Compiler Design

Department of software Engineering

Woldia University

Chapter Two

Lexical Analyzer

Lexical Analyzer

- The function of the lexical analyzer is to read the source program, one character at a time, and to translate it into a sequence of primitive units called *"tokens"*.
- how tokens are expressed using Regular Expression?
- regular grammars for generating languages.
- how Deterministic Finite State Automata recognize tokens?

Tokens

- Token represents a set of strings described by a pattern.
  - Identifier represents a set of strings which start with a letter continues with letters and digits
  - The actual string is called as *lexeme*.
  - Tokens: identifier, number, operations, delimiter, …
- Since a token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the *attribute* of the token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.
- Some attributes:

– <id, attr>	where attr is pointer to the symbol table
– <assg, op, _>	no attribute is needed (if there is only one assignment operator)
– <num, val>	where val is the actual value of the number.

- Token type and its attribute uniquely identifies a lexeme.
- ***Regular expressions*** are widely used to specify patterns.

Alphabet, String & Languages

- **Alphabets**:
  - An alphabet is a finite, nonempty set of symbols.
  - Conventionally, we use the symbol $\sum$ for an alphabet.
  - Common alphabet include:
    - $\sum = \{ 0, 1 \}$, the *binary* alphabet.
    - $\sum = \{ a, b, \ldots, z \}$, the set of all lower-case letters.
- **Strings**:
  - A string (or sometimes word) is a finite sequence of symbols chosen from some alphabet.
  - Example: 01101, 111, 0001, 111 … are strings from the binary alphabet $\sum = \{ 0, 1 \}$.

Alphabet, String & Languages

- **Empty string:**
  - The empty string is the string with zero occurrences of symbols and is denoted by ε. (i.e. the string consisting of no symbols)
- **Length of Strings:**
  - Let X be a string, the notation |X| denotes the *length* of X (i.e. the number of symbols contained in the string).
  - Example: |aba|=3, |a|=1, |ϵ |=0, etc.

Alphabet, String & Languages

- **Power of an alphabet:**
  - If ∑ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an *exponential notation*. We define $\sum^k$ to be the set of strings of length k, each of whose symbol is in ∑**.**
    - $\sum^0 = \{\epsilon\}$, regardless of what alphabet ∑ is.
    - If $\sum = \{0,1\}$, then $\sum^1 = \{0,1\}$, $\sum^2 = \{00, 01, 10, 11\}$, $\sum^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- The set of all strings over an alphabet ∑ is conventionally denoted $\sum^*$.
  - Example: $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$
  - $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \sum^3 \cup \cdots$

Alphabet, String & Languages

- The set of non empty strings from alphabet $\sum$ is denoted $\sum^{+}$ (excluding the empty string from the set of strings)
  - $\sum^{+} = \sum^{1} \cup \sum^{2} \cup \sum^{3} \cup \cdots$
  - $\sum^{*} = \sum^{+} \cup \{\varepsilon\}$.

Alphabet, String & Languages

- **Operation on strings**
  - **concatenation (product):**
    - Let x and y be strings. Then xy denotes the concatenation of x and y is, the string formed by making a copy of x and followed it by a copy of y.
    - *Example*: Let x= 01101 and y= 110, then xy= 01101110 (yx=11001101)
- **Note:** For any string w,

  εw = wε = w (i.e. ϵ is the identity for concatenation)

Alphabet, String & Languages

- **Languages:** A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a *language*.
- If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a Language over Σ.
- **Example:** A language L over an alphabet V is a subset of V*. For instance, if V= {a, b, c}, the following are languages on V.
  - $L_1 = \emptyset$ (the empty language; i.e. the empty subset of V)
  - $L_2 = \{\varepsilon\}$ (The language containing just the empty string; notice that $L_1 \neq L_2$)
  - $L_3 = \{a, b, c\} = V$ (the language whose elements are just the strings of length 1)
  - $L_4 = \{aa, ba, ab\}$

Alphabet, String & Languages

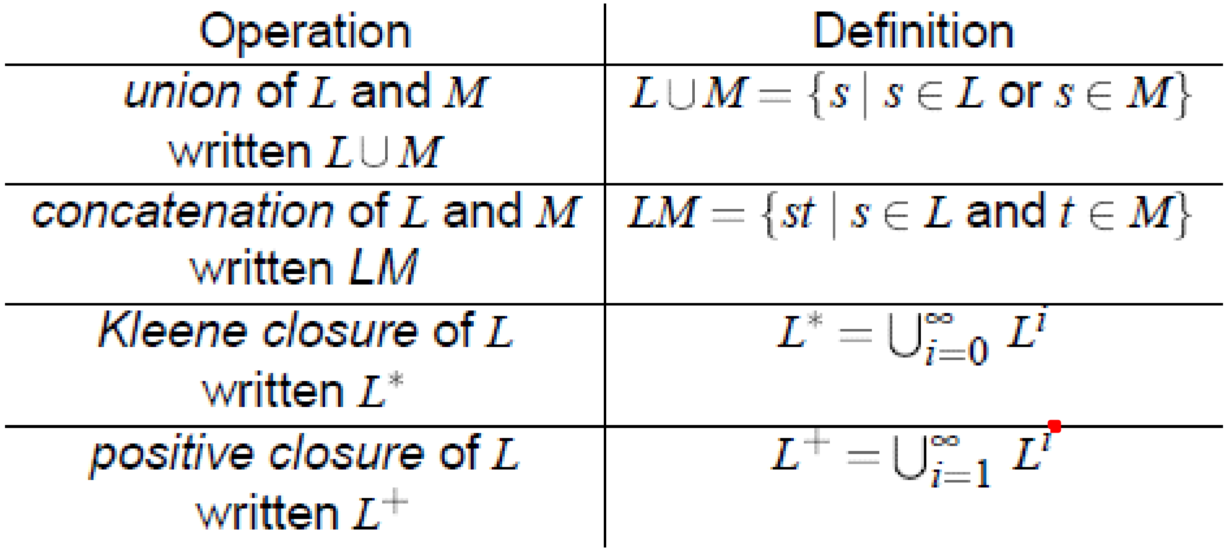
- L_5 = {a, aaa, aaaaa, bc}
- L_6 = {ab, aab, aaab, aaaab, …} (the infinite language whose strings consists of any number of a's followed by a single b; L_6 can also be defined in the more compact way $L_6 = \{a^n b | n \geq 1\}$)
- $L_7 = \{(ab)^n c^m | n \geq 1, m \geq 2\}$
- $L_8 = \{\{(a^n b^n | n \geq 1\} = \{ab, aabb, aaabbb, \ldots\}$

• **Note:** It's common to describe a language using a "set former"

{w| something about w} this expression is read "the set of words w such that (whatever is said about w to the right of the vertical bar)"

Alphabet, String & Languages

- Operations on languages



Operation	Definition
union of L *and* M written $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
concatenation of L *and* M written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular Expression (RE)

- A regular expression is a “user-friendly," declarative way of describing a regular language.
- We use regular expressions to describe tokens of programming language.
- A RE is built up of simpler regular expressions (using defining rules)
- Each RE denotes a language.
- A language denoted by a RE is called as a regular set.
- Regular expressions are used in e.g.
  1. UNIX grep command
  2. UNIX Lex (Lexical analyzer generator) and Flex (Fast Lex) tools.

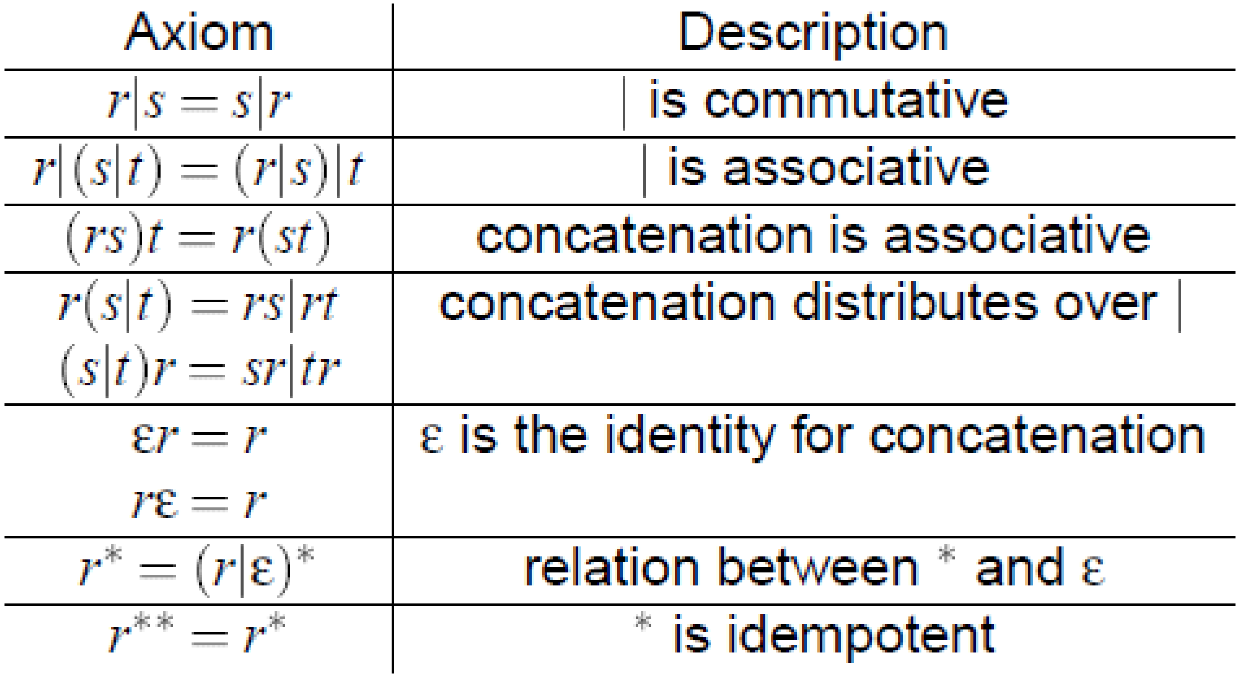
Definition: Regular Expressions

- **Regular Expressions** (RE) (over an alphabet ∑):
  - ε is a RE denoting the set {ε}
  - If a ∈ ∑**,** then a is RE denoting { a }
  - If r and s are Res, denoting L® and L(s), then
    1. (r) is a RE denoting L(r)
    2. (r)|(s) is RE denoting L(r) ∪L(s)
    3. (r)(s) is a RE denoting L(r)L(s)
    4. (r)* is RE denoting L(r)*

Regular Expression Operators

X Y concatenation	X followed by Y
X | Y alternation	X or Y (alternatives)
X * Kleene closure	Zero or more occurrences of X
X +	One or more occurrence of X
(X) grouping	Used for grouping (as in programming languages)

Algebraic properties of REs

- 

Axiom	Description
$r \vert s = s \vert r$	$\vert$ is commutative
$r \vert (s \vert t) = (r \vert s) \vert t$	$\vert$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s \vert t) = rs \vert rt$ $(s \vert t)r = sr \vert tr$	concatenation distributes over $\vert$
$\varepsilon r = r$ $r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r \vert \varepsilon)^*$	relation between * and ε
$r^{**} = r^*$	* is idempotent

Example

- Let ∑= {a,b}

1. a|b denotes {a, b}
2. (a|b)(a|b) denotes {aa, ab, ba, bb}
   i.e., (a|b)(a|b) = aa|ab|ba|bb
3. a* denotes {ε, a, aa, aaa, …}
4. (a|b)* denotes the set of all strings of a’s and b’s (including ε)
   i.e., (a|b)* = (a*|b*)*
5. a|a*b denotes {a, b, ab, aab, aaab, aaaab, …}

Describing Tokens by RE

- **digit** = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- **unsigned_integer** = digit digit*
- **signed_integer =** (+ | - | e) unsigned_integer
- **Letters** = A|B|C|…|Y|Z
- **Keywords =** BEGIN|END|IF|THEN|ELSE
- **Identifier** = letter (letter|digit)*
- Given two strings:
  - L = { a, b, c, ..., z }
  - D = { 0, 1, 2, ..., 9 }
    - L ((L | D)*) = “Set of strings that start with a letter, followed by zero or more letters and digits.”

RE Examples

- Given an Alphabet ∑={a,b}, construct a RE for:

  a) All strings beginning with *a*:

     a(a | b)*

  b) All strings containing *aba*:

     (a | b)*aba(a | b)*

  c) All strings of *even length*:

     $((a \mid b)(a \mid b))^* = (aa \mid ba \mid ab \mid bb)^* = ((a \mid b)^2)^*$

  d) All strings of *odd length*:

     $(a|b)((a \mid b)^2)^* = (a|b)\ (aa \mid ba \mid ab \mid bb)^*$

RE exercise

- Given an Alphabet ∑={0,1}, construct a RE for:

  Q1. The set of all strings which have at least one occurrence of the substring 001.

  Q2. The set of all strings that contain an even number of 0s or an even number of 1s.

  Q3. the set of all strings with an even number of 0's followed by an odd number of 1's.

  Q4. The set of all strings whose fifth symbol from right is 0.

  Q5. The set of all strings that start with 0 and end with 1.

Regular Grammars

- A grammar is a list of rules which can be used to produce or generate all the strings of a language, and which does not generate any strings which are not in the language.
- Grammar: generative description of a language
- Automaton: analytical description.
- A ***grammar*** is a quadruple

  $G = (V, T, S, P)$ where

  - V is a finite set of ***variables***
  - T is a finite set of symbols, called ***terminals***
  - S is in V and is called the ***start symbol***
  - P is a finite set of ***productions***, which are ***rules.***

Regular Grammars

- **Notation**:
  - *Terminals* (lower-case letters, operator symbols, digits, keywords, Punctuation symbols, etc…)
  - *Non-Terminals* (Upper-case letters, special symbols such as statement, expression, A, B, C and etc…)
- In a regular grammar, all *productions* have one of two forms:
  1. $A \rightarrow aA$
  2. $A \rightarrow a$

  Where A is any *non-terminal* and a is any *terminal* symbol.

Example

1. $S \rightarrow abS \mid a$

   Can you figure out what language it generates?

   - ***L* = {*w*** ϵ ***{a,b}** ** **| *w*** contains alternating ***a***'s and ***b***'s , begins with an ***a***, and ends with a ***b*}** ∪ ***{a}***
   - ***L((ab)*a)***

2. S → aaA

   A → abA | aB

   B → b

   Can you figure out what language it generates?

   - ***L* = {*w*** ϵ ***{a,b}** ** **| *w*** is ***aa*** followed by at least one set of alternating ***ab***'s**}**
   - ***L(aaab(ab)*)***

Finite Automata/Machine (FA)

- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a finite automata.
- A finite automata can be:
    - Deterministic FA (DFA) or
    - Non-deterministic FA (NFA)
- This means that we may use a deterministic or non-deterministic automata as lexical analyzer.
- Both deterministic and non-deterministic automata recognize regular sets.

FA

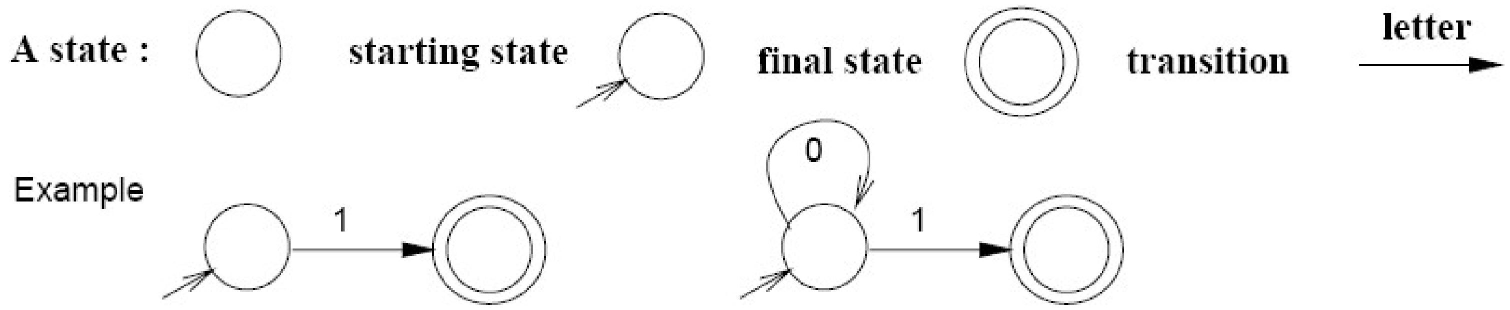
- Which one?
  - Deterministic – faster recognizer, but it may take more space
  - Non-deterministic – slower, but it may take less space.
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

Conti…

Language	Machine	Grammar
Regular	Finite Automaton	**Regular Expression, Regular Grammar**
Context-Free	Pushdown Automaton	Context-Free Grammar
Recursively Enumerable	Turing Machine	Unrestricted Phrase-Structure Grammar

FA Representation

- A finite state automata is a model of behavior composed of finite number of states, transitions between those states and actions.
- **FA components:**



Formal Definition of FA

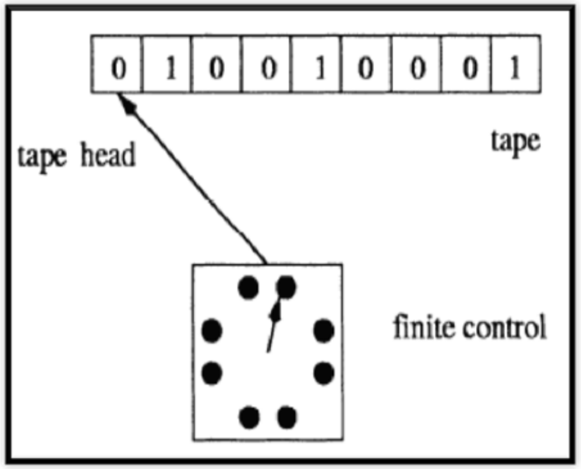
An finite automaton is a 5-tuple = $(\Sigma, Q, q_0, F, \delta)$

- $\sum$ is a finite set called the **alphabet**,
- Q is a finite set called **states,**
- $q_0 \in Q$ is the **start state**,
- $F \subseteq Q$ is the set of **final states** (Accept states)
- A transition function:

$$\delta : Q \times \Sigma \longrightarrow Q$$

How Machine M operates.

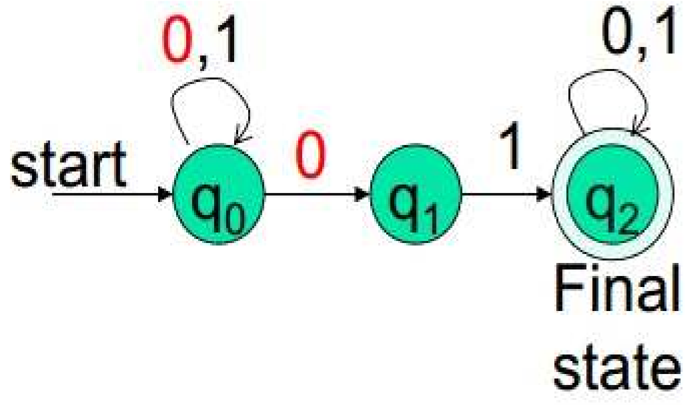
- M "reads" one letter at a time from the input string (going from left to right)
- M starts in state q_0.
- If M is in state q_i reads the letter a then
  - If $\delta(q_i, a)$ is undefined then CRASH.
  - Otherwise M moves to state $\delta(q_i,a)$



- The output of a finite automaton is "accepted" if the automaton is now in an accept state (double circle) and reject if it is not.

Con't

- We can describe the given FA (M1) formally by writing M 1 = (Q, ∑, δ, ql, F), where



- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

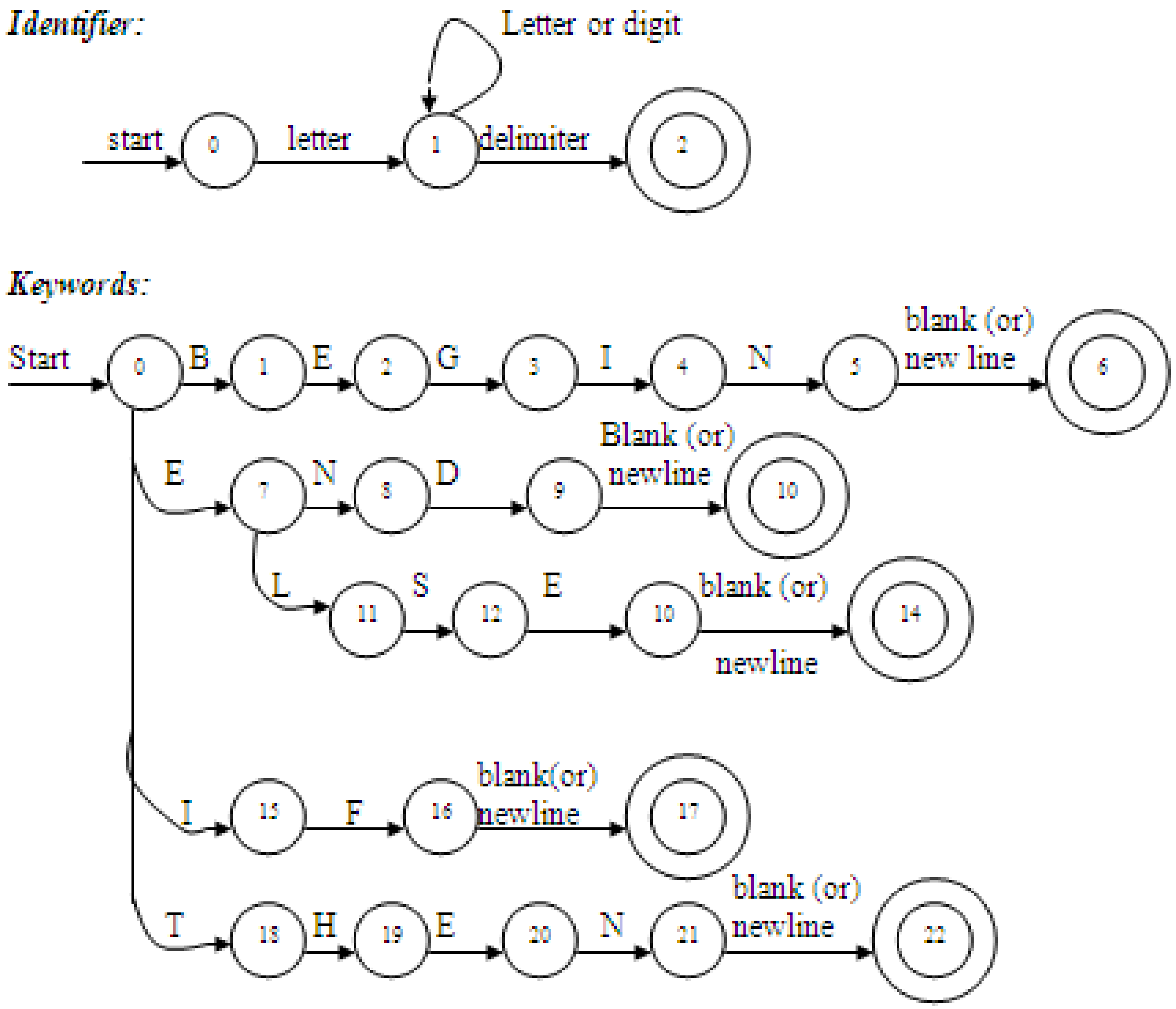
symbols

δ	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$

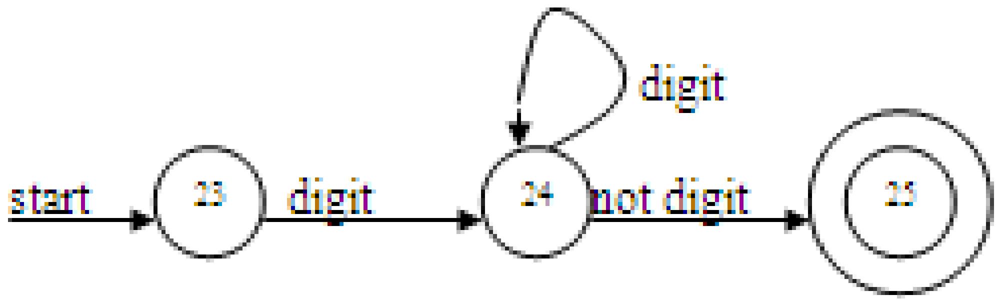
states

FA for recognizing Tokens

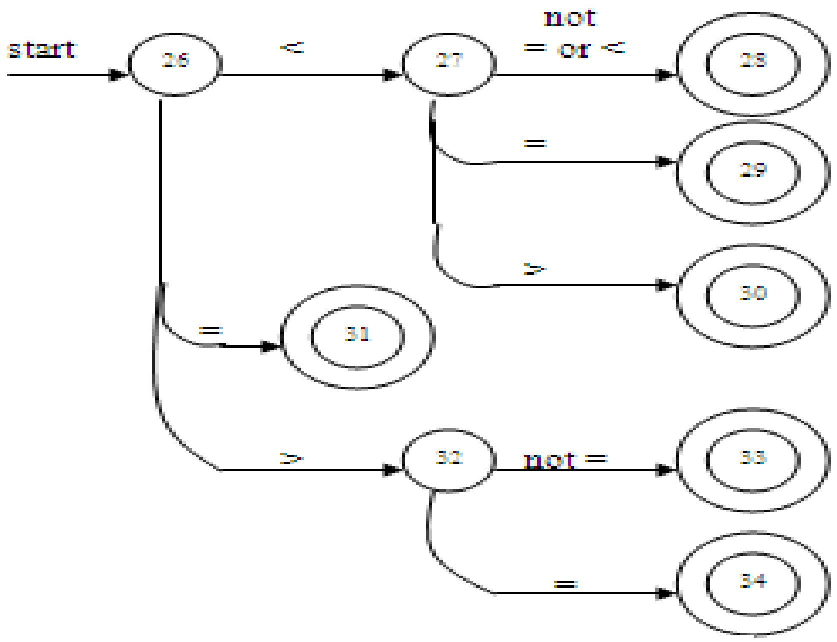
•



Constant:



Relops:

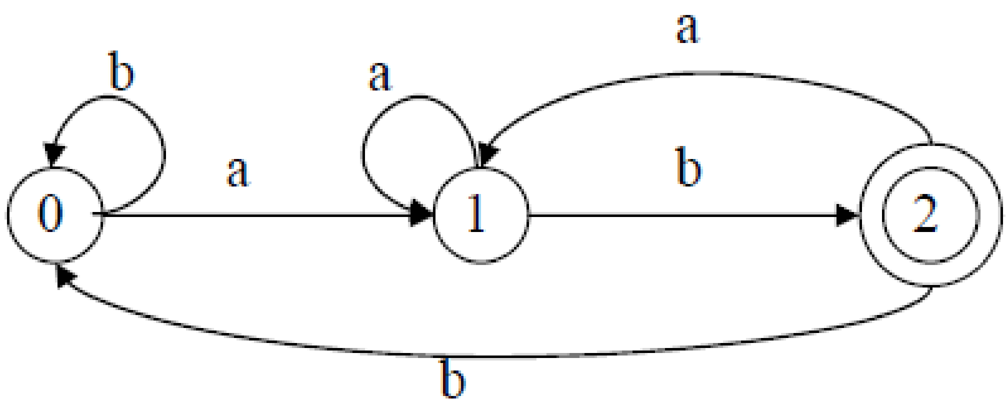


DFA Vs NFA

- When the machine is in a given state and reads the next input symbol, we know what the next state will be – it is determined.
- In nondeterministic machine, several choices may exist for the next state at any point.
- Non-determinism is a generalization of determinism, so every *deterministic finite automaton* is automatically a *non-deterministic finite automaton.*
- DFAs are clearly a subset of NFAs.

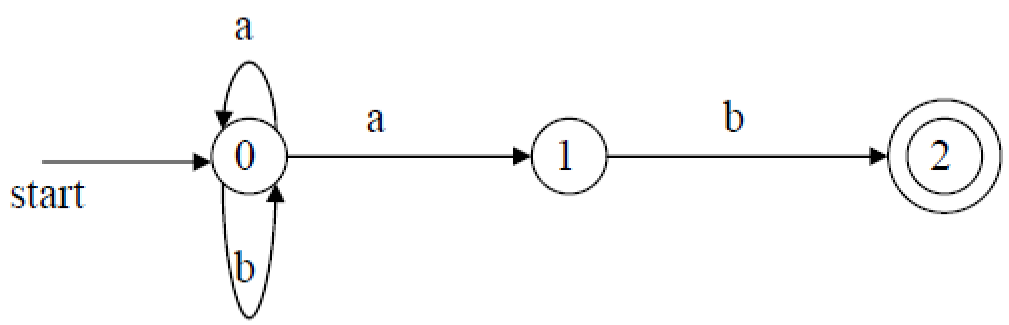
DFA

- Every state of a DFA always has exactly one existing transition arrow for each symbol in the alphabet. (one transition per input per state)
- No **ε**-moves.
- **Example**: The DFA to recognize the language **(a|b)* ab** is as follows:

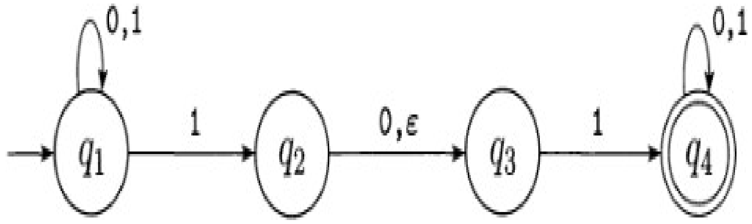

$$\begin{tikzcd} 0 \arrow[loop above, "b"] \arrow[r, "a"] & 1 \arrow[loop above, "a"] \arrow[r, "b"] & 2 \arrow[l, bend right, "a"'] \arrow[ll, bend left, "b"] \end{tikzcd}$$

NFA

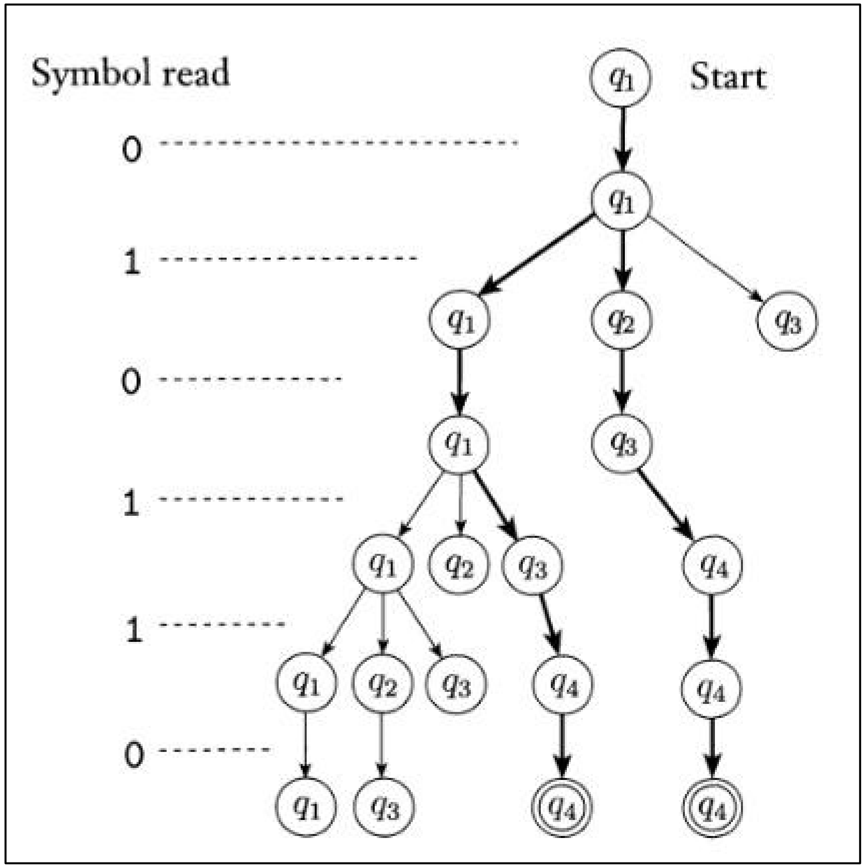
- In any NFA a state may have zero, one, or many existing arrows for each alphabet symbol.
- Can have **ε**-moves. (in other words, we can move from one state to another one without consuming any symbol.)
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states that edge labels along this path spell out x.
- Example: The NFA to recognize the language **(a|b)* ab** is as follows:



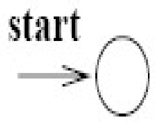
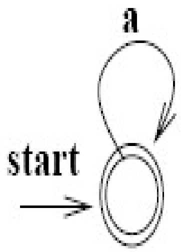
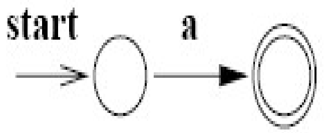
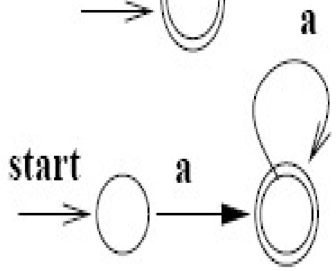
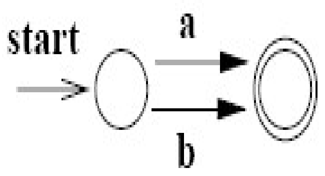
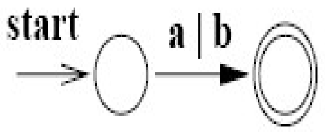
How does an NFA computes?



Computation

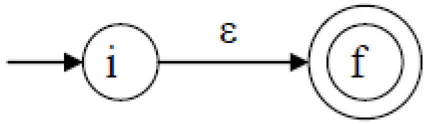


From Regular Expression to DFA

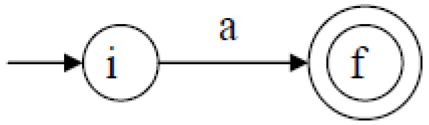
Regular Exp.	DFA	Regular Exp.	DFA
e		a*	
a		a+	
a | b	 ≡ 		

RE to NFA (Thomson Construction)

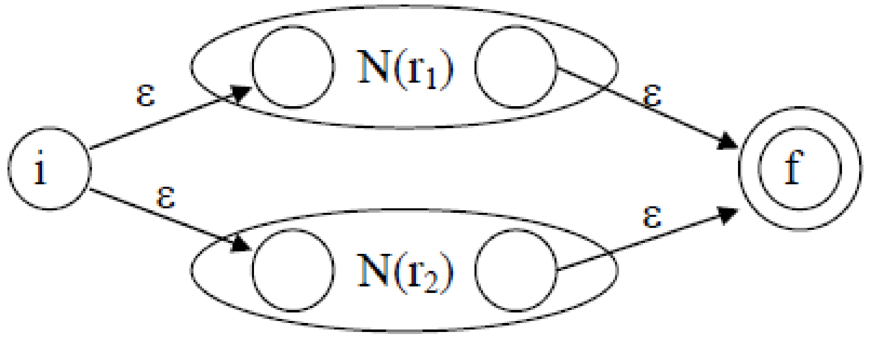
1. To recognize an empty string **ε:**



2. To recognize a symbol **a** in the alphabet ∑:

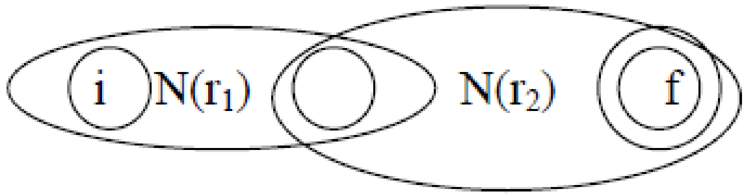


3. For regular expression **r1| r2** : (N(r1) and N(r2) are NFAs for regular expressions r1 and r2)

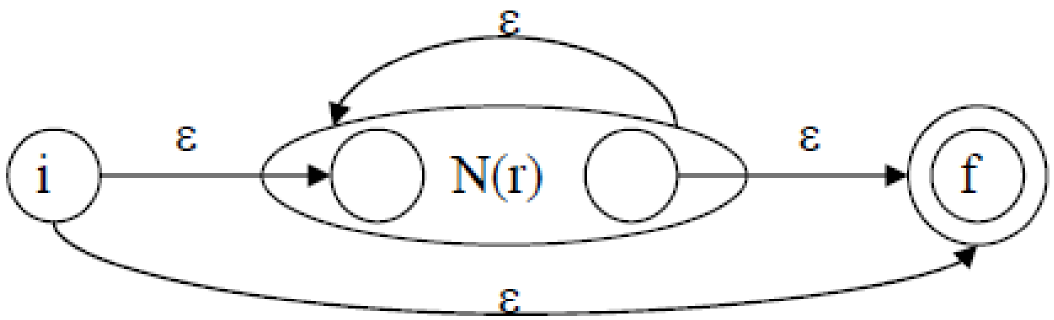


RE to NFA (Thomson Construction)

4. For regular expression **r1r2**:

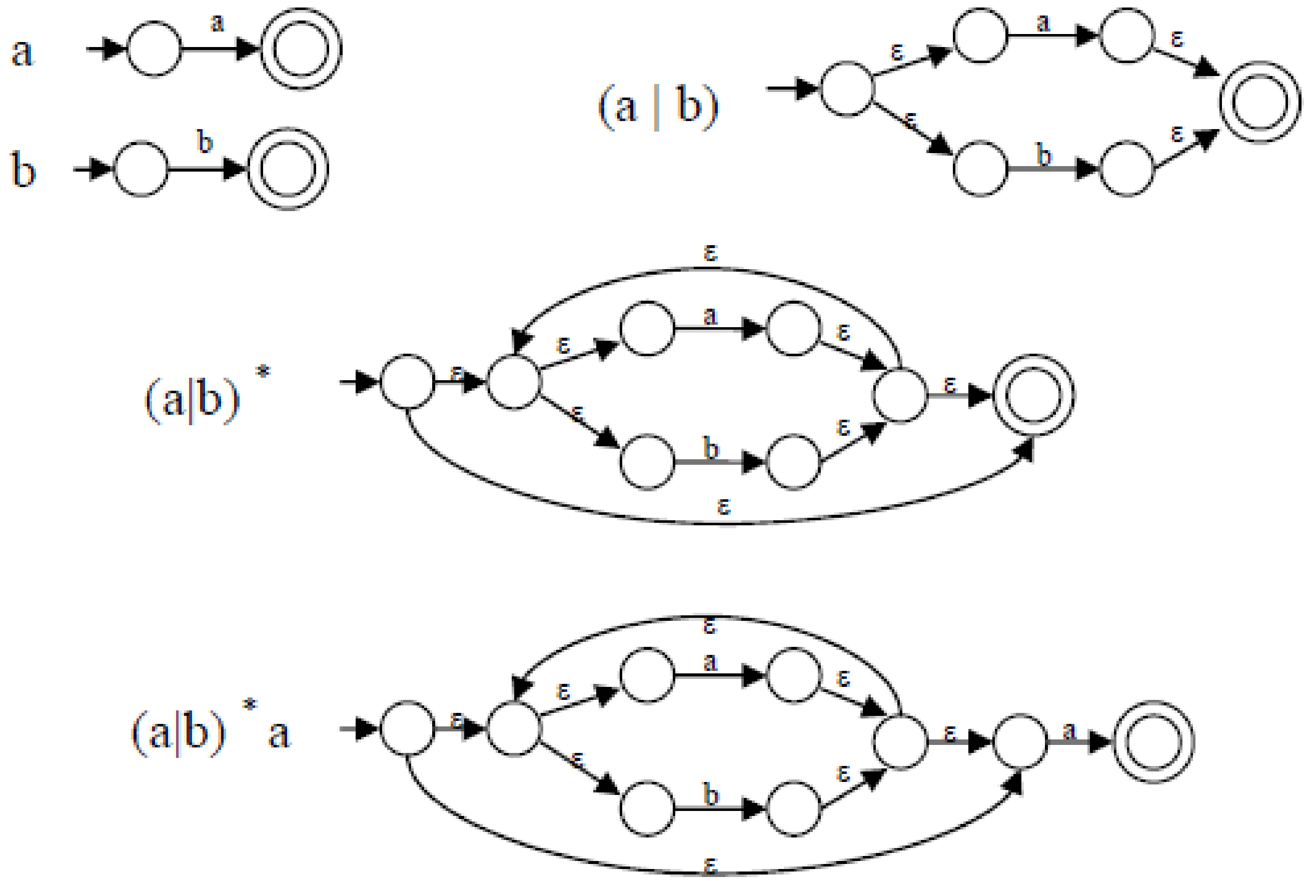


5. For regular expression **r***:



RE to NFA: Example

- For a RE **(a|b)* a**, the NFA construction is shown below.

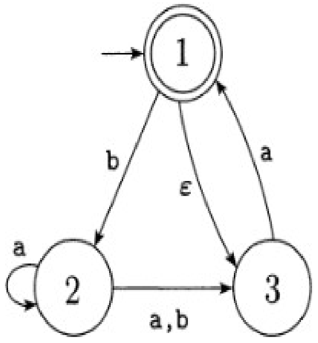


NFA to DFA

- The conversion from NFA to DFA:
  - Create a new state for each equivalent class in NFA.
  - The max number of states in DFA is $\mathbf{2^N}$, where **N** is the number of states in NFA.
- Steps to construct DFA that is an equivalent a given NFA:
  a. First determine DFA's states.
  b. Then, Determine the start and accept states of the DFA.
  c. Finally, determine DFA's transition function.

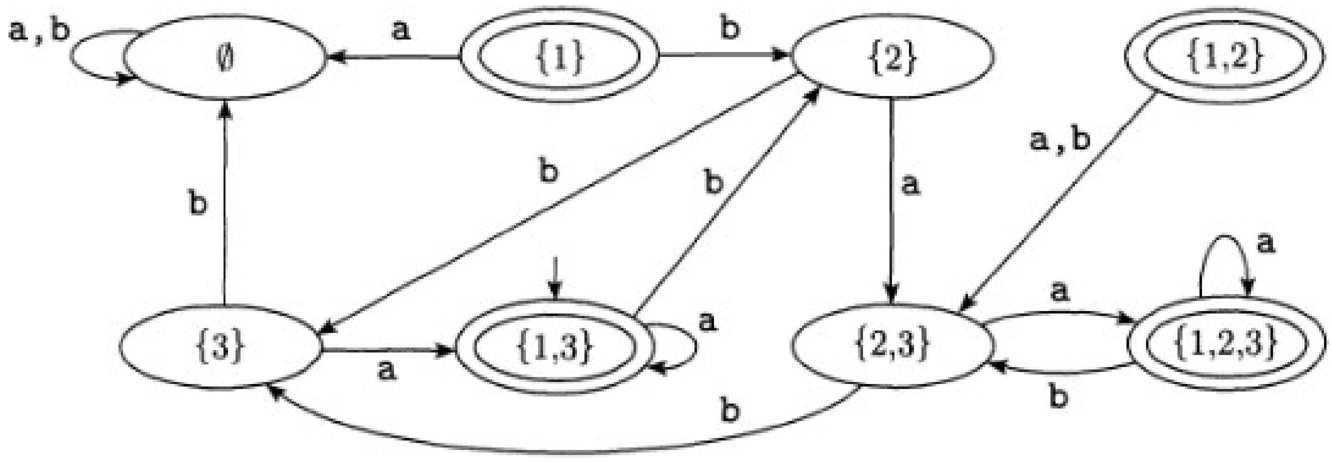
Example:

Construct an equivalent DFA from the given NFA.



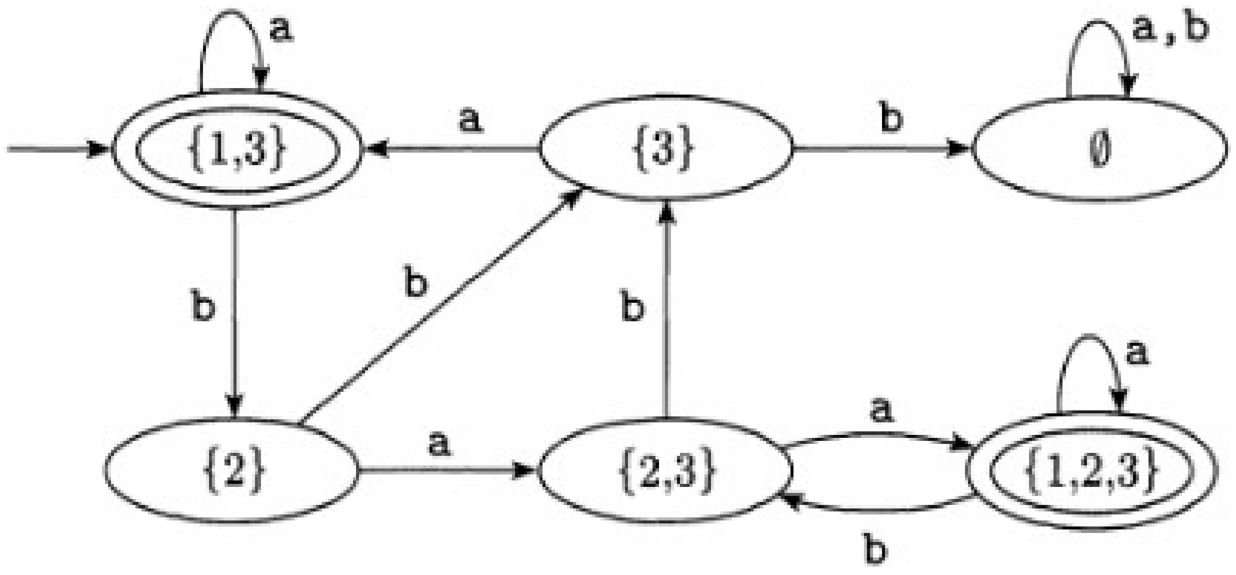
- **Step 1**: Determine DFA's number of states:
    - NFA {1, 2,3 } → DFA {**∅, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}**}.
- **Step 2**: Determine the start and accept states of DFA:
    - **Start states**: the set of states that are reachable from NFA's start state (1) by traveling ε arrow, plus the start state of NFA (1). Therefore **{1,3} are start state.**
    - **Accept states:** The new accept states (of DFA) are those containing NFA's accept state; thus {{1}, {1,2}, {1,3}, {1,2,3}}
- **Step 3: D**etermine DFA's transition function.

State	a	b
∅	∅	∅
{1}	∅	{2}
{2}	{2,3}	{3}
{3}	{1,3}	∅
{1,2}	{2,3}	{2,3}
{1,3}	{1,3}	{2}
{2,3}	{1,2,3}	{3}
{1,2,3}	{1,2,3}	{2,3}



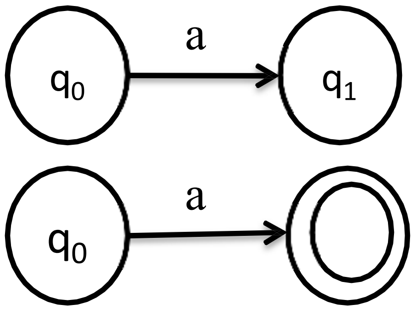
Con’t

•



From DFA to Regular Grammar(RG)

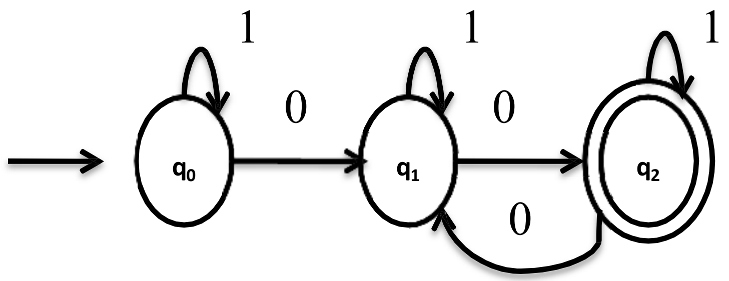
- We can determine a RG directly from a DFA.
- Rules:



$q_0 \to aq_1$

$q_0 \to a$

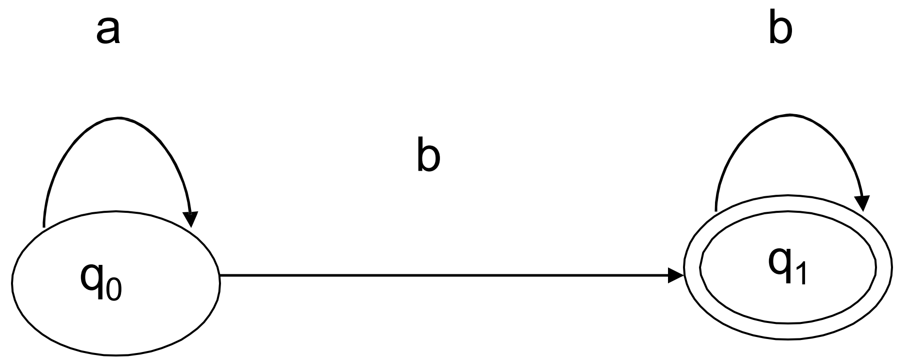
- Example:



$q_0 \to 1q0|0q1$
$q_1 \to 1q1|0q2$
$q_2 \to 1q2|0q1|\ \varepsilon$

FA Vs RE Vs RL Vs RG

FSA:



Regular language: {b, ab, bb, aab, abb, …}

Regular expression: $a^* \, b^+$

Regular grammar:

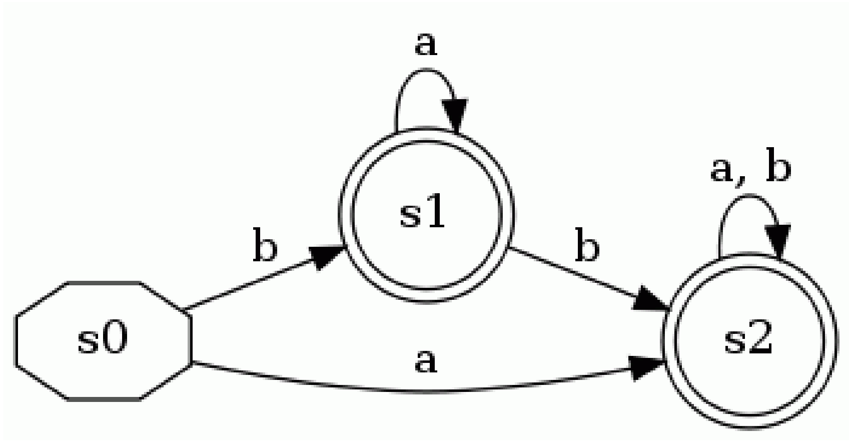
q_0 ➔ $a \; q_0$
q_0 ➔ $b \; q_1$
q_1 ➔ $b \; q_1$
q_1 ➔ ^{2}

DFA Minimization

- Questions of DFA size:
  - Given a DFA, can we find one with fewer states that accepts the same language?
  - What is the smallest DFA for a given language?
  - Is the smallest DFA unique, or can there be more than one "smallest" DFA for the same language?
- All these questions have neat answers…
- The task of *DFA minimization*, then, is to automatically transform a given DFA into a state-minimized DFA
  - Several algorithms and variants are known
  - Note that this also in effect can minimize an NFA (since we know algorithm to convert NFA to DFA)

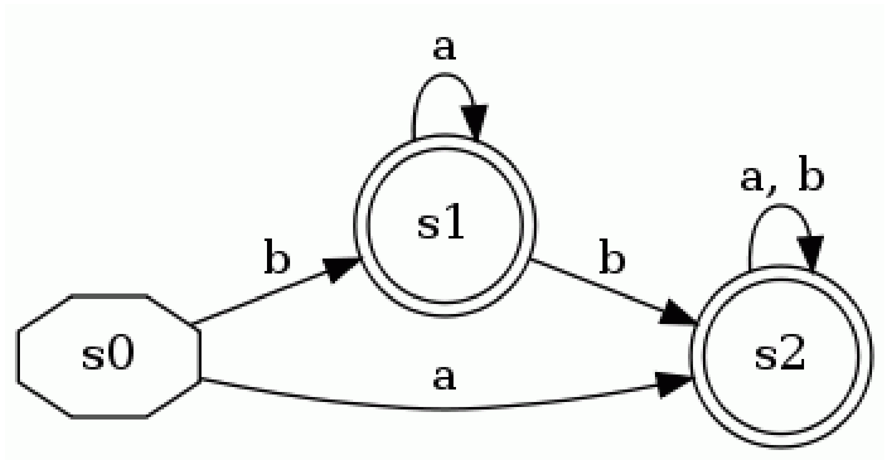
DFA Minimization

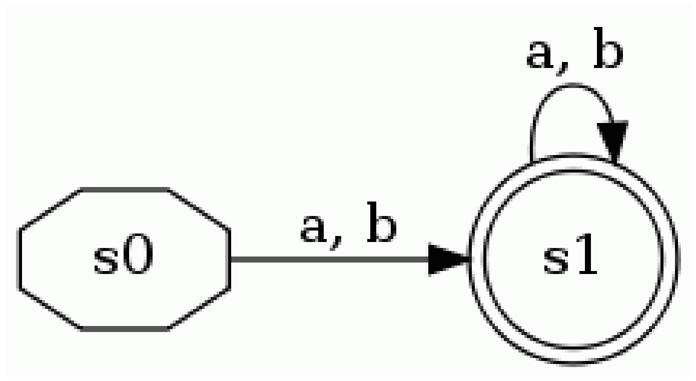
- Some states can be redundant:
  - The following DFA accepts **(a|b)+**
  - State s1 is not necessary



DFA Minimization

- So these two DFAs are *equivalent*:





State Reduction by Partitioning

- We say two states **p** and **q** are **equivalent** (or indistinguishable), if, for every string $w \in \Sigma^*$, transition $\delta($ ***p*** *, w) ends in an accepting state if and only if* $\delta($ ***q*** *, w) does*. In the preceding slide states S_1 and S_2 are equivalent.
- There are efficient algorithms available for computing the sets of equivalent states of a given DFA.
- The following two slides show:
  - the detailed steps for computing equivalent state sets of the DFA
  - constructing the reduced DFA.

State Reduction by Partitioning

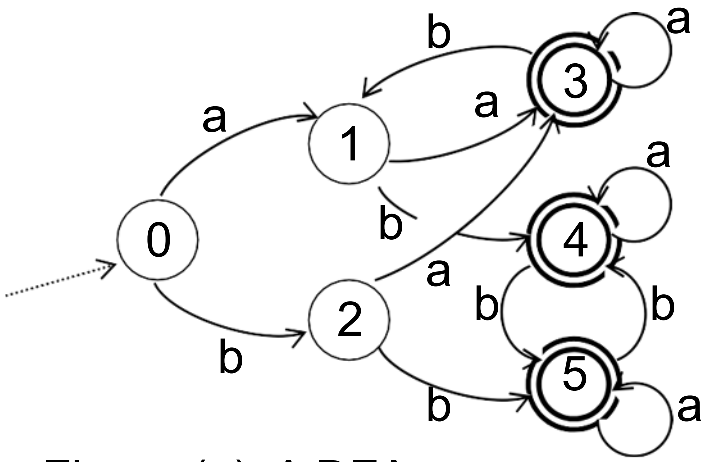


Figure (a) A DFA

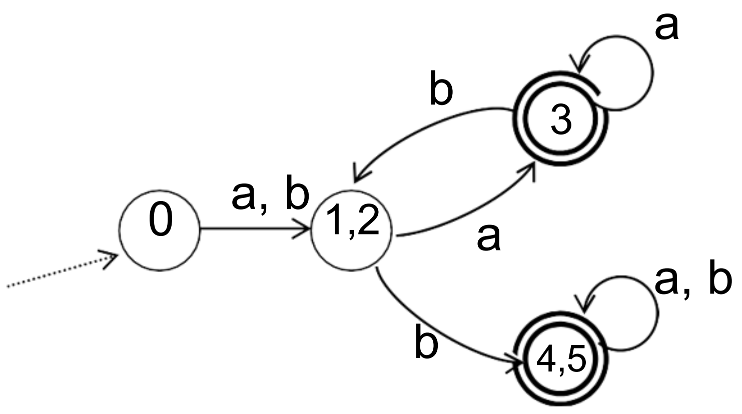


Figure (b) Reduced DFA

- **Step 0**: Partition the states according to accepting/non-accepting.

P_1	P_2
{ 3, 4, 5 }	{ 0, 1, 2 }

State Reduction by Partitioning(cont’ed)

- **Step 1**: Get the response of each state for each input symbol. Notice that States 3 and 0 show different responses from the ones of the other states in the same set.

$$\begin{array}{lccc}
\multicolumn{4}{c}{\underline{\qquad\boldsymbol{P_1}\qquad}} \\
 & p_1 & p_1 & p_1 \\
\mathbf{a}\rightarrow & \uparrow & \uparrow & \uparrow \\
 & \{3, & 4, & 5\,\} \\
\mathbf{b}\rightarrow & \downarrow & \downarrow & \downarrow \\
 & p_2 & p_1 & p_1
\end{array}
\qquad\qquad
\begin{array}{lccc}
\multicolumn{4}{c}{\underline{\qquad\boldsymbol{P_2}\qquad}} \\
 & p_2 & p_1 & p_1 \\
\mathbf{a}\rightarrow & \uparrow & \uparrow & \uparrow \\
 & \{0, & 1, & 2\,\} \\
\mathbf{b}\rightarrow & \downarrow & \downarrow & \downarrow \\
 & p_2 & p_1 & p_1
\end{array}$$

Record responses for each input symbol

•**Step 2**: Partition the sets according to the responses, and go to Step 1 until no partition occurs.

$$\begin{tikzcd}
 & \boldsymbol{P_{11}} & & \boldsymbol{P_{12}} & & \boldsymbol{P_{21}} & & \boldsymbol{P_{22}} \\
 & p_{11} & p_{11} & & & p_{12} & p_{12} & \\
a\rightarrow & & & & a\rightarrow & & & \\
 & \mathbf{\{4,} \arrow[uu] \arrow[dd] & \mathbf{5\}} \arrow[uu] \arrow[dd] & \mathbf{\{3\}} & & \mathbf{\{1,} \arrow[uu] \arrow[dd] & \mathbf{2\}} \arrow[uu] \arrow[dd] & \mathbf{\{0\}} \\
b\rightarrow & & & & b\rightarrow & & & \\
 & p_{11} & p_{11} & & & p_{11} & p_{11} &
\end{tikzcd}$$

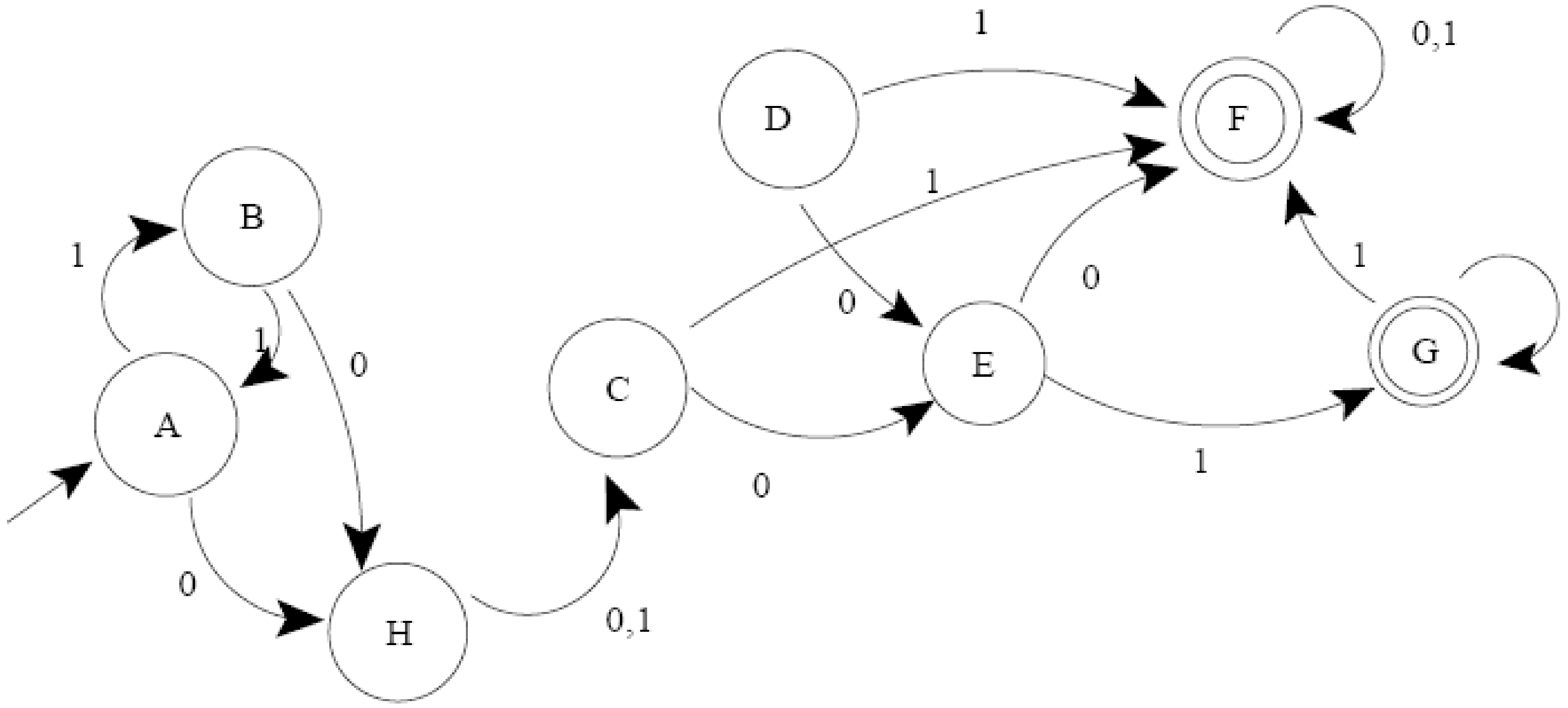
Partition the set, and record responses for each input symbol

•No further partition is possible for the sets $\boldsymbol{P_{11}}$ and $\boldsymbol{P_{21}}$. So the final partition results are as follows.

{4, 5} {3} {1, 2} {0}

Exercise

- Minimize the given DFA.



A
B
C
D
E
F
G
H
0
1
0,1