Chapter 1

DFA and NFA

1.1 DETERMINISTIC FINITE AUTOMATA (DFA)

1.1.1 Automata—What is it?

An automaton is an abstract model of a digital computer. An automaton has a mechanism to read input, which is a string over a given alphabet. This input is actually written on an "input file", which can be read by the automaton but cannot change it.

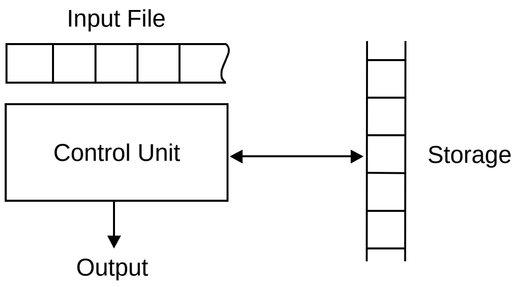


Fig. Automaton

Input file is divided into cells, each of which can hold one symbol. The automaton has a temporary "storage" device, which has unlimited number of cells, the contents of which can be altered by the automaton. Automaton has a control unit, which is said to be in one of a finite number of "internal states". The automaton can change state in a defined way.

1.1.2 Types of Automaton

(a) Deterministic Automata
(b) Non-deterministic Automata

A deterministic automata is one in which each move (transition from one state to another) is unequally determined by the current configuration.

If the internal state, input and contents of the storage are known, it is possible to predict the future behaviour of the automaton. This is said to be deterministic automata otherwise it is nondeterminist automata.

An automaton whose output response is “yes” or “No” is called an “Acceptor”.

1.1.3 Definition of Deterministic Finite Automaton

A Deterministic Finite Automator (DFA) is a 5-tuple

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

where

Q	=	Finite state of “internal states”
Σ	=	Finite set of symbols called “Input alphabet”
$\delta : Q \times \Sigma \to Q$	=	Transition Function
$q_0 \in Q$	=	Initial state
$F \subseteq Q$	=	Set of Final states

The input mechanism can move only from left to right and reads exactly one symbol on each step.

The transition from one internal state to another are governed by the transition function δ.

If $\delta(q_0, a) = q_1$, then if the DFA is in state q_0 and the current input symbol is a, the DFA will go into state q_1.

Example 1.1.1: Design a DFA, M which accepts the language $L(M) = \{w \in (a, b)^* : w$ does not contain three consecutive b’s).

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

where

Q = $\{q_0, q_1, q_2, q_3\}$

Σ = $\{a, b\}$

q_0 is the initial state

F = $\{q_0, q_1, q_2,\}$ are initial states

and δ is defined as follows:

Initial state q	Symbol σ	Final state $\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_0
q_1	b	q_2
q_2	a	q_0
q_2	b	q_3
q_3	a	q_3
q_3	b	q_3

Solution

M does not accept specified language, as long as three consecutive b's have not been read.

It should be noted that

(i) M is in state q_i (where $i = 0,1$, or 2) immediately after reading a run of i consecutive b's that either began the input string or was preceded by an 'a'.

(ii) If an 'a' is read and M is in state, q_0, q_1, or M returns to its initial state q_0.

q_0, q_1 and q_2 are "Final states" (as given in the problem). Therefore any input string not containing three consecutive b's will be accepted.

In case we get three consecutive b's then the q_3 state is reached (which is not final state), hence M will remain in this state, irrespective of any other symbol in the rest of the string. This state q_3 is said to be "dead state" or M is said to be "trapped" at q_3.

The DFA schematic is shown below based on the discussion above.

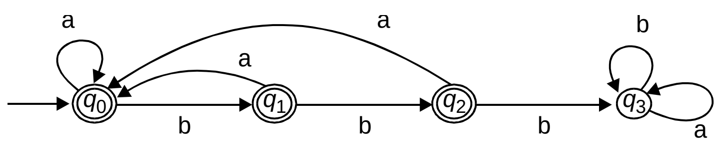


Fig. Finite Automaton with four states

Example 1.1.2: Determine the DFA schematic for $M = (Q, \Sigma, \delta, q, F)$ where $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$, q_1 is the start state, $F = \{q_2\}$ and δ is given by the table below.

Initial state q	Symbol σ	Final state $\delta(q,\sigma)$
q_1	0	q_1
q_1	1	q_2
q_2	0	q_3
q_2	1	q_2
q_3	0	q_2
q_3	1	q_2

Also determine a Language L recognized by the DFA.

Solution

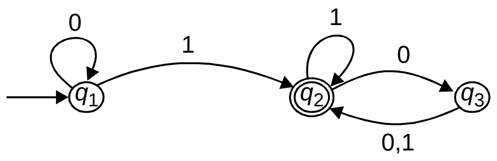


Fig. Finite Automaton having three states.

From the given table for δ, the DFA is drawn, where q_2 is the only final state.

(It is to be noted that a DFA can "accept" a string and it can "recognize" a language. Catch here is that "accept" is used for strings and "recognize" for that of a language).

It could be seen that the DFA accepts strings that has at least one 1 and an even number of 0s following the last 1.

Hence the language L is given by

$$\boxed{L = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}}$$

where $L = L(M)$ and M recognized the RHS of the equation above.

Example 1.1.3: Sketch the DFA given

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

and δ is given by

$$\delta(q_1, 0) = q_1 \text{ and } \delta(q_2, 0) = q_1$$
$$\delta(q_1, 1) = q_2 \qquad \delta(q_2, 1) = q_2$$

Determine a Language $L(M)$, that the DFA recognizes.

Solution

From the given data, it is easy to predict the schematic of DFA as follows.

Internal states = q_1, q_2.
Symbols = 0, 1.
Transition function = δ (as defined above in the given problem)
q_1 = Initial state
q_2 = Final state.

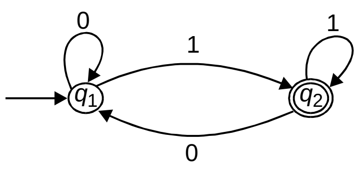


Fig. State diagram of DFA

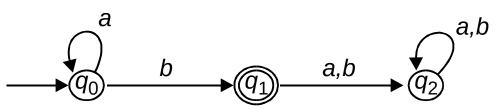
If a string ends in a 0, it is "rejected" and "accepted" only if the string ends in a 1. Therefore the language

$$L(M) = \{w \mid w \text{ ends in a } 1\}.$$

Example 1.1.4: Design a DFA, the language recognized by the Automaton being

$$L = \{a^n b : n \geq 0\}$$

Solution



For the given language $L = \{a^n b : n \geq 0\}$, the strings could be $b, ab, a^2b, a^3b, \ldots$.

Therefore the DFA accepts all strings consisting of an arbitrary number of a's, followed by a single *b*. All other input strings are rejected.

Example 1.1.5: Obtain the state table diagram and state transistion diagram (DFA Schematic) of the finite state Automaton $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, q_0 is the initial state, F is the final state with the transistion defined by

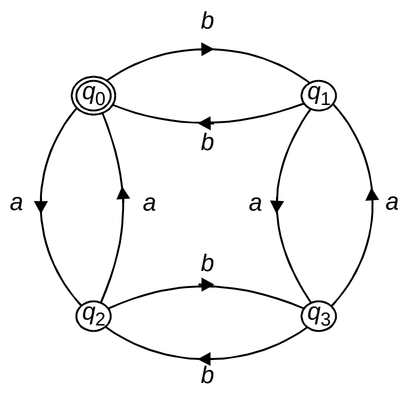
$$\begin{array}{lll} \delta(q_0, a) = q_2 & \delta(q_3, a) = q_1 & \delta(q_2, b) = q_3 \\ \delta(q_1, a) = q_3 & \delta(q_0, b) = q_1 & \delta(q_3, b) = q_2 \\ \delta(q_2, a) = q_0 & \delta(q_1, b) = q_0 & \end{array}$$

Solution

The State Table diagram is as shown below

δ	a	b
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

With the given definitions, the State Transition diagram/DFA Schematic is shown on next page.



Example 1.1.6: Obtain the DFA that accepts/recognizes the language

$L(M) = \{w \mid w \in \{a, b, c\}^*$ and w contains the pattern $abac\}$

(*Note:* This is an application of DFA's involving searching a text for a specified pattern)

Solution

Let us begin by "hard coding" the pattern into the machines states as shown in fig. (a) below.

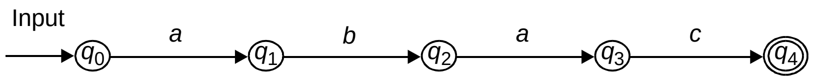


Fig. (a)

As the pattern '*abac*' has length four, there are four states required in addition to one intial state q_0, to remember the pattern. q_4 is the only accepting state required and this state q_4 can be reached only after reading '*abac*'.

The complete DFA is as shown below in Fig. (b).

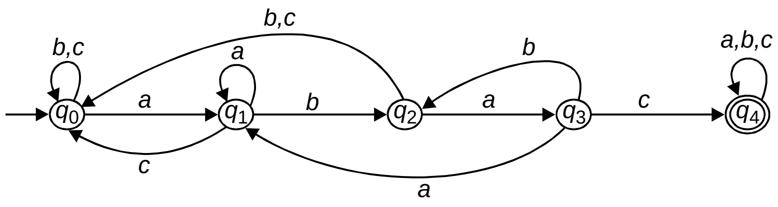
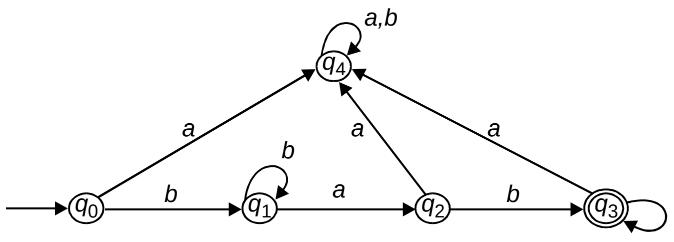


Fig. (b)

Example 1.1.7: Given $\Sigma = \{a, b\}$, construct a DFA that shall recognize the language $L = \{b^m ab^n : m, n > 0\}$.

Solution

The given language $L = \{b^m ab^n : m, n > 0\}$ has all words with exactly one 'a' which is neither the first nor last letter of the word i.e., there is one or more b's before or after 'a'.



DFA is drawn above for the automaton M,

where $M = (Q, \Sigma, \delta, q_0, F)$ with
$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$\Sigma = \{a, b\}$; q_0 = Initial state,
$F = \{q_3\}$ = Final state.

and δ is defined as per the language L. (q_4 is "dead" state)

Example 1.1.8: Given $\Sigma = \{a, b\}$, construct a DFA which recognize the language $L = \{a^m b^n : m, n > 0\}$.

Solution

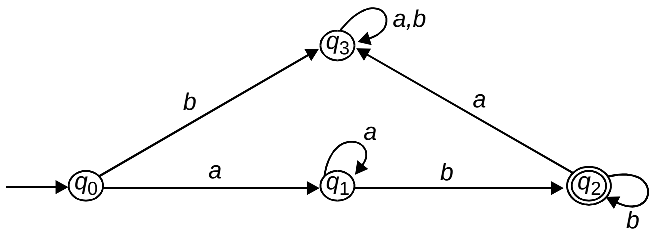
The given language $L = \{a^m b^n : m, n > 0\}$ has all words which begin with one or more a's followed by one or more b's.

The finite automaton $M(Q, \Sigma, \delta, q_0, F)$ is with

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3\} \\ \Sigma &= \{a, b\} \\ q_0 &= \text{Initial state} \\ F = \{q_2\} &= \text{Final state} \end{aligned}$$

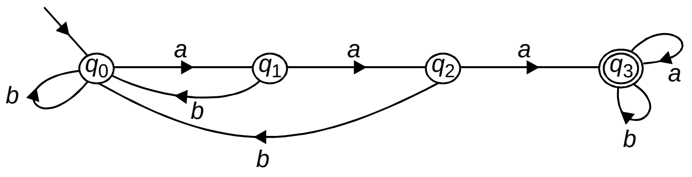
and δ as defined by language L.

The DFA is as shown below.

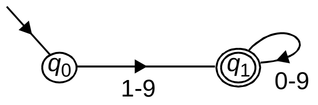


Here q_3 is a "dead" state.

(c) Set of strings having '*aaa*' as a subword.

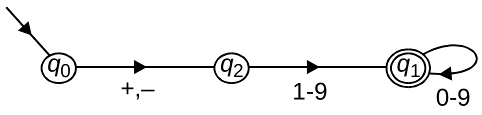


(d) Set of integers.



Alphabet $\Sigma = \{0,1, \ldots, 9\}$

(e) Set of signed Integers.



1.2 NON-DETERMINISTIC FINITE AUTOMATA (NFA)

Definition

A Nondeterministic Finite Automata (NFA) is defined by a 5-tuple

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

where $Q, \Sigma, \delta, q_0, F$ are defined as follows:

Q = Finite set of internal states
Σ = Finite set of symbols called "Input alphabet"
$\delta = Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$
$q_0 \in Q$ is the Initial states
$F \subseteq Q$ is a set of Final states

NFA differs from DFA in that, the range of δ in NFA is in the powerset 2^Q.

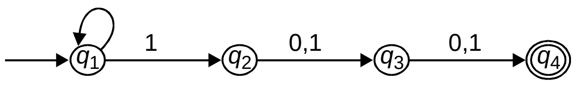
A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in the final state at the end of the string.

Example 1.2.1: Obtain an NFA for a language consisting of all strings over {0,1} containing a 1 in the third position from the end.

Solution

q_1, q_2, q_3 are initial states

q_4 is the final state.



Please note that this is an NFA as $\delta(q_2,0) = q_3$ and $\delta(q_2,1) = q_3$.

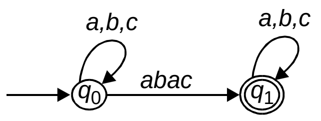
Example 1.2.2: Determine an NFA accepting the language

(a) $L_1 = \{x \mid x \in \{a, b, c\}^*$ and x contains the pattern $abac\}$

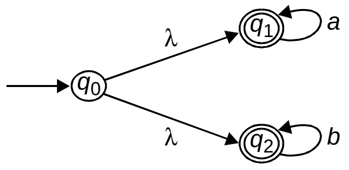
(b) $L_2 = \{a^* \cup b^*\}$

Solution

(a)



(b)



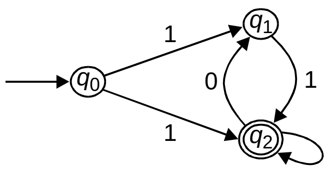
Example 1.2.3: Determine an NFA accepting all strings over {0,1} which end in 1 but does not contain the substring 00.

Solution

The conditions to be satisfied are:

(a) String should end in a 1

(b) String should not contain 00.



The NFA is shown in figure.

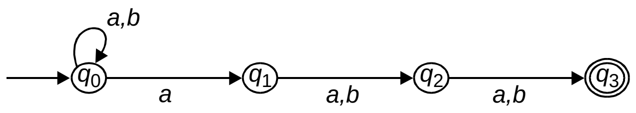
Example 1.2.4: Obtain an NFA which should accept a language L_A, given by $L_A = \{x \in \{a, b\}^* : |x| \geq 3$ and third symbol of x from the right is $\{`a'\}$.

Solution

The conditions are

(a) the last two symbols can be '*a*' or '*b*'.
(b) third symbol from the right is '*a*'
(c) symbol in any position but for the last three position can be '*a*' or '*b*'.

The NFA is shown in fig. below.



Example 1.2.5: Sketch the NFA state diagram for

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

with the state table as given below.

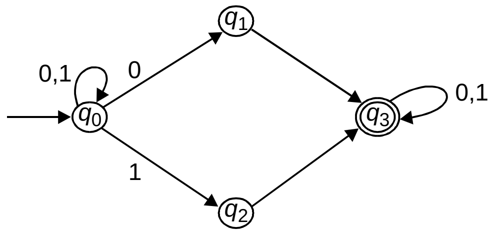
δ	0	1
q_0	q_0, q_1	q_0, q_2
q_1	q_3	$\varnothing$
q_2	$\varnothing$	q_3
q_3	q_3	q_3

Solution

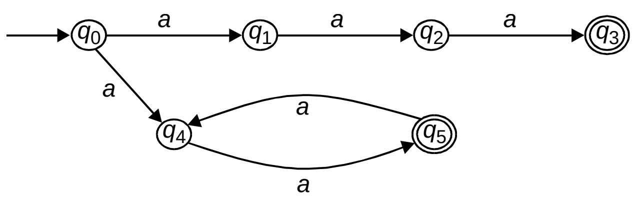
The NFA states are q_0, q_1, q_2 and q_3.

$$\delta(q_0,0) = \{q_0, q_1\} \qquad \delta(q_0,1) = \{q_0, q_2\}$$
$$\delta(q_1,0) = \{q_3\} \qquad \delta(q_2,1) = \{q_3\}$$
$$\delta(q_3,0) = \{q_3\} \qquad \delta(q_3,1) = \{q_3\}\ .$$

The NFA is as shown below.



Example 1.2.6: Given L is the language accepted by NFA in Fig. Determine an NFA that accepts $L \cup \{a^5\}$.



Solution

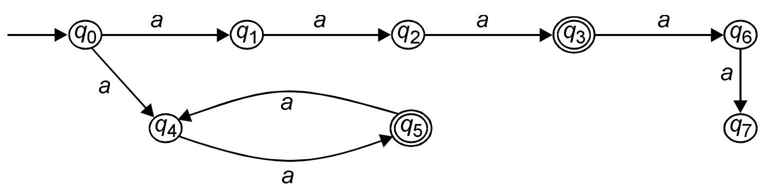
The language accepted by the given NFA is

$$L = \{a^3\} \cup \{a^n : n \text{ is odd}\}.$$

Now to make an NFA accepting the language:

$$L = \{a^3\} \cup \{a^n : n \text{ is odd}\} \cup \{\mathrm{a}^5\}.$$

This is accomplished by adding two states after state q_3 viz., q_6 and q_7 as shown in fig.



The NFA is given by

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}, \{a\}, \delta, q_0, \{q_3, q_5, q_7\})$$

Example 1.2.7: Find an NFA with four states for

$$L = \{a^n : n \geq 0\} \cup \{b^n a : n \geq 1\}$$

Solution

NFA for the language:

$$L = \{a^n : n \geq 0\} \cup \{b^n a : n \geq 1\}$$

For such a language two cases are to be considered.

Case (i): $a^n, n \geq 0$

q_0 goes to a state q_3 where all a's are absorbed. Hence a^n is accepted.

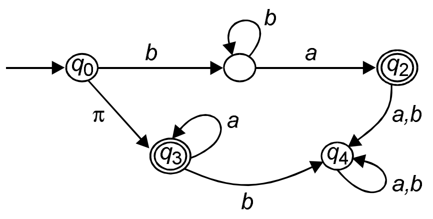
Case (ii): $b^n a : n \geq 1$

q_0 goes to a state q_1 where all b's are accepted and when an 'a' is encountered it goes to final state q_2. An additional state q_4 is added as a rejection state for the cases when 'b' is encountered after a's of case (i) or when 'a' or 'b' is encountered after $b^n a$ of case (ii).

The NFA is given by

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_2, q_3\})$$

which is shown in the fig. below.



Example 1.2.8: Design an NFA with no more than five states for the set

$$\{abab^n : n \geq 0\} \cup \{aba^n : n \geq 0\}.$$

Solution

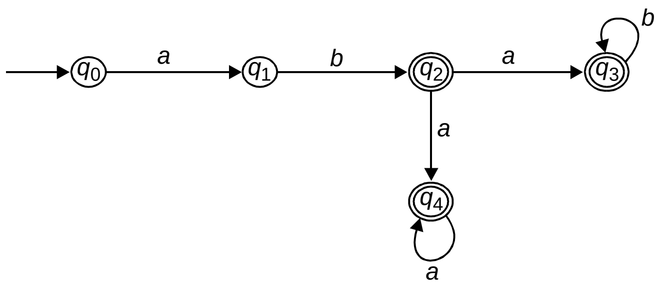
NFA for the language

$$L = \{abab^n : n \geq 0\} \cup \{aba^n : n \geq 0\}$$

is

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_2, q_3, q_4\}).$$

Here the NFA is such that it accepts all strings of the type aba^n and $abab^n$ where $n \geq 0$.

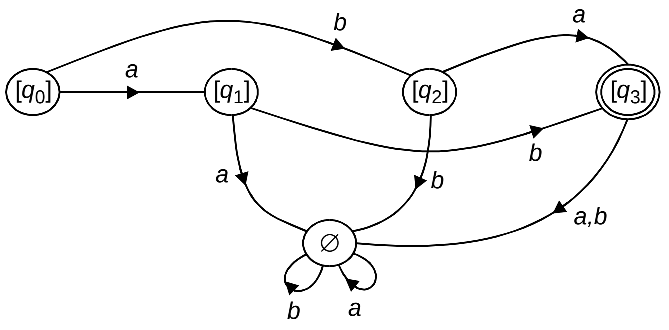


q_2 is for the case when string is ab, i.e. ab^n with $n = 0$.
q_3 is for the case when string is $abab^n$ with $n \geq 0$.
q_4 is for the case when string is aba^n with $n \geq 0$

This NFA is shown in the fig. above.

The state table corresponding to the DFA is derived by using subset construction. State table for DFA is as shown below.

	a	b
$[q_0]$	$[q_1]$	$[q_2]$
$[q_1]$	$\varnothing$	$[q_3]$
$[q_2]$	$[q_3]$	$\varnothing$
$[q_3]$	$\varnothing$	$\varnothing$
$\varnothing$	$\varnothing$	$\varnothing$



The DFA is as shown above.

1.4 REGULAR EXPRESSION

1.4.1 Regular Languages

The regular languages are those languages that can be constructed from the "big three" set operations viz., (a) Union (b) Concatenation (c) Kleene star.

A regular language is defined as follows.

Definition: Let Σ be an alphabet. The class of "regular languages" over Σ is defined inductively as follows:

(a) $\varnothing$ is a regular language
(b) For each $\sigma \in \Sigma$, $\{\sigma\}$ is a regular language
(c) For any natural number $n \geq 2$ if $L_1, L_2, \ldots\ldots L_n$ are regular languages, then so is $L_1 \cup L_2 \cup \ldots \ \ldots \cup L_{n.}$
(d) For any natural number $n \geq 2$, if $L_1, L_2, \ldots\ldots L_n$ are regular languages, then so is $L_1 \circ L_2 \circ \ldots \ \ldots \circ L_n$.
(e) If L is a regular language, then so is L^*.
(f) Nothing else is a regular language unless its construction follows from rules (a) to (e).

Examples:

(i) $\varnothing$ is a regular language (by rule (a))

(ii) $L = \{a, ab\}$ is a language over $\Sigma = \{a, b\}$ because, both $\{a\}$ and $\{b\}$ are regular languages by rule (b). By rule (d) it follows that $\{a\} \circ \{b\} = \{ab\}$ is a regular language. Using rule (c), we see that $\{a\} \cup \{ab\} = L$ is a regular language.

(iii) The language over the alphabet {0,1} where strings contain an even number of 0's can be constructed by

$$(1^*((01^*)(01^*))^*)$$

or simply $1^*(01^*\, 01^*)^*$.

1.4.2 Regular Expressions

Regular expressions were designed to represent regular languages with a mathematical tool, a tool built from a set of primitives and operations.

This representation involves a combination of strings of symbols from some alphabet Σ, parantheses and the operators +, $\cdot$, and *.

A regular expression is obtained from the symbol $\{a, b, c\}$, empty string $\in$, and empty-set $\varnothing$ perform the operations +, $\cdot$ and * (union, concatenation and Kleene star).

Examples

0 + 1 represents the set {0, 1}

1 represents the set {1}

0 represents the set {0}

(0 + 1) 1 represents the set {01, 11}

$(a + b) \cdot (b + c)$ represents the set $\{ab, bb, ac, bc\}$

$$(0 + 1)^* = \in + (0 + 1) + (0 + 1)\,(0 + 1) + \cdots\cdots = \Sigma^*$$

$$(0+1)^+ = (0+1)\,(0+1)^* = \Sigma^+ = \Sigma^* - \{\varepsilon\}$$

1.4.3 Building Regular Expressions

Assume that $\Sigma = \{a, b, c\}$

Zero or more: a^* means "zero or more *a*'s",

To say "zero or more *ab*'s," i.e., $\{\lambda, ab, abab, \ldots\ldots\}$ you need to say (*ab*)*.

One or more: Since a^* means "zero or more *a's*", you can use aa^* (or equivalently a^*a) to mean "one or more *a*'s". Similarly to describe 'one or more *ab*'s", that is {*ab, abab, ababab,* ……}, you can use *ab* (*ab*)*.

Zero or one: It can be described as an optional '*a*' with ($a + \lambda$).

Any string at all: To describe any string at all (with $\Sigma = \{a, b, c\}$ you can use $(a + b + c)^*$.

Any nonempty string: This is written any character from $\Sigma = \{a, b, c\}$ followed by any string at all: $(a + b + c)(a + b + c)^*$

Any string not containing ....: To describe any string at all that does not contain an 'a' (with $\Sigma = \{a, b, c\}$), you can use $(b + c)^*$.

Any string containing exactly one ....: To describe any string that contains exactly one 'a' put "any string not containing an a", on either side of the 'a' like: $(b + c)^* a(b + c)^*$.

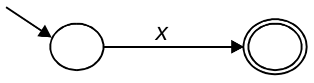
1.4.4 Languages defined by Regular Expressions

There is a very simple correspondence between regular expressions and the languages they denote:

Regular expression	L (Regular Expression)
x, for each $x \in \Sigma$	$\{x\}$
λ	$\{\lambda\}$
$\varnothing$	$\{\ \}$
(r_1)	$L(r_1)$
r_1^*	$(L(r_1))^*$
$r_1 r_2$	$L(r_1)L(r_2)$
$r_1 + r_2$	$L(r_1) \cup L(r_2)$

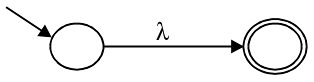
1.4.5 Regular Expressions to NFA

(i) For any x in Σ, the regular expression denotes the language $\{x\}$. The NFA (with a single start state and a single final state) as shown below, represents exactly that language.



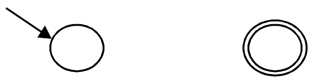
NFA for x

(ii) The regular expression λ denotes the language $\{\lambda\}$that is the language containing only the empty string.



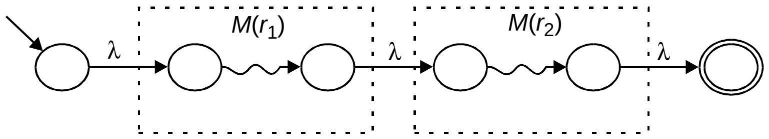
NFA for λ

(iii) The regular expression ∅ denotes the language ∅; no strings belong to this language, not even the empty string.



NFA for ∅

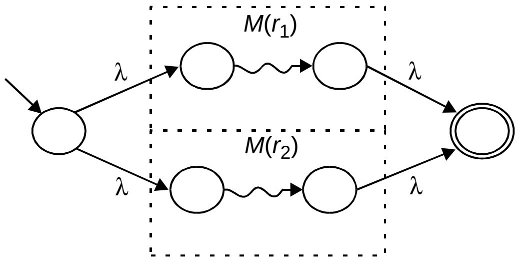
(iv) For juxtaposition, strings in $L(r_1)$ followed by strings in $L(r_2)$, we



NFA for r_1r_2

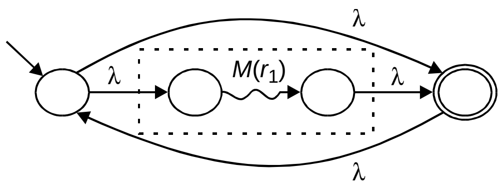
chain the NFAs together as shown.

(v) The "+" denotes "or" in a regular expression, we would use an NFA with a choice of paths.



NFA for $r_1 + r_2$

(vi) The star (*) denotes zero or more applications of the regular expression, hence a loop has to be set up in the NFA.



1.4.6 NFAs to Regular Expression

The basic approach to convert NFA, to Regular Expressions is as follows:

(i) If an NFA has more than one final state, convert it to an NFA with only one final state. Make the original final states nonfinal, and add a λ-transition from each to the new (single) final state.

(ii) Consider the NFA to be a generalised transition graph, which is just like an NFA except that the edges may be labeled with arbitrary regular expressions. Since the labels on the edges of an NFA may be either λ or members of each of these can be considered to be a regular expression.

(iii) Removes states one by one from the NFA, relabeling edge as you go, until only the initial and the final state remain.

(iv) Read the final regular expression from the two state automaton that results.

The regular expression derived in the final step accepts the same language as the original NFA.

Example 1.4.1: Represent the following sets by regular expression

(a) $\{\wedge, ab\}$

(b) $\{1, 11, 111, \ldots\ldots\}$

(c) $\{ab, a, b, bb\}$

Solution

(a) The set $\{\wedge, ab\}$ is represented by the regular expression $\wedge + ab$

(b) The set $\{1, 11, 111, \ldots\ldots\}$ is got by concatenating 1 and any element of $\{1\}^*$. Therefore $1(1)^*$ represent the given set.

(c) The set $\{ab, a, b, bb\}$ represents the regular expression $ab + a + b + bb$.

Example 1.4.2: Obtain the regular expressions for the following sets:

(a) The set of all strings over $\{a, b\}$ beginning and ending with 'a'.

(b) $\{b^2, b^5, b^8, \ldots\ldots\}$

(c) $\{a^{2n+1} \mid n > 0\}$

Solution

(a) The regular expression for 'the set of all strings over $\{a, b\}$ beginning and ending with 'a' is given by:

$$a\,(a + b)^* a$$

(b) The regular expression for $\{b^2, b^5, b^8, \ldots\ldots\}$ is given by:

$$bb\,(bbb)^*$$

(c) The regular expression for $\{a^{2n+1} \mid n > 0\}$ is given by:

$$a\,(aa)^*$$

Example 1.4.3: Obtain the regular expressions for the languages given by

(a) $L_1 = \{a^{2n} b^{2m+1} \mid n \geq 0, m \geq 0\}$
(b) $L_2 = \{a, bb, aa, abb, ba, bbb, \ldots\ldots\}$
(c) $L_3 = \{w \in \{0,1\}^* \mid w$ has no pair of consecutive zeros$\}$
(d) $L_4 = \{$strings of 0's and 1's ending in 00$\}$

Solution

(a) $L_1 = \{a^{2n} b^{2m+1} \mid n \geq 0, m \geq 0\}$ denotes the regular expression

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

(b) The regular expression for the language $L_2 = \{a, bb, aa, abb, ba, bbb, \ldots\ldots\}$

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

(c) The regular expression for the language $L_3 = \{w \in \{0,1\}^* \mid w$ has no pair of consecutive zeros$\}$ is given by

$$(1^* 011^*)^* (0 + \lambda) + 1^* (0 + \lambda)$$

(d) The regular expression for the language $L_4 = \{$strings of 0's and 1's beginning with 0 and ending with 1$\}$ is given by

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

Example 1.4.4: Describe the set represented by the regular expression $(aa + b)^* (bb + a)^*$

Solution

The given regular expression is

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

The English language description is as follows: "The set of all the strings of the form *uv* where *a*'s are in pairs in *u* and *b*'s are in pairs in *v*".

Example 1.4.5: Give Regular expressions for the following on $\Sigma = \{a, b, c\}$

(a) all strings containing exactly one *a*
(b) all strings containing no more than three *a*'s
(c) all strings which contain at least one occurrence of each symbol in Σ.

(d) all strings which contain no runs of a's of length greater than two.

(e) all strings in which all runs of a's have lengths that are multiples of three.

Solution

(a) R.E = $(b + c)^* \, a \, (b + c)^*$ [for all strings containing exact one a]

(b) All strings containing no more than three a's: We can describe the string containing zero, one, two or three a's (and nothing else) as

$$(\lambda + a)\,(\lambda + a)\,(\lambda + a)$$

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

$$X\,(\lambda + a)X\,(\lambda + a)X\,(\lambda + a)X$$

Therefore we put $(b + c)^*$ for each X.

$$(b+c)^*\,(\lambda + a)\,(b+c)^*\,(\lambda + a)\,(b+c)^*\,(\lambda + a)\,(b+c)^*$$

(c) *All strings which contain at least one occurrence of each symbol in* Σ*:*
Here we cannot assume the symbols are in any particular order. We have no way of saying "in any order', so we have to list the possible orders:

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

Let us put X in every place where we want to allow an arbitrary string:

$$XaXbXcX + XaXcXbX + XbXaXcX + XbXcXaX \\ + XcXaXbX + XcXbXaX$$

Finally, we replace all X's with $(a + b + c)^*$ to get the final regular expression:

$$(a+b+c)^*\,a(a+b+c)^*\,b(a+b+c)^*\,c(a+b+c)^* \; + \\ (a+b+c)^*\,a(a+b+c)^*\,c(a+b+c)^*\,b(a+b+c)^* \; + \\ (a+b+c)^*\,b(a+b+c)^*\,a(a+b+c)^*\,c(a+b+c)^* \; + \\ (a+b+c)^*\,b(a+b+c)^*\,c(a+b+c)^*\,a(a+b+c)^* \; + \\ (a+b+c)^*\,c(a+b+c)^*\,a(a+b+c)^*\,b(a+b+c)^* \; + \\ (a+b+c)^*\,c(a+b+c)^*\,b(a+b+c)^*\,a(a+b+c)^*$$

(d) *All strings which contain no runs of a's of length greater than two:* An expression containing no a, one a, or one aa:

$$(b+c)^*\,(\lambda + a + aa)(b+c)^*$$

But if we want to repeat this, we have to ensure to have least one non-*a* between repetitions:

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

(e) *All strings in which all runs of* a*'s have lengths that are multiples of three:*

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

Example 1.4.6: Find regular expressions over $\Sigma=\{a,b\}$ for the language defined as follows:

(a) $L_1=\{a^m b^m : m>0\}$

(b) $L_2=\{b^m ab^n : m>0, n>0\}$

(c) $L_3=\{a^m b^m, m>0, n>0\}$

Solution

(a) Given $L_1=\{a^m b^m : m>0\}$,
L_1 has those words beginning with one or more *a*'s followed by one or more *b*'s.
Therefore the regular expression is

$$aa^*bb^* \text{ (or) } a^*ab^*b$$

(b) Given $L_2=\{b^m ab^n : m>0, n>0\}$. This language has those words *w* whose letters are all *b* except for one '*a*' that is not the first or last letter of *w*.
Therefore the regular expression is

$$bb^*abb^*$$

(c) Given $L_3=\{a^m b^m, m>0\}$.
There is no regular expression for this beginning as L_3 is not regular.

Example 1.4.7: Determine all strings in $L((a+b)^*b(a+ab)^*)$ of length less than four.

Solution

b, ab, bb, ba, aab, abb, bab, bbb, baa, bba, aba

Example 1.4.8: Find the regular expressions for the languages defined by

(i) $L_1 = \{a^n b^m : n \geq 1, m \geq 1, nm \geq 3\}$

(ii) $L_2 = \{ab^n w : n \geq 3, w \in \{a, b\}^+\}$

(iii) $L_3 = \{vwv : v, w \in \{a, b\}^*, |v| = 2\}$

(iv) $L_4 = \{w : |w| \bmod 3 = 0\}$

Solution:

(i) Regular Expression for $L_1 = \{a^n b^m : n \geq 1, m \geq 1, nm \geq 3\}$ is given by

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

(ii) Regular Expression for $L_2 = \{ab^n w : n \geq 3, w \in \{a, b\}^+\}$ is given by

$$abbb\,(b^*)\,(a + b)\,(a + b)^*$$

(iii) Regular Expression for $L_3 = \{vwv : v, w \in \{a, b\}^*, |v| = 2\}$ is given by

$$(a + b)\,(a + b)\,(a + b)^*\,(a + b)\,(a + b)$$

(iv) The regular expression for $L_4 = \{w : |w| \bmod 3 = 0\}$ is given by

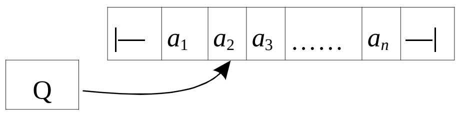
$$(aaa + bbb + ccc + aab + aba + abb + bab + bba + cab + cba + cbb + caa)^*$$

1.5 TWO-WAY FINITE AUTOMATA

Two-way finite automata are machines that can read input string in either direction. This type of machines have a “read head”, which can move left or right over the input string.

Like the finite automata, the two-way finite automata also have a finite set Q of states and they can be either deterministic (2DFA) or nondeterministic (2NFA).

They accept only regular sets like the ordinary finite automata. Let us assume that the symbols of the input string are occupying cells of a finite tape, one symbol per cell as shown in fig. The left and right endmarkers |— and —| enclose the input string. The endmarkers are not included in the input alphabet Σ.



Definition

A 2DFA is an octuple

$$M = (Q,\ \Sigma, \vdash,\ \dashv, \delta, s, t, r)$$

where, Q is a finite set of states

Σ is a finite set of input alphabet.

$\vdash$ is the left endmarker, $\vdash\ \notin \Sigma$,

$\dashv$ is the right endmarker, $\dashv\ \notin \Sigma$,

$\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to (Q \times \{L, R\})$ is the transition function.

$s \in Q$ is the start state,

$t \in Q$ is the accept state, and

$r \in Q$ is the reject state, $r \neq t$

such that for all the states q,

$$\delta\,(q, t) = (u, R) \text{ for some } u \in Q,$$
$$\delta\,(q, \dashv) = (v, L) \text{ for some } v \in Q$$

and for all symbols $b \in \Sigma \cup \{\vdash\}$

$$\delta\,(t, b) = (t, R), \qquad \delta\,(r, b) = (r, R)$$
$$\delta\,(t, \dashv) = (t, L), \qquad \delta\,(r, \dashv) = (r, L).$$

δ takes a state and a symbol as arguments and returns a new state and a direction to move the head i.e., if $\delta\,(p, b) = (q, d)$, then whenever the machine is in state p and scanning a tape cell containing symbol b, it moves its head one cell in the direction d and enters the state q.

1.6 FINITE AUTOMATA WITH OUTPUT

1.6.1 Definition

A finite-state machine $M = (Q, \Sigma, O, \delta, \lambda, q_0)$ consists of a finite set Q of states, a finite input alphabet Σ, a finite output alphabet O, a transition function δ that assigns to each state and input pair a new state, an output function λ that assigns to each state and input pair an output, and an initial state q_0.

Let $M = (Q, \Sigma, O, \delta, \lambda, q_0)$ be a finite state machine. A state table is used to denote the values of the transition function δ and the output function λ for all pairs of states and input.

1.6.2 Mealey Machine

Usually the finite automata have binary output, i.e., they accept the string or do not accept the string. This is basically decided on the basis of whether the final state is reached by the initial state. Removing this restriction, we are trying to consider a model where the outputs can be chosen from some other alphabet.

The values of the output function $F(t)$ in the most general case is a function of the present state $q(t)$ and present input $x(t)$.

$$F(t) = \lambda(q(t),\ x(t))$$

where λ is called the output function.

This model is called the "Mealey machine".

A "Mealey machine" is a six-tuple $(Q, \Sigma, O, \delta, \lambda, q_0)$ where all the symbols except λ have the same meaning as discussed in the sections above.

λ is the output function mapping $\Sigma \times Q$ into O.

1.6.3 Moore Machine

If the output function $F(t)$ depends only on the present state and is independent of the present input $q(t)$, then we have the output function $f(t)$ given by

$$F(t) = \lambda(q(t))$$

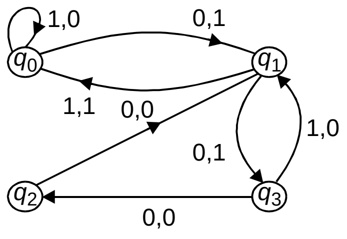
A Moore machine is a six-tuple $(Q, \Sigma, O, \delta, \lambda, q_0)$ with the usual meanings for symbols.

Example 1.6.1: Given state table as shown below that describes a finite-state machine with states $Q = \{q_0, q_1, q_2, q_3\}$, input alphabet $\Sigma = \{0,1\}$ and output alphabet $O = \{0, 1\}$, sketch the state diagram.

State	δ		λ	
	Input		Output	
	0	1	0	1
q_0	q_1	q_0	1	0
q_1	q_3	q_0	1	1
q_2	q_1	q_2	0	1
q_3	q_2	q_1	0	0

Solution

The given state table corresponds to finite-state machine with output. The corresponding state diagram is shown below.



Example 1.6.2: Give examples for Moore and Mealy Models of finite automata with outputs.

Solution

State Table shown in Fig. (a) represents a Moore Machine and that of Fig. (b) shows a Mealey Machine.

	Current State	Next State δ		Output λ
		Input		
		0	1	
Input $\rightarrow$	q_0	q_3	q_1	0
	q_1	q_1	q_2	1
	q_2	q_2	q_3	0
	q_3	q_3	q_0	0

Fig. (a)

		Next State			
		Input 0		Input 1	
		State	Output	State	Output
Input $\rightarrow$	q_1	q_3	0	q_2	0
	q_2	q_1	1	q_4	0
	q_3	q_2	1	q_1	1
	q_4	q_4	1	q_3	0

Fig. (b)

1.7 PROPERTIES OF REGULAR SETS (LANGUAGES)

A regular set (language) is a set accepted by a finite automaton.

1.7.1 Closure

A set is closed under an operation if, whenever the operation is applied to members of the set, the result is also a member of the set.

For example, the set of integers is closed under addition, because $x + y$ is an integer whenever x and y are integers. However, integers are not closed under division: if x and y are integers, x/y may or may not be an integer.

There are several operations defined on languages:

$L_1 \cup L_2$	:	strings in either L_1 or L_2.
$L_1 \cap L_2$	:	strings in both L_1 and L_2.
$L_1 L_2$	:	strings composed of one string from L, followed by one string from L_2.
$-L_2$	:	All strings (over the same alphabet) not in L_1.
L_1^*	:	Zero or more strings from L_1 concatenated together
$L_1 - L_2$	:	strings in L_1 that are not in L_2.
L_1^R	:	strings in L_1, reversed.

We shall show that the set of regular languages is closed under each of these operations.

1.7.2 Union, Concatenation, Negation, Kleene Star, Reverse

The general approach is as follows:

(i) Build automata (DFA or NFA) for each of the languages involved.
(ii) Show how to combine the automata in order to form a new automaton which recognizes the desired language.
(iii) Since the language is represented by NFA/DFA, we shall conclude that the language is regular.

Union of L_1 and L_2

(a) Create a new start state
(b) Make a λ-transition from the new start state to each of the original start states.

Concatenation of L_1 and L_2

(a) Put a λ-transition from each final state of L_1 to the initial state of L_2.
(b) Make the original final states of L_1 nonfinal.

1.7.3 Intersection and Set Difference

Just as with the other operations, it can be proved that regular languages are closed under intersection and set difference by starting with automata for the initial languages, and constructing a new automaton that represents the operation applied to the initial languages.

In this construction, a completely new machine is formed, whose states are labelled with an ordered pair of state names: the first element of each pair is a state from L_1 and the second element of each pair is a state from L_2.

(a) Begin by creating a start state whose label is (start state of L_1, start state of L_2).

(b) Repeat the following until no new arcs can be added:

(1) Find a state (A, B) that lacks a transition for some x in Σ.
(2) Add a transition on x from state (A, B) to state $(\delta(A, x), \delta(B, x))$. (If this state does not already exist, create it).

Negation of L_1

(a) Start with a complete DFA, not with an NFA
(b) Make every final state nonfinal and every nonfinal state final.

Kleene star of L_1

(a) Make a new start state; connect it to the original start state with a λ-transition.
(b) Make a new final state; connect the original final state (which becomes nonfinal) to it with λ-transitions.
(c) Connect the new start state and new final state with a pair of λ-transitions.

Reverse of L_1

(a) Start with an automaton with just one final state.
(b) Make the initial state final and final state initial.
(c) Reverse the direction of every arc.

The same construction is used for both intersection and set difference. The distinction is in how the final states are selected.

Intersection

Make a state (A, B) as final if both

(i) A is a final state in L_1 and
(ii) B is a final state in L_2

Set Difference

Mark a state (A, B) as final if A is a final state in L_1, but B is not a final state in L_2.

1.8 PUMPING LEMMA

1.8.1 Principle of Pumping Lemma

- If an infinite language is regular, it can be defined by a DFA.
- The DFA has some finite number of states (say).
- Since the language is infinite, some strings of the language should have length > n.

- For a string of length > n accepted by the DFA, the walk through the DFA must contain a cycle.
- Repeating the cycle an arbitrary number of times should yield another string accepted by the DFA.

The “pumping lemma” for regular languages is another way of showing that a given infinnite language is not regular. The proof is always done by “contradiction”. The technique that is followed is as outlined below:

(i) Assume that the language L is regular.
(ii) By Pigeon-hole principle, any sufficiently long string in L should repeat some state in the DFA, and therefore, the walk contains a “cycle”.
(iii) Show that repeating the cycle some number of times (“pumping” the cycle) yields a string that is not in L.
(iv) Conclude that L is not regular.

1.8.2 Applying the Pumping Lemma

Definition of Pumping Lemma

If L is an infinite regular language, then there exists some positive integer ‘m’ such that any string $w \in L$, whose length is ‘m’ or greater can be decomposed into three parts, xyz where

(i) $|xy|$ is less than or equal to m.
(ii) $|y| > 0$,
(iii) $w_i = xy^i z$ is also in L for all i = 0, 1, 2, 3, ……

To use this lemma, we need to show:

(i) For any choice of m,
(ii) For some $w \in L$ that we get to choose (and we will choose one of length at least ‘m’).
(iii) For any way of decomposing w into xyz, so long as $|xy|$ is not greater than m and y is not λ,
(iv) We can choose an i such that xy^iz is not in L.

Example 1.8.1: Prove that $L = \{a^n b^n : n \geq 0\}$ is not regular.

Solution

(i) We don’t know m, but let us assume that there is one.
(ii) Choose a string $w = a^n b^n$, where $n > m$, so that any prefix of length ‘m’ consists only of a’s.

(iii) We don't know the decomposition of w into xyz, but since $|xy| \le m$, xy must consist entirely of a's. Moreover, y cannot be empty.

(iv) Choose $i = 0$. This has the effect of dropping $|y|$ a's out of the string, without affectng the number of b's. The resultant string has fewer a's than b's, hence does not belong to L.

Therefore L is not regular.

Example 1.8.2: Prove that $L = \{a^n b^k \ : n > k \text{ and } n \ge 0\}$ is not regular.

Solution

(i) We do not know 'm', but assume there is one.

(ii) Choose a string $w = a^n b^k$, where $n > m$, so that any prefix of length 'm' consists entirely of a's, and $k = n - 1$, so that there is just one more a than b.

(iii) We do not know the decomposition of w into xyz, but since $|xy| \le m$, xy must consist entirely of a's. Moreover, y cannot be empty.

(iv) Choose $i = 0$. This has the effect of dropping $|y|$ a's out of the string, without affecting the number of b's. The resultant string fewer a's than before, so it has either fewer a's than b's, or the same number of each. Either way, the string does not belong to L, so L is not regular.

Example 1.8.3: Show that $L = \{a^n : n \text{ is a prime number}\}$ is not regular.

Solution

(i) We don't know m, but assume there is one.

(ii) Chose a string $w = a^n$ where n is a prime number and $|xyz| = n > m + 1$. (This can always be done because there is no largest prime number). Any prefix of w consists entirely of a's.

(iii) We do not know the decomposition of w into xyz but since $|xy| \le m$, it follows that $|z| > 1$. As usual, $|y| > 0$.

(iv) Since $|z| > 1$, $|xy| > 1$. Choose $i = |xz|$. Then $|xy^i z| = |xz| + |y|\ |xz|$

$$= (1 + |y|)\ |xz|.$$

Since $(1 + |y|)$ and $|xz|$ are each greater than 1, the product must be a composite number.

Therefore $|xy^i z|$ is a composite number.

Hence L is not regular.

1.9 CLOSURE PROPERTIES OF REGULAR LANGUAGES

THEOREM 1: If L_1 and L_2 are regular over Σ, then $L_1 \cup L_2$ is regular i.e., union of two regular sets is also regular. [Regular sets are closed w.r.t. union].

Proof: As L_1 and L_2 are given to be regular; there exists finite automata $M_1 = (Q, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ such that $L_1 = T(M_1)$ and $L_2 = T(M_2)$.

$$[T(M) = \{x \in \Sigma^* : \delta(q_0, x) \in F\}$$

is a language $L(M)$ accepted by M]

Let us assume that $Q_1 \cap Q_2 = \varnothing$.

Let us define NFA with $\in$-transitions as follows:

$$M_3 = (Q, \Sigma, \delta, q_0, F)$$

where

(i) $Q = Q_1 \cup Q_2 \cup \{q_0\}$ where q_0 is a new state not in $Q_1 \cup Q_2$

(ii) $F = F_1 \cup F_2$

(iii) δ is defined by $\delta(q_0, \in) = \{q_1, q_2\}$ (a)

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{if } q \in Q_1 \\ \delta_2(q, a) & \text{if } q \in Q_2 \end{cases} \qquad \text{(b)}$$

It is obvious that $\delta(q_0, \in) = \{q_1, q_2\}$ induces $\in$-transitions either to the initial state q_1 of M_1 or initial state q_2 of M_2.

From (b), the transitions of M are the same as transitions M_1 or M_2 depending on whether q_1 or q_2 reached by $\in$-transitions from q_0.

Since $F = F_1 \cup F_2$, any string accepted by M_1 or M_2 accepted by M.

Therefore $L_1 \cup L_2 = T(M)$ and so is regular. □

THEOREM 2: If L is regular and $L \subseteq \Sigma^*$, then $\Sigma^* - L$ is also a regular set.

Proof: Let $L = T(M)$ where $M = (Q, \Sigma, \delta, q_0, F)$ is an FA.

Though $L \subseteq \Sigma^*$, $\delta(q, a)$ need not be defined as for all 'a' in Σ.

$\delta(q, a)$ is defined for some 'a' in Σ eventhough 'a' does not find a place in the strings accepted by M.

Let us now modify Σ, Q and δ as defined below.

(i) If $a \in \Sigma_1 - \Sigma$, then the symbol 'a' will not appear in any string of $T(M)$. Therefore we delete 'a' from Σ_1 and all transitions defined by the symbol 'a'. $T(M)$ is not affected by this).

(ii) If $\Sigma - \Sigma_1 \neq \varnothing$, we add a dead state d to Q. We define $\delta(d, a) = d$ for all 'a' in Σ and $\delta(d, a) = d$ for all q in Q and 'a' in $\Sigma - \Sigma_1$.

Once again $T(M)$ is not affected by this.

Let us consider M got after applying (i) and (ii) to Σ, Q and δ. We write the modified M as

$$(Q, \Sigma, \delta, q_0, F).$$

Let us now define a new automaton M' by

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

We can see that $w \in T(M')$ iff $\delta(q_0, w) \in Q - F$ and $w \notin T(M)$.

Therefore $\Sigma^* - L = T(M')$ and therefore regular. □

THEOREM 3: If L_1 and L_2 are regular, so is $L_1 \cap L_2$ [Regular sets are closed w.r.t. Intersection]

Proof: It is important to note that

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

If L_1 and L_2 are regular, then L_1^c, L_2^c are regular by theorem 1.

Therefore $(L_1^c \cup L_2^c)^c$ is regular by theorem 2.

Hence $L_1 \cap L_2$ is regular. □

1.10 MYHILL-NERODE THEOREM

1.10.1 Myhill-Nerode Relations

Isomorphism

Two DFAs given by $M = (Q_M, \Sigma, \delta_m, s_m, F_m)$ and $N = (Q_N, \Sigma, \delta_n, s_n, F_n)$ are said to be "isomorphic" if there is a one-to-one and onto mapping $f : Q_M \to Q_N$ such that

(i) $f(s_M) = s_N$,
(ii) $f(\delta_M(p, a)) = \delta_N(f(p), a)$ for all $P \in Q_M$, $a \in \Sigma$,
(iii) $p \in F_M$ *if* $f(p) \in F_N$.

Isomorphic automata accept same set.

Myhill-Nerode Relations

Let $R \subseteq \Sigma^*$ be a regular set, and let $M = (Q, \Sigma, \delta, s, F)$ be a DFA for R with no inaccessible states.

The automaton M induces an equivalence relation $\equiv_M$ on Σ^* defined by

$$x \equiv_M y \overset{def}{\Leftrightarrow} \hat{\delta}(s, x) = \hat{\delta}(s, y)$$

It is easy to show that the relation $\equiv_M$ is an equivalence relation, meaning it is reflexive, symmetric and transitive.

A few properties satisfied by $\equiv_M$ are as follows:

(a) It is a *right congruence:* for any $x, y \in \Sigma^*$ and $a \in \Sigma$,

$$x \equiv_M y \Rightarrow xa \equiv_M y_a$$

Proof: Assume $x \equiv_M y$.
Therefore we have

$$\begin{aligned} \hat{\delta}(s, xa) &= \delta(\hat{\delta}(s, x), a) \\ &= \delta(\hat{\delta}(s, y), a) \quad \text{(by assumption)} \\ &= \hat{\delta}(s, ya) \end{aligned}$$

□

(b) It refines R : for any $x, y \in \Sigma^*$,

$$x \equiv_M y \Rightarrow (x \in R \Leftrightarrow y \in R).$$

Proof: Since $\hat{\delta}(s, x) = \hat{\delta}(s, y)$, which is either an accept state or a reject state, so either both x and y are accepted or both are rejected. □

(c) It is of "Finite index": i.e., it has only finitely many equivalence class.
This is because there is exactly one equivalence class

$$\{x \in \Sigma^* \mid \hat{\delta}(s, x) = q\}$$

corresponding to each state q of M.

Hence the equivalence relation $\equiv$ on Σ^* is a "Myhill-Herode relation" for R if it satisfies properties (a), (b) and (c). i.e., if it is a right congruence of finite index refining R.

1.10.2 Myhill-Nerode Theorem

Let $R \subseteq \Sigma^*$. The following statements are equivalent.

(i) R is regular
(ii) There exists a Myhill-Nerode relation for R
(iii) The relation $\equiv_R$ is of finite index.

(The proof is beyond the scope of this book).

Example 1.10.1: Using Myhill-Nerode Theorem verify whether $L = \{a^n b^n : n \geq 0\}$ is regular or not.

Solution

This is done by determining the $\equiv_R$-classes. If $k \neq m$, then $a^k \not\equiv_L a^m$, since

Chapter 2

Context-Free Grammars

2.1 INTRODUCTION

2.1.1 Definition of CFG

A context-free grammar is a 4-tuple (V, T, S, P) where

(i) V is a finite set called the variables
(ii) T is a finite set, disjoint from V, called the terminals
(iii) P is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
(iv) $S \in V$ is the start variable.

If u, v and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv yields uwv, written $uAv \Rightarrow uwv$.

2.1.2 Example of CFG

Given a grammar $G = (\{S\}, \{a, b\}, R, S)$.
The set of rules R is

$$S \to aSb$$
$$S \to SS$$
$$S \to \in$$

This grammar generates strings such as

$$abab, aaabbb, \text{ and } aababb$$

If we assume that a is left paranthesis '(' and b is right paranthesis ')', then $L(G)$ is the language of all strings of properly nested parantheses.

2.1.3 Right-Linear Grammar

In general productions have the form:

$$(V \cup T)^+ \to (V \cup T)^*.$$

In right-linear grammar, all productions have one of the two forms:

$$V \to T^*V$$

or $$V \to T^*$$

i.e., the left hand side should have a single variable and the right hand side consists of any number of terminals (members of *T*) optionally followed by a single variable.

2.1.4 Right-Linear Grammars and NFAs

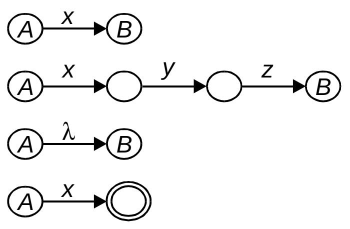
There is a simple connection between right-linear grammars and NFAs, as shown in the following illustration.

$A \to xB$

$A \to xyzB$

$A \to B$

$A \to x$



As an example of the correspondence between an NFA and a right linear grammar, the following automaton and grammar both recognize the set of set of strings consisting of an even number of 0's and an even number of 1's.

$S \to \lambda$

$S \to 0B$

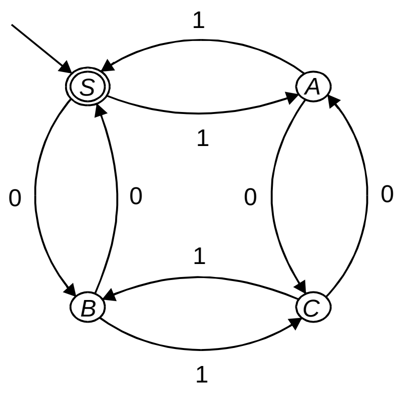
$S \to 0A$

$A \to 0C$

$A \to 1S$

$B \to 0S$

$B \to 1C$



2.1.5 Left-Linear Grammar

In a left-linear grammar, all productions have one of the two forms:

$$V \to VT^*$$

or $$V \to T^*$$

i.e., the left hand side must consist of a single varibale, and the right-hand side consists of an optional single variable followed by one number of terminals.

2.1.6 Conversion of Left-linear Grammar into Right-Linear Grammar

	Step	Method
(a)	Construct a right-linear grammar for the different languages L^R.	Replace each production $A \to x$ of L with a production $A \to x^R$ and replace each production $A \to Bx$ with a production $A \to x^R B$
(b)	Construct an NFA for L^R from the right-linear grammar. This NFA should have just one final state.	Refer to section 2.1.4 for deriving an NFA from a right-linear grammar.
(c)	Reverse the NFA for L^R to obtain an NFA for L.	(i) Construct an NFA to recognize the language L. (ii) Ensure the NFA has only a single final state (iii) Reverse the direction of arcs (iv) Make the initial state final and final state initial
(d)	Construct a right-linear grammar for L from the NFA for L.	This is the technique described in the previous section.

Example 2.1.1: Give some example of context-free languages.

Solution

(a) The grammar $G = (\{S\}, \{a, b\}, S, P)$ with productions

$$S \to aSa, \quad S \to bSb, \quad S \to \lambda$$

is context free.

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$$

Thus we have $L(a) = \{ww^R \ : w \in \{a, b\}^*\}$.

This language is context free.

(b) The grammar G, with production rules given by

$$\begin{aligned} &S \to abB, \\ &A \to aaBb, \\ &B \to bbAa, \\ &A \to \lambda \end{aligned}$$

is context free.

The language is

$$L(G) = \{ab\,(bbaa)^n\,bba(ba)^n : n \geq 0\}$$

Example 2.1.2: Construct right-and left-linear grammars for the language $L = \{a^n b^m \ : n \geq 2,\ m \geq 3\}$.

Solution

Right-Linear Grammar:

$$\begin{aligned} S &\to aS \\ S &\to aaA \\ A &\to bA \\ A &\to bbb \end{aligned}$$

Left-Linear Grammar:

$$\begin{aligned} S &\to Abbb \\ S &\to Sb \\ A &\to Aa \\ A &\to aa \end{aligned}$$

2.2 DERIVATION TREES

A 'derivation tree' is an ordered tree which the the nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides.

2.2.1 Definition of a Derivation Tree

Let $G = (V, T, S, P)$ be a CFG. An ordered tree is a derivation tree for G iff it has the following properties:

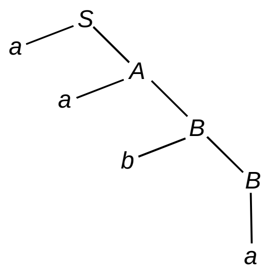
(i) The root of the derivation tree is S.
(ii) Each and every leaf in the tree has a label from $T \cup \{\lambda\}$.
(iii) Each and every interior vertex (a vertex which is no a leaf) has a label from V.
(iv) If a vertex has label $A \in V$, and its children are labeled (from left to right) $a_1, a_2, \ldots\ldots\ a_n$, then P must contain a production of the form

$$A \to a_1,\ a_2,\ \ldots\ \ldots\ a_n$$

(v) A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ can have no other children.

2.2.2 Sentential Form

For a given CFG with productions $S \to aA, A \to aB, B \to bB, B \to a$. The derivation tree is as shown below.



$$S \Rightarrow aA \Rightarrow aaB \Rightarrow aabB \Rightarrow aaba$$

The resultant of the derivation tree is the word $w = aaba$.
This is said to be in "Sentential Form".

2.2.3 Partial Derivation Tree

In the definition of derivation tree given, if every leaf has a label from $V \cup T \cup \{\lambda\}$ it is said to be "partial derivation tree".

2.2.4 Right Most/Left Most/Mixed Derivation

Consider the grammar G with production

$$\begin{Bmatrix} 1.\ S \to aSS \\ 2.S \to b \end{Bmatrix}$$

Now, we have

$$\begin{aligned} S &\overset{1}{\Rightarrow} aSS \\ &\overset{1}{\Rightarrow} aaSSS \\ &\overset{2}{\Rightarrow} aabSS \qquad \text{(Left Most Derivation)} \\ &\overset{1}{\Rightarrow} aabaSSS \\ &\overset{2}{\Rightarrow} aababSS \\ &\overset{2}{\Rightarrow} aababbS \\ &\overset{2}{\Rightarrow} aababbb \end{aligned}$$

The sequence followed is "left most derivation", following "1121222", giving "*aababbb*".

$$\begin{aligned} S &\overset{1}{\Rightarrow} aSS \\ &\overset{2}{\Rightarrow} aSb \\ &\overset{1}{\Rightarrow} aaSSb \qquad \text{(Mixed Derivation)} \\ &\overset{2}{\Rightarrow} aabSb \\ &\overset{1}{\Rightarrow} aabaSSb \\ &\overset{2}{\Rightarrow} aabaSbb \\ &\overset{2}{\Rightarrow} aababbb \end{aligned}$$

The sequence 1212122 represents a "Mixed Derivation", giving "*aababbb*".

$$\begin{aligned} S &\overset{1}{\Rightarrow} aSS \\ &\overset{2}{\Rightarrow} aSb \\ &\overset{1}{\Rightarrow} aaSSb \\ &\overset{1}{\Rightarrow} aaSaSSb \qquad \text{(Right Most Derivation)} \\ &\overset{2}{\Rightarrow} aaSaSbb \\ &\overset{2}{\Rightarrow} aaSabbb \\ &\overset{2}{\Rightarrow} aababbb \end{aligned}$$

The sequence 1211222 represents a "Right Most Derivation", giving "*aababbb*".

Example 2.2.1: A grammar *G* which is context-free has the productions

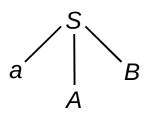
$$\begin{aligned} &S \to aAB \\ &A \to Bba \\ &B \to bB \\ &B \to c. \end{aligned}$$

(The word *w* = *acbabc* is derived as follows)

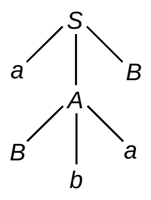
$$S \Rightarrow aAB \to a(Bba)B \Rightarrow acbaB \Rightarrow acba(bB) \Rightarrow acbabc.$$

Obtain the derivation tree.

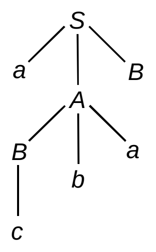
Solution



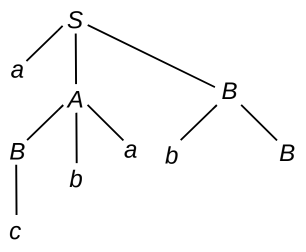
(a) $S \to aAB$



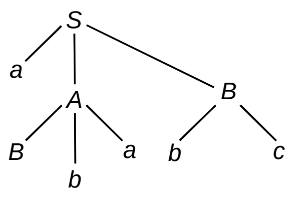
(b) $A \to Bba$



(c) $B \to c$



(d) $B \to bB$



(e) $B \to c$

Example 2.2.2: A CFG given by productions is

$$S \to a,$$
$$S \to aAS,$$
$$\text{and} \quad A \to bS$$

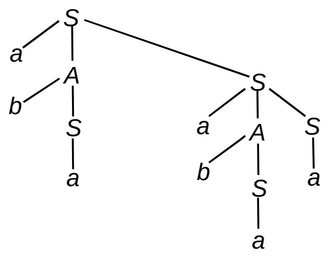
Obtain the derivation tree of the word $w = abaabaa$.

Solution

$w = abaabaa$ is derived from S as

$$\begin{aligned} S \Rightarrow aAS \Rightarrow a(bS)S \Rightarrow abaS &\Rightarrow aba(aAS) \\ &\Rightarrow abaa(bs)S \\ &\Rightarrow abaabaS \\ &\Rightarrow abaabaa \end{aligned}$$

The derivation tree is sketched below.



(c) $S \to SaSaSaSbS$
$S \to SaSaSbSaS$
$S \to SaSbSaSaS$
$S \to SbSaSaSaS$
$S \to \in$

Example 2.2.11: Given a grammar G with production rules

$$\begin{aligned} S &\to aB \\ S &\to bA \\ A &\to aS \\ A &\to bAA \\ A &\to a \\ B &\to bS \\ B &\to aBB \\ B &\to b \end{aligned}$$

Obtain the (i) leftmost derivation, and (ii) rightmost derivation for the string "*aaabbabbba*".

Solution

(i) *Leftmost derivation:*

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aaaBBB \Rightarrow aaabBB \Rightarrow aaabbB \\ &\Rightarrow aaabbabB \Rightarrow aaabbabbB \Rightarrow aaabbabbbS \Rightarrow aaabbabbba \\ &\Rightarrow aaabbabb \end{aligned}$$

(ii) *Rightmost derivation:*

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbbA \Rightarrow aaaBBbba \\ &\Rightarrow aaabBbba \Rightarrow aaabbSbba \Rightarrow aaabbaBbba \Rightarrow aaabbabbba \end{aligned}$$

2.3 PARSING AND AMBIGUITY

2.3.1 Parsing

A grammar can be used in two ways:

(a) Using the grammar to generate strings of the language.
(b) Using the grammar to recognize the strings.

"Parsing" a string is finding a derivation (or a derivation tree) for that string.

Parsing a string is like recognizing a string. The only realistic way to recognize a string of a context-free grammar is to parse it.

2.3.2 Exhaustive Search Parsing

The basic idea of the "Exhaustive Search Parsing" is to parse a string *w*, generate all strings in *L* and check if *w* is among them.

Problem arises when *L* is an infinite language. Therefore a systematic approach is needed to achieve this, as it is required to know that no strings are overlooked. And also it is necessary so as to stop after a finite number of steps.

The idea of exhaustive search parsing for a string is to generate all strings of length no greater than $| w |$, and see if *w* is among them.

The restrictions that are placed on the grammar will allow us to generate any string $w \in L$ in at most $2 | w | - 1$ derivation steps.

Exhaustive search parsing is inefficient. It requires time exponential in $|w|$.

There are ways to further restrict context free grammar so that strings may be parsed in linear or non-linear time (which methods are beyond the scope of this book).

There is no known linear or non-linear algorithm for parsing strings of a general context free grammar.

2.3.3 Topdown/Bottomup Parsing

Sequence of rules are applied in a leftmost derivation in Topdown parsing. (Refer to section 2.2.4.)

Sequence of rules are applied in a rightmost derivation in Bottomup parsing.

This is illustrated below.

Consider the grammar *G* with production

$$1.\ S \to aSS$$
$$2.\ S \to b.$$

The parse trees are as follows.

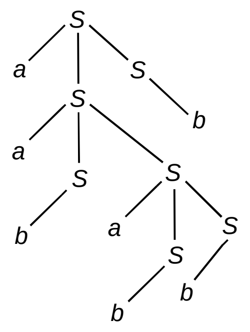


Fig. Topdown parsing.

aababbb $\to$ Left parse of the string with the sequence 1121222.

This is known as "Topdown Parsing."

"Right Parse" is the reversal of sequence of rules applied in a rightmost derivation.

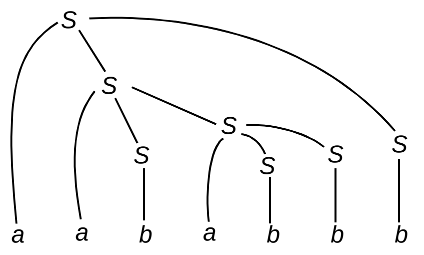


Fig. Bottom-up parsing.

aababbb $\rightarrow$ Right parse of the string with the sequence 2221121.
This is known as "Bottom-up Parsing."

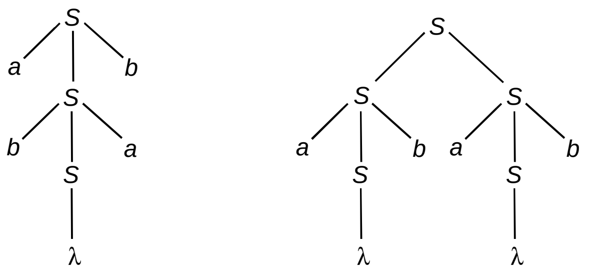
2.3.4 Ambiguity

The grammar given by

$$G = (\{S\}, \{a, b\}, S, S \rightarrow aSb \mid bSa \mid SS \mid \lambda)$$

generates strings having an equal number of *a*'s and *b*'s.

The string "*abab*" can be generated from this grammar in two distinct ways, as shown in the following derivation trees:



Similarly, "*abab*" has two distinct leftmost derivations:

$$S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$$
$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab.$$

Also, "*abab*" has two distinct rightmost derivations:

$$S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$$
$$S \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \Rightarrow abab$$

Each of the above derivation trees can be turned into a unique rightmost derivation, or into a unique leftmost derivation. Each leftmost or rightmost derivation can be turned into a unique derivation tree. These representations are largely interchangeable.

2.3.5 Ambiguous Grammars/Ambiguous Languages

Since derivation trees, leftmost derivations, and rightmost derivations are equivalent rotations, the following definitions are equivalent:

Definition: Let $G = (N, T, P, S)$ be a CFG.

A string $w \in L(G)$ is said to be "ambiguously derivable "if there are two or more different derivation trees for that string in *G*.

Definition: A CFG given by $G = (N, T, P, S)$ is said to be "ambiguous" if there exists at least one string in $L(G)$ which is ambiguously derivable. Otherwise it is unambiguous.

Ambiguity is a property of a grammar, and it is usually, but not always possible to find an equivalent unambiguous grammar.

An "inherantly ambiguous language" is a language for which no unambiguous grammar exists.

Example 2.3.1: Prove that the grammar

$$\begin{aligned} &S \to aB \mid ab, \\ &A \to aAB \mid a, \\ &B \to ABb \mid b \end{aligned}$$

is ambiguous.

Solution

It is easy to see that "*ab*" has two different derivations as shown below.

Given the grammar *G* with production

1. $S \to aB$
2. $S \to ab$
3. $A \to aAB$
4. $A \to a$
5. $B \to ABb$
6. $B \to b$

$$\left.\begin{array}{ll} \text{Using (2),} & S \Rightarrow ab \\ \text{Using (1),} & S \Rightarrow aB \Rightarrow ab \\ \text{and then (6).} & \end{array}\right\}$$

Example 2.3.2: Show that the grammar $S \to S \mid S,\ S \to a$ is ambiguous.

Solution

In order to show that *G* is ambiguous, we need to find a $w \in L(G)$, which is ambiguous.

Therefore, the new set of productions $\hat{P}$ for the grammar equivalent to the given CFG is

$$\begin{aligned} &S \to AB \,|\, A \,|\, B \\ &S \to aAA \,|\, aA \,|\, a \\ &B \to bBB \,|\, bB \,|\, b. \end{aligned}$$

2.5 NORMAL FORMS

Two kinds of normal forms viz., Chomsky Normal Form and Greibach Normal Form (GNF) are considered here.

2.5.1 Chomsky Normal Form (CNF)

Any context-free language L without any λ-production is generated by a grammar is which productions are of the form $A \to BC$ or $A \to a$, where $A, B \in V_N$, and $a \in V_{T.}$

Procedure to find Equivalent Grammar in CNF

(i) Eliminate the unit productions, and λ-productions if any,
(ii) Eliminate the terminals on the right hand side of length two or more.
(iii) Restrict the number of variables on the right hand side of productions to two.

Proof:

For Step (i): Apply the following theorem:

“Every context free language can be generated by a grammar with no useless symbols and no unit productions”.

At the end of this step the RHS of any production has a single terminal or two or more symbols. Let us assume the equivalent resulting grammar as $G = (V_N, V_T, P, S)$.

For Step (ii): Consider any production of the form

$$A \to y_1 y_2 \ \ldots\ldots\ y_m, \quad m \geq 2.$$

If y_1 is a terminal, say ‘a’, then introduce a new variable B_a and a production

$$B_a \to a$$

Repeat this for every terminal on RHS.

Let P' be the set of productions in P together with the new productions

$B_a \rightarrow a$. Let V_N' be the set of variables in V_N together with $B_a's$ introduced for every terminal on RHS.

The resulting grammar $G_1 = (V_N', V_T, P', S)$ is equivalent to G and every production in P' has either a single terminal or two or more variables.

For step (iii): Consider $A \rightarrow B_1 B_2 \ldots\ldots B_m$

where B_i's are variables and $m \geq 3$.

If $m = 2$, then $A \rightarrow B_1, B_2$ is in proper form.

The production $A \rightarrow B_1 B_2 \ldots\ldots B_m$ is replaced by new productions

$$\begin{aligned} A &\rightarrow B_1 D_1, \\ D_1 &\rightarrow B_2 D_2, \\ \ldots\ldots & \ \ldots\ldots \\ \ldots\ldots & \ \ldots\ldots \\ D_{m-2} &\rightarrow B_{m-1} B_m \end{aligned}$$

where $D_i'S$ are new variables.

The grammar thus obtained is G_2, which is in CNF.

Example 2.5.1: Obtain a grammar in Chomsky Normal Form (CNF) equivalent to the grammar G with productions P given

$$\begin{aligned} S &\rightarrow aAbB \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b. \end{aligned}$$

Solution

(i) There are no unit productions in the given set of P.

(ii) Amongst the given productions, we have

$$\begin{aligned} A &\rightarrow a, \\ B &\rightarrow b \end{aligned}$$

which are in proper form.

For $S \rightarrow aAbB$, we have

$$\begin{aligned} S &\rightarrow B_a A B_b B, \\ B_a &\rightarrow a \\ B_b &\rightarrow b. \end{aligned}$$

For $A \rightarrow aA$, we have

$$A \rightarrow B_a A$$

For $B \rightarrow bB$, we have

$$B \rightarrow B_b B.$$

Therefore, we have G_1 given by

$$G_1 = (\{S, A, B, B_a, B_b\}, \{a, b\}, P', S)$$

where P' has the productions

$$\begin{aligned} S &\to B_a AB_b B \\ A &\to B_a A \\ B &\to B_b B \\ B_a &\to a \\ B_b &\to b \\ A &\to a \\ B &\to b \end{aligned}$$

(iii) In P' above, we have only

$$S \to B_a AB_b B$$

not in proper form.

Hence we assume new variables D_1 and D_2 and the productions

$$\begin{aligned} S &\to B_a D_1 \\ D_1 &\to AD_2 \\ D_2 &\to B_b B \end{aligned}$$

Therefore the grammar in Chomsky Normal Form (CNF) is G_2 with the productions given by

$$\begin{aligned} S &\to B_a D_1, \\ D_1 &\to AD_2, \\ D_2 &\to B_b B, \\ A &\to B_a A, \\ B &\to B_b B, \\ B_a &\to a, \\ B_b &\to b, \\ A &\to a, \end{aligned}$$

and $B \to b.$

Example 2.5.2: Obtain a grammar in Chomsky Normal Form (CNF) equivalent to the grammar G with productions P given by

$$\begin{aligned} S &\to ABa \\ A &\to aab \\ B &\to AC \end{aligned}$$

Solution

(i) The given set P does not have any unit productions or λ-productions.

(ii) None of the given rules is in proper form.

For $S \rightarrow ABa$, we have

$$S \rightarrow ABB_a$$

and $$B_a \rightarrow a$$

For $A \rightarrow aab$, we have

$$A \rightarrow B_a B_a B_b$$

and $$B_b \rightarrow b.$$

For $B \rightarrow Ac$, we have

$$B \rightarrow AB_c$$

and $$B_c \rightarrow c$$

Therefore G_1 has a set of productions P' given by

$$\begin{aligned} S &\rightarrow ABB_a \\ A &\rightarrow B_a B_a B_b \\ B &\rightarrow AB_c \\ B_a &\rightarrow a \\ B_b &\rightarrow b \\ B_c &\rightarrow c \end{aligned}$$

(iii) In P' above, we have

$$\begin{aligned} S &\rightarrow ABB_a \\ A &\rightarrow B_a B_a B_b \end{aligned}$$

not in proper form.

Hence we assume new variables D_1 and D_2 and the productions

$$\begin{aligned} S &\rightarrow AD_1, \\ D_1 &\rightarrow BB_a, \\ A &\rightarrow B_a D_2, \\ D_2 &\rightarrow B_a B_b. \end{aligned}$$

Thus the grammar in Chomsky Normal Form (CNF) is G_2 given by the productions given by

$$\begin{aligned} S &\rightarrow AD_1, \\ D_1 &\rightarrow BB_a, \\ A &\rightarrow B_a D_2, \\ D_2 &\rightarrow B_a B_b, \end{aligned}$$

Chapter 3

Pushdown Automata

3.1 DEFINITIONS

Let us consider a finite automata which accepts the language

$$L_1(M) = \{a^m b^n \mid m, n \geq 1\}.$$

We see that M moves from q_0 to q_1, on the occurence of a's. On seeing 'b', M moves from q_1 to q_2 and continues to be in the state q_2 on getting more b's.

Assume that the input string is given by

$$a^m b^n,$$

then the resulting state is final state and so M accepts $a^m b^n$.

Consider the language $L_2(M) = \{a^n b^n \mid n \geq 1\}$ where the number of b's and a's are equal. The FA constructed for L_1 differs from that of L_2.

For the language $L_1(M) = \{a^m b^n \mid m, n \geq 1\}$ there is not necessity to remember the number of a's. The following have to be remembered.

(a) Whether the first symbol is 'b' (to reject the string)
(b) Whether 'a' follows 'b' (to reject the string)
(c) Whether 'a' follows 'a' and 'b' follows 'b' (to accept the string).

We know that FA has only a finite number of states, M cannot remember the number of a's in $a^n b^n$ where 'n' is larger than the number of states of M.

The FA does not accept the sets of the form $\{a^n b^n \mid n \geq 1\}$. This is taken care by a "PUSHDOWN AUTOMATA".

Let us illustrate a Pushdown Automata (PDA) model.

3.1.1 Nondeterministic PDA (Definition)

An NPDA is defined by the 7-tuple

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

where	Q	=	Finite set of internal states of the control unit
	Σ	=	Input alphabet
	Γ	=	Finite set of symbols called "Stack alphabet"

δ : $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to$ finite subsets of $Q \times \Gamma^*$ is the transition function

q_0 = Initial state of the control unit $\in Q$

Z = Stack start symbol

$F \subseteq Q$ = Set of Final states.

The arguments of δ are the current state of the control unit, the current input symbol, and the current symbol on the top of the stack.

The result is a set of pairs (q, x)

where q = next state of the control unit

x = string that is put on top of the stack in place of the single symbol there before.

The 'stack' is an additional component available as part of PDA. The 'stack' increases its memory. With respect to $\{a^n b^n \mid n \geq 1\}$, we can store a's in the stack. When the symbol 'b' is encountered, an 'a' from the stack can be removed. If the stack becomes empty on the completion of processing a given string, then the PDA accepts the string.

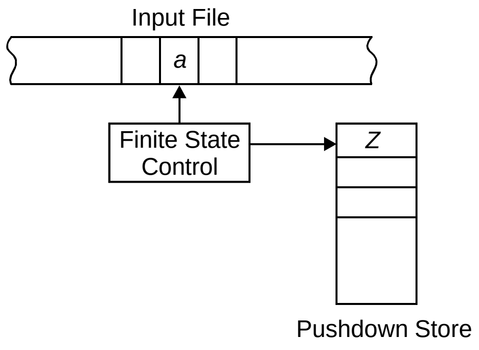


Fig. Model of Pushdown Automaton (PDA)

3.1.2 Transition Functions for NPDA

The transition function for an NPDA has the form

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to \text{Finite subsets of } Q \times \overline{\Gamma}$$

δ is now a function of three arguments.

The first two arguments are the same as before:

(i) the state

(ii) either λ, or a symbol from the input alphabet.

The third argument is the symbol on top of the stack. Just as the input symbol is "consumed" when the function is applied, the stack symbol is also "consumed" (removed from the stack).

Note that while the second argument may be λ, rather than a member of the input alphabet (so that no input symbol is consumed), there is no such option for the third argument.

δ always consumes a symbol from the stack, no move is possible if the stack is empty.

There may also be a λ-transition, where the second argument may be λ, which means that a move that does not consume an input symbol is possible. No move is possible if the stack is empty.

Example: Consider the set of transition rules of an NPDA given by

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}$$

If at any time the control unit is in state q_1, the input symbol read is 'a', and the symbol on the top of stack is 'b', then one of the following two cases can occur:

(a) The control unit tends to go into the state q_2 and the string 'cd' replaces 'b' on top of the stack.

(b) The control unit goes into state q_3 with the symbol b removed from the top of the stack.

In the deterministic case, when the function δ is applied, the automaton moves to a new state $q \in Q$ and pushes a new string of symbols $x \in \Gamma^*$ onto the stack. As we are dealing with nondeterministic pushdown automaton, the result of applying δ is a finite set of (q, x) pairs.

3.1.3 Drawing NPDAs

NPDAs are not usually drawn. However, with a few minor extensions, we can draw an NPDA similar to the way we draw an NFA.

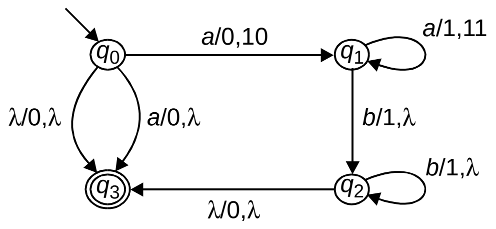
Instead of labeling an are with an element of Σ, we can label arcs with $a\,|\,x, y$ where $a \in \Sigma, x \in \Gamma$ and $y \in \Gamma^*$.

Let us consider the NPDA given by

$$(Q = \{q_0, q_1, q_2, q_3\},\ \Sigma = \{a, b\},\ \Gamma = \{0, 1\}, \delta, q_0, Z = 0, F = \{q_3\})$$

where
$$\begin{aligned}
&\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}\\
&\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}\\
&\delta(q_1, a, 1) = \{(q_1, 11)\}\\
&\delta(q_1, b, 1) = \{(q_2, \lambda)\}\\
&\delta(q_1, b, 1) = \{(q_2, \lambda)\}\\
&\delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}
\end{aligned}$$

This NPDA is drawn as follows.



Please note that the top of the stack is considered to be the left, so that, for example, if we get an '*a*' from the starting position, the stack changes from '0' to '10.'.

3.1.4 Execution of NPDA

Assume that someone is in the middle of stepping through a string with a DFA, and we need to take over and finish the job. There are two things that are required to be known:

(a) the state of the DFA is in, and
(b) what the remaining input is.

But if the automaton is an NPDA we need to know one more viz., contents of the stack.

Instantaneous Description of a PDA

The Instantaneous description of a PDA is a triplet (q, w, u),

where	q	=	current state of the automaton
	w	=	unread part of the input string
	u	=	stack contents (written as a string, with the leftmost symbol at the top of the stack).

Let the symbol "$\vdash$" denote a move of the NPDA, and suppose that $\delta(q_1, a, x) = \{(q_2, y), \dots\}$, then the following is possible:

$$(q_1, aW, xz) \vdash (q_2, W, yz)$$

where W indicates the rest of the string following 'a' and Z indicates the rest of the stack contents underneath the x.

This notation tells that in moving from state q_1 to state q_2, an 'a' is consumed from the input string aW, and the x at the top (left) of the stack xZ is replaced with y, leaving yZ on the stack.

3.1.5 Accepting Strings with an NPDA

Assume that you have the NPDA given by

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

To recognize string w, begin with the instantaneous assumption

$$(q_0, w, z)$$

where q_0 = start state
w = entire string to be processed, and
z = start stack symbol.

Starting with this instantaneous description, make zero or more moves, just as is done with an NFA.

There are two kinds of moves that can be made:

(a) λ-*Transitions:* If you are in state q_1, x is the top (leftmost) symbol in the stack, and

$$\delta(q_1, \lambda, x) = \{(q_2, w_2), \ \ldots\ldots\}$$

then you can replace the symbol x with the string w_2 and move to q_2.

(b) *Nonempty transitions:* If you are in the state q_1, 'a' is the next unconsumed input symbol, x is the top (leftmost) symbol in the stack, and

$$\delta(q_1, a, x) = \{(q_2, w_2), \ \ldots\ldots\}$$

then you can remove the 'a' from the input string, replace the symbol x with the string w_2, and move to state q_2.

If you are in the final state when you reach the end of the string (and may be make some λ-transition after reaching the end), then the string is accepted by the NPDA. It does not matter what is on the stack.

3.1.6 An Example of NPDA Execution

Let us consider the NPDA given by

$$\begin{aligned}
&\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}\\
&\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}\\
&\delta(q_1, a, 1) = \{(q_1, 11)\}\\
&\delta(q_1, b, 1) = \{(q_2, \lambda)\}\\
&\delta(q_2, b, 1) = \{(q_2, \lambda)\}\\
&\delta(q_1, \lambda, 0) = \{(q_3, \lambda)\}
\end{aligned}$$

It is possible for us to recognize the string "*aaabbb*" using the following sequence of "Moves":

$$\begin{aligned}
(q_0, aaabbb, 0) &\vdash (q_1, aabbb, 10)\\
&\vdash (q_1, abbb, 110)\\
&\vdash (q_1, bbb, 1110)\\
&\vdash (q_2, bb, 110)\\
&\vdash (q_2, b, 10)\\
&\vdash (q_2, \lambda, 0)
\end{aligned}$$

3.1.7 Accepting Strings with NPDA (Formal Version)

The notation "$\vdash$" is used to indicate a single move of an NPDA.

"$\vdash^*$" is used to indicate a sequence of zero or more moves.

"$\vdash^+$" is used to indicate a sequence of one or more moves.

If $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ is an NPDA, then the language accepted by M, L(M), is given by

$$L(M) = \{w \in \Sigma^* : (q_0, w, z) \overset{*}{\vdash} (p, \lambda, u),\ p \in F, u \in \Gamma^*\}.$$

Example 3.1.1: Construct a Push Down Automata (PDA) accepting $\{a^n b^m a^n \mid m, n \geq 1\}$ by empty store.

Solution

The PDA which will accept

$$\{a^n b^m a^n \mid m, n \geq 1\}$$

is given below

$$\text{PDA} = (\{q_0, q_1\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \phi)$$

where δ is given by

(1) $\delta(q_0, a, z_0) = \{(q_0, a z_0)\}$
(2) $\delta(q_0, a, a) = \{(q_0, aa)\}$
(3) $\delta(q_0, b, a) = \{(q_1, a)\}$
(4) $\delta(q_1, b, a) = \{(q_1, a)\}$
(5) $\delta(q_1, a, a) = \{(q_1, \lambda)\}$
(6) $\delta(q_1, \lambda, z_0) = \{(q_1, \lambda)\}$

Therefore we can see that we start storing a's till b occurs ((1) and (2)). When the current input symbol is b, the state changes, but no change in PDS occurs ((3)). Once all the b's in the input string acts over ((4)), the remaining a's are erased ((5)).

Using (6), z_0 is erased.

Therefore we have

$$(q_0, a^n\ b^m\ a^n, z_0) \overset{*}{\longmapsto} (q_1, \lambda, z_0) \longmapsto (q_1, \lambda, \lambda)$$

Therefore we see that $a^n\ b^m\ a^n \in N$ (PDA).

Example 3.1.2: Construct a PDA accepting $\{a^n\ b^{2n} \mid n \geq 1\}$ by empty store.

Solution

The PDA that will accept $\{a^n\ b^{2n} \mid n \geq 1\}$ is given by

$$\text{PDA} = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \phi)$$

where δ is given by

$$\begin{aligned}
\delta(q_0, a, z_0) &= \{(q_1, a\, z_0)\}\\
\delta(q_1, a, a) &= \{(q_1, a\, a)\}\\
\delta(q_1, b, a) &= \{(q_2, a)\}\\
\delta(q_2, b, a) &= \{(q_1, \lambda)\}\\
\delta(q_1, \lambda, z_0) &= \{(q_1, \lambda)\}
\end{aligned}$$

Example 3.1.3: Obtain the PDA accepting $\{a^m\ b^m\ c^n \mid m, n \geq 1\}$ by empty store.

Solution

The PDA which will accept $\{a^m\ b^m\ c^n \mid m, n \geq 1\}$ by empty store is given below.

$$\text{PDA} = (\{q_0, q_1\}, (a, b, c\}, \{z_0, z_1\}, \delta, q_0, z_0, \phi)$$

where δ is given by

$$\begin{aligned}
\delta(q_0, a, z_0) &= \{(q_0, z, z_0)\}\\
\delta(q_0, a, z_1) &= \{(q_0, z_1 z_1)\}\\
\delta(q_0, b, z_1) &= \{(q_1, \lambda)\}\\
\delta(q_1, b, z_1) &= \{(q_1, \lambda)\}\\
\delta(q_1, c, z_0) &= \{(q_1, z_0)\}\\
\delta(q_1, \lambda, z_0) &= \{(q_1, \lambda)\}
\end{aligned}$$

When an '*a*' is read z_1 is added. When a '*b*' is read then z_1 is removed.

Example 3.1.4: Construct a PDA accepting $\{a^n\ b^m\ a^n \mid m, n \geq 1\}$ by final state.

Solution

THEOREM: If $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is a PDA accepting L by null store, we can find a PDA

$$B = (Q', \Sigma, \Gamma', \delta_B, q'_0, z'_0, F')$$

which accept L by final state, i.e.,

$$L = N(A) = T(B).$$

Using the above theorem, we have

$$B = (\{q_0, q_1, q'_0, q_f\}, \{a, b\}, \{a, z_0, z'_0\}, \delta, q'_0, z'_0, \{q_f\})$$

where δ is given by

$$\delta(q'_0, \lambda, z'_0) = \{(q_0, z_0\ z'_0)\}$$

$\delta(q_0, \lambda, z_0') = \{(q_f, \lambda)\} = \delta(q_0, \lambda, z_0')$
$\delta(q_1, \lambda, z_0') = \{(q_f, \lambda)\} = \delta(q_1, \lambda, z_0')$
$\delta(q_0, a, z_0) = \{(q_0, a\, z_0)\}$
$\delta(q_0, a, a) = \{(q_0, aa)\}$
$\delta(q_0, b, a) = \{(q_1, a)\}$
$\delta(q_1, b, a) = \{(q_1, a)\}$
$\delta(q_1, a, a) = \{(q_f, \lambda)\}$
$\delta(q_1, \lambda, z_0) = \{(q_1, \lambda)\}$

Example 3.1.5: Given $L = \{a^m\, b^n \mid n < m\}$.

Derive (i) a context-free grammar that accepts L
(ii) a PDA accepting L by empty store
(iii) a PDA accepting L by final state.

Solution

(i) Given $L = \{a^m\, b^n \mid n < m\}$

CFG is given by $G = (\{S\}, \{a, b\}, P, S)$, where productions P are

$$\left.\begin{array}{l} S \to aSb \\ S \to aS \\ S \to a \end{array}\right\}$$

(ii) The PDA that will accept $L(G)$ by empty store is given by

$$A = (\{q\}, \{a, b\}, \{S, a, b\}, \delta, q, S, \phi),$$

where δ is defined by the rule:
$\delta(q, \lambda, S) = \{(q, aSb), (q, aS), (q, a)\}$
$\delta(q, a, a) = \delta(q, b, b) = \{(q, \lambda)\}$

(iii) $B = (Q', \Sigma, \Gamma', \delta_B, q_0', z_0', F')$, where
$Q' = \{q_0, q_0', q_f\}, \Gamma' = \{S, a, b, z_0'\}, F = \{q_f\}$.
δ_B is given by
$\delta_B\ (q_0', \lambda, z_0') = \{(q_1, z_0\, z_0')\}$
$\delta_B\ (q, \lambda, S) = \{(q, aSb), (q, aS), (q, a)\}$
$\delta_B\ (q, a, a) = \{(q, \lambda)\} = \delta_B\ (q, b, b)$
$\delta_B\ (q_1, \lambda, z_0') = \{(q_f, \lambda)\}$
where
$\delta_B\ (q, a, S) = \delta(q, a, S) = \phi$ and
$\delta_B\ (q, b, S) = \delta(q, b, S) = \phi$.

3.2 RELATIONSHIP BETWEEN PDA AND CONTEXT FREE LANGUAGES

3.2.1 Simplifying CFGs

The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammar.

(a) Empty Production Removal

If the empty string does not belong to a language, then there is no way to eliminate production of the form $A \to \lambda$ from the grammar.

If the empty string belongs to a language, then we can eliminate λ from all productions same for the single productions $S \to \lambda$. In this case we can eliminate any occurrences of S from the right-hand-side of productions.

(b) Unit Production Removal

We can eliminate productions of the form $A \to B$ from a CFG.

(c) Left Recursion Removal

A variable A is left-recursive if it occurs in a production of the form

$$A \to Ax$$

for any $x \in (V \cup T)^*$. A grammar is left-recursive if it contains at least one left-recursive variable.

Every CFL can be represented by a grammar that is not left-recursive.

3.2.2 Normal Forms of Context-Free Grammars

(a) Chomsky Normal Form

A grammar is in Chomsky Normal form if all productions are of the form

$$A \to BC$$

or

$$A \to a$$

where A, B and C are variables and 'a' is a terminal. Any context-free grammar that does not contain λ can be put into Chomsky Normal Form.

(b) Greibach Normal Form (GNF)

A grammar is in Greibach Normal Form if all productions are of the form

$$A \to ax$$

where 'a' is a terminal and $x \in V^*$.

Grammars in Greibach Normal Form are much longer than the CFG from which they were derived. GNF is useful for proving the equivalence of NPDA and CFG.

Thus GNF is useful in converting a CFG to NPDA.

3.2.3 CFG to NPDA

For any context-free grammar in GNF, it is easy to build an equivalent nondeterministic pushdown automaton (NPDA).

Any string of a context-free language has a leftmost derivation. We set up the NPDA so that the stack contents "corresponds" to this sentential form: every move of the NPDA represents one derivation step.

The sentential form is

(The characters already read) + (symbols on the stack)
– (Final z (initial stack symbol)

In the NPDA, we will construct, the states that are not of much importance. All the real work is done on the stack. We will use only the following three states, irrespective of the complexity of the grammar.

(i) start state q_0 just gets things initialized. We use the transition from q_0 to q_1 to put the grammar's start symbol on the stack.

$$\delta(q_0, \lambda, Z) \to \{(q_1, Sz)\}$$

(ii) State q_1 does the bulk of the work. We represent every derivation step as a move from q_1 to q_1.

(iii) We use the transition from q_1 to q_f to accept the string

$$\delta(q_1, \lambda, z) \to \{(q_f, z)\}$$

Example Consider the grammar $G = (\{S, A, B\}, \{a, b\}, S, P)$, where

$$P = \{S \to a, S \to aAB, A \to aA, A \to a, B \to bB, B \to b\}$$

These productions can be turned into transition functions by rearranging the components.

$$\begin{tikzcd}
\underline{S} \arrow[r] \arrow[d] & \underline{a} \arrow[dl] & \underline{AB} \arrow[d] \\
\delta(q_1, a, S) \arrow[rr] & & \{(q_1, AB)\}
\end{tikzcd}$$

Thus we obtain the following table:

(Start)	$\delta(q_0, \lambda, z) \to \{(q_1, Sz)\}$
$S \to a$	$\delta(q_1, a, S) \to \{(q_1, \lambda)\}$
$S \to aAB$	$\delta(q_1, a, S) \to \{(q_1, AB)\}$
$A \to aA$	$\delta(q_1, a, A) \to \{(q_1, A)\}$
$A \to a$	$\delta(q_1, a, A) \to \{(q_1, \lambda)\}$
$B \to bB$	$\delta(q_1, b, B) \to \{(q_1, B)\}$
$B \to b$	$\delta(q_1, b, B) \to \{(q_1, \lambda)\}$
(finish)	$\delta(q_1, \lambda, z) \to \{(q_f, z)\}$

For example, the derivation

$$S \Rightarrow aAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aabb$$

maps into the sequence of moves

$$\begin{aligned}(q_0, aabb, z) &\vdash (q_1, aabb, Sz)\\ &\vdash (q_1, abb, ABz)\\ &\vdash (q_1, bb, Bz)\\ &\vdash (q_1, b, Bz)\\ &\vdash (q_1, \lambda, z)\\ &\vdash (q_2, \lambda, \lambda)\end{aligned}$$

3.2.4 NPDA to CFG

(a) We have shown that for any CFG, an equivalent NPDA can be obtained. We shall show also that, for any NPDA, we can produce an equivalent CFG. This will establish the equivalence of CFGs and NFDAs.

We shall assert without proof that any NPDA can be transformed into an equivalent NPDA which has the following form:

(i) The NPDA has only one final state, which it enters if and only if the stack is empty.

(ii) All transitions have the form

$$\delta(q, a, A) = \{c_1, c_2, c_3, \ \ldots\ldots\}$$

where each c_i has one of the two forms

$$(q_j, \lambda)$$

or $$(q_j, BC)$$

(b) When we write a grammar, we can use any variable names we choose. As in programming languages, we like to use "meaningful" variable names.

When we translate an NPDA into a CFG, we will use variable names that encode information about both the state of the NPDA and the stack content variable names will have the form

$$[q_i A q_j],$$

where q_i and q_j are states and A is a variable.

The "meaning" of the variable $[q_i Aq_j]$ is that the NPDA can go from state q_i with Ax on the stack to state q_j with x on the stack.

Each transition of the form $\delta(q_i, a, A) = (q_j, \lambda)$ results in a single grammar rule.

Each transition of the form

$$\delta(q_i, a, A) = \{q_j, BC)$$

results is a multitude of grammar rules, one for each pair of states q_x and q_y in the NPDA.

3.2.5 Deterministic Pushdown Automata

A Non-deterministic finite acceptor differs from a deterministic finite acceptor in two ways:

(i) The transition function δ is single-valued for a DFA, but multi-valued for an NFA.

(ii) An NFA may have λ-transitions.

A non-deterministic pushdown automaton differs from a pushdown automaton in the following ways:

(i) The transition function δ is at most single-valued for a DPDA, multi-valued for an NPDA.
Formally: $|\delta(q_1, a, b)| = 0$ or 1, for every $q \in Q, a \in \Sigma \cup \{\lambda\}$, and $b \in \Gamma$.

(ii) Both NPDA and DPDA may have λ-transitions; but a DPDA may have a λ-transition only if no other transition is possible.
Formally: If $|\delta(q, \lambda, b)| \neq \varnothing$, then $\delta(q, c, b) = \varnothing$ for every $c \in \Sigma$.

A deterministic CFL is a language that can be recognized by a DPDA. The deterministic context-free languages are a proper subset of the context-free languages.

3.3 PROPERTIES OF CONTEXT FREE LANGUAGES

3.3.1 Pumping Lemma for CFG

A "Pumping Lemma" is a theorem used to show that, if certain strings belong to a language, then certain other strings must also belong to the language.

Let us discuss a Pumping Lemma for CFL.

We will show that , if L is a context-free language, then strings of L that are at least 'm' symbols long can be "pumped" to produce additional strings in L. The value of 'm' depends on the particular language.

Let L be an infinite context-free language. Then there is some positive integer 'm' such that, if S is a string of L of Length at least 'm', then

(i) $S = uvwxy$ (for some u, v, w, x, y)

(ii) $|vwx| \leq m$

(iii) $|vx| \geq 1$

(iv) $uv^i wx^i y \in L$.

for all non-negative values of i.

It should be understood that

(i) If S is sufficiently long string, then there are two substrings, v and x, somewhere in S. There is stuff (u) before v, stuff (w) between v and x, and stuff (y), after x.

(ii) The stuff between v and x won't be too long, because $|vwx|$ can't be larger than m.

(iii) Substrings v and x won't both be empty, though either one could be.

(iv) If we duplicate substring v, some number (i) of times, and duplicate x the same number of times, the resultant string will also be in L.

3.3.2 Definitions

A variable is useful if it occurs in the derivation of some string. This requires that

(a) the variable occurs in some sentential form (you can get to the variable if you start from S), and

(b) a string of terminals can be derived from the sentential form (the variable is not a "dead end").

A variable is "recursive" if it can generate a string containing itself. For example, variable A is recursive if

$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y.

A recursive variable A can be either

(i) "Directly Recursive", i.e., there is a production

$$A \to x_1 A x_2$$

for some strings $x_1, x_2 \in (T \cup V)^*$, or

(ii) "Indirectly Recursive", i.e., there are variables x_i and productions

$$\begin{aligned} &A \to X_1 \ldots \\ &X_1 \to \ldots X_2 \ldots \\ &X_2 \to \ldots X_3 \ldots \\ &X_N \to \ldots A \ldots \end{aligned}$$

3.3.3 Proof of Pumping Lemma

(a) Suppose we have a CFL given by L. Then there is some context-free Grammar G that generates L. Suppose

(i) L is infinite, hence there is no proper upper bound on the length of strings belonging to L.
(ii) L does not contain λ.
(iii) G has no productions or λ-productions.

There are only a finite number of variables in a grammar and the productions for each variable have finite lengths. The only way that a grammar can generate arbitrarily long strings is if one or more variables is both useful and recursive.

Suppose no variable is recursive.

Since the start symbol is nonrecursive, it must be defined only in terms of terminals and other variables. Then since those variabls are non recursive, they have to be defined in terms of terminals and still other variables and so on. After a while we run out of "other variables" while the generated string is still finite. Therefore there is an upperbond on the length of the string which can be generated from the start symbol. This contradicts our statement that the language is finite.

Hence, our assumption that no variable is recursive must be incorrect.

(b) Let us consider a string X belonging to L.

If X is sufficiently long, then the derivation of X must have involved recursive use of some variable A.

Since A was used in the derivation, the derivation should have started as

$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y. Since A was used recursively the derivation must have continued as

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy$$

Finally the derivation must have eliminated all variables to reach a string X in the language.

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = x$$

This shows that derivation steps

$$A \overset{*}{\Rightarrow} vAx$$

and $$A \overset{*}{\Rightarrow} w$$

are possible. Hence the derivation

$$A \overset{*}{\Rightarrow} vwx$$

must also be possible.

It should be noted here that the above does not imply that a was used recursively only once. The * of $\overset{*}{\Rightarrow}$ could cover many uses of A, as well as other recursive variables.

There has to be some "last" recursive step. Consider the longest strings that can be derived for v, w and x without the use of recursion. Then there is a number 'm' such that $| vwx | < m$.

Since the grammar does not contain any λ-productions or unit productions, every derivation step either introduces a terminal or increases the length of the sentential form. Since $A \overset{*}{\Rightarrow} vAx$, it follows that $|vx| > 0$.

Finally, since $uvAxy$ occurs in the derivation, and $A \overset{*}{\Rightarrow} vAx$ and $A \overset{*}{\Rightarrow} w$ are both possible, it follows that $uv^i wx^i y$ also belongs to L.

This completes the proof of all parts of Lemma.

3.3.4 Usage of Pumping Lemma

The Pumping Lemma can be used to show that certain languages are not context free.

Let us show that the language

$$L = \{a^i b^i c^i \mid i > 0\}$$

is not context-free.

Proof: Suppose L is a context-free language.

If string $X \in L$, where $| X | > m$, it follows that $X = uvwxy$, where $| vwx | \leq m$.

Choose a value i that is greater than m. Then, wherever vwx occurs in the string $a^i b^i c^i$, it cannot contain more than two distinct letters it can be all a's, all b's, all c's, or it can be a's and b's, or it can be b's and c's.

Therefore the string vx cannot contain more than two distinct letters; but by the "Pumping Lemma" it cannot be empty, either, so it must contain at least one letter.

Now we are ready to "pump".

Since $uvwxy$ is in L, $uv^2 wx^2 y$ must also be in L. Since v and x can't both be empty,

$$| uv^2 wx^2 y| > | uvwxy|,$$

so we have added letters.

Both since vx does not contain all three distinct letters, we cannot have added the same number of each letter.

Therefore, uv^2wx^2y cannot be in L.

Thus we have arrived at a "contradiction".

Chapter 4

Turing Machines

4.1 TURING MACHINE MODEL

4.1.1 What is a Turing Machine?

A Turing Machine is like a Pushdown Automaton. Both have a finite-state machine as a central component, both have additional storage.

A Pushdown Automaton uses a "stack" for storage whereas a Turing Machine usea a "tape", which is actually infinite in both the directions. The tape consists of a series of "squares", each of which can hold a single symbol. The "tape-head", or "read-write head", can read a symbol from the tape, write a symbol to the tape and move one square in either direction.

There are two kinds of Turing Machine available.

(a) Deterministic Turing Machine.
(b) Non-deterministic Turing Machine.

We will discuss about Deterministic Machines only. A Turing Machine does not read "input", unlike the other automata. Instead, there are usually symbols on the tape before the Turing Machine begins, the Turing Machine might read some. all, or none of these symbols. The initial tape may, if desired, be thought of as "input".

"Acceptors" produce only a binary (accept/reject) output. "Transducers" can produce more complicated results. So far all our previous discussions were only with acceptors. A Turing Machine also accepts or rejects its input. The results left on the tape when the Turing Machine finshes can be regarded as the "output" of the computation. Therefore a Turing Machine is a "Transducer".

4.1.2 Definition of Turing Machines

A Turing Machine M is a 7-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, \#, F)$$

where Q is a set of states
Σ is a finite set of symbols, "input alphabet".
Γ is a finite set of symbols, "tape alphabet".
δ is the partial transition function

$\# \in T$ is a symbol called 'blank'
$q_0 \in Q$ is the initial state
$F \subseteq Q$ is a set of final states

As the Turing machine will have to be able to find its input, and to know when it has processed all of that input, we require:

(a) The tape is initially "blank" (every symbol is #) except possibly for a finite, contiguous sequence of symbols.
(b) If there are initially nonblank symbols on the tape, the tape head is initially positioned on one of them.

This emphasises the fact that the "input" viz., the non-blank symbols on the tape does not contain #.

4.1.3 Transition Function, Instantaneous Description and Moves

The "*Transition Function*" for Turing Machine is given by

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

When the machine is in a given state (Q) and reads a given symbol (Γ) from the tape, it replaces the symbol on the tape with some other symbol (Γ), goes to some other state (Q), and moves the tape head one square left (L) or right (R).

An "*Instantaneous Description*" or "*Configuration*" of a Turing machine requires.

(a) the state the Turing machine is in
(b) the contents of the tape
(c) the position of the tape head on the tape.

This is written as a string of the form

$$x_i \ldots\ldots x_j q_m x_k \ldots\ldots x_l$$

where the x's are the symbols on the tape, q_m is the current state, and the tape head is on the square containing x_k (the symbol immediately following q_m).

The "*Move*" of a Turing machine can therefore be expressed as a pair of instantaneous descriptions, separated by a symbol "$\vdash$".
For example, if

$$\delta(q_5, b) = (q_8, c, R)$$

then a possible move can be

$$abbabq_5babb \vdash abbabcq_8abb$$

4.1.4 Programming a Turing Machine

As we have the "productions" as the control theme of a grammar, the "transitions" are the central theme of a Turing machine. These transitions are given as a table or list of S-tuples, where each tuple has the form

(current state, symbol read, symbol written, direction, next state)

Creating such a list is called "programming" a Turing machine.

A Turing machine is often defined to start with the read head positioned over the first (leftmost) input symbol. This is not really necessary, because if the Turing machine starts anywhere on the nonblank portion of the tape, it is simple to get to the first input symbol.

For the input alphabet $\Sigma = \{a, b\}$, the following program fragment does the trick, then goes to state q_1.

$$(q_0, a, a, L, q_0)$$
$$(q_0, b, b, L, q_0)$$
$$(q_0, \#, \#, R, q_1)$$

4.1.5 Turing Machines as Acceptors

A Turing machine halts when it no longer has available moves. If it halts in a final state, it accepts its input, otherwise it rejects its input.

A Turing machine $T = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ accepts a language L(M), where

$$L(M) = (w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_i q_f x_j \text{ for some } q_f \in F, x_i, x_j \in \Gamma^*),$$

with the assumption that the Turing machine starts with its tape head positioned on the leftmost symbol.

A Turing Machine accepts its input if it halts in a final state. There are two ways this could fail to happen:

(a) The Turing machine could halt in a nonfinal state or
(b) The Turing machine could never stop i.e., it enters an "infinite loop".

4.1.6 How to Recognize a Language

This machine will match strings of the form

$$\{a^n b^n : n \geq 0\}$$

q_1 is the only "final state".
q_4 (which has no available moves at all) serves as an "error state".

Current state	Symbol read	Symbol written	Direction	Next state
Find the left end of the input				
q_0	a	a	L	q_0
q_0	b	b	L	q_0
q_0	#	#	R	q_1
If leftmost symbol is "*a*", erase it, if "*b*" fail				
q_1	a	#	R	q_2
q_1	b	#	R	q_4
Find the right end of the input				
q_2	a	a	R	q_2
q_2	b	b	R	q_2
q_2	#	#	L	q_3
Erase the "*b*" at the left end of the input				
q_3	b	#	L	q_0

The basic operation of this machine is a loop:

```
q0: move all the way to the left
q1: erase on 'a'
q2: move all the way to the right
q3: Erase a 'b'
Repeat
```

If the string is not of the form $\{a^n b^n : n \geq 0\}$, it will finally either

(a) See an '*a*' in nonfinal state q_3, and halt, or
(b) see a '*b*' in final state q_1, move to nonfinal state q_4, and halt.

4.1.7 Turing Machines as Transducers

To use a Turing machine as a transducer, treat the entire nonblank portion of the initial tape as input, and treat the entire nonblank portion of the tape when the machine halts as output.

A Turing machine defines a function $y = f(x)$ for strings $x, y \in \Sigma^*$ if

$$q_0 x \overset{*}{\vdash} q_f y$$

where q_f is the final state.