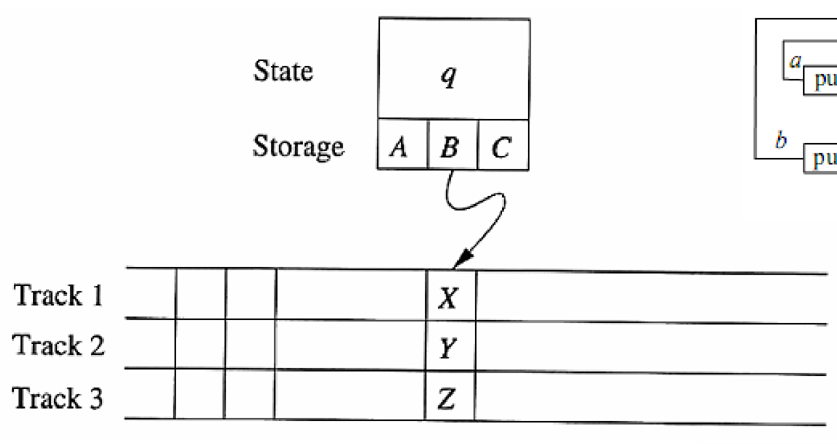
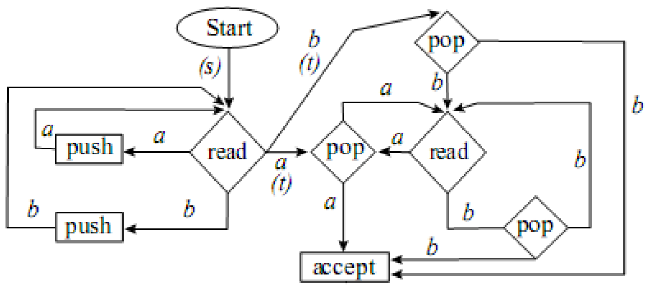
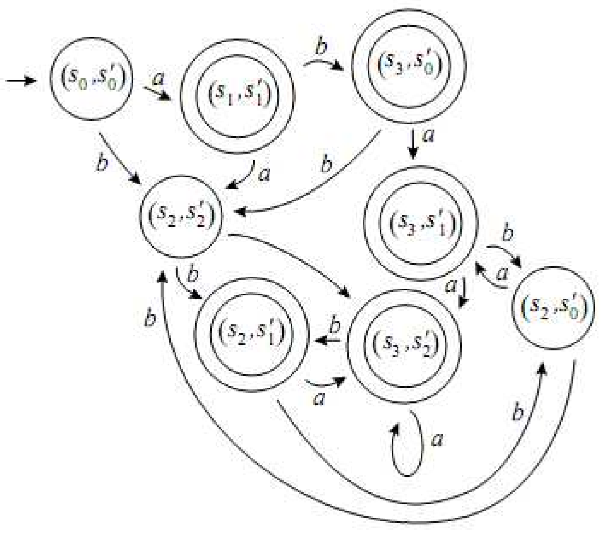
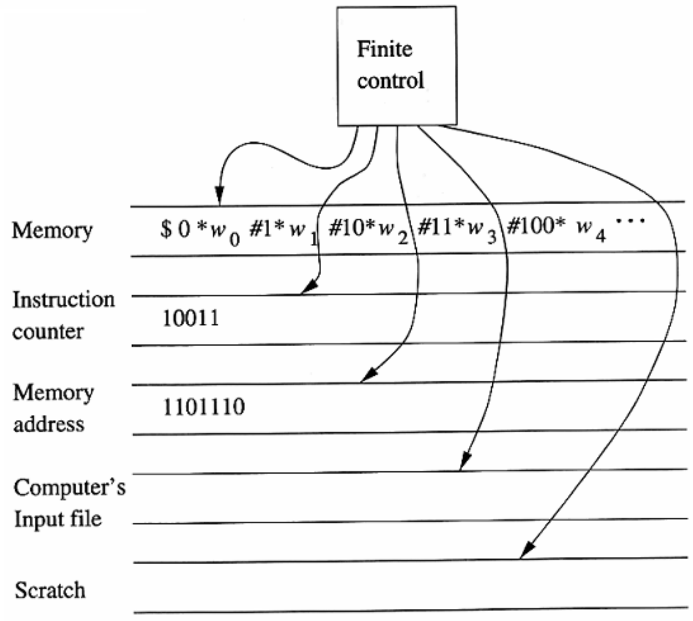
AUTOMATA THEORY

STUDY GUIDE









Prepared By

Sharafat Ibn Mollah Mosharraf

12th Batch (05-06)

Dept. of Computer Science & Engineering

University of Dhaka

TABLE OF CONTENTS

CHAPTER 1
THE METHODS AND THE TECHNIQUES

Concepts

1.1 Easier representation of *union* and *intersection* of sets

$$\begin{aligned} A_1 \cup A_2 \cup A_3 &= \{\, x \mid x \in A_1 \ \textbf{or}\ x \in A_2 \ \textbf{or}\ x \in A_3 \,\} \\ &= \{\, x \mid x \text{ is an element of at least one of the sets } A_1, A_2 \text{ and } A_3 \,\} \end{aligned}$$

More generally, if $A_1, A_2, \ldots$ are sets, we can write

$$\bigcup_{i=1}^{n} A_i = \{\, x \mid x \in A_i \ \text{ for at least one } i \text{ with } 1 \le i \le n \,\}$$

and
$$\bigcup_{i=1}^{\infty} A_i = \{\, x \mid x \in A_i \ \text{ for at least one } i \ \ge 1 \,\}$$

and
$$\bigcap_{i=1}^{n} A_i = \{\, x \mid x \in A_i \ \text{ for every } i \text{ with } 1 \le i \le n \,\}$$

1.2 Power set of a set

For any set A, the set of all subsets of A is referred to as the *power set of* A, written as $\mathbf{2^A}$.

The reason for this terminology and this notation is that if A has n elements, then 2^A has 2^n elements.

For example, suppose **A = {1, 2, 3}**. Then,

$$2^A = \{\, \emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \,\}$$

Notice that $\emptyset$ and A are both elements of 2^A. Therefore, the empty set ($\emptyset$) is a subset of every set, and every set is a subset of itself.

1.3 Shortening the description of a set

To say that x is an element of the set A, we write $\boldsymbol{x \in A}$.

Using this notation, we might describe a set A as following:

$$B = \{\, x \mid x \in A \text{ and } x \le 10 \,\}$$

which we read "*B is the set of all x such that x belongs to A and x* ≤ 10".

A common way to shorten this slightly is to write

$$B = \{\, x \in A \mid x \le 10 \,\}$$

which we read "*B is the set of x in A such that x* ≤ 10."

1.4 Logical Quantifiers and Quantified Statements

A statement "*there exists an x such that* $x^2 < 4$" is called a *quantified statement*.

The phrase "*there exists*" is called the *existential quantifier*; the variable x is said to be *bound* to the quantifier and is referred to as a *bound variable*.

Existential Quantifier

$\exists x\,(x^2 < 4)$	-	"*There exists an x such that* $x^2 < 4$"
$\exists x \in A(x^2 < 4)$	-	"*There exists an x* ***in A*** *such that* $x^2 < 4$"

Universal Quantifier

$\forall x\,(x^2 < 4)$	-	"*For every x,* $x^2 < 4$"
$\forall x \in A(x^2 < 4)$	-	"*For every x* ***in A***, $x^2 < 4$"

1.5 Alphabet

An alphabet is a finite set of symbols. We denote it by Σ. For example:

Σ_1 = { a, b, c, d, …, z } : the set of (lowercase) letters in English
Σ_2 = { 0, 1, …, 9 } : the set of (base 10) digits

- The letters in an alphabet are usually denoted using the letters from the beginning portion in English alphabet, for example, *a*, *b*, *c*, *d* etc.

1.6 String

A string over alphabet Σ is a finite sequence of symbols in Σ.

- Strings are usually denoted using the letters from the ending portion in English alphabet, for example, *s*, *t*, *u*, *v*, *w*, *x*, *y*, *z* etc.
- The length of a string x over Σ is the number of symbols in the string, and we denote this number by $|x|$.

  For example, some of the strings over the alphabet $\{a, b\}$ are *a*, *baa*, *aab* and *aabba*; and we have $|a| = 1$, $|baa| = |aab| = 3$ and $|aabba| = 5$.
- The *null string* or *empty string* (the string of length 0) is a string over Σ, no matter what alphabet Σ is. We denote it by ϵ (epsilon), or λ (lowercase Greek Lambda), or Λ (uppercase Greek Lambda).
- For any alphabet Σ, the set of all strings over Σ is denoted by Σ^*.

  For example, for $\mathbf{\Sigma = \{a, b\}}$, we have

$$\Sigma^* = \{a, b\}^* = \{\ \epsilon, a, b, aa, ab, bb, ba,\ aaa, aab, aba, \ldots\}$$

1.7 Language

A language is a set of strings over an alphabet Σ. Languages are usually denoted by **L**.

A few examples of languages over Σ are:

$L_1 = \{\ \epsilon, a, b, aa, ab, bb, ba\}$

$L_2 = \{\ ab, bab\ \}^* \cup \{\ \text{b}\ \}\{\ \text{bb}\ \}^*$

$L_3 = \{\ x \in \{a, b\}^*\ \mid\ |x| \le 10\}$

$L_4 = \{\ x \in \{a, b\}^*\ \mid\ |x| \text{ is odd}\}$

- Σ^* is a language for any alphabet Σ.
- $\emptyset$, the *empty* or *null language*, is a language over *any* alphabet. A *null language* means that there is no string in that language.
- $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Note that $\emptyset \neq \{\epsilon\}$; the former has no strings and the latter has one string.
- As the set of all strings over Σ is Σ^*, therefore, a language over Σ is a subset of Σ^*. That is,

$$\mathbf{L \subseteq \Sigma^*}$$

- The number of languages that can be constructed from the alphabet Σ is $\mathbf{2^{\Sigma^*}}$.
- The only important constraint on *what can be a language* is that all alphabets are finite.

Construction of new languages from an existing language

Because languages are *sets* of strings, new languages can be constructed using *set operations*.

The operations we can use to construct new languages from existing languages are as follows:

1. Union

2. Intersection
3. Difference (which includes the *complement* operation)
4. Concatenation (which includes the *Kleen star* operation)

For any two languages L_1, L_2 over an alphabet Σ, their *union* ($L_1 \cup L_2$), *intersection* ($L_1 \cap L_2$) and *difference* ($L_1 - L_2$) are also languages over Σ.

When we speak of the *complement* of a language L over Σ, we take the universal set to be the language Σ^* so that $L' = \Sigma^* - L$.

The concatenation operation

If x and y are elements of Σ^*, the concatenation of x and y is the string xy formed by writing the symbols of x and the symbols of y consecutively.

For example, if $x = abb$ and $y = ba$, then $xy = abbba$ and $yx = baabb$.

- For any string x, $x\epsilon = \epsilon x = x$.
- A string x is a ***substring*** of another string y if there are strings w and z (either or both of which may be null), so that $y = wxz$.
- A ***prefix*** of a string is an initial substring. For example, the prefixes of aba are ϵ (which is a prefix of every string), a, ab and aba.
- A ***suffix*** is a final substring.

Concatenation of languages

If $L_1, L_2 \subseteq \Sigma^*$, then $L_1L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$

For example,

$L_1 = \{hope, fear\}$
$L_2 = \{ful, less\}$
$\therefore L_1L_2 = \{hopeful, hopeless, fearful, fearless\}$

- Just as concatenating a string x with ϵ produces x, concatenating any language L with $\{\epsilon\}$ produces L. In other words, $L\{\epsilon\} = \{\epsilon\}L = L$.

Exponential notation in concatenation

We use exponential notation to indicate the number of items being concatenated. These can be individual symbols, strings, or languages. Thus, if Σ is an alphabet, $a \in \Sigma$, $x \in \Sigma^*$ and $L \subseteq \Sigma^*$, then

$a^k = aa\ldots a$
$x^k = xx\ldots x$
$\Sigma^k = \Sigma\Sigma\ldots\Sigma = \{x \in \Sigma^* \mid |x| = k\}$
$L^k = LL\ldots L = \bigcup_{i=0}^{k} L^i$ [L^i is the language L concatenated with itself i times]

where in each case there are k factors altogether.

An important special case is the one in which $k = 0$:

$a^0 = \epsilon$
$x^0 = \epsilon$
$\Sigma^0 = \{\epsilon\}$
$L^0 = \{\epsilon\}$

Therefore, the unit of concatenation for strings is ϵ, and for languages is $\{\epsilon\}$.

- The set of all strings that can be obtained by concatenating *any* number of elements of *L*:

$$L^{*} = \bigcup_{i=0}^{\infty} L^{i}$$

- The set of all strings that can be obtained by concatenating *one or more* elements of *L*:

$$L^{+} = \bigcup_{i=1}^{\infty} L^{i}$$

- $L^{+} = L^{*}L = LL^{*}$; $L^{*} = \epsilon \cup L^{+}$ and $L^{+} \subseteq L^{*}$.
- The operation * is also called the *star*, *closure*, *Kleene star* or *Kleene closure* operator.

Chapter 2 & 3
Finite Automata, Regular Expressions and Languages

Concepts

3.1 Regular Language and Regular Expression

Simple Language

A language that contains a single string of length one or does not contain any string at all is called a simple language.

For example, let, $\Sigma = \{1, 2, 3\}$.

$\therefore$ Simple languages over Σ are $\emptyset$, $\{\epsilon\}$, $\{1\}$, $\{2\}$ and $\{3\}$.

- $\emptyset$, $\{\epsilon\}$ are common to all simple languages.

Regular Language

Languages which are constructed from simple languages by applying the operations union, concatenation and Kleene star are called regular languages.

Regular Expression

A *regular expression* is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of the expression.

For example, let $\boldsymbol{L = \{w \mid w}$ **ends with 11}** be a language over the alphabet $\{0, 1\}$.

Then the regular expression for L is: $\mathbf{(0 + 1)^*11}$

Recursive Definition[1] of Regular Languages and Regular Expressions over Σ

The set R of *regular languages* over Σ and the corresponding regular expressions are defined as follows:

1. $\emptyset$ is an element of R, and the corresponding regular expression is $\emptyset$.
2. $\{\epsilon\}$ is an element of R, and the corresponding regular expression is ϵ.
3. For each $a \in \Sigma$, $\{a\}$ is an element of R, and the corresponding regular expression is a.
4. If L_1 and L_2 are any elements of R, and r_1 and r_2 are the corresponding regular expressions, then
   (a) $L_1 \cup L_2$ is an element of R, and the corresponding regular expression is $(r_1 + r_2)$;
   (b) L_1L_2 is an element of R, and the corresponding regular expression is (r_1r_2);
   (c) L_1^* is an element of R, and the corresponding regular expression is (r_1^*).
5. Only those languages that can be obtained by using statements 1 to 4 are regular languages over Σ.

3.2 Finite Automata (FA) / Deterministic Finite Automata (DFA) / Finite State Machine (FSM)

An FA is a 5-tuple $(Q, \Sigma, q_0, F, \delta)$, where

Q is a non-empty finite set (whose elements are thought of as *states*);

---

[1] In a *recursive definition*, there are three steps:
1. Certain objects are defined to be in the set.
2. Rules are listed for producing new objects in the set from other objects already in the set.
3. There are no other elements in the set.

Mathematical induction is a special case of a recursive definition.

Σ is a non-empty finite alphabet of *input symbols*;

$q_0 \in Q$ (the *initial* state);

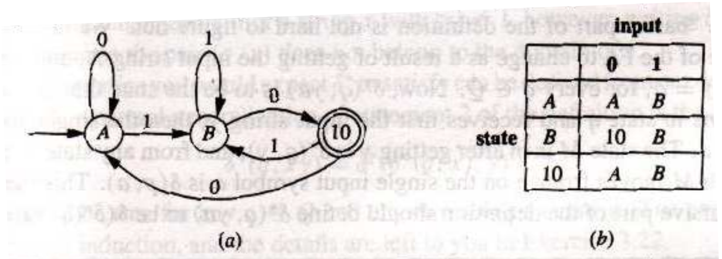
$F \subseteq Q$ (the set of *final* / *accepting* states);

δ is a function from $Q \times \Sigma$ to Q (i.e., $\delta : Q \times \Sigma \to Q$, the *transition* function)

For any element $q \in Q$ and any symbol $a \in \Sigma$, we interpret $\delta(q, a)$ as the state to which the FA moves if it is in state q and receives the input a.

A DFA is usually denoted by M.

Example of DFA



		input	
		0	1
state	A	A	B
	B	10	B
	10	A	B

Figure 3.3 | A simplified finite automaton recognizing $\{0, 1\}^*\{10\}$: (*a*) Transition diagram; (*b*) Transition table.

The Extended Transition Function $\hat{\delta}$ for a DFA

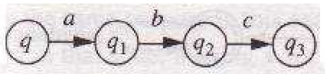
Let $M = (Q, \Sigma, q_0, F, \delta)$ be an FA. We define the function

$$\hat{\delta} : Q \times \Sigma^* \to Q$$

as follows:

1. For any $q \in Q$, $\hat{\delta}(q, \epsilon) = q$
2. For any $q \in Q$, $y \in \Sigma^*$, and $a \in \Sigma$, $\hat{\delta}(q, ya) = \delta(\hat{\delta}(q, y), a)$

Example:



$$\begin{aligned}
\delta^*(q, abc) &= \delta(\delta^*(q, ab), c) \\
&= \delta(\delta(\delta^*(q, a), b), c) \\
&= \delta(\delta(\delta^*(q, \Lambda a), b), c) \\
&= \delta(\delta(\delta(\delta^*(q, \Lambda), a), b), c) \\
&= \delta(\delta(\delta(q, a), b), c) \\
&= \delta(\delta(q_1, b), c) \\
&= \delta(q_2, c) \\
&= q_3
\end{aligned}$$

Acceptance of a string or language by a DFA

Let $M = (Q, \Sigma, q_0, F, \delta)$ be a DFA.

A string $x \in \Sigma^*$ is *accepted* by M if $\hat{\delta}(q_0, x) \in F$.

The language *accepted* / *recognized* by M, is the set

$$L(M) = \{ x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F \}$$

➢ If $L = L(M)$ for some DFA, then L is a regular language.

3.3 Nondeterministic Finite Automata (NFA)

An NFA is a 5-tuple $(Q, \Sigma, q_0, F, \delta)$, where

Q is a non-empty finite set (whose elements are thought of as *states*);

Σ is a non-empty finite alphabet of *input symbols*;

$q_0 \in Q$ (the *initial* state);

$F \subseteq Q$ (the set of *final* / *accepting* states);

δ is a function from $Q \times \Sigma$ to 2^Q (i.e., $\delta : Q \times \Sigma \to 2^Q$, the *transition* function)

For any element $q \in Q$ and any symbol $a \in \Sigma$, we interpret $\delta(q, a)$ as the set of states to which the NFA moves if it is in state q and receives the input a.

An NFA is usually denoted by N.

Example of NFA

Consider the language $L = \{w \mid w \text{ ends with } 01\}$.

The NFA would be defined as $N = (\{q_0, q_1, q_2\}, \{0, 1\}, q_0, \{q_2\}, \delta)$, where δ is defined using the following rules:

$\delta(q_0, 0) = \{q_0, q_1\}$,
$\delta(q_0, 1) = \{q_0\}$,
$\delta(q_1, 1) = \{q_2\}$

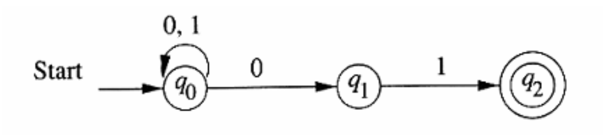
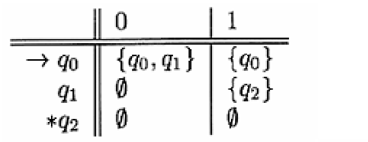


Figure 2.9: An NFA accepting all strings that end in 01



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

Transition table for an NFA that accepts all strings ending in 01

The Extended Transition Function $\hat{\delta}$ for an NFA

Let $N = (Q, \Sigma, q_0, F, \delta)$ be an NFA. We define the function

$$\hat{\delta} : Q \times \Sigma^* \to 2^Q$$

as follows:

1. For any $q \in Q$, $\hat{\delta}(q, \epsilon) = \{q\}$

2. For any $q \in Q, y \in \Sigma^*$, and $a \in \Sigma$, $\quad \hat{\delta}(q, ya) = \bigcup_{r \in \hat{\delta}(q,y)} \delta(r, a)$

Acceptance of a string or language by an NFA

Let $N = (Q, \Sigma, q_0, F, \delta)$ be an NFA.

A string $x \in \Sigma^*$ is *accepted* by N if $\hat{\delta}(q_0, x) \cap F \neq \emptyset$ (i.e., $\hat{\delta}(q_0, x)$ contains *at least one* accepting state).

The language *accepted* / *recognized* by N, is the set

$$L(N) = \{ x \in \Sigma^* \mid \hat{\delta}(q_0, x) \cap F \neq \emptyset \}$$

3.4 ϵ-NFA (ϵ-Transition of an NFA)

An NFA with ϵ-transitions (abbreviated ϵ-NFA) is a 5-tuple $(Q, \Sigma, q_0, F, \delta)$, where

Q is a non-empty finite set (whose elements are thought of as *states*);

Σ is a non-empty finite alphabet of *input symbols*;

$q_0 \in Q$ (the *initial* state);

$F \subseteq Q$ (the set of *final* / *accepting* states);

δ is a function from $Q \times (\Sigma \cup \{\epsilon\})$ to 2^Q (i.e., $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$, the *transition* function)

Example of ϵ-NFA

In Fig. 2.18 is an ϵ-NFA that accepts decimal numbers consisting of:

1. An optional + or − sign,
2. A string of digits,
3. A decimal point, and
4. Another string of digits. Either this string of digits, or the string (2) can be empty, but at least one of the two strings of digits must be nonempty.

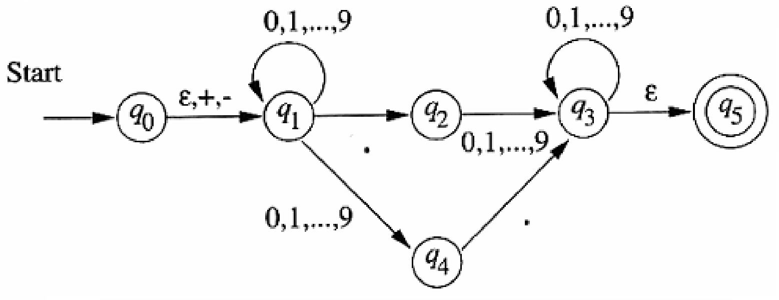
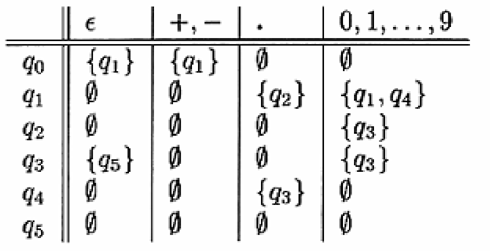


Figure 2.18: An ϵ-NFA accepting decimal numbers



	ϵ	$+,-$	$.$	$0,1,\ldots,9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 2.20: Transition table for Fig. 2.18

ϵ-Closure of a Set of States

[***Informal definition:*** If S is the set of states from where we can go to other states using ϵ-transitions, then the ϵ-closure of S is the set of all states that can be reached from elements of S by using zero, one or more ϵ-transitions.]

Let $N = (Q, \Sigma, q_0, F, \delta)$ be an ϵ-NFA, and let S be any subset of Q. The ϵ-closure of S is the set $\epsilon(S)$ defined as follows:

1. Every element of S is an element of $\epsilon(S)$;
2. For any $q \in \epsilon(S)$, every element of $\delta(q, \epsilon)$ is in $\epsilon(S)$;
3. No other elements of Q are in $\epsilon(S)$.

The Extended Transition Function $\hat{\delta}$ for an ϵ-NFA

Let $N = (Q, \Sigma, q_0, F, \delta)$ be an ϵ-NFA. We define the function

$$\hat{\delta} : Q \times \Sigma^* \to 2^Q$$

as follows:

1. For any $q \in Q$, $\hat{\delta}(q, \epsilon) = \epsilon(\{q\})$

2. For any $q \in Q, y \in \Sigma^*$, and $a \in \Sigma$, $\quad \hat{\delta}(q, ya) = \epsilon\left(\bigcup_{r \in \hat{\delta}(q,y)} \delta(r, a) \right)$

Acceptance of a string or language by an ϵ-NFA

Let $N = (Q, \Sigma, q_0, F, \delta)$ be an ϵ-NFA.

A string $x \in \Sigma^*$ is *accepted* by N if $\hat{\delta}(q_0, x) \cap F \neq \emptyset$ (i.e., $\hat{\delta}(q_0, x)$ contains *at least one* accepting state).

The language *accepted* / *recognized* by N, is the set

$$L(N) = \{\, x \in \Sigma^* \mid \hat{\delta}(q_0, x) \cap F \neq \emptyset \,\}$$

3.5 Kleene's Theorem

Part 1: Any regular language can be accepted by a finite automaton.
Part 2: The language accepted by any finite automaton is regular.

Theorem 4.3 (*of Martin*) [*It's just Kleene's theorem expressed in another way*]

For any alphabet Σ, and any language $L \subset \Sigma^*$, these three statements are equivalent ***if L is regular***:

1. L is recognized by an FA / DFA.
2. L is recognized by an NFA.
3. L is recognized by an ϵ-NFA.

3.6 Algebraic Laws for Regular Expressions

$x + y = y + x$	Commutative Law
$(x + y) + z = x + (y + z)$	Associative Law for Union
$(xy)z = x(yz)$	Associative Law for Concatenation
$\emptyset + x = x + \emptyset = x$	Identity Law for Union
$\epsilon x = x\epsilon = x$	Identity Law for Concatenation
$\emptyset x = x\emptyset = \emptyset$	Annihilator Law for Concatenation
$x(y + z) = xy + xz$	Left Distributive Law of Concatenation Over Union
$(x + y) z = xz + yz$	Right Distributive Law of Concatenation Over Union
$x + x = x$	Idempotence Law for Union
$(x^*)^* = x^*x^* = x^*$	Laws Involving Closures
$(x + y)^* = (x^*y^*)^* = (x^* + y^*)^*$	
$(xy)^*x = x(yx)^*$	
$\emptyset^* = \epsilon$ [2]	
$\epsilon^* = \epsilon$	
$x^+ = xx^* = x^*x$ [3]	
$x^* = x^+ + \epsilon = xx^* + \epsilon = x^*x + \epsilon$ [4]	
$x? = \epsilon + x = x + \epsilon$	This is actually the definition for the '?' operator.
$x + yz \neq (x + y)z$	Common mistakes made
$xx \neq x$	

3.7 Building Regular Expressions

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

Zero or more – a^*

a^* means "zero or more a's." To say "zero or more ab's," that is, $\{\epsilon, ab, abab, ababab, \ldots\}$, you need to say $(ab)^*$. Don't say ab^*, because that denotes the language $\{a, ab, abb, abbb, \ldots\}$.

One or more – $a^+ = aa^* = a^*a$

Zero or one – $a? = a + \epsilon = \epsilon + a$

Any string *at all* – $(a + b + c)^*$

Any *non-empty* string – $(a + b + c)(a + b + c)^*$

This can be written as any character from Σ followed by any string at all: $(a + b + c)(a + b + c)^*$.

[2] The *closure* of $\emptyset$ contains only the string ϵ.

[3] $x^+ = x + xx + xxx + \ldots$
$x^* = \epsilon + x + xx + xxx + \ldots$
$\therefore xx^* = x\epsilon + xx + xxx + \ldots$
$= x + xx + xxx + \ldots$
$= x^+$

[4] $x^* = \epsilon + x + xx + xxx + \ldots$
$= \epsilon + x^+$
$= x^+ + \epsilon$

Strings *not* containing '*a*' – $(b + c)^*$

Strings containing *exactly one* '*a*' – $(b + c)^*a(b + c)^*$

To describe any string that contains exactly one *a*, put "any string not containing an *a*," on either side of the *a*.

Strings containing *no more than* three *a*'s - $(b + c)^*a?(b + c)^*a?(b + c)^*a?(b + c)^*$

We can describe the string containing zero, one, two, or three *a*'s (and nothing else) as

$$a?a?a?$$

Now we want to allow arbitrary strings not containing *a*'s at the places marked by X's:

$$\mathrm{X}a?\mathrm{X}a?\mathrm{X}a?\mathrm{X}$$

So we put in $(b + c)^*$ for each X:

$$(b + c)^*a?(b + c)^*a?(b + c)^*a?(b + c)^*$$

This is equivalent to: $(b + c)^*(a + \epsilon)(b + c)^*(a + \epsilon)(b + c)^*(a + \epsilon)(b + c)^*$.

Strings containing *at least one* occurrence of *each* symbol in Σ

The problem here is that we cannot assume the symbols are in any particular order. We have no way of expressing "in any order", so we have to list the possible orders:

$$abc + acb + bac + bca + cab + cba$$

To make it easier to see what's happening, let's put an X in every place we want to allow an arbitrary string:

$$\mathrm{X}a\mathrm{X}b\mathrm{X}c\mathrm{X} + \mathrm{X}a\mathrm{X}c\mathrm{X}b\mathrm{X} + \mathrm{X}b\mathrm{X}a\mathrm{X}c\mathrm{X} + \mathrm{X}b\mathrm{X}c\mathrm{X}a\mathrm{X} + \mathrm{X}c\mathrm{X}a\mathrm{X}b\mathrm{X} + \mathrm{X}c\mathrm{X}b\mathrm{X}a\mathrm{X}$$

Finally, replacing the X's with $(a + b + c)^*$ gives the final answer:

$$(a + b + c)^*a(a + b + c)^*b(a + b + c)^*c + (a + b + c)^*a(a + b + c)^*c(a + b + c)^*b +$$
$$(a + b + c)^*b(a + b + c)^*a(a + b + c)^*c + (a + b + c)^*b(a + b + c)^*c(a + b + c)^*a +$$
$$(a + b + c)^*c(a + b + c)^*a(a + b + c)^*b + (a + b + c)^*c(a + b + c)^*b(a + b + c)^*a$$

Strings containing *no* runs of *a*'s of length *greater than* two

We can fairly easily build an expression containing no *a*, one *a*, or one *aa*:

$$(b + c)^*(\epsilon + a + aa)(b + c)^*$$

But if we want to repeat this, we need to be sure to have at least one non-*a* between repetitions:

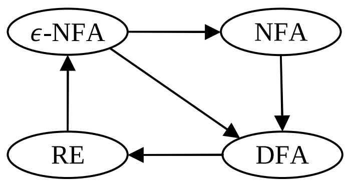
$$(b + c)^*(\epsilon + a + aa)(b + c)^*((b + c)(b + c)^*(\epsilon + a + aa)(b + c)^*)^*$$

Strings in which *all* runs of *a*'s have lengths that are multiples of three – $(aaa + b + c)^*$

3.8 Precedence of Regular Expression Operators

* $^+$?	(Unary operators)	*Highest*
.	(Concatenation operator)	
+	(Union operator)	*Lowest*

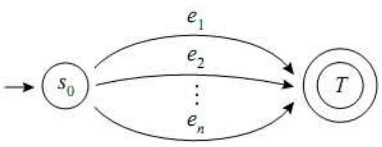
3.9 Conversion Among RE, DFA, NFA and ϵ-NFA



$$\begin{tikzcd} \epsilon\text{-NFA} \arrow[r] \arrow[dr] & \text{NFA} \arrow[d] \\ \text{RE} \arrow[u] & \text{DFA} \arrow[l] \end{tikzcd}$$

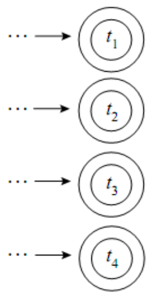
3.10 Converting DFA / NFA / ϵ-NFA into RE by Eliminating States

The process for constructing the regular expression is to first have *only one initial and one terminal state*. We then eliminate one state at a time from the state diagram and resulting transition graphs and in each case get a transition graph with *e*-arrows between states, where ***e*** is a regular expression. Eventually we get a transition graph of the form:

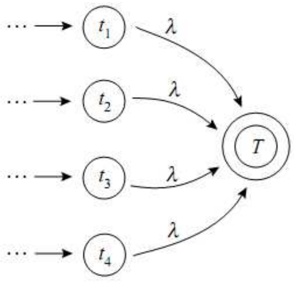


which accepts the expression $e_1 + e_2 + \ldots + e_n$.

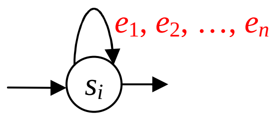
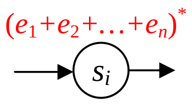
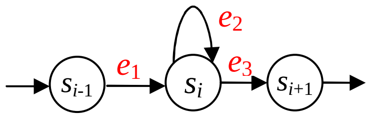
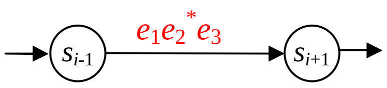
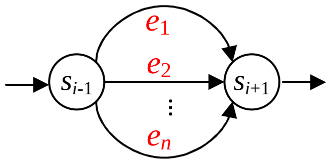
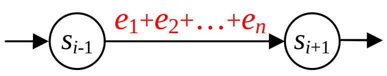
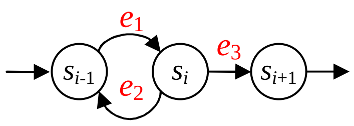
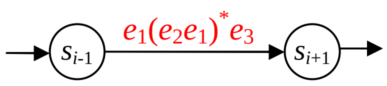
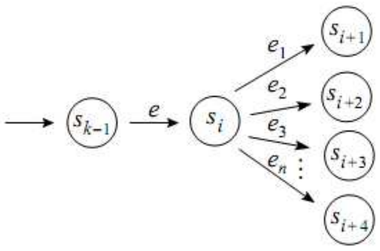
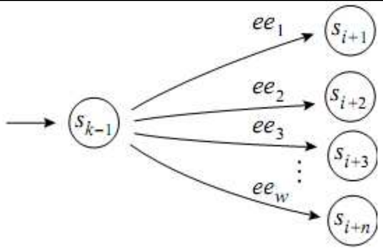
If there are more than one terminal states, say there are terminal states $t_1, t_2, t_3, \ldots, t_n$, then replace the states



with

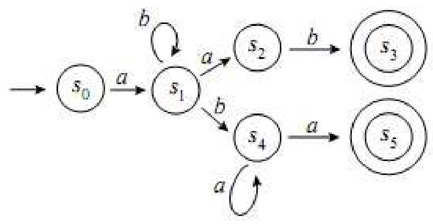


To eliminate the state s_i, we use the following rules:

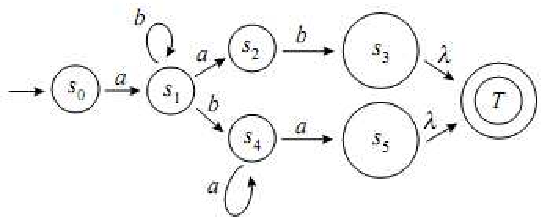
Original Diagram	**Equivalent RE**
	
	
	
	
 s_{k-1} e s_i e_1 e_2 e_3 e_n s_{i+1} s_{i+2} s_{i+3} s_{i+4}	 s_{k-1} ee_1 ee_2 ee_3 ee_w s_{i+1} s_{i+2} s_{i+3} s_{i+n}

Example 1

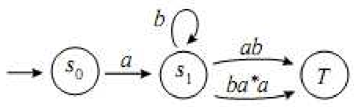
Assume we begin with automaton



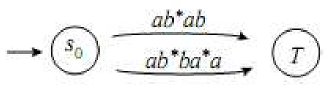
We then add a new terminal state T to get the automaton



We now apply rule (2) to get the automaton



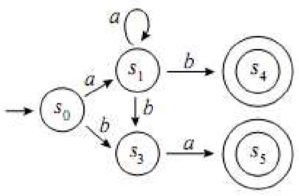
Apply rules (2) and (3) to get the automaton



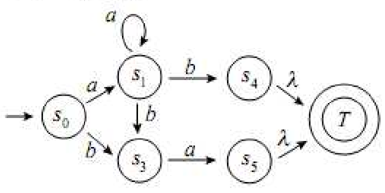
Hence the regular expression is $\boldsymbol{ab^*ab \vee ab^*ba^*a}$.

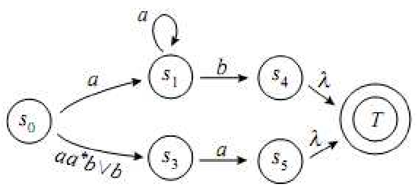
Example 2

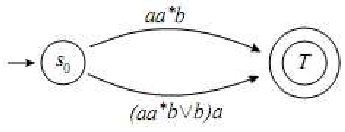
Given the automaton



we go through the following steps



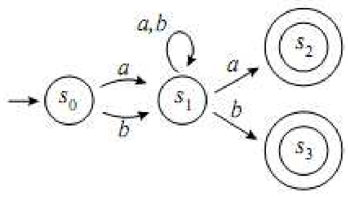




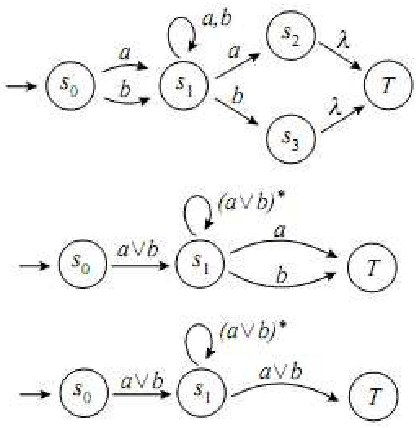
to get the regular expression $((\boldsymbol{aa^*b} \vee \boldsymbol{b})\boldsymbol{a}) \vee \boldsymbol{aa^*b})$.

Example 3

Given the automaton

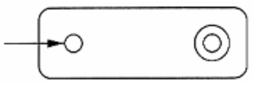
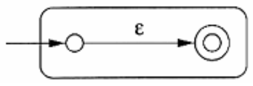
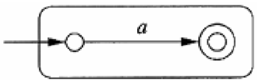
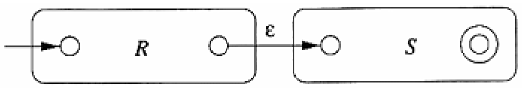
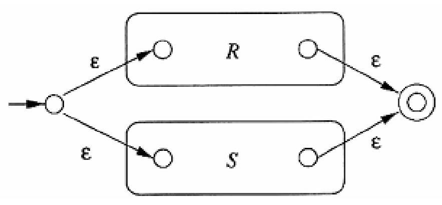
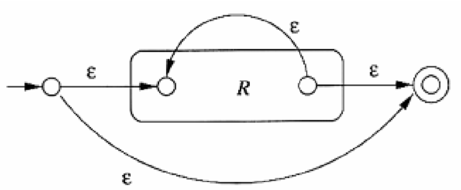


we go through the following steps,



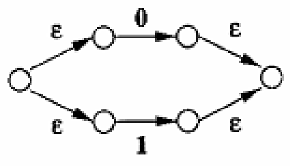
to get the regular expression $(\boldsymbol{a} \vee \boldsymbol{b})(\boldsymbol{a} \vee \boldsymbol{b})^*(\boldsymbol{a} \vee \boldsymbol{b})$. Note that the process is not unique and that by taking different steps, we would have had a different, but equivalent, regular expression. Thus both expressions would have described the same set.

3.11 Converting RE into ϵ-NFA

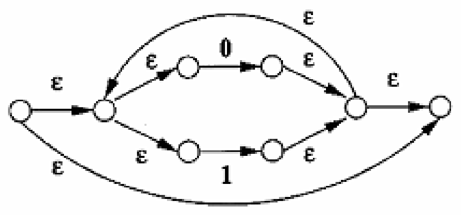
Original RE	Equivalent Diagram
∅	
ϵ	
a	
RS	
R + S	
R^*	

Example

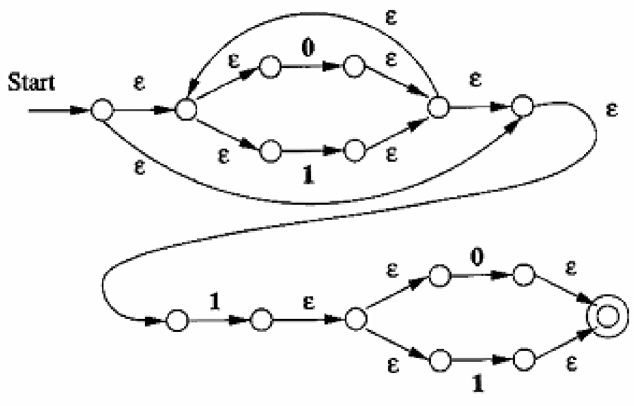
Let's convert the RE (0 + 1)*1(0 + 1) to an ϵ-NFA.



(a)



(b)



(c)

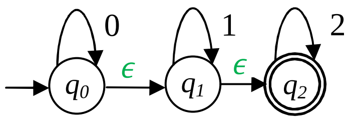
3.12 **Converting ϵ-NFA into NFA by Eliminating ϵ-Transitions**

Suppose we want to replace an ϵ-move from vertex v_1 to vertex v_2. Then we proceed as follows:

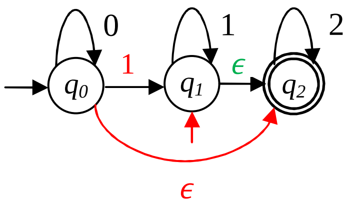
1. Find all the edges starting from v_2.
2. Duplicate all these edges starting from v_1, without changing the edge labels.
3. If v_1 is an initial state, make v_2 also an initial state.
4. If v_2 is a final state, make v_1 as the final state.

Example:

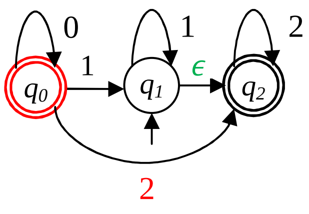
Consider the ϵ-NFA below:



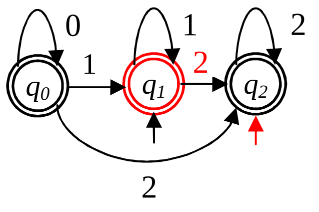
We go through the following steps to remove the ϵ-moves:



(a)



(b)



(c)

3.13 **Converting ϵ-NFA into DFA**

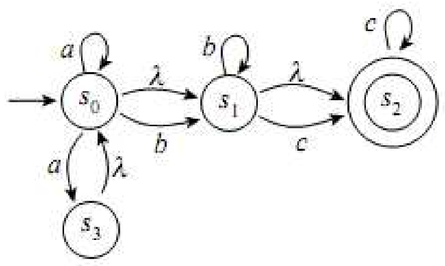
Let, $E = (Q, \Sigma, q_0, F, \delta)$ be an ϵ-NFA.

The corresponding DFA is $M = (Q', \Sigma, q_0', F', \delta')$ such that

1. Q' is defined as: $\epsilon(q) \in Q'$ where $q \in Q$.
2. q_0' is $\epsilon(q_0)$.
3. $F' = \{\epsilon(q_a) \in F' \mid \epsilon(q_a) \cap F \neq \emptyset\}$.
4. δ' is defined as:
   For each element of $\epsilon(q)$, find $\delta(q, a\epsilon)$. The destination DFA state is the state which contains only the resultant ϵ-NFA states.

Example

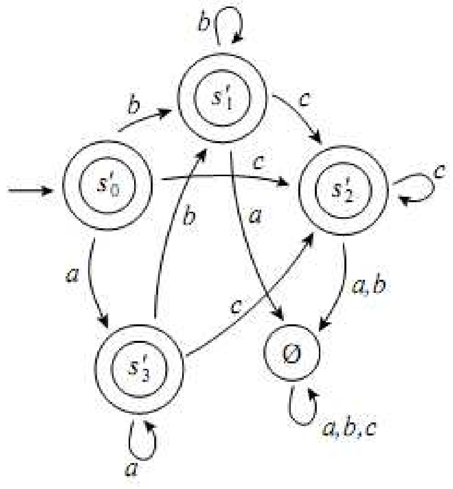
Example 3.12 Given the automaton $(M = (\Sigma, Q, s_0, \Upsilon, F)$



which has λ-moves, we construct $M' = (\Sigma, Q', s_0', \Upsilon', F')$ containing no λ-moves: $E(s_0) = \{s_0, s_1, s_2\}$, $E(s_1) = \{s_1, s_2\}$, $E(s_2) = \{s_2\}$, and $E(s_3) = \{s_0, s_1, s_2, s_3\}$. Denote these sets by s_0', s_1', s_2', and s_3' respectively. Then Υ' is given by the following table

	a	b	c
s_0'	s_3'	s_1'	s_2'
s_1'	$\emptyset$	s_1'	s_2'
s_2'	$\emptyset$	$\emptyset$	s_2'
s_3'	s_3'	s_1'	s_2'

giving the λ-free automaton



Both automata generate the language $\boldsymbol{a^*b^*c^*}$.

3.14 Converting NFA into DFA by Subset Construction

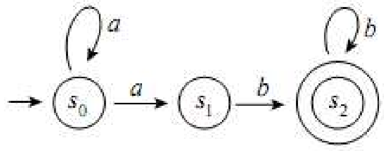
Let, $N = (Q, \Sigma, s_0, F, \delta)$ be an NFA.

The corresponding DFA $M = (Q', \Sigma, \{s_0\}, F', \delta')$ can be constructed using the following algorithm:

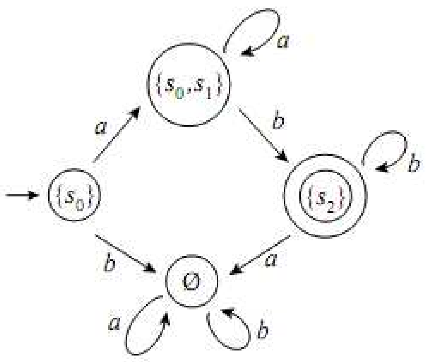
1. Begin with the state $\{s_0\}$ where s_0 is the start state of the nondeterministic automaton.
2. For each $a_i \in \Sigma$, construct an a_i-arrow from $\{s_0\}$ to the set consisting of all states such that there is an a_i-arrow from s_0 to that state.
3. For each newly constructed set of states s_j and for each $a_i \in \Sigma$, construct an a_i-arrow from s_j to the set consisting of all states such that there is an a_i-arrow from an element of s_j to that state.
4. Continue this process until no new states are created.
5. Make each set of states s_j that contains an element of the acceptance set of the nondeterministic automaton, into an acceptance state.

Example 1

Consider the nondeterministic automaton N



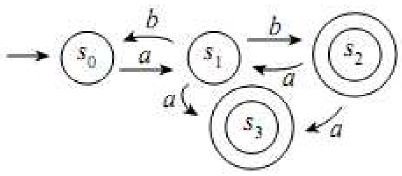
Construct an a-arrow from $\{s_0\}$ to the set of all states so that there is an a-arrow from s_0 to that state. Since there is an a-arrow from s_0 to s_0 and an a-arrow from s_0 to s_1, we construct an a-arrow from $\{s_0\}$ to $\{s_0, s_1\}$. There is no b-arrow from s_0 to any state. Hence the set of all states such that there is a b-arrow to one of these states is empty and we construct a b-arrow from $\{s_0\}$ to the empty set $\varnothing$. We now consider the state $\{s_0, s_1\}$. We construct an a-arrow from $\{s_0, s_1\}$ to the set of all states such that there is an a-arrow from either s_0 or s_1 to that state. Thus we construct an a-arrow from $\{s_0, s_1\}$ to itself. We construct a b-arrow from $\{s_0, s_1\}$ to the set of all states such that there is a b-arrow from either s_0 or s_1 to that state. Thus construct a b-arrow from $\{s_0, s_1\}$ to $\{s_2\}$. Since there are no a-arrows or b-arrows from any state in the empty set to any other state, we construct an a-arrow and a b-arrow from the empty set to itself. Consider $\{s_2\}$. Since there is no a-arrow from s_2 to any other state, we construct an a-arrow from $\{s_2\}$ to the empty set. Since the only b-arrow from s_2 is to itself, we construct a b-arrow from $\{s_2\}$ to itself. The acceptance states consist of all sets which contain an element of the terminal set of N. In this case $\{s_2\}$ is the only acceptance state. We have now completed the state diagram



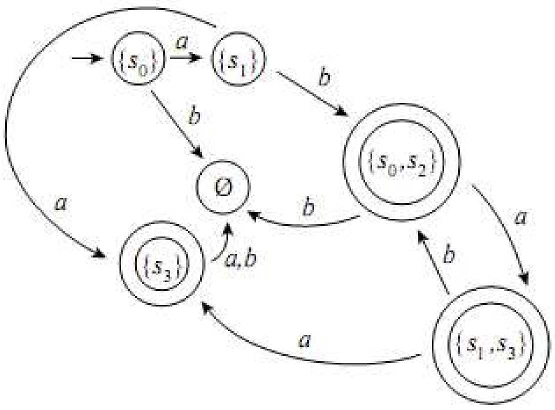
which is easily seen to be the state diagram of a deterministic automaton. This automaton also reads the same language as N, namely the language described by the expression $\boldsymbol{aa^*bb^*}$.

Example 2

Given the nondeterministic automaton



using the same method as above we complete the deterministic automaton



3.15 Applications of Regular Expressions

1. **Finding Patterns in Text.** The regular expression notation is valuable for describing searches for interesting patterns. For example, the regular expression for a valid e-mail address is as follows:

   ^[_a-z0-9-]+(.[_a-z0-9-]+)*@[a-z0-9-]+(.[a-z0-9-]+)*(.[a-z]{2,8})

2. **Lexical Analysis.** One of the oldest applications of regular expressions was in specifying the component of a compiler called a *lexical analyzer*. This component scans the source program and recognizes all tokens, those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens, but there are many others.

CHAPTER 4
PROPERTIES OF REGULAR LANGUAGES

Concepts

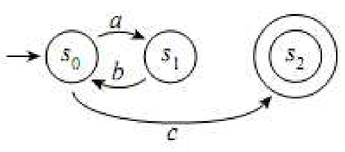
4.1 **Concatenation of Two DFAs**

Let, M_1, M_2 be two DFAs. To make M_1M_2, the following rules are followed:

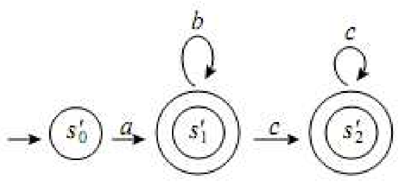
1. Place M_2 after M_1.
2. For each of the states in M_1 which has an arrow – e.g., labeled a – towards the accepting states, place another arrow from that state towards the initial state of M_2 with the same label (in this case, a).
3. Make all the accepting states in M_1 non-accepting.

Example 1

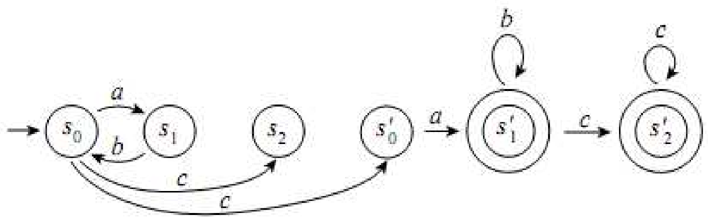
Let L_1 be the language described by the language $(ab)^*c$ and having automaton M_1 with state diagram



Let L_2 be the language described by the language ab^*c^* and having automaton M_2 with state diagram



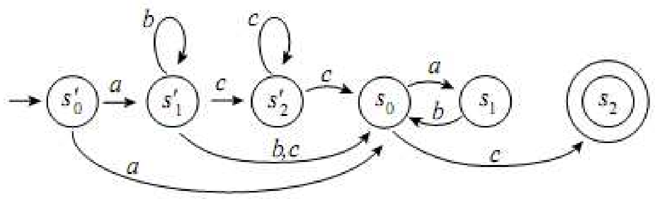
To find the state diagram for the language L_1L_2, place the state diagram for M_2 after the state diagram for M_1. Since there is a c-arrow from s_0 to s_2, and s_2 is an acceptance state, add a c-arrow from s_0 to s'_0. The state diagram



is the state diagram for M, the automaton for L_1L_2.

Example 2

Let L_1 and L_2 and their respective automata be the same as those in the previous example. To find the automaton for the language for L_2L_1 is slightly more complicated. First we place the state diagram for M_1 after the state diagram for M_2. There is an a-arrow from s'_0 to s'_1 and s'_1 is an acceptance state, so place an a-arrow from s'_0 to s_0. There is a b-arrow from s'_1 to s'_1 and s'_1 is an acceptance state, so place a b-arrow from s'_1 to s_0. There is a c-arrow from s'_1 to s'_2 and s'_2 is an acceptance state, so place a c-arrow from s'_1 to s_0. There is a c-arrow from s'_2 to s'_2 and s'_2 is an acceptance state, so place a c-arrow from s'_2 to s_0. Then change s'_1, s'_2 so that they are not acceptance states. Thus we have the state diagram



which is the state diagram for M, the automaton for $L_2 L_1$.

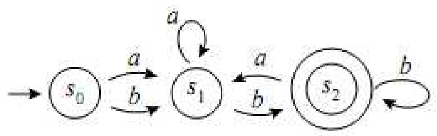
4.2 **Union of Two DFAs**

Let, M_1, M_2 be two DFAs. To make $M_1 \cup M_2$, the following rules are followed:

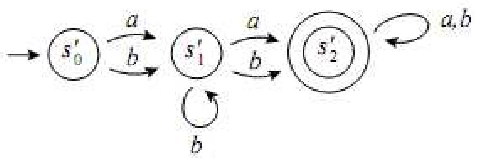
1. Create a new initial state.
2. Detect which arrows are out from each of the initial states of M_1 and M_2. Add similar arrows from the new initial state towards those states where those arrows were gone.

Example 1

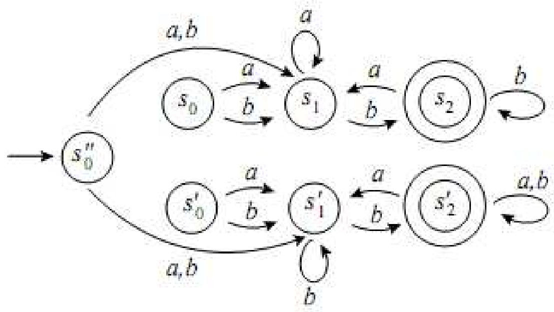
Let M be the automaton



and M' be the automaton

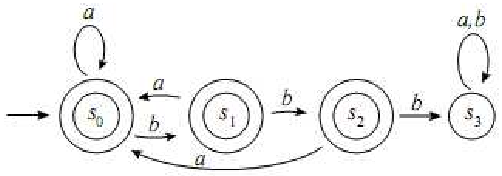


Using the above procedure we have the automaton M'' which accepts the union of $M(L)$ and $M'(L)$ given by

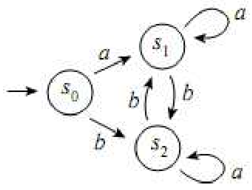


Example 2

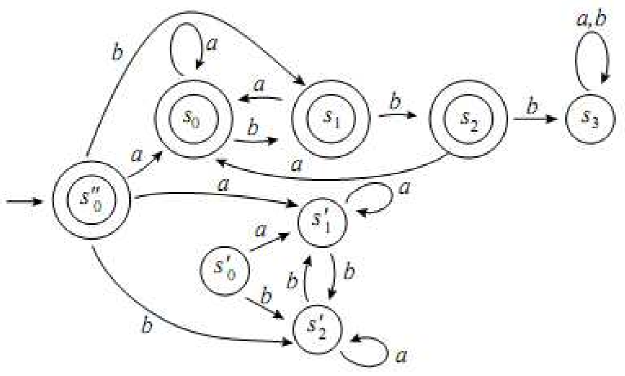
Let M be the automaton



and M' be the automaton



Using the above procedure we have the automaton M which accepts the union of $M_1(L)$ and $M_1'(L)$



4.3 **Complement / Transpose of a DFA**

Let, M be a DFA. To make $\overline{M}$, just make the non-final states final and the final states non-final.

Example

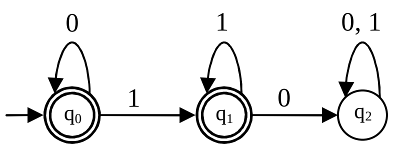


Figure: an FA M.

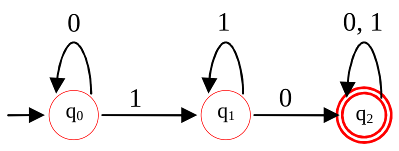


Figure: an FA $\overline{M}$

4.4 **Intersection of Two Regular Languages**

Let, M_1, M_2 be two DFAs. Then, $M_1 \cap M_2 = \overline{\overline{M_1} \cup \overline{M_2}}$.

Difference of Two Regular Languages

Let, M_1, M_2 be two DFAs. Then, $M_1 - M_2 = M_1 \cap \overline{M_2}$ is the set of strings that are in language M_1 but not in language M_2.

4.5 **Equivalence of States**

Two states q_1 and q_2 are equivalent (denoted by $q_1 \equiv q_2$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states, or both of them are non-final states for all $x \in \Sigma^*$.

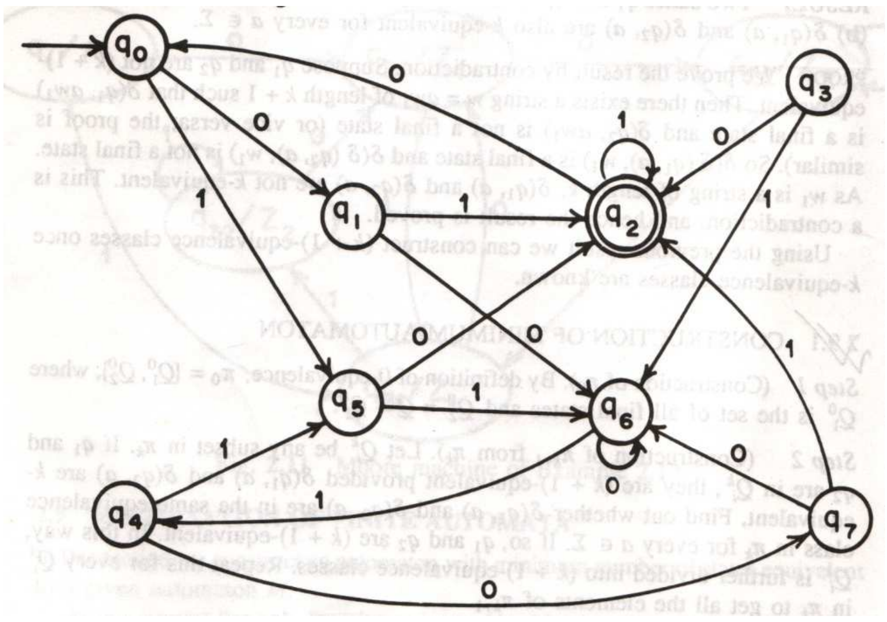
Minimization of DFA

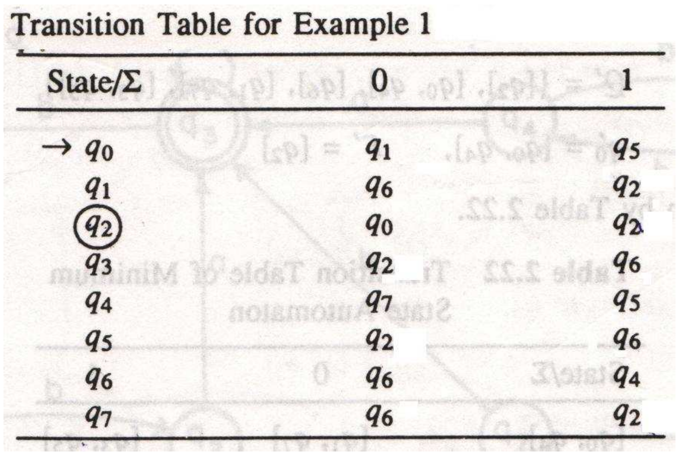
To minimize a DFA, the following rules are followed:

1. For each set of pair of states $\{p, q\}$, determine whether one of these states is final and the other is not. If so, p and q can *never* be collapsed without altering the language accepted. Mark this pair for "non-collapse"!
2. For each remaining *unmarked* pair $\{p, q\}$ and each symbol b in the alphabet, note $\{\delta(p, b), \delta(q, b)\}$. If $\delta(p, b)$ and $\delta(q, b)$ are distinct and the pair they form was *marked* in the *previous* round, then p and q can *never* be collapsed without altering the language recognized. Mark such pair for "non-collapse"!
3. Repeat step 2 until, when the step is completed, no new pairs has been *marked*.

   Note that for each pair $\{p, q\}$ remaining unmarked at this stage: for any string s of symbols of the alphabet, $\delta(p, b)$ and $\delta(q, b)$ (starting in states p and q, the string s is read) must be either both *final* states or both *non-final* states.
4. Collapse the states in each group into a single state. A group is final if it contains a final state of the original DFA.

Example





Transition Table for Example 1

State/Σ	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
$\boxed{q_2}$	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

$\pi_0 = \{\{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}\}$

$\pi_1 = \{\{q_2\}, \{q_0, q_4, q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

$\pi_2 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

∴ The minimum state automaton is

$$M' = (Q', \{0, 1\}, \delta', q_0', F')$$

where $Q' = \{[q_2], [q_0, q_4], [q_6], [q_1, q_7], [q_3, q_5]\}$,
$q_0' = [q_0, q_4]$,
$F' = [q_2]$

and δ' is given by the following table:

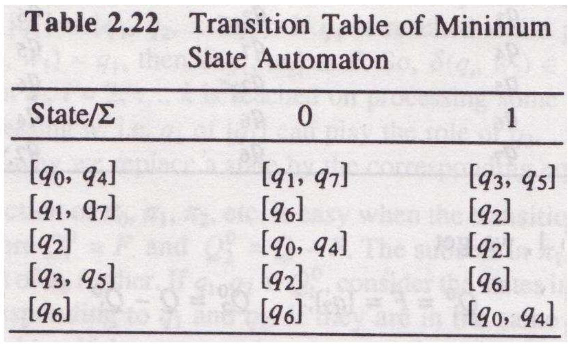


Table 2.22 Transition Table of Minimum State Automaton

State/Σ	0	1
$[q_0, q_4]$	$[q_1, q_7]$	$[q_3, q_5]$
$[q_1, q_7]$	$[q_6]$	$[q_2]$
$[q_2]$	$[q_0, q_4]$	$[q_2]$
$[q_3, q_5]$	$[q_2]$	$[q_6]$
$[q_6]$	$[q_6]$	$[q_0, q_4]$

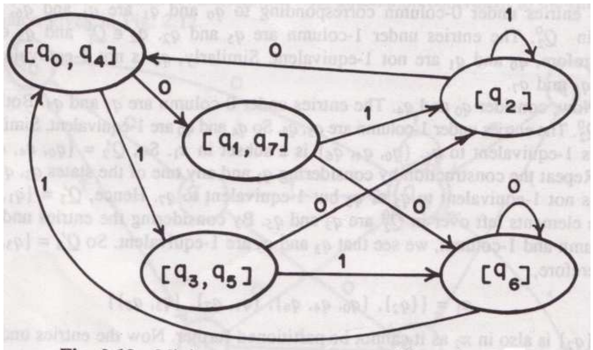


Fig. 2.13 Minimum state automaton of Example 1.

4.4 The Pumping Lemma for Regular Languages

Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

Prove that the language L consisting of all strings with an equal number of 0's and 1's (not in any particular order) is not a regular language.

Let, L is regular and n be the number of states in the FA accepting L.

Let, $w = 0^n1^n$. Then, $|w| = 2n > n$.

By pumping lemma, we can write $w = xyz$ such that $|xy| \leq n$ and $y \neq \epsilon$.

Since $|xy| \leq n$ and xy comes at the front of w, therefore, x and y consist of only 0's.

Consider the string xy^0x, i.e., xz.

Now, xz has n 1's, since all the 1's of w are in z.

However, xz has fewer than n 0's, because we lost the 0's of y and $y \neq \epsilon$.

$\therefore$ $xz \notin$ L.

This is a contradiction.

$\therefore$ L is not regular.

Show that L = {0^n | n is a perfect square} is not regular.

Let, L is regular and n be the number of states in the FA accepting L.

Let, $w = 0^{n^2}$. Then, $|w| = n^2 > n$.

By pumping lemma, we can write $w = xyz$ such that $|xy| \leq n$ and $y \neq \epsilon$.

Since $|xy| \leq n$, then y consists of between 1 and n 0's.

Consider the string xy^2z, i.e., $xyyz$.

$xyyz$ has length between $n^2 + 1$ (when y consists of 1 0) and $n^2 + n$ (when y consists of n 0's).

Since the next perfect square after n^2 is $(n + 1)^2 = n^2 + 2n + 1$, we know that the length of $xyyz$ lies strictly between the consecutive perfect squares n^2 and $(n + 1)^2$. Thus, the length of $xyyz$ cannot be a perfect square. But if the language were regular, then $xyyz$ would be in the language, which contradicts the assumption that the language of strings of 0's whose length is a perfect square is a regular language.

$\therefore$ L is not regular.

Prove that L = {ww^R | $w \in \{0, 1\}^*$} is not regular.

Let, L is regular and n be the number of states in the FA accepting L.

Let, $w = 0^n1^n1^n0^n$. Then, $|w| = 4n > n$.

By pumping lemma, we can write $w = xyz$ such that $|xy| \leq n$ and $y \neq \epsilon$.

Since $|xy| \leq n$ and xy comes at the front of w, therefore, x and y consist of only 0's.

Consider the string xy^0x, i.e., xz.

Now, xz has fewer than n 0's on the left, because we lost the 0's of y and $y \neq \epsilon$.

$\therefore$ xz has fewer 0's on the left than on the right and so cannot be of the form ww^R.

$\therefore$ $xz \notin$ L.

This is a contradiction.

$\therefore$ L is not regular.

Prove that L = $\{w \in \{1\}^* \mid |w|$ is a prime$\}$ is not regular.

Let, L is regular and n be the number of states in the FA accepting L.

Consider some prime $p \geq n + 2$; there must be such a p, since there are infinite primes.

Let, $w = 1^p$. Then, $|w| > n$.

By pumping lemma, we can write $w = xyz$ such that $|xy| \leq n$ and $y \neq \epsilon$.

Let, $|y| = m$. Then, $|xz| = p - m$.

Now consider the string $xy^{p-m}z$, which must be in L by the pumping lemma, if L really is regular.

However, $|xy^{p-m}z| = |xz| + (p - m)|y| = p - m + (p - m)m = (m + 1)(p - m)$

It looks like $|xy^{p-m}z|$ is not a prime, since it has two factors $m + 1$ and $p - m$. However, we must check that neither of these factors are 1, since then $(m + 1)(p - m)$ might be a prime after all.

But $m + 1 > 1$, since $y \neq \epsilon$ tells us that $m \geq 1$.

Also, $p - m > 1$, since $p \geq n + 2$ was chosen, and $m \leq n$ since $m = |y| \leq |xy| \leq n$.

$\therefore$ $p - m \geq 2$.

Therefore, we have derived a contradiction by showing that some string not in the language was required by the pumping lemma to be in the language.

Thus, we conclude that L is not a regular language.

Chapter 5 & 6
Context-Free Grammar & Push-Down Automata

Concepts

5.1

Grammar / Context-Free Grammar (CFG)

A CFG is a 4-tuple (T, V, S, P), where

T is a finite alphabet of *symbols*, called ***terminals*** or ***terminal symbols***, that form the strings of the language being defined;

V is a finite set of variables, also called ***nonterminals*** or ***nonterminal symbols*** or ***syntactic categories***, where each variable represents a language, i.e., a set of strings;

$S \in V$ (the ***start symbol***, which represents the language being defined);

P is a finite set of ***productions*** or ***grammar rules*** of the form $A \rightarrow \alpha$, where A $\in V$ and $\alpha \in (V \cup T)$; A is called the ***head*** of the production, and the string α is called the ***body*** of the production.

A CFG is usually denoted by G.

Example

Let's define a CFG that represents expressions in a typical programming language. First, we hall limit ourselves to the operators + and *, representing addition and multiplication. We shall allow only the letters a and b and the digits 0 and 1 as identifiers. Every identifier must begin with a or b, which may be followed by any string in $\{a, b, 0, 1\}^*$.

Then, the CFG is G = (T, V, S, P), where

$T = \{+, *, (,), a, b, 0, 1\}$
$V = \{E, I\}$
$S = E$
$P = \{E \rightarrow I \mid E + E \mid E * E \mid (E),$
$\quad I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1\}$

Context-Free Language (CFL) / The Language Generated by a CFG

Let $G = (T, V, S, P)$ be a CFG. The language generated by G is

$$L(G) = \{x \in T^* \mid S \Rightarrow_G^* x\}$$

A language L is a CFL if there is a CFG G so that $L = L(G)$.

Justification for the Name *Context-Free Grammar*

Considering the expression grammar above, it is easy to check that there is a derivation

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

As a result, for any strings α and β, it is also true that

$$\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$$

The justification is that we can make the same replacements of production bodies for heads in the context of α and β as we can in isolation. In fact, it is this property of being able to make a string-for-variable substitution regardless of context that gave rise originally to the term "context-free". There is a more powerful classes of grammars, called "context-sensitive", where replacements are permitted only if certain strings appear to the left and/or right. Context-sensitive grammars do not play a major role in practice today.

5.2 Conventional Notations for CFG Derivations

1. Lower-case letters near the beginning of the alphabet, *a*, *b* and so on, are terminal symbols. We shall also assume that digits and other characters such as + or parentheses are terminals.
2. Upper-case letters near the beginning of the alphabet, *A*, *B* and so on, are variables.]
3. Lower-case letters near the end of the alphabet, *w*, *x* and so on, are strings of terminals.
4. Upper-case letters near the end of the alphabet, X, Y and so on, are either terminals or variables.
5. Lower-case Greek letters, such as α and β, are strings consisting of terminals and/or variables.

5.3 Leftmost and Rightmost Derivations

In order to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost variable by one of its production bodies. Such a derivation is called a *leftmost* derivation, and is indicated using the relations $\underset{lm}{\Rightarrow}$ and $\overset{*}{\underset{lm}{\Rightarrow}}$, for one or many steps respectively.

Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies. If so, we call the derivation *rightmost* and use the symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$, for one or many rightmost derivation steps respectively.

Example of Leftmost and Rightmost Derivations

Given the grammar G = (T, V, S, P), where

$T = \{+, *, (,), a, b, 0, 1\}$
$V = \{E, I\}$
$S = E$
$P = \{E \rightarrow I \mid E + E \mid E * E \mid (E),$
$\quad I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1\}$

Find the leftmost and rightmost derivations of the string $a * (a + b00)$.

$$\begin{aligned}
E &\underset{lm}{\Rightarrow} E * E \\
&\underset{lm}{\Rightarrow} I * E \\
&\underset{lm}{\Rightarrow} a * E \\
&\underset{lm}{\Rightarrow} a * (E) \\
&\underset{lm}{\Rightarrow} a * (E + E) \\
&\underset{lm}{\Rightarrow} a * (I + E) \\
&\underset{lm}{\Rightarrow} a * (a + E) \\
&\underset{lm}{\Rightarrow} a * (a + I) \\
&\underset{lm}{\Rightarrow} a * (a + I0) \\
&\underset{lm}{\Rightarrow} a * (a + I00) \\
&\underset{lm}{\Rightarrow} a * (a + b00)
\end{aligned}$$

$$\begin{aligned}
E &\underset{rm}{\Rightarrow} E * E \\
&\underset{rm}{\Rightarrow} E * (E) \\
&\underset{rm}{\Rightarrow} E * (E + E) \\
&\underset{rm}{\Rightarrow} E * (E + I) \\
&\underset{rm}{\Rightarrow} E * (E + I0) \\
&\underset{rm}{\Rightarrow} E * (E + I00) \\
&\underset{rm}{\Rightarrow} E * (E + b00) \\
&\underset{rm}{\Rightarrow} E * (I + b00) \\
&\underset{rm}{\Rightarrow} E * (a + b00) \\
&\underset{rm}{\Rightarrow} I * (a + b00) \\
&\underset{rm}{\Rightarrow} a * (a + b00)
\end{aligned}$$

5.4 Parse Trees

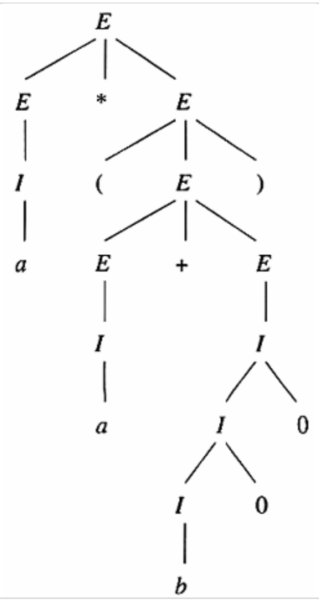
There is a tree representation for derivations that has proved extremely useful. This tree shows us clearly how the symbols of a terminal string are grouped into substrings, each of which belongs to the language of one of the variables of the grammar. But perhaps more importantly, the tree, known as a *parse tree* when used in compiler, is the data structure of choice to represent the source program. In a compiler, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

Yield of a Parse Tree

If we look at the leaves of any parse tree and concatenate them from left, we get a string, called the *yield* of the tree, which is always a string that is derived from the root variable.

Example

The figure above shows the parse tree for the string $a * (a + b00)$ which belongs to the CFG described in the example of 5.3.



5.5 Applications of CFG

1. Grammars are used to describe programming languages. More importantly, there is a mechanical way of turning the language description as a CFG into a parser, the component of the compiler that discovers the structure of the source program and represents that structure by a parse tree.
2. The development of XML (*Extensible Markup Language*) is widely predicted to facilitate electronic commerce by allowing participants to share conventions regarding the format of orders, product descriptions, and many other kinds of documents. An essential part of XML is the *Document Type Definition* (DTD), which is essentially a CFG that describes the allowable tags and the ways in which these tags may be nested.

5.6 Ambiguous CFG

A CFG G is ambiguous if there is at least one string in $L(G)$ having two or more distinct derivation (parse) trees (or, equivalently, two or more distinct leftmost or rightmost derivations).

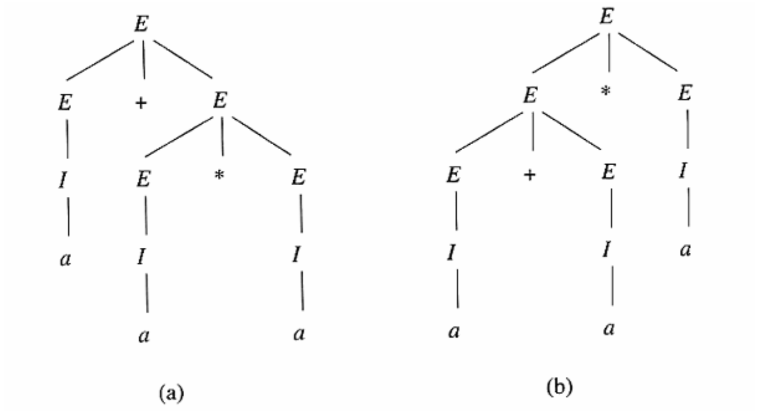


Figure: Trees with yield $a + a * a$, demonstrating the ambiguity of our expression grammar.

The Causes of Ambiguity in the Grammar Above

1. The precedence of operators is not respected.
2. A sequence of identical operators can group either from the left or from the right. For example, $E + E + E$ can be grouped by either $(E + E) + E$ or $E + (E + E)$.

Removing the Ambiguity in the Grammar Above

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of “binding strength”. Specifically:

1. A ***factor*** is an expression that cannot be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
   a. Identifiers.
   b. Any parenthesized expression, no matter what appears inside the parentheses.
2. A ***term*** is an expression that cannot be broken by the + operator. In our example where + and * are the only operators, a term is a product of one or more factors. For instance, the term $a * b$ cannot be broken apart by placing an additive term, such as a + to its left or + a to its right.
3. An ***expression*** will henceforth refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +. Thus, an expression for our example is a sum of one or more terms.

The following figures show an unambiguous grammar that generates the same language as the grammar above, and the sole parse tree for the expression $a + a * a$.

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

Figure 5.19: An unambiguous expression grammar

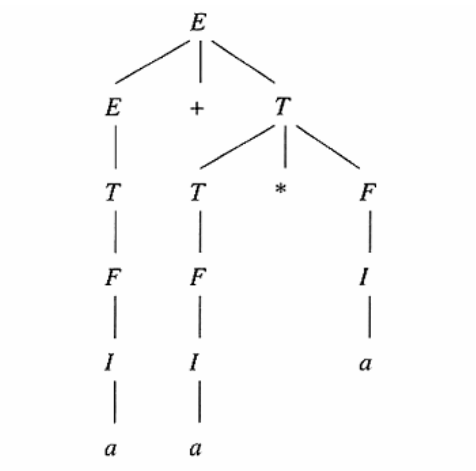


Figure 5.20: The sole parse tree for $a + a * a$

6.1

Push-Down Automata (PDA)

A PDA is a 7-tuple $(Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$, where

Q is a non-empty finite set of as *states*;

Σ and Γ are non-empty finite sets (the input and stack alphabets, respectively);

$q_0 \in Q$ (the *initial* state);

$Z_0 \in \Gamma$ (the *initial* stack symbol);

$F \subseteq Q$ (the set of *final* / *accepting* states);

$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow$ the set of finite subsets of $Q \times \Gamma^*$

A PDA is usually denoted by P.

Note that the PDA is essentially an ϵ-NFA with the addition of a stack.

The Transition Function of a PDA

δ takes as argument a triple $\delta(q, a, X)$, where:

1. $q \in Q$ (q is the state),
2. $a \in \Sigma \cup \epsilon$ (a is either an input symbol in Σ or $a = \epsilon$),
3. $X \in \Gamma$ (X is a stack symbol)

The output of δ is a finite set of pairs (p, γ), where p is the new state, and γ is the string of stack symbols that replaces X at the top of the stack. For example, if $\gamma = \epsilon$, then the stack is popped, if $\gamma = X$, then the stack is unchanged, and if $\gamma = YZ$, then X is replaced by Z, and Y is pushed onto the stack.

Example

Let's design a PDA that accepts the language $L_{ww^R} = \{ww^R \mid w \text{ is in } (0+1)^*\}$

The PDA for L_{ww^R} can be described as

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

Where δ is defined by the following rules:

$\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$
$\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$
…
$\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$

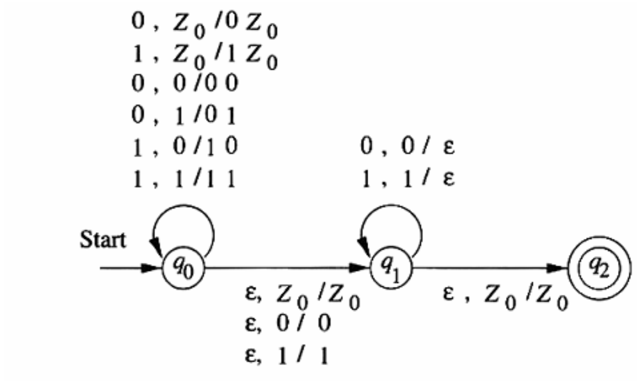


Figure 6.2: Representing a PDA as a generalized transition diagram

6.2 Language of a PDA

Acceptance by Final State

Let $P = (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$ be a PDA. Then L(P), the *language accepted by* P *by final state*, is

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \alpha)\}$$

for some state q in F and any stack string α. That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state. The content of the stack at that time is irrelevant.

Acceptance by Empty Stack

For each PDA $P = (Q, \Sigma, \Gamma, q_0, Z_0, F, \delta)$, we also define

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)\}$$

for any state q. That is, $N(P)$ is the set of inputs w that P can consume and at the same time empty its stack.

Note: The N in $N(P)$ stands for *null stack*, a synonym for *empty stack*.

CHAPTER 8
INTRODUCTION TO TURING MACHINES

Concepts

8.1 **Turing Machine (TM)**

The formal notation we shall use for a *Turing machine* (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple

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

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols*.

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function*. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of *final* or *accepting* states, a subset of Q.

The Language of a Turing Machine

Let $M = (Q, \Sigma, \Gamma, q_0, B, F, \delta)$ be a TM. Then L(M), the *language accepted by* M, is

$$\{w \in \Sigma^* \mid q_0 w \vdash^* \alpha p \beta\}$$

for some state p in F and any tape strings α and β.

The set of languages accepted by a TM is often called the *recursively enumerable* (*RE*) *languages*.

Example

Design a TM that accepts the language $\{0^n1^n \mid n \geq 1\}$.

The TM is

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$,

where δ is given by the following table:

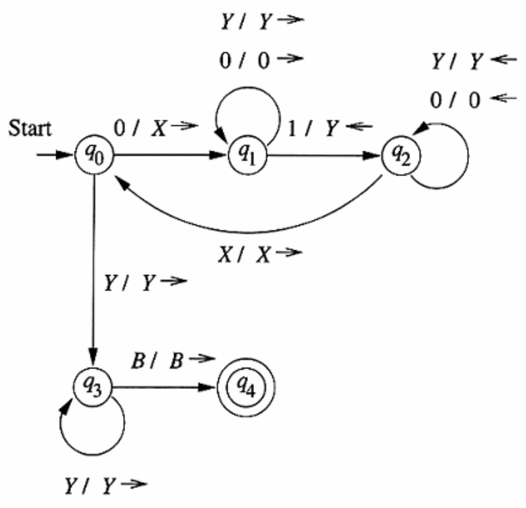
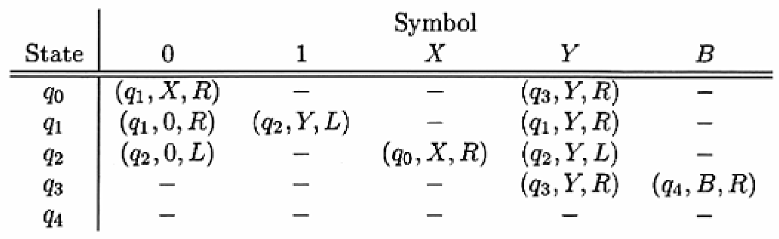


Figure 8.10: Transition diagram for a TM that accepts strings of the form 0^n1^n



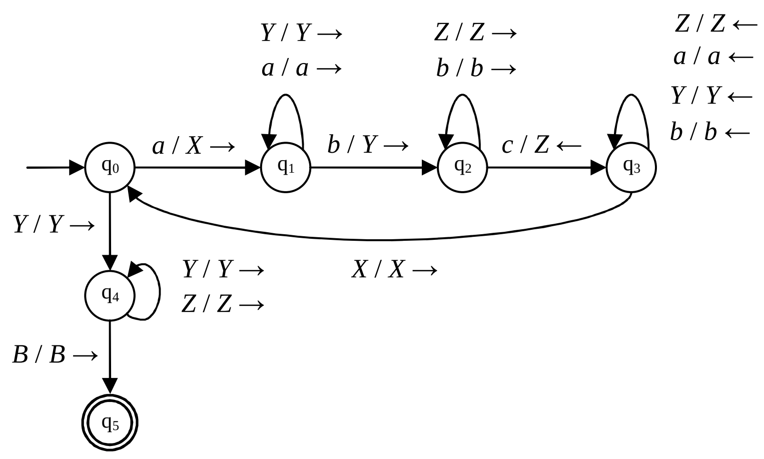
State	0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure 8.9: A Turing machine to accept $\{0^n1^n \mid n \geq 1\}$

8.2 Problem

Design a Turing Machine taking advantage of the programming techniques discussed in section 8.3 that accepts the language $a^nb^nc^n$ ($n \geq 1$).

General Design of the Turing Machine

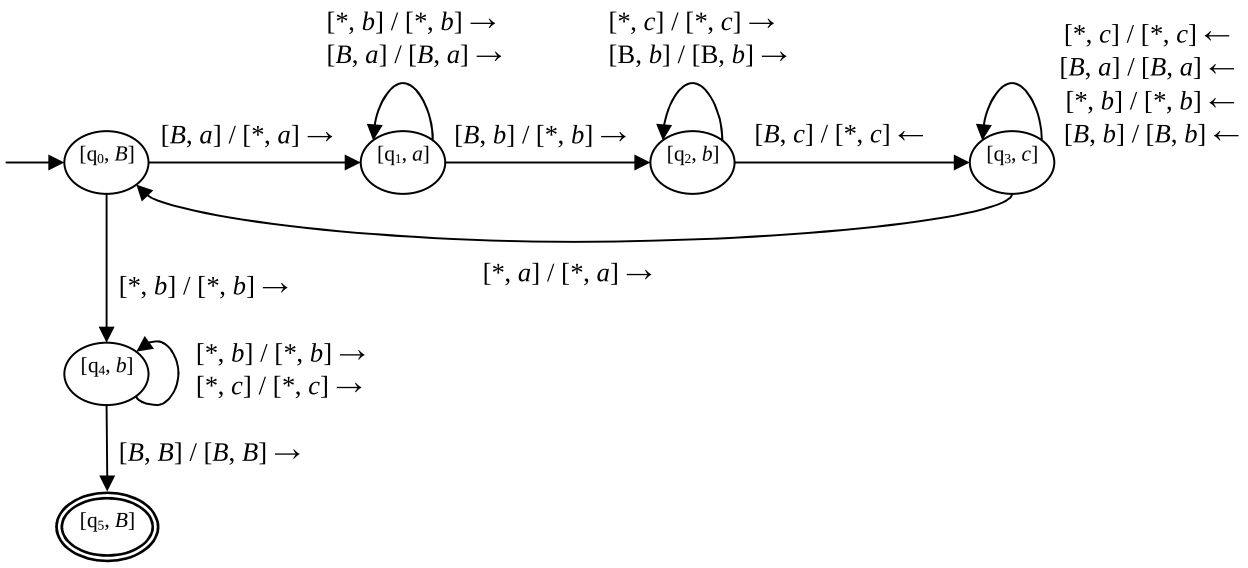


Design of the Turing Machine using *storage in the state* and *multiple tracks* techniques:

Let the Turing Machine be defined as M = (Q, Σ, Γ, δ, [q_0, *B*], [*B*, *B*], {[q_5, *B*]})

where	Q	= Set of States	= {q_0, q_1, … q_5} × {*a*, *b*, *c*, *B*}
	Σ	= Set of Input Symbols	= {*a*, *b*, *c*}
	Γ	= Set of Tape Symbols	= {*B*, *} × {*a*, *b*, *c*, *B*}
	δ	= Transition Function	
	[q_0, *B*]	= Initial State	
	[*B*, *B*]	= Blank Symbol	
	{[q_5, *B*]}	= Set of Final States	

Transition Diagram of M:



Transition Table of M:

States	**Input Symbols**						
	$[B, a]$	$[B, b]$	$[B, c]$	$[*, a]$	$[*, b]$	$[*, c]$	$[B, B]$
$\rightarrow[q_0, B]$	$([q_1,a],[*,a],R)$	—	—	—	$([q_4,B],[*,b],R)$	—	—
$[q_1, a]$	$([q_1,a],[B,a],R)$	$([q_2,b],[*,b],R)$	—	—	$([q_1,a],[*,b],R)$	—	—
$[q_2, b]$	—	$([q_2,b],[B,b],R)$	$([q_3,c],[*,c],L)$	—	—	$([q_2,b],[*,c],R)$	—
$[q_3, c]$	$([q_3,c],[B,a],L)$	$([q_3,c],[B,b],L)$	—	$([q_0,B],[*,a],L)$	$([q_3,c],[*,b],L)$	$([q_3,c],[*,c],L)$	—
$[q_4, b]$	—	—	—	—	$([q_4,B],[*,b],R)$	$([q_4,B],[*,c],R)$	$([q_5,B],[B,B],R)$
$[q_5, B]$	—	—	—	—	—	—	—

Design of the Turing Machine using *subroutine*:

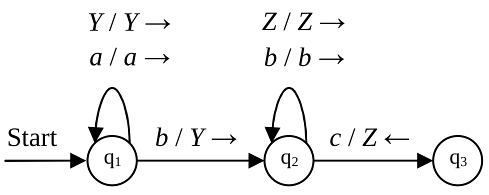


Figure 1: The subroutine *Check*.

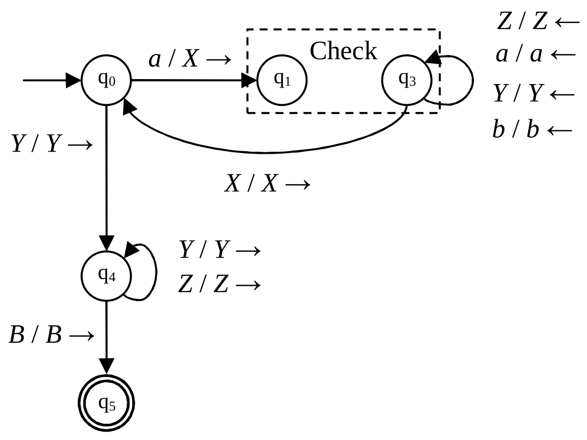


Figure 2: The complete program that accepts strings of the language $a^nb^nc^n$ ($n \geq 1$) using the subroutine *Check*.

Appendix A
Building Regular Expressions

(Solution to Exercises of Ullman and Martin)

Solved by Md. Abdul Kader

Martin: The language of all strings containing exactly two 0's.
Answer: 1*01*01*

Martin: The language of all strings containing at least two 0's.
Answer: (1+0)* 01*0 (1+0)*

Martin: The language of all strings that do not end with 01.
Answer: (1+0)*(00+10+11)+(1+0+e)

Martin: The language of all strings that begin or end with 00 or 11.
Answer: (00+11)(1+0)*+(0+1)*(00+11)

Martin: The language of all strings not containing the substrings 00.
Answer: (01+1)*(0+e)

Martin: The language of all strings in which the number of 0's is even.
Answer: (1*01*01*)*

Martin: The language of all strings containing no more than one occurrence of the string 00.
Answer: (01+1)*(00+0+e)(10+1)* //special case 0

Martin: The language of all strings in which every 0 is followed immediately by 11.
Answer: 1*(011)*1*

Martin: The language of all strings containing both 11 and 010 as substrings.
Answer: (1+0)*11(1+0)* 010(1+0)* + (1+0)*010(1+0)* 11(1+0)*

Ullman: The set of strings of 0's and 1's whose tenth symbol from the right end is 1.
Answer: (1+0)*1(1+0)

Ullman: The language of all strings with at most one pair of consecutive 1's.
Answer: (10+0)* (11+1+e) (01+0)*

Ullman: The language of all strings whose no of 0's is divisible by five.
Answer: (1*01*01*01*01*01*)*

Unknown: The language of all strings whose no of 0's are even and no of 1's are odd.
Answer: X = (1+010) //base case
Y = (1010+1001+0101+0110+00+11)* //recursive case
Final RE = YXY

Ullman: The set of strings over alphabet {a,b,c} containing at least one 'a' and at least one 'b'.
Answer: $X = (a+b+c)^*$
Final RE = $XaXbX + XbXaX$

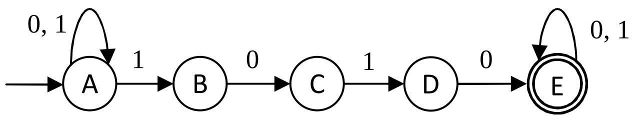
Ullman: The set of all strings of 0's and 1's such that every pair of adjacent 0's appears before any pair of adjacent 1's.
Answer: X = Does not contain 00 as a sub-set.
X = (01+1)*(0+e)
Final RE = (0+1)*11X + X

APPENDIX B

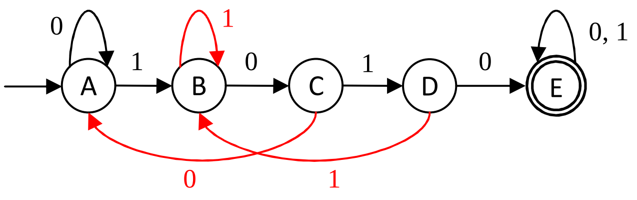
SOLUTION TO SELECTED QUESTIONS FROM THE IN-COURSE EXAMS OF PREVIOUS YEARS

1. **Draw the transition diagram of the DFA accepting the set of all strings over {0, 1} that contain 1010 as a substring.**

Step 1: (Draw the NFA)

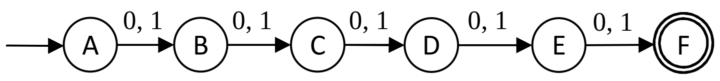


Step 2: (Make it DFA)

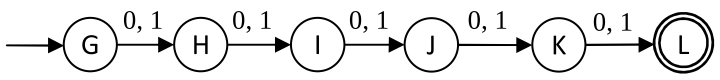


2. **Draw an FA accepting the language {*x*0*y* : *x*, *y* ∈ {0, 1}* and |*x*| + |*y*| = 5}.**

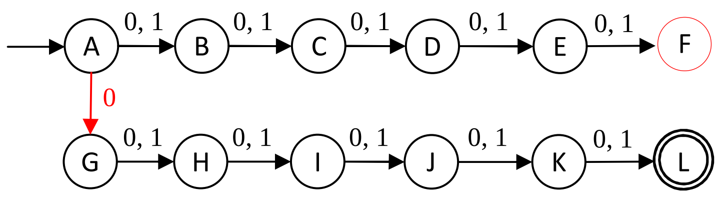
Step 1: (Draw the NFA accepting the language where |*x*| = 5)



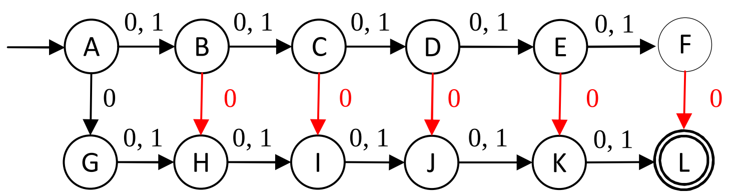
Step 2: (Draw the NFA accepting the language where |*y*| = 5)



Step 3: Now, as the string is like ***x*0*y***, therefore, we must start from the NFA of step 1 and finish at the NFA at step 2. However, we have to insert a *0*-transition from the DFA of step 1 to the DFA of step 2. Now, if |*x*| = 0, then |*y*| must be 5. So, we can insert a *0*-transition from state *A* to *G*.



Step 4: Similarly, if |*x*| = 1, then |*y*| must be 4. So, we can insert a *0*-transition from state *B* to *H*. And in similar manner, we get:



3. Simplify the RE as much as possible: $a^*b + a^*bbb^* + a^*$ and draw the equivalent FA.

$a^*b + a^*bbb^* + a^*$

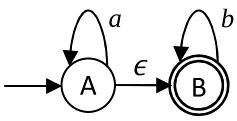
$= a^*(b + bbb^* + \epsilon)$

$= a^*(b\,(\epsilon + bb^*) + \epsilon)$

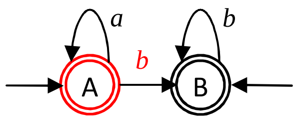
$= a^*(bb^* + \epsilon)$ [$\because \epsilon + xx^* = x^*$]

$= a^*b^*$ [$\because \epsilon + x^* = x^*$]

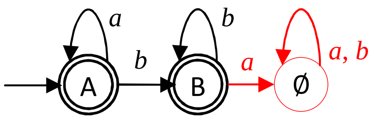
Equivalent ϵ-NFA:



Equivalent NFA:



Equivalent DFA:



4. Find a string that

(i) is represented by *both* the regular expressions *r* and *s* and
(ii) is *not* represented by either *r* or *s*

where $r = a^* + b^*$ and $s = ab^* + ba^* + b^*a + (a^*b)^*$

(i) *a*
(ii) *aba*

5. Write down an RE that represents all strings *w* over {0, 1} such that the first and last characters of *w* are the same and *w* contains at least three 0's.

Step 1: The string contains at least three 0's.

$$000$$

Step 2: There may be any number of 0's or 1's before and after each 0.

$$(0 + 1)^*\ 0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0\ (0 + 1)^*$$

Step 3: The first and the last characters must be the same. Let, the first and last characters are 1.

$$1\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 1$$

Step 4: Let, the first and last characters are 0.

$$0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0$$

Step 5: Combine the two using *union*.

$$1\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 1 + 0\ (0 + 1)^*\ 0\ (0 + 1)^*\ 0$$

6. **Given a DFA accepting a language *L*, how do we construct the DFA accepting the complement of *L*? Does your construction work for NFAs and NFAs with epsilon transitions? Justify your answer. [*Marks: 4*]**

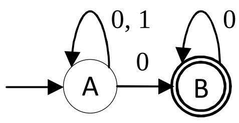
Given a language *L*, we can construct the complement DFA of *L* by making the *final* states of the DFA accepting *L* as *non-final* and vice-versa.

This construction does not work for NFAs or ϵ-NFAs. Because, in NFAs or ϵ-NFAs, there may be states which have multiple transitions. In that case, the final states cannot be identified properly and hence the construction will not work. But in DFAs, the final states are fixed and can be identified properly. So, this construction works for DFAs only.

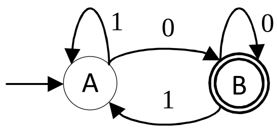
For example, let $L = \{w \mid w \text{ ends with } 0\}$.

Then, $L' = \{w \mid w$ **does not** end with $0\}$.

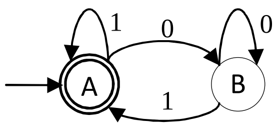
Now, the NFA for *L* is:



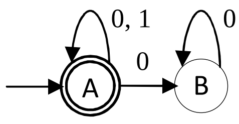
And the DFA for *L* is:



Therefore, the DFA for *L'* is:

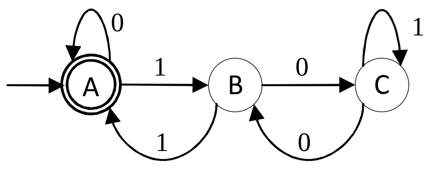


But the NFA for *L'* would be:



Which is not correct, as it also accepts the strings ending with 0.

7. **Construct a DFA accepting *binary* numbers that are divisible by 3.**



8. **Simplify the following RE: $(a + b)(\epsilon + aa)^+ + (a + b)$.**

$(a + b)(\epsilon + aa)^+ + (a + b)$

$= (a + b)\,((\epsilon + aa)^+ + \epsilon)$

$= (a + b)(\epsilon + aa)^*$ $[\because x^+ + \epsilon = x^*]$

$= (a + b)(\epsilon^*(aa)^*)^*$ $[\because (x + y)^* = (x^*y^*)^* = (x^* + y^*)^*]$

$= (a + b)((aa)^*)^*$

$= (a + b)(aa)^*$ $[\because (x^*)^* = x^*]$

9. Find an FA that accepts all strings over {a, b} that begin or end with aa or bb.

Step 1: RE that accepts all strings that ***begin*** with aa or bb:

$$(aa + bb)(a + b)^*$$

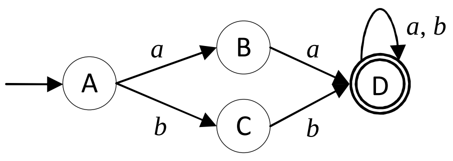
Step 2: RE that accepts all strings that ***end*** with aa or bb:

$$(a + b)^*(aa + bb)$$

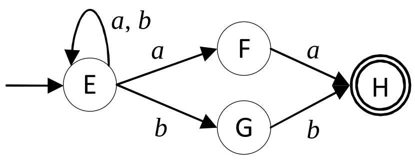
Step 3: RE that accepts all strings that ***begin or end*** with aa or bb:

$$(aa + bb)(a + b)^* + (a + b)^*(aa + bb)$$

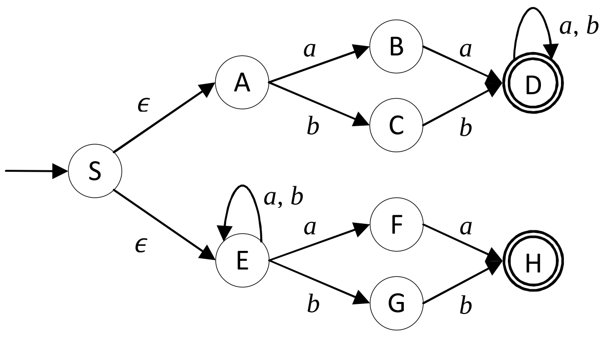
Step 4: NFA for $(aa + bb)(a + b)^*$:



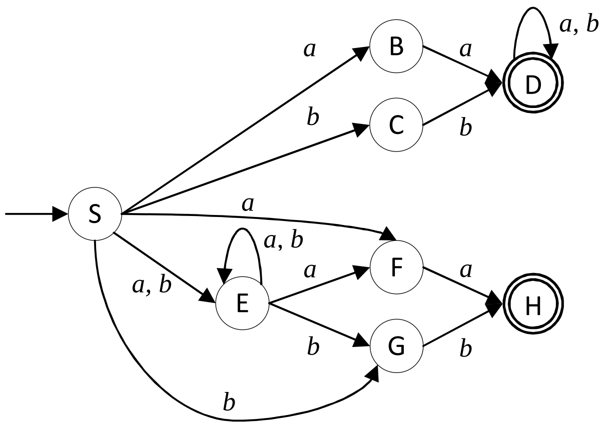
Step 5: NFA for $(a + b)^*(aa + bb)$:



Step 6: ϵ-NFA for $(aa + bb)(a + b)^* + (a + b)^*(aa + bb)$:

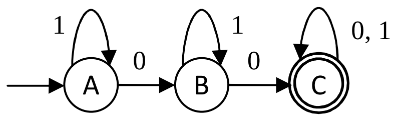


Equivalent NFA:

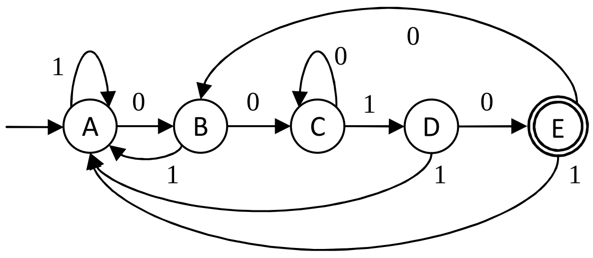


10. Draw the transition diagram of the DFA accepting the languages over {0, 1}
(i) The set of all strings that has at least two 0's.
(ii) The set of all strings that does not end in 0010.

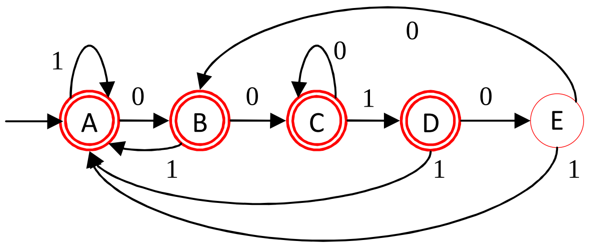
(i)



(ii) **Step 1:** DFA that accepts strings that *end* with 0010:



Step 2: DFA that accepts strings that ***does not end*** with 0010:



11. Write down a regular expression for the language L = {$w \in \{a, b\}^*$ | w does not contain an occurrence of ab}.

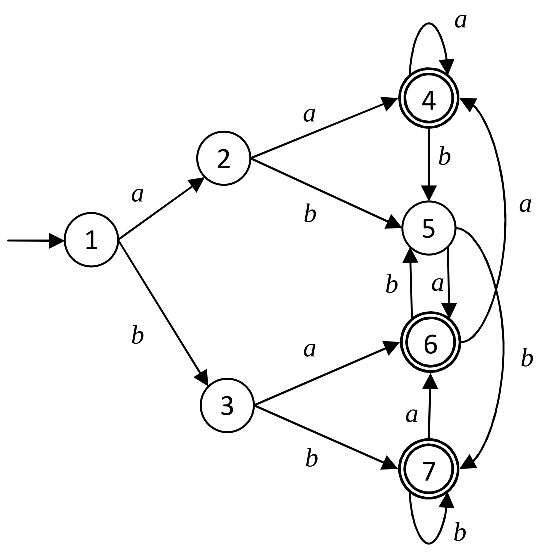
Step 1: w should not contain ab. So, the RE for w ***cannot*** be a^*b, ab^* or a^*b^*. However, the RE for w may be ϵ, a^*, b^*, or b^*a^*. Note that the RE b^*a^* includes the REs ϵ, a^* and b^*.

Step 2: Therefore, the final RE is b^*a^*.

12. Find and draw the minimum-state DFA for the DFA below:

Transition Table of the given DFA:

State	a	b
→ **1**	2	3
2	4	5
3	6	7
* **4**	4	5
5	6	7
* **6**	4	5
* **7**	6	7



Minimization of the given DFA:

Step 1: Separating final and non-final states, we get the following groups:

{1, 2, 3, 5} {4, 6, 7}

Step 2: From the group {4, 6, 7}, we separate the group {4, 6} as they move to the non-final group {1, 2, 3, 5} for the input symbol *b*.

{1, 2, 3, 5} {4, 6} {7}

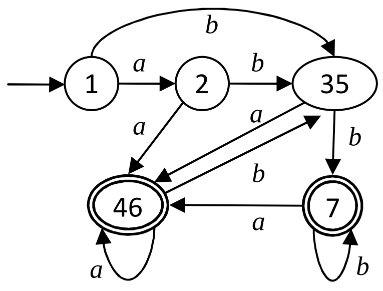
Step 3: From the group {1, 2, 3, 5}, we separate the group {3, 5} as they move to the final group {7} for the input symbol *b*.

{1, 2} {3, 5} {4, 6} {7}

Step 4: From the group {1, 2}, we separate the elements into two groups as for the symbol *a*, state 1 remains in the same group whereas state 2 moves to the group {4, 6}.

{1} {2} {3, 5} {4, 6} {7}

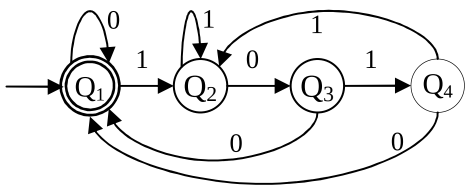
Diagram of the minimum-state DFA:



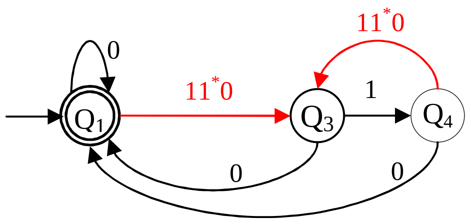
13. Find the regular expression equivalent to the DFA in the following figure:

States	Inputs	
	0	**1**
→* Q_1	Q_1	Q_2
Q_2	Q_3	Q_2
Q_3	Q_1	Q_4
Q_4	Q_1	Q_2

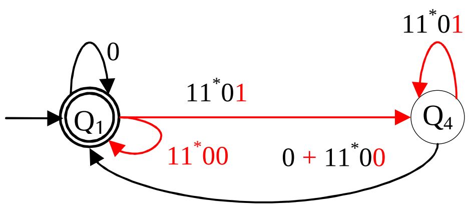
Step 1: Draw the transition diagram.

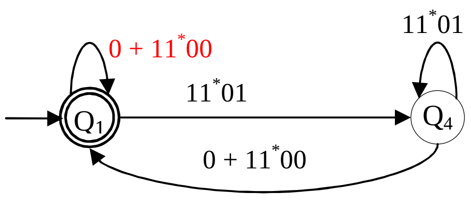


Step 2: Eliminate Q_2.

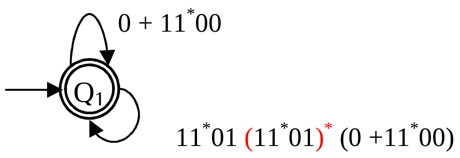


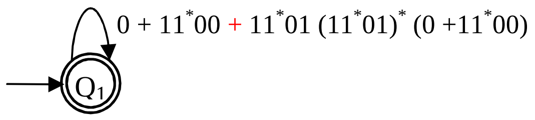
Step 3: Eliminate Q_3.



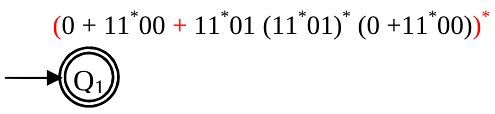


Step 4: Eliminate Q_4.





Step 5: Eliminate loop.



Final RE: $\mathbf{(0 + 11^*00 + 11^*01\ (11^*01)^*\ (0 + 11^*00))^*}$