THEORY OF COMPUTATION
UNIT 1

Learning Objective

- Explain the concepts of computation and their applications.
- Describe how does a compiler work and what are the phases of a typical compiler.
- Explain the concept of automata theory and finite automata .
- Explain the concepts of DFA, NFA, Mealy and Moore machine.
- Discuss the concept of regular language and grammar and their properties.

Introduction

Learning Objective

- Understand the purpose of theory of computation
- Distinguish between decidable and undecidable problems
- Understand the concept of tractable and intractable problem
- Applications of theory of computation
- Describe various phases of compiler construction

Introduction

- What do you mean by computation?
  - ❑ To process the sequence of well defined operations
  - ❑ Fewer and simpler operations are better.
- What computer can do?
  - ❑ Decidable problem
- What computer can not do?
  - ❑ Undecidable problem
    - ✓ Post Correspondence problem
    - ✓ Turing Machine halting problem

Introduction

- What do you mean by efficiency of an algorithm?
  - The amount of resources used by an algorithm
    - ✓ Time
    - ✓ Space
- How efficient the algorithm is ?
  - Computation can be performed in Polynomial time
    - ✓ Tractable problem
  - If not then
    - ✓ Intractable problem
      - ➢ Knapsack problem
      - ➢ Travelling Salesman Problem

What is a Theory of computation?

- **Theory of computation** is the branch that deals with how efficiently problems can be solved on a model of computation , using an algorithm.

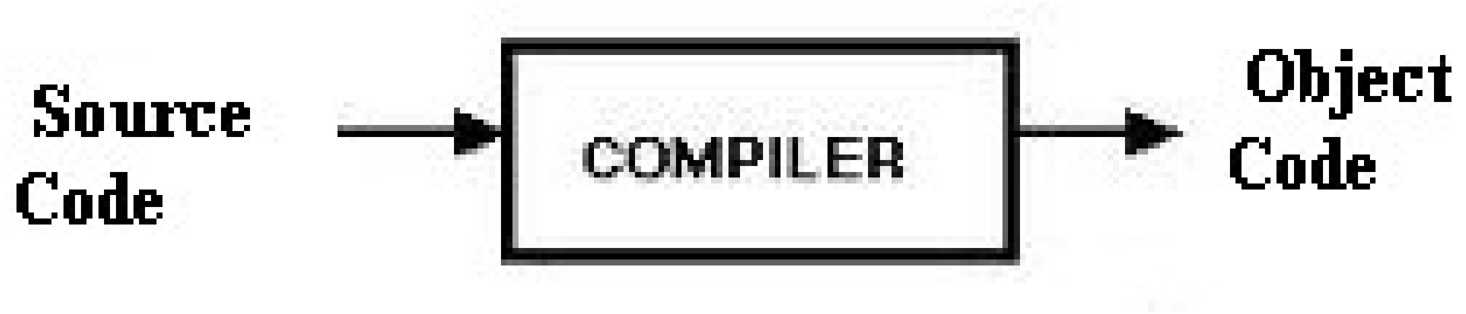
- Theory of computation:
  - Automata Theory
  - Computability Theory
  - Computational complexity theory

Applications of TOC

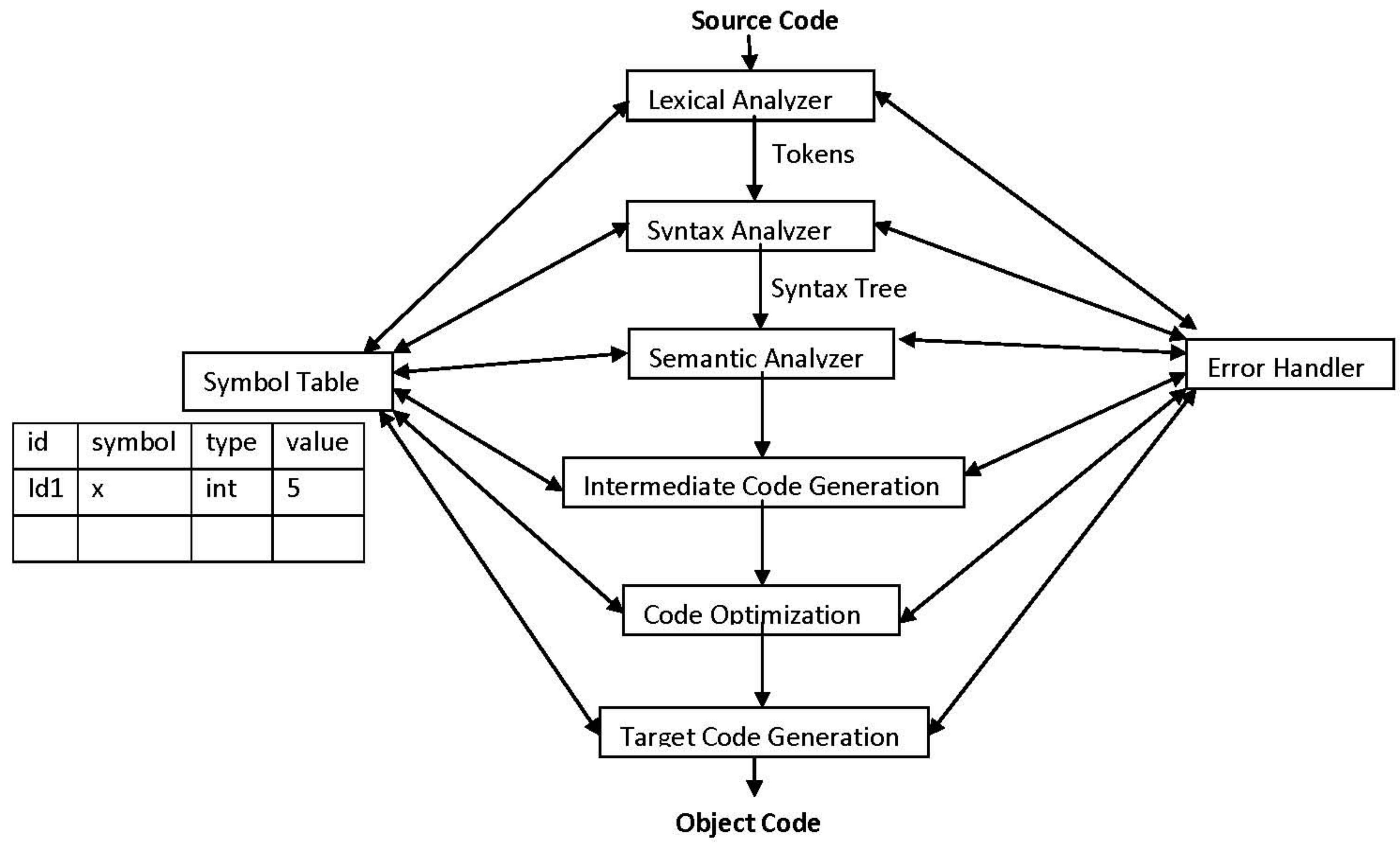
1. Text processing
   - String matching
2. Verification through model checking
3. Compiler design
4. Circuit Design
5. DNA computing
6. Artificial intelligence

Typical compiler and their stages

- What do you mean by compiler?
  - A compiler is a set of programs that transforms source code written in a programming language into another computer language (object code).



Typical compiler and their stages



id	symbol	type	value
Id1	x	int	5

Phases of Compiler

- Compiler can be broadly categorized into following two steps
  - Analysis of a source program
  - Synthesis of a source program
- Analysis of source program:
  - The analysis step itself consists of three phase.
    - ✓ Lexical analysis
    - ✓ Syntax analysis
    - ✓ Semantic analysis
  - A phase is independent task in the compilation process.

Phases of Compiler

- Synthesis of source program:
  - The synthesis step itself consists of two phase.
    - ✓ Code optimization
    - ✓ Code generation
- There are two other activities or phases which interact with all the other phases of compiler.
  - **Symbol Table management**: Symbol table records the identifiers used in the source program and also the information about various attribute like its type , scope etc.
  - **Error Detection and Handling** : Whenever a phase encounters an error, it must report it. It must deal with the errors like
    - When lexical analyzer do not form any token,
    - When token string violates the structural rule of the language.
    - Type mismatch in symatic analysis

Analysis of source program

- **Lexical Analysis**
  - Reads the stream of characters from left to right and groups ther into tokens.
  - A token is a sequence of characters having a collective meaning like keyword, identifier, constants, operator etc. .
  - It removes comments and white spaces from the source progran
  - It also makes a copy of the source program with the error message .
  - Each error message may also be associated with a line number.

Analysis of source program

- **Syntax Analysis**
  - Syntax analysis or parsing is a major component of the front end of compiler
  - It obtains a string of tokens from the lexical analyzer.
  - It groups the tokens and verify that the string can be generated by the grammar (context-free) for the source language.
  - It reports any syntax error in program
- **Semantic Analysis**
  - Determines the meaning of the source string.
  - It finds out that the types of operand are identical, the scope of the operand , name etc. .
  - It can convert one data type to another.

Intermediate Code Generation

- **Intermediate code generator**
  - It is not always possible to generate target code in one pass therefore intermediate code is generated.
  - Takes a parse tree from the semantic analyzer
  - Generates a program in the intermediate language.
  - Using intermediate code is beneficial when compilers which translates a single source language to many target languages are required.
    - ✓ The front-end of a compiler – *scanner to intermediate code generator* – can be used for every compilers.
    - ✓ Different back-ends – *code optimizer and code generator*– is required for each target language.
  - One of the popular intermediate code is *three-address code*. A three-address code instruction is in the form of $x = y \ op \ z$.

Synthesis of source program

- **Code optimization**
  - It tries to improve the intermediate code to achieve faster running machine code without changing the algorithm.
  - It must ensures that the transformed program is semantically equivalent to the original program.

- **Code Generation**
  - It generates the target code
  - Allocate memory for each of the variables
  - Intermediate instruction are then translated into a sequence of machine instructions.

Example of compilation

```
position := initial + rate * 60
```

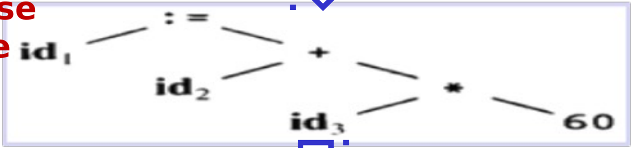
Scanner
[Lexical Analyzer]

Tokens

id_1 := id_2 + id_3 * 60

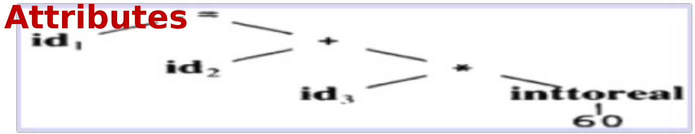
Parser
[Syntax Analyzer]

Parse tree



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes



SYMBOL TABLE

1	position	. . .
2	initial	. . .
3	rate	. . .
4		

Code Generator
[Intermediate Code Generator]

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimizer

Optimized Intermediate C

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Optimizer

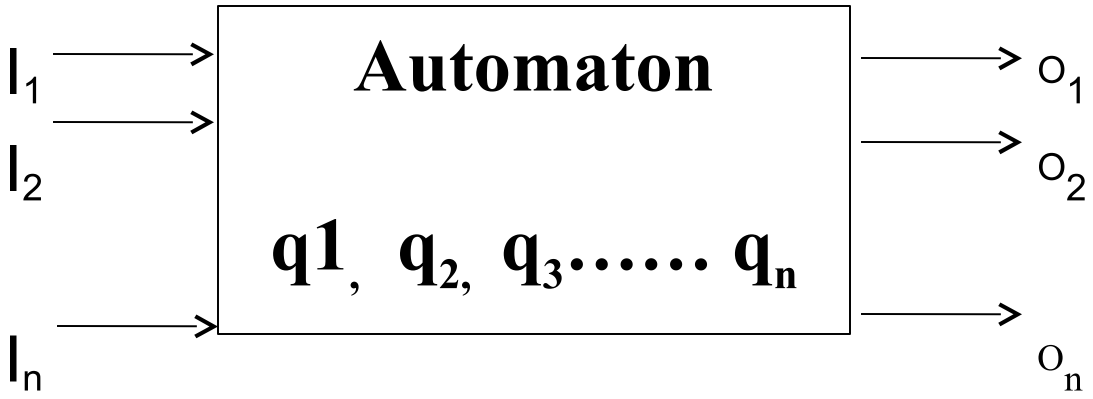
Target machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Finite Automata

- What is Automaton?

  An automaton is a system where energy, material and information are transformed, transmitted and used for performing some function without direct participation of man.



Basic Concepts

- An ***alphabet*** is a finite, non-empty set of symbols.
  - {*0*,*1* } is a binary alphabet.
  - { *A*, *B*, …, *Z*, *a*, *b*, …, *z* } is an English alphabet.
- A ***string*** over an alphabet Σ is a sequence of any number of symbols from Σ.
  - *0*, *1*, *11*, *00*, and *01101* are strings over {*0*, *1* }.
  - *Cat*, *CAT*, and *compute* are strings over the English alphabet.

Basic Concepts (contd.)

- An ***empty string***, denoted by ε or λ , is a string containing no symbol.
  - λ is a string over any alphabet.
- The ***length of a string*** *x*, denoted by *length*(*x*) or |x|, is the number of positions of symbols in the string.

Let $\Sigma = \{a, b, \ldots, z\}$

$length(automata) = 8$

$length(computation) = 11$

$length(\varepsilon) = 0$

- $x(i)$, denotes the symbol in the i^{th} position of a string x, for $1 \leq i \leq length(x)$.

Basic concepts(Cont.)

- The ***concatenation*** of strings x and y, denoted by $x \cdot y$ or $x\ y$, is a string z such that:
  - $z(i) = x(i)$ for $1 \leq i \leq length(x)$
  - $z(i) = y(i)$ for $length(x) < i \leq length(x) + length(y)$

- Example
  - $automata \cdot computation = automatacomputation$

Basic concepts(Cont.)

The ***concatenation*** of string x for n times, where $n \geq 0$, is denoted by x^n

$x^0 = \varepsilon$

$x^1 = x$

$x^2 = x\ x$

$x^3 = x\ x\ x$

…

Basic concepts(Cont.)

Substring

Let x and y be strings over an alphabet Σ

The string x is a substring of y if there exist strings w and z over Σ such that $y = w\,x\,z$.

- ε is a substring of every string.
- For every string x, x is a substring of x itself.

Example

- ε, *comput* and *computation* are substrings of *computation*.

Basic concepts(Cont.)

Let x be a string over an alphabet Σ

The ***reversal of the string*** x, denoted by x^r, is a string such that

- if x is ε, then x^r is ε.
- If a is in Σ, y is in Σ^* and $x = a\ y$, then $x^r = y^r\ a$.

Basic concepts(Cont.)

Example of substring:

$$
\begin{aligned}
& (automata)^r \\
&= (utomata)^r\ a \\
&= (tomata)^r\ ua \\
&= (omata)^r\ tua \\
&= (mata)^r\ otua \\
&= (ata)^r\ motua \\
&= (ta)^r\ amotua \\
&= (a)^r\ tamotua \\
&= (\varepsilon)^r\ atamotua \\
&= \ atamotua
\end{aligned}
$$

Basic concepts(Cont.)

- The set of strings created from any number (0 or 1 or …) of symbols in an alphabet Σ is denoted by Σ^*.
- That is, $\Sigma^* = \cup_{i=\infty_0} \Sigma^i$
  - Let $\Sigma = \{\mathbf{\mathit{0}}, \mathbf{\mathit{1}}\}$.
  - $\Sigma^* = \{\varepsilon$, *0*, *1*, *00*, *01*, *10*, *11*, ***000***, ***001***, ***010***, ***011***, … }.

Basic concepts(Cont.)

Meaning of ∑*

- The set of strings created from any number (0 or 1 or …) of symbols in an alphabet Σ is denoted by Σ*.
- That is, $\Sigma^* = \cup_{i=\infty_0} \Sigma^i$
  - Let Σ = {***0, 1***}.
  - Σ* = {ε, ***0, 1***, ***00, 01, 10, 11***, ***000, 001, 010, 011***, … }.

Basic concepts(Cont.)

Meaning of Σ^+

- The set of strings created from at least one symbol (1 or 2 or …) in an alphabet Σ is denoted by Σ^+.
- That is, $\Sigma^+ = \cup_{i=}{}^{\infty}{}_{1} \Sigma^{i}$

  $= \cup_{i=0..\infty} \Sigma^i - \Sigma^0$

  $= \cup_{i=0..\infty} \Sigma^i - \{\varepsilon\}$
- Let $\Sigma = \{0, 1\}$. $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots\}$.

  Σ^* and Σ^+ are infinite sets.

Basic concepts(Cont.)

Language:

- A language over an alphabet Σ is a set of strings over Σ.
    - Let $\Sigma = \{0, 1\}$ be the alphabet.
    - L_e = $\{\omega \in \Sigma^* \mid$ the number of 1's in ω is even$\}$.
    - ε, $0, 00, 11, 000, 110, 101, 011, 0000, 1100, 1010, 1001, 0110, 0101, 0011, \ldots$ are in L_e

Σ^* and Σ^+ are infinite sets.

Kleene's closure

Let L be a language over an alphabet Σ.

The Kleene's closure of L, denoted by L^*, is $\{x$ | for an integer $n \geq 0$ $x = x_1 x_2 \ldots x_n$ and $x_1, x_2, \ldots, x_n$ are in $L\}$.

That is, $L^* = \cup_{i=0}^{\infty} L^i$

Example: Let $\Sigma = \{0,1\}$ and

$L_e = \{\omega \in \Sigma^* \mid$ the number of 1's in ω is even$\}$

$L_e^* = \{\omega \in \Sigma^* \mid$ the number of 1's in ω is even$\}$

$(\overline{L_e})^* = \{\omega \in \Sigma^* \mid$ the number of 1's in ω is odd$\}^*$

$= \{\omega \in \Sigma^* \mid$ the number of 1's in $\omega > 0\}$

Kleene's closure (cont.)

Let L be a language over an alphabet Σ.

The closure of L, denoted by L^+, is { x |for an integer $n \geq 1$, $x = x_1x_2...x_n$ and $x_1, x_2, ..., x_n$ are in L}

That is, $L^+ = \cup_{i=1}^{\infty} L^i$

Example:

Let $\Sigma = \{0, 1\}$ be the alphabet.

L_e = {$\omega \in \Sigma^*$ | the number of 1's in ω is even}

L_e^+ = {$\omega \in \Sigma^*$ | the number of 1's in ω is even} = L_e^*

Kleene's closure (cont.)

$L^+ = L^* - \{\varepsilon\}$?

Example:

$L = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$

$L^+ = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\} = L_e{}^*$

Why?

$L^* = L^+ \cup \{\varepsilon\}$?

Kleene's closure (cont.)

$L^{+} = L^{*} - \{\varepsilon\}$?

Example:

$L = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$

$L^{+} = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\} = L_e^*$

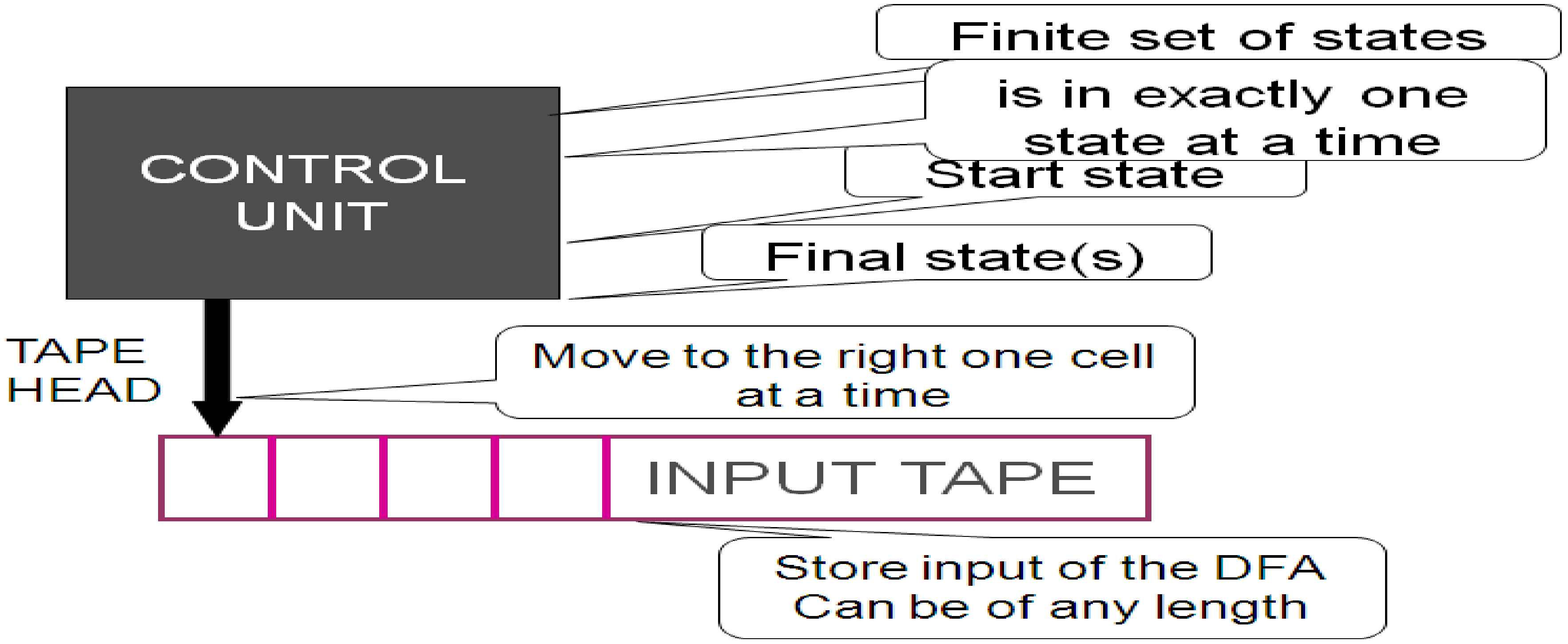
Why?

$L^{*} = L^{+} \cup \{\varepsilon\}$?

Finite Automata (contd.)

- What do we mean by finite?

  An automaton is finite when the number of states are finite. When output is depend on the input as well as states the it is called automaton with finite memory.


Finite set of states
is in exactly one state at a time
Start state
Final state(s)
CONTROL UNIT
TAPE HEAD
Move to the right one cell at a time
INPUT TAPE
Store input of the DFA Can be of any length

Finite Automata (contd.)

What does a Finite automata do?

- Read an input string from tape
- Determine if the input string is in a language
- Determine if the answer for the problem is “YES” or “NO” for the given input on the tape

Finite Automata (contd.)

How does Finite automata works?

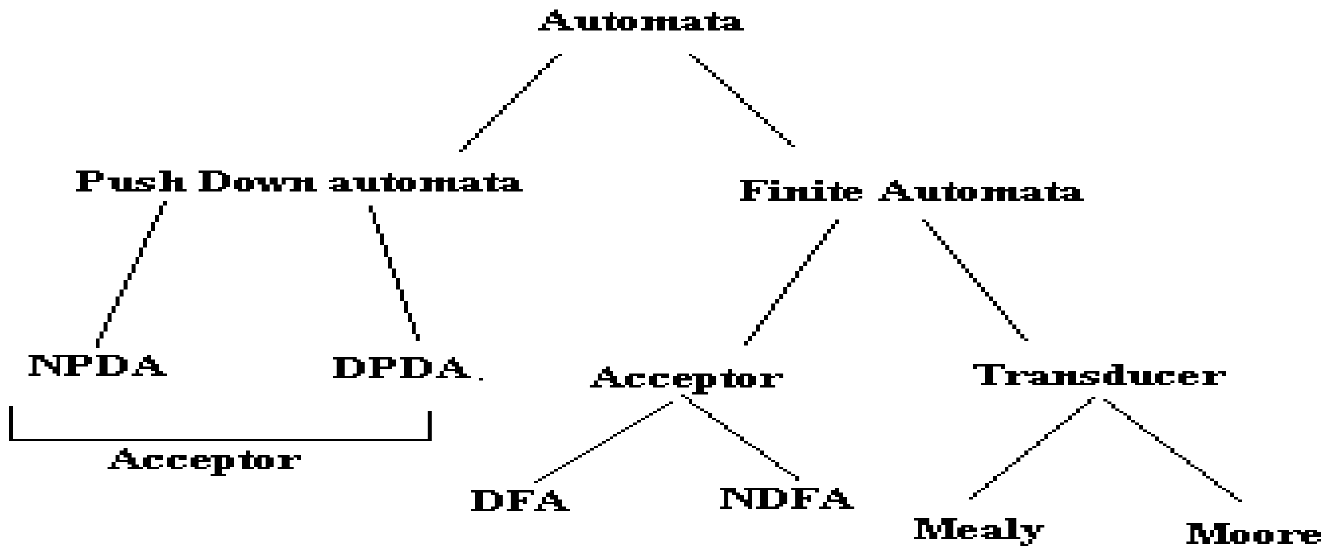
- At the beginning,
  - an FA is in the *start state* (*initial state*)
  - its tape head points at the first cell
- For each move, FA
  - reads the symbol under its tape head
  - changes its state (according to the *transition function*) to the *next state* determined by the symbol read from the tape and its current state
  - move its tape head to the right one cell

Finite Automata (contd.)

How does Finite automata stop?

- When it reads all symbols on the tape
- Then, it gives an answer if the input is in the specific language:
  - Answer "YES" if its last state is a *final state*
  - Answer "NO" if its last state is not a *final state*

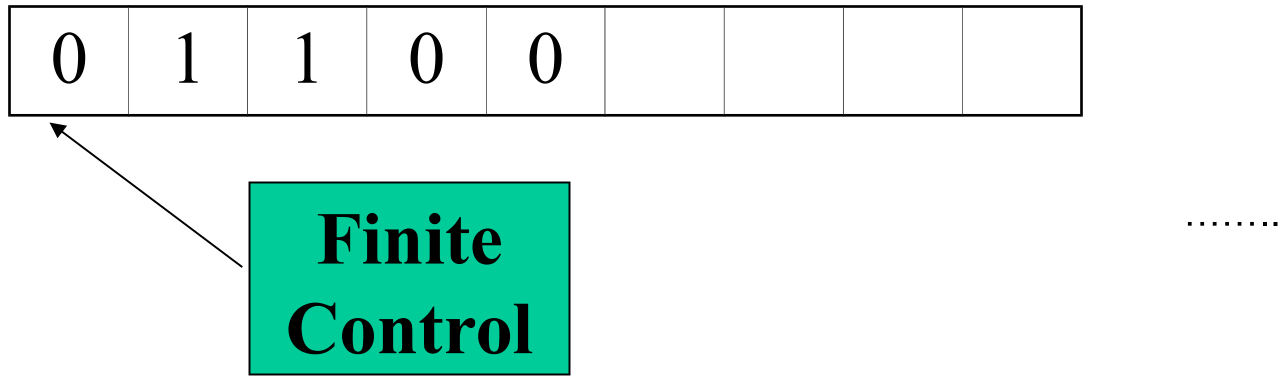
Finite Automata (contd.)



- Acceptor :- Automata whose output response is limited to "Yes" or "No".
- Transducer :- An automata capable of producing strings of symbols as output.

Deterministic Finite Automata

Deterministic Finite State Automata (DFA)



- One-way, infinite tape, divided into cells
- One-way, read-only tape head
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current "state."
- A string is placed on the tape, read head is positioned at the left end, and the DFA will read the string one symbol at a time until all symbols have been read. The DFA will then either accept or reject.

Formal Definition of DFA

A DFA is a five-tuple:

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

Q A finite set of states

Σ A finite set of input alphabets

q_0 The initial/starting state, q_0 is in Q

F A set of final/accepting states, which is a subset of

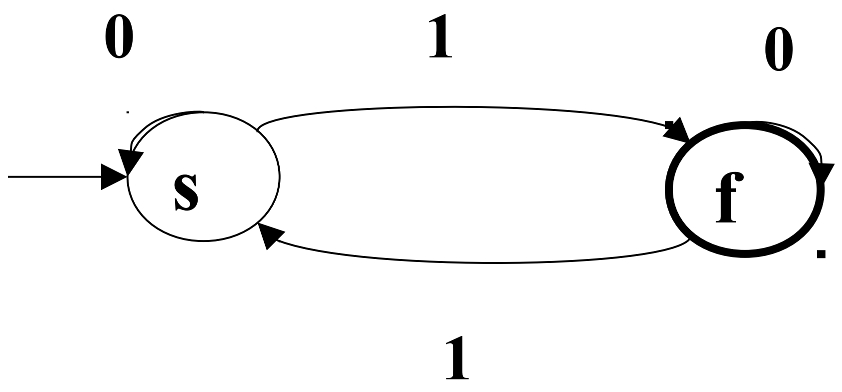
δ A transition function, which is a total function from Q x Σ to Q

$\delta: (Q \times \Sigma) \rightarrow Q$

Example of DFA

State	∑	δ(*q*,*a*)
s	0	s
s	1	f
f	0	f
f	1	s

Transition function


$$\begin{tikzcd} \arrow[r] & s \arrow[loop above, "0"] \arrow[r, bend left, "1"] & f \arrow[loop above, "0"] \arrow[l, bend left, "1"] \end{tikzcd}$$

Transition diagram

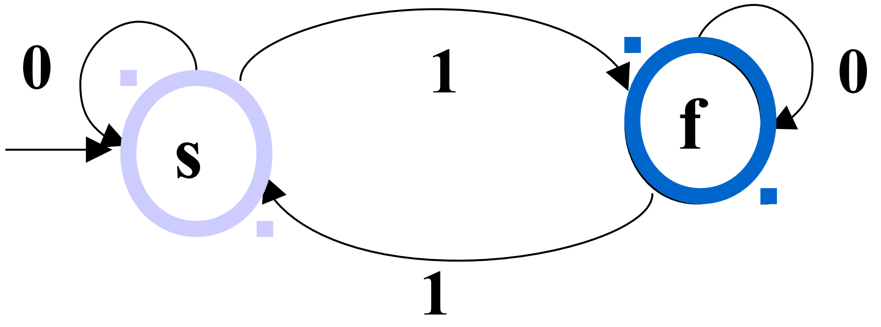
- ∑ *represents the input symbols (Here in this case 0 and 1)*

Acceptance of String in DFA

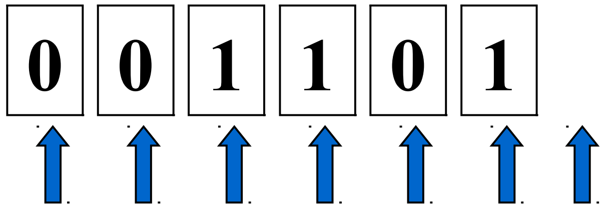
A string *x* is accepted by a finite automata

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

If $\delta(q_0, x) = p$ for some $p \in F$.

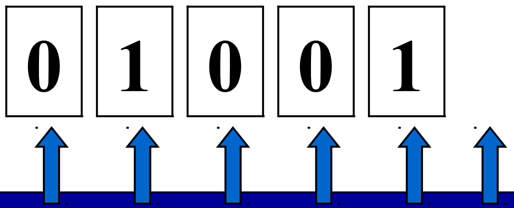


Input: 001101



ACCEPT

Input: 01001



REJECT

Extended Transition Function

Definition :- This is the extension of normal transition function. It receives the state and a string as a parameter and returns a state "p" after processing the sequence of string.

Notation : - $\delta^\wedge$ or δ^*

Formally:

1) $\delta^\wedge(q, \varepsilon) = q$, and

2) For all w in Σ^* and a in Σ

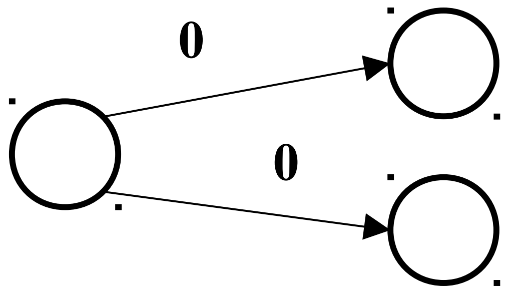
$$\delta^\wedge(q,wa) = \delta\,(\delta^\wedge(q,w), a)$$

Non-deterministic Finite Automata

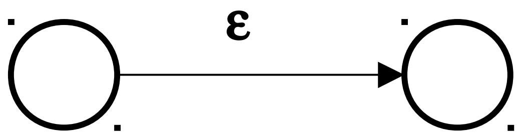
- Why Non-determinism ?
  - One can make a choice at some state, for example in game playing program.
  - Helpful solving problem easily
  - It can serve as a model of search and backtrack algorithm.

Non-deterministic Finite Automata

- Similar to DFA
- Nondeterministic move
  - On reading an input symbol, the automaton can choose to make a transition to one of selected states.



  - Without reading any symbol, the automaton can choose to make a transition to one of selected states or not.



Non-deterministic Finite Automata

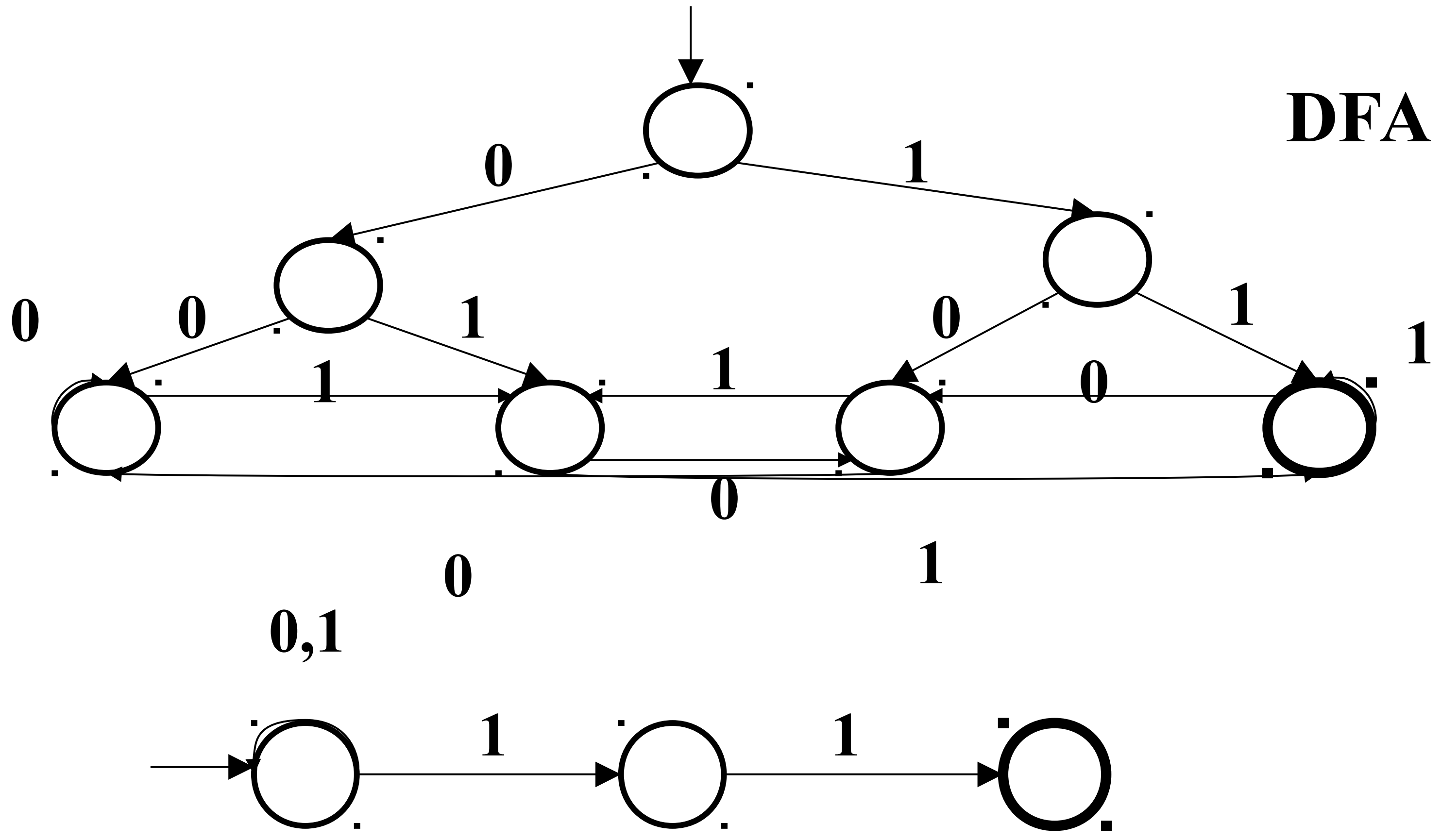
- **Formal Definition**

A NFA is a quintuple $A=(Q,\Sigma,\delta,q_0,F)$ Where

- Q is a set of *states*
- Σ is the alphabet of *input symbols*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final states*
- $\delta: Q \times (\Sigma \cup \lambda) \rightarrow 2^Q$ is the *transition function*

Example of NFA

- An NFA accepting $\{w \in \{0,1\}^* \mid w \text{ ends with } 11\}$



Acceptance of string in NFA

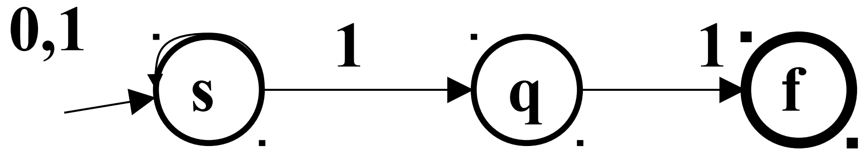
(s,01111) |- (s,1111) |- (s,111) |- (s,11) |- (s,1) |- (s,ε)

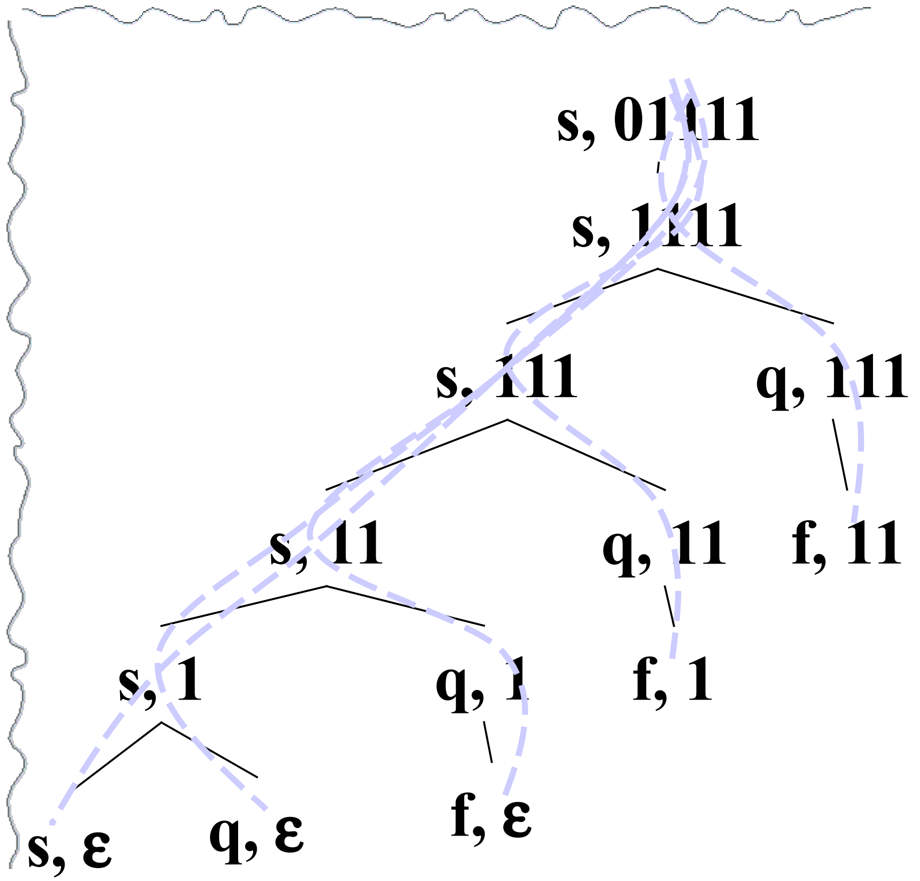
(s,01111) |- (s,1111) |- (s, 111) |- (s,11) |- (s,1) |- (q,ε)

(s,01111) |- (s, 1111) |- (s, 111) |- (s,11) |- (q,1) |- (f, ε)

(s,01111) |- (s, 1111) |- (s, 111) |- (q,11) |- (f,1)

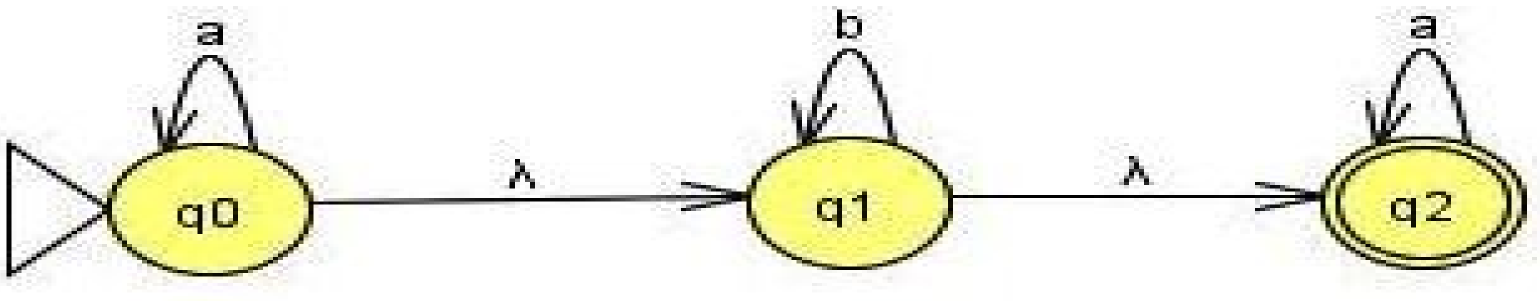
(s,01111) |- (s, 1111) |- (q, 111) |- (f,11)





λ Closure in NFA

The **λ -closure (q)** is a function which gives the set of all the states of the automata which can be reached from q without consuming any input symbol.



λ -closure (q_0) $=\{q_0,q_1,q_2\}$

λ -closure (q_1) $=\{q_1,q_2\}$

λ -closure (q_2) $=\{q_2\}$

Removing λ transition from NFA

- Step 1 : Find the Initial states of NFA
  - Initial state will be the **λ -closure** of initial state of NFA
  - In previous example λ -closure (q_0) ={ q_0,q_1,q_2} will be the new initial states.
- Step 2: Find the λ –closure of other states.
  - In previous example

$$\lambda\text{-closure }(q_1) = \{q_1, q_2\}$$

$$\lambda\text{-closure }(q_2) = \{q_2\}$$

Removing λ transition from NFA

- Step 3 : Find The final States
  - Final states will be all those new states which contains final state of NFA
  - In Previous example , { q_0,q_1,**q_2**} , {q_1,**q_2**}, {**q_2**} are the final states as they contain **q_2**
- Step 4 : Now Decide the transition of newly created states.
  - Find the transition by following formula

$$\delta(\{q_0,q_1,q_2\}\ ,\ a\ ) = \lambda\text{ -closure}(\delta(q_0,q_1,q_2)\ ,\ a\ )$$
$$= \lambda\text{ -closure}(\delta(q_0,a)\ U\ \delta(q_1,a)\ U\ \delta(q_2\ ,a))$$

Conversion of NFA to DFA

- Theorem : For every NFA, there exists a DFA which simulates the behavior of NFA.

- Conversion Process
  - Step 1: Start with the initial state and find the transition of all input alphabets.
  - Step 2: Find the transitions for all new states appearing under the input columns.
  - Step 3 : Repeat step 2 until we get any new states.

Minimization of DFA

- Step 1: Remove all unreachable states (The states which can never be reached from initial state)
- Step 2 : Divide all vertices into two sets i.e. One set of final states and second one for non-final states.
- Step 3: For each state in both the sets , find the transition of an alphabet (only one alphabet) .
  - If the transition state is the element of another set or class then separate these states from that set and make another class.
- Step 4: Repeat the Step 3 until all the classes are made.
- Step 5: Repeat Step 3 and 4 for all alphabets.
- Step 6 : After completion of all steps, draw the DFA .

Automata With Output

- This type of automata is also called as transducers.
- It is useful to design a machine which performs calculations and conveying results.
- There are two approaches :
  - Mealy Machine
    - ✓Invented by G.H. Mealy in 1955
  - Moore Machine
    - ✓ Invented by E.F. Moore in 1956

Mealy Machine

- In this machine, output is associated with the transition.
- Every transition for a given input has an output.
- For every string of length n, n length output string is generated
- Format Definition :

  Mealy machine is a six-tuple machine and defined as

  M= (Q, $\sum$, Δ, δ, λ, q_0) where

a) Q is a finite set of states

b) $\sum$ is the set of input symbols.

c) Δ is the set of output alphabet.

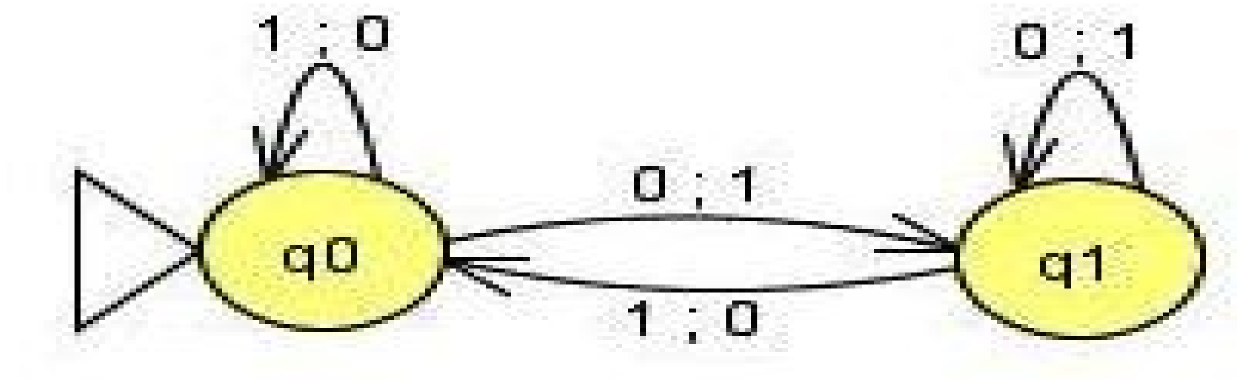
d) δ is the transition function $\sum$ x Q -> Q

e) λ is the output function mapping $\sum$ x Q -> Δ

f) q_0 is the initial state.

Representation of Mealy Machine

- By Transition Diagram



- By Transition table

Present State	For input a= 0		For input a=1	
	State	Output	State	Output
-> q_0	q_1	1	q0	0
q_1	q_1	1	q0	0

Moore Machine

- In this machine, output is associated with each state.
- Every state for a given input has an output.
- For every string of length n, n+1 output string is generated
- Format Definition :

  Mealy machine is a six-tuple machine and defined as

  $M = (Q, \sum, \Delta, \delta, \lambda, q_0)$ where

a) Q is a finite set of states

b) $\sum$ is the set of input symbols.

c) Δ is the set of output alphabet.

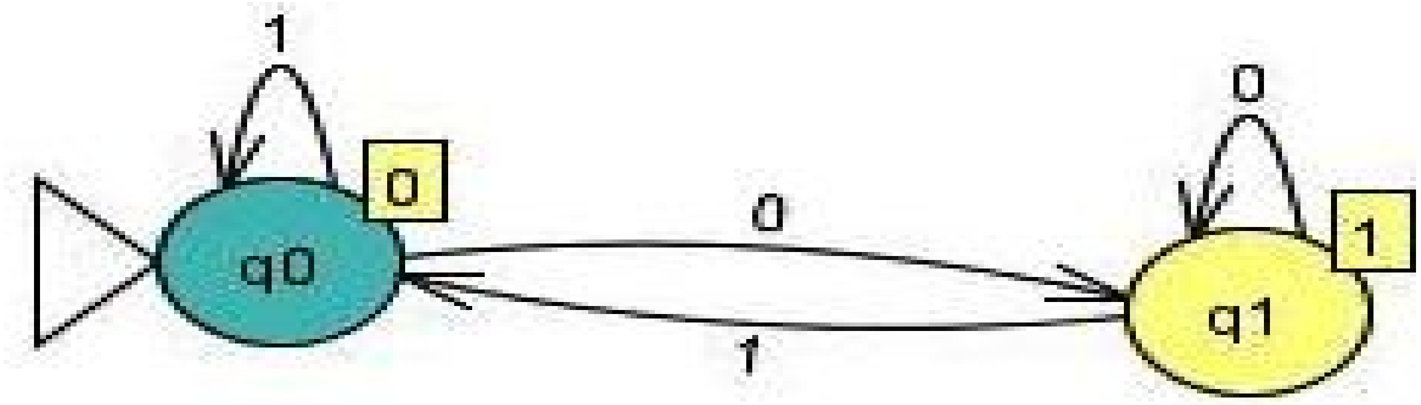
d) δ is the transition function $\sum \times Q \to Q$

e) λ is the output function mapping $Q \to \Delta$

f) q_0 is the initial state.

Representation of Moore Machine

- By Transition Diagram



- By Transition table

Present State	Next State δ		Output
	a=0	a=1	λ
-> q_0	q_1	q0	0
q_1	q_1	q0	1

Difference between Mealy and Moore Machine

Moore Machine	Mealy Machine
It's output depends only on present state	Its output depends on the input and current state
Its transition function maps $Q \to \Delta$	Its transition function maps $\sum \times Q \to \Delta$
Produces output of length n+1 for every input string of length n	Produces output of length n+1 for every input string of length n
Its output is same for different inputs	Different input on same state may have the different output

Conversion from Moore to Mealy machine

Procedure:

- Step 1 : Write down the next state for the present state from the more machine for every input.
- Step 2 : For each state q , find λ^1 (q) from Moore machine. and write in output section.
- Step 3 : If two rows are identical in the transition table of Mealy machine then we can delete any one of the row.

Conversion from Moore to Mealy machine

Example: Consider following Moore Machine

Present State	Next States		Output
	A=0	A=1	
q0	q3	q1	0
q1	q1	q2	1
q2	q2	q3	0
q3	q3	q0	0

Equivalent Mealy Machine

Present State	A=0		A=1	
	Next State	Output	Next State	Output
q0	q3	0	q1	1
q1	q1	1	q2	0
q2	q2	0	q3	0
q3	q3	0	q0	0

Conversion from Mealy to Moore machine

Procedure:

- Step 1 : For each state q_i determine the output .
- Step 2 : For any state q_i having two different outputs , split the state q_i into two different states and assign output to them.
    - ✓ $q_i \Rightarrow q_i^j$ and q_i^{j+1} ……. q_i^n
- Step 3 : Remake the transition table of given Mealy machine according to new states .
- Step 4 : From the newly created table find the next state and output and put it into Moore machine transition diagram.

Conversion from Mealy to Moore machine

Example: Consider following Mealy Machine

Present State	A=0		A=1	
	Next State	Output	Next State	Output
q1	q3	0	q2	0
q2	q1	1	q4	0
q3	q2	1	q1	1
q4	q4	1	q3	0

Step 1: q2=0,1 q3=0 q1=1 q4=0,1

Step 2: q20=0 q21=1 q40=0 q41=1

$$\begin{tikzcd}
 & q2=0,1 \arrow[dl] \arrow[dr] & & & q4=0,1 \arrow[dl] \arrow[dr] & \\
q20=0 & & q21=1 & q40=0 & & q41=1
\end{tikzcd}$$

Conversion from Mealy to Moore machine

Step 3.

Present State	A=0		A=1	
	Next State	Output	Next State	Output
q1	q3	0	q20	0
q20	q1	1	q40	0
q21	q1	1	q40	0
q3	q21	1	q1	1
q40	q41	1	q3	0
q41	q41	1	q3	0

Step 4:

q1=1 q20=0 q21=1 q3=0 q40=0 q41=1

Conversion from Mealy to Moore machine

Step 4:

Present State	Next State		Output
	a=0	a=1	
->q1	q3	q20	1
q20	q1	q40	0
q21	q1	q40	1
q3	q21	q1	0
q40	q41	q3	0
q41	q41	q3	1

Note : *Here the starting state's output 1. It means it will not accept the λ but in our given Mealy machine accepts λ. So we add a new state say q0 whose output is 0 and make the q0 as a initial state.*

Conversion from Mealy to Moore machine

So final Moore machine looks like

Present State	Next State		Output
	a=0	a=1	
->q0	q3	q20	0
q1	q3	q20	1
q20	q1	q40	0
q21	q1	q40	1
q3	q21	q1	0
q40	q41	q3	0
q41	q41	q3	1

Structural Representation

- There are two important notation that are not automation-like but plays the important role in the study of automata.
  - **Regular Expr**ession: It can be thought of as a algebraic description of DFA and NDFA. It is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching.
  - **Grammar** : Finite automata can be written as structural rules that governs the composition of words in any given language. This is the useful model to when designing software that process data. The best example is "parser".

Regular Expression

- We can recursively define a regular expression as following
  - Any terminal (symbol in ∑), λ, ф are regular expression
  - If "**a**" belongs to ∑ then it is also a regular expression
  - The union of two regular expression R1 and R2 is a regular expression and is written as R1+R2
  - The concatenation of two regular expression R1 and R2 is a regular language and is written as R1.R2
  - The iteration of a regular expression R is a regular expression and is written as R*.

Regular Expression

- **Regular Set:** Any set represented by a regular expression is called a regular set.
- The set represented by R is denoted by *L*(**R**).
- Some Regular expression and their Regular sets.

Regular Exp	Regular set
a	{a}
a+b+c	{a,b,c}
ab+ba	{ab,ba}
ab(d+e)	{abd,abe}
a*	{λ,a,aa,aaa……}
abc*de	(abde,abcde,abccde,…}
(a+b)*	{λ,a,b,aa,ab,ba,bb,aaa,aab,bba……..}

Identities of Regular Expression

$\phi + R = R$
$\phi R = R\phi = \phi$
$\lambda R = R\lambda = R$
$\lambda^* = \lambda$ and $\phi^* = \lambda$
$R + R = R$
$R^*R^* = R^*$
$RR^* = R^*R$
$(R^*)^* = R^*$
$\lambda + RR^* = R^* = \lambda + R^*R$
$(PQ)^*P = P(QP)^*$
$(P + Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$
$(P + Q)R = PR + QR$
$R(P+Q) = RP + RQ$

Regular Expression and Regular Sets

- Discuss the following regular sets by regular expression
  - L= {0,1,2}
  - R= 0+1+2
  - L= $\{x^{2n+1};\ n>0\}$

    R= 1(11)*
  - L= set of all string starts with 1 and ends with 00.

    R= 1(1+0)*00
  - L= Set of all strings with three consecutive b.

    R= (a+b)*bbb(a+b)*
  - L= set of all string having at most two a

    R=b* + b*ab* +b*ab*ab*
  - L= Set of all string with even no of a's followed by an odd no of b's   R= (aa)*(bb)*b

Arden's Theorem

Theorem : *Let P and Q be two regular expressions over ∑. If P does not contain λ then the following equation*

$$R = Q + RP \;\ldots\ldots (A)$$

can be written as $R = QP^*$

Proof:-

To prove this theorem we put the value of ***R*** in equation A. Hence , we get $Q + RP = Q+(Q+RP)P$

$$= Q+QP+RPP$$

$$= Q+QP+RP^2$$

$$= Q+QP+QP^2+\ldots.QP^i+RP^{i+1}$$

$$= Q(\lambda + P + P2 + \ldots + P^i)+RP^i$$

Arden’s Theorem

$$Q+RP = Q(\lambda +P +P2+\ldots+P^i)+RP^i$$

for all i>=0 …..(B)

Let us assume *w* be the string of length in the set R. then *w* belongs to the equation B. But as P does not contain λ , hence RP^{i+1} does not contain any string of length less than i+1 and so *w* does not belongs to the set RP^{i+1}. This means *w* belongs to the QP*.

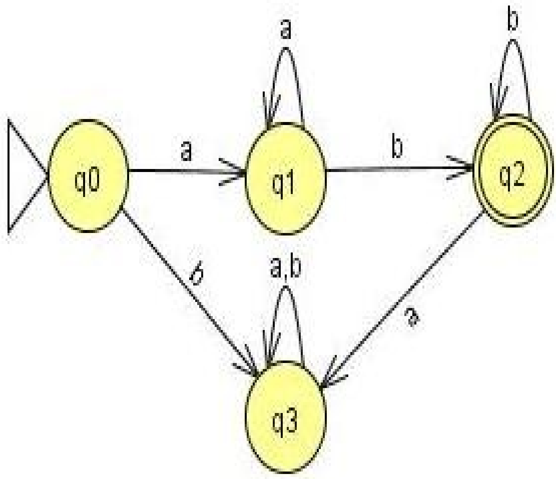
Finite Automata to Regular Expression

- We can use Arden's theorem to find the regular expression from a given finite automata.
- Assumptions;
  - Finite automata has only one initial state.
  - No λ move
- From this graph we can write equations as

  Consider the incoming arrows for a node

  - q0= λ (No incoming arrow on q0)
  - q1=q0.a + q1.a (from q0,a and q1,b)
  - q2=q1.b+q2.b (from q1,b and q2,b)
  - q3 = q0.b + q3.a +q3.b +q2.a

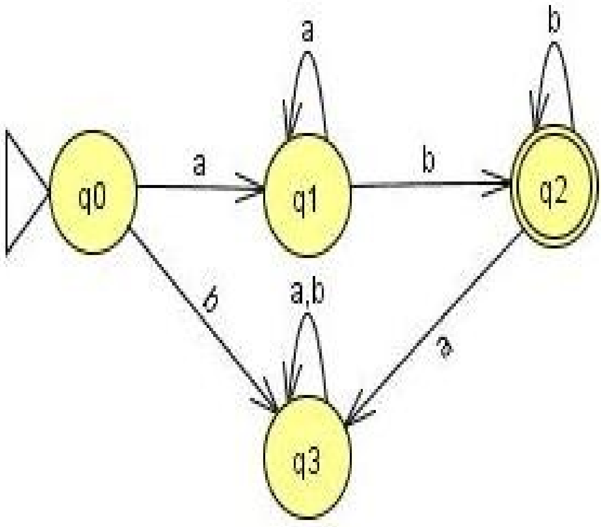
  (from q0,b; from q3,b; from q3,a; from q2,a)



Finite Automata to Regular Expression

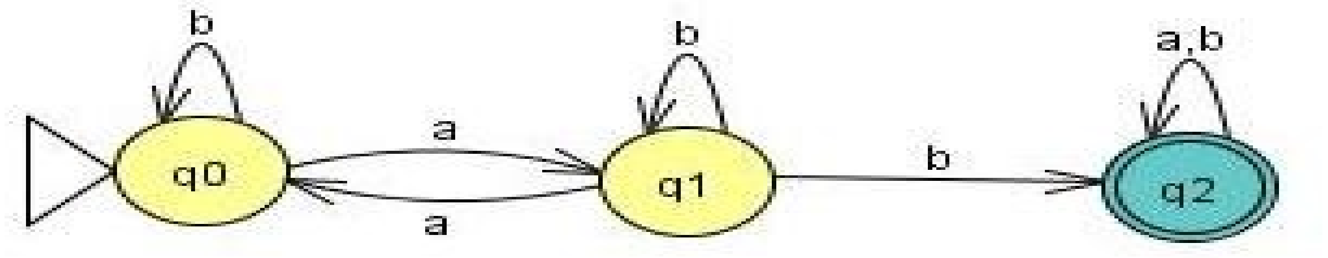
- q0= λ
- q1=q0.a + q1.a
- q2=q1.b+q2.b
- q3 = q0.b + q3.a +q3.b +q2.a
- q1= λa+q1a = a+q1a=Q+RP=>R=QP*
- q1= aa*
- q2=aa*b+q2b = Q+RP => R=QP*
- q2=aa*bb*

Now q2 is a final state therefore our conversion ends here.

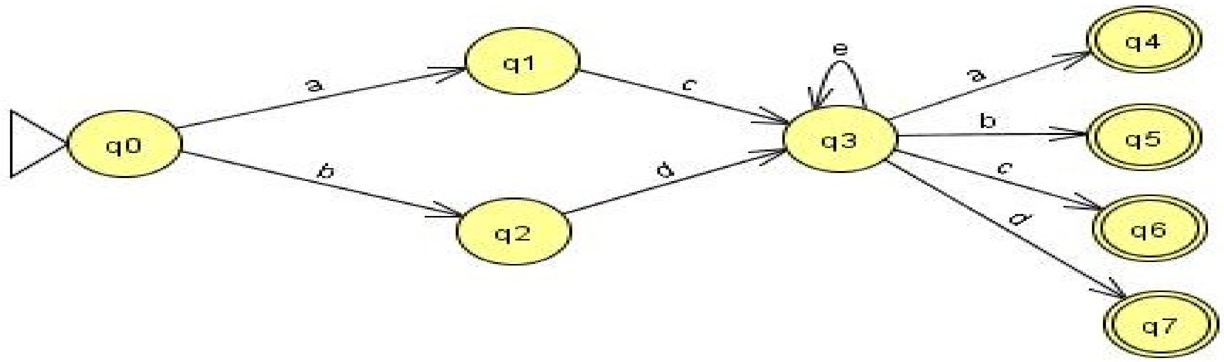


Finite Automata to Regular Expression

- Now Try Some More Finite Automata…



Answer => (b+ab*)*ab*b(a+b)*



Answer => (ac+bd)e*(a+b+c+d)

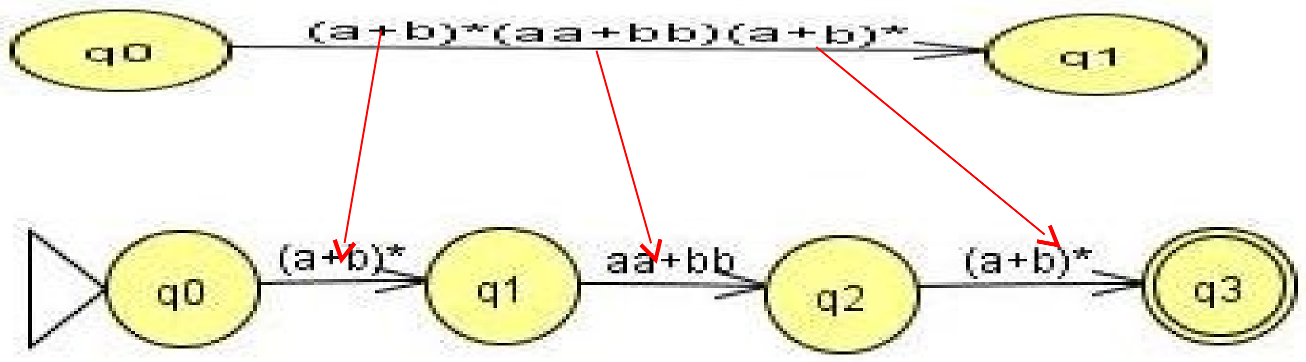
Construction of FA from Regular Expression

- Remove the concatenation("." symbol) by adding a new state. If required use λ move.
- For every "+" symbol use the parallel arrow.

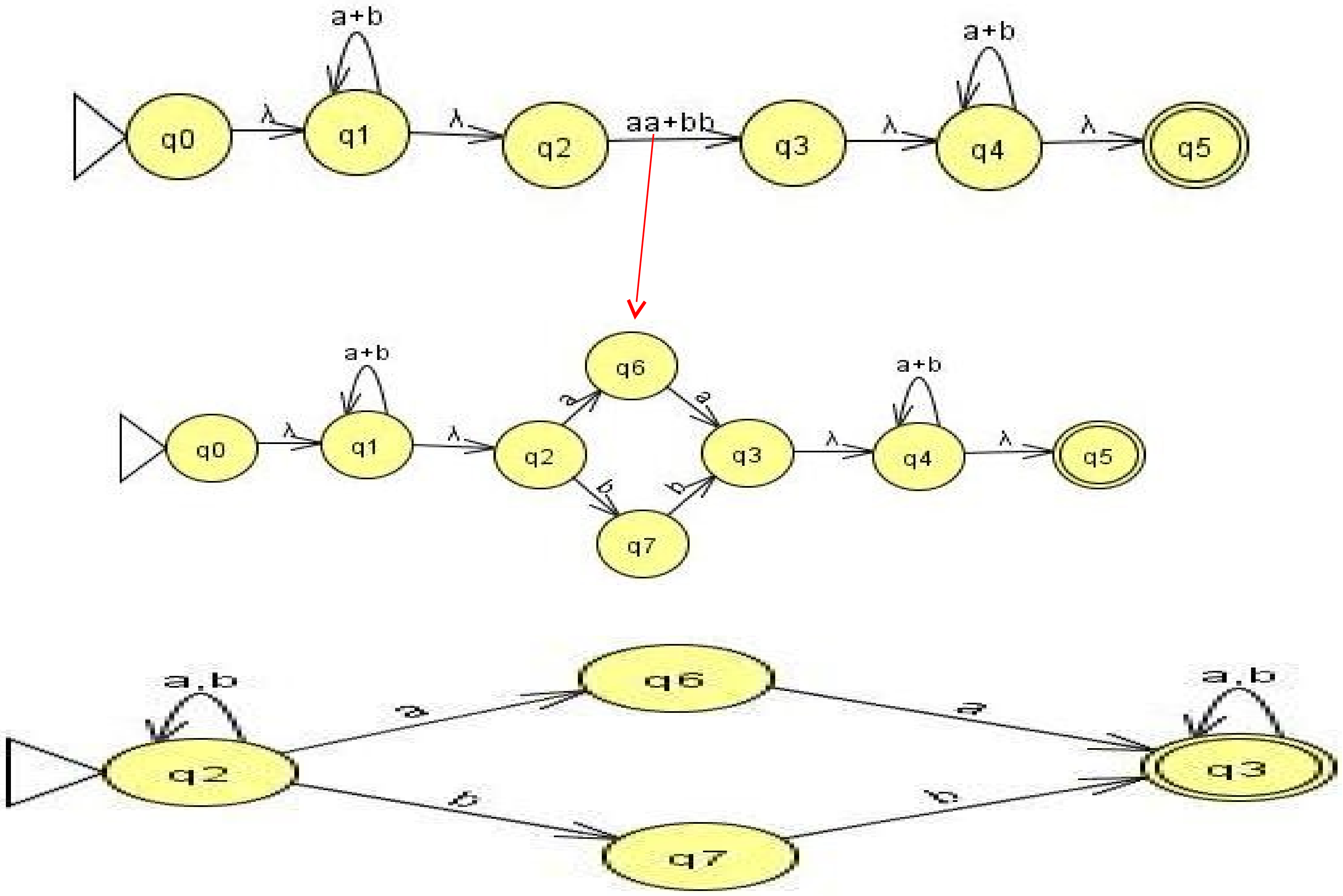
Let us take an example……

R=(a+b)*(aa+bb)(a+b)*

1) First remove concatenation step by step



Construction of FA from Regular Expression



Kleene's Theorem Statement

Any language that can be defined by:

- Regular Expression (RE), or
- Finite Automata (FA), or
- Transition Graph (TG)

Can be defined by all three methods

Kleene's Theorem Proof

The three sections of the proof will be:

Part 1: Every language that can be defined by FA can also be defined by TG

Part 2: Every language that can be defined by TG can also be defined by RE

Part 3: Every language that can be defined by RE can also be defined by FA

Kleene’s Theorem Proof

- **Proof of Part 1**
  - This is the easier part. Every FA is itself already TG. Therefore, any language that has been defined by FA has already been defined by TG. We have shown in previous slides

- **Proof of Part 2**
  - We will give a constructive algorithm for proving part 2.
  - we have described an algorithm to take any transition graph T and form a regular expression corresponding to it.
  - The algorithm will work for any transition graph T
  - The algorithm will finish in finite time

Kleene's Theorem Proof

- **Proof of Part 3**
  - Every language that can be defined by a regular expression can also be defined by a FA
  - We will do this by using a recursive definition and a constructive algorithm.
  - We have discussed the recursive definition of regular expression.
  - Based on the above recursive definition for regular expressions, we have the following recursive definition for FA's associated with regular expressions

Kleene's Theorem Proof

- Recall that we had the following recursive definition for regular expressions:

- **Rule 1:** If $x \in \Sigma$, then x is a regular expression. Λ is a regular expression. □ is a regular expression.
- **Rule 2:** If r_1 and r_2 are regular expressions, then $r_1 + r_2$ is a regular expression.
- **Rule 3:** If r_1 and r_2 are regular expressions, then r_1r_2 is a regular expression.
- **Rule 4:** If r_1 is a regular expression, then $r_1{}^*$ is a regular expression

Kleene's Theorem Proof

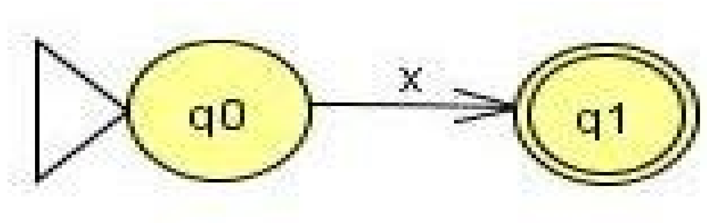
➢ Recall that we had the following recursive definition for regular expressions:

- **Rule 1:** If $x \in \Sigma$, then x is a regular expression. Λ is a regular expression. □ is a regular expression.
- **Rule 2:** If r_1 and r_2 are regular expressions, then $r_1 + r_2$ is a regular expression.
- **Rule 3:** If r_1 and r_2 are regular expressions, then r_1r_2 is a regular expression.
- **Rule 4:** If r_1 is a regular expression, then r_1* is a regular expression

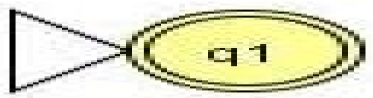
Kleene's Theorem Proof

Based on Rule 1, we get following definition of FA:

- There is an FA that accepts the language L defined by the regular expression x ; i.e., L = { x } , where x ∈ Σ, so language L consists of only a single word and that word is the single letter x

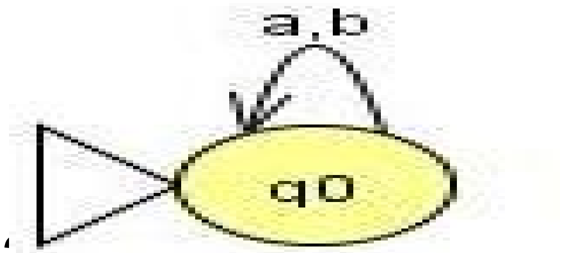


- There is an FA that accepts the language defined by regular expression Λ; i.e., the language {Λ} .



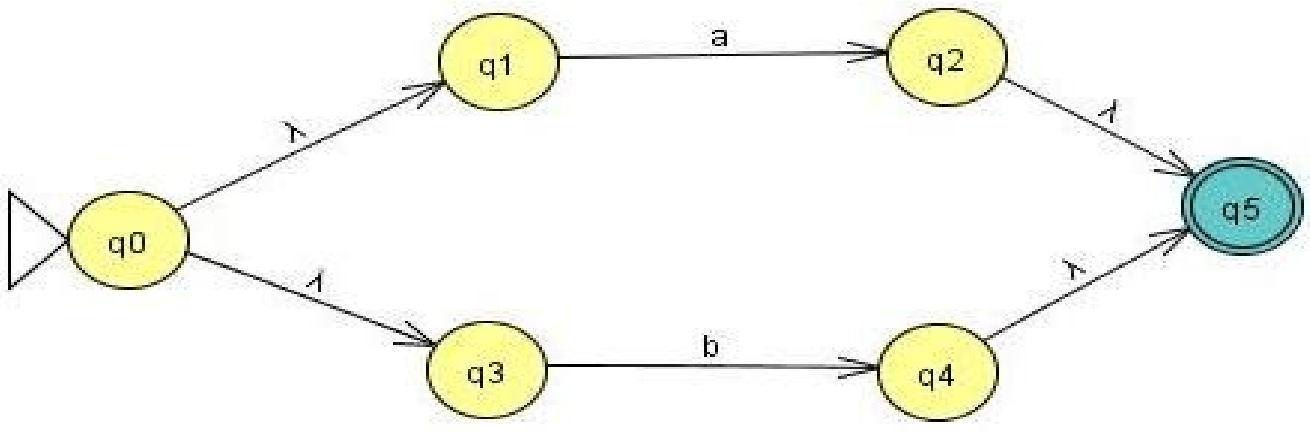
Kleene's Theorem Proof

- There is an FA defined by the regular expression ∅ ; i.e., the language with no words, which is ∅. [Let ∑ ={a,b}].



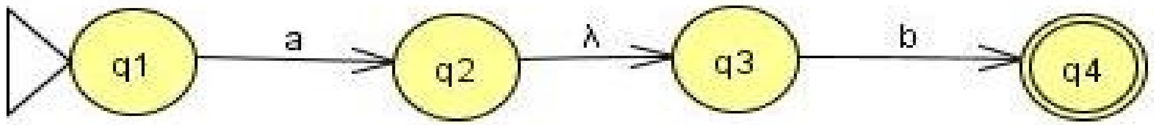
- **Based on Rule 2, we get following definition of FA:**
  - If there is an FA called FA_1 that accepts the language defined by the regular expression r_1 and there is an FA called FA_2 that accepts the language defined by the regular expression r_2 , then there is an FA called FA_3 that accepts the language defined by the regular expression r_1+r_2.

Kleene's Theorem Proof

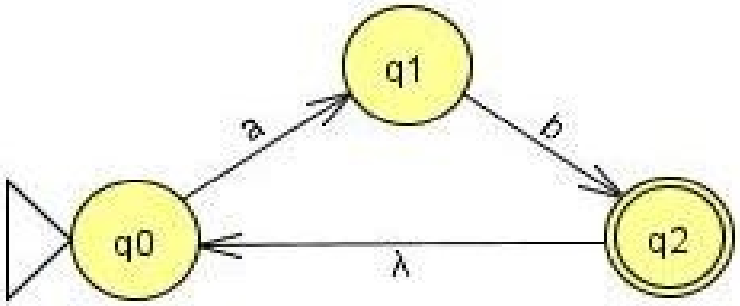


- **Based on Rule 3, we get following definition of FA:**
  - If there is an FA called FA$_1$ that accepts the language defined by the regular expression r1and there is an FA called FA$_2$ that accepts the language defined by the regular expression r_2, then there is an FA called FA$_3$ that accepts the language defined by the regular expression r_1r_2 , which is the concatenation.

Kleene’s Theorem Proof



- **Based on Rule 4, we get following definition of FA:**
  - If there is an FA called FA_1 that accepts the language defined by the regular expression r_1, then there is an FA called FA_2 that accepts the language defined by the regular expression r_1*.



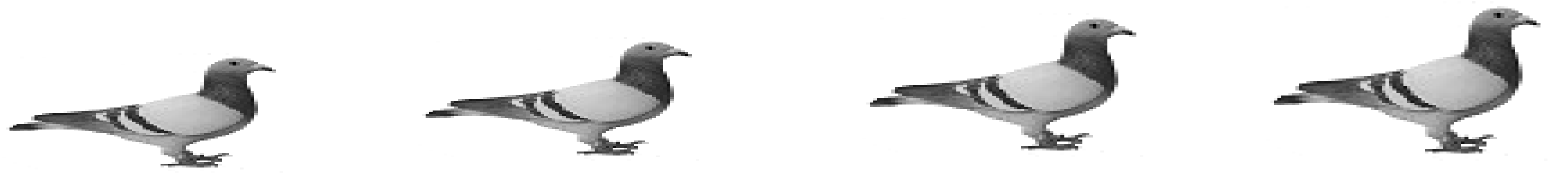
Pumping Lemma

- Mr. X : How can one prove that the given language is not regular?
- Mr. Y : Prove that there is no DFA or NFA for the given language.
- Mr. X : It is very difficult to prove that there is no DFA or NFA because of the infinite no. of DFA and NFA .So what is the solution???????
- Mr. Y: The Pumping Lemma…..
- Mr. X : Please Tell me about the Pumping Lemma..
- Mr. Y: Before discussing pumping lemma. Let me discuss the pigeon hole principle because it is used in Pumping Lemma….

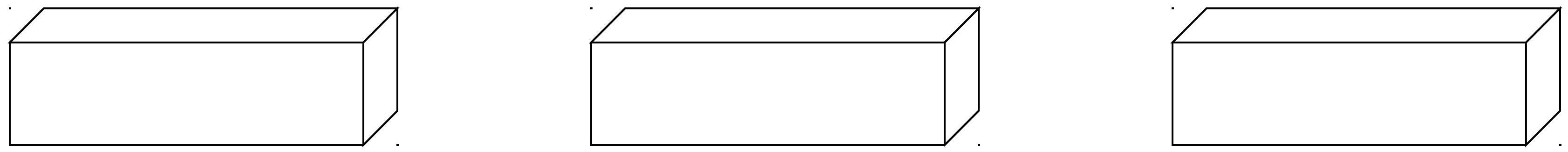
Pigeonhole Principle

- Mr. Y: let us see . Let us play an interesting game. Suppose we have 4 pigeons and 3 pigeonholes . Then how can you adjust 4 pigeons in three pigeonholes such that one pigeonhole contains at least one pigeon?

4 pigeons

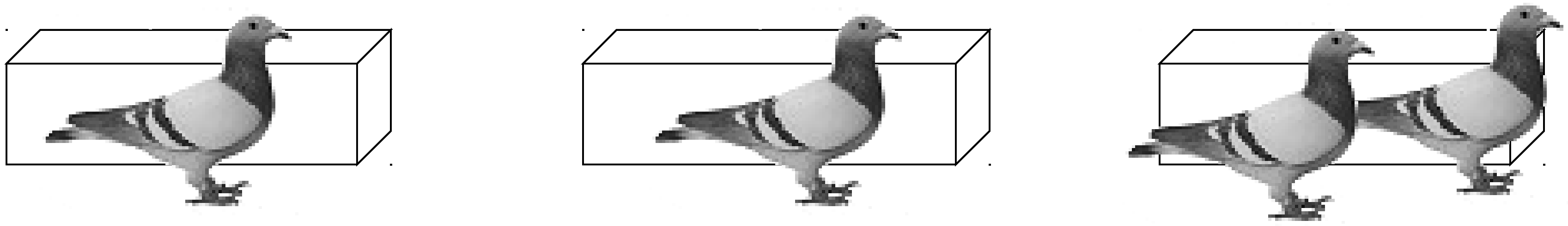


3 pigeonholes



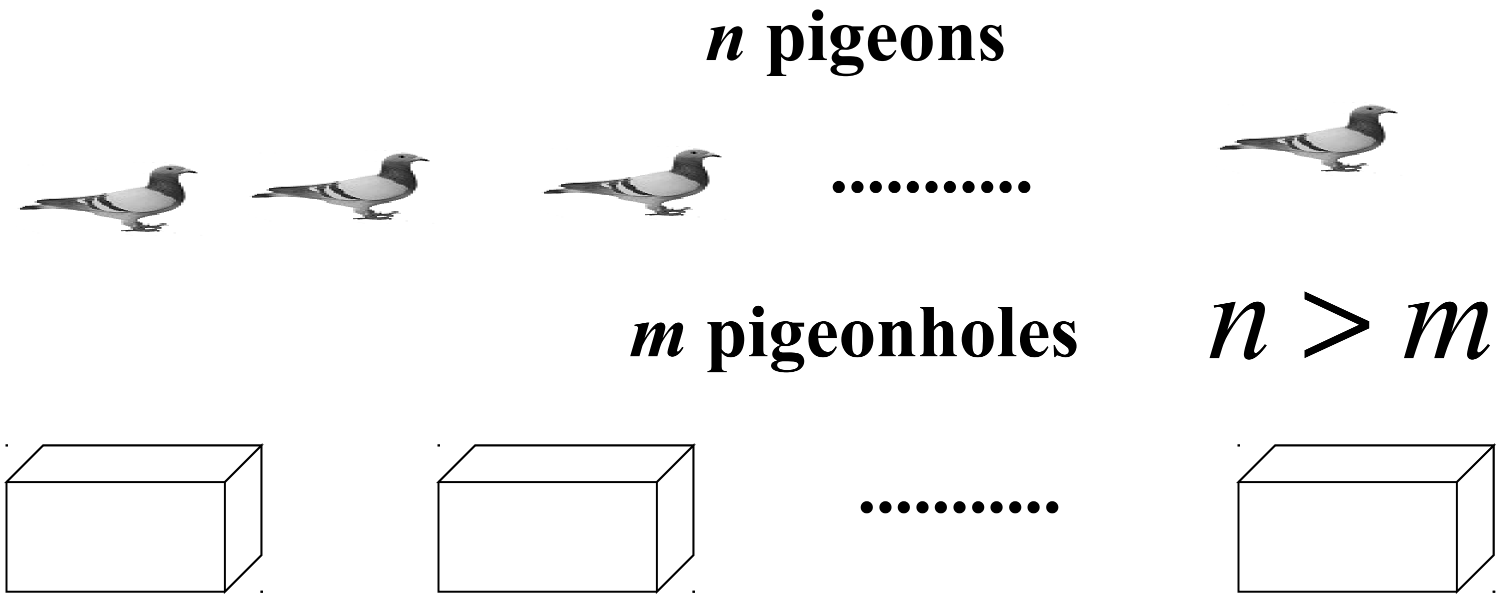
Pigeonhole Principle

- Mr. X: It is very simple . I will have to adjust two pigeons in one pigeonhole.
- Mr. Y: Very good. You are genius. Means that at least one pigeonhole contains two pigeons.
- Mr. X: This what exactly I want to say …



Pigeonhole Principle

- Mr. Y : Formally we can say that if we have ***n*** number of pigeons and ***m*** number of pigeonholes such that ***n>m*** then there is a pigeonhole which contains at least 2 Pigeon.



Pigeonhole Principle and DFA

- Mr. X : What is the relation between pigeonhole principle and DFA?
- Mr. Y : Yes. There is a relation between pigeonhole principle and DFA. Consider a DFA with 4 states.

$$\begin{tikzcd}
q_1 \arrow[r, "a"] & q_2 \arrow[r, "a"] & q_3 \arrow[r, "a"] & q_4 \arrow[r, "a"] & q_f
\end{tikzcd}$$

- Suppose this DFA can accept any no of ***a***. If we take an input of the length 5 i.e. ***"aaaaa"*** then according to pigeonhole principle at least one state accepts two ***a*** or the state is repeated. *Consider inputs as pigeons and states as pigeonholes.*

Pumping Lemma

- Mr. Y (cont.) : Means that if we have n number of states and input string w, and |w| >= n then at least one state is repeated or two states are common.
- Mr. X: Yes absolutely right.
- MR. Y : Now Let me define Pumping Lemma
  - Let M = $(Q,\sum,\delta,q_0,F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let w ε L and |w| >= m. If m>=n then there exists x, y, z such that
    1. w=xyz
    2. $y \neq \lambda$ i.e. |y| >0 and |xy|<= n
    3. xy^iz ε L for each i >=0

Pumping Lemma Proof

- Mr. Y (cont.) : Means that if we have n number of states and input string w, and |w| >= n then at least one state is repeated or two states are common.
- Mr. X: Yes absolutely right.
- MR. Y : Now Let me define Pumping Lemma
  - Let M = $(Q,\sum,\delta,q_0,F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let w ε L and |w| >= m. If m>=n then there exists x, y, z such that
    1. w=xyz
    2. $y \neq \lambda$ i.e. |y| >0 and |xy|<=m
    3. xy^iz ε L for each i >=0

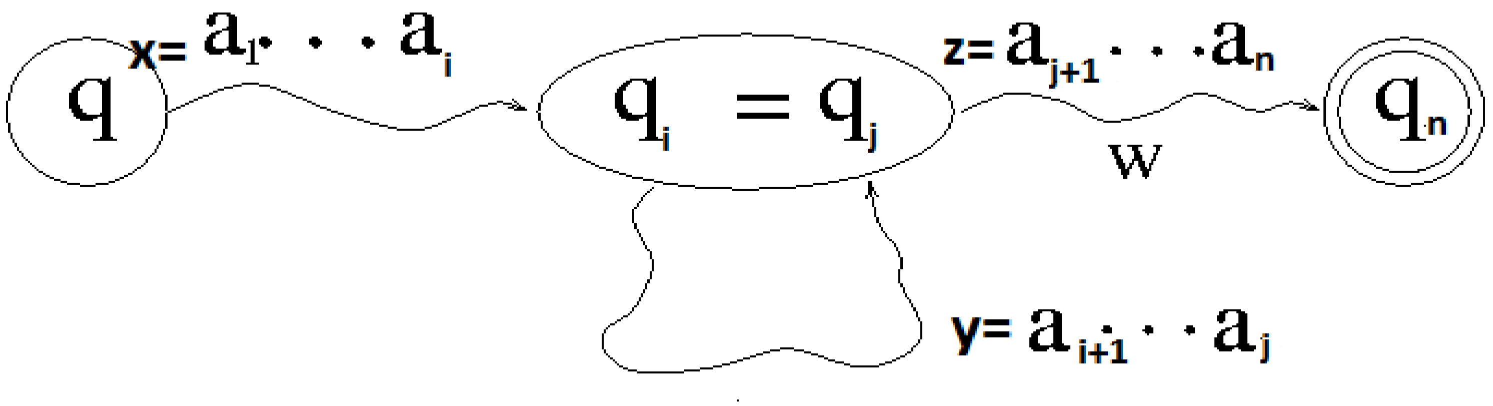
Pumping Lemma Proof

Proof:

- Suppose L is regular then there exists a DFA for L.
- Now consider a string "w" such that |w| >n (n is the no. of states in DFA.
- By Pigeonhole Principle at least one state is repeated or two states say q_i and q_j are similar with $0 <= i < j <= n$.
- Now we can break "w" into xyz as:
  - ✓ $x= a_1,a_2,\ldots\ldots\ldots, a_i$
  - ✓ $y= a_{i+1}, a_{i+2}, \ldots\ldots\ldots, a_j$
  - ✓ $z= a_{j+1}, a_{j+2},\ldots\ldots\ldots\ldots, a_n$

This can be represented by following DFA……

Pumping Lemma Proof


q
$x = a_1 \cdot\cdot\cdot a_i$
$q_i = q_j$
$z = a_{j+1} \cdot\cdot\cdot a_n$
w
q_n
$y = a_{i+1} \cdot\cdot\cdot a_j$

Note 1. X can be empty when i=0

2. Z may be empty when j=n

3. But Y cant be empty.

Application of Pumping Lemma

- Select any language L Which you have to prove non-regular.
- Pick n any integer
- Select a string "w" in L such that $|w| >= n$
- Break "w" into xyz such that $|xy| <= n$ and $|y| >=1$
- Chose the value of $i >=0$ such that xy^iz is not in L.
- Hence we find the contradiction and can prove the language L is not regular.

Application of Pumping Lemma

- Example : L={$a^n b^n$ | n>0} is not regular?
- Proof:

Let us assume L is regular. Let us choose n=2. hence string w= aabb belongs to the language.

Let x=a, y=a and z=bb. Now according to pumping lemma xy^iz does belong to the language. But if we choose i=2 then xy^2z = xyyz =aaabb should also belong to L . We see that aaabb does not belong to language. Hence we find the contradiction. Therefore language L is not a Regular Language

Myhill –Nerode Theorem

Statement: Myhill – Nerode theorem says following three statements are equivalent.

1. **The set $L \subseteq \Sigma^*$ is accepted by a DFA**
2. **The set L is the union of some of *equivalence classes* of a *right invariant* equivalence relation with *finite index.***
3. **Let equivalence relation R_L be defined as $x R_L y$ iff $xz \in L \Leftrightarrow yz \in L$ then R_L has finite index.**

Proof :

We shall prove it by proving

a. Statement 1=> Statement 2

b. Statement 2 => Statement 3

c. Statement 3 => Statement 1

Myhill –Nerode Theorem

1=>2

Let M be a DFA that accepts L.

Let RM is an equivalent relation such that

$$xR_My \Rightarrow \delta^*(q_0,x) = \delta^*(q_0,y) \text{ ....(A)}$$

Let z ϵ Σ*

$$\begin{aligned}\delta^*(q_0,xz) &= \delta^*(\delta(q_0,x),z)\\ &= \delta^*(\delta(q_0,y),z) \text{ .... From (A)}\\ &= \delta^*(q_0,yz)\end{aligned}$$

This means that xzR_Myz

Hence R_M is *right invariant*.

Since DFA has the finite no. of states hence R_M has *finite index*.

Let us assume in machine M .

$F=\{ q_{f1}, q_{f2}\}$

Myhill –Nerode Theorem

If L is accepted by M then for any string "w" in L must reach on q_{f1} or q_{f2} from initial state q_0.

$$q_0 \ldots\ldots\ldots\ldots q_{f1} = w_1 \in L$$
$$q_1 \ldots\ldots\ldots\ldots q_{f2} = w_2 \in L$$

Hence L is union on some equivalent classes.

1=>2 proved

<u>2 => 3</u>

Let us assume equivalence relation E of finite index and L is the union of its equivalence classes.

Suppose xEy ……………………….(A)

Hence for any z in Σ^* , xzEyz ..[E is right invariant]

Myhill –Nerode Theorem

Since L is the union of equivalence classes so xz ϵ L and yz ϵ L

This means xzRLyz ……….(B)

From eqn (A) and eqn(B) we get,

$$xEy \Rightarrow xR_Ly$$

Hence E is the refinement of R_L and E is of finite index.

Therefore, index of R_L <= index of E

Hence R_L is of finite index.

<u>3 => 1</u>

Suppose xR_Ly it means xzR_Lyz [As R_L is right invariant and of finite index from statement 2]

This means $xzwR_Lyzw$ for any w ϵ Σ*

So we can write xR_Ly as xv ϵ L ⇔ yv ϵ L

If v=zw then xzw ϵ L ⇔ yzw ϵ L

So xR_Ly

Myhill –Nerode Theorem

Let us consider DFA M = { Q, Σ, δ, $q_{0,}$ F} from $R_{L.}$

Let Q be the set of equivalence classes of L

Now we define [x] ϵ Q be the equivalence class containing x

Let δ([x],a)= [xa]

Let F={[x] | x ϵ L}

Now we can construct a DFA. Hence statement 3 => statement 1

Closure Properties of Regular Language

- The property of Regular language that tells if a certain language are regular and a language L is formed from them by certain operations, then L is also regular .

- Closure properties of regular language:
  1. The union of two regular languages is regular.
  2. The intersection of two regular language is regular.
  3. The complement of a regular language is regular.
  4. The difference of two regular language is regular.
  5. The concatenation of regular language is regular.

Closure Properties of Regular Language

Theorem : If L_1 and L_2 are regular then $L_1 \cup L_2$ (Union) , L_1L_2 (Concatenation) and $L_1{}^*$ (Kleen star) are also regular.

Proof:

- ✓ According to Kleen theorem there exists a regular expression for every regular language.
- ✓ Therefore, there will be regular expression say $L(r_1)$ for L_1 and $L(r_2)$ for L_2.
- ✓ By the definition of regular expression r_1+r_2, r_1r_2, and $r_1{}^*$ is regular expression denoting the language $L_1 \cup L_2$, L_1L_2 and $L_1{}^*$
- ✓ Thus regular language is closed under union, concatenation, and star-closure.

Closure Properties of Regular Language

Theorem : if L_1 and L_2 are regular language then $L_1 \cap L_2$ is also regular language.

Proof:

- ✓ By the DE Morgan's Law, we know that
- ✓ We know that L1 and L2 are regular therefore complement of L1 and complement of L2 are also regular.
- ✓ Hence union of L1 and L2 are also regular.
- ✓ Hence L1+L2 is also regular.

Closure Properties of Regular Language

Theorem : if L_1 and “m” are regular language then L- m (Difference) is also regular language.

Proof:

- ✓ We know that $L\text{-}m=L\cap m$
- ✓ Here “m” is also regular
- ✓ We also know that intersection of two regular language is also regular.
- ✓ Hence L-m is regular.

Introduction of Grammar

- Another important notation that plays the important role in the study of automata is Grammar.
- The grammar is a set of rules to define valid sentences in any language.
- A grammar is constructed by the combination of Variables or Non-terminals denoted as "V" and terminals denoted as "T".
- Usually Variables are represented by Capital letters for example A,B, C.
- Terminals are represented by small letters like a,b,c
- Individual rule like A->BC is called production.

Chomsky’s Classification

- Noam Chomsky, a founder of formal language theory, gave a mathematical model for grammars in 1956.
- He shown the relationship between grammars by Chomsky Hierarchy.
- He classified the language into four types.

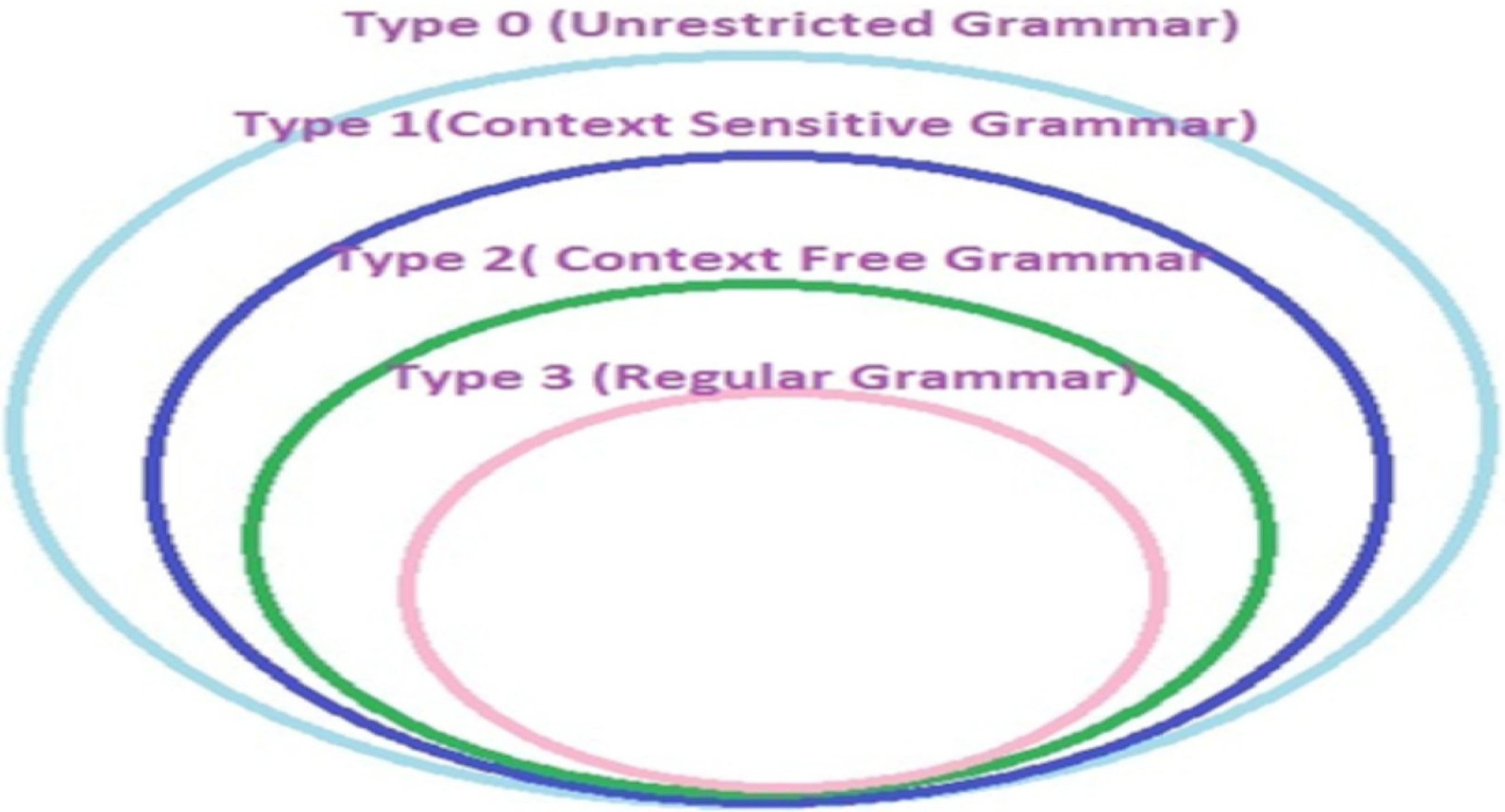
Type 0-------------------Unrestricted Grammar

Type 1-------------------Context sensitive grammar

Type 2-------------------Context Free Grammar

Type 3-------------------Regular Grammar

Chomsky's Classification


Type 0 (Unrestricted Grammar)
Type 1(Context Sensitive Grammar)
Type 2(Context Free Grammar
Type 3 (Regular Grammar)

Chomsky's Classification

- Type 0 Grammar:
  - ✓ No restrictions are made on the left and right sides of the grammar's productions
  - ✓ The grammar have the rule of the form α -> β where $\alpha \in (V+T)^+$ and $\beta \in (V+T)^*$ is in type 0 grammar.
- Type 1 Grammar:
  - ✓ The left-hand sides and right-hand sides of any production rule may be surrounded by a context of terminal and nonterminal symbols.
  - ✓ If a grammar is of type 0 and have the rule of the form α -> β where $|\alpha| \leq |\beta|$ then it is called as in Type 1 grammar.

Chomsky's Classification

- Type 2 Grammar:
  - ✓ If a grammar is of type 1 and have the rule of the form α -> β where α is a single Variable and $\beta \in$ (V+T)* then it is called as in Type 2 grammar.

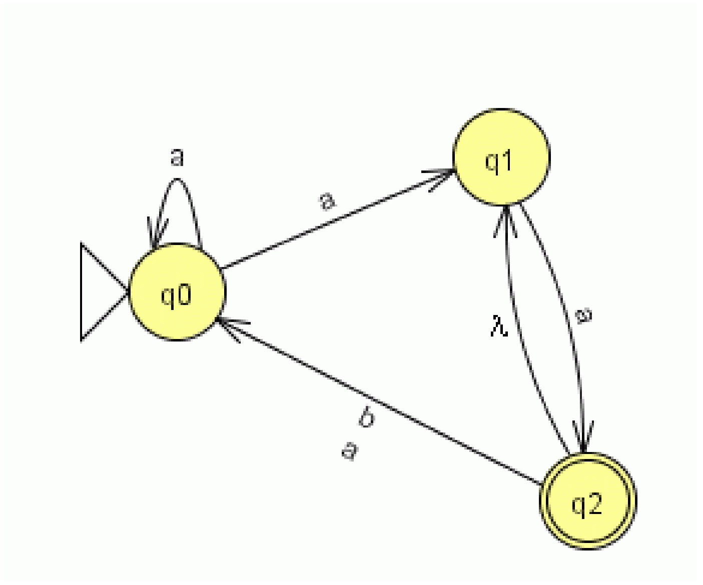
- Type 3 Grammar:
  - ✓ If a grammar is of type 2 and have the rule of the form α -> β and $\beta \in$ T*V (Right Linear) or $\beta \in$ VT* (Left Linear) then it is called as in Type 3 grammar.

JFLAP

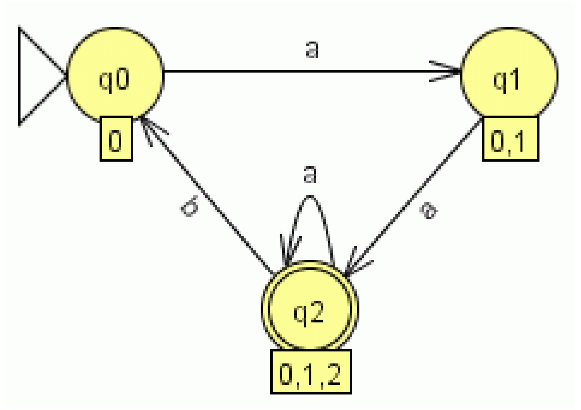
- **J**ava **F**ormal **L**anguages and **A**utomata **P**ackage
- Instructional tool to learn concepts of Formal Languages and Automata Theory
- Topics:
  - Regular Languages
  - Context-Free Languages
  - Recursively Enumerable Languages

DFA and NFA in JFLAP

- Create
  - DFA and NFA
  - Moore and Mealy
  - regular grammar
  - regular expression



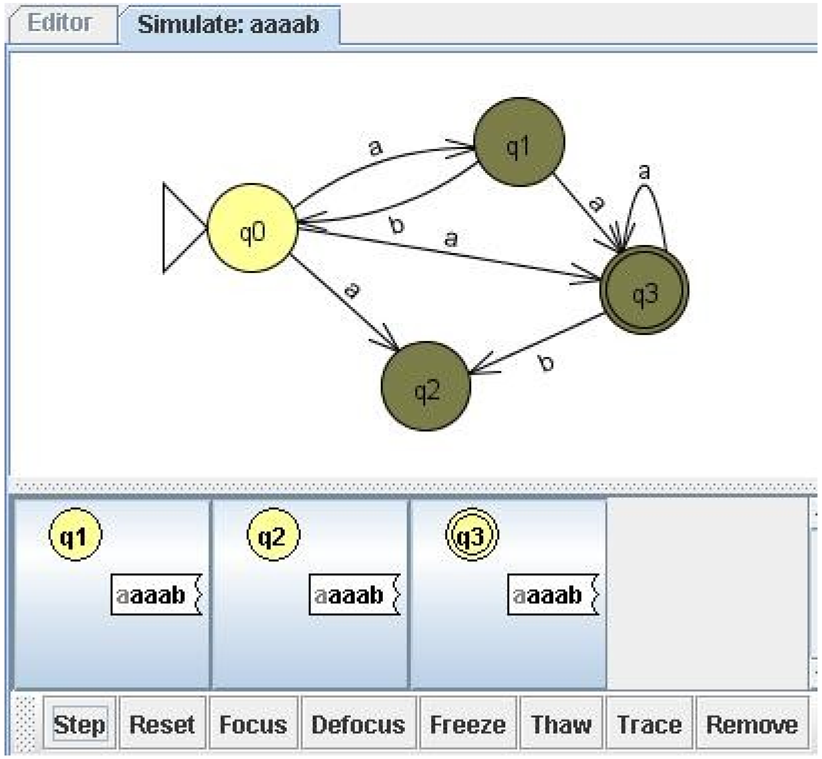
- Conversions
  - NFA to DFA to minimal DFA
  - NFA ←→ regular expression
  - NFA ←→ regular grammar



DFA and NFA in JFLAP

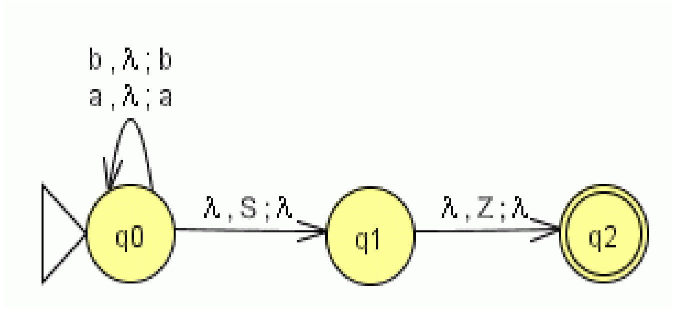
- Simulate DFA and NFA
  - Step with Closure or Step by State
  - Fast Run
  - Multiple Run

- Combine two DFA
- Compare Equivalence
- Brute Force Parser
- Pumping Lemma

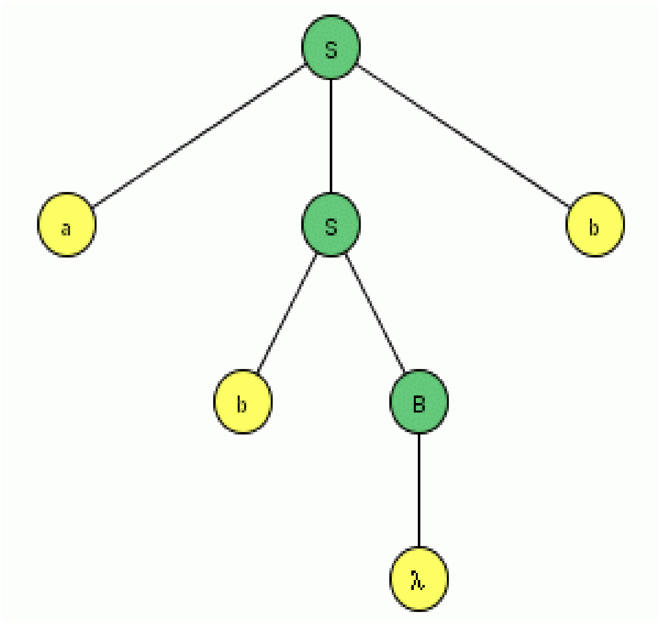


CFG and PDA in JFLAP

- Create
  - Nondeterministic PDA
  - Context-free grammar
  - Pumping Lemma

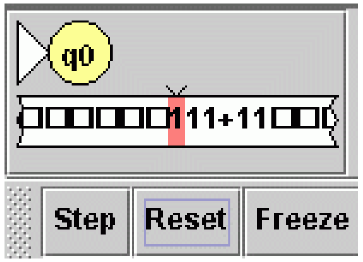


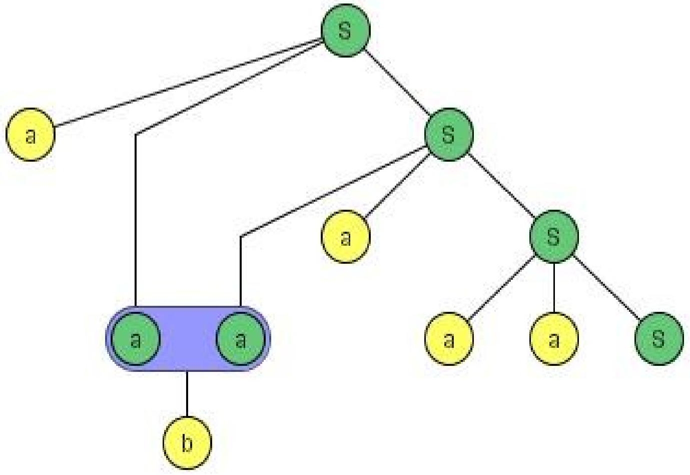
- Transform
  - PDA → CFG
  - CFG → PDA (LL & SLR parser)
  - CFG → CNF
  - CFG → Parse table (LL and SLR)
  - CFG → Brute Force Parser



Turing Machine in JFLAP

- Create
  - Turing Machine (1-Tape)
  - Turing Machine (multi-tape)
  - Building Blocks
  - Unrestricted grammar
- Parsing
  - Unrestricted grammar with brute force parser


q0
111+11
Step
Reset
Freeze


S
a
S
a
S
a
a
a
a
S
b