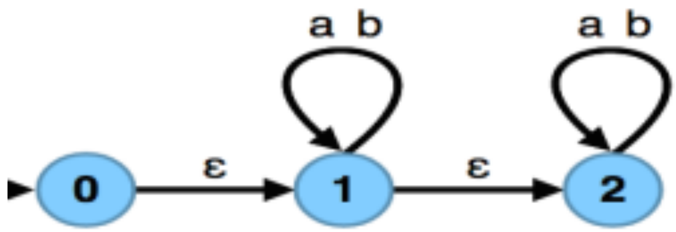
Allowing Undefined Transitions





- Undefined transitions are equivalent to transition into a sink state (from which one cannot recover)

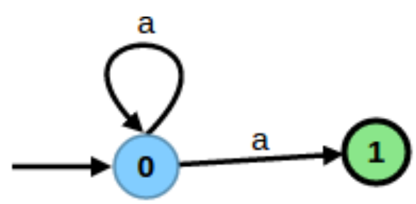
Allowing Epsilon Transitions



- Epsilon transitions:
  - traversing them does not consume anything
- Transitions labeled by a word:
  - traversing them consumes the entire word

When Automaton Accepts a Word

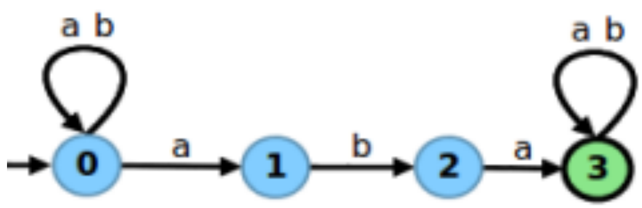
Automaton accepts a word *w* iff there **exists a path** in the automaton from the starting state to some accepting state such that concatenation of words on the path gives *w*.



- Does the automaton accept the word ***a*** ?

Exercise

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



aababb

$$0 \xrightarrow{a} 0,1 \xrightarrow{a} 0,1 \xrightarrow{b} 0,2 \xrightarrow{a} 0,1,3 \xrightarrow{b} 0,2,3$$

$$\downarrow b$$

$$\{0,3\}$$

Running NFA (without epsilons)

```
def δ(a : Char)(q : State) : Set[States] = { ... }
def δ'(a : Char, S : Set[States]) : Set[States] = {
  for (q1 <- S, q2 <- δ(a)(q1)) yield q2  // S.flatMap(δ(a))
}
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'(a,S)         // next set of states
 }
 !(S.intersect(finalStates).isEmpty)
}
```

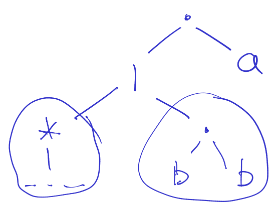
NFA Vs DFA

- Every DFA is also a NFA (they are a special case)
- For every NFA there exists an equivalent DFA that accepts the same set of strings
- But, NFAs could be exponentially smaller (succinct)
- There are NFAs such that **every** DFA equivalent to it has exponentially more number of states

Regular Expressions and Automata

Theorem:

Let L be a language. There exists a regular expression that describes it if and only if there exists a finite automaton that accepts it.



$$(\underbrace{(ab^* \mid a^*)^*} \mid bb)\,a$$

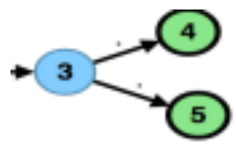
Algorithms:

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

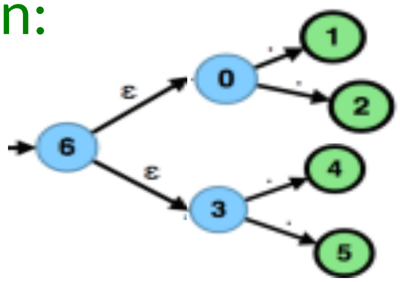
Recursive Constructions

A_1:

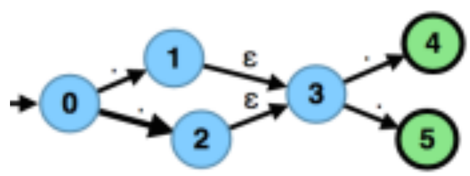
A_2:



Union:

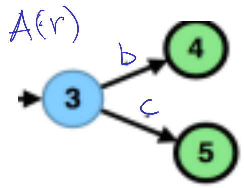


Concatenation:

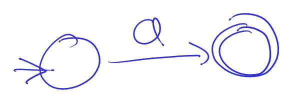


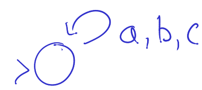
Recursive Constructions

r :



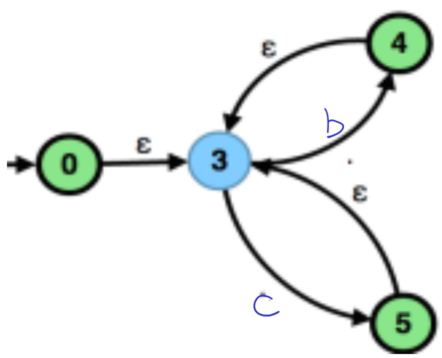
{b, c}





Star:

A(r*)

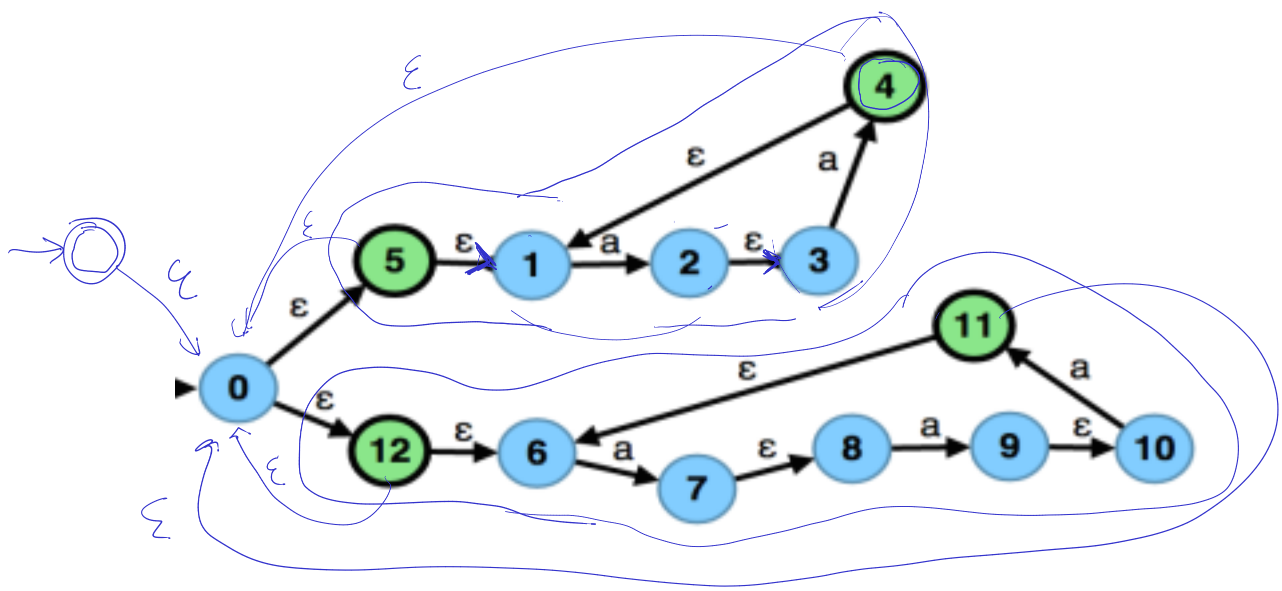


(b|c)*

Exercise: ((aa)* | (aaa)*)*

- Construct an NFA for the regular expression

ε
aa
aaaa
aaa
aaaaaa

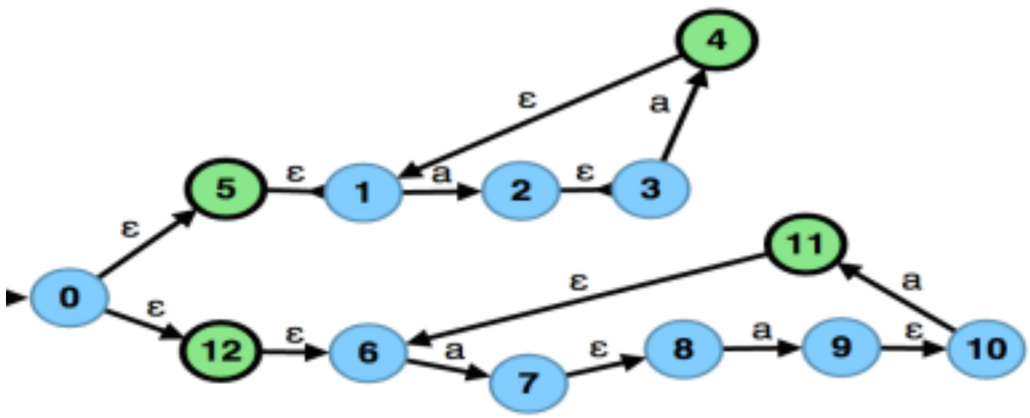


NFAs to DFAs (Determinization)

- 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

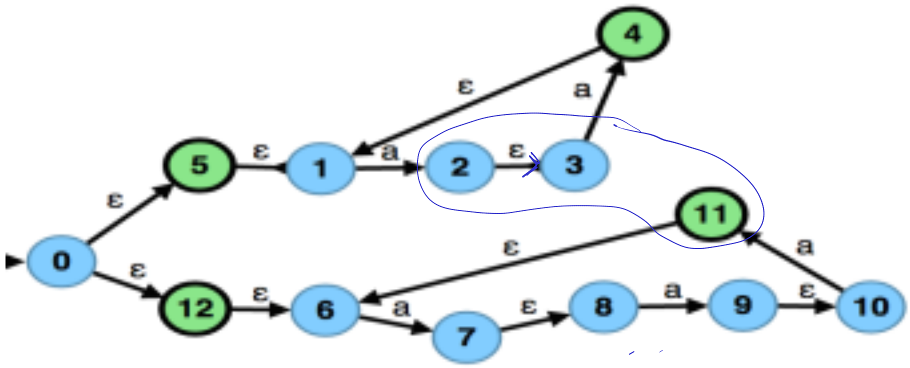


Possible states of the DFA: 2^Q

{ { } , { 0},...{12}, {0,1}, ...,{0,12}, ...{12, 12},
{0,1,2} ..., { 0,1,2...,12 } }

NFA to DFA Conversion

0,5,12,1,6



Epsilon Closure

–All states reachable from a state through epsilon

– q $\in E(q)$

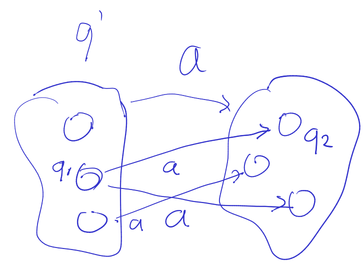
– If $q_1 \in E(q)$ and $\delta(q_1, \epsilon, q_2)$ then $q_2 \in E(q)$

E(0) = {0,5,12,1,6} E(1) = { 1} E(2) = {2,3}

NFA to DFA Conversion

(Σ, Q, δ, F)

- DFA: $(\Sigma, 2^Q, q_0', \delta', F')$

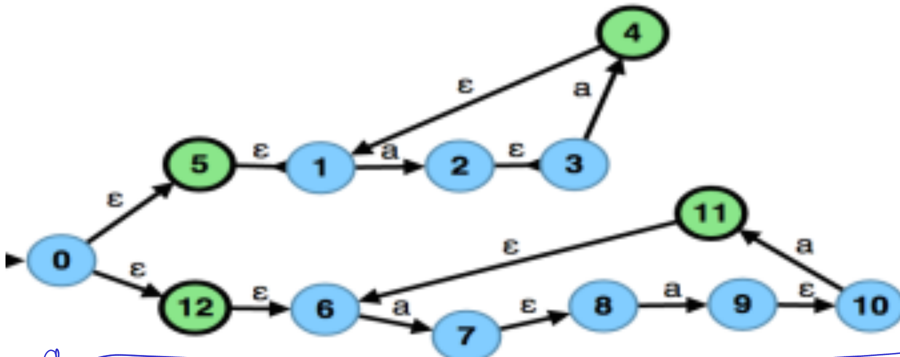


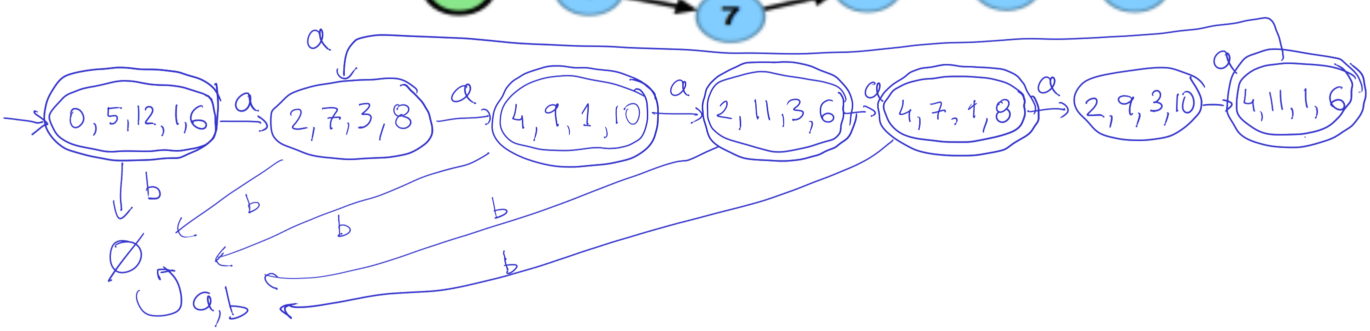
- $q_0' = E(q_0)$
- $\delta'(q', a) = \bigcup_{\{\exists q_1 \in q', \delta(q_1, a, q_2)\}} E(q_2)$
- $F' = \{\, q' \mid q' \in 2^Q, q' \cap F \neq \emptyset\}$

$q' \subseteq Q$

NFA to DFA Conversion through Examle

$\Sigma = \{a, b\}$





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.