2 Finite State Machines

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Concept and notations of finite state machine (FSM)
- Formalism of FSM as finite automata (FA)
- FA as language acceptor (or recognizer)
- Deterministic finite automata (DFA) and non-deterministic finite automata (NFA)
- Equivalence of NFA and DFA
- DFA minimization
- NFA with ϵ-transitions
- Equivalence of NFA with ϵ-transitions, NFA, and DFA
- Moore and Mealy machines and their equivalence
- Moore's algorithm for FSM equivalence
- Properties and limitations of FSM
- Two-way finite automata

2.1 INTRODUCTION

The term finite state machine (FSM) is used for all such programs that have a finite number of functions, but do not have any memory for storing the intermediate results. It is a simple and primitive computational model, which has many applications, but limited power due to lack of memory.

The term 'machine' is used throughout this book to refer to predictable programs, whose behaviour can be understood without executing them. The term 'state' is typically used for different functions that are constituents of a program. A 'program', here, is defined as a collection of unique functions, each one performing an atomic and unique task. A program is said to be completely executed when the task of each function is performed one by one, in order, until the last.

2.1.1 Concept of Basic Machine

As we know, every machine ideally takes an input and produces an output. A basic machine recognizes an input from an input set, I, and produces an output from an output set, O, where I and O are finite sets. This definition is more like an abstraction of any machine or program: Here, we are not interested in what is happening inside the machine, that is, *how* the output is produced by the machine; we are only interested in *what* output it is producing. Effectively, such abstraction of machine behaviour is more like representing the machine as a mapping function between the input and the output sets, showing the output that is generated after providing a certain input.

A basic machine can be viewed as a function, which maps the input set, I, to the output set, O. This function is known as a machine function (MAF); it is expected to be an 'onto' relation:

$$\text{MAF: } I \rightarrow O$$

Let us look at a few examples of a basic machine:

1. All logic gates, such as AND, and OR, can be viewed as basic machines. For example, let us consider the AND gate where,

   $I = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$
   $O = \{0, 1\}$

   Table 2.1 shows the MAF for the aforementioned AND gate.

   Table 2.1 MAF for AND gate

I	(0, 0)	(0, 1)	(1, 0)	(1, 1)
O	0	0	0	1

   Here, we observe that the input is a combination of multiple input symbols. Such a machine is also called a 'combinational machine'.
2. A decimal-to-binary converter can be treated as a basic machine having the following inputs and outputs:

   $I = \{0, 1, 2, \ldots\}$
   $O = \{000, 001, 010, \ldots\}$

3. A weighing machine that we normally see at railway stations or bus stands is also a good example of a basic machine. Here the output is the weight ticket, which is obtained after inserting a coin:

   $I = \{\text{coin}\}$
   $O = \{\text{printed weight ticket}\}$

4. Electrical appliances, such as electric fans, are basic machines with regulator positions as the input set, and the speed in revolutions per minute (rpm) as the output set:

   $I = \{\text{pos1, pos2, pos3, pos4, pos5}\}$
   $O = \{\text{speed A, speed B, speed C, speed D, speed E}\}$

In all the aforementioned examples, we see that there is an in input, which, after going into the machine, gives a particular output. Thus, ignoring the internal details and concerning only with inputs and outputs, every machine can be viewed as a basic machine. A basic machine performs only a table look-up from a finite-sized table called the MAF table. This table has neither memory nor internal states.

It is impossible to create a virtual table that can store infinite word sets in a tabular form. Let us consider a basic machine that produces output in the form of 'yes' or 'no' to check whether a given word is from a given infinite language. This is possible only with the help of a machine having internal states; on reading the input, the machine selects a particular path from the initial state to reach the final state, and produces a valid output. Such a machine has a finite number of internal states, and is called an FSM.

2.2 FINITE STATE MACHINE

In case of an FSM, unlike for a basic machine, we are concerned with the machine (or program) behaviour, especially, regarding *what* internal states the machine has, and *how* each state behaves on receiving different intermediate inputs.

An FSM is represented by a pair of functions, namely:

Machine function: MAF: $I \times S \rightarrow O$
State transition function: STF: $I \times S \rightarrow S$

where, S: Finite set of internal states of the machine
I: Finite set of input symbols (or input alphabet)
O: Finite set of output symbols (or output alphabet)

If we know the current state of the machine and the current input symbol, the MAF tells us the (intermediate) output; and the STF indicates the next state of the machine. Both these functions collectively define an FSM. The behaviour of such a machine can be completely determined (or predicted) provided its initial state and the input are known.

2.2.1 Examples

Let us try to understand the concept through a few simple examples:

Example 2.1 Construct a binary adder as a finite state machine.

Solution We need to construct two tables, namely MAF and STF, to represent the working of a binary adder.

The input, being in binary form, consists of two symbols, either 0, or 1. For a binary adder, the input set, I, is a combination of the two input symbols, that is, a pair of input symbols, as shown here:

$$I = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$$

A finite set of states, S, is defined as:

$$S = \{\text{carry, no carry}\}.$$

Initially, the machine is always in the 'no carry' state; and at any given time, the machine may be in one of the two states, that is, 'carry' or 'no carry', depending on the result of the previous addition.

Let us consider a situation in which the current state of the machine is 'carry' and the current input is '(0, 0)'. Then the output will be 0 + 0 + 1 (carry) = 1, and the machine moves to the next state, which will be a 'no carry' state. Similarly, if the current state of the machine is 'no carry' and the input is '(1, 1)', then the output will be 1 + 1 + 0 (no carry) = 0. The machine again moves to the next state, that is, the 'carry' state.

Thus, the addition of the two input symbols at any given point depends not only on the current input, but also on the current state of the machine. For example, '(0 + 0)' is not always 0; it can also be 1, if the machine is in the 'carry' state, as we have already seen.

The MAF and STF for the binary adder are shown in Table 2.2.

Table 2.2 MAF and STF for binary adder (a) MAF: $I \times S \rightarrow O$ (b) STF: $I \times S \rightarrow S$

I \ S	Carry	No carry
(0, 0)	1	0
(0, 1)	0	1
(1, 0)	0	1
(1, 1)	1	0

Outputs

(a)

I \ S	Carry	No carry
(0, 0)	No carry	No carry
(0, 1)	carry	No carry
(1, 0)	carry	No carry
(1, 1)	carry	carry

Next states

(b)

Let us simulate the working of the binary adder that we have designed for adding two numbers: 1011 and 1111. The initial state will always be 'no carry' when we begin the addition. The simulation is shown in Fig. 2.1.

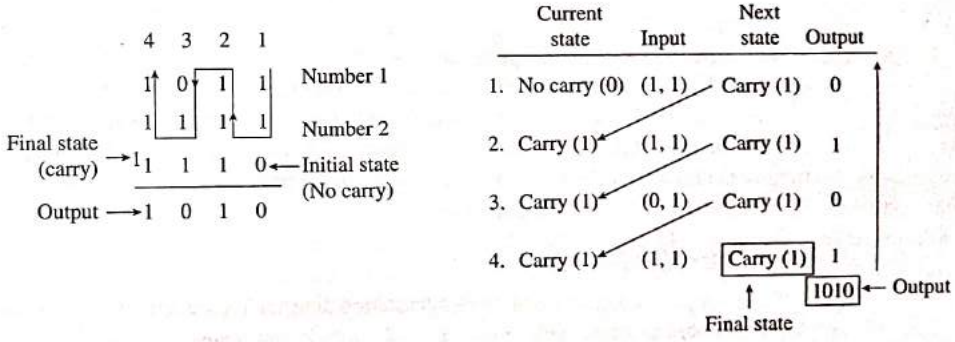


Figure 2.1 Binary adder simulation for input '1011 + 1111'

The two input binary numbers considered for simulation in Fig. 2.1 are 1011 and 1111. The first pair of input symbols is (1, 1), which is obtained from the least significant bits (right-most bits) of both the binary input numbers. 'No carry' is the initial state, or the

beginning state. Upon reading the input (1, 1), the FSM makes a transition from 'no carry' state to 'carry' state, and generates an intermediate output, 0. Then, it reads the next pairs of digits—(1, 1), (0, 1), and (1, 1)—and transits at every stage, generating some intermediate output. Thus, the addition of the two input binary numbers yields the output '1010' with carry. Figure 2.1 explains each step in detail.

A more convenient representation of the FSM is possible using a transition diagram instead of state tables.

2.2.2 Transition Diagram (or Transition Graph)

A transition diagram, also known as transition graph, is a directed graph (or digraph), whose vertices correspond to the states of an FSM and the directed edges correspond to the transitions from one state to another on reading the input symbol, which is written above the directed edge.

Sometimes, the input symbol is also followed by a punctuation character, such as '/' or '→', followed by the output symbol. For example, '(0, 0) → 1'; this can also be written as '(0, 0) / 1'. Figure 2.2 shows the transition graph for the binary adder constructed in Example 2.1.

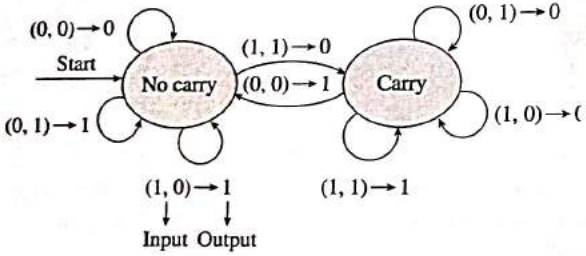


Figure 2.2 Transition graph (TG) for binary adder

The initial state, or 'Start' state, is normally represented by an arrow, pointing towards the state. Many times, the word 'Start' is also written above this arrow.

Each transition, that is, each arc of the graph, gives information about the output symbol as well the next state, if we know the current input and the current state. Thus, it is easier to understand the working of a machine with the help of this diagram, rather than looking at two different tables simultaneously.

2.2.3 Transition Matrix

Although representing an FSM using a transition diagram is convenient, it becomes increasingly complex to draw such diagrams as the number of states increases. The alternative, in such cases, is to use a transition matrix.

A transition matrix has its rows and columns labelled by the states from the set, S. Along the intersection of the ith row and the jth column in the matrix, there is an input symbol from the input set, I, that causes the transition from the current state S_i to the next new state S_j, followed by a punctuation, '/', and the output symbol from the output set, O.

In order to represent a transition from state S_i to S_j for more than one input symbol, the 'logical OR' symbol, that is, the 'V' symbol, is used; and if there is no transition from state S_i to state S_j, then the symbol, '—', is placed for that entry in the matrix.

The transition matrix for the binary adder in Example 2.1 is shown in Table 2.3.

Table 2.3 Transition matrix for binary adder

Current state \ Next state	No carry	Carry
No carry	(0, 1)/1 V (0, 0)/0 V (1, 0)/1	(1, 1)/0
Carry	(0, 0)/1	(0, 1)/ 0 V (1, 0)/0 V (1, 1)/1

Example 2.2 Design an FSM for a divisibility-by-3 tester for decimal numbers.

Solution Decimal numbers are base-10 numerals containing digits from 0 to 9. Hence, the input set, I, is:

$$I = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

From this set, digits 0, 3, 6, and 9 are of same type, that is, they are all divisible by 3. Similarly, digits 1, 4, and 7 generate remainder 1, when divided by 3; and digits 2, 5, and 8 generate remainder 2 when divided by 3.

Based on this similarity, we can group the digits together, and the resultant set of inputs will effectively consist of only three different classes of inputs:

$$I = \{(0, 3, 6, 9), (1, 4, 7), (2, 5, 8)\}$$

Let us consider the possible outputs: If the number is divisible by 3, the output is '1', which means yes, it is divisible; and if the number is not divisible by 3, then the output is '0', which implies no, it is not divisible.

Hence, the output set is:

$$O = \{0, 1\}$$

When we divide a number by 3, it is possible to get three different remainder values: 0, 1, and 2. If we get 0 as the remainder, then the decimal number is divisible by 3, and the machine produces output 1 (yes, it is divisible); and if the remainder is either 1 or 2, the machine produces output 0 (no, it is not divisible).

Depending on these remainder values, we can have three different states for the machine:

$$S = \{S_0, S_1, S_2\}$$

where, S_0: Zero-remainder state
S_1: One-remainder state
S_2: Two-remainder state

The state tables for the divisibility-by-3 tester are shown in Table 2.4.

Table 2.4 State tables for the divisibility-by-3 tester (a) MAF: $I \times S \rightarrow O$ (b) STF: $I \times S \rightarrow S$

S \ I	(0, 3, 6, 9)	(1, 4, 7)	(2, 5, 8)
S_0	1	0	0
S_1	0	0	1
S_2	0	1	0

(a)

S \ I	(0, 3, 6, 9)	(1, 4, 7)	(2, 5, 8)
S_0	S_0	S_1	S_2
S_1	S_1	S_2	S_0
S_2	S_2	S_0	S_1

(b)

At every step in the division of any multi-digit decimal input, the remainder from the previous division step is concatenated to the next input digit to form the number to be considered for division in the next step. For example, if the machine is in state S_1, that is, one-remainder state, and the input is (1, 4, 7)—that is, either 1, or 4, or 7—then the next step considers for division either '11', or '14', or '17', respectively. Now, if we divide these numbers by 3, we get remainder 2. Hence, the machine moves from state S_1 to state S_2 with output '0'. This output indicates that the number '11', '14', or '17', as the case may be, is not divisible by 3.

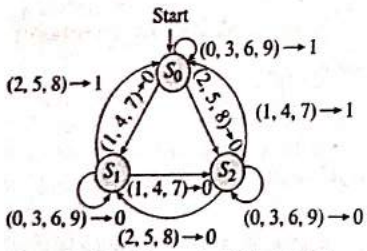


Figure 2.3 Transition graph (TG) for divisibility-by-3 tester

Similarly, if the machine is in state S_2, and the input is (1, 4, 7), then the numbers formed are respectively '21', '24', or '27', which are divisible by 3. Therefore, the machine moves to zero-remainder state, that is, state S_0, with output '1', meaning that the number is divisible by 3. The transition diagram for the machine is shown in Fig. 2.3.

We see that S_0—the zero-remainder state—is the initial state. The final state could be either S_0, S_1, or S_2, depending on the input number.

Now, let us simulate the working of the machine that we have designed on two arbitrary numbers, '112' and '1416', as shown in Fig. 2.4.

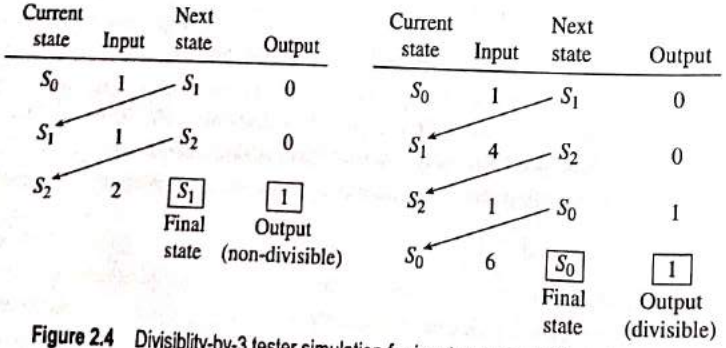


Figure 2.4 Divisiblity-by-3 tester simulation for input numbers '112' and '1416'

The transition matrix for the divisibility-by-3 tester is shown in Table 2.5.

Table 2.5 Transition matrix for divisibility-by-3 tester

Current state \ Next state	S_0	S_1	S_2
S_0	0/1 V 3/1 V 6/1 V 9/1	1/0 V 4/0 V 7/0	2/0 V 5/0 V 8/0
S_1	2/1 V 5/1 V 8/1	0/0 V 3/0 V 6/0	1/0 V 4/0 V 7/0
S_2	1/1 V 4/1 V 7/1	2/0 V 5/0 V 8/0	0/0 V 3/0 V 6/0 V 9/0

Example 2.3 Design an FSM for divisibility-by-5 tester for decimal numbers.

Solution We can use an approach similar to the one in Example 2.2. In this case, it is possible to have five different remainder values: 0, 1, 2, 3, and 4, as we are dividing by 5. If we do not want to know whether machine is in state S_1, S_2, S_3, or S_4 after reading the input number, that is, if we do not want to know about the exact value of the remainder, but only want an answer in the form of 'yes' (it is divisible), or 'no' (it is not divisible), then we require only two states:

$$S = \{q_0 \text{(divisible)}, q_1 \text{(not divisible)}\}$$

We know that if any decimal number ends in 0 or 5, then it is divisible by 5; else it is not divisible by 5. Depending on this fact, we can group the input digits into two categories: (0, 5) and (1, 2, 3, 4, 6, 7, 8, 9) Therefore, the input set, I, and the state set, S, are:

$$I = \{(0, 5), (1, 2, 3, 4, 6, 7, 8, 9)\}$$
$$S = \{q_0, q_1\}$$

In this example, we are not interested in the different remainder values. Hence, when the remainder is either 1, 2, 3, or 4, that is, when the number is not divisible by 5, the machine will lie in state q_1; and if the remainder is 0, that is, the number is divisible by 5, then it will lie in state q_0. However, the initial state of the machine is always q_0.

The STF and MAF tables for the divisibility-by-5 tester are shown in Table 2.6.

Table 2.6 STF and MAF tables for the divisibility-by-5 tester (a) MAF: $I \times S \rightarrow 0$ (b) STF: $I \times S \rightarrow S$

S \ I	(0, 5)	(1, 2, 3, 4, 6, 7, 8, 9)
q_0	1	0
q_1	1	0

(a)

S \ I	(0, 5)	(1, 2, 3, 4, 6, 7, 8, 9)
q_0	q_0	q_1
q_1	q_0	q_1

(b)

The transition diagram is shown in Fig. 2.5.

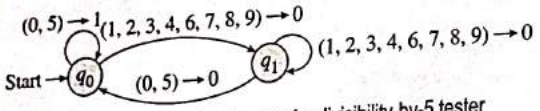


Figure 2.5 Transition diagram for divisibility-by-5 tester

The transition matrix for the same is shown in Table 2.7.

Table 2.7 Transition matrix for divisibility-by-5 tester

Current state \ Next state	q_0	q_1
q_0	(0, 5)/1	(1, 2, 3, 4, 6, 7, 8, 9)/0
q_1	(0, 5)/1	(1, 2, 3, 4, 6, 7, 8, 9)/0

Example 2.4 Design an FSM for divisibility-by-2 tester for unary numbers.

Solution Unary numbers use a single letter or digit to represent a number. For example, decimal number 5 can be represented in unary form as '11111', or 'aaaaa', or 'bbbbb', or '00000', and so on, depending on what letter or digit we choose to represent.

Let us consider the case: $\Sigma = \{a\}$, which means that we choose the letter 'a' to represent the numbers. Hence, the set of all non-zero positive integers over Σ can be represented as {a, aa, aaa, aaaa, ...}.

Now, if we divide a number by 2, the possible remainder values are 0, which means it is divisible; or 1, which means it is not divisible by 2. Therefore, the set of states can be represented as:

$$S = \{p \text{ (divisible)}, q \text{ (non-divisible)}\}$$

As we are going to represent unary numbers using symbol 'a', the set of input symbols is:

$$I = \{a\}$$

Table 2.8 STF and MAF tables for the divisibility-by-2 tester (a) MAF: $I \times S \rightarrow O$ (b) STF: $I \times S \rightarrow S$

S \ I	a
p	0
q	1

(a)

S \ I	a
p	q
q	p

(b)

Further, as discussed in the previous examples, the set of output symbols is:

$$O = \{0 \text{ (not divisible)}, 1 \text{ (divisible)}\}$$

The STF and MAF tables for the divisibility-by-2 tester for unary numbers are shown in Table 2.8.

The initial state of the machine is p, which is the zero-remainder state. If we get one 'a', the machine moves from state p to state q, that is, the one-remainder state. Next, in state q, if it gets one more 'a', then the number becomes 2, which is divisible by 2. Therefore, the machine moves to state p, and gives the output as 1.

The transition diagram for this machine is shown in Fig. 2.6, and the transition matrix is shown in Table 2.9.

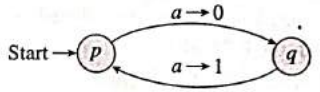


Figure 2.6 Transition diagram for divisibility-by-2 tester for unary numbers

Table 2.9 Transition matrix for divisibility-by-2 tester for unary numbers

Current state \ Next state	p	q
p	—	a/0
q	a/1	—

2.3 FINITE AUTOMATA

Finite automaton (FA) is the mathematical model (or formalism) of an FSM. Mathematical models of machines are the abstractions and simplifications of how certain machines actually work. They help us understand specific properties of these machines. An FA portrays the FSM as a language acceptor. The formal definition of an FA is as follows:

Formal Definition

FA is denoted by a five-tuple:

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

where,

Q: Finite set of states
Σ: Finite input alphabet
q_0: Initial state of FA, $q_0 \in Q$
F: Set of final states, $F \subseteq Q$
δ: STF that maps $Q \times \Sigma$ to Q, i.e., δ: $Q \times \Sigma \rightarrow Q$

The aforementioned definition of the transition function, δ, is for deterministic FA, or DFA. The transition function for non-deterministic FA, or NFA, is different and is discussed later in this chapter.

As the definition is formalized, we only need to get familiar with the standard names or symbols that are used. A few notational differences between FA and FSM that we can identify are: Q is analogous to state set, S; Σ is analogous to input set, I; and the state transition function, 'δ: $Q \times \Sigma \rightarrow Q$', is analogous to 'STF: $I \times S \rightarrow S$'.

We observe that the formal definition for FA does not include output set, O; therefore, this formalism may be called FA without output. We further observe that FA, instead, has final states that can be reached only when the input is acceptable (or valid).

Any subset of Q can be marked as the set of final states, F, depending on the solution. Upon reading the entire input string, if the machine resides in any of the final states, then the

input string is considered to be 'accepted' by the machine. On the other hand, if the machine resides in any of the non-final states, then the input read is considered to be 'rejected' by the machine. Conceptually, this is equivalent to generating the output 'true' if valid and 'false' if otherwise—more like a Boolean function. Thus, FA acts as an input acceptor or rejecter.

Usually, in the transition graph of the problem solution, rejection is not explicitly shown. The paths in the transition graph, which are unspecified, are considered as rejection paths. At times, these can be explicitly shown with the help of non-accepting 'trap states'. This will be dealt with in greater detail while discussing the examples.

In case of DFA, neither for state q in Q, nor for the input symbol 'a' in Σ, does '$\delta(q, a)$' contain more than one element. Thus, the transition would be of the form:

$$\delta(q, a) = p$$

where, p is the unique next state in Q, to which the machine makes the transition (p may or may not be equal to q). This means that if the DFA is in state q and reads a symbol 'a', then it moves to the unique next state 'p'.

For example, let us consider the FA:

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

Here, q_0 is the initial state as well as the final state. Hence, the set of all states, Q, is:

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

The set of input symbols, or input alphabet, is:

$$\Sigma = \{0, 1\}$$

The transition function, δ, will be as shown in Table 2.10.

Table 2.10 Example state transition table (δ: $Q \times \Sigma \rightarrow Q$)

Q \ Σ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

The aforementioned FA is the automata that accepts all strings containing even number of 0's and even number of 1's over the input alphabet, $\Sigma = \{0, 1\}$.

2.3.1 Transition Graph

The transition graph for FA is almost the same as that for FSM, as we have seen earlier in Section 2.2.2. The only difference is that in case of FA, we need to represent the final states differently, and we do not have any output symbols. The final states are represented by two concentric circles, instead of a single circle that is used to represent the normal (or non-final) states.

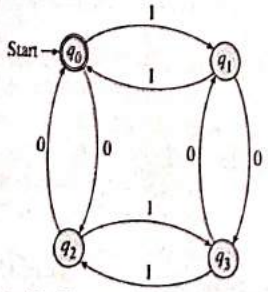


Figure 2.7 Transition graph for the example FA

The transition diagram for FA, whose transition table is shown in Table 2.10, is shown in Fig. 2.7.

We observe that the transitions, that is, the directed edges in the graph are labelled only using the input symbols, as there are no output symbols to be considered. Here, q_0 is the initial state and is represented by an arrow, which has the string 'start' associated with it. Besides, q_0 is also the final state, so it is indicated by two concentric circles; we observe that q_1, q_2, and q_3 are non-final states, and are hence represented by single circles (refer to 'notation (1)' in Fig. 2.8).

There is a slightly different notation that is used at times for representing the transition graph for an FA. The additional notation—notation (2)—for each representation is shown in Fig. 2.8.

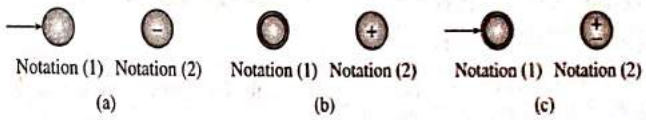


Figure 2.8 Some additional notations for transition graph of FA (a) Initial state (b) Final state (c) If initial and final states are same

This figure explains the different types of notations used to denote the initial and final states. According to notation (2) in Fig. 2.8(a), the initial state can be denoted by a symbol, '−' (minus sign), inside the state circle, whereas according to notation (1), it is represented by an arrow pointing to the state circle along with the string 'start' attached to it. The final state is denoted by a symbol, '+' (plus sign), inside a state circle according to notation (2) in Fig. 2.8(b), whereas according to notation (1), it is represented by two concentric circles. If initial state is also a final state, then we write a '±' sign inside the state circle in notation (2) in Fig. 2.8(c), while according to notation (1), this is represented by an arrow pointing towards two concentric circles.

2.3.2 Functions

The example FA in Fig. 2.7 accepts only the strings over $\Sigma = \{0, 1\}$ that contain even number of 0's and even number of 1's. All these acceptable strings can be aggregated into an infinite set or language L defined as:

L = {set of all strings over $\Sigma = \{0, 1\}$ containing even number of 0's and even number of 1's}

A conclusion that can be drawn from the aforementioned finding is that though the language L is infinite, we can construct a machine (FA) that can accept any string from the language L and reject any other string which is not part of the language L.

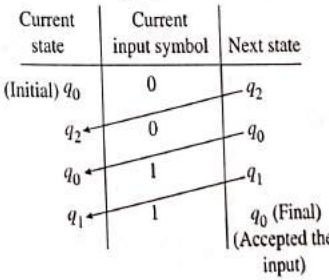


Figure 2.9 Example FA accepting the input '0011'

In other words, the FA constructed for a given language checks the validity of the strings from the language. It checks whether the input string is a part of the language or not. If the input string is part of the language, the FA accepts it by making a transition to the final state after reading the string. Otherwise, it resides in one of the non-final states, indicating that it has rejected the string.

Let us simulate the working of the FA represented in Fig. 2.7 for the input string '0011' (refer to Fig. 2.9).

Current state	Current input symbol	Next state
q_0 (Initial)	0	q_2
q_2	1	q_3
q_3	0	q_1
q_1	1	q_0
q_0	1	q_1 (Non-final state) (Input rejected)

Figure 2.10 Example FA rejecting the input '01011'

Input string '0011' contains even number of 0's as well as even number of 1's. Hence, the input string is accepted by the FA, which is indicated by the final state, q_0, in which the machine resides after reading the string '0011'.

Let us now simulate the working of FA for the input '01011'; we see that this is not a valid string as it does not contain even number of 1's (refer to Fig. 2.10).

We see that the input string '01011' is rejected by the FA as it resides in the non-final state, q_1, after reading the string.

Thus, the FA is used as a language acceptor. It only relies on the transitions based on the input symbols, and these transitions are based on the input patterns only. Hence, the FA can accept an infinite language as well.

2.3.3 Acceptance of a String

A string 'x' is said to be accepted by an FA given by:

$$M = (Q, \Sigma, \delta, q_0, F),$$
$$\text{if } \delta(q_0, x) = p,$$

for some $p \in F$ (i.e., p is a member of F)

As we know, δ is a state transition function that maps $Q \times \Sigma$ to Q. Thus, each individual transition can be denoted by:

$$\delta(q_0, a_1) = q_1$$

where, 'q_0' is the current state, 'a_1' is the current input symbol, and 'q_1' is the next state.

Similarly, we can have another transition represented as:

$$\delta(q_1, a_2) = q_2$$

We may combine these two as:

$$\delta(q_0, a_1 a_2) = \delta(\delta(q_0, a_1), a_2) = \delta(q_1, a_2) = q_2$$

This means that if the current state is q_0, then after reading string '$a_1 a_2$', the state reached is q_2.

Similarly, the definition, '$\delta(q_0, x) = p$' means that after reading string 'x' symbol by symbol (one symbol at a time), the machine reaches state p. If this state p is a final state, that is, if p is a member of the set F, then 'x' is accepted by the automata M; else if $p \notin F$, then it is rejected by M.

2.3.4 Acceptance of a Language

A language is a set of strings over some alphabet. If there is a language L such that:

$$L = \{x \mid \delta(q_0, x) = p, \text{ for some } p \in F\},$$

then it is said to be accepted by the FA, M, and is denoted by $L(M)$. In such a case the language accepted by automata M is also called a 'regular set' or 'regular language'.

If all the strings 'x' of a language are accepted by an FA, then the language is said to be accepted by the FA, that is, for every string 'x', $\delta(q_0, x)$ makes the FA move to the final state, after reading all symbols in 'x'.

2.3.5 Some Examples of FA as Acceptor

In this section, let us look at some examples, where FA acts as an acceptor or recognizer.

Example 2.5 Design an FA that reads strings made up of the letters in the word 'CHARIOT' and recognizes those strings that contain the word 'CAT' as a substring.

Solution Here the alphabet Σ consists of seven symbols:

$$\Sigma = \{C, H, A, R, I, O, T\}$$

We need to consider all possible strings made up of these seven symbols and design an FA, which would recognize only those which contain 'CAT' as a substring. In turn, we can say that the FA we are going to design would make a transition to the final state only if the input string contains 'CAT' as a substring.

The working of the FA will be as follows:

If the FA reads the symbol 'C' while in the start state, that is, q_0, then it makes a transition to state q_1; otherwise remains in the same state. Now in state q_1, if the FA reads symbol 'A', then it makes a transition to another state, that is, state q_2. Similarly, in q_2, if FA reads symbol 'T', then it makes a transition to q_3, which is the final state.

Essentially, states q_1, q_2, and q_3 respectively indicate that the substrings 'C', 'CA', and 'CAT' have been read.

Figure 2.11 shows the transition graph for the recognizer FA. The FA that is constructed here is actually a sequence detector program, while the sequence it is trying to detect is 'CAT'.

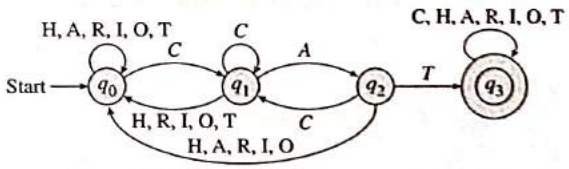


Figure 2.11 Recognizer FA as a sequence detector

We observe that the state q_0 is the start (or initial) state, and q_3 is the final state. If the FA reads any symbol from Σ while in q_3 (final state), it does not make a transition to any other state as it has already recognized the substring 'CAT' by then.

The state transition table for the FA is shown in Table 2.11.

Table 2.11 State transition table for recognizer FA

Q \ Σ	C	H	A	R	I	O	T
q_0	q_1	q_0	q_0	q_0	q_0	q_0	q_0
q_1	q_1	q_0	q_2	q_0	q_0	q_0	q_0
q_2	q_1	q_0	q_0	q_0	q_0	q_0	q_3
q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3

Example 2.6 Design an FA that reads strings made up of {0, 1} and accepts only those strings which end in either '00' or '11'.

Solution Here, the alphabet Σ is given as:

$$\Sigma = \{0, 1\}$$

We need to construct an FA that is capable of reading all possible strings over Σ, but accepts only those strings which end in '00' or '11'. This is yet another example of a sequence detector (refer Fig. 2.12).

The FA has two final states: One that accepts strings ending with '00', that is, state q_2, and the other that accepts strings ending with '11', that is, state q_4.

In Fig. 2.12, states q_1 and q_2 respectively indicate that the substrings '0' and '00' have been read. Similarly, states q_3 and q_4 respectively indicate that the substrings '1' and '11' have been read. The self-loop in state q_2 on symbol '0' indicates that any string that ends in three or more 0's anyway ends with two 0's; thus, once two 0's have been read, more 0's does not change the acceptance criteria—that the string should end in '00'. Similarly, one can explain the self-loop on state q_4 on symbol '1'.

The states q_1 and q_2, upon reading '1' transit to state q_3, indicating that one '1' has been read. Similarly, states q_3 and q_4 transit to state q_1 upon reading the '0', indicating that the string ends with '0'. If q_1 reads one more '0', then it transits to the final state q_2, indicating that the sequence ends with '00'.

The state transition table for the FA in Fig. 2.12 is shown in Table 2.12.

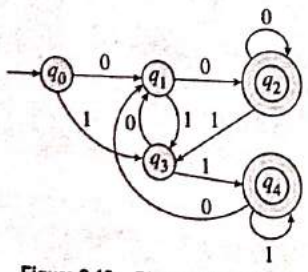


Figure 2.12 FA accepting strings which end in either '00' or '11'

Table 2.12 State transition table for acceptor FA

Q \ Σ	0	1
q_0	q_1	q_3
q_1	q_2	q_3
q_2	q_2	q_3
q_3	q_1	q_4
q_4	q_1	q_4

Let us trace the FA for input, '111'. The sequence of states starting with the initial (start) state q_0 will be: $q_0 \rightarrow q_3 \rightarrow q_4 \rightarrow q_4$. Thus, the last state in the sequence is q_4, which is a final state, and hence, the input '111' is accepted by the FA.

Now, let us trace the input, '0110', starting with the start state q_0. On input '0', the FA makes a transition from state q_0 to state q_1. On reading the next symbol, that is, '1', it changes to state q_3. Similarly, for the third symbol, that is, '1', it switches from state q_3 to state q_4. Though q_4 is a final state, there is one more symbol, that is, '0', that needs to be read; and which is the last symbol of the string '0110'. On reading the last symbol, the FA makes a transition from state q_4 to state q_1. Thus, the sequence of states for the input string '0110' is: $q_0 \rightarrow q_1 \rightarrow q_3 \rightarrow q_4 \rightarrow q_1$. As q_1 is a non-final state, and is the last state of the sequence after reading the input string '0110', the string is not accepted (that is, it is rejected) by the FA.

2.3.6 FA as Finite Control

The FA can be visualized as a finite control with the input string written on a tape, and each tape cell containing one symbol from the string; let '$' denote the end of the string.

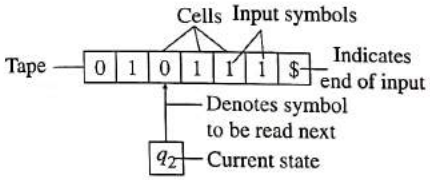


Figure 2.13 Finite control representation of FA

A pointer or head points to a cell on the tape from which the next symbol will be read. The head is labelled by the state label that represents the current state of the machine (refer to Fig. 2.13).

The FA reads symbols one by one from a finite tape, whose end of string is indicated by '$'. Let us say the FA in Fig. 2.12 resides in state q_2 at a given instance, and is about to read the symbol '0' from the tape cell. After reading the symbol, the FA's head will point to the next adjacent cell on the right-hand side, and will transit to the next state (which may or may not be the same as the current state). This means that after reading the current symbol in the cell, the head always moves to the right to read the next symbol. If it reads '$' (indicating the end of input), and at this stage, if the FA is in the final state, then the input string is accepted, otherwise it is rejected.

Theorem 2.1

Every FA can be represented using a transition graph (TG), but not every TG satisfies the definition of an FA.

Proof

Consider the transition graph consisting of only one node as shown in Fig. 2.14(a).

Figure 2.14 Transition graph and FA (a) TG (b) FA

Figure 2.14(a) represents a TG that accepts no string—not even an empty string (ϵ) of zero length—as there is no final state. This is because, in order to be able to accept anything, the FA must have a final state. Hence, the TG in Fig. 2.14(a) cannot be called an FA.

Now, Fig. 2.14(b) represents an FA, which accepts only the empty string, that is, ϵ, because its initial and final states are same—there is only one state. We also observe from the figure that there is no other transition. Therefore, Fig. 2.14(b) represents the TG for an FA accepting only ϵ string.

Hence, the theorem is proved.

2.4 DETERMINISTIC FINITE AUTOMATA

We have already mentioned about the deterministic finite automata (DFA) in Section 2.3. An FA is said to be deterministic if for every state there is a unique input symbol, which takes the state to the required next unique state. This means that given a state S_j, the same input symbol does not cause the FA to move into more than one state—there is always a unique next state for an input symbol (read, for any state transition. Refer to Fig. 2.15).

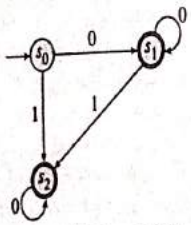


Figure 2.15 Example DFA

Figure 2.15 shows that from state S_0, there is only one transition on reading symbol '0' and symbol '1'. Similarly, for states S_1 and S_2, there are unique transitions on reading symbols '0' and '1'. No state exists from which there is more than one transition for the same input symbol. Therefore, the example FA, represented using Fig. 2.14, is a DFA.

We now see that all the examples we have seen before (i.e., Figures 2.7, 2.11, and 2.12) are DFAs.

2.5 NON-DETERMINISTIC FINITE AUTOMATA

In a non-deterministic finite automata (NFA) model, it is possible to have more than one transition on reading the same input symbol from a given state. Such a machine is not even probabilistic, as no weights are assigned to the different possible transitions from the state for the same symbol; this is also known as a 'possibilistic' machine.

An NFA is denoted by the five-tuple notation:

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

where, Q, Σ, q_0, and F have their usual meanings.

The only change is the state transition function, δ, that maps from $Q \times \Sigma$ to 2^Q:

$$\delta : Q \times \Sigma \to 2^Q$$

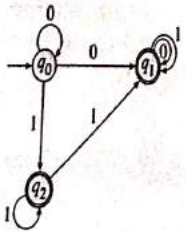


Figure 2.16 Example NFA

For an example, refer to Fig. 2.16. The FA shown in the figure is an NFA. We observe that from state q_0, on reading the input symbol '0', there are two different transitions: first to state q_0, and the other to state q_1. There is no unique next state for the transition on symbol '0' from state q_0. This means that state q_0, on reading symbol '0', can go to either q_1 or to itself, that is, 'q_0'. Hence, it is possibilistic; and so, the behaviour of the NFA, cannot be predicted.

Table 2.13 State transition function, δ, for NFA in Fig. 2.16

Q \ Σ	0	1
q_0	$\{q_0, q_1\}$	q_2
q_1	q_1	q_1
q_2	ϕ	$\{q_1, q_2\}$

The transitions can be represented by the following equation:

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

We observe from Table 2.13 that even from state q_2 there are two different transitions on reading the same input symbol '1', and it is not possible to determine the next state. Therefore, this is called a non-deterministic FA or NFA.

For the example NFA in Fig. 2.16, set of states Q is $\{q_0, q_1, q_2\}$. Hence, the power set of Q, that is, the set of all possible subsets of Q can be expressed as:

$$2^Q = \{\phi, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

where, 'ϕ' is an empty set, as we know.

We observe that $\delta(q_0, 0) = \{q_0, q_1\}$

where, $\{q_0, q_1\}$ is a member of 2^Q.

Now, it is clear why the state transition function for an NFA is:

$$\delta : Q \times \Sigma \to 2^Q$$

Similarly, $\delta(q_2, 1) = \{q_1, q_2\}$, which is also a member of 2^Q.

2.6 EQUIVALENCE OF NFA AND DFA

The NFA and DFA are equivalent to each other; in other words, for every NFA, there exists an equivalent DFA accepting the same set of words (or language). Therefore, the capabilities of an NFA and its equivalent DFA are the same. Hence, NFA and DFA have equal powers.

Let us consider a simple example shown in Fig. 2.17.

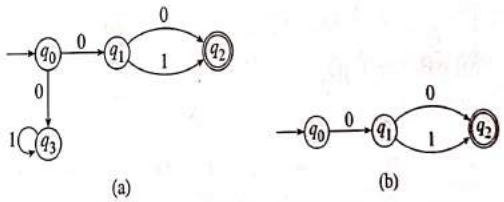


Figure 2.17 NFA vs DFA (a) Example NFA (b) Equivalent DFA

Figure 2.17(a) shows an example NFA, with two different transitions from state q_0 on reading symbol '0'. The string is said to be accepted only if, after reading that string, the machine resides in one of the final states. Therefore, only two strings, '00' and '01' are accepted by the NFA in Fig. 2.17(a). The second path, on reading symbol '0' from state q_0, does not reach the final state anyway.

Thus, the language accepted by the NFA in Fig. 2.17(a) is:

$$L_1 = \{00, 01\}$$

Now, the DFA in Fig. 2.17(b) accepts the language:

$$L_2 = \{00, 01\}$$

We observe that $L_1 = L_2$.

Therefore, the NFA and DFA in Fig. 2.17 are equivalent to each other, and accept the same language {00, 01}. Therefore, their power is same.

Let us now study some methods for converting an NFA to its equivalent DFA.

2.6.1 NFA to DFA Conversion (Method I)

As we know, an NFA can be represented by a five-tuple:

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

where,

$$\delta = Q \times \Sigma \rightarrow 2^Q$$

The main difference between NFA and DFA is the state transition function, δ. The entries in the state transition table for NFA are sets, instead of singular entries as in DFA.

Hence, while converting an NFA into its equivalent DFA, let us consider $Q' = 2^Q$, as the set of states for the resulting DFA.

Then, the state function for the DFA will be:

$$\delta' : Q' \times \Sigma \rightarrow Q'$$

This means that every combination of states can be considered as a new state, and can be given a new label.

For example, for the DFA equivalent to the example NFA in Fig. 2.16, we consider $\{q_2\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$ as three different states: Let us label them as p_0, p_1, and p_2, respectively.

The transitions from these states can be obtained by combining the transitions for the constituent states. Thus, Q' can be represented as:

$$Q' = \{\phi, [q_0], [q_1], [q_2], [q_0, q_1], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$$

The transition $\delta'([q_0, q_1], 1)$ can be obtained as:

$$\delta'([q_0, q_1], 1) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{[q_2] \cup [q_1]\} = \{q_1, q_2\}$$

Thus, we can say that state $\{q_0, q_1\}$ for the equivalent DFA makes transition on reading symbol '1' to state $\{q_1, q_2\}$. If we label these states with the singular labels, as discussed earlier, the equivalent DFA will contain the unique next states for each symbol on which the transition is made.

A combination of states is considered as a final state for the resultant DFA, if at least one of the states constituting the combination is a final state for the given NFA. The initial state remains unchanged for the resultant DFA.

Let us study some examples to explore this further.

Example 2.7 Convert the NFA: $M = [\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\}]$, where δ is as shown in Table 2.14, to its equivalent DFA.

Solution For the given NFA, $Q = \{q_0, q_1\}$, $\Sigma = \{0, 1\}$, $F = \{q_1\}$, and initial state is q_0. Let us denote the resultant DFA as:

$$M' = \{Q', \Sigma, \delta', [q_0], F'\}$$

where,

$\Sigma = \{0, 1\}$ and the initial state, q_0, are the same as that of the given NFA—the entry point cannot be changed if the language accepted is required to be same for equivalence.

Table 2.14 State transition table for example NFA

Q \ Σ	0	1
q_0	$\{q_0, q_1\}$	$\{q_1\}$
q_1	ϕ	$\{q_0, q_1\}$

The power set of Q is given by:

$$2^Q = \{\phi, [q_0], [q_1], [q_0, q_1]\}$$

As per the method discussed, the finite set of states for the resultant (and equivalent) DFA, that is, Q' can be written as:

$$Q' = \{[q_0], [q_1], [q_0, q_1]\}$$

We see that ϕ, is excluded, as it does not denote any state of the given NFA, and hence, is irrelevant.

Now, we need to find the state transition function for the DFA:

$$\delta' : Q' \times \Sigma \rightarrow Q'$$

This means that we need to find all the transitions from every state in Q' on reading both the input symbols '0' and '1'.

From Table 2.14, which shows the state table of the given NFA, we have:

$$\delta'([q_0], 0) = [q_0, q_1]$$
$$\delta'([q_0], 1) = [q_1]$$

$$\delta'([q_1], 0) = \phi \quad \text{(this means no transition)}$$
$$\delta'([q_1], 1) = [q_0, q_1]$$

Now, there is one more state remaining, to determine the transitions, that is, for $[q_0, q_1]$:

$$\delta'([q_0, q_1], 0) = [\delta(q_0, 0) \cup \delta(q_1, 0)] = [\{q_0, q_1\} \cup \phi] = [q_0, q_1]$$
$$\delta'([q_0, q_1], 1) = [\delta(q_0, 1) \cup \delta(q_1, 1)] = [\{q_1\} \cup \{q_0, q_1\}] = [\{q_0, q_1\}] = [q_0, q_1]$$

Table 2.15 State transition table for resultant DFA

Q' \ Σ	0	1
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_1]$	–	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

Using the aforementioned information, the state transition table for resultant DFA is written as shown in Table 2.15.

The transition diagram for the resultant equivalent DFA can be drawn as shown in Fig. 2.18(a).

As q_0 is the initial state of the given NFA, $\{q_0\}$ is the initial state of the resultant DFA. Similarly, 'q_1' is the final state for the given NFA;

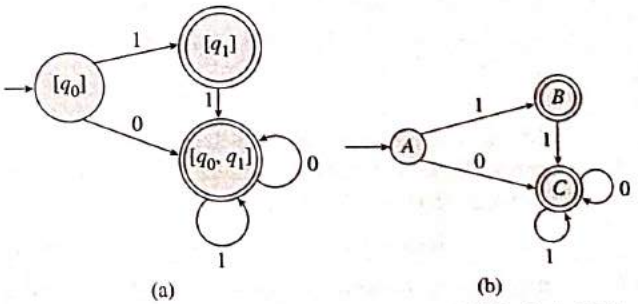


Figure 2.18 DFA equivalent to the NFA in Table 2.14 (a) Equivalent DFA (b) Equivalent DFA after relabelling

hence, for the resultant DFA, we need to consider all such states, which contain q_1 as one of the constituents of the final states.

Therefore, $F' = \{q_1, [q_0, q_1]\}$

The machine necessarily should only have one initial state. However, it can have multiple final states.

Let us now change the labels of the states for the DFA as:

A: $[q_0]$; B: $[q_1]$; and C: $[q_0, q_1]$

Then, the DFA can be redrawn as shown in Fig. 2.18(b). For this DFA after relabelling, we have:

$Q' = \{A, B, C\}$,
$\Sigma = \{0, 1\}$,
initial state = A, and
$F' = \{B, C\}$

Likewise, the state transition function, δ', for the equivalent DFA can now be rewritten as shown in Table 2.16.

Table 2.16 DFA state transition table

Q' \ Σ	0	1
A	C	B
B	–	C
C	C	C

Example 2.8 Convert the NFA $[\{p, q, r, s\}, \{0, 1\}, \delta, p, \{s\}]$ to its equivalent DFA, where the state transition function, δ, is as shown in Table. 2.17.

Solution For the resultant DFA, $\Sigma = \{0, 1\}$, the initial state is also p, and

$$Q' = \{p, q, r, s, pq, pr, ps, qr, qs, rs, pqr, pqs, prs, qrs, pqrs\}$$

Note that we have excluded ϕ (null set) from 2^Q. All the states having 's' as one of the constituents are the final states for the resultant DFA, because 's' is the final state for the given NFA. Therefore, the set of final states for the resultant DFA is given by:

$$F' = \{s, ps, qs, rs, pqs, prs, qrs, pqrs\}$$

Table 2.17 State transition table for example NFA

Q \ Σ	0	1
p	p, q	p
q	r	r
r	s	–
s	s	s

We can relabel the states from Q' by numbers from 1 to 15, in sequence.

For the states p, q, r, and s, the state transition function, δ', can be directly obtained from the given state transition table in Table 2.17.

For example; $\delta'(p, 0) = pq$
$\delta'(q, 1) = r$
$\delta'(s, 1) = s$

For the combination states, which have two or more symbols together representing a state, δ' can be obtained from the union of their respective transitions in the NFA state transition table.

For example:

$$\delta'(pqr, 0) = [\delta(p, 0) \cup \delta(q, 0) \cup \delta(r, 0)] = [\{p, q\} \cup \{r\} \cup \{s\}] = [pqrs]$$

Thus, the entire state transition table for the resultant DFA is as shown in Table 2.18. Observe that along with the normal state symbol, the new labels are also shown in the brackets. The symbol, '*', indicates the final states.

Table 2.18 State transition table for resultant DFA

	Q' \ Σ	0	1
(1)	p	pq (5)	p (1)
(2)	q	r (3)	r (3)
(3)	r	s (4)	–
*(4)	s	s (4)	s (4)
(5)	pq	pqr (11)	pr (6)
(6)	pr	pqs (12)	p (1)
*(7)	ps	pqs (12)	ps (7)
(8)	qr	rs (10)	r (3)
*(9)	qs	rs (10)	rs (10)
*(10)	rs	s (4)	s (4)
(11)	pqr	$pqrs$ (15)	pr (6)
*(12)	pqs	$pqrs$ (15)	prs (13)
*(13)	prs	pqs (12)	ps (7)
*(14)	qrs	rs (10)	rs (10)
*(15)	$pqrs$	$pqrs$ (15)	prs (13)

↑ New labels

'*': Final states

2.6.2 DFA Minimization

The DFA thus obtained can be minimized. In order to minimize the DFA, one needs to identify the equivalent states.

Equivalent States

Two or more states are said to be equivalent states if they undergo the same transitions on reading the same input symbol (true for all the transitions) and are of the same type, that is, either final states or non-final. This means that all these states move to the same next state after reading the same input symbol. Furthermore, they all are of the same type—either final or non-final). Hence, if one or more states have the same transitions, but they are of different types—that is, one is a final transition and the other, a non-final transition—they are not considered as equivalent states.

The equivalent states are used to minimize the DFA: If there is more than one state that is equivalent, we can remove all the equivalent states but one. The one state thus kept replaces all other equivalent states in the DFA. However, there are certain rules that we need to follow.

DFA Minimization Rules

For reducing equivalent states to a single state, one should follow the following rules:

1. We can replace one non-final state by its equivalent non-final state only.
2. We can replace one final state by its equivalent final state only.
3. We *cannot* replace the initial state by any other state.
4. We *cannot* replace one final state by a non-final state, or one non-final state by a final state.
5. Replacing state *A* by state *B* means deleting all entries related to state *A*, that is, all the transitions for state *A* from the state transition table. Furthermore, wherever we find any transition where '*A*' is the next state; we replace it by state *B*; and state *A* can be deleted from the set of states *Q* for the DFA.
6. After minimizing the DFA applying all the aforementioned five steps, if we find any more equivalent states as a result of the reduction, we repeat the same five steps again; else we stop.

For example, in Table 2.18, the states *rs* (10) and *s* (4) both transit to state *s* (4) on reading input '0'; as well as on reading input '1'. This means that they have the same transition on reading both the input symbols. Moreover, they are both final states. Therefore, these two states are equivalent.

Similarly, *prs* (13) is equivalent to *ps* (7); *qrs* (14) is equivalent to *qs* (9); and *pqrs* (15) is equivalent to *pqs* (12).

Therefore, we can replace:

rs (10)	by	*s* (4)
prs (13)	by	*ps* (7)
qrs (14)	by	*qs* (9)
pqrs (15)	by	*pqs* (12)

After these replacements, and now using only the new labels (i.e., numbers only) the reduced transition table for the resultant DFA is as shown in Table 2.19. The symbol '*' represents the final states; and the black circle '•' indicates the modified entries due to replacements.

Table 2.19 Minimized state transition table

Q' \ Σ	0	1
1	5	1
2	3	3
3	4	–
*4	4	4
5	11	6
6	12	1
*7	12	7
8	4 •	3
*9	4 •	4 •
11	12 •	6
*12	12 •	7 •

'*': Final states
'•': Modified entries due to replacement

We now observe from Table 2.19 that state 4 and state 9 are equivalent, and both are final. Similarly, states 7 and 12 are both equivalent and final; this indicates that further reduction is possible. Hence, replacing state 9 by 4; and state 12 by 7, we get a further minimized table, as shown in Table 2.20.

We see that no further minimization of Table 2.20 is possible. Therefore, we can now draw the transition graph for the resultant DFA, as shown in Fig. 2.19(a).

We observe from Fig. 2.19(a) that it is a disconnected graph having two parts: Part I contains one initial and one final state, and part II contains only final state, but no initial state.

As we know, the working of a machine always starts from an initial state; hence, we can directly exclude part II from the transition graph. Therefore, part I of the graph is the final DFA. We can again change the state labels using symbols 'A' to 'E', and the final DFA can be drawn as shown in Fig. 2.19(b).

Table 2.20 Further minimization of the state transition table

Q' \ Σ	0	1
1	5	1
2	3	3
3	4	–
*4	4	4
5	11	6
6	7 •	1
*7	7 •	7
8	4	3
11	7 •	6

'*': Final states
'•': Modified entries due to replacement

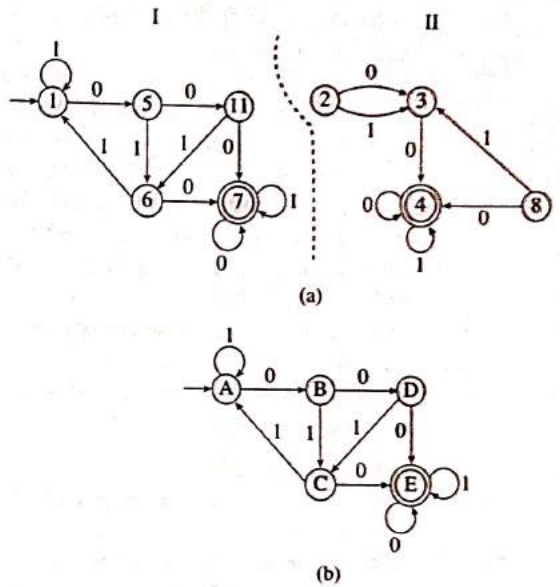


Figure 2.19 Transition graph for the minimized DFA (a) TG for DFA drawn from Table 2.20 (b) Final minimized DFA

Unreachable States

A state is said to be an 'unreachable state' if it cannot be reached from the initial state on reading any input symbol. These unreachable states do not take part in string acceptance, and hence must be removed in order to minimize the DFA; they are like pieces of dead code that never get executed in any run of a computer program. For example, the states labelled 2, 3, 4, and 8 in Fig. 2.19(a) are unreachable states; they make a disconnected graph, as discussed earlier. Hence, these states along with their transitions are removed in order to minimize the DFA.

Dead (or Trap) States

Dead states (or trap states) are those non-final (or non-accepting) states whose transitions for every input symbol terminate on them. This means that these states have no outgoing transitions, except to themselves. These are called trap states because once entered, there is no escape (as in the hang state of a computer program). Moreover, any transition to a dead state becomes undefined. Just as the unreachable states, these dead states and transitions that are incident on these dead states must be removed in order to minimize the DFA. For example, refer to the dead state q_3 in Fig. 2.17(a).

2.6.3 NFA to DFA Conversion (Method II)

In method I, we considered all possible subsets of Q (i.e., 2^Q) as the possible states for the resultant DFA. Later, however, we had to remove many of the state combinations during the minimization process.

Method II, which is described in this section, is much easier and direct—it takes lesser number of steps and time to build an equivalent DFA using this method. Instead of considering the set $Q' = 2^Q$ and then removing the states that are not required, this method considers only those states (or combinations of states) that are required. The effort required for minimization is hence lesser compared to the previous method.

This method directly starts with the transition diagram. Instead of considering all possible states, it starts finding the transitions, one state at a time. If the next state of a given transition is a new combination, then it gets added to the set of states for the resultant DFA. Due to this, we never get a disconnected transition graph as we obtained in method I (refer to Fig. 2.19).

Let us now look at some examples to illustrate this method.

Example 2.9 Convert the NFA: $M = [\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\}]$ where, the state transition function, δ, is given in Table 2.21, to its equivalent DFA.

Solution This is the same example we considered earlier (refer to Example 2.7).

The transition graph for the given NFA is as shown in the Fig. 2.20.

We observe from Fig. 2.20 that state q_0, on reading input '0', makes transition either to state q_0 or state q_1, that is, multiple states. This means that for the equivalent resultant DFA, we need the combination state $[q_0\, q_1]$. Let us therefore create a new state with label '$q_0\, q_1$' as shown in Fig. 2.21(a). As this new state contains the label q_1, which is the final state for the given NFA, we mark this state as the final state for the resultant DFA, as shown—with double concentric circles—in Fig. 2.21(a).

Table 2.21 State transition table for example NFA

Q \ Σ	0	1
q_0	$\{q_0, q_1\}$	$\{q_1\}$
q_1	ϕ	$\{q_0, q_1\}$

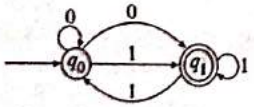


Figure 2.20 TG for example NFA given in Table 2.21

State q_0 on reading input '1' goes to only 'q_1', therefore, we add one more state to the resultant DFA labelled 'q_1', which is also a final state (refer to Fig. 2.21b).

Since there is no transition from state q_1 on reading input '0' for the given NFA, therefore, no transition from state q_1 is labelled with a '0'.

However, state q_1, on reading input '1', goes either to state q_0 or state q_1; therefore, we show a transition from state q_1 to state $q_0\, q_1$. Since, state $q_0 q_1$ already exists as a part of the resultant DFA, we do not need to create a new state (refer to Fig. 2.21c).

Now, there is only one state $q_0\, q_1$ for which the transitions need to be realized:

$$\begin{aligned}\delta'(q_0 q_1, 0) &= \delta(q_0, 0) \cup \delta(q_1, 0)\\ &= \{q_0, q_1\} \cup \phi\\ &= q_0\, q_1\end{aligned}$$

Similarly,

$$\begin{aligned}\delta'(q_0 q_1, 1) &= \delta(q_0, 1) \cup (q_1, 1)\\ &= \{q_1\} \cup \{q_0, q_1\}\\ &= q_0\, q_1\end{aligned}$$

Figure 2.21(d) shows these two transitions, which are self loops.

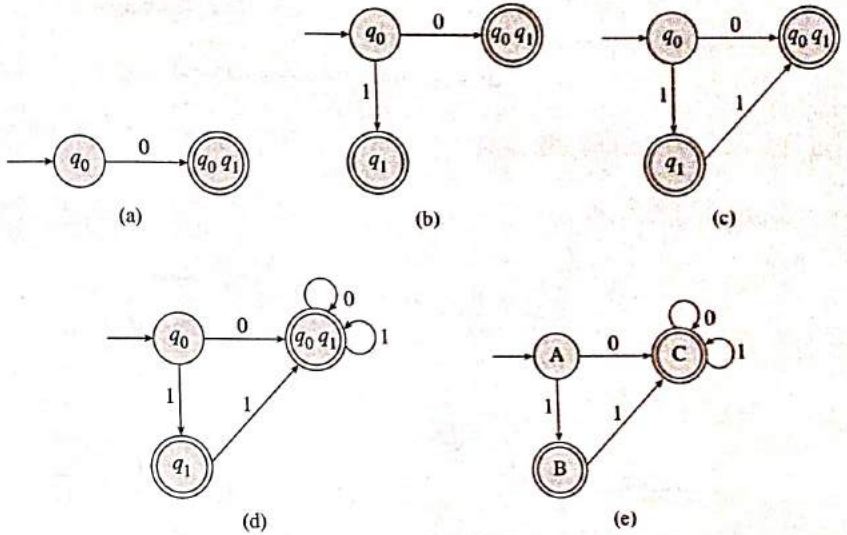


Figure 2.21 DFA construction from the given NFA (a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4 (e) Final DFA

Since, there is no state remaining to be considered for finding the transitions, Fig. 2.21(d) could be the final DFA. However, according to our convention, we change the labels for the states so as to make them singular labels: We label state q_0 as 'A'; state q_1 as 'B'; and state $q_0\, q_1$ as C. Thus, we draw the transition graph for the final DFA, as shown in Fig. 2.21(e).

We observe that this DFA is exactly the same as that we have obtained by applying the first method—refer to Fig. 2.18(b). The state transition table will also be the same as Table 2.16.

For the aforementioned example, we see that the complexity of both the methods is the same, as there are only three state combinations to work with, and all are required combinations.

Let us now look at some non-trivial examples to illustrate that method II is more efficient than method I.

Example 2.10 Construct DFA equivalent to NFA: $M = [\{p, q, r, s\}, \{0, 1\}, \delta, p, \{q, s\}]$ where, δ is given in Table 2.22.

Table 2.22 State transition table for example NFA

Q \ Σ	0	1
p	q, r	q
q	r	q, r
r	s	p
s	–	p

Solution The transitions from the states p, q, r, and s for the resultant DFA can be obtained directly from Table 2.22, which are:

$\delta'(p, 0) = qr$ (new state required) $\quad \delta'(p, 1) = q$

$\delta'(q, 0) = r \quad \delta'(q, 1) = qr$

$\delta'(r, 0) = s \quad \delta'(r, 1) = p$

$\delta'(s, 0) = \phi \quad \delta'(s, 1) = p$

where, δ' is the state transition function for the equivalent DFA.

These transitions are reflected in Fig. 2.12(a), with five states p, q, r, s, and qr, of which q, s, and qr are the final states because for the given

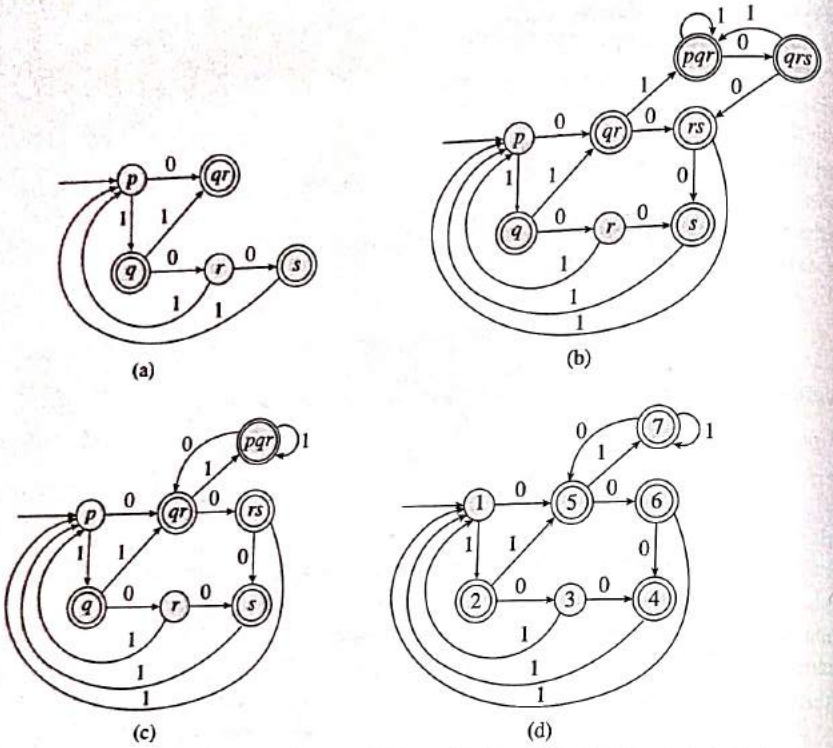


Figure 2.22 NFA to DFA conversion steps (a) Step 1 (b) Step 2 (c) DFA (without relabelling) (d) Final DFA

NFA, $F = \{q, s\}$. Hence, for the resultant DFA, we mark those states as final, which contain either q, s, or both the symbols.

The initial state will be same as that of the given NFA, that is, p—refer to Fig. 2.22(a).

Now, we have finished with transitions from states p, q, r, and s. However, there is one new state qr that we need to introduce as per the method, and find the transitions from the same:

$$\begin{aligned}\delta'(qr, 0) &= \delta(q, 0) \cup \delta(r, 0)\\ &= \{r\} \cup \{s\}\\ &= rs \quad \text{(new state)}\end{aligned}$$

Similarly,

$$\begin{aligned}\delta'(qr, 1) &= \delta(q, 1) \cup \delta(r, 1)\\ &= \{q, r\} \cup \{p\}\\ &= pqr \quad \text{(new state)}\end{aligned}$$

Thus, we must add two new states, rs and pqr, and both are final states. Let us now find transitions for both these states:

$$\begin{aligned}\delta'(rs, 0) &= \delta(r, 0) \cup \delta(s, 0) = \{s\} \cup \phi\\ &= s \quad \text{(already existing state)}\end{aligned}$$

$$\begin{aligned}\delta'(rs, 1) &= \delta(r, 1) \cup \delta(s, 1) = \{p\} \cup \{p\}\\ &= p \quad \text{(already existing state)}\end{aligned}$$

$$\begin{aligned}\delta'(pqr, 0) &= \delta(p, 0) \cup \delta(q, 0) \cup \delta(r, 0) = \{q, r\} \cup \{r\} \cup \{s\}\\ &= qrs \quad \text{(new state)}\end{aligned}$$

$$\begin{aligned}\delta'(pqr, 1) &= \delta(p, 1) \cup \delta(q, 1) \cup \delta(r, 1) = \{q\} \cup \{q, r\} \cup \{p\}\\ &= pqr \quad \text{(already added state)}\end{aligned}$$

Table 2.23 State transition table for resultant DFA

Σ / Q'	0	1
p	qr	q
$*q$	r	qr
r	s	p
$*s$	–	p
$*qr$	rs	pqr
$*rs$	s	p
$*pqr$	qrs	pqr
$*qrs$	rs	pqr

'*': Final states

We now have a new state, qrs, which is a final state, and we need to find the transitions from that state:

$$\begin{aligned}\delta'(qrs, 0) &= \delta(q, 0) \cup \delta(r, 0) \cup \delta(s, 0)\\ &= \{r\} \cup \{s\} \cup \phi\\ &= rs \quad \text{(already added state)}\end{aligned}$$

$$\begin{aligned}\delta'(qrs, 1) &= \delta(q, 1) \cup \delta(r, 1) \cup \delta(s, 1)\\ &= \{q, r\} \cup \{p\} \cup \{p\}\\ &= pqr \quad \text{(already added state)}\end{aligned}$$

The state transition table for the resultant DFA is as shown in Table 2.23.

Now, there is no new state remaining to be added, or to find the transitions form. Hence, the transition graph now can be drawn as in Fig. 2.22(b). We observe from the figure that we have considered only eight states, whereas using method I, we would have 15 states to begin with. Further, this method can never generate a disconnected graph, unlike in method I. In this way, we save on the time that is consumed to find out the transitions for unnecessary states.

Let us now draw the state transition table using the TG shown in Fig. 2.22(b), and check if it can be reduced further.

Table 2.24 Reduced state transition table for resultant DFA

Σ / Q'	0	1
p	qr	q
$*q$	r	qr
r	s	p
$*s$	–	p
$*qr$	rs	pqr
$*rs$	s	p
$*pqr$	qr	pqr

We observe from Table 2.23 that states 'r' and 'rs' have the same transitions on the same input symbols; but 'r' is a non-final state, while 'rs' is a final state. Therefore, we cannot replace 'rs' by 'r', or vice versa.

However, we can replace state qrs by state qr, as these are equivalent states. The modified state transition table can be redrawn as in Table 2.24, after replacing 'qrs' by 'qr'.

As Table 2.24 cannot be reduced further, we now draw the TG as shown in Fig. 2.22(c). After changing the state labels, we can further redraw it as in Fig. 2.22(d), which is the final equivalent DFA.

2.7 NFA WITH ϵ-TRANSITIONS

NFA with ϵ-transitions is an NFA that includes transitions on the empty input symbol, that is, 'ϵ' (epsilon). This is also called an NFA with ϵ-moves.

Formal Definition

NFA with ϵ-moves is denoted by a 5-tuple:

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

where,

Q: Finite set of states
Σ: Finite input alphabet
q_0: Initial state contained in Q
F: Set of final states $\subseteq Q$
δ: State function mapping $Q \times (\Sigma \cup \{\epsilon\})$ to 2^Q
i.e., $\delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$

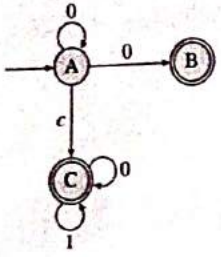


Figure 2.23 Example NFA with ϵ-transitions

Basically, it is an NFA having transitions on the empty input 'ϵ'.

For example, let us refer to Fig. 2.23, which shows one ϵ-transition from state A to state C.

The term ϵ-transition means a transition on an empty input; it is actually a path whose length is zero. In the aforementioned example, we can say that the distance between state A and state C is zero—reading 'ϵ' is as good as reading no input symbol. Figure 2.24 shows another example NFA with ϵ-transitions.

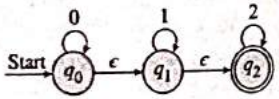


Figure 2.24 Example NFA with ϵ-transitions

We observe from Fig. 2.24 that there are two ϵ-transitions: One from state q_0 to state q_1, and another from state q_1 to state q_2. We also see that from every state there is a unique transition on the same symbol. Then, the question may arise: Why is this FA considered an NFA? The answer to this question is that any FA with ϵ-moves is always an NFA. A DFA can never have ϵ-moves.

There is one more reason, which is explained as follows:

From Fig. 2.24, we can write:

$$\delta(q_0, 0) = q_0$$
$$\delta(q_0, \epsilon) = q_1 \qquad (2.1)$$

Therefore, $\delta(q_0, 0\epsilon) = \delta(\delta(q_0, 0), \epsilon)$
$= \delta(q_0, \epsilon)$
$= q_1$

Here, '0ϵ' means zero concatenated with an empty string, that is, only '0'.
Therefore,

$$\delta(q_0, 0\epsilon) = \delta(q_0, 0)$$
$$= q_0 \qquad (2.2)$$

From Eqs (2.1) and (2.2), we can write,

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

Similarly,

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

Thus, on a single symbol, from state q_0, the FA has two different transitions. The same thing holds for state q_1 as well. Therefore, the diagram must be an NFA (obviously, with ϵ-moves).

2.7.1 Significance of NFA with ϵ-Transitions

Now, let us try to address the question: Why would we need NFA with ϵ-moves?

If we want to construct an FA, which accepts the language:

L = Set of strings with zero or more number of 0's, followed by zero or more number of 1's, followed by zero or more number of 2's

It becomes very difficult and may even seem impossible to directly draw an NFA or DFA from the aforementioned language description. However, we may observe that the NFA with ϵ-moves in Fig. 2.24 accepts exactly the same language described here. Hence, it becomes very easy to draw an NFA with ϵ-moves from the language description, with the help of ϵ-transitions.

Now, let us check whether the NFA accepts the string 'ϵ', that is, with zero number of 0's, followed by zero number of 1's, followed by zero number of 2's.

Let us start with the initial state:

$$\delta(q_0, \epsilon) = q_1$$
$$\delta(q_1, \epsilon) = q_2$$

Therefore, $\delta(q_0, \epsilon\epsilon) = q_2$

Here, '$\epsilon\epsilon$' is nothing but 'ϵ';

hence, $\delta(q_0, \epsilon) = q_2$ (final state)

We see that starting from the initial state, q_0, and reading string ϵ, the machine reaches the final state q_2; hence, the machines accepts 'ϵ'.

Let us now check the acceptance of '002'—two number of 0's, followed by zero number of 1's, followed by one number of 2's:

$$\delta(q_0, 0) = q_0$$
$$\delta(q_0, 0) = q_0$$
$$\delta(q_0, \epsilon) = q_1$$
$$\delta(q_1, \epsilon) = q_2$$
$$\delta(q_2, 2) = q_2 \text{ (final state)}$$

Therefore, $\delta(q_0, 00\epsilon\epsilon 2) = q_2$

i.e., $\delta(q_0, 002) = q_2$

This means that '002' is a valid string, as q_2 is the final state, which is reached upon reading the input string '002'.

Thus, we see that NFA with ϵ-moves helps us split complex language acceptance problems into smaller ones. The solutions to these problems can then be integrated with the help of ϵ-transitions.

For example, in Fig. 2.24, state q_0 accepts inputs consisting of zero or more number of '0's; state q_1 accepts inputs with zero or more number of '1's; and state q_2 accepts zero or more number of '2's. Hence, the original language L now is formed by sequencing these three parts, so as to make one follow the other.

2.7.2 State Transition Table for NFA with ϵ-Transitions

The state transition table for an NFA with ϵ-moves (or transitions) is similar to that of an NFA or DFA. The only difference is that along with the columns labelled by input symbols from Σ, there is an additional column labelled 'ϵ'.

This is in line with the transition function defined earlier:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$$

The state transition table for the NFA with ϵ-moves in Fig. 2.24 is as shown in Table 2.25.

Table 2.25 State transition table for NFA with ϵ-moves in Fig. 2.24

Q \ $\Sigma \cup \{\epsilon\}$	0	1	2	ϵ
q_0	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
q_1	ϕ	$\{q_1\}$	ϕ	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$	ϕ

Hence, a transition for NFA with ϵ-moves can be described as:

$\delta(q, a)$ = set of all states p such that there is a transition labelled a from state q to state p, where a is either ϵ or $a \subseteq \Sigma$

2.7.3 ϵ-Closure of a State

The set of all states p, such that there is a path from state q to state p labelled 'ϵ', is known as ϵ-closure (q). In other words, it is the set of all the states having distance zero from state q. It is pronounced as 'epsilon closure of q'.

For example, let us consider the NFA with ϵ-moves in Fig. 2.24:

$$\epsilon\text{-closure } (q_0) = \{q_0, q_1, q_2\}$$

Here, 'q_0' is also added to the set because every state is at distance zero from itself.

Similarly, ϵ-closure $(q_1) = \{q_1, q_2\}$, and ϵ-closure $(q_2) = \{q_2\}$.

We use a separate denotation, $\hat{\delta}$, to represent the ϵ-closure of a state. It is defined as:

$$\hat{\delta}(q_0, \epsilon) = \epsilon\text{-closure } (q_0) = \{q_0, q_1, q_1\}$$

2.8 EQUIVALENCE OF NFA AND NFA WITH ϵ-TRANSITIONS

Let us consider an NFA with ϵ-moves, which is given by:

$$M_1 = (Q, \Sigma, \delta, q_0, F)$$

where, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$

This can be converted to an NFA without ϵ-moves, as follows:

$$M_2 = (Q, \Sigma, \delta', q_0, F')$$

where, $\delta' : Q \times \Sigma \to 2^Q$

We observe here that Q, Σ, and the initial state q_0 are the same in M_1 and M_2, but the set of final states, F and F', might not be same.

Likewise, the state transition function, δ', for the required NFA is given as:

$$\delta'(q, a) = \epsilon\text{-closure } (\delta(\hat{\delta}(q, \epsilon), a)) \tag{2.1}$$

where, $\hat{\delta}(q, \epsilon) = \epsilon\text{-closure } (q)$

Let us look at an example to illustrate this conversion method.

Example 2.11 Convert the NFA with ϵ-moves in Fig. 2.24 to its equivalent NFA without ϵ-moves, accepting the same language.

Solution Using the definition of ϵ-closure, we have:

$$\epsilon\text{-closure } (q_0) = \{q_0, q_1, q_2\}$$
$$\epsilon\text{-closure } (q_1) = \{q_1, q_2\}$$
$$\epsilon\text{-closure } (q_2) = \{q_2\}$$

We observe that 'q_2' is a member of ϵ-closure (q_0), which means that 'q_2' is at zero distance from 'q_0'. Similarly, 'q_2' is at distance zero from 'q_1' as well. Note that 'q_2' is the only final state for given NFA with ϵ-moves:

$$F = \{q_2\}$$

From, this observation, the set of final states for the resultant equivalent NFA without ϵ-moves is given by:

$$F' = \{q_0, q_1, q_2\}$$

Now, let us find the state transition function, δ', for the resultant NFA. This can be obtained with the help of the rule in Eq. (2.1):

$$\begin{aligned}
\delta'(q_0, 0) &= \epsilon\text{-closure } (\delta(\hat{\delta}(q_0, \epsilon), 0)) \\
&= \epsilon\text{-closure } (\delta(\{q_0, q_1, q_2\}, 0)) \\
&= \epsilon\text{-closure } (\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)) \\
&= \epsilon\text{-closure } (\{q_0\} \cup \phi \cup \phi) \\
&= \epsilon\text{-closure } (q_0) \\
&= \{q_0, q_1, q_2\}
\end{aligned}$$

$$\begin{aligned}
\delta'(q_0, 1) &= \epsilon\text{-closure } (\delta(\hat{\delta}(q_0, \epsilon), 1)) \\
&= \epsilon\text{-closure } (\delta(\{q_0, q_1, q_2\}, 1)) \\
&= \epsilon\text{-closure } \{\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) \\
&= \epsilon\text{-closure } (\phi \cup \{q_1\} \cup \phi) \\
&= \epsilon\text{-closure } (q_1) \\
&= \{q_1, q_2\}
\end{aligned}$$

$$\begin{aligned}
\delta'(q_0, 2) &= \epsilon\text{-closure } (\delta(\hat{\delta}(q_0, \epsilon), 2)) \\
&= \epsilon\text{-closure } (\delta(\{q_0, q_1, q_2\}, 2)) \\
&= \epsilon\text{-closure } (\delta(q_0, 2) \cup \delta(q_1, 2) \cup \delta(q_2, 2))
\end{aligned}$$

$$= \epsilon\text{-closure}(\phi \cup \phi \cup \{q_2\})$$
$$= \epsilon\text{-closure}(q_2)$$
$$= \{q_2\}$$

$$\delta'(q_1, 0) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1, \epsilon), 0))$$
$$= \epsilon\text{-closure}(\delta(\{q_1, q_2\}, 0))$$
$$= \epsilon\text{-closure}(\delta(q_1, 0) \cup \delta(q_2, 0))$$
$$= \epsilon\text{-closure}(\phi \cup \phi)$$
$$= \epsilon\text{-closure}(\phi)$$
$$= \phi$$

Similarly, we can obtain:

$$\delta'(q_1, 1) = \{q_1, q_2\}$$
$$\delta'(q_1, 2) = \{q_2\}$$
$$\delta'(q_2, 0) = \phi$$
$$\delta'(q_2, 1) = \phi$$
$$\delta'(q_2, 2) = \{q_2\}$$

Table 2.26 State transition table for NFA without ϵ-transitions

Q \ Σ	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$

From the aforementioned transitions, we construct the state transition table for the resultant NFA without ϵ-transitions as shown in Table 2.26.

The resultant NFA without ϵ-moves can be represented by a TG shown in Fig. 2.25(a).

We observe that there is a transition from state q_0 to state q_1, labelled '0, 1', meaning that there are two transitions from state q_0 to state q_1; one on reading input symbol '0', and the other on reading input symbol '1'. Here they are combined for simplicity. For a more specific diagram, refer to Fig. 2.25(b), where all transitions are shown separately.

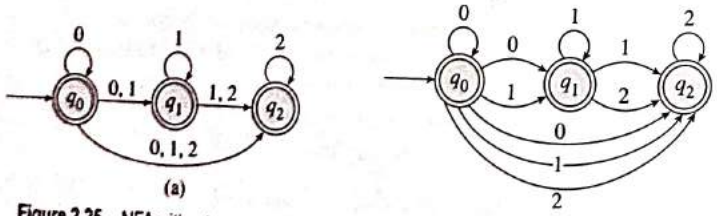


Figure 2.25 NFA without ϵ-moves (a) Resultant NFA without ϵ-moves (b) NFA without ϵ-moves (all transitions separately shown)

As we have already discussed, 'q_2' is a final state here because it is also a final state for the given NFA with ϵ-moves. In addition, 'q_0' also becomes a final state for the resultant NFA without ϵ-moves, because ϵ-closure (q_0) contains 'q_2', which is the final state of given NFA with ϵ-moves. For the same reason, 'q_1' is also a final state now for the resultant NFA without ϵ-moves.

2.9 EQUIVALENCE OF DFA AND NFA WITH ϵ-TRANSITIONS

There are two different approaches (or methods) for constructing a DFA from a given NFA with ϵ-moves, namely:

1. Indirect conversion method
2. Direct conversion method

The direct method takes lesser number of steps to obtain the equivalent DFA and hence, it is faster than the indirect method.

2.9.1 Indirect Conversion Method

This method consists of two different sub-parts:

1. Construct the NFA without ϵ-moves from the given NFA with ϵ-moves; and
2. Construct an equivalent DFA from this newly-constructed NFA without ϵ-moves, using either of the two methods discussed in Section 2.6.

Let us look at an example to demonstrate the indirect conversion method.

Example 2.12 Construct an equivalent DFA for the NFA with ϵ-moves shown in Fig. 2.24.

Solution For the given NFA with ϵ-moves in Fig. 2.24, we have already constructed an equivalent NFA without ϵ-moves in Fig. 2.25.

For converting this NFA without ϵ-moves in Fig, 2.25 to its equivalent DFA, let us use conversion method II (refer to Section 2.6.3).

From Table 2.26, which is the state transition table for the NFA in Fig. 2.25, we can directly write the state transition function, δ_1, for the resultant DFA:

$\delta_1(q_0, 0) = q_0 q_1 q_2$ (new state)
$\delta_1(q_0, 1) = q_1 q_2$ (new state)
$\delta_1(q_0, 2) = q_2$ (already existing state)
$\delta_1(q_1, 1) = \phi$
$\delta_1(q_1, 1) = q_1 q_2$
$\delta_1(q_1, 2) = q_2$
$\delta_1(q_2, 0) = \phi$
$\delta_1(q_2, 1) = \phi$
$\delta_1(q_2, 2) = q_2$

These transitions are shown in Fig. 2.26(a); here, transitions from the states '$q_0 q_1 q_2$' and '$q_1 q_2$' are unprocessed.

The new states '$q_1 q_2$' and '$q_0 q_1 q_2$' are also final states, because 'q_0', 'q_1', and 'q_2' are the final states of the original NFA.

Now,

$$\delta_1(q_1, q_2, 0) = \delta'(q_1, 0) \cup \delta'(q_2, 0) = \phi \cup \phi$$
$$= \phi$$

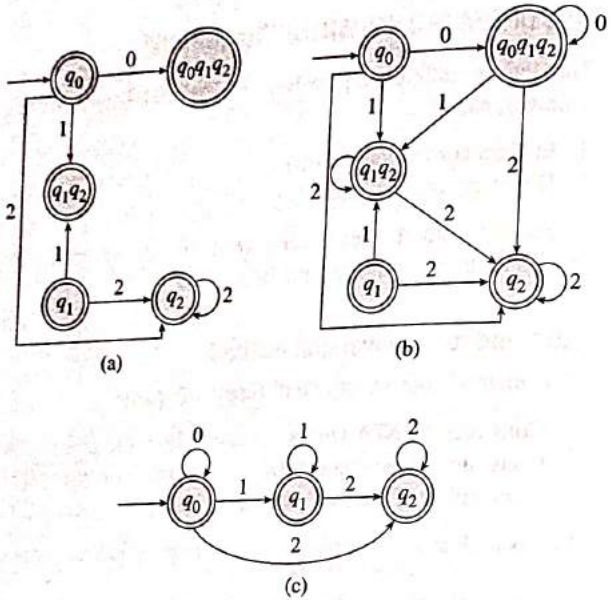


Figure 2.26 Construction of DFA from the given NFA (a) Step 1 (b) Step 2 (all transitions shown, no state unprocessed) (c) Final DFA

$$\delta_1 (q_1 q_2, 1) = \delta' (q_1, 1) \cup \delta' (q_2, 1) = \{q_1, q_2\} \cup \phi$$
$$= q_1 q_2 \text{ (already existing state)}$$

$$\delta_1 (q_1 q_2, 2) = \delta' (q_1, 2) \cup \delta' (q_2, 2)$$
$$= \{q_2\} \cup \{q_2\}$$
$$= q_2 \text{ (already existing state)}$$

$$\delta_1 (q_0 q_1 q_2, 0) = \delta' (q_0, 0) \cup \delta' (q_1, 0) \cup \delta' (q_2, 0)$$
$$= \{q_0, q_1, q_2\} \cup \phi \cup \phi$$
$$= q_0 q_1 q_2 \text{ (already existing state)}$$

$$\delta_1 (q_0 q_1 q_2, 1) = \delta' (q_0, 1) \cup \delta' (q_1, 1) \cup \delta' (q_2, 1)$$
$$= \{q_1, q_2\} \cup \{q_1, q_2\} \cup \phi$$
$$= q_1 q_2 \text{ (already existing state)}$$

$$\delta_1 (q_0 q_1 q_2, 2) = \delta' (q_0, 2) \cup \delta' (q_1, 2) \cup \delta' (q_2, 2)$$
$$= \{q_2\} \cup \{q_2\} \cup \{q_2\}$$
$$= q_2 \text{ (already existing state)}$$

Since we are now left with no new state, the modified state transition diagram for the equivalent DFA, with all the states and their respective transitions, can be drawn as shown in Fig. 2.26(b). The state transition table for the same is as shown in Table 2.27.

From the table, we observe that we can replace state '$q_0 q_1 q_2$' by state 'q_0', and state '$q_1 q_2$' by 'q_1' as these are equivalent states. We now have the minimized state transition table as shown in Table 2.28. The TG for this can be drawn as in Fig. 2.26 (c).

Table 2.27 State transition table for the equivalent DFA

Q' \ Σ	0	1	2
*q_0	$q_0 q_1 q_2$	$q_1 q_2$	q_2
*q_1	–	$q_1 q_2$	q_2
*q_2	–	–	q_2
*$q_1 q_2$	–	$q_1 q_2$	q_2
*$q_0 q_1 q_2$	$q_0 q_1 q_2$	$q_1 q_2$	q_2

'*': Final states

Table 2.28 Reduced state transition table for the DFA

Q' \ Σ	0	1	2
*q_0	q_0 •	q_1 •	q_2
*q_1	–	q_1 •	q_2
*q_2	–	–	q_2

'*': Final states
'•': Modified entries

2.9.2 Direct Conversion Method

In this approach, we directly begin with the initial state, and go on adding states as and when required to the diagram, till we come to the stage when there exists no state without having the transitions specified. Here, each state label consists of two parts: The first part is the state label, and the other part is a combination of all the state symbols, which are reachable from the given state.

States that are reachable for a given state p are the states, which are at zero distance from the state p, that is, all such states 'q', to which there are paths from the state p labelled as 'ϵ', excluding the state p itself.

Example 2.13 Consider the NFA with ϵ-moves shown in Fig. 2.24. Construct the DFA equivalent to it using direct approach.

Solution Let us begin with the start state q_0 of the given NFA with ϵ-moves. We create a new state as a part of the resultant DFA; the first part of the label is 'q_0', and the second part includes all the reachable states of state q_0, that is, 'q_1, q_2', as shown in Fig. 2.27(a).

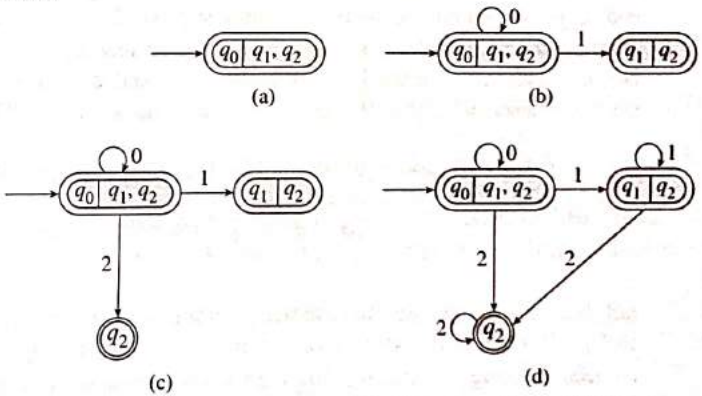


Figure 2.27 DFA construction from NFA with ϵ-moves (a) Step 1 (b) Step 2 (c) Step 3 (d) DFA

Since state q_2, which is a final state for the given NFA with ϵ-moves, is reachable from state q_0; we mark this new state as a final state.

We now proceed to find the transitions from this new state:

Let us create new states if we find any new combination for the next state. From Table 2.25, which represents the state transition function, δ, for the NFA with ϵ-moves in Fig. 2.25, we can directly write the state function, δ_1, for the resultant DFA.

We observe that in the given NFA with ϵ-moves, state q_0 on reading the input symbol '0' goes to itself. We now proceed with the next symbol in the same state label, that is, 'q_1'. State q_1 on reading input symbol '0' goes nowhere. Similarly, the other part of the label, that is, state q_2 also does not have any transition on reading input symbol '0'. Hence, the next state is the same state; so we add the self loop on symbol '0'. This can be explained with the help of the following equation:

$$\begin{aligned}\delta_1([q_0, q_1q_2], 0) &= [\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0), \text{ reachable states from the resultant states}]\\ &= [\{q_0\} \cup \phi \cup \phi, \text{ reachable states for } q_0]\\ &= [q_0, q_1q_2]\end{aligned}$$

There are no transitions from state q_0 on reading input symbols '1' and '2': state q_1 on reading input symbol '1' goes to state q_1; state q_2 has no transition on reading input symbol '1'. Therefore, we create a new state with first part of the label as q_1, and the second part as all reachable states from 'q_1', that is, state q_2. Hence, we mark this state also as a final state, as state q_2 is reachable from state q_1. The current stage of the TG is reflected in Fig. 2.27(b).

$$\begin{aligned}\delta_1([q_0, q_1q_2], 1) &= [\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1), \text{ reachable states from the resultant states}]\\ &= [\phi \cup \{q_1\} \cup \phi, \text{ reachable states for } q_1]\\ &= [q_1, q_2]\end{aligned}$$

Now, we proceed with the next symbol in state q_0: state q_2 on reading input symbols '0' and '1' goes nowhere, but on reading input symbol '2', it moves to state q_2. Furthermore, there is no transition from states 'q_0' and 'q_1' on reading input symbol '2'. Hence, we create a new state with label 'q_2' as the first part, and, as no other state is reachable from state q_2, the second part of the state is empty, as shown in Fig. 2.27(c).

$$\begin{aligned}\delta_1([q_0, q_1q_2], 2) &= [\delta(q_0, 2) \cup \delta(q_1, 2) \cup \delta(q_2, 2), \text{ reachable states from the resultant states}]\\ &= [\phi \cup \phi \cup \{q_2\}, \text{ reachable states for } q_2]\\ &= [q_2,]\end{aligned}$$

Let us now proceed with the state whose first part is 'q_1'. It consists of two symbols—q_1 and q_2. We observe that state q_1 on reading input symbol '1' goes to itself; hence, there is no need to create a new state; also, it goes nowhere on reading input symbols '0' and '2'. Likewise, the next state, q_2, transits to nowhere on reading input symbols '0' and '1', but on reading '2', it goes to state q_2 which is already created.

Therefore, we have:

$$\begin{aligned}\delta_1([q_1, q_2], 1) &= [\delta(q_1, 1) \cup \delta(q_2, 1), \text{ reachable states}]\\ &= [\{q_1\} \cup \phi, \text{ reachable states for } q_1]\\ &= [q_1, q_2]\\ \delta_1([q_1, q_2], 2) &= [\delta(q_1, 2) \cup \delta(q_2, 2), \text{ reachable states}]\\ &= [\phi \cup \{q_2\}, \text{ reachable states for } q_2]\\ &= [q_2,]\end{aligned}$$

Finally, the only remaining state is q_2, which on reading input symbol '2', goes to itself. This ends the process of finding new states and transitions. This stage of the DFA is reflected in Fig. 2.27(d).

Let us compare this DFA with the DFA in Fig. 2.26(c). We observe that it is the same DFA. Hence,

$$\begin{aligned}\delta_1([q_2,], 2) &= [\delta(q_2, 2), \text{ reachable states}]\\ &= [\{q_2\}, \text{ reachable states for } q_2]\\ &= [q_2,]\end{aligned}$$

2.10 FINITE AUTOMATA WITH OUTPUT

As we know, FA is the mathematical model of FSM, and is defined by a five-tuple: $(Q, \Sigma, \delta, q_0, F)$, which does not include information regarding the output. After reading a string, if the FA resides in a final state, the string is considered as 'accepted' by the FA; otherwise, it is 'rejected'. Since FA is the language acceptor (or recognizer), it can generate only these two outcomes—more like a Boolean function.

The definition of FA we have studied so far includes the limited power of FSM as language acceptor. We can now formalize the definition of FSM with output and machine function (MAF) as well. There are two different types (or visualizations) of the FSM that generate output, namely:

1. Moore machine
2. Mealy machine

In this section, we are going to study the formalism around these two different FSM types.

2.10.1 Moore Machine

A Moore machine is a machine with finite number of states, and for which the output symbol at any given time depends only upon the current state of the machine (and not on the input symbol read).

Formal Definition

A Moore machine is a six-tuple that is defined as follows:

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

where,

Q: Finite set of states
Σ: Finite input alphabet
Δ: An output alphabet
δ: State transition function (STF); $\delta: Q \times \Sigma \rightarrow Q$
λ: Machine function (MAF); $\lambda: Q \rightarrow \Delta$
q_0: Initial state of the machine

Thus, for a Moore machine, an output symbol is associated with each state. When the machine is in a particular state, it generates the output, irrespective of the input that caused the transition.

Let us look at an example to illustrate this machine.

Example 2.14 Construct a Moore machine to find out the residue-modulo-3 for binary numbers.

Solution If i is a binary number, and if we write '0' after i then, its value becomes $2i$. For example, consider a binary number:

$i = 1$ (value = 1)

If we write '0' after i, then we have:

$i.0 = 10$ (value = 2×1)

As another example, let us we consider:

$i = 100$ (value = 4)

Then, $i.0 = 1000$ (value = $8 = 2 \times 4$)

Similarly, if we write '1' after i, where i is any binary number, its value becomes '$2i + 1$'. For example, consider the binary number:

$i = 1$ (value = 1)

Then, $i.1 = 11$ (value = $3 = 2 \times 1 + 1$)

As another example, let us consider:

$i = 100$ (value = 4)
$i.1 = 1001$ (value = $9 = 2 \times 4 + 1$)

As we are constructing a machine to determine the remainder (or residue) when we divide any binary number by 3, the different remainder values that we can have are: 0, 1, and 2.

In Table 2.29, let us consider a case in which the remainder from the previous result is 2—that is, 10 in binary form. Now, if we write 0 after it, it becomes 4—that is, 100 in binary form. When we divide this by 3, the remainder will be 1. In the division process, we read the input from left to right, one digit at a time. The digit we read is divided by the divisor (in this example, the divisor is 3). The next digit is then concatenated to the remainder of this division to form the next number to be divided. We continue this process till all the digits in the input string are exhausted. We are going to use the same in our computations.

Table 2.29 Different remainder values

Remainder value (R)	0	1	2
When we write '0' after (R) and divide by 3, remainder = $2R$ mod 3	0	2	1
When we write '1' after (R) and divide by 3, remainder = $(2R + 1)$ mod 3	1	0	2

Table 2.30 Machine function for Moore machine, $\lambda : Q \rightarrow \Delta$

State Q	q_0	q_1	q_2
Output Δ	0	1	2

As we are interested in constructing a Moore machine for which the output depends only on the current state of the machine, we can associate the three different remainder values with three different states: remainder 0 with state q_0, remainder 1 with state q_1, and remainder 2 with state q_2, as described in the MAF shown in Table 2.30.

Now, Table 2.31 shows the state transition table for the required Moore machine. This is same as Table 2.29 depicting formal notation.

We observe that Table 2.31 is an exact reflection of Table 2.29 that we prepared earlier. The transition graph for this can be drawn as in Fig. 2.28. In the diagram, along with every state vertex, there is a symbol written below, which is an output symbol associated with that state.

Table 2.31 State transition table for Moore machine

Q \ Σ	0	1
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

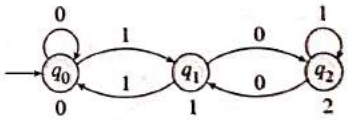


Figure 2.28 TG for Moore machine

2.10.2 Mealy Machine

A Mealy machine is a machine with finite number of states, and for which the output symbol at any given time is a function of (i.e., it depends on) the current input symbol as well as the current state of the machine.

Formal Definition

A Mealy machine is denoted by a six-tuple:

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

where,

Q: Finite set of states
Σ: Finite input alphabet

Δ: Finite output alphabet
δ: State transition function (STF); $\delta: Q \times \Sigma \rightarrow Q$
λ: Machine function (MAF); $\lambda: Q \times \Sigma \rightarrow \Delta$
q_0: Initial state of the machine

Thus, for this type of machine, the output depends on both current state and the current input symbol. We may recall that this is the same FSM that we have discussed in the beginning of this topic, that is, Section 2.2. All the examples we have seen earlier in the section are Mealy machines.

Example 2.15 Design a Mealy machine that accepts the language consisting of strings from Σ^*, where $\Sigma = \{0, 1\}$, and ending with double '0's or double '1's.

Solution Let us assume that the output alphabet, $\Delta = \{y, n\}$, indicating whether the input string is accepted or not: y—that is, yes—will be the output if the string is accepted, and n—that is, no—will be the output if the string is not accepted by the machine.

Let us assume that the initial state of the machine is 'q_0'. From state q_0 there will be two transitions: one on reading input symbol '0', and the other on reading input symbol '1', as $\Sigma = \{0, 1\}$.

On reading input symbol '0', the machine makes the transition to some other state, say 'p_0', which looks for a second consecutive zero to get double '0's. Similarly, on reading input symbol '1', the machine transits to another state, say 'p_1', which looks for a second consecutive one to get double '1's. The situation is represented in Fig. 2.29(a).

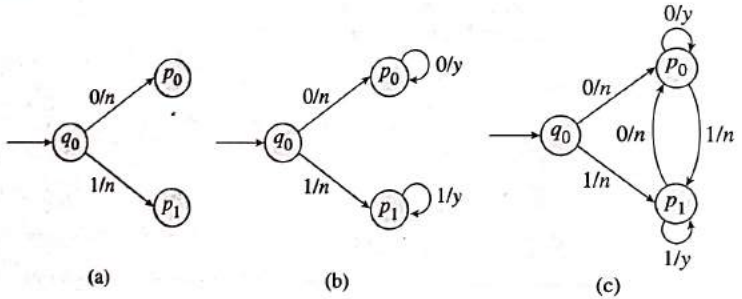


Figure 2.29 Construction of Mealy machine (a) Step 1 (b) Step 2 (c) Final Mealy machine

In state p_0 if the machine reads '0', it remains in the same state and produces output 'y', as consecutively two 0's have been read. It continues to remain in the same state on reading more 0's, because the string may be of the form '0000'—any number of '0's more than two '0's always end in at least two '0's. Similarly, in state p_1 if machine reads '1', it transits to the same state with output 'y', indicating it has read double '1's. This state is reflected in Fig. 2.29(b).

Now, in state p_0, if the machine reads symbol '1', it makes a transition to state p_1, which checks for a second consecutive '1'. Next, in state p_1, if the machine reads '0', it transits to state p_0 which looks for a second consecutive '0'. The output in both these cases is 'n', as the second consecutive letter is yet to be read.

Here ends the process of finding transitions on both '0' and '1' from every state of the machine. Figure 2.29(c) shows the final Mealy machine that complies with the given requirements.

Example 2.16 Design a Mealy machine for incrementing the value of any binary number by one. The output should also be a binary number, whose value is one more than the number given.

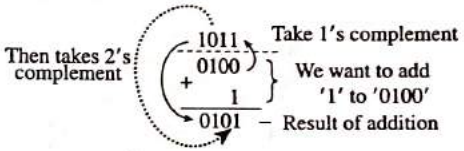
Solution In order to obtain the 2's complement of a binary number, we first take the 1's complement and add '1' to it.

For example, let 1011 be the given binary number. Then we calculate the 2's complement as follows:

1011	–	given binary number
0100	–	1's complement
+ 1	–	add one
0101	–	2's complement

We use a similar method to design a machine that will add '1' to a given binary number.

We will revise the aforementioned method, as shown here:



Thus, in order to add '1' to any binary number, we find the 1's complement of the given number and then find the 2's complement of the 1's complement that we have obtained.

In other words, given a number, the output should be the incremented result of adding '1' to the given number. For example:

0100	–	Given number
0101	–	Incremented result

Instead of first finding the 1's complement and then the 2's complement, we can adopt a simple direct approach, by following the simple steps given here:

1. Read bit by bit from the least significant bit (LSB) of the given binary number. This is more like reading the input string in the reverse order, that is, from right to left.
2. Keep on replacing the 1's by 0's, till we reach the first 0 (from the right).
3. Replace this first 0 by 1.
4. After this first 0, keep the remaining bits as they are.

In our example:

0100
→ First '0' replace by '1'

0101— Result of adding '1'
Remaining bills as they are without any change

In our design, there will be two states:

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

State q_0 is the initial state, which is associated with replacing all 1's by 0's till it reaches the first 0. After reaching the first 0, while reading from right to left, the machine replaces it by 1 and moves to the next state q_1. State q_1 on reading '0' generates output 0, and on reading '1', generates output 1. It thus ensures that the remaining bits are not changed.

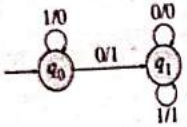


Figure 2.30 Mealy machine to increment value of a binary number by '1'

Figure 2.30 depicts the final Mealy machine. For this machine, as we know:

$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
$\Delta = \{0, 1\}$

Tables 2.32(a) and 2.32(b) respectively represent the state transition function, δ, and the machine function, λ.

Table 2.32 STF and MAF (a) $\delta : Q \times \Sigma \to Q$
(b) $\lambda : Q \times \Sigma \to \Delta$

Q \ Σ	0	1
q_0	q_1	q_0
q_1	q_1	q_1

(a)

Q \ Σ	0	1
q_0	1	0
q_1	0	1

(b)

Example 2.17 Design a Mealy machine to find the 2's complement of a given binary number.

Solution Let us have a simple algorithm similar to that of the previous example:

1. Read bit by bit from LSB (right to left, in the reverse order).
2. Keep the bits unchanged till you reach the first '1' from the right side; do not replace this first '1' by '0'. That remains unchanged as well.
3. The remaining bits are to be changed from 0 to 1, and from 1 to 0.

The design here again requires two states:

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

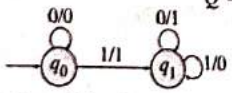


Figure 2.31 Mealy machine to find 2's complement of a binary number

The initial state, q_0 reads all the 0's (from right to left) without replacing them, till it reaches the first '1'. Upon reading the first '1', it makes a transition to state q_1. In this state, the machine replaces each '0' that is read, by 1, and every '1' that is read, by 0. The Mealy machine can be constructed as shown in Fig. 2.31.

Simulation

Let us now simulate the working of the Mealy machine in Fig. 2.31 on an input binary number, '1010' (shown in Fig. 2.32). Note that the machine reads the input string from right to left, one bit at a time.

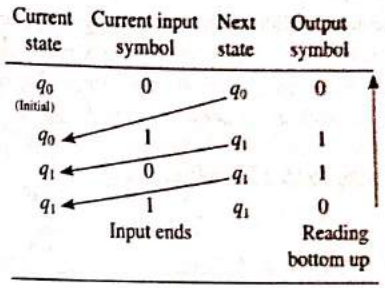


Figure 2.32 Simulation of Mealy machine

The output can be obtained by reading from bottom to top as we are considering the input in the reverse order, that is, right to left. The output will be '0110'. Hence, '0110' is the 2's complement of '1010'.

Let us check the validity of the answer that we have obtained, by our usual method:

```
  1010   - number given
  0101   - 1's complement
+    1   - add 1
  0110   - 2's complement
```

Hence, the answer obtained is correct.

2.10.3 Finite State Transducer

A finite state transducer (FST) is an FSM with output tape. As we know, an ordinary FSM has only an input tape from where the input symbols are read.

As we know, Moore and Mealy machines generate output for every input that is read—every transition yields some output symbol. In a way, these machines generate the output string upon reading the input string or sequence. Let us assume that these machines can write the output symbols one at each transition onto an output tape that stores the output string generated. Such machines are called finite state transducers—a transducer translates (or transduces) the input string into an output string.

We observe that examples such as finding the residue-modulo-3 do not require the intermediate outputs to be stored. Since only the final remainder value is of importance, there is no need to store the intermediate outcomes. However, let us consider examples such as obtaining 1's or 2's complement of a binary number, or incrementing a binary number by one. These examples require to store the output somewhere, such as, the output tape.

Essentially, Moore and Mealy machines can be implemented as FSTs if the example at hand must store the output sequence upon reading the input sequence, one symbol at a time.

2.11 EQUIVALENCE OF MOORE AND MEALY MACHINES

In the previous sections, we have seen the equivalence of NFA with DFA, as well as with ϵ-moves. Similarly, converting a Moore machine to its equivalent Mealy is also possible.

As discussed earlier, Moore and Mealy machines are two different visualizations of an FSM. For every Mealy machine, there exists an equivalent Moore machine, and vice versa. This means that both these types of machines are equivalent to one another.

Let us now look at the conversion methods.

2.11.1 Moore to Mealy Conversion

If a Moore machine is given as:

$$M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$$

then, its equivalent Mealy machine is:

$$M_2 = (Q, \Sigma, \Delta, \delta, \lambda', q_0)$$

where, $\lambda'(q, a) = \lambda(\delta(q, a))$

$\forall (q \subset Q$ and $\forall a \subset \Sigma)$

In case of a Moore machine, the output only depends on the machine's state. On the other hand, in case of a Mealy machine, it depends on the machine's state as well as the input symbol read. Hence, the only difference between the two is the machine function, λ.

The aforementioned rule stated, that is, $\lambda'(q, a) = \lambda(\delta(q, a))$, associates the output of the next state, that is, $\lambda(\delta(q, a))$ with the transition for the resultant Mealy machine, that is, $\lambda'(q, a)$.

Example 2.18 Consider a Moore machine that we have already designed (refer to Fig. 2.28) for finding residue-mod-3 for any binary number, redrawn as Fig. 2.33(a). Convert this Moore machine to its equivalent Mealy machine.

Solution Consider the Moore machine in Fig. 2.33(a).

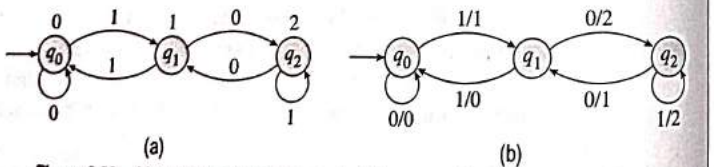


Figure 2.33 Moore to Mealy Conversion (a) Moore machine (b) Equivalent Mealy machine

Table 2.33 State transition function, $\delta : Q \times \Sigma \to Q$

Q \ Σ	0	1
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

The state transition function, δ, for this Moore machine is as shown in Table 2.33.

Similarly, the machine function, λ, is as shown in the Table 2.34.

Table 2.34 Machine function, $\lambda : Q \to \Delta$

Q	q_0	q_1	q_2
Δ	0	1	2

We need to convert this Moore machine to its equivalent Mealy machine using the state transition function, δ, shown in Table 2.33. For this we only need to compute the changed λ', that is, the machine function.

As per the conversion rule:

$$\lambda'(q, a) = \lambda(\delta(q, a))$$

We may write:

$$\lambda'(q_0, 0) = \lambda(\delta(q_0, 0)) = \lambda(q_0) = 0$$
$$\lambda'(q_0, 1) = \lambda(\delta(q_0, 1)) = \lambda(q_1) = 1$$
$$\lambda'(q_1, 0) = \lambda(\delta(q_1, 0)) = \lambda(q_2) = 2$$
$$\lambda'(q_1, 1) = \lambda(\delta(q_1, 1)) = \lambda(q_0) = 0$$
$$\lambda'(q_2, 0) = \lambda(\delta(q_2, 0)) = \lambda(q_1) = 1$$
$$\lambda'(q_2, 1) = \lambda(\delta(q_2, 1)) = \lambda(q_2) = 2$$

Table 2.35 MAF ($\lambda' : Q \times \Sigma \to \Delta$) for the equivalent Mealy machine

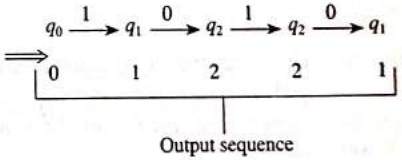
Q \ Σ	0	1
q_0	0	1
q_1	2	0
q_2	1	2

From this information, we can now create a table for the MAF of the equivalent Mealy machine as shown in Table 2.35; and the TG for the equivalent Mealy machine can be drawn as shown in Fig. 2.33(b).

Here, the output is associated with the transition, that is, it depends on the state making the transition, as well as on the input that causes the transition.

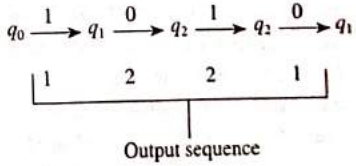
Note: Let us consider both Moore and Mealy machines shown in Fig. 2.33 for the input sequence (or string) '1010' of length, $n = 4$.

After simulation, the output sequence for the Moore machine is:



Length of the output sequence for Moore machine = 5 = $(n + 1)$

Now, let us observe the the output sequence for the Mealy machine after simulation, which is as follows:



The length of the output sequence for Mealy machine = 4 = n.

Thus, for the Moore machine, the output sequence contains one symbol more than the symbols in the input sequence, while the output sequence for the Mealy machine contains exactly the same number of symbols as in the input sequence. The additional output is generated by the Moore machine as soon as it enters the initial state, even before any input symbol is read and hence, is insignificant.

2.11.2 Mealy to Moore Conversion

Let us consider a given Mealy machine:

$$M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$$

Then its equivalent Moore machine is:

$$M_2 = ([Q \times \Delta], \Sigma, \Delta, \delta', \lambda', [q_0, b_0])$$

where, 'b_0' is an arbitrarily selected member of Δ

$$\delta' ([q, b], a) = [\delta (q, a), \lambda (q, a)]$$
$$\lambda' ([q, b]) = b$$

This conversion is slightly non-trivial. As we cannot determine the output symbol that the equivalent Moore machine holds in each state; we end up creating all possible combinations of the state symbols and the output symbols, $Q \times \Delta$. It is almost like creating multiple variants of the same state that only differ in the associated output symbol.

Each state label, thus, has two symbols: one that is a state symbol and the other its associated output symbol. Hence, in order to find the new machine function, λ', for the resultant Moore machine, it simply needs to return the associated output symbol:

$$\lambda' ([q, b]) = b$$

As we create the multiple variants from the same state that differ only in the output symbol, the state function yields the same next state, as one cannot change the state behaviour. This is very clear from the rule for finding the new state transition function, δ', for the resultant Moore machine:

$$\delta' ([q, b], a) = [\delta (q, a), \lambda (q, a)]$$

We see that the output symbol, b, has no role to play in determining the next state. This means that all the state variants obtained by associating the different output symbols—$[q, b_1]$, $[q, b_2]$, etc.—make transitions to the same next state.

Another important thing is to decide the initial state for the resultant Moore machine. Now, there exist multiple variants for the initial state, which could be of the form: $[q_0, b_i]$, $[q_0, b_j]$, ..., $[q_0, b_n]$, etc., as there are multiple output symbols. As per the rule, any state $[q_0, b_0]$ can be considered as an initial state, where b_0 is an arbitrarily selected member of Δ. The output symbol that is associated with the initial state is irrelevant here, as it gets generated even before any input symbol is read and is hence, insignificant.

Example 2.19 Consider the Mealy machine that we have designed to accept strings ending with '00' or '11' as in Fig. 2.29(c), which is redrawn in Fig. 2.34(a). The state transition tables for this machine are shown in Table 2.36. Construct the equivalent Moore machine.

Table 2.36 State transition tables for Mealy machine in Fig. 2.34(a) (a) $\delta: Q \times \Sigma \rightarrow Q$ (b) $\lambda: Q \times \Sigma \rightarrow \Delta$

Q \ Σ	0	1
q_0	p_0	p_1
p_0	p_0	p_1
p_1	p_0	p_1

(a)

Q \ Σ	0	1
q_0	n	n
p_0	y	n
p_1	n	y

(b)

Solution According to the conversion rule, the resultant Moore machine consists of:

$$Q' = [Q \times \Delta] = \{[q_0, n], [q_0, y], [p_0, n], [p_0, y], [p_1, n], [p_1, y]\}$$

Similarly, δ' and λ' can be determined, as follows:

1. $\delta' ([q_0, n], 0) = [\delta (q_0, 0), \lambda (q_0, 0)] = [p_0, n]$
   $\delta' ([q_0, n], 1) = [\delta (q_0, 1), \lambda (q_0, 1)] = [p_1, n]$
   $\lambda' ([q_0, n]) = n$
2. $\delta' ([q_0, y], 0) = [\delta (q_0, 0), \lambda (q_0, 0)] = [p_0, n]$
   $\delta' ([q_0, y], 1) = [\delta (q_0, 1), \lambda (q_0, 1)] = [p_1, n]$
   $\lambda' ([q_0, y]) = y$
3. $\delta' ([p_0, n], 0) = [\delta (p_0, 0), \lambda (p_0, 0)] = [p_0, y]$
   $\delta' ([p_0, n], 1) = [\delta (p_0, 1), \lambda (p_0, 1)] = [p_1, n]$
   $\lambda' ([p_0, n]) = n$

Similarly,

4. $\delta' ([p_0, y], 0) = [p_0, y]$
   $\delta' ([p_0, y], 1) = [p_1, n]$
   $\lambda' ([p_0, y]) = y$
5. $\delta' ([p_1, n], 0) = [\delta (p_1, 0), \lambda (p_1, 0)] = [p_0, n]$
   $\delta' ([p_1, n], 1) = [\delta (p_1, 1), \lambda (p_1, 1)] = [p_1, y]$
   $\lambda' ([p_1, n]) = n$

Similarly,

6. $\delta' ([p_1, y], 0) = [p_0, n]$
   $\delta' ([p_1, y], 1) = [p_1, y]$
   $\lambda' ([p_1, y]) = y$

The transition graph for the equivalent Moore machine is as shown in Fig. 2.34(b).

Note that states, $[q_0, n]$ and $[q_0, y]$, have the same transitions or behaviour. Similarly, $[p_0, n]$ and $[p_0, y]$ as well as $[p_1, n]$ and $[p_1, y]$ have the same behaviour, except that these states generate different outputs.

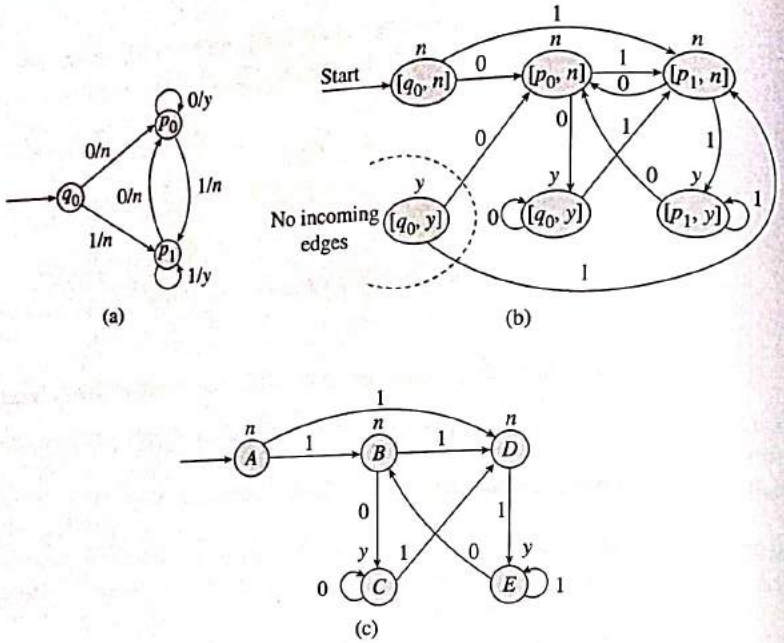


Figure 2.34 Mealy to Moore conversion (a) Mealy machine (b) Equivalent Moore machine (c) Minimized Moore machine after relabelling

We may relabel the states as we wish. Note that the state $[q_0, n]$ has been arbitrarily selected as the initial state, from among $[q_0, n]$ and $[q_0, y]$. We can remove state $[q_0, y]$, as there are no incoming edges to this state. After removing this state and relabelling the remaining states, we get the final Moore machine as shown in Fig. 2.34(c).

2.11.3 Additional Examples on Moore and Mealy Machines

Let us examine some more examples on Moore and Mealy machines and their inter-conversion.

Example 2.20 Construct Mealy and Moore machines for the following:

For input from, Σ^*, where $\Sigma = (0, 1)$, if the input ends in '101', the output should be 'x'; if the input ends in '*110*', output should be 'y'; otherwise, output should be 'z'.

Solution This is a simple sequence detector design problem. Let us start with the construction of the Mealy machine:

Consider the first string pattern '101', and draw a minimal diagram as shown in Fig. 2.35(a). We can modify this diagram to accept '110' as shown in Fig. 2.35(b). The remaining possible transitions are as shown in Fig. 2.35(c). The STF and MAF for this Mealy machine are as shown in Table 2.37.

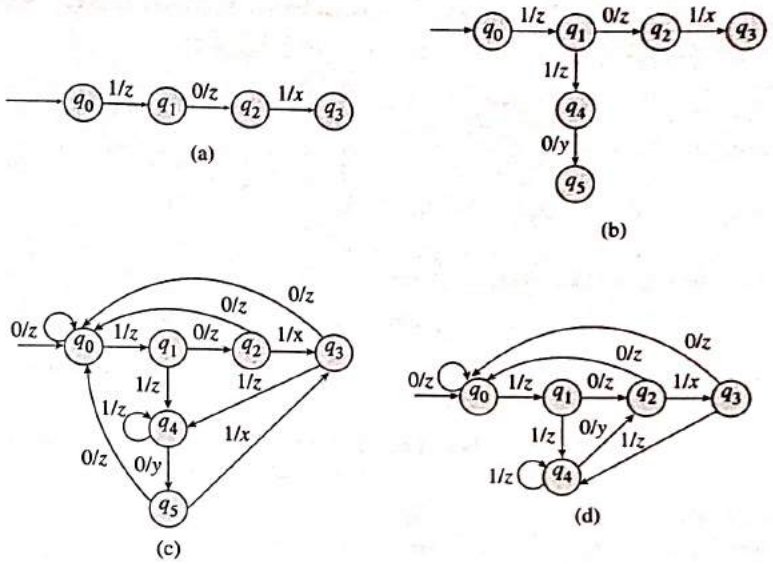


Figure 2.35 Construction of Mealy machine (a) Step 1 (b) Step 2 (c) Step 3 (d) Mealy machines

Table 2.37 STF and MAF for Mealy machine in Figure 2.34(c)
(a) $\delta : Q \times \Sigma \to Q$ (b) $\lambda : Q \times \Sigma \to \Delta$

Q \ Σ	0	1
q_0	q_0	q_1
q_1	q_2	q_4
q_2	q_0	q_3
q_3	q_0	q_4
q_4	q_5	q_4
q_5	q_0	q_3

(a)

Q \ Σ	0	1
q_0	z	z
q_1	z	z
q_2	z	x
q_3	z	z
q_4	y	z
q_5	z	x

(b)

We observe that states q_2 and q_5 are equivalent; therefore, we replace q_5 by q_2. The reduced STF and MAF tables are obtained as in Table 2.38.

The reduced Mealy machine transition graph is as shown in Fig. 2.35(d). The STF and MAF tables for the equivalent Moore machine can be obtained by using the conversion method (refer to Table 2.39). Out of the states, $[q_0, x]$, $[q_0, y]$, and $[q_0, z]$, any one can be considered as the initial state, and the other two can be omitted in order to get the final answer.

Table 2.38 Reduced STF and MAF tables (a) $\delta: Q \times \Sigma \to Q$ (b) $\lambda: Q \times \Sigma \to \Delta$

Q \ Σ	0	1
q_0	q_0	q_1
q_1	q_2	q_4
q_2	q_0	q_3
q_3	q_0	q_4
q_4	q_2•	q_4

'•' modified entry

(a)

Q \ Σ	0	1
q_0	z	z
q_1	z	z
q_2	z	x
q_3	z	z
q_4	y	z

'•' modified entry

(b)

Table 2.39 STF and MAF for equivalent Moore machine (a) $\delta': Q' \times \Sigma \to Q'$ (b) $\lambda': Q' \to \Delta$

Q' \ Σ	0	1
$[q_0, x]$	$[q_0, z]$	$[q_1, z]$
$[q_0, y]$	$[q_0, z]$	$[q_1, z]$
$[q_0, z]$	$[q_0, z]$	$[q_1, z]$
$[q_1, x]$	$[q_2, z]$	$[q_4, z]$
$[q_1, y]$	$[q_2, z]$	$[q_4, z]$
$[q_1, z]$	$[q_2, z]$	$[q_4, z]$
$[q_2, x]$	$[q_0, z]$	$[q_3, x]$
$[q_2, y]$	$[q_0, z]$	$[q_3, x]$
$[q_2, z]$	$[q_0, z]$	$[q_3, x]$
$[q_3, x]$	$[q_0, z]$	$[q_4, z]$
$[q_3, y]$	$[q_0, z]$	$[q_4, z]$
$[q_3, z]$	$[q_0, z]$	$[q_4, z]$
$[q_4, x]$	$[q_2, y]$	$[q_4, z]$
$[q_4, y]$	$[q_2, y]$	$[q_4, z]$
$[q_4, z]$	$[q_2, y]$	$[q_4, z]$

(a)

Q	Δ
$[q_0, x]$	x
$[q_0, y]$	y
$[q_0, z]$	z
$[q_1, x]$	x
$[q_1, y]$	y
$[q_1, z]$	z
$[q_2, x]$	x
$[q_2, y]$	y
$[q_2, z]$	z
$[q_3, x]$	x
$[q_3, y]$	y
$[q_3, z]$	z
$[q_4, x]$	x
$[q_4, y]$	y
$[q_0, z]$	z

(b)

Example 2.21 Construct Mealy and Moore machines for the following:

For the input from Σ^*, where $\Sigma = \{0, 1, 2\}$, print the residue-modulo-5 of the input treated as a ternary (base 3, with digits 0, 1, and 2) number.

Solution Let us first look at some properties of ternary numbers:

1. If i is a ternary number, and if we write 0 after it, then its value becomes $3i$.

   For example, $1.0 = 1 \times 3^1 + 0 \times 3^0$
   $= 3 + 0$
   $= 3$
   $= 3 \times 1$

   Similarly, $2.0 = 2 \times 3^1 + 0 \times 3^0 = 6 = 3 \times 2$

2. If i is a ternary number, and if we write 1 after it, then its value becomes '$3i + 1$'.

   For example, $1.1 = 1 \times 3^1 + 1 \times 3^0 = 3 + 1 = 3 \times 1 + 1$
   $2.1 = 2 \times 3^1 + 1 \times 3^0 = 6 + 1 = 3 \times 2 + 1$

3. If i is a ternary number, and if we write 2 after it, then its value becomes '$3i + 2$'.

   For example, $1.2 = 1 \times 3^1 + 2 \times 3^0 = 3 + 2 = 3 \times 1 + 2$
   $2.2 = 2 \times 3^1 + 2 \times 3^0 = 6 + 2 = 3 \times 2 + 2$

As we are dividing a ternary number by 5, we can have five different values for the remainders; namely, 0, 1, 2, 3, and 4. These are the five outputs that the machine generates. The remainder values are expressed in decimal (base 10) number format as usual.

It is easier to construct a Moore machine in which each output is associated with a unique state. This means, depending on the value of the remainder to be printed, there are five different states for the Moore machine:

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

The machine function is as depicted in the Table 2.40.

Table 2.40 MAF ($\lambda : Q \to \Delta$) for the Moore machine

Q	q_0	q_1	q_2	q_3	q_4
Δ	0	1	2	3	4

Depending on the remainder in the previous state, if the next digit is 0, 1, or 2, then, the next remainder value can be obtained as shown in Table 2.41.

Table 2.41 Different remainders

Remainder (R)	0	1	2	3	4
After writing 0 after R ($3R$ mod 5)	0	3	1	4	2
After writing 1 after R ($3R + 1$) mod 5	1	4	2	0	3
After writing 2 after R ($3R + 2$) mod 5	2	0	3	1	4

If the current remainder is: $R = 3$, and if the next digit is 0, then:

$$3.0 \text{ (ternary)} = 3 \times 3 = 9 \text{ (decimal)}$$

If we divide 9 by 5, the remainder is 4.

Similarly, if $R = 1$, and if 2 is the next digit, then,

$$1.2 \text{ (ternary)} = 3 \times 1 + 2 = 5 \text{ (decimal)}$$

If we divide 5 by 5, the remainder is 0.

Table 2.41 can be directly converted into the state transition table as shown in Table 2.42. Each column in Table 2.41 is analogous to a row in Table 2.42.

Using the MAF (Table 2.40) and the STF (Table 2.42), we can draw TG for the Moore machine as shown in Fig. 2.36.

Table 2.42 STF ($\delta : Q \times \Sigma \rightarrow Q$) for Moore machine

Q \ Σ	0	1	2
q_0	q_0	q_1	q_2
q_1	q_3	q_4	q_0
q_2	q_1	q_2	q_3
q_3	q_4	q_0	q_1
q_4	q_2	q_3	q_4

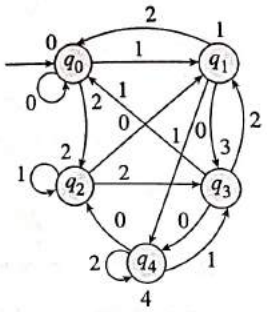


Figure 2.36 Moore machine

Now, the equivalent Mealy machine can be obtained using the conversion rule. The state function, δ, for the resultant Mealy machine is the same as in Table 2.42; and the revised machine function, λ', can be obtained using the rule:

$$\lambda'(q, a) = \lambda(\delta(q, a))$$

Using the aforementioned rule, the machine function for the Mealy machine can be written as shown in the Table 2.43.

Using the STF (Table 2.42) and MAF (Table 2.43), the transition graph for the equivalent Mealy machine can be drawn as shown in Fig. 2.37.

Table 2.43 MAF ($\lambda' : Q \times \Sigma \rightarrow \Delta$) for equivalent Mealy machine

Q \ Σ	0	1	2
q_0	0	1	2
q_1	3	4	0
q_2	1	2	3
q_3	4	0	1
q_4	2	3	4

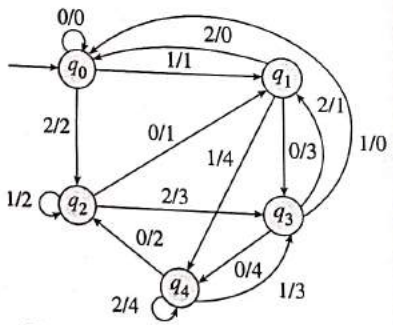


Figure 2.37 Equivalent Mealy machine

Example 2.22 Design a Moore machine that will read sequences made up of letters *A*, *E*, *I*, *O*, *U* and will give an output having the same sequences, except that in those cases where an '*I*' directly follows an '*E*', it will be changed to '*U*'.

Solution This is again a simple sequence detector problem. Let us first design the Mealy machine, which we can then convert to its equivalent Moore machine.

The constraint here is that if '*I*' directly follows '*E*' in any string over $\Sigma = \{A, E, I, O, U\}$, it must be replaced with '*U*'; the rest of the symbols remain as they are.

Step 1 in Fig. 2.38 takes care of identifying the special case, where '*I*' directly follows '*E*' in any string over $\Sigma = \{A, E, I, O, U\}$.

State q_0, on reading input symbol '*E*' moves to a new state q_1 just to remember that '*E*' has been read. Now, in state q_1 if '*I*' is the current symbol read, that means, it is the symbol that directly follows the previous symbol '*E*'; hence, it needs to be replaced with '*U*' as shown in Fig. 2.38(a). State q_1, on reading symbol '*E*' retains the same state, as the next symbol could be 'I'—a string might contain the pattern '...*EEI*...'; hence the self loop from state q_1 on reading symbol '*E*'.

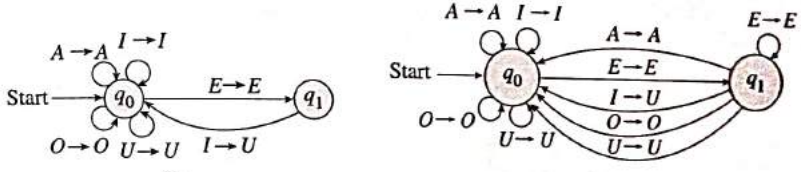


Figure 2.38 Mealy machine as a sequence detector (a) Step 1 (b) Step 2

Figure 2.38(b) gives the complete Mealy machine. It keeps the rest of the symbols as they are. We now need to convert this Mealy machine to its equivalent Moore machine.

As we know, the states of the equivalent Moore machine can be obtained as $[Q \times \Delta]$, where Q is the set of states and Δ is the output set of the Mealy machine. Thus, the STF and MAF for the equivalent Moore machine can be obtained as shown in Table 2.44.

Table 2.44 STF and MAF for the equivalent Moore machine (a) $\lambda' : Q' \times \Delta$ (b) $\delta' : Q' \times \rightarrow O'$

Q	Δ
$[q_0, A]$	A
$[q_0, E]$	E
$[q_0, I]$	I
$[q_0, O]$	O
$[q_0, U]$	U
$[q_1, A]$	A
$[q_1, E]$	E
$[q_1, I]$	I
$[q_1, O]$	O
$[q_1, U]$	U

(a)

Q' \ Σ	A	E	I	O	U
$[q_0, A]$	$[q_0, A]$	$[q_1, E]$	$[q_0, I]$	$[q_0, O]$	$[q_0, U]$
$[q_0, E]$	$[q_0, A]$	$[q_1, E]$	$[q_0, I]$	$[q_0, O]$	$[q_0, U]$
$[q_0, I]$	$[q_0, A]$	$[q_1, E]$	$[q_0, I]$	$[q_0, O]$	$[q_0, U]$
$[q_0, O]$	$[q_0, A]$	$[q_1, E]$	$[q_0, I]$	$[q_0, O]$	$[q_0, U]$
$[q_0, U]$	$[q_0, A]$	$[q_1, E]$	$[q_0, I]$	$[q_0, O]$	$[q_0, U]$
$[q_1, A]$	$[q_0, A]$	$[q_1, E]$	$[q_0, U]$	$[q_0, O]$	$[q_0, U]$
$[q_1, E]$	$[q_0, A]$	$[q_1, E]$	$[q_0, U]$	$[q_0, O]$	$[q_0, U]$
$[q_1, I]$	$[q_0, A]$	$[q_1, E]$	$[q_0, U]$	$[q_0, O]$	$[q_0, U]$
$[q_1, O]$	$[q_0, A]$	$[q_1, E]$	$[q_0, U]$	$[q_0, O]$	$[q_0, U]$
$[q_1, U]$	$[q_0, A]$	$[q_1, E]$	$[q_0, U]$	$[q_0, O]$	$[q_0, U]$

(b)

In Table 2.44(b), if we consider $[q_0, A]$ as an initial state, the states, which can be included in the minimized Moore machine, are: $[q_0, A]$, $[q_1, E]$, $[q_0, I]$, $[q_0, O]$, and $[q_0, U]$. No transition from the initial state will ever reach any other state. It is clear from Table 2.44(b) that these are the only five states that are the next states for all the transitions.

Let us now relabel the states as:

$[q_0, A] \equiv P_0$ $\quad$ $[q_1, E] \equiv P_1$
$[q_0, I] \equiv P_2$ $\quad$ $[q_0, O] \equiv P_3$
$[q_0, U] \equiv P_4$

The reduced STF and MAF tables of the equivalent Moore machine are as shown in Table 2.45.

Table 2.45 Reduced STFs and MAFs for the equivalent Moore machine
(a) $\lambda'' : Q'' \to \Delta$ (b) $\delta'' : Q'' \times \to Q''$

Q	Δ
P_0	A
P_1	E
P_2	I
P_3	O
P_4	U

(a)

Q'' \ Σ	A	E	I	O	U
P_0	P_0	P_1	P_2	P_3	P_4
P_1	P_0	P_1	P_4	P_3	P_4
P_2	P_0	P_1	P_2	P_3	P_4
P_3	P_0	P_1	P_2	P_3	P_4
P_4	P_0	P_1	P_2	P_3	P_4

(b)

Table 2.45(a) gives the machine function and Table 2.45(b) gives the state transition function for the equivalent Moore machine.

Example 2.23 Consider the Moore machine described by the state transition table given in Table 2.46. Construct the corresponding Mealy machine.

Table 2.46 State transition table for a Moore machine

Current state	Next state $a=0$	Next state $a=1$	Output
q_1	q_1	q_2	0
q_2	q_1	q_3	0
q_3	q_1	q_3	1

Table 2.47 State transition table for the equivalent Mealy machine

Q \ Σ	0	1
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_3

Solution As we know, while obtaining the equivalent Mealy machine, the state transition function, δ, remains the same, as shown in Table 2.47.

The machine function (or the output function) of the equivalent Mealy machine is given by:

$$\lambda'(q, a) = \lambda(\delta(q, a))$$

Applying this rule, we get:

$$\lambda'(q_1, 0) = \lambda(\delta(q_1, 0)) = \lambda(q_1) = 0$$

Similarly, $$\lambda'(q_1, 1) = 0$$

The complete machine function for the equivalent Mealy machine is as shown in Table 2.48.

The transition graph for the equivalent Mealy machine is as shown in Fig. 2.39.

Table 2.48 Machine function (λ) for the equivalent Mealy machine

Q \ Σ	0	1
q_1	0	0
q_2	0	1
q_3	0	1

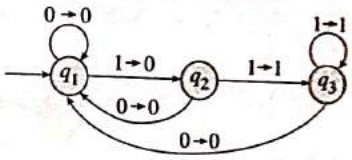


Figure 2.39 TG for the equivalent Mealy machine

2.12 FSM EQUIVALENCE

Two FSM's, M and M', are said to be equivalent, if for each state S_j in M, there corresponds at least one state S_j' in M', such that S_j' is equivalent to S_j; and if, for each S_j' in M', there corresponds at least one S_j in M, such that S_j is equivalent to S_j'. We have already learnt about state equivalence earlier in Section 2.6.2.1. In simple terms, this means that two FSMs are equivalent if they accept the same language.

2.12.1 Moore's Algorithm

The Moore's algorithm for FSM equivalence is one of the decision algorithms for regular sets. The language accepted by any FSM is called a regular language, or a regular set.

Algorithm

If M and M' are two FSM's over Σ, where $|\Sigma| = n$ ('n' is the number of input signals); then, apply the following steps to check the equivalence of M and M':

1. Construct a comparison table consisting of $(n + 1)$ columns. In column 1, list all ordered pairs of vertices of the form (v, v'), where $v \subset M$, and $v' \subset M'$.
   In column 2, list all pairs of the form (v_a, v_a'), if there is a transition labelled $a \subset \Sigma$ leading from v to v_a and from v' to v_a'.
   In column 3, list all pairs of the form (v_b, v_b'), if there is a transition labelled $b \subset \Sigma$ leading from v to v_b and from v' to v_b'. Repeat this for all 'n' symbols from Σ.
2. If the pair (v_a, v_a'), has not occurred previously in column 1, place it in the column 1 and repeat step 1 for that pair.
   Repeat step 2 for each pair, (v_b, v_b'), (v_c, v_c'), and so on.

3. If in the table, suppose a pair (v, v') is reached in which, v is the final vertex (or final state) of M, and v' is a non-final vertex of M'; or vice-versa, then stop and declare that the two FSMs, M and M', are not equivalent.
   Otherwise, stop when there are no more new pairs in columns 2, 3, …, n that do not occur in column 1. In this case, the two FSM's M and M' are equivalent.

Example 2.24 Two FSMs, M and M', are given in Fig. 2.40. Check the equivalence of the two FSMs by applying Moore's algorithm.

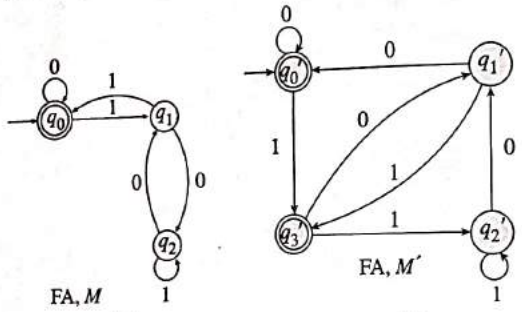


Figure 2.40 Example FSMs (a) FSM M (b) FSM M′

Solution For the given FSMs,

$$\Sigma = \{0, 1\}$$

Hence, $|\Sigma| = n = 2$

Table 2.49 Comparison table for FSMs in Fig. 2.45

(v, v')	(v_0, v_0')	(v_1, v_1')
(q_0, q_0')	(q_0, q_0')	(q_1, q_3')
(q_1, q_3')	(q_2, q_1')	(q_0, q_2')

As per the algorithm, we must create $(n + 1) = 2 + 1 = 3$ columns labelled (v, v'), (v_0, v_0'), and (v_1, v_1'). Table 2.49 shows the comparison between M and M'.

We begin with the pair of initial states (q_0, q_0'). State (q_0, q_0'), on reading input symbol '0', goes to state (q_0, q_0'), where both q_0 and q_0' are final states; and on reading input symbol '1', it goes to (q_1, q_3'), where both q_1 and q_3' are non-final states.

Therefore, we can proceed to the next step: we observe that (q_0, q_0') is already there in column 1, but (q_1, q_3') is not there. Therefore, we place it in column 1 and repeat the procedure. State (q_1, q_3'), on reading input symbol '0', goes to state (q_2, q_1')—here, both q_2 and q_1 are non-final states; and on reading input symbol '1', it goes to state (q_0, q_2')—here, q_0 is the final state in M, while q_2' is a non-final state in M'; therefore, we stop here and declare that the given FSMs, M and M', are non-equivalent.

Example 2.25 For the two FSMs given in Fig. 2.41, check the equivalence using Moore's algorithm.

Solution For the given FSMs,

$$\Sigma = \{0, 1\}$$

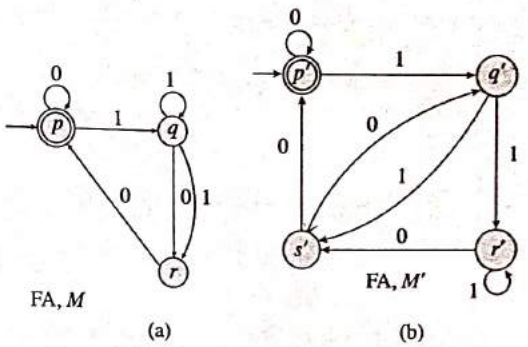


Figure 2.41 Example FSMs (a) FSM M (b) FSM M′

Hence, $|\Sigma| = n = 2$

Therefore, we need to create $(n + 1) = 3$ columns labelled (v, v'), (v_0, v_0'), and (v_1, v_1'). The comparison between the two FSMs, M and M', can be made as in Table 2.50.

Table 2.50 Comparison table for FSMs in Fig. 2.46

(v, v')	(v_0, v_0')	(v_1, v_1')
(p, p')	(p, p')	(q, q')
(q, q')	(r, s')	(q, r')
(r, s')	(p, p')	(q, q')
(q, r')	(r, s')	(q, r')

We begin with both the initial states (p, p'), which yield a new pair, (q, q'), for the transition on reading input symbol '1'. Placing (q, q') in column 1 and finding the transitions for input symbols '0' and '1', we see that the next states are: (r, s') and (q, r')—both new pairs.

Let us first consider (r, s'), and place it is column 1, which yields pairs (p, p') and (q, q') for transition on reading input symbols '0' and '1' respectively—these are already there in column 1.

Therefore, we consider the only remaining new pair, (q, r'), and place it in column 1. Neither of the transitions from (q, r') yield any new pair. Hence, there is no other new pair to process; so, we stop, and declare that the FSMs, M and M', in Fig. 2.46 are equivalent.

2.13 DFA MINIMIZATION (ANOTHER APPROACH)

The previous technique that we have seen in Section 2.6.2 uses the notion of equivalent states to reduce the state transition table. The only problem it has is that it requires more number of reduction iterations, which might be repetitive reduction. For instance, in Example 2.8, we have seen that while converting NFA to DFA (refer to Section 2.6.2.2), we had to make two reduction iterations over the transition table.

The technique that we are now going to see in this section is systematic, and finds all the equivalent states in a single step. Therefore, in one step we get a reduced or minimized DFA. This approach gives the same result, but is more efficient in terms of time.

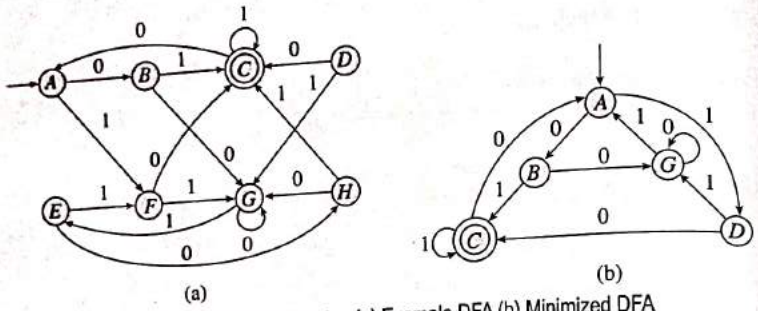


Figure 2.42 DFA minimization (a) Example DFA (b) Minimized DFA

Table 2.51 State transition table for the DFA in Fig. 2.47(a)

Q \ Σ	0	1
A	B	F
B	G	C
C	A	C
D	C	G
E	H	F
F	C	G
G	G	E
H	G	C

Table 2.52 Finding equivalent states

B							
C	X	X					
D			X				
E			X				
F			X				
G			X				
H			X				
	A	B	C	D	E	F	G

Let us consider the DFA in Fig. 2.42(a). We want to minimize this DFA. The state transition table for this DFA is as shown in Table 2.51.

First draw a table as shown in Table 2.52 and put an 'X' (cross) for all combinations of final and non-final states. Here, consider combinations of '*C*' (final state) with all other states.

Now, start with the last column, that is, with combination (*G*, *H*).

$$\delta(G, 0) = G \qquad \delta(G, 1) = E$$

and

$$\delta(H, 0) = G \qquad \delta(H, 1) = C$$

$\Big\}$ (*E*, *C*)

For pair (*E*, *C*), there is already an 'X' marked in the table, which means that these two states are non-equivalent. Therefore, we put 'X' for the pair (*G*, *H*).

We prepare the whole table as shown in Table 2.53.

Table 2.53 Finding equivalent states

B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G	X	X	X	X	X	X	
H	X		X	X	X	X	X
	A	B	C	D	E	F	G

Now, for combination (*D*, *F*),

$$\delta(D, 0) = \delta(F, 0) = C$$
$$\delta(D, 1) = \delta(F, 1) = G$$

As both transitions are equal, we do not put an 'X' for the pair (*D*, *F*)—they are equivalent.

Similarly for (*B*, *H*),

$$\delta(B, 0) = \delta(H, 0) = G$$
$$\delta(B, 1) = \delta(H, 1) = C$$

Table 2.54 Reduced state transition table

Q \ Σ	0	1
A	B	D•
B	G	C
C	A	C
D	C	G
G	G	A•

'•': Modified entries due to replacement of states

Therefore, *B* and *H* are equivalent. Hence, we do not put an 'X' for the pair (*B*, *H*).

Now, let us observe the combination (*A*, *E*):

$$\delta(A, 0) = B$$
$$\delta(E, 0) = H$$

$\Big\}$ (*B*, *H*) is an equivalent state combination

Similarly, $\delta(A, 1) = \delta(E, 1) = F$

Therefore, states *A* and *E* are equivalent. Hence, we do not place an 'X' for the pair (*A*, *E*).

Thus, in one step, we have all pairs of equivalent states. Now we can reduce the STF Table 2.51 as shown in Table 2.54.

The minimized DFA can now be drawn as in Fig. 2.47(b).

2.14 PROPERTIES AND LIMITATIONS OF FSM

Listed here are a few important properties and limitations of an FSM:

Periodicity A limitation of an FSM is that it does not have the capacity to arbitrarily remember large amounts of information. Since it has only a fixed number of states, the length of a sequence that it can remember is limited.

Moreover, we have seen a finite control representation of the FSM, where the read head always moves one position to the right after reading an input symbol (refer to Section 2.3.6). The head can never move in the reverse direction. Therefore, the FSM cannot retrieve what it has read previously, before coming to the current position on the tape; and since it cannot retrieve anything that is read earlier, it cannot remember them. This also means that the FSM eventually will always repeat a state or produce a periodic sequence of states.

State determination Since the initial state of an FSM and the input sequence given to it determines the output sequence, it is always possible to discover the unknown state in which the FSM resides at a particular instance.

Impossibility of multiplication As we have seen, an FSM cannot remember arbitrarily long sequences. We know that during multiplication operation, it is required to remember two full sequences corresponding to the multiplier and the multiplicand. Moreover, while multiplying, it is also required to store the partial sums that are obtained during the intermediate stages. Therefore, no FSM can multiply, given any two arbitrarily long numbers. This is because, essentially, an FSM does not have any memory.

Impossibility of palindrome recognition No FSM can recognize a palindrome string, because it does not have the capability to remember all the symbols it reads until the half-way point of input sequence. Hence, it cannot match them in reverse order, with the symbols in second half of the sequence. This is true even if we assume that the given FSM can recognize the mid-point of the sequence, which is also actually impossible.

Impossibility to check if parentheses are well-formed The aforementioned reason holds here also. As an FSM has no capability to remember the earlier input symbols that it reads, it cannot compare with the remaining input symbols to check for well-formed parentheses. This is an impossible task for any FSM.

In order to accomplish all these complex jobs, we require a more capable and powerful machine, which is has a finite number of states, but unlimited memory to remember arbitrarily long sequences. Further, its head should be able to move to the left as well as to the right, that is, in both directions, so that it can read whatever it has stored on its tape. In other words, it should have a capability to retrieve whatever is stored in the memory.

Since the FSM lacks the ability to store, we can say that an FSM is a program without variables and without assignment statement.

2.15 ADDITIONAL FSM EXAMPLES

Let us now look at some more examples of FSM.

Example 2.26 Construct an NFA that accepts any positive number of occurrences of various strings from the following language L:

$L = \{x \mid x \text{ is made up of } \{a, b\}, \text{ and } x \text{ ends with } 'aab'\}$

Solution This is a typical sequence detector problem that we have earlier solved in the context of Mealy machines. Let us construct an NFA, which moves to a final state if

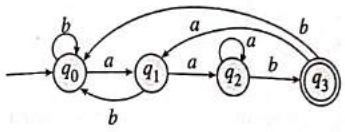


Figure 2.43 NFA as a sequence detector

the string matching the expected pattern is found; otherwise it moves to a non-final state. Figure 2.42 shows an NFA that works as a detector for the given sequence.

We observe from Fig. 2.43 that state q_0 loops on reading input symbol 'b' till it receives the input symbol 'a'. State q_1 proceeds to q_2 on reading input symbol 'a', but goes back to state q_0 on reading input symbol 'b'—as the sequence of choice 'aab' needs to begin with two 'a's.

Observe that state q_2 loops on input symbol 'a', as it has already seen two consecutive 'a's; any more 'a's is not a problem, as we are waiting for a 'b' after that—any string of the form '$aa...aaaaaab$' anyway ends in 'aab'.

State q_3 is the final state and can be reached upon reading the string that ends in 'aab'. In state q_3, if there is another input symbol 'a', it means that the string does not end in 'aab'; hence, it jumps to state q_1, on reading symbol 'a'. Similarly, state q_3 goes back to state q_0 upon reading symbol 'b', as the string does not end in 'aab'—it ends in '$aabb$'.

Note: We may notice that Figure 2.42 is actually a DFA. Every DFA is, in a way, a specialization of the NFA. The converse may not be true, though.

Example 2.27 Convert the Mealy machine in Fig. 2.44 to its equivalent Moore machine.

Solution We begin with four state labels, using $Q' = [Q \times \Delta]$, for the equivalent Moore machine that we want to construct.

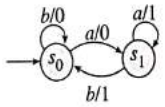


Figure 2.44 Example Mealy machine

$$Q' = \{[S_0, 0], [S_0, 1], [S_1, 0], [S_1, 1]\}$$

Hence, we have:

$$\begin{aligned}\delta'([S_0, 0], a) &= [\delta(S_0, a), \lambda(S_0, 0)] \\ &= [S_1, 0] \\ &= \delta'([S_0, 1], a)\end{aligned}$$

Similarly, the transitions for the remaining states are obtained. The STF and MAF tables for the equivalent Moore machine are shown in Table 2.55.

Table 2.55 STF and MAF tables for the equivalent Moore machine

STF (δ')			MAF (λ')	
Σ / Q'	a	b	Q'	Δ
$[S_0, 0]$	$[S_1, 0]$	$[S_0, 0]$	$[S_0, 0]$	0
$[S_0, 1]$	$[S_1, 0]$	$[S_0, 0]$	$[S_0, 1]$	1
$[S_1, 0]$	$[S_1, 1]$	$[S_0, 1]$	$[S_1, 0]$	0
$[S_1, 1]$	$[S_1, 1]$	$[S_0, 1]$	$[S_1, 1]$	1

Example 2.28 Design an NFA to recognize the set of strings, {'*abc*', '*abd*', '*aacd*'}. Assume that the input alphabet is $\{a, b, c, d\}$. Give the transition table for the NFA.

Solution This is also a sequence detector problem as in Example 2.26 discussed earlier in this section.

The difference between Example 2.26 and this example is that this FA, as in Fig. 2.45, is designed to accepted only the designated three strings, {'*abc*', '*abd*', '*aacd*'}; while the NFA in Example 2.26 accepts all strings that end in a specified substring, the NFA in Fig. 2.45 accepts only the finite set of strings.

The state transition table for the NFA can be drawn as shown in Table 2.56.

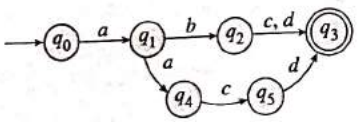


Figure 2.45 NFA accepting the given set of strings

Table 2.56 State transition table for the NFA

δ / Q	a	b	c	d
→ q_0	q_1	—	—	—
q_1	q_4	q_2	—	—
q_2	—	—	q_3	q_3
(q_3)	—	—	—	—
q_4	—	—	q_5	—
q_5	—	—	—	q_3

Example 2.29 Construct the NFA/DFA for the following languages:

(a) $L = \{x \mid \Sigma = \{a, b, c\}$; '$x$' contains exactly one '$b$' immediately following 'c'$\}$
(b) $L = \{x \mid \Sigma = \{0, 1\}$; '$x$' starts with '1' and $|x|$ is divisible by 3$\}$
(c) $L = \{x \mid \Sigma = \{a, b\}$; '$x$' contains any number of 'a's followed by at least one 'b'$\}$

Solution

(a) This is a sequence detector, where each string can contain exactly one 'b' immediately following 'c'. This means that the required machine must detect the substring 'cb'. Figure 2.46 provides the solution—a DFA that detects substring 'cb'.

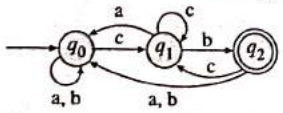


Figure 2.46 DFA that detects substring '*cb*'

(b) This is also a sequence detector problem, where each string should start with '1', and the length of the string (i.e., $|x|$) should be divisible by 3. As the string should start with '1', its length cannot be zero. Therefore, the minimum length of the string is 3, though zero is also divisible by 3.

If we begin with strings having length 3 (as this is the minimum length of the string), the allowed strings are: '100', '101', '110', '111'. For strings having length greater than 4, one can have any combination of 0's or 1's, with lengths in multiples of 3. The DFA that provides the solution is shown in Fig. 2.47.

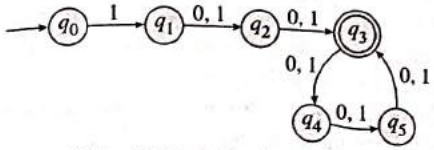


Figure 2.47 DFA that detects sequence required in Example (b)

We can see from Fig. 2.47 that state q_0, which is the initial state, makes a transition to state q_1 on reading input symbol '1', as it is expected to begin every string by '1'.

States q_1 and q_2 helps in recognizing the four combinations: '100', '101', '110', '111'. State q_3 is reached upon reading any of these 4 strings.

Beyond state q_3, the length of the string must be in multiples of 3, with any combination of 0's and 1's. This is signified by the loop '(0, 1) (0, 1) (0, 1)' which is repeated any number of times, in order to ensure that the length of the string is divisible by 3.

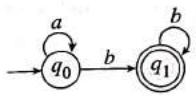


Figure 2.48 DFA that detects sequence in Example (c)

(c) The required sequence detector required can be depicted as shown in Fig. 2.48.

We observe that there is a loop labelled 'a' over the state q_0, which represents any number of a's, that is, zero or more number of a's. On reading one 'b', the DFA reaches the final state q_1. This state has a loop on 'b' in order to cater to more number of b's, as required.

2.16 TWO-WAY FINITE AUTOMATON

Two-way finite automaton (2FA) is similar to the FA that we have studied so far, except that these can read the input in either direction—left-to-right, or right-to-left. As we know, FA can read the input only from left-to-right (refer to Section 2.3.6 for details). The 2FA machines have a read head that can move in any direction from the current position, either left or right.

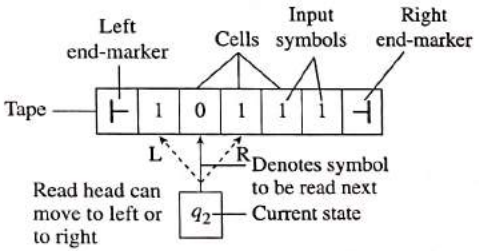


Figure 2.49 Finite control representation for 2FA

Though 2FA sounds to be more powerful than the normal single-way FA, its power is equivalent to that of the FA. The 2FA can accept only the regular sets as does the FA.

Finite control representation for 2FA is shown in Fig. 2.49.

Initially, the 2FA starts with the read head pointing to the cell containing the left end-marker. At any given point in time, the 2FA is in any state, say q_i, and makes a transition to some next state upon reading the current symbol on the tape cell. Moreover, unlike the FA, which can move only to one cell on the right, a 2FA can move to one cell left as well, after reading the current input symbol. The left and the right end-markers are the tape bounds, and at any given point in time, the read head cannot move beyond these two bounds. The input tape is thus finite just as that of the FA.

Since the 2FA head can move in any direction—either left or right—unlike the FA, there are certain differences in the 2FA formalism as well. For example, in case of an FA, the transition function only records the next state; while in the case of a 2FA, we need to record the direction (left or right) as well, in order to indicate whether it will point to the

Two or more states are said to be 'equivalent states' if they have the same transitions on the same input symbol, and this is true for all the transitions—but they should be of the same type, that is, either final or non-final. If there are multiple equivalent states in a given FSM, one can remove all but one and reduce (or minimize) the FSM.

FSMs lack the ability to store. We may say that an FSM is a program without variables and without assignment statement. Due to lack of memory, the FSM cannot solve problems such as checking whether a given input string is a palindrome or not; multiplication of numbers; and checking if parentheses are well-formed.

Two-way finite automata (2FA) are machines with a read head that can move in any direction from the current position—either towards the left or right. A 2FA is formally denoted by an eight-tuple (or octuple):

$M = (Q, \Sigma, \vdash, \dashv, \delta, q_0, q_A, q_R)$

where,

Q: Finite set of states

Σ: Finite input alphabet

q_0: Initial state of FA; $q_0 \in Q$

q_A: Accept halt (final) state of 2FA; $q_A \in Q$

q_R: Reject halt (final) state of 2FA; $q_R \in Q$

$\vdash$: Left end-marker symbol; $\notin \Sigma$

$\dashv$: Right end-marker symbol; $\notin \Sigma$

The 'δ' function for a 2DFA is given by:

$\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \rightarrow (Q \times \{L, R\})$

The 'δ' function for a 2NFA is given by:

$\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \rightarrow$ finite subsets of $(Q \times \{L, R\})$

Though the 2FA seems to be more powerful than an FA; it is in fact equivalent to the FA. Hence, a 2FA can only accept regular languages. The only difference between the two is that the FA reads the input symbols only in one direction—left-to-right—and not in the reverse. This makes the FA a special case of the 2FA. Any FA which is equivalent of a 2FA generally has exponentially more number of states than its equivalent 2FA.

EXERCISES

This section lists a few unsolved problems for the readers to help understand the topic better and practice some FSM construction examples.

Objective Questions

(E) 2.1 The smallest finite automata which accepts the language $\{x \mid$ length of x is divisible by 3$\}$ has
- (a) 2 states
- (b) 3 states
- (c) 4 states
- (d) 5 states

(U) 2.2 Which of the following is true?
- (a) DFA and NFA have same power
- (b) DFA is more powerful than NFA
- (c) NFA is more powerful than DFA
- (d) All of these

(U) 2.3 Which of the following is true about finite automata?
- (a) It has no memory
- (b) A finite automata can have more than one initial state
- (c) Finite automaton uses stack as a memory
- (d) All of these

(R) 2.4 The language accepted by DFA is called a _____ language.

(R) 2.5 NFA means _____.

(A) 2.6 Which of the following statements are true for the NFA: $(\{p, q, r, s\}, \{0, 1\}, \delta, p, \{q, s\})$, where '$\delta$' is given by:

Table 2.58 State transition table

Q \ Σ	0	1
p	q, r	q
q	r	q, r
r	s	p
s	—	p

- (a) NFA accepts the string 00
- (b) NFA does not accept the string 001
- (c) NFA accepts the string 1111110
- (d) All of these
- (e) None of these

(A) 2.7 If M is a DFA accepting a language consisting of 0's and 1's that end in either '00' or '11'. What is the minimum number of states in M?
- (a) 2
- (b) 3
- (c) 4
- (d) 5
- (e) 6

(U) 2.8 Every DFA is also an NFA. Is this statement true or false?

Review Questions

(E) 2.1 Construct Mealy and Moore machines for the following:
For the input from Σ^*, where $\Sigma = \{0, 1, 2\}$ print the residue-modulo-5 of the input treated as a ternary (base 3 with digits 0, 1, and 2) number.

(L) 2.2 Discuss the relative powers of NFA and DFA.

(A) 2.3 Write the machine function and the state transition function for a binary adder. Support your answer with a transition diagram.

(U) 2.4 Prove the following statement: 'Corresponding to every transition graph, there need not exist an FSM, but the converse is always true'.

(R) 2.5 Define and give suitable examples for a transition graph.

(E) 2.6 Construct a Mealy machine that accepts the strings from $(0 + 1)^*$ and produces the following output:

Table 2.59 Output

End of string	Output
101	x
110	y
Otherwise	z

(L) 2.7 Explain whether a language of palindromes is accepted by an FSM. Justify.

(U) 2.8 Describe the following:
- (a) State equivalence
- (b) FSM equivalence

(U) 2.9 Write a short note on Mealy and Moore machines.

(U) 2.10 Explain with an example, the process of converting a Mealy machine to its corresponding Moore machine.

(L) 2.11 Write a short note on the properties and limitations of FSM.

(L) 2.12 Compare Moore and Mealy machines.

(E) 2.13 Design an FSM to check divisibility by three, where $\Sigma = \{0, 1, \ldots, 9\}$.

(A) 2.14 What are finite automata? Construct the minimum state automata equivalent to the state transition diagram in Fig. 2.51.

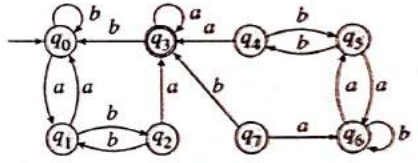


Figure 2.51 Example automata

(A) 2.15 Construct NFA without ϵ-transitions for the NFA with ϵ-transitions shown in Fig. 2.52.

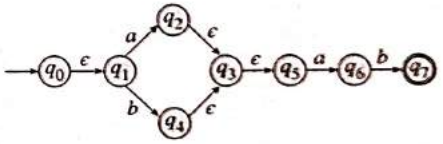


Figure 2.52 Example NFA with ϵ-transitions

(E) 2.16 Design an FSM that reads strings made up of letters in the word 'CHARIOT' and recognizes those strings that contain the word 'CAT' as a substring.

(A) 2.17 Construct a Moore machine equivalent to the Mealy machine represented by the TG in Fig. 2.53.

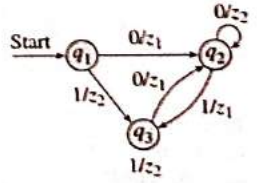


Figure 2.53 Example Mealy machine

Ⓐ 2.18 Consider the DFA as shown in Fig. 2.54. Obtain the minimum state DFA.

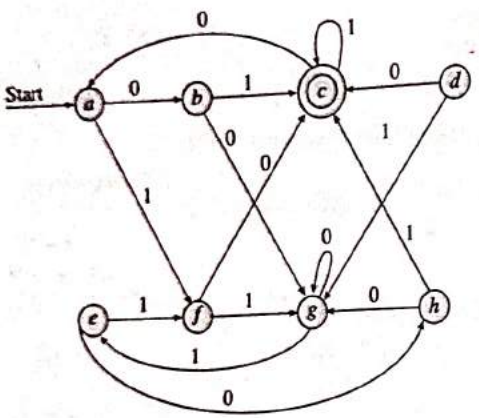


Figure 2.54 Example DFA

Ⓐ 2.19 Consider the Moore machine described by the transition table given here. Construct the corresponding Mealy machine.

Table 2.60 Example Moore Machine

Current state	Next state $a=0$	Next state $a=1$	Output
$\rightarrow q_1$	q_1	q_2	0
q_2	q_1	q_3	0
q_3	q_1	q_3	1

Ⓐ 2.20 Construct a DFA equivalent to the NFA: $(\{p, q, r, s\}, \{0, 1\}, \delta, p, \{q, s\})$, where '$\delta$' is given by:

Table 2.61 Example NFA

Q \ Σ	0	1
p	q, r	q
q	r	q, r
r	s	p
s	—	p

Ⓤ 2.21 Write and explain all the steps required for the conversion of an NFA to a DFA using a suitable example.

Ⓐ 2.22 Convert the Mealy machine in Fig. 2.55 to a Moore machine.

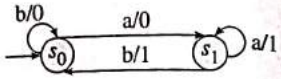


Figure 2.55 Example Mealy machine

Ⓐ 2.23 Consider the following NFA with ϵ-transitions. Assume 'p' to be the initial state and 'r' as the final state.

Table 2.62 Example NFA

	ϵ	a	b	c
p	ϕ	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	ϕ
r	$\{q\}$	$\{r\}$	ϕ	$\{p\}$

(a) Compute the ϵ-closure of each state
(b) List all the strings of length three or less accepted by the automata
(c) Convert the automaton to its equivalent DFA

Ⓐ 2.24 Construct a DFA for the NFA, whose state transition function is given here. Assume 'p' to be the initial state and $F = \{q, r\}$.

Table 2.63 Example NFA

	0	1
p	$\{p, q\}$	ϕ
q	r	s
f	s	ϕ
s	s	s

Ⓤ 2.25 Explain Moore and Mealy machines using suitable examples. How do we construct the equivalent Mealy machine for a given Moore machine? Give a suitable example.

Ⓔ 2.26 Compare Mealy and Moore machines. Design a Mealy machine to replace each occurrence of sub-string '*abb*' by '*aba*', where $\Sigma = \{a, b\}$.

Ⓔ 2.27 Design a Moore machine that changes all the vowels to '$'.

Ⓐ 2.28 Construct a DFA equivalent to the NFA: $(\{p, q, r, s\}, \{0, 1\}, \delta_N, p, \{q, s\})$, where δ_N is as given in the following table:

Table 2.64 Example NFA

Q \ Σ	0	1
$\rightarrow p$	$\{q, r\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	–	$\{p\}$

Ⓤ 2.29 Write short notes on:
(a) Deterministic finite automata
(b) Moore and Mealy machines
(c) Moore's algorithm for FSM equivalence
(d) Relative powers of NFA and DFA
(e) Limitations of FSM

Ⓡ 2.30 Give formal definitions for the following:
(a) Deterministic finite automata
(b) NFA with ϵ-transitions
(c) Moore machine
(d) Acceptance of a string by FA

Ⓐ 2.31 Translate the Mealy machine in Fig. 2.56 into its equivalent Moore machine.

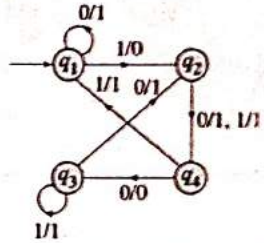


Figure 2.56 Example Mealy machine

Ⓤ 2.32 Convert the following NFA into NFA without ϵ-moves.

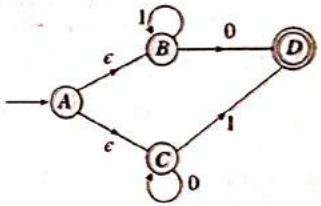


Figure 2.57 Example NFA with ϵ-moves

Answers to Objective Questions

2.1. (b) 2.2. (a) 2.3. (a) 2.4. regular 2.5. non-deterministic finite automata
2.6. (a) 2.7. (b) 2.8. True

3 Regular Expressions

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Concept of regular expressions (REs)
- Formal definition of regular expressions
- Writing regular expressions from a given language description
- Equivalence of regular expressions and finite automata
- Construction of a deterministic finite automata (DFA) from the given regular expression
- Obtaining a regular expression for the language accepted by a DFA
- Arden's theorem
- Closure properties of regular languages (or sets)
- Pumping lemma for regular languages
- Applications of regular expressions/finite automata
- Myhill–Nerode theorem and Kleene's theorem

3.1 INTRODUCTION

The languages accepted by finite automata (FA) are described or represented by simple expressions called *regular expressions* (RE). Since FA accepts regular languages (RL), regular expressions are also used to denote regular languages.

Regular expressions are like the short-form notations that denote regular languages (or regular sets). This is analogous to the set labels such as I, which denotes the set of integers, and ϕ, which denotes an empty set. The only difference in the case of regular expressions is that these are composed of few operators as well; hence, the term 'expressions'. As expressions are composed of some operands and some operators, it can be concluded that regular expressions are short notations that can even denote complex and infinite regular languages.

3.2 REGULAR EXPRESSION FORMALISM

The class of regular expressions over Σ is defined recursively as follows:

1. Regular expressions over Σ, include letters, ϕ (empty set), and ϵ (empty string of length zero).
2. Every symbol $a \in \Sigma$ is a regular expression over Σ.
3. If R_1 and R_2 are regular expressions over Σ, then so are $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and $(R_1)^*$, where '+' indicates *alternation* (parallel path), the operation '.' denotes *concatenation* (series connection), and '*' denotes *iteration* (*closure* or repetitive concatenation).
4. Regular expressions are only those that are obtained using rules 1–3.

As per the definition, ϕ, ϵ, and every input symbol from the input alphabet Σ, itself is a regular expression.

The expression, '$R_1 + R_2$' means *OR-ing* the two regular expressions, which is similar to the *exclusive OR* that we know. This operation is analogous to the 'if ... else ...' construct in programming languages. When the program is executed, only one of the two program paths is followed, based on whether the condition is true or false.

The expression '$R_1 \cdot R_2$' means R_1 followed by R_2. This is analogous to sequential execution in programmes—one step follows the other.

The expression 'R_1^*' denotes zero or more occurrences of R_1, that is, repetitive (zero or more times) concatenation of R_1 to itself. Even an empty string ϵ is included herein, if R_1 is repeated zero times. This is analogous to the looping construct or recursion in programming languages.

The expression 'R_1^+' is used to denote *positive closure* of R_1, and represents one or more occurrences of R_1. This means that at least one occurrence of R_1 is assured.

We can write this as: $R^+ = R \cdot R^*$.

Hence, we see that R_1^+ can be composed out of other primitive operations, and so it is not considered in the regular expression formalism.

Parallel paths, series connection, and repetitive concatenation can be explained with the help of transition graphs (TGs) as shown in Fig. 3.1. The TG in Fig. 3.1(a) represents the parallel paths operation, '$a + b$', which stands for either a or b. Similarly, Fig. 3.1(b) represents the series operation, '$a \cdot b$', which stands for a followed by b; and Fig. 3.1(c) represents repetitive concatenation, or the closure operation, 'a^*', which stands for any number of occurrences of a, with the help of a self-loop.

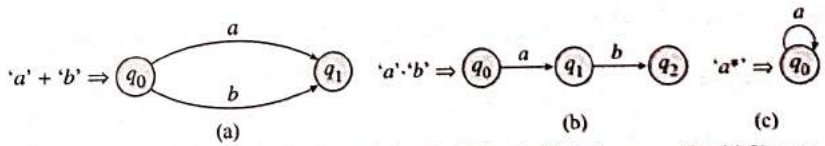


Figure 3.1 Operators in regular expressions (a) Parallel paths (b) Series connection (c) Closure

If r is a regular expression, then the language represented by r is denoted by $L(r)$.

For example:

If $r = a + b$, then $L(r) = \{a, b\}$.
If $r = a \cdot b$, then $L(r) = \{ab\}$
If $r = a^*$, then $L(r) = \{c, a, aa, aaa, aaaa, \ldots\}$

Here c stands for zero occurrences of a. Note that a^* denotes an infinite set of strings, which includes all possible strings that can be composed out of a.

3.3 EXAMPLES OF REGULAR EXPRESSIONS

Let us look at some examples of regular expressions, and attempt to write regular expressions using a given regular language description.

Example 3.1 Using a regular expression, describe the language consisting of all strings over $\Sigma = \{0, 1\}$ with at least two consecutive 0's.

Solution The set of all strings over $\Sigma = \{0, 1\}$ is given as:

$$\Sigma^* = \{c, 0, 1, 00, 01, 10, 11, 000, 001, 010, \ldots\}$$

This set of all strings over $\Sigma = \{0, 1\}$ can be represented by the regular expression, $(0 + 1)^*$. This set includes all possible combinations of 0's and 1's.

However, we require at least one occurrence of '00', that is, two consecutive 0's. We might have any number of trailing 1's and 0's, and any number of leading 1's and 0's.

Therefore, the required regular expression for the language described is:

$$r = (0 + 1)^* \cdot 0 \cdot 0 \cdot (0 + 1)^*$$

Example 3.2 Using a regular expression, represent the language defined over $\Sigma = \{0, 1, 2\}$, such that every string from the language contains 'any number of 0's followed by any number of 1's followed by any number of 2's'.

Solution 'Any number of '0's' means zero or more occurrences of 0's. This can be denoted by '0^*'. Similarly, 'any number of 1's' can be denoted by '1^*', and 'any number of 2's' by '2^*'.

Therefore, the regular expression is given by:

$$r = 0^* \cdot 1^* \cdot 2^*$$

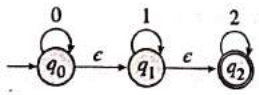


Figure 3.2 NFA with c-moves accepting language, $0^* \cdot 1^* \cdot 2^*$

We may recall that in Chapter 2 (refer to Section 2.7), we have seen an NFA with c-moves that accepts the same language. The equivalent NFA is shown in Fig. 3.2.

Example 3.3 If $L(r)$ = set of all strings over $\Sigma = \{0, 1, 2\}$, such that at least one 0 is followed by at least one 1, which is followed by at least one 2, find a regular expression r representing this language.

Solution This is very similar to the previous problem. The only difference here is that we require at least one occurrence of each symbol.

The phrase, 'at least one occurrence' means one or more occurrences of the symbol.

Now, at least one occurrence of 0 can be represented by '$0 \cdot 0^*$', or 0^+. We similarly represent at least one occurrence of 1's and 2's as well.

Therefore, we can write r as:

$$r = 0 \cdot 0^* \cdot 1 \cdot 1^* \cdot 2 \cdot 2^*, \text{ or}$$
$$r = 0^+ \cdot 1^+ \cdot 2^+$$

Example 3.4 Using a regular expression, represent the language over $\Sigma = \{a, b\}$ with all strings starting and ending with a's and with any number of b's in between.

Solution In this language, each string must begin and end with an a, and in between there may be zero or more number of b's.

Therefore, we may write $L(r)$ as:

$$L(r) = \{aa, aba, abba, abbba, \ldots\}$$

Hence, the regular expression for this language is:

$$r = a \cdot b^* \cdot a$$

Example 3.5 If $L(r)$ = set of all strings over $\Sigma = \{0, 1\}$ ending with '011', then find r.

Solution In this language, every string contains any combination of 0's and 1's, and always ends in '011'.

Therefore, we may write $L(r)$ as:

$$L(r) = \{011, 0011, 1011, \ldots\}$$

Hence, the regular expression r is:

$$r = (0 + 1)^* \cdot 011$$

Example 3.6 Describe in simple English the language represented by the regular expression

$$r = (1 + 10)^*$$

Solution Let us try to list out all the strings in the language described by r:

$$L(r) = \{c, 1, 10, 11, 101, 110, 1010, \ldots\}$$

As per the regular expression, we have two parallel paths—'1' and '10', which are put into an iteration (or loop), that is, zero or more number of occurrences.

The empty string c is obtained if we consider zero occurrences.

We get string '1' if we choose 1 from the two parallel paths, and consider only one occurrence. Similarly, we get string '10' if we choose the other path.

The string '11' is obtained if we choose path '1' for both the iterations.

The string '101' is obtained if we consider two occurrences (or iterations)—path '10' for the first iteration, and path '1' for the next.

Hence, we observe that this is an infinite language.

Now, let us attempt to describe the language looking at the string forms we have listed:

We see that the set $L(r)$ can be described as a language over $\Sigma = \{0, 1\}$ having strings beginning with '1', and not having two consecutive 0's.

Example 3.7 Represent the language over $\Sigma = \{0, 1\}$ containing all possible combinations of 0's and 1's, but not having two consecutive 0's.

Solution We have seen in the previous example that the language containing strings that start with '1' and do not have two consecutive 0's is represented by the regular expression:

$$r_1 = (1 + 10)^* \tag{3.1}$$

Similarly, the language containing strings that start with '0' and do not have two consecutive 0's can be represented as:

$$r_2 = 0 \cdot (1 + 10)^* \tag{3.2}$$

Combining Eqs (3.1) and (3.2), we write the required regular expression as:

$$r = (1 + 10)^* + 0 \cdot (1 + 10)^*$$

This can be simplified as:

$$r = (0 + \epsilon) \cdot (1 + 10)^*$$

Example 3.8 If $r = ab^*a$, describe $L(r)$ in the form of a set.

Solution $L(r)$ can be described in the form of a set of strings as shown here:

$$L(r) = \{aa, aba, abba, abbba, \ldots\}$$

The string *aa* is obtained considering zero occurrences of the middle string b. The remaining strings, such as *aba*, *abba*, and so on, are obtained considering one, two, or more occurrences of b respectively.

Example 3.9 Show that $(a \cdot b)^* \neq a^* \cdot b^*$

Solution Let $r_1 = (a \cdot b)^*$, and

$$r_2 = a^* \cdot b^*$$

Let us try listing the sets $L(r_1)$ and $L(r_2)$:

$$L(r_1) = \{\epsilon, ab, abab, ababab, \ldots\} \tag{3.3}$$

$L(r_2)$ is the concatenation of two sets u and v; where,

$u = \{\epsilon, a, aa, aaa, \ldots\}$ that is represented by a^*; and
$v = \{\epsilon, b, bb, bbb, \ldots\}$, which is represented by b^*.

Therefore,

$$\begin{aligned} L(r_2) &= u \cdot v \\ &= \{\epsilon, a, aa, aaa, \ldots\} \cdot \{\epsilon, b, bb, bbb, \ldots\} \\ &= \{\epsilon, a, b, aa, bb, ab, \ldots\} \end{aligned} \tag{3.4}$$

Comparing Eqs (3.3) and (3.4), we can say that:

$$(a \cdot b)^* \neq a^* \cdot b^*$$

Example 3.10 If $L(r) = \{a, c, ab, cb, abb, cbb, abbb, \ldots\}$, what is r?

Solution We observe that the strings in $L(r)$ either begin with a or c and are followed by zero or more occurrences of b.

Hence, the regular expression r that denotes this set is:

$$r = (a + c) \cdot b^*$$

Example 3.11 If $L(r) = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$, find the regular expression r which represents $L(r)$.

Solution We observe that the length of each word in $L(r)$ is three, and $L(r)$ depicts all possible combinations of a's and b's.

We also observe that each letter in any word is either a or b, that is, $(a + b)$.

Therefore, the regular expression r can be written as:

$$\begin{aligned} r &= (a + b) \cdot (a + b) \cdot (a + b) \\ &= (a + b)^3 \end{aligned}$$

However, $(a + b)^3$ is not a formal notation. Hence, $(a + b) \cdot (a + b) \cdot (a + b)$ is the preferred answer.

Example 3.12 Represent the set of all words over $\Sigma = \{a, b\}$ containing at least one a, using a regular expression.

Solution According to the language description, every string must contain at least one a, with any number (zero or more) of a's and b's before and after it.

Thus, the regular expression may be written as:

$$r = (a + b)^* \cdot a \cdot (a + b)^*$$

Example 3.13 Let $r = (a + b)^* \cdot a \cdot (a + b)^* \cdot a \cdot (a + b)^*$. Describe the language $L(r)$ represented by the given regular expression using simple English.

Solution We can easily see that the regular expression r denotes the languages $L(r)$ such that:

$L(r)$ = language over $\Sigma = \{a, b\}$ containing at least two a's

Note: Consider the regular expression:

$$r_1 = b^* \cdot a \cdot b^* \cdot a \cdot (a + b)^*$$

We see that r_1 also generates the same language as that of r. Therefore, regular expressions r and r_1 are equivalent; so we can write:

$$(a + b)^* \cdot a \cdot (a + b)^* \cdot a \cdot (a + b)^* = b^* \cdot a \cdot b^* \cdot a \cdot (a + b)^*$$

Readers may verify this by generating both the languages as an exercise.

Example 3.14 If $r = b^* \cdot a \cdot b^* \cdot a \cdot b^*$, describe $L(r)$ in simple English.

Solution We see that the language description can be written as:

$L(r)$ = language over $\Sigma = \{a, b\}$ containing exactly two a's

---

Example 3.15 Represent the language over $\Sigma = \{a, b\}$ containing at least one a and at least one b, using a regular expression.

Solution Referring to Example 3.13, we may write the regular expression as:

$$r_1 = (a + b)^* a (a + b)^* b (a + b)^*$$

Please note that we have not shown the operator for concatenation '$\cdot$' in the expression, as that is assumed if one symbol follows another.

The language may also be represented using the regular expression r_2, as the position of a and b can be interchanged:

$$r_2 = (a + b)^* b (a + b)^* a (a + b)^*$$

Thus, the end result is either of the aforementioned forms, and hence can be written as:

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

Example 3.16 Show that $(a + b)^* = (a + b)^* + (a + b)^*$

Solution

Let $r_1 = (a + b)^*$ then, $L(r_1) = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$ (3.5)

Let $r_2 = \underbrace{(a + b)^*}_{p} + \underbrace{(a + b)^*}_{q}$

Now, $L(p) = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$,

and $L(q) = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$

Therefore,

$$L(r_2) = L(p) \cup L(q) = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\} \quad (3.6)$$

Comparing Eqs (3.5) and (3.6), we can say:

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

Note: Similarly, we can also show that:

1. $(a + b)^* = (a + b)^* \cdot (a + b)^*$
2. $(a + b)^* = a (a + b)^* + b (a + b)^* + \epsilon$
3. $(a + b)^* = (a + b)^* ab (a + b)^* + b^* a^*$

Example 3.17 Using a regular expression, represent the set of all strings of a's and b's containing at least one combination of double letters.

Solution A double letter combination can either be aa or bb.

Therefore, the regular expression can be written as:

$$r = (a + b)^* \cdot (aa + bb) \cdot (a + b)^*$$

Example 3.18 If $L(r) = \{\epsilon, x, xx, xxx, xxxx, xxxxx\}$, what is r?

Solution The regular expression can be written as:

$$r = (\epsilon + x) \cdot (\epsilon + x) \cdot (\epsilon + x) \cdot (\epsilon + x) \cdot (\epsilon + x) = (\epsilon + x)^5$$

Example 3.19 Let $L(r)$ = set of all strings over $\Sigma = \{a, b\}$ in which the strings either contain all b's or else, there is an a followed by some b's; the set also contains $\in$. Find the regular expression that represents this language.

Solution We may write the required $L(r)$ as:

$$L(r) = \{\in, a, b, ab, bb, abb, bbb, \ldots\}$$

Therefore,

$$r = b^* + a \cdot b^* = (\epsilon + a) \cdot b^*$$

Example 3.20 Find the regular expression for the language consisting of all strings of a's and b's without any combination of double letters.

Solution The required regular expression is given by:

$$r = (\epsilon + b) \cdot (ab)^* \cdot (\epsilon + a)$$

Note: We want all such strings that do not contain double letters. There are only two patterns that, if iterated, do not generate double letters; they are: ab and ba. The aforementioned regular expression uses the pattern ab, which is iterated zero or more times. Any string that is the outcome of $(ab)^*$ never begins with b, and never ends with a. However, the strings can start with either a or b and end in either a or b. Hence, '$(\epsilon + b)$' is concatenated in the beginning, and '$(\epsilon + a)$' at the end of the regular expression.

Using another pattern, that is, ba, we may write the regular expression as:

$$r = (\in + a) \cdot (ba)^* \cdot (\in + b)$$

Both are the equivalent regular expressions though they appear to be different.

Example 3.21 Show that $(a^* b^*)^* = (a + b)^*$

Solution

Let $r_1 = (a^* b^*)^*$, and $r_2 = (a + b)^*$.

Then, $L(r_2) = \{c, a, b, aa, ab, ba, bb, \ldots\}$ (3.7)

$$L(r_1) = \{(c, a, aa, \ldots) \cdot (c, b, bb, \ldots)\}^*$$
$$= \{c, a, b, aa, bb, \ldots\}^*$$
$$= \{c, a, b, aa, bb, ab, ba, \ldots\} \quad (3.8)$$

From Eqs (3.7) and (3.8), it follows that:

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

Note: Similarly, we can also show that:

1. $(a^* + b^*)^* = (a + b)^*$
2. $(ab)^* a = a (ba)^*$

Example 3.22 Represent the language that contains strings over $\Sigma = \{0, 1\}$, and has even number of 0's.

Solution An even number of 0's means either 0, 2, 4, 6, … number of 0's.

Hence, the required regular expression is:

$$r = (1^* \cdot 0 \cdot 1^* \cdot 0 \cdot 1^*)^* + 1^*$$

The path for 1^* ensures there are no 0's, while the path for $(1^* \cdot 0 \cdot 1^* \cdot 0 \cdot 1^*)^*$ ensures 2, 4, 6, … numbers of 0's.

We observe that for the path $(1^* \cdot 0 \cdot 1^* \cdot 0 \cdot 1^*)^*$, the two 0's are placed in such a way that they are preceded, followed, and separated by zero or more number of 1's. This generates all possible strings as required.

3.4 EQUIVALENCE OF REGULAR EXPRESSIONS AND FINITE AUTOMATA

Regular expressions denote a regular set, that is, the language accepted by some finite automata. Further, for every regular expression there exists an equivalent FA, which accepts the same language denoted by the regular expression. This is known as Kleene's theorem.

3.4.1 Kleene's Theorem

Kleene's theorem is stated in two parts:

1. Any regular language is accepted by a finite automaton.
2. Languages accepted by FA are regular.

Proof

1. Section 3.4.2 establishes the equivalence among regular expressions and FA. The way the regular languages are recursively defined is the reason why the regular expression is so constructed. One may refer to the definition of regular language in Section 3.5.1
2. Section 3.4.3 describes the method used to obtain the regular expression denoting the regular language accepted by any FA.

3.4.2 Regular Expression to FA Conversion

Our eventual goal is to construct a DFA that accepts the language denoted by the given regular expression. As per the mechanical process (algorithm), we first need to obtain the NFA with ϵ-transitions, convert it to NFA, which in turn can be converted to its equivalent DFA. Alternately, we can directly construct the DFA from the constructed NFA with ϵ-transitions as discussed in Chapter 2 (refer to sections 2.6 to 2.9).

Regular Expression to NFA with ϵ-moves Conversion

If there is a simple regular expression, we can directly construct its equivalent DFA/NFA without much trouble. However, if the regular expression is complicated, then it is not possible to draw the FA by just looking at it.

There are certain rules to convert a given regular expression to its equivalent NFA with ϵ-moves, which can then be transformed into its equivalent NFA or directly to its equivalent DFA, by the known methods. The rules for converting a given regular expression to its equivalent NFA with ϵ-moves are depicted in Fig. 3.3.

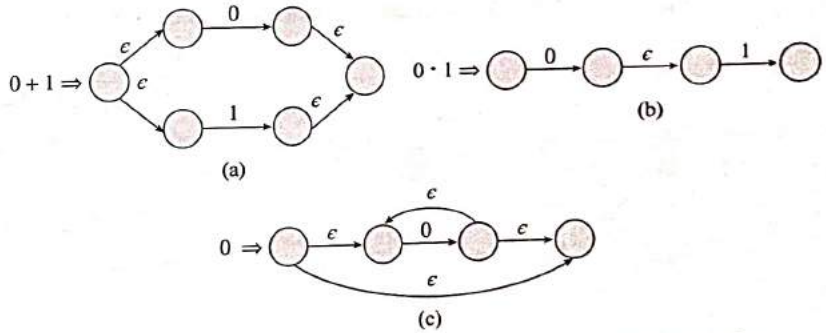


Figure 3.3 Rules for constructing NFA with ϵ-moves from given regular expression (a) Parallel paths (b) Series (c) Closure

Figure 3.3(a) shows how the '+' operator is converted to parallel paths.

One might question the use of ϵ-moves in the figure: The figure illustrates a very simple example of '$r_1 + r_2$', where r_1 is 0 and r_2 is 1. In reality, these can be complex expressions themselves. In such a case, we introduce a new start state that connects with the initial states of the individual FA representing r_1 and r_2 using the ϵ-moves. The individual final states of two FA are also similarly connected with a new final state.

Figure 3.3(b) shows how a final state of the FA for r_1 is connected to the start state for r_2; here r_1 is considered as 0 and r_2 as '1'. It is very important to note here that if these

thumb rules are applied to obtain the NFA with ϵ-moves, then each NFA with ϵ-moves will always has a single final state.

Figure 3.3(c) shows a transition from the first state to the last state on ϵ, which represents zero occurrences of r—taken here as 0. This is done to bypass r. The other path from the first state to the last state, which goes through some intermediate states, represents one or more occurrences of r. In all, the figure depicts 'zero or more' occurrences of r.

In order to represent r^*, we introduce a new start state and a new ending state: for this, we introduce a path from the start to the final state using ϵ to denote zero occurrences. We then connect the new start state to the start state of the FA for r using ϵ. Likewise, we connect the final state of the FA for r to a new end state using ϵ. Thus, the new FA denotes one occurrence of r. We then connect the original final state of FA for r to its original initial state to achieve more than one occurrence. Overall, the new NFA with ϵ-moves thus constructed represents zero or more occurrences of r, that is, r^*.

Similarly, positive closure of 0, that is, 0^+ is represented as an NFA with ϵ-moves as shown in Fig. 3.4. This can be generalized to represent any complex r. Comparing Fig. 3.3(c) and Fig. 3.4, we note that the path from the new initial to new final state on symbol ϵ, representing zero occurrences is removed now.

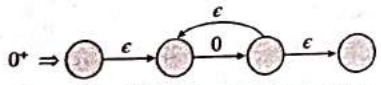


Figure 3.4 NFA with ϵ-moves representing positive closure of 0

Example 3.23 Draw an NFA with ϵ-moves for the regular expression, $r = a \cdot (a + b)^*$, which represents the language consisting of strings of a's and b's, starting with a.

Solution Using the rules given in Fig. 3.3, the steps for converting the given regular expression to NFA with ϵ-moves are shown in Fig. 3.5. Figure 3.5(c) shows the required NFA.

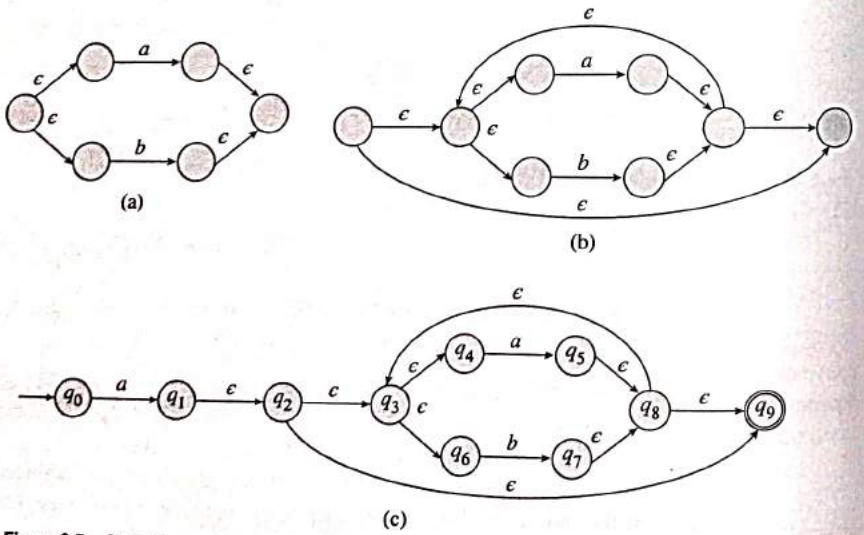


Figure 3.5 Steps for constructing NFA with ϵ-moves for $a \cdot (a + b)^*$ (a) Step 1: NFA with ϵ-moves for $(a + b)$ (b) Step 2: NFA with ϵ-moves for $(a + b)^*$ (c) Step 3: Final NFA with ϵ-moves for $a \cdot (a + b)^*$

Example 3.24 Draw the NFA with ϵ-moves for the regular expression $r = (a^* + b^*)$.

Solution Using the rules for converting the given regular expression to NFA with ϵ-moves, we construct the NFA with ϵ-moves for $(a^* + b^*)$ as shown in Fig. 3.6.

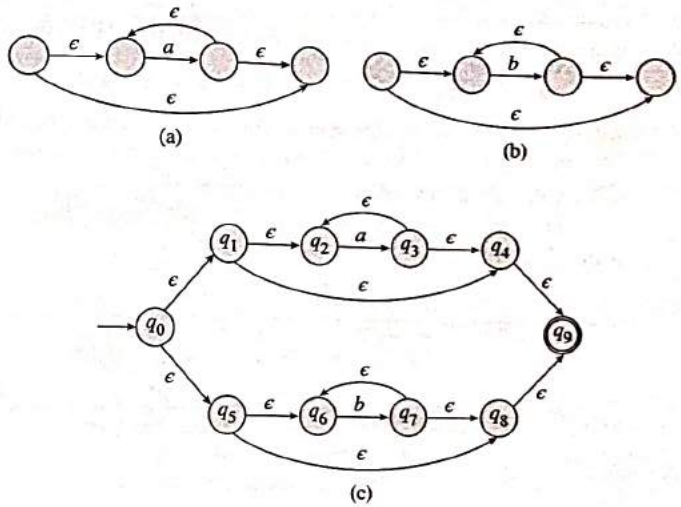


Figure 3.6 NFA with ϵ-moves for regular expression $(a^* + b^*)$ (a) Step 1: a^* (b) Step 2: b^* (c) Step 3: $(a^* + b^*)$

Regular Expression to NFA Conversion

As discussed earlier, we can obtain the equivalent NFA with ϵ-moves from a given regular expression; this can then be transformed into NFA without ϵ-moves as discussed in Chapter 2 (refer to Section 2.8).

We may recall that the rule for obtaining the state function δ' for the equivalent NFA without ϵ-moves from given NFA with ϵ-moves is:

$$\delta'(q, a) = \epsilon\text{-closure}\ \{\delta\ [\hat{\delta}(q, \epsilon), a]\}$$

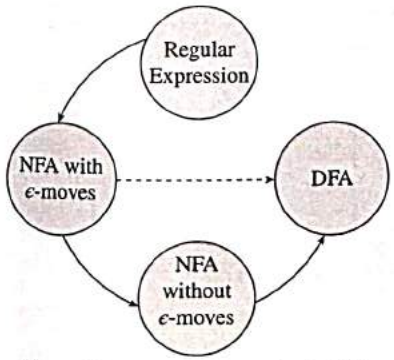


Figure 3.7 Regular expression to DFA conversion

Regular Expression to DFA Conversion

There are many ways to obtain the equivalent DFA from a given regular expression (refer to Fig. 3.7). The different methods are as follows:

1. Obtain from the given regular expression, the NFA with ϵ-transitions, and by the direct method (refer to Chapter 2, Section 2.9.2), which uses the reachable states technique, convert it to its equivalent DFA.

2. Obtain the NFA with ϵ-transitions from the given regular expression. Convert it to its equivalent NFA without ϵ-moves. Convert this NFA to DFA (refer Chapter 2, Section 2.6). There are again two methods of conversion from NFA to DFA that we have discussed in Chapter 2. Use any one of these two methods.

Example 3.25 Construct a transition graph that recognizes the set:

$$r = [1 \cdot (00)^* \cdot 1 + 0 \cdot 1^* \cdot 0]^*$$

Solution The problem statement requires us to construct a TG, which means constructing a TG for its equivalent NFA with ϵ-moves is sufficient. However, let us also construct the equivalent DFA for the given regular expression.

As a first step, we construct the equivalent NFA with ϵ-moves using the rules, as shown in Fig. 3.8.

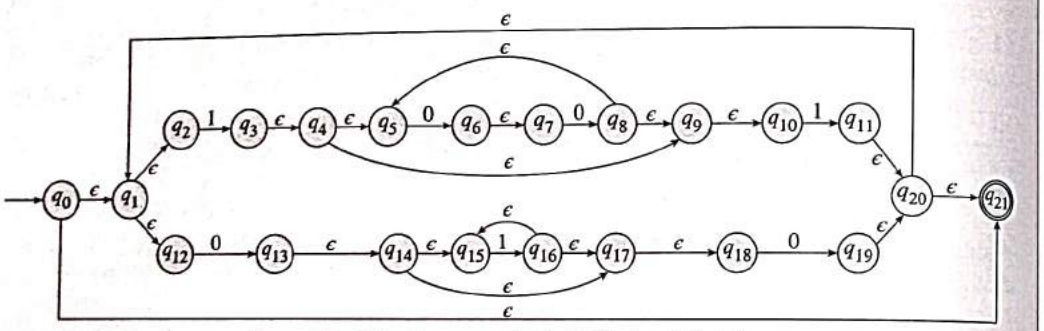
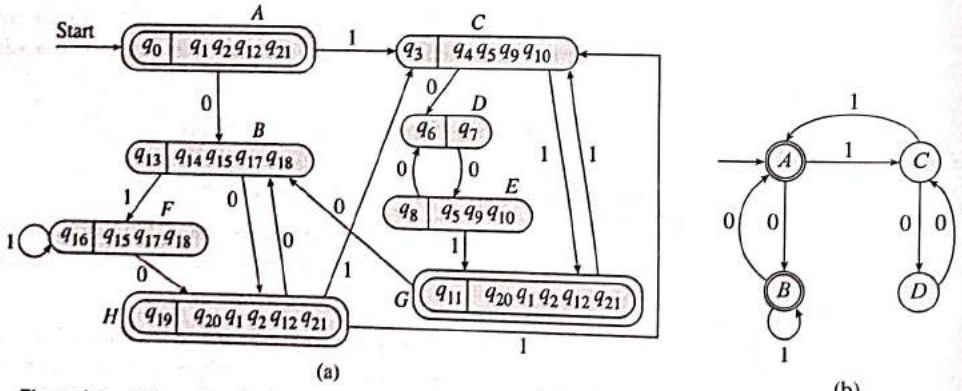


Figure 3.8 NFA with ϵ-moves for $[1 \cdot (00)^* \cdot 1 + 01^* \cdot 0]^*$

Using this NFA with ϵ-transitions, we can obtain the equivalent DFA through the direct method, as shown in Fig. 3.9(a).



(a)

(b)

Figure 3.9 DFA construction from NFA with ϵ-moves (a) DFA directly obtained from NFA with ϵ-moves in Fig. 3.8 (b) Final DFA for the regular expression $[1 \cdot (00)^* \cdot 1 + 01^* \cdot 0]^*$

We begin with the initial state q_0 of the given NFA with ϵ-moves, and collect all reachable states (path labelled by ϵ) from q_0. These are q_1, q_2, q_{12}, and q_{21}. As q_{21} is the final state of the NFA and reachable from q_0, we mark this new state as final. Out of states q_1, q_2, q_{12}, and q_{21}, we see that there are no transitions on 0 or 1 from q_1. Similarly, there are no transitions from q_{21} also, as it is the final state. However, q_2 goes to q_3 on reading 1, while q_{12} goes to q_{13} on reading 0.

Table 3.1 State transition table for DFA in Fig. 3.9(a)

Q \ Σ	0	1
*A	B	C
B	H	F
C	D	G
D	E	—
E	D	G
F	H	F
*G	B	C
*H	B	C

'*': Final states

Table 3.2 Reduced state transition table for DFA

Q \ Σ	0	1
*A	B	C
B	A•	B•
C	D	A•
D	C•	—

'*': Final states
'•': Modified entries

Therefore, we create two new state symbols: q_3, whose reachable states are q_4, q_5, q_9, and q_{10}; and q_{13}, whose reachable states are q_{14}, q_{15}, q_{17}, and q_{18}. We repeat the same process for the newly-created states q_3 and q_{13}, and continue till we reach to the stage where no more new states can be added.

Finally, we obtain eight states in the equivalent DFA, and relabel these states from A to H, as shown in Fig. 3.9 (a). Using these new labels, the state transition table for the equivalent DFA can be constructed as shown in Table 3.1.

We observe that G and H are equivalent to A; F is equivalent to B, and E is equivalent to C. The reduced state transition table after minimization based on the equivalent states is shown in Table 3.2.

Using Table 3.2, the transition graph for the final DFA equivalent to the given regular expression is drawn as shown in Fig. 3.9(b).

Example 3.26 Construct the DFA that accepts the language represented by $0^* \cdot 1^* \cdot 2^*$.

Solution First, let us convert $0^* \cdot 1^* \cdot 2^*$ to its equivalent NFA with ϵ-moves, as shown in Fig. 3.10.

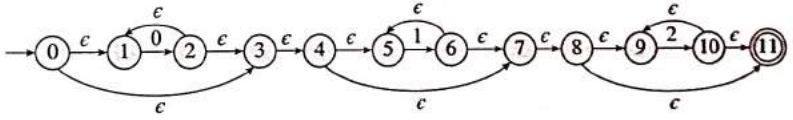


Figure 3.10 NFA with ϵ-moves for $0^* \cdot 1^* \cdot 2^*$

Table 3.3 State transition table

Q \ Σ	0	1	2
*A	B	C	D
*B	B	C	D
*C	—	C	D
*D	—	—	D

'*': Final states

This can be converted to DFA using the direct method, as shown in Fig. 3.11(a).

Overall there are four states in the equivalent DFA that we have obtained. We relabel these states using letters from A to D, as shown in Fig. 3.11(a).

The state transition table for the DFA in Fig. 3.11(a) is as shown in Table 3.3.

We observe that state B is equivalent to state A; hence it can be replaced by state A. Thus, we get the reduced table as shown in Table 3.4.

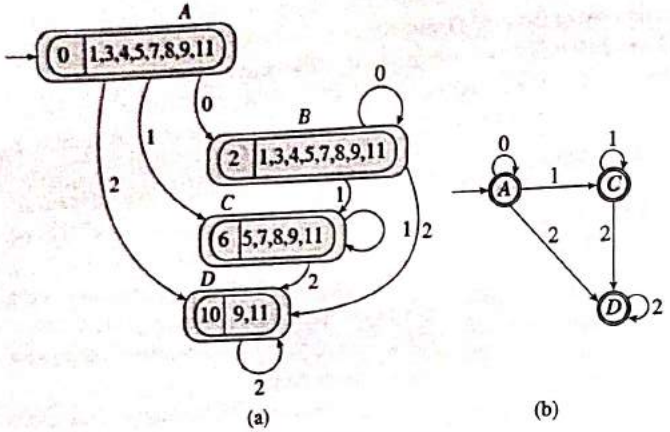


Figure 3.11 DFA construction from NFA with ϵ-moves (a) DFA conversion using direct method (b) DFA for $0^* \cdot 1^* \cdot 2^*$

Table 3.4 Reduced state transition table

Q \ Σ	0	1	2
*A	A•	C	D
*C	—	C	D
*D	—	—	D

'*': Final states
'•': Modified entry

Using the minimized state transition table for the final minimized DFA, as shown in Table 3.4, we can draw the final transition graph as shown in Fig. 3.11(b).

Example 3.27 Construct a DFA that accepts the language represented by:

$$r = (ab \,/\, ba)^* \, aa \, (ab \,/\, ba)^*$$

Solution Note that the operator '+' that is used to denote parallel paths can also be represented as '/'; though '+' is a more formal notation. Thus, the given regular expression is the same as:

$$r = (ab + ba)^* \, aa \, (ab + ba)^*$$

We obtain the NFA with ϵ-moves from the given regular expression as shown in Fig. 3.12.

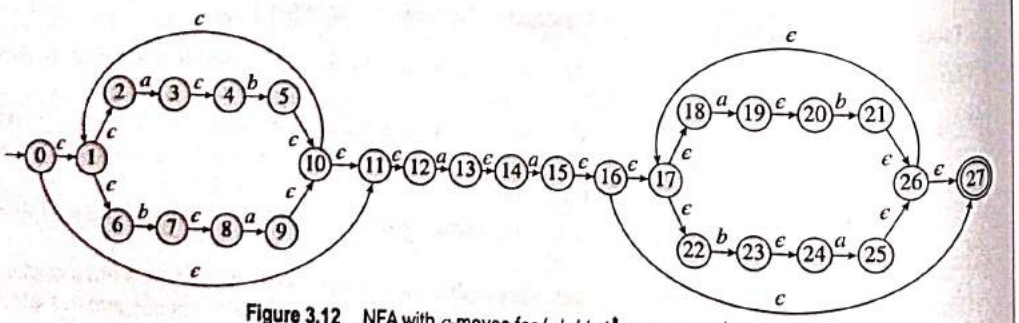


Figure 3.12 NFA with ϵ-moves for $(ab \,/\, ba)^* \, aa \, (ab \,/\, ba)^*$

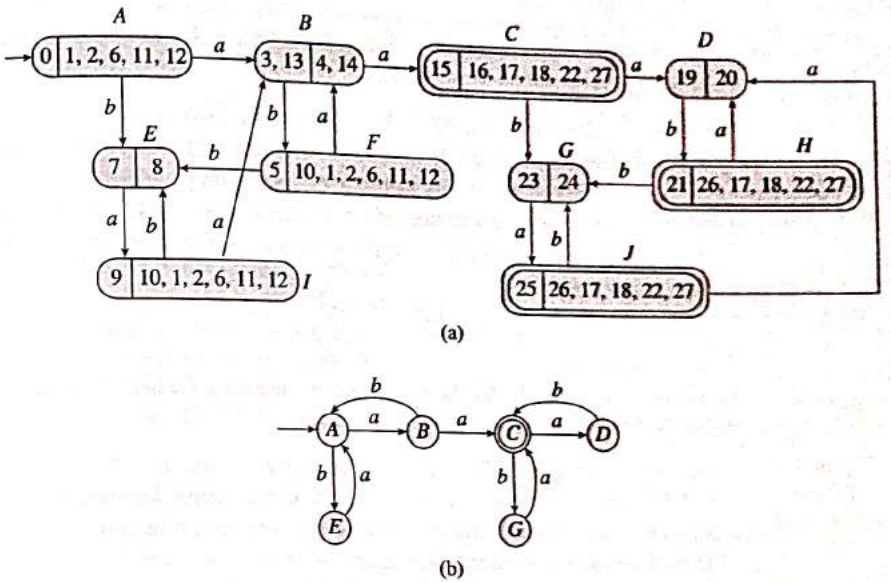


Figure 3.13 Constructing DFA from NFA with ϵ-moves (a) DFA obtained directly from Fig. 3.12 (b) Final DFA for $(ab \,/\, ba)^* \, aa \, (ab \,/\, ba)^*$

Table 3.5 State transition table for the DFA in Fig. 3.13(a)

Q \ Σ	a	b
A	B	E
B	C	F
*C	D	G
D	—	H
E	I	—
F	B	E
G	J	—
*H	D	G
I	B	E
*J	D	G

'*': Final states

Table 3.6 Reduced state transition table

Q \ Σ	a	b
A	B	E
B	C	A•
*C	D	G
D	—	C•
E	A•	—
G	C•	—

'*': Final states
'•': Modified entries

Using the direct method of conversion, the equivalent DFA is obtained as shown in Fig. 3.13(a). The state transition table for the DFA is shown in Table 3.5.

From the table, we identify the equivalent states. Hence, we can replace states *H* and *J* with state *C*, and states *I* and *F* with state *A*.

The reduced table is shown in Table 3.6.

Using Table 3.6, the final DFA that is equivalent to the given regular expression can be drawn as shown in Fig. 3.13(b).

3.4.3 DFA to Regular Expression Conversion

In the previous section, we have seen how to construct a DFA accepting the language denoted by a given regular expression. In this section, we shall prove the equivalence in the reverse direction. We start with a DFA and obtain the regular expression that denotes the language accepted by the DFA.

Let us first see some trivial examples, where we can write the regular expression by simply observing the DFA transitions. We shall discuss the conversion algorithms later in this section.

Example 3.28 Refer to the DFA in Fig. 3.14. Obtain the regular expression that denotes the language accepted by the given DFA.

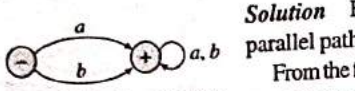


Figure 3.14 Example DFA

Solution From the initial state of the DFA to its final state, there are two parallel paths labelled by symbols a and b. This can be represented as $(a + b)$.

From the final state, there is a self-loop on the parallel paths a or b. This means that we can have any number of occurrences of a or b, which is represented by $(a + b)^*$.

Therefore, the required regular expression representing the language accepted by the DFA in Fig. 3.14 is:

$$r = (a + b) \cdot (a + b)^*$$

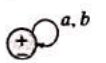


Figure 3.15 DFA for Example 3.29

Example 3.29 Find the regular expression representing the language accepted by the DFA in Fig. 3.15.

Solution In the given DFA, there is only one state, which is the initial as well as final state. There is a self-loop from the state on parallel paths labelled by symbols a and b to the same state. Hence, it represents zero or more occurrences of a or b, i.e., $(a + b)^*$.

Therefore, the required regular expression is:

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

Example 3.30 Find the regular expression equivalent to the DFA in Fig. 3.16.

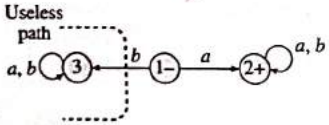


Figure 3.16 DFA for Example 3.30

Solution State 1 is the initial state from which, there is a transition on input a to state 2, which is a final state. The other transition is on input b to state 3, which is a non-final state. As we can see, state 3 is a dead (or trap) state. Therefore, that path is useless. We should remove the trap state and all the transitions that are incident on that trap state. Thus, the DFA without state 3 represents the regular expression:

$$r = a \cdot (a + b)^*$$

Example 3.31 Find the regular expression equivalent to the DFA in Fig. 3.17.

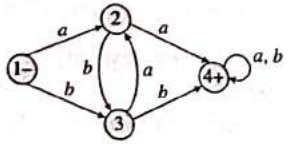


Figure 3.17 DFA for Example 3.31

Solution Let us consider the path for the input symbol a from the initial state 1. We have two different regular expressions to reach the final state 4 from 1 via 2 (i.e., the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 4$), which are as follows:

$$r_1 = a\,(b\,a)^* \cdot a \cdot (a + b)^*$$

and

$$r_2 = a \cdot (b\,a)^* \cdot b\,b \cdot (a + b)^*$$

Similarly, let us consider the path for the input symbol b from the initial state 1. We have the following two regular expressions to reach to final state via state 3 (i.e., path $1 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 4$):

$$r_3 = b \cdot (a\,b)^* \cdot b \cdot (a + b)^*$$

and

$$r_4 = b \cdot (a\,b)^* \cdot a\,a \cdot (a + b)^*$$

Therefore, the required regular expression is obtained by 'OR-ing' all the aforementioned regular expressions:

$$\begin{aligned} r &= r_1 + r_2 + r_3 + r_4 \\ &= [a \cdot (b\,a)^* \cdot a + a \cdot (b\,a)^* \cdot b\,b + b \cdot (a\,b)^* \cdot b + b \cdot (a\,b)^* \cdot a\,a] \cdot (a + b)^* \end{aligned}$$

We note that this example is slightly non-trivial for obtaining the required regular expression. Essentially, one needs to take into account all possible paths from the initial state to all final states (remember, there can be more than one final state).

Example 3.32 Find the regular expression denoting the language accepted by the DFA in Fig. 3.18.

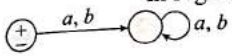


Figure 3.18 DFA for Example 3.32

Solution As the initial state is the only final state in Fig. 3.18, the other non-final state will never be reached in any string that is accepted by the machine. Hence, the non-final state here is a trap (or dead) state.

Thus, the only string accepted by this DFA is ϵ. Therefore, the required regular expression is:

$$r = \epsilon$$

Example 3.33 For the DFA in Fig. 3.19, find the equivalent regular expression.

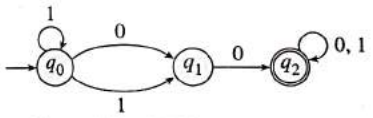


Figure 3.19 DFA for Example 3.33

Solution The first part of the regular expression is obtained by traversing the graph path, $q_0 \rightarrow q_1 \rightarrow q_2$:

$$r_1 = 1^*\,0\,0\,(0 + 1)^*$$

The second part is obtained from, $q_0 \rightarrow q_1 \rightarrow q_0 \rightarrow q_1 \rightarrow q_2$:

$$r_2 = (1^*\,01)^*\,00\,(0 + 1)^*$$

Hence, the required regular expression is:

$$\begin{aligned} r &= r_1 + r_2 \\ &= 1^*\,0\,0\,(0 + 1)^* + (1^*\,01)^*\,00\,(0 + 1)^* \end{aligned}$$

Example 3.34 For the DFA in Fig. 3.20, find the equivalent regular expression.

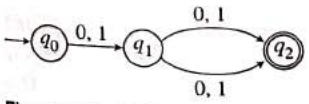


Figure 3.20 DFA for Example 3.34

Solution This DFA accepts all strings over $\Sigma = \{0, 1\}$, of even (but non-zero) length.

The regular expression is:

$$r = (0 + 1)\,(0 + 1)\,[(0 + 1)\,(0 + 1)]^*$$

Example 3.35 For the DFA in Fig. 3.21, find equivalent regular expression.

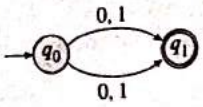


Figure 3.21 DFA for Example 3.35

Solution If we list all the strings for the DFA, we get:

$$L = \{0, 1, 000, 001, 010, 011, 100, 101, 110, 111, 00000, \ldots\}$$

We observe that the language consists of only odd length strings over $\Sigma = \{0, 1\}$.

Hence, the required regular expression is:

$$r = (0 + 1) \cdot [(0 + 1) \cdot (0 + 1)]^*$$

Example 3.36 For the DFA in Fig. 3.22, find the equivalent regular expression.

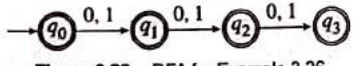


Figure 3.22 DFA for Example 3.36

Solution The language accepted by the DFA is:

$$L = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111\}$$

Hence, the regular expression is:

$$r = (0 + 1) + (0 + 1) \cdot (0 + 1) + (0 + 1) \cdot (0 + 1) \cdot (0 + 1)$$
$$= (0 + 1) + (0 + 1)^2 + (0 + 1)^3$$

Example 3.37 Find the regular expression that represents the language that is accepted by the DFA in Fig. 3.23.

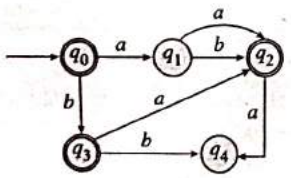


Figure 3.23 DFA for Example 3.37

Solution The language accepted by the DFA is:

$$L = \{c, ab, aa, ba\}$$

We observe that state q_4 is a dead state.

Therefore, the regular expression is:

$$r = c + aa + ab + ba$$

Here, c is included as the initial state is also a final state.

Iterative Method for DFA to Regular Expression Conversion

In the previous section, we have studied many examples, where we represented a language accepted by given DFAs using different regular expressions. Let us now discuss an algorithm, which obtains such regular expression for any given DFA as input.

Theorem 3.1

If $L = L(M)$ for some DFA M, then there is a regular expression r such that $L(R) = L(M)$.

Proof

Let us consider any general DFA M, with $Q = \{1, 2, 3, 4, \ldots, n\}$. Let us also use the label $R_{ij}^{(k)}$, which represents a regular expression, and whose language is the set of all strings w such that there is a path w available from state i to state j in the transition graph for M. The only restriction here is that the path does not traverse through any state, whose number is

greater than k. Note here that i and j may be greater than k, as they are not intermediate states, but the end points of the path.

Let us build the expression $R_{ij}^{(k)}$ through inductive definition, where we start with $k = 0$ and incrementally build the expression till $k = n$. In this way, we achieve all possible paths from i to j that traverse through all the possible states available in M.

As we have considered the state numbers to begin with 1, for $k = 0$ there will be no intermediate state. Hence, for $k = 0$, we rely only on the direct transitions that are available.

Now, there can only be two possibilities about i and j in such a case: Either $i = j$ or $i \neq j$, that is, there can be zero or more direct transitions available from i to j on some input symbol. Let us consider each case:

Case 1

If $i \neq j$, $k = 0$, and if:

1. There is no path from i to j; then, $R_{ij}^0 = \phi$.
2. There is a single symbol a, which takes the DFA M from i to j with single transition; then, $R_{ij}^0 = a$.
3. There are multiple symbols, say l in number, on which M makes transitions from i to j then, $R_{ij}^0 = a_1 + a_2 + \ldots + a_l$.

Case 2

If $i = j$, $k = 0$, and if:

1. There is no symbol which takes M from i to j (i.e., from i to i); then, $R_{ij}^0 = \epsilon$. This is always true for any state i. Hence, there is always a path ϵ from i to i.
2. There is a single symbol a such that $\delta(i, a) = i$; then, $R_{ij}^0 = (\epsilon + a)$.
3. There are multiple symbols $a_1, a_2, \ldots, a_l$ such that, $\delta(i, a_1) = i$; $\delta(i, a_2) = i$; $\ldots$, $\delta(i, a_l) = i$; then, $R_{ij}^0 = \epsilon + a_1 + a_2 + \ldots a_l$

Now, let us consider a path from state i to j that goes through intermediate states labelled either as k or lower in number. When the path from i to j does not traverse through k at all, then the regular expression can be represented as: $R_{ij}^{(k-1)}$. Alternatively, when the path traverses through k, it can be broken into three segments:

1. The first from i to k, without passing through k, that is, $R_{ik}^{(k-1)}$.
2. There could be multiple sub-paths going from k to k, without passing through k as an intermediate state—this is equivalent to $R_{kk}^{(k-1)*}$.
3. The third segment is from k to j, without passing through k—this is equivalent to $R_{kj}^{(k-1)}$.

Combining all the aforementioned expressions, that is, when there is a path from i to j that does not pass through k; and when it passes through k at least once, we get the expression as:

$$R_{ij}^k = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} (R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

The algorithm for obtaining the regular expression from the given DFA is thus based on the aforementioned rule, where we build the expressions in the order of increasing superscript. Observe that the regular expression R_{ij}^k only depends upon the expression with smaller superscript, that is, '$k - 1$', which in turn depends on '$k - 2$', and so on, till $k = 0$, which is nothing but the direct transition.

Example 3.46 Design a Mealy machine for the language represented as: $(0 + 1)^* (00 + 11)$

1. Construct a DFA for the same
2. Convert this Mealy machine to a Moore machine

Solution

1. Let us construct the equivalent NFA with ϵ-moves to start with, as shown Fig. 3.32.

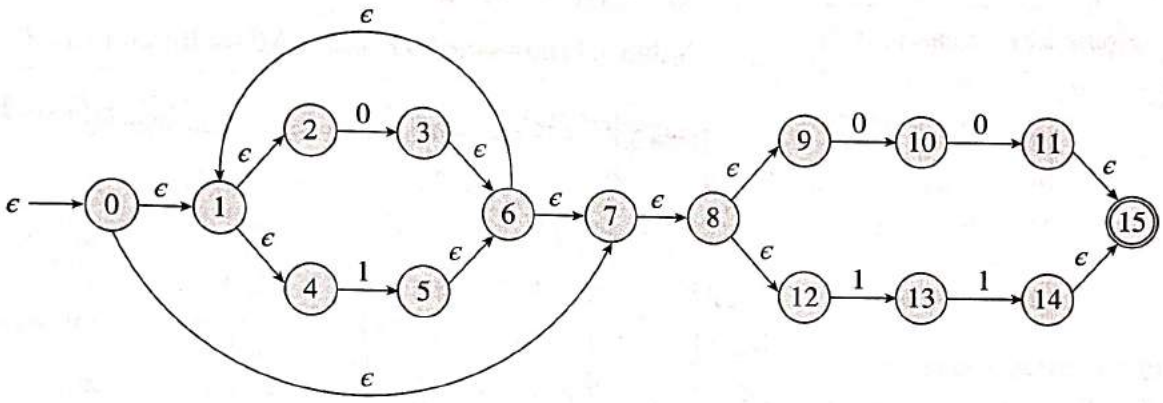
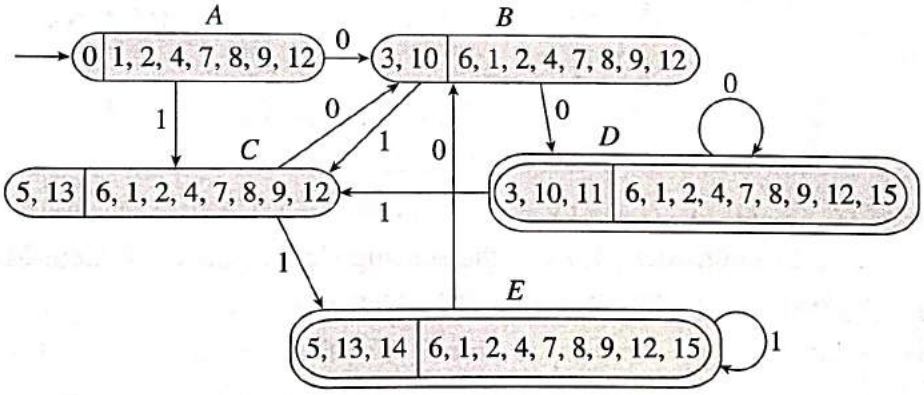
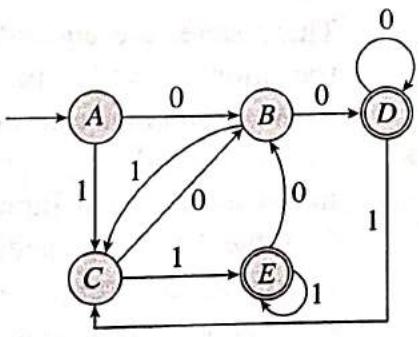


Figure 3.32 NFA with ϵ-moves for $(0 + 1)^* (00 + 11)$

We can now convert this NFA with ϵ-moves to its equivalent DFA, as shown in Fig. 3.33.



(a)



Q' \ Σ	0	1
$\rightarrow A$	B	C
B	D	C
C	B	E
(D)	D	C
(E)	B	E

(b)

Figure 3.33 NFA with ϵ-moves to DFA conversion (a) Step 1 (b) Step 2

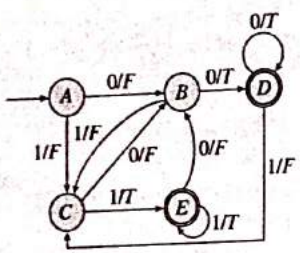


Figure 3.34 Mealy machine

The DFA in Fig. 3.33(b) can be seen as a Mealy machine, where any edge approaching the final state will carry output T (true); while the others will carry output F (false). The Mealy machine thus can be drawn as shown in Fig. 3.34.

2. Let us now construct a Moore machine equivalent to the Mealy machine in Fig. 3.34. We need to consider a new set of states:

$Q' = \{[Q \times \Delta] \mid Q \text{ is original set of states and is output alphabet}\}$

Table 3.7 shows the STF and MAF for the equivalent Moore Machine.

Table 3.7 STF and MAF for equivalent Moore machine

Σ / Q'	STF - δ'		MAF - λ'	
	0	1	Q'	Δ
$[A, F]$	$[B, F]$	$[C, F]$	$[A, F]$	F
$[A, T]$	$[B, F]$	$[C, F]$	$[A, T]$	T
$[B, F]$	$[D, T]$	$[C, F]$	$[B, F]$	F
$[B, T]$	$[D, T]$	$[C, F]$	$[B, T]$	T
$[C, F]$	$[B, F]$	$[E, T]$	$[C, F]$	F
$[C, T]$	$[B, F]$	$[E, T]$	$[C, T]$	T
$[D, F]$	$[D, T]$	$[C, F]$	$[D, F]$	F
$[D, T]$	$[D, T]$	$[C, F]$	$[D, T]$	T
$[E, F]$	$[B, F]$	$[E, T]$	$[E, F]$	F
$[E, T]$	$[B, F]$	$[E, T]$	$[E, T]$	T

Let us consider $[A, F]$ as the starting state of the equivalent Moore machine. We observe that the equivalent states are:

$[B, F] = [D, F]$
$[B, T] = [D, T]$
$[C, F] = [E, F]$
$[C, T] = [E, T]$

Table 3.8 Minimized STF and MAF for Moore machine

Σ / Q'	STF		MAF	
	0	1	Q'	Δ
$[A, F]$	$[B, F]$	$[C, F]$	$[A, F]$	F
$[B, F]$	$[B, T]$	$[C, F]$	$[B, F]$	F
$[B, T]$	$[B, T]$	$[C, F]$	$[B, T]$	T
$[C, F]$	$[B, F]$	$[C, T]$	$[C, F]$	F
$[C, T]$	$[B, F]$	$[C, T]$	$[C, T]$	T

These states are equivalent as they have the same transitions as well as the same output.

We can minimize the tables by replacing the equivalents states and removing repetitions. The minimized tables are shown in Table 3.8.

Table 3.8 is obtained by removing $[A, T]$, which is now an unnecessary entry point, since $[A, F]$ is decided to be the initial state. Further, all transitions which were depicting the removed states are altered by using their equivalent states.

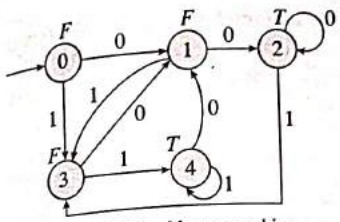


Figure 3.35 Moore machine

Now, let us rename the state labels as follows:

$[A, F] = 0$
$[B, F] = 1$
$[B, T] = 2$
$[C, F] = 3$
$[C, T] = 4$

After renaming the state labels, the modified Moore machine can be drawn as shown in Fig. 3.35.

Example 3.47 Construct the regular expressions for the following:

1. The set of all strings of 0's and 1's, such that the tenth symbol from the right is 1.
2. The set of strings in $(0 + 1)^*$, such that some two 0's are separated by a string, whose length is $4i$, for some $i \geq 0$.

Solution

1. It is required that the 10th symbol from the right should be 1. This means that the last nine symbols in the string, as well as those preceding the 10th symbol '1', are either 0 or 1. Therefore, the required regular expression is:

$$r = (0 + 1)^* \cdot 1 \cdot (0 + 1)^9$$

2. It is required that some two 0's are separated by a string of length $4i$, $i \geq 0$. There can be any prefix or postfix string for these two 0's.
Therefore, we may write the regular expression as:

$$r = (0 + 1)^* \cdot 0 \cdot [(0 + 1)^4{}^*] \cdot 0 \cdot (0 + 1)^*$$

Example 3.48 Construct a DFA for the language over {0, 1} having all strings such that the third symbol from the right end is 0.

Solution We require that the third symbol from the right end should be 0. The two symbols that follow it can be either 0 or 1. Similarly, all other symbols are also either 0 or 1.

Thus, the RE can be written as:

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

The equivalent NFA with ϵ-transitions can be drawn as shown in Fig. 3.36.

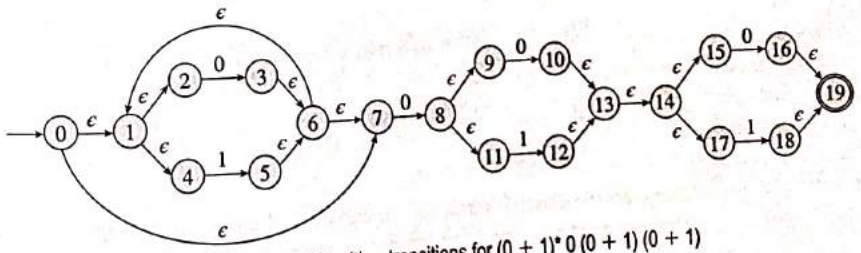


Figure 3.36 NFA with ϵ-transitions for $(0 + 1)^* 0 (0 + 1) (0 + 1)$

The equivalent DFA can be obtained as in Fig. 3.37. We can relabel the states as shown in the figure.

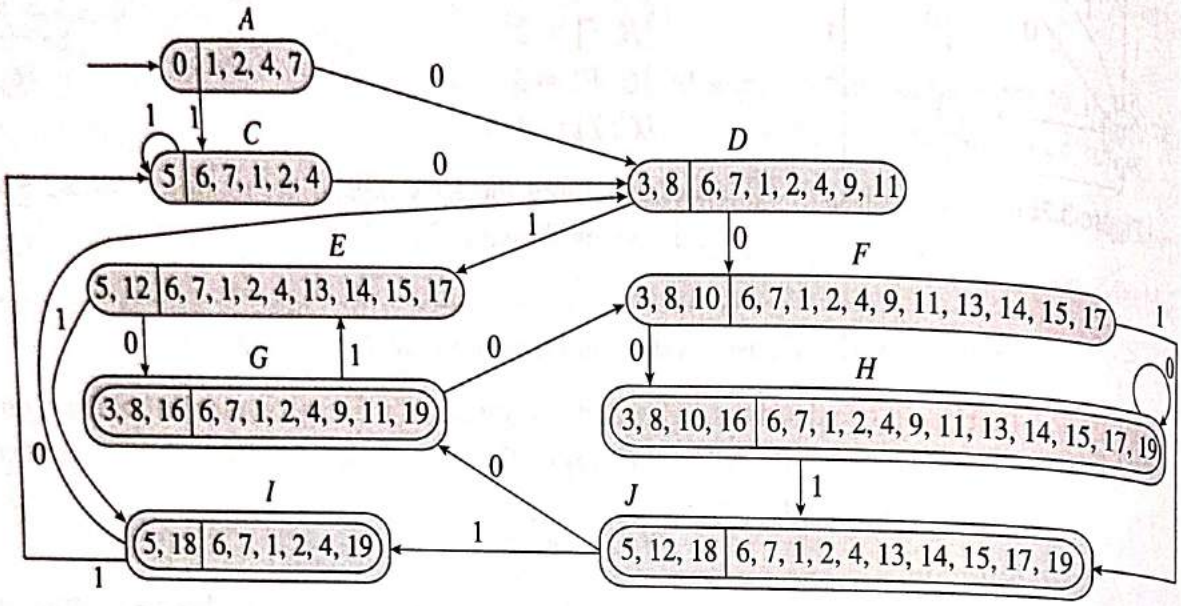


Figure 3.37 DFA equivalent to NFA with ϵ-transitions in Fig. 3.35

Let us draw the state transition table to see if we can reduce the diagram. Refer to Table 3.9.

We see that states A and C are equivalent, as both have the same transitions and both are non-final. Therefore, we can remove C and replace it by A. Refer to the reduced Table 3.10.

Table 3.9 State transition table for example DFA

Q' \ Σ	0	1
$\rightarrow A$	D	C
C	D	C
D	F	E
E	G	I
F	H	J
$*G$	F	E
$*H$	H	J
$*I$	D	C
$*J$	G	I

'*': Final states

Table 3.10 Reduced state transition table for example DFA

Q' \ Σ	0	1
$\rightarrow A$	D	A
D	F	E
E	G	I
F	H	J
$*G$	F	E
$*H$	H	J
$*I$	D	A
$*J$	G	I

'*': Final states

Using the state transition table, we can draw the transition graph of the required DFA.

3. If S is a regular set over Σ, then so its closure, that is, S*.

In other words, the class of regular sets over Σ is the smallest class containing all finite sets of words

nite automata range from system software such as language compilers, operating system utilities, and program development tools, to application programs such as text editors and syntactic pattern recognizers.

EXERCISES

This section lists a few unsolved problems to help the readers understand the topic better and practise examples on regular expressions.

Objective Questions

(E) 3.1 Let P be the language represented by the regular expression p^*q^*, and Q be $\{p^n q^n \mid n \geq 0\}$. Then which of the following is *always* regular?
(a) $P \cap Q$
(b) $P - Q$
(c) $\Sigma^* - P$
(d) $\Sigma^* - Q$

(U) 3.2 In a compiler, keywords of a language are recognized during
(a) parsing of the program
(b) code generation
(c) lexical analysis of the program
(d) dataflow analysis

(E) 3.3 Let S and T be languages over $\Sigma = \{a, b\}$ represented by the regular expressions $(a + b^*)^*$ and $(a + b)^*$, respectively. Which of the following is true?
(a) $S \neq T$
(b) $T \supseteq S$
(c) $S = T$
(d) $S \supseteq T$

(A) 3.4 Which of the following are regular languages?
$L1 = \{a^n \mid n \text{ is odd}\}$
$L2 = \{a^n \mid n \text{ is even}\}$
$L3 = \{a^n \mid n \text{ is prime}\}$
$L4 = \{a^n \mid n \text{ is a perfect square}\}$

(a) $L1$ and $L2$
(b) $L1$ only
(c) $L4$ only
(d) $L2$ and $L3$
(e) None of these
(f) All of these

(L) 3.5 Consider the regular expression $(0 + 1)(0 + 1)$ ... n times. The minimum state finite automaton that recognizes the language represented by this regular expression contains:
(a) n states
(b) $n + 1$ states
(c) $n + 2$ states
(d) None of these

(R) 3.6 A tokenizer is also called a _____.

(U) 3.7 Regular languages are closed under:
(a) Union, intersection
(b) Union, Kleene closure
(c) Intersection, complement
(d) Complement, Kleene closure

(U) 3.8 Regular languages are:
(a) Closed under union
(b) Closed under complementation
(c) Closed under intersection
(d) All of these

(L) 3.9 Which of the following languages are regular?
(i) $L = \{a^n b^n \mid n = 0, 1, 2 \ldots\}$
(ii) The set of palindromes over alphabet $\{a, b\}$
(iii) $L = \{a^n, b^m c^{2m} \mid n, m \geq 0\}$
(a) Only (i)
(b) Only (ii)
(c) (i) and (iii)
(d) All of these
(e) None of these

(L) 3.10 Which of the following regular expressions over {0, 1} denotes the set of all strings not containing '100' as a substring?
(a) $0^*(1 + 0)^*$
(b) $0^* 1010^*$
(c) $0^* 1 01$
(d) $0^*(10 + 1)^*$

(L) 3.11 Consider the following two statements:
$S1$: $\{0^{2n} \mid n \geq 1\}$ is a regular language
$S2$: $\{0^m 1^n 0^{m+n}) \mid m \geq 1$ and $n \geq 1\}$ is a regular language
Which of the following statements is true?
(a) Only $S1$
(b) Only $S2$
(c) Both $S1$ and $S2$
(d) Neither $S1$ nor $S2$

(U) 3.12 Given any regular expression, one can always construct a DFA that accepts the language denoted by it. Is this statement true or false?

(L) 3.13 Consider the following languages:
$L1 = \{ww \mid w$ belongs to $(a, b)^*\}$
$L2 = \{ww^R \mid w$ belongs to (a, b)*; and w^R is the reverse of $w\}$
$L3 = \{0^{2i} \mid i$ is an integer, $i \geq 0\}$
$L4 = \{0^{(i*i)} \mid i$ is an integer$\}$
Which of the languages are regular?
(a) $L1$ and $L2$
(b) $L2$, $L3$, and $L4$
(c) $L3$ and $L4$
(d) Only $L3$

(U) 3.14 Choose the false option:
Regular sets are closed under
(a) intersection
(b) union
(c) complement
(d) inverse substitution

(U) 3.15 Choose the false statement:
Let L be any formal language, then:
(a) L^* is regular
(b) L^* is not necessarily regular
(c) L^* is context-free and not regular
(d) L^* is recursively enumerable and not recursive

Review Questions

(R) 3.1 Define the following and give suitable examples:
(a) Regular set
(b) Regular expression

(A) 3.2 Prove that the language $L = \{a^n b^{n+1} \mid n > 0\}$ is non-regular, using pumping lemma.

(U) 3.3 Explain in brief the applications of finite automata.

(E) 3.4 Construct the NFA with ϵ-transitions, which accepts the language defined by:
$(ab + ba)^* aa (ab + ba)^*$
Convert this NFA to a minimized DFA.

(C) 3.5 Construct regular expressions defined over the alphabet $\Sigma = \{a, b\}$, which denote the following languages:
(a) All strings without a double a.
(b) All strings in which any occurrence of the symbol b, is in groups of odd numbers.
(c) All strings in which the total number of a's is divisible by 2.

(L) 3.6 Check the following regular expressions for equivalence and justify:
(a) $R_1 = (a + bb)^* (b + aa)^*$
$R_2 = (a + b)^*$
(b) $R_1 = (a + b)^* abab^*$
$R_2 = b^* a (a + b)^* ab^*$

(L) 3.7 Describe in English the sets denoted by the following regular expressions:
(a) $(a + \epsilon)(b + ba)^*$
(b) $(0^*1^*)^*$

(A) 3.8 Construct an NFA with ϵ-moves, which accepts the language defined by:
$[(0 + 1)^* 10 + (00)^* (11)^*]^*$

(U) 3.9 Let R_1 and R_2 be two regular expressions. With the help of transition diagrams, illustrate the three operations $(+, \cdot, x)$ on R_1 and R_2.

(A) 3.10 Show that the regular expressions, $(a^* bbb)^*$ a^* and $a^* (bbba^*)^*$, are equivalent.

(C) 3.11 Give a regular expression for representing all strings over $\{a, b\}$ that do not include the substrings 'bba' and 'abb'.

(L) 3.12 Consider the two regular expressions:
$R_1 = a^* + b^*$
$R_2 = ab^* + ba^* + b^* a + (a^* b)^*$
(a) Find a string corresponding to R_1 but not to R_2.
(b) Find a string corresponding to R_2 but not to R_1.
(c) Find a string corresponding to both R_1 and R_2.

(A) 3.13 Construct an NFA for the regular expression, $(a / b)^* ab$. Convert the NFA to its equivalent DFA and validate the answer with suitable examples.

(R) 3.14 Define the term regular language.

(U) 3.15 Write short note on: pumping lemma for regular sets.

(A) 3.16 Construct an NFA $(Q, \Sigma, \delta, q_0, F)$ for the following regular expression:
$01[((10)^+ + 111)^* + 0]^* 1$

(L) 3.17 Prove that the regular expressions given here are equivalent.
(a) $(a^* bbb)^* a^*$
(a) $a^* (bbb\, a^*)^*$

(L) 3.18 Describe the language accepted by the following finite automaton.

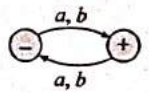


Figure 3.38 Example DFA

(L) 3.19 Describe as simply as possible in English the language represented by: $(0/1)^* 0$.

(A) 3.20 Construct an NFA that recognizes the regular expression $(a / b)^* \cdot a \cdot b$. Convert it to a DFA, and draw the state transition table.

(A) 3.21 Construct a regular expression corresponding to the state diagram shown here, using Arden's theorem.

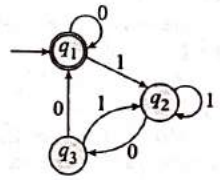


Figure 3.39 Example FA

(U) 3.22 Is the following language regular? Justify.
$L = \{0^p 1^p p^{p+q} \mid p \geq 1, q \geq 1\}$

(C) 3.23 Construct the regular expression and finite automata for: $L = L_1 \cap L_2$ over alphabet $\{a, b\}$, where:
L_1 = all strings of even length
L_2 = all strings starting with b

(L) 3.24 Which of the following are true? Explain.
(a) $baa \in a^* b^* a^* b^*$
(b) $b^* a^* \cap a^* b^* = a^* \cup b^*$
(c) $a^* b^* \cap b^* c^* = \phi$
(d) $abcd \in [a (cd)^* b]^*$

(A) 3.25 Construct the regular expressions for the following DFAs:

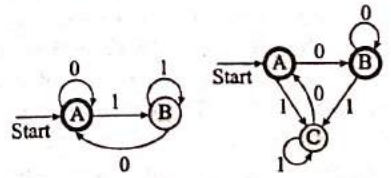


Figure 3.40 Example DFAs

(U) 3.26 Which of the following languages are regular sets? Justify your answer.

(a) $\{0^{2n} \mid n \geq 1\}$

(b) $\{0^m 1^n 0^{m+n} \mid m \geq 1 \text{ and } n \geq 1\}$

(U) 3.27 Find out whether given languages are regular or not:

(a) $L = \{ww \mid w \in \{0, 1\}^*\}$

(b) $L = \{1^k \mid k = n^2, n >= 1\}$

(U) 3.28 With the help of a suitable example, prove that 'regular sets are closed under union, concatenation, and Kleene closure'.

(U) 3.29 Explain the following applications of regular expressions:

(a) grep utility in UNIX

(b) Finding pattern in text

(C) 3.30 Construct the NFA and DFA for the following languages:

(a) $L = \{x \in \{a, b, c\}^* \mid x$ contains exactly one b immediately following $c\}$

(b) $L = \{x \in \{0, 1\}^* \mid x$ starts with 1 and $|x|$ is divisible by $3\}$

(c) $L = \{x \in \{a, b\}^* \mid x$ contains any number of a's followed by at least one $b\}$

(C) 3.31 Let $\Sigma = \{0, 1\}$. Construct regular expressions for each of the following:

(a) $L_1 = \{W = \Sigma^* \mid W$ has at least one pair of consecutive zeros$\}$

(b) $L_2 = \{W \in \Sigma^* \mid W$ has no pair of consecutive zeros$\}$

(c) $L_3 = \{W \in \Sigma^* \mid W$ starts with either '01' or '10'$\}$

(d) $L_4 = \{W \in \Sigma^* \mid W$ consists of even number of 0's followed by odd number of 1's$\}$

(A) 3.32 Construct a regular expression for the following DFA:

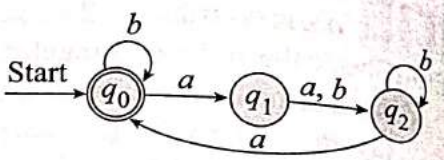


Figure 3.41 Example DFA

(A) 3.33 Let $L = \{0^n \mid n$ is a prime number$\}$; show that L is not regular.

(L) 3.34 Prove or disprove the following for regular expressions r, and s.

(a) $(rs + r)^* r = r(sr + r)^*$

(b) $s(rs + s)^* r = rr^* s(rr^* s)^*$

(c) $(r + s)^* = r^* + s^*$

(d) $(r^* s^*)^* = (r + s)^*$

(U) 3.35 State whether each of the following statements is true of false. Justify your answer. Assume that all languages are defined over the alphabet $\{0, 1\}$.

(a) If $(L1 \subseteq L2)$ and ($L1$ is not regular), then $L2$ is not regular.

(b) If $(L1 \subseteq L2)$ and ($L2$ is not regular), then $L1$ is not regular.

(c) If $L1$ and $L2$ are not regular, then $(L1 \cup L2)$ is not regular.

(A) 3.36 Use pumping lemma to check whether the language, $L = \{ww \mid w \in (0, 1)^*\}$ is regular or not.

5 Grammars

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Concept of context-free grammars (CFGs)
- Formalism of CFGs and context-free languages (CFLs)
- Leftmost/Rightmost derivations and derivation tree
- Concept of ambiguous grammar and removal of ambiguity
- Simplification of CFG
- Chomsky normal form (CNF) and Greibach normal form (GNF)
- Chomsky hierarchy
- Equivalence of regular grammars and finite automata
- Concept of derivation graph
- Applications of CFG
- Backus–Naur form (BNF)
- Kuroda normal form
- Dyck language
- Pumping lemma and Ogden's lemma

5.1 INTRODUCTION

Grammar, for a particular language, is defined as a finite set of formal rules for generating syntactically correct sentences. The language may be a formal programming language, such as C, C++, and Java, or natural language such as English and French. Any type of language requires a grammar, which defines syntactically correct statement formats or constructs allowed for that particular language. Hence, we can also call a grammar for a language as the *syntactic definition* of the language; and we can, therefore, say that *grammar defines syntax of a language.*

For example, if we want to generate the English statement 'dog runs', we may use the following rules:

$$\begin{aligned} &\langle \text{Sentence} \rangle \rightarrow \langle \text{noun} \rangle \langle \text{verb} \rangle \\ &\langle \text{Noun} \rangle \rightarrow \text{dog} \\ &\langle \text{Verb} \rangle \rightarrow \text{runs} \end{aligned}$$

These rules describe how the sentence of the form 'noun' followed by 'verb' can be generated. As we know, there are many such rules in the English language; and these are collectively called the grammar for the language.

In this chapter, we are going to discuss the grammar for formal languages, particularly context-free grammar (CFG).

5.2 CONSTITUENTS OF GRAMMAR

Grammar consists of two types of symbols—terminals and non-terminals (also called variables or auxiliary symbols).

Terminal symbols are those symbols that are part of a generated sentence. For example, in the aforementioned example, 'dog' and 'runs' are the terminal symbols as they collectively formulate the statement and are part of the generated statement.

Non-terminal symbols are those symbols that take part in the formation (or generation) of the statement, but are not part of the generated statement. No statement that is generated using grammar rules will ever contain non-terminals. For example, in the aforementioned example, 'sentence', 'noun', and 'verb' are non-terminals, which are not present in the generated statement; but they took part in the formation of the statement.

We see that:

<sentence> $\xrightarrow{\text{gives}}$ <noun> <verb>

$\xrightarrow{\text{gives}}$ dog runs

Therefore, non-terminals are essential while declaring the rules; without these, the grammar cannot be defined. These rules for grammar are also called *productions*, *production rules*, or *syntactical rules*. The word 'production' is used in relation with the statement-generation process (or derivation process).

5.3 FORMAL DEFINITION OF GRAMMAR

For building a formal model, we must consider two aspects of the given grammar:

1. The generative capacity of the grammar, that is, the grammar used should generate all and only those sentences of the language for which it is written.
2. Constituents of the grammar—terminals and non-terminals.

A grammar that is based on the constituent structure described here is called *constituent structure grammar* or *phase structure grammar.*

A phrase structure grammar is denoted by a quadruple of the form:

$$G = \{V, T, P, S\},$$

where,

V: Finite set of non-terminals (or variables); sometimes this is also denoted by N instead of V

T: Finite set of terminals

S: S is a non-terminal, which is a member of N; it usually denotes the starting symbol

P: Finite set of productions (or syntactical rules)

Productions have the generic form '$\alpha \rightarrow \beta$', where $\alpha, \beta \subset (V \cup T)^*$, and $\alpha \neq \epsilon$. As we know, $V \cap T = \phi$. Note that α and β may consist of any number of terminals as well as non-terminals and they are usually termed as *sentential forms.*

If the production rules have the form '$A \rightarrow \alpha$' where A is any non-terminal and α is any sentential form, then such grammar is called *context-free grammar (CFG).*

Observe that in this type of grammar has a restriction that on the left-hand side of each production there is only one non-terminal. For example, the grammar that we have considered for generating the statement 'dog runs' is context-free grammar (with some notational differences; these will be discussed in sections 5.4 and 5.16).

5.4 GRAMMAR NOTATIONS

There are some typical notations used for grammars. The following are some simple CFG notations:

1. Non-terminals are usually denoted by capital letters: A, B, C, D, etc.
2. Terminals are denoted by small case letters: a, b, c, etc.
3. Strings of terminals (that is, valid words in the language for which the grammar is written) are usually denoted by u, v, w, x, y, z, etc.
4. Sentential forms, that is, strings of the form $(V \cup T)^*$ that are formed using any combination of terminals and non-terminals are denoted by α, β, γ, etc.
5. The arrow symbol, '$\rightarrow$', stands for production. For example, a production denoted by '$A \rightarrow b$' indicates, '*A produces b*'.
6. If the same non-terminal has multiple productions, that is, there is more than one rule for the same non-terminal, we use '|' (which means 'or') to represent the grammar in a simplified form.

   For example, let us consider the following grammar G:

   $$G = \{(E), (+, *, a), P, E\},$$

   where, P consists of the following productions:

   $$P = \{E \rightarrow E + E,\ E \rightarrow E * E,\ E \rightarrow a\}$$

   We see that all the three productions mentioned are for the non-terminal E, that is, all the three productions have the same left-hand side. In such a case, we can combine these with the help of the '|' operator as shown here:

   $$P = \{E \rightarrow E + E \mid E * E \mid a\}$$

7. The '$\Rightarrow$' symbol stands for the process of derivation (or generation) of any string belonging to some language.

For example, let us consider the production set P that consists of the productions: $\{S \rightarrow AB; A \rightarrow a; B \rightarrow b\}$. Let us try to derive or generate the string 'ab' using this set of productions. Every grammar has a start symbol and that is the starting point for any derivation process. Hence, we begin with the start symbol S.

As S produces AB, we can derive or generate AB from S.

$$S \overset{\text{derives}}{\Rightarrow} AB$$

Since A produces a, we have:

$$S \overset{\text{derives 'a'}}{\Rightarrow} a\,B$$

Since B produces b, we have:

$$S \overset{\text{derives 'b'}}{\Rightarrow} ab$$

Here, the symbol '$\Rightarrow$' stands for 'derived in a single step'. Similarly, the symbol '$\overset{*}{\Rightarrow}$' stands for 'derived in any number of steps'. For example, we can write, $S \overset{*}{\Rightarrow} ab$, for the aforementioned generation process. We shall discuss more about the derivation process in the next section.

8. It is possible to have a production of the form '$A \rightarrow \epsilon$', which is termed as an '*ϵ-production*'.

5.5 DERIVATION PROCESS

Any string that can be derived using the grammar rules is said to be part of the language represented by the grammar. If G is the grammar, then the language generated by the grammar is represented as:

$$L(G) = \{x \mid x \text{ can be derived using productions in } G\}$$

Derivation of a string begins with the start symbol of the grammar, and we may apply multiple grammar rules to get the string of all terminals.

There are two different types of derivations—leftmost derivation and rightmost derivation.

5.5.1 Leftmost Derivation

If at each step in a derivation, a production is applied to the leftmost variable (or non-terminal), then the derivation is called leftmost derivation.

For example, if the grammar G consists of:

$$\{(E), (+, *, a), P, E\},$$

where P consists of the following productions:

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow a$

Using this grammar, let us generate the string '$a + a$'.

We begin the derivation with the start symbol, that is, E.

At every stage, the symbol to be replaced by the right-hand side of the applicable production is underlined.

The string is generated as follows:

$$\begin{aligned} E &\Rightarrow \underline{E} + E, && \text{using production 1} \\ &\Rightarrow a + \underline{E}, && \text{using production 3 applied to the leftmost non-terminal } E \\ &\Rightarrow a + a, && \text{using production 3 applied to } E, \text{ which is the only non-terminal} \end{aligned}$$

Observe that in each step of the derivation, the leftmost non-terminal symbol is replaced by the right-hand side of the production applicable.

We also observe that at each step in the derivation, we get some string containing both terminals and non-terminals, that is, sentential form. In the leftmost derivation, these are termed as *left sentential forms*.

In this example, '$E + E$', '$a + E$', and '$a + a$' are left sentential forms. As we know, the final left sentential form, '$a + a$', is the generated string, and does not contain any non-terminal.

Note: Leftmost derivation generates any string from left to right. Hence, it is mainly applicable for top-down parsing. A top-down parser is a program that uses context-free grammar to generate the input string using leftmost derivation. If the input string can be generated, it is considered as a valid input string. As the input string is read from left to right, it is expected to be generated in the same order while checking the validity, terminal by terminal (refer to Chapter 6, Section 6.7 on equivalence of CFG and PDA).

5.5.2 Rightmost Derivation

If at each step in the derivation of a string, a production is always applied to the rightmost non-terminal, then the derivation is called rightmost derivation. It is also termed as *canonical derivation*.

For example, let us consider the same grammar G, where P consists of:

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow a$

Let us derive the string '$a + a * a$' using the aforementioned grammar, applying rightmost derivation.

We begin with the start symbol E.

At every stage, the symbol to be replaced by the right-hand side of the applicable production is underlined.

$$\begin{aligned} E &\Rightarrow E + \underline{E}, && \text{using production 1} \\ &\Rightarrow E + E * \underline{E}, && \text{using production 2, applied to the rightmost } E \\ &\Rightarrow E + \underline{E} * a, && \text{using production 3, applied to the rightmost } E \\ &\Rightarrow \underline{E} + a * a, && \text{using production 3, applied to the rightmost } E \\ &\Rightarrow a + a * a, && \text{using production 3, applied to the only } E \text{ left} \end{aligned}$$

The aforementioned derivation is a rightmost derivation, as we are replacing the rightmost variable by the right-hand side of the production at every step in the derivation. The sentential forms at every step in the rightmost derivation are termed as *right sentential forms*.

For example, in the aforementioned derivation, '$E + E$', '$E + E * E$', '$E + E * a$', '$E + a * a$', and '$a + a * a$' are all right sentential forms. The last right sentential form, that is, '$a + a * a$' is the string that we have derived.

5.5.3 Derivation Examples

Let us now solve some problems illustrating leftmost and rightmost derivations.

Example 5.1 Consider the following CFG:

$$G = \{(S, A), (a, b), P, S\},$$

where P consists of:

$$S \rightarrow aAS \mid a$$
$$A \rightarrow SbA \mid SS \mid ba$$

Derive string '*aabbaa*' using leftmost derivation and rightmost derivation.

Solution From the given information, S is the start symbol. Let us number the productions as:

1. $S \rightarrow aAS$
2. $S \rightarrow a$
3. $A \rightarrow SbA$
4. $A \rightarrow SS$
5. $A \rightarrow ba$

The leftmost derivation for the string '*aabbaa*' can be shown as given here:

Leftmost derivation:

$S \Rightarrow a\underline{A}S$, using rule 1
$\Rightarrow aS\underline{b}AS$, using rule 3
$\Rightarrow aab\underline{A}S$, using rule 2
$\Rightarrow aabba\underline{S}$, using rule 5
$\Rightarrow aabbaa$, using rule 2

The rightmost derivation for the same string is as shown here:

Rightmost derivation:

$S \Rightarrow aA\underline{S}$, using rule 1
$\Rightarrow a\underline{A}a$, using rule 2
$\Rightarrow aSb\underline{A}a$, using rule 3
$\Rightarrow a\underline{S}bbaa$, using rule 5
$\Rightarrow aabbaa$, using rule 2

Example 5.2 For the grammar G, which is defined as:

$$S \rightarrow aB \mid bA,$$
$$A \rightarrow a \mid aS \mid bAA, \text{ and}$$
$$B \rightarrow b \mid bS \mid aBB,$$

where S is the starting symbol, write the leftmost and rightmost derivations for the string '*bbaaba*'.

Solution Let us number the rules as follows

1. $S \rightarrow aB$
2. $S \rightarrow bA$
3. $A \rightarrow a$
4. $A \rightarrow aS$
5. $A \rightarrow bAA$
6. $B \rightarrow b$
7. $B \rightarrow bS$
8. $B \rightarrow aBB$

Leftmost derivation:

$S \Rightarrow b\underline{A}$, using rule 2
$\Rightarrow bb\underline{A}A$, using rule 5
$\Rightarrow bba\underline{S}A$, using rule 4
$\Rightarrow bbaa\underline{B}A$, using rule 1
$\Rightarrow bbaab\underline{A}$, using rule 6
$\Rightarrow bbaaba$, using rule 3

Rightmost derivation:

$S \Rightarrow b\underline{A}$, using rule 2
$\Rightarrow bbA\underline{A}$, using rule 5
$\Rightarrow bb\underline{A}a$, using rule 3
$\Rightarrow bba\underline{S}a$, using rule 4
$\Rightarrow bbaa\underline{B}a$, using rule 1
$\Rightarrow bbaaba$, using rule 6

Example 5.3 For the following grammar, give the leftmost and rightmost derivations for the string '*aaabbb*'.

$$S \rightarrow aSb \mid ab$$

Solution Let us number the productions as follows:

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

Leftmost derivation:

$S \Rightarrow a\underline{S}b$, using rule 1
$\Rightarrow aa\underline{S}bb$, using rule 1
$\Rightarrow aaabbb$, using rule 2

Rightmost derivation:

Rightmost derivation is exactly same as leftmost derivation, as in each step there is only one non-terminal to be replaced by its right-hand side.

5.6 DERIVATION TREE

Derivation tree is a graphical representation, or description, of how the sentence (or string) has been derived, or generated, using a grammar. For every string derivable from the start symbol, there is an associated derivation tree. Derivation tree is also known as *rule tree*, *parse tree*, or *syntax tree*.

Thus, the generation of any string or statement can be represented with the help of a leftmost derivation, rightmost derivation, or derivation tree. All these three representations are equivalent, which means that given any one of them, the other two can be constructed.

Formal Definition

Let $G = \{V, T, P, S\}$ be a CFG. A tree is a derivation tree for G if:

1. Every vertex (or node) has a label, which is a symbol from the set $\{V \cup T \cup (\epsilon)\}$.
2. The label of the root vertex is S, the start symbol of G.
3. If a vertex is interior (or an intermediate node) in the tree and has label A, then A must be in V, that is, A is a non-terminal.
4. If vertex n has label A and vertices $n_1, n_2, \ldots, n_k$ are the children nodes of vertex n, in order from the left, with labels $X_1, X_2, \ldots, X_k$ respectively, then '$A \rightarrow X_1 X_2 \ldots X_k$' must be a production in P.
5. If vertex n has label ϵ, then n is a leaf and is the only son of its father, (i.e., ϵ-production).

Example 5.4 Consider the grammar, $G = \{(S, A), (a, b), P, S\}$,

where P consists of:

$$S \rightarrow a\,A\,S \mid a$$
$$A \rightarrow S\,b\,A \mid S\,S \mid b\,a$$

Draw derivation tree for the string '*aabbaa*'.

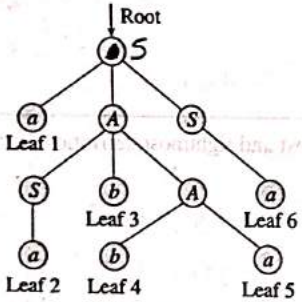


Figure 5.1 Derivation tree for string '*aabbaa*'

Solution In Section 5.5.3, we have seen both leftmost as well the rightmost derivations for the aforementioned string. The third way to represent the derivation of the string is with the help of a derivation tree, as shown in Fig. 5.1.

If we read the leaf nodes from left to right in sequence, it gives us the string '*aabbaa*'. The tree gives us a pictorial representation of the derivation steps for generating the string '*aabbaa*'.

5.7 CONTEXT-FREE LANGUAGES

A context-free language (CFL) is the language generated by a context-free grammar G. This can be described as:

$$L(G) = \{w \mid w \subset T^*, \text{ and is derivable from the start symbol } S\}$$

All the CFGs that we have seen so far represent CFLs.

5.7.1 Examples

Let us discuss some examples of CFLs, and attempt to write their CFGs.

Example 5.5 Let G be a context-free grammar, which is defined as:

$$S \rightarrow a\,S\,b \mid a\,b$$

Find the CFL generated by G.

Solution Let us begin by listing the strings that we can generate using the given CFG, G. We start with minimal length string. Let us number the productions as:

1. $S \rightarrow a\,S\,b$
2. $S \rightarrow a\,b$

Let us now derive different strings using G.

Using production 2, we can derive string '*ab*' in one step as:

$$S \Rightarrow a\,b$$

Let us derive another string as follows:

$S \Rightarrow a\,\underline{S}\,b$, using rule 1
$\Rightarrow a\,a\,b\,b$, using rule 2

Similarly, we have:

$S \Rightarrow a\,\underline{S}\,b$, using rule 1
$\Rightarrow a\,a\,\underline{S}\,b\,b$, using rule 1
$\Rightarrow a\,a\,a\,b\,b\,b$, using rule 2
$S \Rightarrow a\,\underline{S}\,b$, using rule 1
$\Rightarrow a\,a\,\underline{S}\,b\,b$, using rule 1
$\Rightarrow a\,a\,a\,\underline{S}\,b\,b\,b$, using rule 1
$\Rightarrow a\,a\,a\,a\,b\,b\,b\,b$, using rule 2

Thus, the language can be listed in the form of a set, as:

$$L(G) = \{ab, aabb, aaabbb, aaaabbbb, \ldots\},$$

or

$$L(G) = \{a^n b^n \mid n \geq 1\}$$

Thus, the CFG G defines the language containing strings of the form $a^n b^n$ for $n \geq 1$; which can be expressed as, 'n number of a's followed by the same number of b's'.

Example 5.6 Find the CFL associated with the CFG, G, which is defined as follows:

$$S \rightarrow a\,B \mid b\,A$$
$$A \rightarrow a \mid a\,S \mid b\,A\,A$$
$$B \rightarrow b \mid b\,S \mid a\,B\,B$$

Solution Let us number the productions (or rules) as follows:

1. $S \rightarrow a\,B$
2. $S \rightarrow b\,A$
3. $A \rightarrow a$
4. $A \rightarrow a\,S$

5. $A \rightarrow b\,A\,A$
6. $B \rightarrow b$
7. $B \rightarrow b\,S$
8. $B \rightarrow a\,B\,B$

Let us derive different strings using the given CFG, G:

$S \Rightarrow a\,\underline{B}$, using rule 1
$\Rightarrow a\,b$, using rule 6

$S \Rightarrow b\,\underline{A}$, using rule 2
$\Rightarrow b\,a$, using rule 3

$S \Rightarrow a\,\underline{B}$ using rule 1
$\Rightarrow a\,b\,\underline{S}$, using rule 7
$\Rightarrow a\,b\,a\,\underline{B}$, using rule 1
$\Rightarrow a\,b\,a\,b$, using rule 6

$S \Rightarrow a\,\underline{B}$, using rule 1
$\Rightarrow a\,b\,\underline{S}$, using rule 7
$\Rightarrow a\,b\,b\,\underline{A}$, using rule 2
$\Rightarrow a\,b\,b\,a$, using rule 3

$S \Rightarrow a\,\underline{B}$, using rule 1
$\Rightarrow a\,a\,\underline{B}\,B$, using rule 8
$\Rightarrow a\,a\,b\,\underline{B}$, using rule 6
$\Rightarrow a\,a\,b\,b$, using rule 6

$S \Rightarrow b\,\underline{A}$, using rule 2
$\Rightarrow b\,a\,\underline{S}$, using rule 4
$\Rightarrow b\,a\,a\,\underline{B}$, using rule 1
$\Rightarrow b\,a\,a\,b$, using rule 6

$S \Rightarrow b\,\underline{A}$, using rule 2
$\Rightarrow b\,b\,\underline{A}\,A$, using rule 5
$\Rightarrow b\,b\,a\,\underline{A}$, using rule 3
$\Rightarrow b\,b\,a\,a$, using rule 3

$S \Rightarrow b\,\underline{A}$, using rule 2
$\Rightarrow b\,b\,\underline{A}\,A$, using rule 5
$\Rightarrow b\,b\,a\,\underline{S}\,A$, using rule 4
$\Rightarrow b\,b\,a\,a\,\underline{B}\,A$, using rule 1
$\Rightarrow b\,b\,a\,a\,b\,\underline{A}$, using rule 6
$\Rightarrow b\,b\,a\,a\,b\,a$, using rule 3

$S \Rightarrow b\,\underline{A}$, using rule 2
$\Rightarrow b\,a\,\underline{S}$, using rule 4
$\Rightarrow b\,a\,b\,\underline{A}$, using rule 2
$\Rightarrow b\,a\,b\,b\,\underline{A}\,A$, using rule 5
$\Rightarrow b\,a\,b\,b\,a\,\underline{S}\,A$, using rule 4
$\Rightarrow b\,a\,b\,b\,a\,a\,\underline{B}\,A$, using rule 1
$\Rightarrow b\,a\,b\,b\,a\,a\,b\,\underline{A}$, using rule 6
$\Rightarrow b\,a\,b\,b\,a\,a\,b\,a$, using rule 3

Thus, we see that all the strings that we have derived have equal number of a's and b's. Thus, the CFL generated by the given CFG, G is:

$$L(G) = \{x \mid x \text{ containing equal number of } a\text{'s and } b\text{'s}\}.$$

Example 5.7 Write the grammar for generating strings over $\Sigma = \{a\}$, containing any (zero or more) number of a's, and comment upon the language generated by this grammar.

Solution Zero number of a's can be generated using the production:

$S \rightarrow \epsilon$ (i.e., ϵ-production)

If we want one or more a's, we can generate them using the productions:

$$S \rightarrow a \mid a\,S$$

Combining the two, the grammar that we get is:

$$S \rightarrow a\,S \mid a \mid \epsilon$$

We see that the production '$S \rightarrow a$' is unnecessary, as that can be generated by applying the rule '$S \rightarrow aS$', followed by substituting for S using '$S \rightarrow \epsilon$'.

Thus, we can represent the grammar formally as:

$$G = \{(S), (a, \epsilon), (S \rightarrow aS, S \rightarrow \epsilon), S\}$$

Let us try deriving the string '*aaa*'. For this grammar, using leftmost derivation:

$S \Rightarrow a\,\underline{S}$, using rule 1
$\Rightarrow a\,a\,\underline{S}$, using rule 1
$\Rightarrow a\,a\,a\,\underline{S}$, using rule 1
$\Rightarrow a\,a\,a$, using rule 2, this substitutes ϵ for S

Note: The language described in this example is a regular language, and we can denote it using the regular expression, a^*. Hence, the grammar, G, and the regular expression, a^*, are equivalent as they both represent the same language L, containing any number (zero or more) of a's.

Example 5.8 Write a grammar for the language represented by the regular expression, $(a + b)^*$.

Solution The regular expression $(a + b)^*$ represents a regular language containing any number of a's or b's.

The grammar is given by:

$$\underset{1}{S \rightarrow a\,S} \mid \underset{2}{b\,S} \mid \underset{3}{\epsilon}$$

Let us try deriving the string '*aaba*' for this grammar:

$S \Rightarrow a\,\underline{S}$, using rule 1
$\Rightarrow a\,a\,\underline{S}$, using rule 1
$\Rightarrow a\,a\,b\,\underline{S}$, using rule 2

$\Rightarrow a\,a\,b\,a\,\underline{S}$, using rule 1
$\Rightarrow a\,a\,b\,a$, using rule 3

Note: If we want to write the CFG for the language $(a + b)^+$, that is, strings containing at least one occurrence of a or b, it can be written as:

$S \rightarrow a\,S \mid b\,S \mid a \mid b$

This grammar now cannot generate an empty string, that is, the string containing zero number of a's and zero number of b's, because P does not contain a production of the form, $S \rightarrow \epsilon$.

Example 5.9 Write the grammar generating all strings consisting of a's and b's with at least two a's.

Solution The given language can be represented using the following regular expression:

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

The grammar generating this language can be written as:

$$G = \{(S, A), (a, b, \epsilon), P, S\},$$

where, P consists of:

$$\begin{aligned} S &\rightarrow A\,a\,A\,a\,A \\ A &\rightarrow a\,A \mid b\,A \mid \epsilon \end{aligned} \qquad (5.1)$$

(2 3 4)

We observe that A generates the language represented by $(a + b)^*$.

Let us generate string 'aa' for grammar G, using rightmost derivation:

$S \Rightarrow A\,a\,A\,a\,\underline{A}$, using rule 1
$\Rightarrow A\,a\,\underline{A}\,a$, using rule 4
$\Rightarrow \underline{A}\,a\,a$, using rule 4
$\Rightarrow a\,a$, using rule 4

Thus, 'aa' is the minimal length string from the given language.

Example 5.10 Write the grammar for generating the variable names, which can be given by the regular expression: (*letter*) (*letter* / *digit*)*, which means, a letter followed by any number of letters or digits.

Solution Let non-terminal L denote and produce letters, and let non-terminal D produce digits.

The grammar G can be written as:

$S \rightarrow LS'$
$S' \rightarrow LS' \mid DS' \mid \epsilon$
$L \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \ldots \mid 9$

Here, S is the start symbol.

TAR123

Let us derive the variable name '*flag12*', using the aforementioned grammar.

The steps in the (leftmost) derivation are:

$S \Rightarrow \underline{L}S'$ $S \rightarrow LS'$
$\Rightarrow f\,\underline{S}'$ $S \rightarrow T\,LS'$
$\Rightarrow f\,\underline{L}\,S'$ $S \rightarrow TAS'$
$\Rightarrow f\,l\,\underline{S}'$ $S \rightarrow TALS'$
$\Rightarrow f\,l\,\underline{L}\,S'$ $S \rightarrow TARS'$
$\Rightarrow f\,l\,a\,\underline{S}'$ $S \rightarrow TARDS'$
$\Rightarrow f\,l\,a\,\underline{L}\,S'$ $S \rightarrow TARLS'$
$\Rightarrow f\,l\,a\,g\,\underline{S}'$
$\Rightarrow f\,l\,a\,g\,\underline{D}\,S'$
$\Rightarrow f\,l\,a\,g\,1\,S'$
$\Rightarrow f\,l\,a\,g\,1\,\underline{D}\,S'$
$\Rightarrow f\,l\,a\,g\,1\,2\,\underline{S}'$
$\Rightarrow f\,l\,a\,g\,1\,2$ (using $S' \rightarrow \epsilon$)

$S \rightarrow XY$
$X \rightarrow aX \mid bX \mid a$
$Y \rightarrow Ya \mid Yb \mid a$

$S \rightarrow \underline{X}Y$
$S \rightarrow aXY$
$S \rightarrow aaY$
$S \rightarrow aaYB$
$S \rightarrow aaaa$

Example 5.11 Describe the CFL generated by the following grammar G:

$$G = \{(S), (a, b, \epsilon), P, S\},$$

where, P consists of:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon$$
(1 2 3 4 5)

Solution Let us try to derive different strings from the CFL, so that we can list them in the form of a set:

$S \Rightarrow a\,\underline{S}\,a$, using rule 1
$\Rightarrow a\,a$, using rule 5
$S \Rightarrow b\,\underline{S}\,b$, using rule 2
$\Rightarrow b\,b$, using rule 5
$S \Rightarrow a\,\underline{S}\,a$, using rule 1
$\Rightarrow a\,a\,a$, using rule 3
$S \Rightarrow a\,\underline{S}\,a$, using rule 1
$\Rightarrow a\,b\,a$, using rule 4
$S \Rightarrow b\,\underline{S}\,b$, using rule 2
$\Rightarrow b\,a\,b$, using rule 3
$S \Rightarrow b\,\underline{S}\,b$, using rule 2
$\Rightarrow b\,b\,b$, using rule 4

$S \rightarrow aSa$
$S \rightarrow abSba$
$S \rightarrow ababa$
$S \rightarrow a$

The set of all strings will thus be:

$$L = \{\epsilon, a, b, aa, bb, aaa, aba, bbb, bab, \ldots\}$$

Thus, the CFL is a language consisting of all palindrome strings over $\Sigma = \{a, b\}$.

Example 5.12 Write a grammar for the language over $\Sigma = \{a, b\}$ containing at least one occurrence of '*aa*'.

Solution We are given a regular language, which can be represented using $(a + b)^* a a (a + b)^*$. The grammar can be written as follows:

$$S \rightarrow X a a X$$
$$X \rightarrow a X \mid b X \mid \epsilon$$

Here, S is the start symbol.

Let us derive the string, '*bbaaab*' using leftmost derivation, as follows:

$$\begin{aligned}
S &\Rightarrow \underline{X} a a X, && \text{using } (S \rightarrow X a a X)\\
&\Rightarrow b \underline{X} a a X, && \text{using } (X \rightarrow b X)\\
&\Rightarrow b b \underline{X} a a X, && \text{using } (X \rightarrow b X)\\
&\Rightarrow b b a a \underline{X}, && \text{using } (X \rightarrow \epsilon)\\
&\Rightarrow b b a a a \underline{X}, && \text{using } (X \rightarrow a X)\\
&\Rightarrow b b a a a b X, && \text{using } (X \rightarrow b X)\\
&\Rightarrow b b a a a b, && \text{using } (X \rightarrow \epsilon)
\end{aligned}$$

Example 5.13 Find the CFL generated by following grammar:

$$S \rightarrow X Y$$
$$X \rightarrow a X \mid b X \mid a$$
$$Y \rightarrow Y a \mid Y b \mid a$$

Solution If we consider only the productions associated with X, which are:

$$X \rightarrow a X \mid b X \mid a$$

we see that X produces strings containing any number of a's or b's, ending with a.

If we consider the productions associated with Y, which are:

$$Y \rightarrow Y a \mid Y b \mid a$$

we see that Y produces strings starting with a followed by any number of a's or b's.

Now considering the production '$S \rightarrow X Y$', we can conclude that S produces strings with at least one occurrence of '*aa*'. The first a comes from strings ending with a that are derivable from X, and the second a comes from the strings beginning with a that are derivable from Y.

Therefore, this grammar is equivalent to the CFG that we have seen in previous problem—both the CFGs produce the same language consisting of strings over $\Sigma = \{a, b\}$ with at least two consecutive a's.

Example 5.14 Write the grammar to generate the strings containing consecutive a's but not consecutive b's.

Solution The required grammar will contain the following productions:

$$S \rightarrow a S \mid b X \mid a \mid b \mid \epsilon$$
$$X \rightarrow a S \mid a \mid \epsilon$$

with, $V = \{S, X\}$, $T = \{a, b\}$, and S as the start symbol.

We observe from the aforementioned productions that as soon as b is generated, the grammar tries to generate a as the next symbol. This is done using the production '$S \rightarrow b X$'.

Example 5.15 Consider the CFG:

$$S \rightarrow X Y X$$
$$X \rightarrow a X \mid b X \mid \epsilon$$
$$Y \rightarrow b b b$$

Show that this generates the language defined by $(a + b)^* bbb (a + b)^*$.

Solution Let us derive a few strings so that we can list them in the form of a set:

$$\begin{aligned}
S &\Rightarrow \underline{X} Y X, && \text{using } (S \rightarrow X Y X)\\
&\Rightarrow \underline{Y} X, && \text{using } (X \rightarrow \epsilon)\\
&\Rightarrow b b b \underline{X}, && \text{using } (Y \rightarrow b b b)\\
&\Rightarrow b b b, && \text{using } (X \rightarrow \epsilon)\\
S &\Rightarrow \underline{X} Y X, && \text{using } (S \rightarrow X Y X)\\
&\Rightarrow a \underline{X} Y X, && \text{using } (X \rightarrow a X)\\
&\Rightarrow a \underline{Y} X, && \text{using } (X \rightarrow \epsilon)\\
&\Rightarrow a b b b \underline{X}, && \text{using } (Y \rightarrow b b b)\\
&\Rightarrow a b b b, && \text{using } (X \rightarrow \epsilon)
\end{aligned}$$

$$L = \{bbb, abbb, bbbb, bbba, \ldots\}$$

We also see that since $X \rightarrow a X \mid b X \mid \epsilon$, the non-terminal X produces all strings containing any number of a's and b's, that is, the language can be represented as $(a + b)^*$.

Hence, the given CFG generates the language defined by $(a + b)^* bbb (a + b)^*$.

Example 5.16 Write a CFG to generate the language of all strings that have more number of a's than b's.

Solution The expected language set can be written as:

$$L = \{a, aa, aab, aba, baa, aaaa, aaaab, \ldots\}$$

Earlier, in Example 5.6, we have discussed the grammar for a language containing equal number of a's and b's.

The grammar was defined as:

$$S \rightarrow aB \mid bA$$
$$A \rightarrow a \mid aS \mid bAA$$
$$B \rightarrow b \mid bS \mid aBB$$

Using the aforementioned grammar, the grammar for the required language L can be written as:

$$X \to YS \mid SY \mid YSY \mid Y$$
$$S \to aB \mid bA$$
$$A \to a \mid aS \mid bAA$$
$$B \to b \mid bS \mid aBB$$
$$Y \to aY \mid a$$

Here, X is the start symbol.

Example 5.17 Find a CFG for each of the languages defined by the following regular expressions:

(a) ab^*
(b) a^*b^*
(c) $(baa + abb)^*$

Solution

(a) The required CFG for ab^* is given by:

$$S \to aB$$
$$B \to bB \mid \epsilon$$

Here, S is the start symbol.

(b) The required CFG for a^*b^* is:

$$S \to AB$$
$$A \to aA \mid \epsilon$$
$$B \to bB \mid \epsilon$$

Here, S is the start symbol.

(c) The required CFG for $(baa + abb)^*$ is:

$$S \to AS \mid BS \mid \epsilon$$
$$A \to baa$$
$$B \to abb$$

Here, S is the start symbol.

Example 5.18 Find a CFG for the language over $\Sigma = \{a, b\}$ containing words that have different first and last letters.

Solution It is required that the words should have different first and last letters. This means that if the first letter is a, then the last letter in the word should be b. Similarly, if the first letter is b, then the last letter in the word should be a. In between, we can have any combination of a's and b's.

The regular language can be represented using the regular expression:

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

The required CFG, therefore, can be written as:

$$S \to aAb \mid bAa$$
$$A \to aA \mid bA \mid \epsilon$$

Here, S is the starting symbol.

Example 5.19 Find the context-free grammar that generates the language:

$$L = \{a^i b^j c^k \mid i = j + k\}$$

Solution The grammar that generates the given language L can be written as follows:

$$S \to aSc \mid A \mid \epsilon$$
$$A \to aAb \mid \epsilon$$

Here, S is the starting symbol; and the production '$S \to \epsilon$' is added to include the case when the values of i, j, and k are all zero.

The non-terminal S, with the help of the production '$S \to aSc$', generates as many c's as the number of a's. The middle A in the production '$A \to aAb$' generates equal number of a's and b's; and '$S \to A$' is added to include the case when the value of k is zero.

Example 5.20 Write a CFG for the following language:

$$L = \{a^{m+n} b^m c^n \mid n, m \geq 0\}$$

Solution We observe that this language is exactly the same as the language in the previous example, that is, Example 5.19. Hence, the CFG is defined as:

$$S \to aSc \mid A \mid \epsilon$$
$$A \to aAb \mid \epsilon$$

Example 5.21 Write the CFG for the language:

$$L = \{0^m 1^n 0^{m+n} \mid m, n \geq 0\}$$

Solution The grammar for the language is defined as:

$$S \to 0S0 \mid A \mid \epsilon$$
$$A \to 1S0 \mid \epsilon$$

Here, S is the starting symbol. The rule '$S \to 0S0$' adds equal (i.e., m) number of 0's at the beginning and towards the end. The rule '$A \to 1S0$' adds equal number (i.e., n) of 0's and 1's, in order to fulfil the requirement for '$m + n$' number of 0's towards the end.

Example 5.22 Find the CFG that generates each of the following languages:

(a) Set of odd length strings in $\{a, b\}^*$ having a as the middle symbol
(b) Set of even length strings in $\{a, b\}^*$ having the same symbols in the middle

Solution

(a) The required grammar can be written as follows:

$$S \rightarrow a\,S\,a \mid a\,S\,b \mid b\,S\,a \mid b\,S\,b \mid a$$

(b) The required grammar can be written as follows:

$$S \rightarrow a\,S\,a \mid a\,S\,b \mid b\,S\,a \mid b\,S\,b \mid a\,a \mid b\,b$$

Example 5.23 Give the CFG for the following language:

$$L = \{a^n\,b^m\,a^n \mid n \geq 0, m \geq 1\}$$

Solution The grammar generating the required language L can be written as follows:

$$S \rightarrow a\,S\,a \mid B$$
$$B \rightarrow b\,B \mid b$$

Here, S is the starting symbol. After generating the starting and ending a's, the S in the middle can be replaced by B using the rule '$S \rightarrow B$'. The non-terminal B generates one or more b's in the middle as required.

Example 5.24 Construct the CFG for the language:

$$L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^m b^n c^n d^m \mid n, m \geq 1\}$$

Solution The first part $\{a^n b^n c^m d^m \mid n, m \geq 1\}$ can be expressed with the help of the following grammar, G_1:

$$S_1 \rightarrow A\,C$$
$$A \rightarrow a\,A\,b \mid a\,b$$
$$C \rightarrow c\,C\,d \mid c\,d$$

The second part $\{a^m b^n c^n d^m \mid n, m \geq 1\}$ can be expressed with the help of the following grammar, G_2:

$$S_2 \rightarrow a\,S_2\,d \mid a\,B\,d$$
$$B \rightarrow b\,B\,c \mid b\,c$$

As we want to obtain the union of these two languages, the equivalent grammar, G can be obtained as follows:

$$G = G_1 \cup G_2$$

Thus, the final grammar, G, can be written as follows:

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow A\,C$$
$$A \rightarrow a\,A\,b \mid a\,b$$
$$C \rightarrow c\,C\,d \mid c\,d$$
$$S_2 \rightarrow a\,S_2\,d \mid a\,B\,d$$
$$B \rightarrow b\,B\,c \mid b\,c$$

Example 5.25 Construct the CFG for the language containing the set of strings over $\{a, b\}$ with exactly twice as many a's as b's.

Solution Let us consider the case when the string has one b; then, there there will be two a's in the string. The following are the string combinations with one b and two a's:

$$\{aab, aba, baa\}$$

These combinations can be pumped (or repeated) to achieve all possible combinations of such strings. Hence, we can use the following regular expression to represent the set of strings:

$$(aab + aba + baa)^*$$

This is a regular language, and we can write a CFG for this language as follows:

$$S \rightarrow A\,S \mid \epsilon$$
$$A \rightarrow a\,a\,b \mid a\,b\,a \mid b\,a\,a$$

Example 5.26 Construct a CFG for the following language set:

$$L = \{a^{2n} b^n \mid n \geq 1\}$$

Solution The grammar for the required language can be written as follows:

$$S \rightarrow a\,a\,S\,b \mid a\,a\,b$$

For every two a's in the beginning, one b is generated at the end.

Example 5.27 Find the context-free grammar for the following language:

$$L = \{a^n\,b^m\,c^k \mid n = m \text{ or } m \leq k; n \geq 0, m \geq 0, k \geq 0\}$$

Solution There are two alternatives discussed here:

(a) $n = m$; therefore, k can be any number, and is not related to either n or m.
(b) $m \leq k$; therefore, n can be any number, and is not related to m or k.

For alternative (a), the grammar can be written as:

$$S_1 \rightarrow A\,C$$
$$A \rightarrow a\,A\,b \mid \epsilon; \quad n = m$$
$$C \rightarrow c\,C \mid \epsilon; \text{ generates any number of } c\text{'s}$$

For alternative (b), the grammar can be written as:

$$S_2 \rightarrow B\,D\,E$$
$$B \rightarrow a\,B \mid \epsilon; \text{ generates any number of } a\text{'s}$$
$$D \rightarrow b\,D\,c \mid \epsilon; \text{ for } m = k$$
$$E \rightarrow c\,E \mid \epsilon; \text{ for } m < k$$

We now combine these two grammars by taking their union as follows:

$$S \rightarrow S_1 \mid S_2$$

The rest of the productions remain as they are.

Example 5.28 Construct a CFG for the following set:

$\{a^{2n} b c \mid n \geq 1\}$

Solution The grammar G for the given set can be written as follows:

$S \rightarrow A\, b\, c$
$A \rightarrow a\, a\, A \mid a\, a$; for $n \geq 1$

We see that the number of a's generated is always even.

5.8 AMBIGUOUS CONTEXT-FREE GRAMMAR

A CFG for a language is said to be *ambiguous*, if there exists at least one string, which can be generated (or derived) in more than one way. This means that there exists more than one leftmost derivation, more than one rightmost derivation, and more than one derivation tree associated with such a string.

For an *unambiguous* CFG, there exists exactly one way of deriving any string of the language. This means that there is exactly one way to depict the leftmost derivation, one way to describe the rightmost derivation, and exactly one derivation tree that can be associated with such a string. The examples we have seen in Section 5.7.1 are all unambiguous grammars.

Let us now look at an example of ambiguous grammar. Consider the following grammar:

$$E \underset{1}{\rightarrow} E + E \mid E * E \underset{2}{} \mid \underset{3}{id}$$

Let us try deriving the string '$id + id \times id$':

Using leftmost derivation, there are two different ways of deriving the given string:

(a) $E \Rightarrow \underline{E} + E$, using rule 1
$\Rightarrow id + \underline{E}$, using rule 3
$\Rightarrow id + \underline{E} * E$, using rule 2
$\Rightarrow id + id * \underline{E}$, using rule 3
$\Rightarrow id + id * id$, using rule 3

(b) $E \Rightarrow \underline{E} \times E$, using rule 2
$\Rightarrow \underline{E} + E * E$, using rule 1
$\Rightarrow id + \underline{E} * E$, using rule 3
$\Rightarrow id + id * \underline{E}$, using rule 3
$\Rightarrow id + id * id$, using rule 3

Similarly, there are two ways of deriving the same string using rightmost derivation:

(a) $E \Rightarrow E + \underline{E}$, using rule 1
$\Rightarrow E + E * \underline{E}$, using rule 2
$\Rightarrow E + \underline{E} * id$, using rule 3
$\Rightarrow \underline{E} + id * id$, using rule 3
$\Rightarrow id + id * id$, using rule 3

(b) $E \Rightarrow E * \underline{E}$, using rule 2
$\Rightarrow \underline{E} * id$, using rule 3
$\Rightarrow E + \underline{E} * id$, using rule 1
$\Rightarrow \underline{E} + id * id$, using rule 3
$\Rightarrow id + id * id$, using rule 3

Likewise, we can draw two different derivation trees to depict the derivation for the same string, as shown in Fig. 5.2.

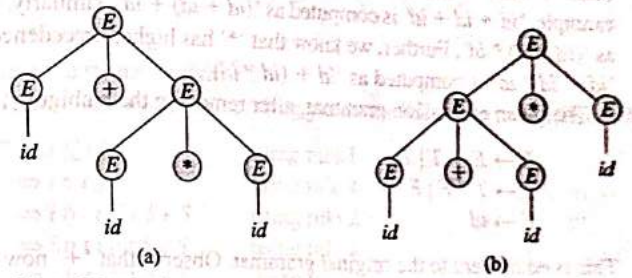


Figure 5.2 Derivation trees for the string '$id + id * id$' (a) First derivation tree for '$id + id * id$' (b) Second derivation tree for '$id + id * id$'

Thus, there are two different ways of deriving the string '$id + id * id$', based on the production that is used first. Hence, the grammar given is ambiguous.

As there are more than one ways of deriving a string, one might wonder which one is the right derivation, or if both are correct. The first derivation, that is, to start with the production '$E \rightarrow E + E$', is considered to be correct as '$id + id * id = id + (id * id)$' because '*' has higher priority than '+'. Therefore, the basic operation is '+', that is, addition of 'id' and '$id * id$'. Hence, the derivation tree in Fig. 5.2(a) is correct. The grammar described here is ambiguous as the information of the operators' precedence is missing.

The reader can try deriving the string '$id + id + id$', which also has more than one ways of deriving, using the aforementioned grammar. Though the same operator '+' is used twice in the string, the information that '+' is left associative is missing from the grammar. Though '$(id + id) + id$' is the right way to group the operands, the grammar also allows '$id + (id + id)$'; that introduces an ambiguity.

5.8.1 Removal of Ambiguity

There is no algorithm for removing ambiguity in any given CFG. Every ambiguous grammar has a different cause for the ambiguity, and hence, the remedy for each would be different. Removing the ambiguity for a specific grammar is feasible; but creating a generic solution to remove the ambiguity of any ambiguous grammar is an *unsolvable problem*.

While we remove the ambiguity of any grammar, we must ensure that the resultant grammar is equivalent to the original one; in other words, it should generate the same CFL as the original grammar generates.

For example, let us consider the grammar we have seen earlier:

$$E \to E + E \mid E * E \mid id$$

Here, the cause of ambiguity is that the grammar does not use the precedence information and associative properties of the operators '+' (addition) and '*' (multiplication). Since both '+' and '*' are left associative, the order of the operations is from left to right. For example, '$id + id + id$' is computed as '$(id + id) + id$'; similarly, '$id * id * id$' is computed as '$(id * id) * id$'. Further, we know that '*' has higher precedence over '+'; for example, '$id + id * id$' is computed as '$id + (id * id)$'.

The given expression grammar, after removing the ambiguity, can be written as:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to id$$

This is equivalent to the original grammar. Observe that '+' now can be derived only using non-terminal E; thus the left operands are automatically grouped now. We further see that only non-terminal T can derive a string containing '×'; hence, when '+' and '×' are used together, '*' is given higher priority.

Let us derive the same string '$id + id * id$', using the new unambiguous grammar.

Leftmost derivation of '$id + id * id$' can be depicted as:

$$\begin{aligned} E &\Rightarrow \underline{E} + T \\ &\Rightarrow \underline{T} + T \\ &\Rightarrow \underline{F} + T \\ &\Rightarrow id + T \\ &\Rightarrow id + \underline{T} * F \\ &\Rightarrow id + \underline{F} * F \\ &\Rightarrow id + id * \underline{F} \\ &\Rightarrow id + id * id \end{aligned}$$

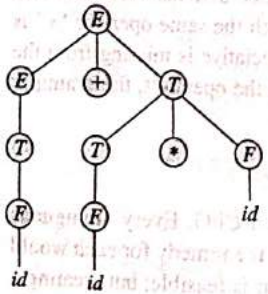


Figure 5.3 Derivation tree for '$id + id * id$' using unambiguous grammar

Rightmost derivation can be written as:

$$\begin{aligned} E &\Rightarrow E + \underline{T} \\ &\Rightarrow E + T * \underline{F} \\ &\Rightarrow E + \underline{T} * id \\ &\Rightarrow E + \underline{F} * id \\ &\Rightarrow \underline{E} + id * id \\ &\Rightarrow \underline{T} + id * id \\ &\Rightarrow \underline{F} + id * id \\ &\Rightarrow id + id * id \end{aligned}$$

The derivation tree for the string is shown in Fig. 5.3. Observe that there is only one way of deriving the required string.

Example 5.29 Check whether or not the following grammar is ambiguous; if it is ambiguous, remove the ambiguity and write an equivalent unambiguous grammar.

$$S \to iCtS \mid iCtSeS \mid a$$
$$C \to b$$

Solution Let us number the productions as follows:

$$\underset{1}{S \to iCtS} \mid \underset{2}{iCtSeS} \mid \underset{3}{a}$$
$$\underset{4}{C \to b}$$

Let us derive the string '*ibtibtaea*'.

The leftmost derivation for the string yields the following two different derivations:

$$\begin{aligned} S &\Rightarrow i\underline{C}tS, && \text{using rule 1} \\ &\Rightarrow ibt\underline{S}, && \text{using rule 4} \\ &\Rightarrow ibti\underline{C}tSeS, && \text{using rule 2} \\ &\Rightarrow ibtibt\underline{S}eS, && \text{using rule 4} \\ &\Rightarrow ibtibtae\underline{S}, && \text{using rule 3} \\ &\Rightarrow ibtibtaea, && \text{using rule 3} \\ S &\Rightarrow i\underline{C}tSeS, && \text{using rule 2} \\ &\Rightarrow ibt\underline{S}eS, && \text{using rule 4} \\ &\Rightarrow ibti\underline{C}tSeS, && \text{using rule 2} \\ &\Rightarrow ibtibt\underline{S}eS, && \text{using rule 4} \\ &\Rightarrow ibtibtae\underline{S}, && \text{using rule 3} \\ &\Rightarrow ibtibtaea, && \text{using rule 3} \end{aligned}$$

There are also two different rightmost derivations as depicted here:

$$\begin{aligned} S &\Rightarrow iCt\underline{S}, && \text{using rule 1} \\ &\Rightarrow iCtiCtSe\underline{S}, && \text{using rule 2} \\ &\Rightarrow iCtiCt\underline{S}ea, && \text{using rule 3} \\ &\Rightarrow iCti\underline{C}taea, && \text{using rule 3} \\ &\Rightarrow i\underline{C}tibtaea, && \text{using rule 4} \\ &\Rightarrow ibtibtaea, && \text{using rule 4} \\ S &\Rightarrow iCtSe\underline{S}, && \text{using rule 2} \\ &\Rightarrow iCt\underline{S}ea, && \text{using rule 3} \\ &\Rightarrow iCtiCt\underline{S}ea, && \text{using rule 1} \\ &\Rightarrow iCti\underline{C}taea, && \text{using rule 3} \\ &\Rightarrow i\underline{C}tibtaea, && \text{using rule 4} \\ &\Rightarrow ibtibtaea, && \text{using rule 4} \end{aligned}$$

We have seen that there can be two leftmost derivations instead of one. Similarly, there are two rightmost derivations instead of one. This means that there are two different ways of deriving the same string: one that starts with '$iCtS$' and the other starts with '$iCtSeS$'.

These two ways can be reflected in the two derivation trees shown in Fig. 5.4.

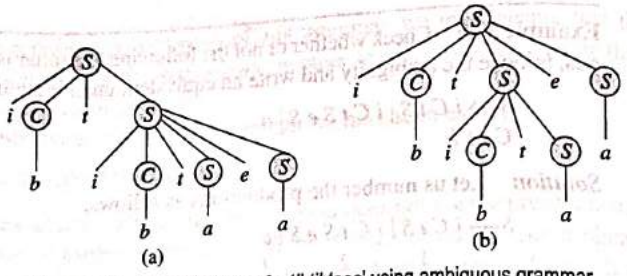


Figure 5.4 Derivation trees for '*ibtibtaea*' using ambiguous grammar
(a) First derivation tree (b) Second derivation tree

Hence, we conclude that the given grammar is ambiguous.

The given grammar is the grammar for the 'if-statement'. Here, the 'else' part is considered associated with the later 'if' part. This information is not used by the grammar writer. Hence, there is ambiguity.

The equivalent unambiguous grammar is:

$$S \to A \mid B$$
$$A \to iCtAeA \mid a$$
$$B \to iCtS \mid iCtAeB$$
$$C \to b$$

Using this unambiguous grammar, we now attempt to derive the string '*ibtibtaea*'. We observe that in this case, there is only one way of deriving the string. Therefore, the grammar is now unambiguous.

Leftmost derivation:

$$S \Rightarrow \underline{B}$$
$$\Rightarrow iC\underline{t}S$$
$$\Rightarrow ibt\underline{S}$$
$$\Rightarrow ibt\underline{A}$$
$$\Rightarrow ibtiC\underline{t}AeA$$
$$\Rightarrow ibtibt\underline{A}eA$$
$$\Rightarrow ibtibtaeA$$
$$\Rightarrow ibtibtaea$$

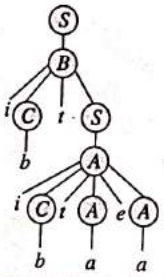


Figure 5.5 Derivation tree for '*ibtibtaea*' using unambiguous grammar

Rightmost derivation:

$$S \Rightarrow \underline{B}$$
$$\Rightarrow iCt\underline{S}$$
$$\Rightarrow iCt\underline{A}$$
$$\Rightarrow iCtiCtAe\underline{A}$$
$$\Rightarrow iCtiCt\underline{A}ea$$
$$\Rightarrow iCti\underline{C}taea$$
$$\Rightarrow iCtibtaea$$
$$\Rightarrow ibtibtaea$$

Similarly, the derivation tree for the string is shown in Fig. 5.5.

5.9 SIMPLIFICATION OF CONTEXT-FREE GRAMMAR

A context-free grammar G can be simplified, and written in a *reduced form*, using the following rules:

1. Each variable and terminal of the CFG should appear in the derivation of at least one word in the $L(G)$.
2. There should not be any production of the form '$A \to B$', where A and B are both non-terminals.

We shall discuss some simplification techniques in the following subsections.

5.9.1 Removal of Useless Symbols

A symbol X is considered useful if there is a derivation of the form, $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, where α, β are sentential forms and w is any string of terminals, that is, $(w \subset T^*)$.

If no such derivation exists, then the symbol X will not appear in any of the derivations; in this case, X is a *useless symbol*.

Three aspects of usefulness of a non-terminal X are as follows:

1. There must be some string that can be derived from X
2. The symbol X must appear in the derivation of at least one string derivable from S (start symbol).
3. The symbol X should not occur in any sentential form that contains a variable from which no terminal string can be derived.

In order to remove the useless symbols, we must either delete the production rules that contain these symbols or rewrite the grammar. Let us now discuss a few example grammars.

Example 5.30 Consider the following grammar:

$$G = \{(S, A), (1, 0), P, S\},$$

where, P consists of the following productions:

$$S \to 10 \mid 0S1 \mid 1S0 \mid A \mid SS$$

Simplify the grammar by removing the useless symbols, if any.

Solution We observe that there is a non-terminal A, which cannot derive any string of terminals, in the given grammar G. This is represented as follows:

$$S \Rightarrow A \Rightarrow ?$$

Hence, A is a useless symbol and we should remove it.

To remove the useless symbol A, we should delete the production '$S \to A$'.

Therefore, the simplified or reduced form grammar is:

$$S \to 10 \mid 0S1 \mid 1S0 \mid SS$$

Note: The grammar, after removing the productions containing useless symbols, should be checked for the purpose it was written for. One should rewrite the grammar if the resultant language is changed in the process.

Example 5.31 Consider the grammar G defined as:

$$G = \{(S, A, B), (a), P, S\},$$

where, P consists of:

$$S \to A\,B \mid a$$
$$A \to a$$

Simplify the grammar by removing useless symbols, if any.

Solution We observe that in the given grammar, there is a variable B, which does not derive any string of terminals. Thus, it is a useless symbol.

In order to simplify the grammar, we should delete all the productions that contain B. There is one such production '$S \to AB$'; hence, we drop this production and write the simplified grammar without useless symbols as:

$$S \to a$$
$$A \to a$$

Here, though A produces a string of terminals, it cannot occur in any derivation of any string derivable from S. Therefore, A is also a useless symbol now and hence, we should drop the production '$A \to a$'.

Hence, the simplified grammar can be written with only one production rule:

$$G = \{(S), (a), (S \to a), S\}$$

This grammar is obviously equivalent to the given grammar, but is in a simplified form.

5.9.2 Removal of Unit Productions

A production rule of the form '$A \to B$', where, A and B are both non-terminals, is called a *unit production*. All other productions (including ϵ-productions) are non-unit productions.

Elimination Rule

For every pair of non-terminals A and B,

1. If the CFG has a unit production of the form '$A \to B$', or
2. If there is a chain of unit productions leading from A to B, such as:

$$A \Rightarrow X_1 \Rightarrow X_2 \Rightarrow \ldots \Rightarrow B,$$

where, all $X_i\,(i > 0)$ are non-terminals, then introduce new production(s) according to the following rule:

'If the non-unit productions for B are:

$$B \to \alpha_1 \mid \alpha_2 \mid \ldots,$$

where, $\alpha_1, \alpha_2, \ldots$, are sentential forms (not containing only one non-terminal); then create the productions for A as:

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots'$$

Example 5.32 Consider the grammar G defined as:

$$G = \{(A, B), (a, b), P, A\},$$

where, P consists of:

$$A \to B$$
$$B \to a \mid b$$

Simplify the grammar by eliminating the unit productions, if any.

Solution We see that '$A \to B$' is the unit production that we need to eliminate.

Applying the elimination rule, we have:

$$B \to \alpha_1 \mid \alpha_2,$$

where $\alpha_1 = a$,
and $\alpha_2 = b$.
Therefore, we can write the reduced form grammar without the unit production as:

$$A \to a \mid b$$

Thus, the simplified G can be written as:

$$G = \{(A), (a, b), (A \to a, A \to b), A\}$$

Example 5.33 Consider the grammar G, which consists of the following productions with S as the start symbol:

$$S \to A \mid b\,b$$
$$A \to B \mid b$$
$$B \to S \mid a$$

Simplify the grammar by eliminating the unit productions, if any.

Solution We see that there is a chain of unit productions: $S \to A \to B \to S$.
Using the productions, '$A \to B$' and '$B \to S \mid a$', we add a new production to A, as follows:

$$A \to S \mid a$$

Therefore, the grammar becomes:

$$S \to A \mid b\,b$$
$$A \to S \mid a \mid b$$

We have another unit production, '$S \to A$'; after removing this, the grammar is reduced to:

$$S \to S \mid a \mid b \mid b\,b$$

However, there is still one more unit production, namely, '$S \to S$', which we can directly remove as the symbol on the left and right are the same. Therefore, the equivalent grammar without unit productions can be written as:

$$S \to a \mid b \mid b\,b$$

Example 5.34 Eliminate unit productions from the following grammar:

$$S \to a \mid X\,b \mid a\,Y\,a \mid b \mid a\,a$$
$$X \to Y$$
$$Y \to b \mid X$$

Solution In the aforementioned grammar, we see that there are two unit productions: '$X \to Y$' and '$Y \to X$'.

From the productions, '$X \to Y$' and '$Y \to b \mid X$', we can derive:

$$Y \to b \mid Y$$

Hence, we can now delete '$X \to Y$'.

Further, we observe that the new rule '$Y \to Y$' does not make any sense, and hence, can be deleted.

After these additions and deletions, we rewrite the given grammar as follows:

$$S \to a \mid Y\,b \mid a\,Y\,a \mid b \mid a\,a, \quad \text{(replacing } X \text{ by } Y\text{)}$$
$$Y \to b$$

5.9.3 Elimination of ϵ-Productions

A production of the form '$A \to \epsilon$', where A is a non-terminal, is known as an *ϵ-production*. If ϵ is a member of $L(G)$ for a given grammar G, then we cannot eliminate all ϵ-productions from G; whereas, if ϵ is not in $L(G)$, then we can remove all the ϵ-productions.

Theorem 5.1

If L is a CFL generated by a CFG that includes ϵ-productions, then there exists another CFG without ϵ-productions, which generates either the whole language L (if L does not contain ϵ), or a language containing all the words in L, except ϵ; that is, $L - \{\epsilon\}$.

Proof

The elimination method is based on the concept of nullable non-terminals:

In a given CFG, if there is a non-terminal N and a production '$N \to \epsilon$', or if N produces ϵ in one or more steps, that is, '$N \overset{*}{\Rightarrow} \epsilon$', then N is called a *nullable non-terminal*.

For example, consider the grammar:

$$A \to B \mid \epsilon$$
$$B \to a$$

In this grammar, A is a nullable non-terminal, but B is not.

On the other hand, consider the grammar:

$$A \to B \mid a$$
$$B \to a \mid \epsilon$$

In this grammar, since '$B \to \epsilon$' is one of the productions, B is a nullable non-terminal. Likewise, since $A \Rightarrow B \Rightarrow \epsilon$, that is, $A \overset{*}{\Rightarrow} \epsilon$, A is also a nullable non-terminal.

The steps in the process of eliminating ϵ-productions are as follows:

1. Delete all ϵ-productions from the grammar.
2. Identify nullable non-terminals.
3. If there is a production of the form '$A \to \alpha$', where α is any sentential form containing at least one nullable non-terminal, then add new productions, whose right-hand side is formed by deleting all possible subsets of nullable non-terminals from α.
4. If using step 3, we get a production of the form '$A \to \epsilon$', then do not add that to the final grammar.

Let us now work on some examples to illustrate this method.

Example 5.35 Eliminate ϵ-productions from G, where G consists of the following productions:

$$S \to a\,S\,a \mid b\,S\,b \mid \epsilon$$

Solution

After deleting the ϵ-productions we have:

$$S \to a\,S\,a \mid b\,S\,b$$

Since '$S \to \epsilon$' is a production in the given grammar, S is a nullable non-terminal.

The following two productions contain nullable non-terminals on the right-hand side:

$$S \to a\,S\,a$$
$$S \to b\,S\,b$$

In the production '$S \to a\,S\,a$', if we delete S from the right-hand side we can add a new production as:

$$S \to a\,a$$

Similarly, from '$S \to b\,S\,b$', deleting S from the right-hand side, we have a new production as follows:

$$S \to b\,b,$$

which we can add to the final grammar.

Thus, the final grammar, G', without ϵ-productions is as follows:

$$S \to a\,S\,a \mid b\,S\,b \mid a\,a \mid b\,b$$

Comparing G and G', we see that:

$$L(G') = L(G) - \{\epsilon\}$$

Example 5.36 Eliminate ϵ-productions from the grammar G, which is defined as:

$$S \to a \mid X\,b \mid a\,Y\,a$$
$$X \to Y \mid \epsilon$$
$$Y \to b \mid X$$

Compare the language generated by the new grammar with the language generated by G.

Solution

Delete '$X \to \epsilon$' from the final set of productions.

Since there is a production, $X \to \epsilon$, therefore X is a nullable non-terminal.

We also have:

$$Y \Rightarrow X \Rightarrow \epsilon$$
$$\text{Or, } Y \overset{*}{\Rightarrow} \epsilon$$

Since Y produces ϵ in two steps, therefore, Y is also a nullable non-terminal.

Consider the following productions having nullable non-terminals existing on their right-hand side:

$$S \to X\,b$$
$$S \to a\,Y\,a$$
$$X \to Y$$
$$Y \to X$$

From '$S \to X\,b$', if we delete X, we can add the production:

$$S \to b$$

to the final set.

Similarly, from '$S \to a\,Y\,a$', if we delete Y we can add:

$$S \to a\,a$$

to the final set of productions.

From the production '$X \to Y$' and '$Y \to X$' if we delete the nullable non-terminals on the right-hand side, we get productions, '$X \to \epsilon$' and '$Y \to \epsilon$', which we cannot add to the final set.

Thus, the final grammar G' is written as:

$$S \to a \mid X\,b \mid a\,Y\,a \mid b \mid a\,a$$
$$X \to Y$$
$$Y \to b \mid X$$

We observe that, both the languages $L(G)$ and $L(G')$ do not contain ϵ. Therefore, by Theorem 5.1, we can say:

$$L(G) = L(G')$$

Recall that in Example 5.34, we have further reduced this grammar by eliminating the unit productions.

Example 5.37 Eliminate the ϵ-productions from grammar G defined as:

$$S \to X\,a$$
$$X \to a\,X \mid b\,X \mid \epsilon$$

Solution We can delete production '$X \to \epsilon$' from final set of productions. Since '$X \to \epsilon$' is a production in the given set, X is a nullable non-terminal.

Let us consider all productions with at least one nullable non-terminal on the right-hand side:

$$S \to X\,a$$
$$X \to a\,X$$
$$X \to b\,X$$

Deleting X from the right-hand side from these, we obtain three new productions to be added to the final set of productions:

$$S \to a$$
$$X \to a$$
$$X \to b$$

Therefore, the final grammar G' is defined as:

$$S \to X\,a \mid a$$
$$X \to a\,X \mid b\,X \mid a \mid b$$

We observe that ϵ is not a member of $L(G)$

Therefore,

$$L(G) = L(G')$$

Example 5.38 Eliminate ϵ-productions from the grammar G, which is defined as:

$$A \to a\,B\,b \mid b\,B\,a$$
$$B \to a\,B \mid b\,B \mid \epsilon$$

Solution We delete '$B \to \epsilon$' from the final set. Since there is a production '$B \to \epsilon$' in the given set, B is a nullable non-terminal.

Let us consider all productions having B on the right-hand side:

$$A \to a\,B\,b \mid b\,B\,a$$
$$B \to a\,B \mid b\,B$$

Deleting B from these productions, we get four more new productions to add to the final set:

$$A \to a\,b$$
$$A \to b\,a$$
$$B \to a$$
$$B \to b$$

Thus, the final grammar G' can be written as:

$$A \to aBb \mid bBa \mid ab \mid ba$$
$$B \to aB \mid bB \mid a \mid b$$

We further observe that:

$$L(G) = L(G')$$

Example 5.39 Eliminate the ϵ-productions from the grammar G, which is defined as:

$$S \to ABA$$
$$A \to aA \mid \epsilon$$
$$B \to bB \mid \epsilon$$

Solution

We delete the productions '$A \to \epsilon$' and '$B \to \epsilon$' from the final set of productions.

Since '$A \to \epsilon$' and '$B \to \epsilon$' are productions in the given set, both A and B are nullable non-terminals.

Further, we have:

$$S \Rightarrow ABA \Rightarrow BA \Rightarrow A \Rightarrow \epsilon, \text{ using rules '}A \to \epsilon\text{' and '}B \to \epsilon\text{', i.e., } S \overset{*}{\Rightarrow} \epsilon$$

Since, S produces ϵ, S is also a nullable non-terminal.

Let us consider the following productions having nullable non-terminals on the right-hand side:

$$S \to ABA$$
$$A \to aA$$
$$B \to bB$$

From the production '$S \to ABA$', after deleting all possible subsets of nullable non-terminals, we get following new productions:

$$S \to AB$$
$$S \to BA$$
$$S \to AA$$
$$S \to A$$
$$S \to B$$
$$S \to \epsilon$$

Among these, we can add all productions to the final set, except '$S \to \epsilon$'.

Similarly, from '$A \to aA$', we get '$A \to a$', and from '$B \to bB$', we get '$B \to b$' to add to the final set.

Therefore, the final grammar G' is defined as follows:

$$S \to ABA \mid AB \mid BA \mid AA \mid A \mid B$$
$$A \to aA \mid a$$
$$B \to bB \mid b$$

Further, we observe that:

$$L(G') = L(G) - \{\epsilon\}$$

5.10 NORMAL FORMS

Context-free grammars can be written in certain standard forms, known as *normal forms*. These normal forms impose certain restrictions on the productions in the CFG. Complex CFGs can thus be reduced to simple forms after modifying them or rewriting them using these normal forms. In this section, we shall discuss two normal forms: Chomsky normal form (CNF) and Greibach normal form (GNF).

Every CFG without ϵ-productions has an equivalent grammar in Chomsky normal form or Greibach normal form. Equivalent here means that the two grammars generate the same language. Let us discuss these two normal forms in detail.

5.10.1 Chomsky Normal Form

Any CFL without ϵ, which is generated by a grammar in which all productions are of the form:

$$\text{'}A \to BC\text{', or '}A \to a\text{'},$$

where, A, B, and C are non-terminals and a is a terminal symbol, is said to be in Chomsky normal form (CNF).

We see that there are restrictions on the form of productions in CNF, as they can be expressed in only two ways: '$A \to BC$' and '$A \to a$'. In other words, we can only have either two non-terminals or a single terminal on the right-hand side of every production.

If the language has an empty string, that is, ϵ, then only the following ϵ-production is allowed in CNF:

$$S \to \epsilon,$$

where, S is the start symbol.

Example 5.40 Convert the following CFG to CNF.

$$S \to aSa \mid bSb \mid a \mid b \mid aa \mid bb$$

Solution In CNF, we can have only two types of productions: either '$A \to BC$' or '$A \to a$'.

If we add two productions for the aforementioned grammar, namely:

$$A \to a$$
$$B \to b$$

We can rewrite the aforementioned grammar as:

$$S \to ASA \mid BSB \mid a \mid b \mid AA \mid BB \quad \text{(using '}A \to a\text{' and '}B \to b\text{')}$$
$$A \to a$$
$$B \to b$$

We need not change the productions '$S \to a$' and '$S \to b$', because they are already in CNF.

Now, there are only two productions which are not in CNF and hence need to be changed:

$$S \to A\,S\,A$$
$$S \to B\,S\,B$$

To bring them to CNF we introduce two new variables, R_1 and R_2, such that:

$$S \to A\,R_1$$
$$S \to B\,R_2$$
$$R_1 \to S\,A$$
$$R_2 \to S\,B$$

Thus, an equivalent grammar expressed in CNF can be written as:

$$S \to A\,R_1 \mid B\,R_2 \mid a \mid b \mid A\,A \mid B\,B$$
$$A \to a$$
$$B \to b$$
$$R_1 \to S\,A$$
$$R_2 \to S\,B$$

Example 5.41 Convert the following grammar to Chomsky normal form.

$$S \to b\,A \mid a\,B$$
$$A \to b\,A\,A \mid a\,S \mid a$$
$$B \to a\,B\,B \mid b\,S \mid b$$

Solution We see that productions '$A \to a$' and '$B \to b$' are already in CNF.

Now, if we introduce two more productions, R_1 and R_2, such that:

$$R_1 \to a$$
$$R_2 \to b$$

then, the given grammar can be written as:

$$S \to R_2\,A \mid R_1\,B$$
$$A \to R_2\,A\,A \mid R_1\,S \mid a$$
$$B \to R_1\,B\,B \mid R_2\,S \mid b$$
$$R_1 \to a$$
$$R_2 \to b$$

We see that there are still two more productions which are not in CNF; namely, '$A \to R_2\,A\,A$' and '$B \to R_1\,B\,B$'.

After introducing two more new symbols—R_3 and R_4—and further break-up, we write the given grammar in CNF as:

$$S \to R_2\,A \mid R_1\,B$$
$$A \to R_2\,R_3 \mid R_1\,S \mid a$$
$$B \to R_1\,R_4 \mid R_2\,S \mid b$$
$$R_1 \to a$$
$$R_2 \to b$$
$$R_3 \to A\,A$$
$$R_4 \to B\,B$$

Example 5.42 Express the following grammar using CNF.

$$S \to ABA$$
$$A \to a\,A \mid \epsilon$$
$$B \to b\,B \mid \epsilon$$

Solution As we know, before we express a grammar in CNF, we should first remove the ϵ-productions. Hence, we get a new grammar G' such that:

$$L(G') = L(G) - \{\epsilon\}$$

Refer to Example 5.39 in which we have already dealt with this grammar.

The new grammar G' without the ϵ-productions is:

$$S \to A\,B\,A \mid A\,B \mid B\,A \mid A\,A \mid A \mid B$$
$$A \to a\,A \mid a$$
$$B \to b\,B \mid b$$

Let us first eliminate the unit productions; namely, '$S \to A$' and '$S \to B$'.

This is achieved by substituting '$A \to a\,A \mid a$' in '$S \to A$', and '$B \to b\,B \mid b$' in '$S \to B$'.

After removing the unit productions the grammar can be written as follows:

$$S \to A\,B\,A \mid A\,B \mid B\,A \mid A\,A \mid a\,A \mid a \mid b\,B \mid b$$
$$A \to a\,A \mid a$$
$$B \to b\,B \mid b$$

Now, let us introduce three new variables, R_1, R_2, and R_3, as shown in the final grammar. Observe that we have also added the ϵ-productions for S because the language generates the empty string ϵ.

The final grammar is written as follows:

$$S \to A\,R_1 \mid A\,B \mid B\,A \mid A\,A \mid R_2\,A \mid a \mid R_3\,B \mid b \mid \epsilon$$
$$A \to R_2\,A \mid a$$
$$B \to R_3\,B \mid b$$
$$R_1 \to B\,A$$
$$R_2 \to a$$
$$R_3 \to b$$

5.10.2 Greibach Normal Form

Every CFL without ϵ can be generated by a grammar, whose every production is of the form, $A \to a\alpha$, where A is a non-terminal, a is a terminal, and α is a (possibly empty) string containing only non-terminals. This type of grammar is said to be in *Greibach normal form* (GNF).

Thus, in GNF, the right-hand side of each production should contain only one terminal symbol—and that should be the first symbol on the right-hand side—followed by zero or more non-terminals.

If the language has an empty string, that is, ϵ, then only the following ϵ-production is allowed:

$$S \to \epsilon,$$

where, S is the start symbol.

Example 5.43 Convert the following grammar to GNF.

$$S \to A\,B\,A \mid A\,B \mid B\,A \mid A\,A \mid A \mid B$$
$$A \to a\,A \mid a$$
$$B \to b\,B \mid b$$

Solution We see that the productions from S are not in GNF. To bring them to the proper form, we replace the leading A's and B's by their right-hand sides in the productions, '$A \to a\,A$', '$A \to a$', '$B \to b\,B$', and '$B \to b$'.

Therefore, the final grammar in GNF is:

$$S \to a\,A\,B\,A \mid a\,B\,A \mid a\,A\,B \mid a\,B \mid b\,B\,A \mid b\,A \mid a\,A\,A \mid a\,A \mid a\,A \mid a \mid b\,B \mid b$$
$$A \to a\,A \mid a$$
$$B \to b\,B \mid b$$

Example 5.44 Convert the following grammar G to GNF:

$$G = \{(A_1, A_2, A_3), (a, b), P, A_1\},$$

where, P consists of the following productions:

$$A_1 \to A_2\,A_3$$
$$A_2 \to A_3\,A_1 \mid b$$
$$A_3 \to A_1\,A_2 \mid a$$

Solution We observe that '$A_1 \to A_2\,A_3$' is the only production with A_1 on the left-hand side. Let us substitute $A_2\,A_3$ for A_1 in the production '$A_3 \to A_1\,A_2$'.

The resulting set of productions is:

$$A_1 \to A_2\,A_3$$
$$A_2 \to A_3\,A_1 \mid b$$
$$A_3 \to A_2\,A_3\,A_2 \mid a$$

Similarly, let us substitute for the first occurrence of A_2 in the production rule for A_3 with $A_3\,A_1$ and b.

The new set will be:

$$A_1 \to A_2\,A_3$$
$$A_2 \to A_3\,A_1 \mid b$$
$$A_3 \to A_3\,A_1\,A_3\,A_2 \mid b\,A_3\,A_2 \mid a$$

The production '$A_3 \to A_3\,A_1\,A_3\,A_2$' is recursive (the first symbol on the right-hand side of the production is the same as the non-terminal on the the left-hand side). Hence, we introduce a new symbol B_3, and write the following productions:

$$A_3 \to b\,A_3\,A_2\,B_3$$
$$A_3 \to a\,B_3$$
$$B_3 \to A_1\,A_3\,A_2 \mid A_1\,A_3\,A_2\,B_3$$

The resulting set is:

$$A_1 \to A_2\,A_3$$
$$A_2 \to A_3\,A_1 \mid b$$
$$A_3 \to b\,A_3\,A_2\,B_3 \mid a\,B_3 \mid b\,A_3\,A_2 \mid a$$
$$B_3 \to A_1\,A_3\,A_2 \mid A_1\,A_3\,A_2\,B_3$$

Now, the productions for A_3 are all in GNF. Therefore, we can substitute for A_3 in the production '$A_2 \to A_3\,A_1$'. The resulting set of productions is:

$$A_1 \to A_2\,A_3$$
$$A_2 \to b\,A_3\,A_2\,B_3\,A_1 \mid a\,B_3\,A_1 \mid b\,A_3\,A_2\,A_1 \mid a\,A_1 \mid b$$
$$A_3 \to b\,A_3\,A_2\,B_3 \mid a\,B_3 \mid b\,A_3\,A_2 \mid a$$
$$B_3 \to A_1\,A_3\,A_2 \mid A_1\,A_3\,A_2\,B_3$$

Now, the productions for A_2 are also in GNF.

Hence, we substitute for A_2 in '$A_1 \to A_2\,A_3$'. We now have:

$$A_1 \to b\,A_3\,A_2\,B_3\,A_1\,A_3 \mid a\,B_3\,A_1\,A_3 \mid b\,A_3\,A_2\,A_1\,A_3 \mid a\,A_1\,A_3 \mid b\,A_3$$
$$A_2 \to b\,A_3\,A_2\,B_3\,A_1 \mid a\,B_3\,A_1 \mid b\,A_3\,A_2\,A_1 \mid a\,A_1 \mid b$$
$$A_3 \to b\,A_3\,A_2\,B_3 \mid a\,B_3 \mid b\,A_3\,A_2 \mid a$$
$$B_3 \to A_1\,A_3\,A_2 \mid A_1\,A_3\,A_2\,B_3$$

Now, we see that the productions for A_1 are also in GNF. Hence, we substitute these in the production for B_3 to get the final grammar in GNF.

An equivalent grammar in GNF, thus, can be written as:

$$A_1 \to b\,A_3\,A_2\,B_3\,A_1\,A_3 \mid a\,B_3\,A_1\,A_3 \mid b\,A_3\,A_2\,A_1\,A_3 \mid a\,A_1\,A_3 \mid b\,A_3$$
$$A_2 \to b\,A_3\,A_2\,B_3\,A_1 \mid a\,B_3\,A_1 \mid b\,A_3\,A_2\,A_1 \mid a\,A_1 \mid b$$
$$A_3 \to b\,A_3\,A_2\,B_3 \mid a\,B_3 \mid b\,A_3\,A_2 \mid a$$
$$B_1 \to b\,A_3\,A_2\,B_3\,A_1\,A_3\,A_3\,A_2 \mid b\,A_3\,A_2\,B_3\,A_1\,A_3\,A_3\,A_2\,B_3$$
$$\mid a\,B_3\,A_1\,A_3\,A_3\,A_2 \mid a\,B_3\,A_1\,A_3\,A_3\,A_2\,B_3$$
$$\mid bA_3\,A_2\,A_1\,A_3\,A_3\,A_2 \mid b\,A_3\,A_2\,A_1\,A_3\,A_3\,A_2\,B_3$$
$$\mid a\,A_1\,A_3\,A_3\,A_2 \mid a\,A_1\,A_3\,A_3\,A_2\,B_3$$
$$\mid b\,A_3\,A_3\,A_2 \mid bA_3\,A_3\,A_2\,B_3$$

5.11 CHOMSKY HIERARCHY

The class of phrase structure grammars is very large. However, imposing certain constraints on the production rules, different classes of phrase structure grammar can be obtained. It is more of a containment hierarchy, as one class of grammars is more powerful than the

other, and so on. The hierarchy thus obtained is called *Chomsky hierarchy*. This hierarchy of grammars was described by Noam Chomsky in 1956. It is occasionally referred to as *Chomsky–Schützenberger hierarchy*, after Marcel-Paul Schützenberger, who played a crucial role in the development of the theory of formal languages.

Chomsky suggested four different classes of phrase structure grammars, as follows:

1. Type-0 (unrestricted grammar)
2. Type-1 (context-sensitive grammar)
3. Type-2 (context-free grammar)
4. Type-3 (regular grammar)

5.11.1 Unrestricted Grammar (Type-0 Grammar)

There are no restrictions on the productions of a grammar of this type. This type of grammar permits productions of the form '$\alpha \rightarrow \beta$'; $\alpha \neq \epsilon$, where, α and β are sentential forms; that is, any combination of any number of terminals and non-terminals, $\alpha, \beta \subset (V \cup T)^*$; and $\alpha \neq \epsilon$, because there must be something to be replaced by the right-hand side of the production. Such a grammar is called *semi-Thue grammar* or *unrestricted grammar*.

Type-0 grammars (unrestricted grammars) include all formal grammars; they generate exactly all languages that can be recognized by a Turing machine. These languages are also known as *recursively enumerable languages*. This means that we need to construct Turing machines (TMs) to recognize the languages generated by this class of grammars.

For example, let us consider the following grammar:

$$V = \{A, B, C\}$$
$$T = \{a, b, c\}$$
$$G = \{V, T, P, A\},$$

where, P consists of:

$$A \rightarrow A\,B$$
$$A\,B \rightarrow B\,C$$
$$B \rightarrow A\,c\,D$$
$$A\,c \rightarrow a\,b\,c$$
$$D \rightarrow c$$

As we can see, there are no restrictions on the production forms, except that $\alpha \neq \epsilon$. Therefore, it is a type-0 grammar.

5.11.2 Context-sensitive Grammar (Type-1 Grammar)

The restrictions on this type of grammar are as follows:

1. For each production of the form '$\alpha \rightarrow \beta$', the length of β is at least as much as the length of α, except for '$S \rightarrow \epsilon$'.
2. The rule '$S \rightarrow \epsilon$' is allowed only if the start symbol S does not appear on the right-hand side of any production.

3. The term 'context sensitive' is used because the grammar has productions of the form:

$$\alpha_1 A\,\alpha_2 \rightarrow \alpha_1 \beta\,\alpha_2; (\beta \neq \epsilon),$$

where, the replacement of a non-terminal A by β is allowed only if α_1 precedes A and α_2 succeeds A; α_1 and α_2 may or may not be empty.

Context-sensitive grammar (CSG) or type-1 grammar generates *context-sensitive language* (CSL). Turing machines can be constructed to recognize such context-sensitive languages.

For example, consider the grammar G, which is defined as:

$$G = \{(A, B, C), (a, b), P, A\},$$

where, P consists of the productions:

$$A \rightarrow A\,B$$
$$A\,B \rightarrow A\,C$$
$$A\,C \rightarrow a\,b$$

We observe that this grammar is in accordance with the restrictions placed in type-1 or context-sensitive grammar.

5.11.3 Context-free Grammar (Type-2 Grammar)

In this class of grammar:

The only allowed type of production is '$A \rightarrow \alpha$', where, A is a non-terminal, and α is in sentential form, that is, $\alpha \subset (V \cup T)^*$; and α may also be equal to ϵ. The left-hand side of the production, thus, contains only one non-terminal.

The start symbol of the grammar can also appear on right-hand side.

Non-deterministic pushdown automata (NPDA) can accept the whole class of context-free languages (CFL), generated by CFG.

For example, let us consider the following grammar G, which is a CFG:

$$G = \{(S), (a, b), P, S\},$$

where, P consists of the following productions:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b$$

We have already seen many examples and discussed many concepts regarding CFGs in the previous sections.

5.11.4 Regular Grammar (Type-3 Grammar)

This type of grammar has the following restrictions on its productions:

1. The left-hand side of each product should contain only one non-terminal.
2. The right-hand side can contain at most one non-terminal symbol, which is allowed to appear as the rightmost symbol or the leftmost symbol.

The language generated using this grammar is called a *regular language* (RL). Regular languages are too primitive and can be generated and recognized by *finite state machines* (FSMs). These languages are denoted by simpler expressions called regular expressions.

Depending on whether the position of a non-terminal on the right-hand side of the production (if it exists) is leftmost or rightmost, the regular grammar is further classified as follows: Left-linear grammar (G_L) and Right-linear grammar (G_R)

Left-linear Grammar

As we know, regular grammar can contain at the most one non-terminal on the right-hand side of every production. If this non-terminal appears as the leftmost symbol on the right-hand side, then it is said to be left-linear grammar.

The allowed types of productions in left-linear grammar are:

$$A \to B\,w$$
$$A \to w,$$

where, A and B are non-terminals, and w is a string of terminals.

The rule '$S \to \epsilon$' is allowed only if the start symbol S does not appear on the right-hand side of any production.

Consider the following grammar G:

$$G = \{(S, B, C), (a, b), P, S\}$$

where, P consists of:

$$S \to C\,a \mid B\,b$$
$$C \to B\,b$$
$$B \to B\,a \mid b$$

The aforementioned grammar is left-linear, as each production has at the most one non-terminal at the leftmost position on the right-hand side.

Right-linear Grammar

A regular grammar consisting of productions with at most one non-terminal as the rightmost symbol on the right-hand side of every production (if it exists), is said to be right-linear grammar.

The allowed forms of the productions are:

$$A \to w\,B$$
$$A \to w,$$

where, A and B are non-terminals and w is the string of terminals.

The rule '$S \to \epsilon$' is allowed only if the start symbol S does not appear on the right-hand side of any production.

For example, the following grammar G is right-linear:

$$G = \{(S, A), (0, 1), P, S\},$$

where, P consists of:

$$S \to 0A$$
$$A \to 0A \mid 1$$

5.12 EQUIVALENCE OF RIGHT-LINEAR AND LEFT-LINEAR GRAMMARS

The left and the right-linear grammars are said to be equivalent if the sets generated (languages) by them are equal. For every right-linear grammar G_R there exists an equivalent left-linear grammar G_L. Further, we can say that a language is said to be regular if it is either generated by a right-linear grammar or a left-linear grammar.

5.12.1 Conversion of Right-linear Grammar to Equivalent Left-linear Grammar

The following are the steps to convert right-linear grammar to its equivalent left linear grammar:

1. Represent the given right-linear grammar by a transition diagram with vertices labelled by the symbols from $\{V \cup (\epsilon)\}$ and transitions labelled by the symbols from $\{T \cup (\epsilon)\}$.
2. Interchange the positions of the initial and final states.
3. Reverse the directions of all the transitions, keeping the positions of all intermediate states unchanged.
4. Rewrite the grammar from this new transition graph in the left-linear fashion.

The reversal proposed here is due to the fact that right-linear grammar derives the string from left to right, while left-linear grammar derives it in the reverse way, that is, from right to left. This means that the terminal symbol first generated in the case of right-linear grammar will be generated last in case of its equivalent left-linear grammar.

Example 5.45 Convert the following right-linear grammar G_R to its equivalent left-linear grammar:

$$S \to b\,B$$
$$B \to b\,C$$
$$B \to a\,B$$
$$B \to b$$
$$C \to a$$

Solution Let us construct a transition graph (TG) using the given right-linear grammar. The initial state will be labelled by start symbol S, and the final state will be labelled by ϵ. The TG is shown in Fig. 5.6(a).

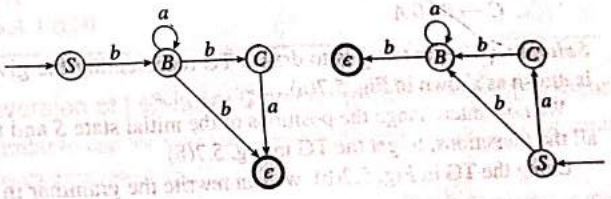


Figure 5.6 Conversion of right-linear grammar to equivalent left-linear grammar (a) TG constructed from right-linear grammar (b) Resultant TG after interchanging initial and final states and reversing all the transitions

After interchanging the positions of the initial and the final states and reversing the directions of all the transitions of the TG, we get the TG for the equivalent left-linear grammar as shown in Fig. 5.6(b).

From the TG in Fig. 5.6(b), we can rewrite the left-linear grammar G_L as follows:

$$S \to C\,a$$
$$S \to B\,b$$
$$C \to B\,b$$
$$B \to B\,a$$
$$B \to b$$

This left-linear grammar is equivalent to the given right-linear grammar. Let us check this for a string '*baab*'.

Using G_R, we can derive the given string as:

$$S \Rightarrow b\,B \Rightarrow b\,a\,B \Rightarrow b\,a\,a\,B \Rightarrow b\,a\,a\,b$$

Now, using the equivalent left-linear grammar G_L, we derive the same string as:

$$S \Rightarrow B\,b \Rightarrow B\,a\,b \Rightarrow B\,a\,a\,b \Rightarrow b\,a\,a\,b$$

Thus, we see that the left-linear grammar that we have obtained generates the same language as that of the given right-linear grammar. The only change we observe is that the right-linear grammar begins to derive the given string from the leftmost symbol, while the left-linear grammar derives it in the reverse direction, that is, from the rightmost symbol. In this example, we see that the first symbol of the string '*baab*', that is, '*b*', is generated first using right-linear grammar, and last using left-linear grammar.

We also see that the intermediate sentential forms in both the derivations are mirror images of each other. For example, '$B\,b$' and '$b\,B$'; '$b\,a\,B$' and '$B\,a\,b$'; and '$b\,a\,a\,B$' and '$B\,a\,a\,b$'.

Example 5.46 Write an equivalent left-linear grammar for the right-linear grammar, which is defined as:

$$S \to 0\,A \mid 1\,B$$
$$A \to 0\,C \mid 1\,A \mid 0$$
$$B \to 1\,B \mid 1\,A \mid 1$$
$$C \to 0 \mid 0\,A$$

Solution The first step is to draw a TG representing the given right-linear grammar. It is drawn as shown in Fig. 5.7(a).

We now interchange the positions of the initial state S and the final state ϵ and reverse all the transitions, to get the TG in Fig. 5.7(b).

Using the TG in Fig. 5.7(b), we can rewrite the grammar in left-linear fashion as:

$$S \to B\,1 \mid A\,0 \mid C\,0$$
$$B \to B\,1 \mid 1$$
$$A \to A\,1 \mid B\,1 \mid C\,0 \mid 0$$
$$C \to A\,0$$

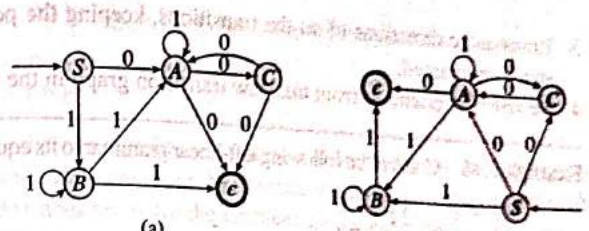


Figure 5.7 G_R to G_L conversion (a) TG representing G_R (right-linear grammar) (b) TG after interchanging initial and final states and reversing the transitions

---

Example 5.47 Write an equivalent left-linear grammar for the following right-linear grammar:

$$S \to 0\,A$$
$$A \to 1\,0\,A \mid \epsilon$$

Solution To start with, we draw the TG representing the aforementioned right-linear grammar, as shown in Fig. 5.8(a).

After interchanging the positions of the initial state S and final state ϵ, and reversing all the transitions of the TG in Fig. 5.8(a), we get the TG as shown in Fig. 5.8(b).

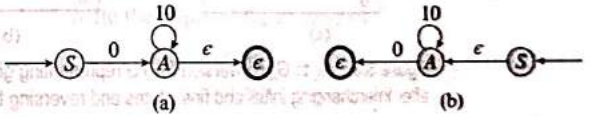


Figure 5.8 G_R to G_L conversion (a) TG representing G_R (b) TG after interchanging initial and final vertices and reversing the transitions

From the TG in Fig. 5.8(b), we can write the left-linear grammar as follows:

$$S \to A\,\epsilon$$
$$A \to A\,1\,0 \mid 0$$

The grammar can be further simplified as follows:

$$S \to A$$
$$A \to A\,1\,0 \mid 0$$

5.12.2 Conversion of Left-linear Grammar to Equivalent Right-linear Grammar

A method similar to that we have seen while converting G_R to G_L can be applied to convert a given G_L to its equivalent G_R.

1. Represent the given left-linear grammar by a transition diagram (or transition graph) with vertices labelled by the symbols from $\{V \cup (\epsilon)\}$, and transitions labelled by the symbols from $\{T \cup (\epsilon)\}$.
2. Interchange the positions of the initial and final states.

3. Reverse the directions of all the transitions, keeping the positions of all intermediate states unchanged.
4. Rewrite the grammar from this new transition graph in the right-linear fashion.

Example 5.48 Convert the following left-linear grammar to its equivalent right-linear grammar.

$$S \to B1 \mid A0 \mid C0$$
$$A \to C0 \mid A1 \mid B1 \mid 0$$
$$B \to B1 \mid 1$$
$$C \to A0$$

Solution The transition graph (TG) representing the given G_L is drawn as shown in Fig. 5.9(a).

The TG can be modified by interchanging the positions of the initial and final states and reversing all the transitions as shown in Fig. 5.9(b).

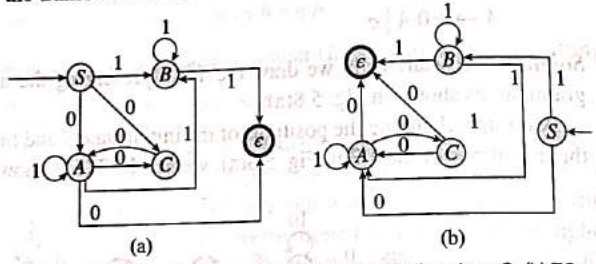


Figure 5.9 G_L to G_R conversion (a) TG representing given G_L (b) TG after interchanging initial and final states and reversing the transitions

Using the TG in Fig. 5.9(b), we can write the equivalent G_R as:

$$S \to 0A \mid 1B$$
$$A \to 1A \mid 0C \mid 0$$
$$B \to 1B \mid 1A \mid 1$$
$$C \to 0A \mid 0$$

Example 5.49 Write an equivalent G_R for the following G_L:

$$S \to S10 \mid 0$$

Solution The TG for the given G_L can be drawn as shown in Fig. 5.10(a).

After interchanging the positions of the initial and final states, and reversing all the transitions, we can get the TG as shown in Fig. 5.10(b).

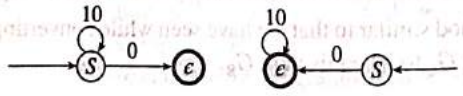


Figure 5.10 G_L to G_R conversion (a) TG representing G_L (b) TG after interchanging initial and final states and reversing the transitions

Using the TG in Fig. 5.10(b), we write the required G_R as follows:

$$S \to 0\epsilon$$
$$\epsilon \to 10\epsilon$$

In this case, there is a problem, because ϵ seems to be part of a production, which is not really allowed in any grammar. An alternative approach is to change the label of the ϵ state. However, this does not solve the problem, as the grammar we obtained is not equivalent.

This problem occurs because S has a self-loop in the TG in Fig. 5.10(a), because of the production '$S \to S10$'. As a result, in Fig. 5.10(b), the loop is on the state labelled ϵ, obviously the language generated has changed.

In order to solve this problem, we first modify the given G_L by introducing a new variable A, and rewriting the productions as follows:,

$$S \to A$$
$$A \to A10 \mid 0$$

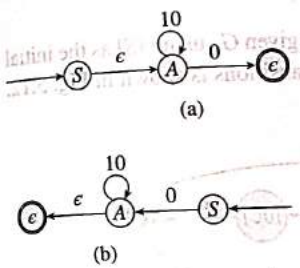


Figure 5.11 G_L to G_R conversion (a) TG representing G_L (b) TG after modification

We see that now there is no loop on the start symbol S. Hence, we can solve the problem using the usual method:

The modified G_L is represented as a TG shown in Fig. 5.11(a). After reversing the transitions and interchanging the initial and final vertices, we get the TG shown in Fig. 5.11(b).Using the TG in Fig. 5.11(b), we write the required G_R as follows:

$$S \to 0A$$
$$A \to 10A \mid \epsilon$$

We see that the right-linear grammar obtained is now equivalent to the original left-linear grammar.

5.13 EQUIVALENCE OF REGULAR GRAMMARS AND FINITE AUTOMATA

As we know, regular grammars can generate regular languages (or regular sets) and the same can be accepted by an FA. In Section 5.12, we have represented right-linear as well the left-linear grammar using TGs, which are nothing but finite automata. This means that while inter-converting G_R and G_L, we have assumed the fact that FA and regular grammars are equivalent. Observe that he transition graphs that we have obtained are NFA.

Hence, let us discuss the conversion algorithm for obtaining FA equivalent to a given regular grammar in more detail.

5.13.1 Right-linear Grammar and FA

Let us suppose that $G = \{V, T, P, S\}$ is some right-linear grammar. We can construct an NFA with ϵ-moves (ϵ-transitions), $M = \{Q, T, \delta, (S), (\epsilon)\}$, that simulates derivations in G, where:

1. Q consists of a symbol $[\alpha]$ such that α is either S or a (not necessarily proper) suffix of some right-hand side of a production in P;

2. δ is defined as:
   (a) If A is a variable, then $\delta([A], \epsilon) = \{[\alpha] \mid A \to \alpha$ is a production$\}$
   (b) If a is in T, and α is in $(T^* \cup T^* \cdot V)$, then $\delta([a\alpha], a) = \{[\alpha]\}$

This NFA with ϵ-moves can then be converted to DFA (refer to Chapter 2 for the detailed conversion algorithm).

Let us discuss some examples to illustrate the construction of FA from a given right-linear grammar.

Example 5.50 Find the equivalent DFA accepting the regular language defined by the following right-linear grammar:

$S \to 0A \mid 1B$
$A \to 0C \mid 1A \mid 0$
$B \to 1B \mid 1A \mid 1$
$C \to 0 \mid 0A$

Solution We can construct an NFA with ϵ-moves for the given G_R using $[S]$ as the initial state and $[\epsilon]$ as the final state. The resultant NFA with ϵ-transitions is shown in Fig. 5.12.

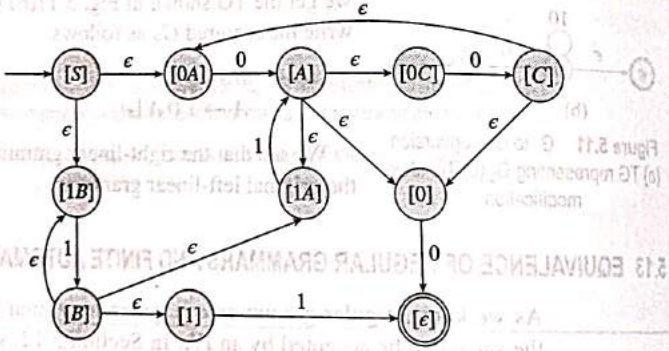


Figure 5.12 NFA with ϵ-moves drawn from given G_R

Table 5.1 Transition table for DFA in Fig. 5.13(a)

Q \ Σ	0	1
A	B	D
B	C	B
*C	E	—
D	—	F
*E	C	B
*F	C	F

'*': Final states

As S produces '$S \to 0\ A$' and '$S \to 1\ B$', we have introduced two ϵ-transitions from $[S]$. Since $[0A]$ is in the form $[a\alpha]$, the transition from $[0A]$ to $[A]$ is shown on symbol 0.

Similarly, for state $[1B]$ it is shown on 1 to state $[B]$. Repeating the same procedure for $[A]$ and $[B]$, we get the NFA with ϵ-moves as shown in Fig. 5.12. In the process, we must *take care that no state is repeated.*

The equivalent DFA can then be obtained as shown in Fig. 5.13(a).

We can change the labels of the states as shown in Fig. 5.13(a). The new labels are the symbols from A to F.

The state transition table for the same is shown in Table 5.1.

We see that it is not possible to reduce this table further—though states B and E have the same transitions, they are not equivalent because B is a non-final state and E is a final state. Hence, the final DFA remains with the six states as shown in Fig. 5.13(b).

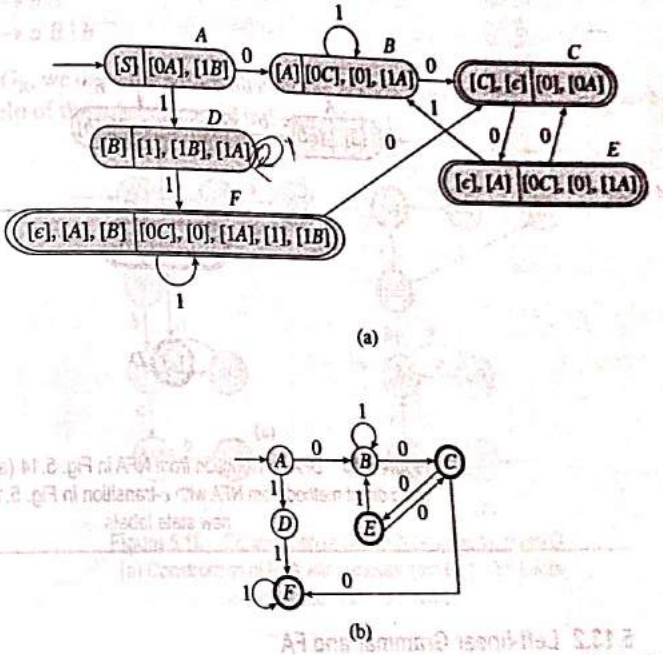


Figure 5.13 Equivalent DFA (a) DFA obtained from NFA with ϵ-moves in Fig. 5.12 (b) DFA for the right-linear grammar

Example 5.51 Construct a DFA that accepts the same regular language defined by the following right-linear grammar:

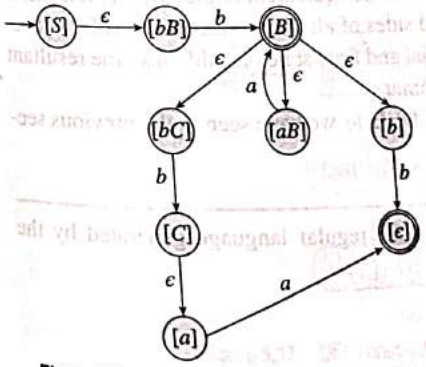


Figure 5.14 NFA with ϵ-moves for given G_R

$S \to b\ B$
$B \to b\ C \mid a\ B \mid b$
$C \to a$

Solution Using the given G_R, we construct an NFA with ϵ-transitions as shown in Fig. 5.14.

The NFA with ϵ-moves thus obtained contains $[S]$ as the initial state and $[\epsilon]$ as the final state. Starting with the production '$S \to bB$', and thereafter repeatedly applying the rules given, we obtain this diagram.

This NFA with ϵ-moves can be directly converted to its equivalent DFA as shown in Fig. 5.15(a).

Table 5.2 State transition table for DFA in Fig. 5.15(a)

	a	b
A	—	B
B	B	C
*C	D	—
*D	—	—

'*': Final states

We can change the labels of the states as shown in Fig. 5.15(a). The state transition table for the resultant DFA is as shown in Table 5.2.

The table cannot be reduced further and therefore, we proceed to draw the required DFA with new labels as shown in Fig. 5.15(b).

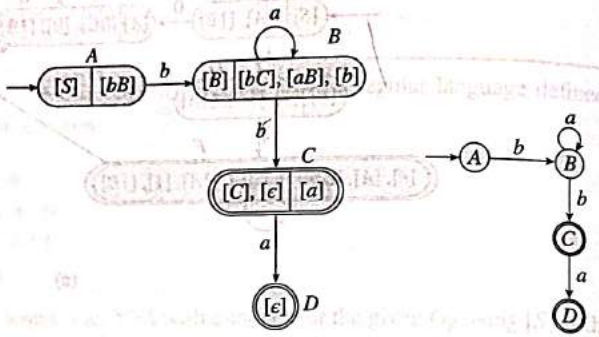


Figure 5.15 DFA construction from NFA in Fig. 5.14 (a) DFA obtained using direct method from NFA with ϵ-transition in Fig. 5.14 (b) DFA with new state labels

5.13.2 Left-linear Grammar and FA

Let $G = \{V, T, P, S\}$ be a left-linear grammar. The first step is to obtain $G' = \{V, T, P', S\}$, the right-linear grammar with P′ consisting of the productions in G with right sides reversed, which are not equivalent to the original G_L.

From this right-linear grammar, an NFA with ϵ-moves is constructed as discussed in Section 5.13.1. We now reverse all the transitions and interchange the positions of the initial and final states to obtain the NFA with ϵ-moves equivalent to the original left-linear grammar. Note that we reversed the right-hand sides of all the productions just for convenience—the reversal of transitions and the initial and final states would make the resultant NFA equivalent to the original left-linear grammar.

This NFA is now converted to its equivalent DFA, as we have seen in the previous section (and in Chapter 2).

Example 5.52 Construct a DFA that accepts the regular language generated by the following left-linear grammar:

$$S \to C\,a \mid B\,b$$
$$C \to B\,b$$
$$B \to B\,a \mid b$$

Solution As the first step, we reverse the right-hand sides of the productions in the given G_L, to obtain a right-linear grammar (not equivalent to the given G_L). After reversing we get the productions:

$$S \to a\,C \mid b\,B$$
$$C \to b\,B$$
$$B \to a\,B \mid b$$

Using this G_R, we draw the NFA with ϵ-moves as shown in Fig. 5.16(a), which is obtained with the help of the rules discussed in Section 5.13.1.

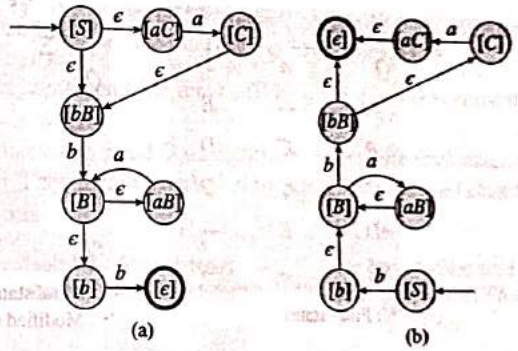


Figure 5.16 Construction of NFA with ϵ-moves for given G_L (a) Construction of NFA with ϵ-moves from G_R (b) NFA with ϵ-moves equivalent to given G_L

Figure 5.16(a) can be modified by interchanging the positions of the initial and final states, and by reversing the transitions to get the modified NFA with ϵ-moves equivalent to the given G_L. This is shown in Fig. 5.16(b).

This NFA with ϵ-moves can then be converted to its equivalent DFA as shown in Fig. 5.17(a).

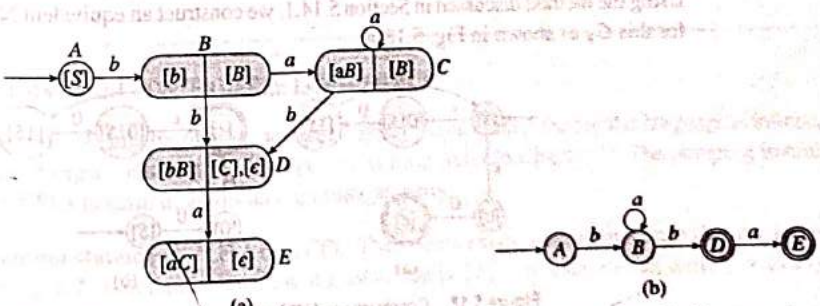


Figure 5.17 DFA construction from the NFA in Fig. 5.16 (a) DFA obtained from NFA with ϵ-moves in Fig. 5.16(b) (b) Final minimized DFA after relabelling the states

The DFA states are relabelled using letters from A to E, as shown in Fig. 5.17(a). Now, let us check if we can reduce this diagram further. For this, we draw the state transition table for the DFA as shown in Table 5.3.

We observe that states B and C are equivalent, as both are non-final states and have the same transitions on a and b.

Thus, we replace C by B and accordingly modify the state transition table as shown in Table 5.4.

Table 5.3 State transition table for the DFA in Fig. 5.17(a)

Q \ Σ	a	b
A	—	B
B	C	D
C	C	D
$*D$	E	—
$*E$	—	—

'*': Final states

Table 5.4 Reduced state transition table for the DFA in Fig. 5.17(b)

Q \ Σ	a	b
A	—	B
B	$B•$	D
$*D$	E	—
$*E$	—	—

'*': Final states
'•': Modified entry

The final minimized DFA is shown in Fig. 5.17(b).

Example 5.53 Construct a DFA equivalent to the following left-linear grammar:

$$S \rightarrow S\,1\,0 \mid 0$$

Solution We first reverse the right-hand sides of all the productions of the given G_L to obtain a right-linear grammar (not equivalent to the given G_L).

Thus, we have:

$$S \rightarrow 0\,1\,S \mid 0$$

Using the method discussed in Section 5.14.1, we construct an equivalent NFA with ϵ-moves for this G_R as shown in Fig. 5.18(a).

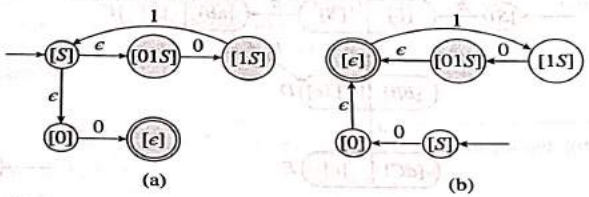


Figure 5.18 Construction of NFA with ϵ-moves equivalent to given G_L (a) NFA with ϵ-moves for G_R obtained by reversing productions of given G_L (b) NFA with ϵ-moves equivalent to given G_L

We then interchange the positions of the initial and final states and reverse the transitions to get the NFA with ϵ-moves that is equivalent to the given left-linear grammar. This is shown in Fig. 5.18(b).

This NFA with ϵ-moves is converted into its equivalent DFA as shown in Fig. 5.19(a).

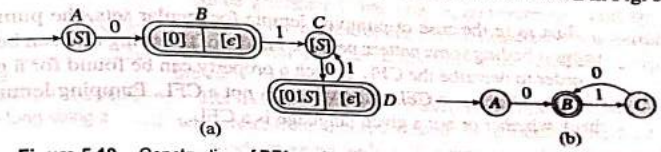


Figure 5.19 Construction of DFA equivalent to NFA in Fig. 5.18 (a) DFA equivalent to NFA with ϵ-moves in Fig. 5.18(b) (b) Final DFA

Next, the state transition table for the DFA in Fig. 5.19(a) is constructed as shown in Table 5.5.

We observe that states B and D are equivalent as both are final states and both have the same transitions. Therefore, we replace D by B to get the reduced state transition table as shown in Table 5.6.

Table 5.5 State transition table for DFA in Fig. 5.19(a)

Q \ Σ	0	1
A	B	—
$*B$	—	C
C	D	—
$*D$	—	C

'*': Final states

Table 5.6 Modified state transition table for the DFA in Fig. 5.19(b)

Q \ Σ	0	1
A	B	—
$*B$	—	C
C	$B•$	—

'*': Final states
'•': Modified entry

Using this reduced transition table, we draw the TG for the final DFA as shown in Fig. 5.19(b).

5.14 PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

Pumping lemma for regular languages signify the fact that if the regular language is infinite, then it must contain strings of a particular form (refer to Chapter 3). The pumping lemma for CFLs is similar, and is an important property.

Lemma statement Let L be a CFL. Then, there exists a constant n such that if z is any string in L such that its length is at least n, that is, $|z| \geq n$, then we can write $z = uvwxy$, where:

1. $|vwx| \leq n$; that is, the middle portion of the string is not too long.

2. $|vx| \geq 1$; that is, $vx \neq c$. Since, v and x are the substrings that get pumped, it is required that at least one of them should be non-empty.
3. For all values of i; $i \geq 0$, uv^iwx^iy is in L; that is, the two substrings v and x can be pumped as many times as required and the resultant string obtained will be a member of L.

Just as in the case of pumping lemma for regular sets, the pumping lemma for CFLs suggest finding some pattern near the middle of the string that can be pumped (or repeated) in order to describe the CFL. If such a property can be found for a given language L, then L is considered as a CFL; otherwise L is not a CFL. Pumping lemma can thus be used to check whether or not a given language is a CFL.

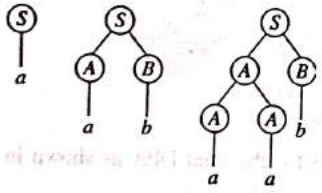
Alternate statement Let $G = \{V, T, P, S\}$ be a context-free grammar represented in Chomsky normal form with m number of non-terminals, that is, $|V| = m$. Then if z is any string in $L(G)$, and $|z| \geq 2^{m-1} + 1$, then z can be written as, $z = uvwxy$ such that, $|vx| \geq 1$ and $|vwx| \leq 2^m$; and for all $i \geq 0$, uv^iwx^iy is in $L(G)$.

Proof

Since G is in CNF, each production is of the form '$A \to BC$' or '$A \to a$'. Thus, any sub-tree of a parse tree of height h will have a yield at most 2^{h-1} number of leaves. This is the property of any binary tree, and is true even in case of CNF grammar—as any parse (or derivation) tree for a CNF grammar is a binary tree.

For example, consider grammar G expressed in CNF with the following productions:

$$S \to AB \mid a$$
$$A \to AA \mid a$$
$$B \to b$$



(a) (b) (c)

Figure 5.20 Parse trees with different heights for CNF grammar (a) Height = 1 (b) Height = 2 (c) Height = 3

Parse trees of heights 1, 2, and 3 for these productions are depicted as shown in Fig. 5.20.

We observe that in Fig. 5.20(a), the parse tree with height 1 has $2^{1-1} = 2^0 = 1$ leaf. Similarly, in Fig. 5.20(b), the parse tree with height 2 has $2^{2-1} = 2^1 = 2$ leaves; that is, its yield length is 2.

However, Fig. 5.20(c) is not a complete binary tree and has yield 3. Still, even if we consider a balanced binary tree with height 3, it can have a maximum of 4 leaves: $2^{3-1} = 2^2 = 4$. Hence, the maximum possible yield for a parse tree of height h is 2^{h-1}.

We also observe that the longest path for a binary parse tree of height h is h; and it contains at least h non-terminals.

For example, in Fig. 5.20(b) with parse tree of height 2, the length of the longest path is 2—there are 2 paths actually, both of length 2; one leads to a and the other leads to b—and each path has 2 non-terminals in it—the path that leads to a has S followed by A, and the path that leads to b has S followed by B in it. Similarly, we see that for the binary parse tree in Fig. 5.20(c) with height 3, the length of the longest path is 3, and has 3 non-terminals in it.

Now, if there is a string z in $L(G)$ such that $|z| \geq 2^{m-1} + 1$ then, from the aforementioned properties of the binary trees, we can say that the height of the parse tree for z is

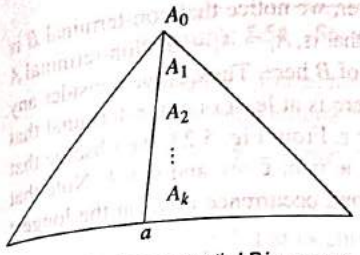


Figure 5.21 Longest path LP in a parse tree for a sufficiently long string z

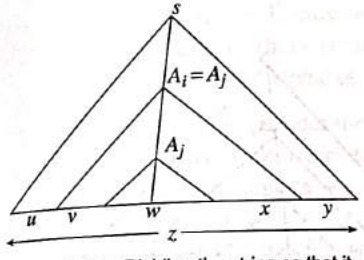


Figure 5.22 Dividing the string so that it can be pumped

greater than m. The minimum height of such a parse tree is '$m + 1$', which means that the length of any longest path, LP, on this parse tree is more than m; that is, $|LP| \geq m + 1$. Thus, the longest path LP contains more than m non-terminals in it.

Since grammar G has only m non-terminals and any longest path LP in the parse tree for z has more than m non-terminals, we conclude that there are a few non-terminals that are repeated on the LP.

Let us consider one such longest path LP as shown in Fig. 5.21 with more than m non-terminals in it, labelled as, $A_0, A_1, \ldots, A_k$.

Since there are only m distinct non-terminals in set V, at least two of the *last* '$m + 1$' *non-terminals* on the longest path LP must be the same non-terminals.

Let us suppose $A_i = A_j$, where $k - m \leq i < j \leq k$. Hence, it is possible to divide the parse tree for string z as shown in Fig. 5.22.

Substring w is the yield of the sub-tree, whose root node is A_j; and substrings v and x that are to the left and right of w respectively, are part of the yield of the larger sub-tree, whose root is A_i. Similarly, u and y are the portions of z that are to the left and right respectively of the sub-tree rooted at A_i.

Remember that we picked up A_i to be close to the bottom of the parse tree, that is, $k - i \leq m$. We have seen earlier that at least two of the *last* $m + 1$ *non-terminals* on the longest path LP must be the same non-terminals. Hence, we conclude that the longest path for the sub-tree rooted at A_i is no longer than $m + 1$; this means that the length of the yield of the sub-tree rooted at A_i, which is vwx, is given by: $|vwx| \leq 2^m$. This proves condition one of the pumping lemma.

Next, we note that both v and x cannot be c at the same time; but, one of them could be. This is beeause, in CNF we do not have any unit productions. Hence, if a non-terminal is repeated on the longest path to the second occurrence of the same non-terminal, there would be some non-empty substring already derived to the left of the second occurrence of the same non-terminal. This means that $|vx| \geq 1$. This proves condition two of the pumping lemma.

In order to prove condition three of the pumping lemma, let us consider the grammar G expressed in CNF having productions:

$$\{S \to AB \mid a; A \to AB \mid a; B \to BA \mid b\}$$

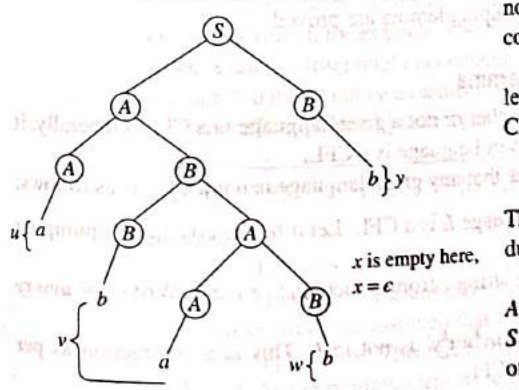


Figure 5.23 Parse tree illustrating the string division

The parse tree for the string '*ababb*' using these productions is shown in Fig. 5.23.

We see that there are only 3 non-terminals in G: A, B, and S, that is, $m = 3$. Refer to the longest path, $S \to A \to B \to A \to B \to b$. We see that the length of the path is 6, which is more than the number of non-terminals. We also see that the string z is '*ababb*',

which satisfies the condition $|z| \geq 2^{m-1} + 1$. However, we notice that non-terminal B is marked grey in Fig. 5.23, showing that it is repeated, that is, $A_i = A_j = B$. Non-terminal A is also repeated, but we are considering the example of B here. Thus, if we consider any sufficiently long string z from the language, then, there is at least one non-terminal that is repeated in the longest path from the parse tree for z. From Fig. 5.23, we observe that z can be written as $z = uvwxy$, where $u = a$, $v = ba$, $w = b$, $x = c$ and $y = b$. Note that $v = ba$, is the string generated to the left of the second occurrence of B on the longest path considered here.

Since A_i and A_j are the same non-terminals in the parse tree in Fig. 5.22, we replace A_i by A_j and draw a new parse tree as shown in Fig. 5.24(a). Similarly, if we replace, A_j by A_i, we get a parse tree as shown in Fig. 5.24(b).

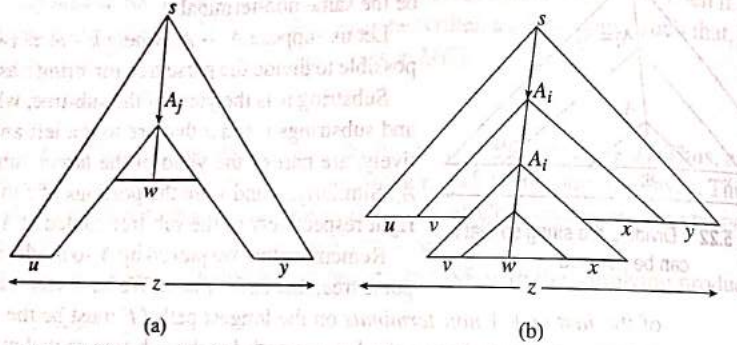


Figure 5.24 Pumping the substrings v and x (a) Parse tree after replacing A_i with A_j (b)Parse tree after replacing A_j with A_i

In the parse tree shown in Fig. 5.24(a), the yield is 'uwy', as we have replaced A_i by A_j; on the other hand, in parse tree shown in Fig. 5.24(b), the yield is 'uv^2wx^2y' as we have replaced A_j by A_i. Thus, both represent the yield of the form 'uv^iwx^iy', for $i = 0$ and $i = 2$ respectively. This proves condition 3. of the pumping lemma.

Hence, all the conditions of the pumping lemma are proved.

5.14.1 Application of Pumping Lemma

Pumping lemma is used to check whether or not a given language is a CFL. Generally, it is used as a tool to disprove that a given language is a CFL.

The procedure that is used to show that any given language is not a CFL is as follows:

Step 1: Assume that the given language L is a CFL. Let n be the constant of pumping lemma.

Step 2: Choose a sufficiently long string z from L such that $|z| \geq n$. Write $z = uvwxy$ using pumping lemma.

Step 3: Find a suitable k so that 'uv^kwx^ky' is not in L. This is a contraction as per pumping lemma; hence, L is not a CFL.

Example 5.54 Show that $L = \{a^nb^nc^n \mid n \geq 1\}$ is not a context-free language.

Solution Let us not confuse with the n used in the language definition as the constant n of pumping lemma. Hence, we rewrite the language definition as:

$$L = \{a^mb^mc^m \mid m \geq 1\}.$$

Step 1: Let us assume that the language L is a CFL.

Step 2: Let us choose a sufficiently large string z, such that $z = a^lb^lc^l$, for some large $l > 0$. Since we have assumed that L is a CFL and is an infinite language, pumping lemma can be applied. This means that we should be able to write z as: $z = uvwxy$.

Step 3: As per pumping lemma, every string 'uv^iwx^iy', for all $i \geq 0$ is in L. Since $|vx| \geq 1$, only one among v and x can be empty. Without loss of generality, let us assume that v is non-empty. As v is a substring, it can contain any of the three symbols from the alphabet $\{a, b, c\}$ for the language L.

Let us consider the case in which v contains a single symbol from $\{a, b, c\}$. Therefore, $z = uvwxy = a^lb^lc^l$; this means that the number of a's, b's, and c's in z are same. As per pumping lemma, we would expect 'uv^2wx^2y' to be a member of L, which does not seem to be the case as v contains only a single symbol and pumping v would yield different number of a's, b's, and c's. Thus, 'uv^2wx^2y' is not a member of L. This contradicts our assumption that L is a CFL.

Now, let us consider the case in which v contains two of the three symbols, say, a and b. The sample v could be written as ab, $aabb$, and so on. When we try to pump v multiple times; for example, $v^2 = abab$, or $v^2 = aabbaabb$, and so on, we find that even number of b's follow a in the string, which is against the language definition: '$a^mb^mc^m$' where the a's are followed by b's, which are followed by c's. Thus, uv^2wx^2y is not a member of L. This again contradicts our assumption that L is a CFL.

Similarly, let us consider the case in which v contains all the three symbols; pumping it will result in a situation, where we will have b's or c's followed by a's, which is not allowed as per language definition.

Finally, let us consider the case when v contains one symbol; x is also non-empty and contains one symbol; for example, $v = a^i$, and $x = c^j$, for some $i, j \leq m$. Even if we pump v and x, the resultant string might not contain the same number of a's, b's, and c's. Thus, 'uv^iwx^iy' for all $i \geq 0$ might not be a member of L. This contradicts our assumption that L is a CFL.

Hence, $L = \{a^mb^mc^m \mid m \geq 1\}$ is not a CFL.

Example 5.55 Show that $L = \{a^p \mid p \text{ is prime}\}$ is not a context-free language.

Solution

Step 1: Let us assume that the language L is a CFL.

Step 2: Let us choose a sufficiently large string z such that $z = a^l$ for some large l, which is prime. Since we assumed that L is a CFL and an infinite language; pumping lemma can be applied now. This means that we should be able to write a string: $z = uvwxy$.

Step 3: As per pumping lemma, every string 'uv^iwx^iy', for all $i \geq 0$ is in L.

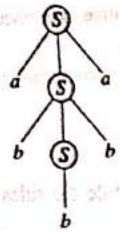


Figure 5.25 Derivation tree for 'abbba'

The derivation tree for the same can be drawn as shown in Fig. 5.25. We see that the node S occurs three times in the tree.

Moreover, the derivation tree does not give any information about the production rule that has been applied at each particular stage in the derivation.

On the other hand, in the derivation graph, the nodes are labelled by the production rule numbers that are applied at each stage in the derivation process; and there is no repetition of any node unless the same rule is applied again.

Usually, derivations of any string from the languages generated using type-0 and type-1 grammars are represented by a more general directed planar graph, known as a *derivation graph*.

Note: A planar graph is one that can be drawn on a two-dimensional plane without intersecting any two edges, except at the nodes.

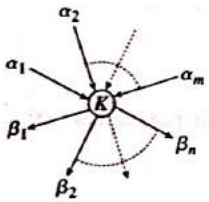


Figure 5.26 Node K in a derivation graph representing the production, $K: \alpha \rightarrow \beta$

Let a set P of productions be ordered and labelled as $\{1, 2, 3, \ldots, K\}$, and let '$\alpha \rightarrow \beta$' be the Kth production rule in P. Further, let $\alpha_1, \alpha_2, \ldots, \alpha_m$, and $\beta_1, \beta_2, \ldots, \beta_n$, be the left to right ordering of the symbols in the sentential forms, α and β respectively. Then, the node representing the production K, whose incoming directed edges (or arcs) from left to right are labelled as $\alpha_1, \alpha_2, \ldots, \alpha_m$ and outgoing arcs from left to right are labelled as $\beta_1, \beta_2, \ldots, \beta_n$ in the derivation graph for any string, as shown in Fig. 5.26.

Further, when β_i for $i = 1, 2, \ldots, n$ is a terminal, it ends in a node labelled ϵ.

The set of edges ending in the ϵ-nodes is called the *frontier* and the sequence of edge labels from left to right is called the *yield*, and gives a sentential form in the derivation.

The sequence of labels of edges ending in the ϵ-nodes from left to right gives us the string for which the derivation graph is prepared.

Example 5.56 Consider a type-0 grammar G defined as:

$$G = \{(S, A, B, C, D), (a, b, +), P, S\},$$

where P consists of the following productions:

1: $S \rightarrow ACS$
2: $S \rightarrow BDS$
3: $S \rightarrow +$
4: $CA \rightarrow AC$
5: $CB \rightarrow BC$
6: $DA \rightarrow AD$
7: $DB \rightarrow BD$
8: $D+ \rightarrow +b$
9: $C+ \rightarrow +a$
10: $A \rightarrow a$
11: $B \rightarrow b$

Prepare the derivation graph for the string '$ab+ab$'.

Solution Let us first derive the string '$ab+ab$' using the given set of productions:

$S \Rightarrow ACS$, (rule 1)
$\Rightarrow ACBDS$, (rule 2)
$\Rightarrow ACBD+$, (rule 3)
$\Rightarrow ACB+b$, (rule 8)
$\Rightarrow ABC+b$, (rule 5)
$\Rightarrow AB+ab$, (rule 9)
$\Rightarrow aB+ab$, (rule 10)
$\Rightarrow ab+ab$, (rule 11)

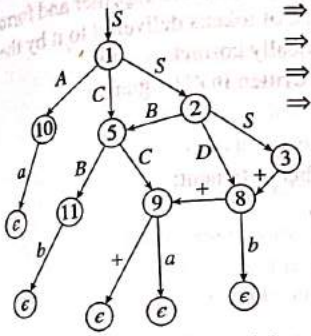


Figure 5.27 Derivation graph for 'ab +ab'

This derivation process can now be represented with the help of a derivation graph, as shown in Fig. 5.27.

Observing the figure, we can easily conclude that derivation graphs are more helpful than derivation trees, as we get exact information about how the string is derived and which productions are applied at every stage in the derivation. Moreover, the derivation graph is suitable for all types of grammars unlike derivation trees, which only can support type-2 and type-3 grammars.

5.18 APPLICATIONS OF CONTEXT-FREE GRAMMAR

Context-free grammars have applications in many areas, from compiler construction to syntactic pattern recognition. Some subsets of deterministic CFLs (DCFLs) are being used to represent the programming languages. Hence, CFGs are used to denote the syntax of these programming languages.

5.18.1 Parser (or Syntax Analyser)

In Chapter 3, we have discussed in detail, the applications of regular expressions and FA, and how these can be used in the construction of lexical analysers, which is the first phase of a compiler. Any compiler can be subdivided into five different phases/modules; namely:

1. Lexical analyser (or scanner)
2. Syntax analyser (or parser)
3. Intermediate code generator
4. Code optimizer
5. Code generator

Lexical analyser uses a set of regular expressions and builds a general-purpose DFA. Such DFA are used to detect the valid words—called tokens—from a given source file, considering it as a text file. The validity of the words depends on whether or not they fit any one of the patterns defined by the set of regular expressions. Any programming language, for example C, contains only five types of valid words, namely, identifiers, literals, punctuations, keywords, and operators.

After identifying the tokens, the lexical analyser passes these tokens to the second phase of the compiler, that is, the parser or syntax analyser, which checks the syntax. Syntax checking here means checking whether the tokens are in the right sequence as defined by the context-free grammar. Since CFGs are used to define the language syntax, they can be used to check the syntax of any program written in the given language.

A parser basically uses the context-free grammar to group the tokens together and form sentences. If it is able to form a sentence from a sequence of tokens delivered to it by the lexical analyser, the sentence is considered to be syntactically correct.

For example, let us consider the following statement written in *C* language:

$a = b;$

The lexical analyser identifies the following tokens for this statement:

a	identifier	token: '*id*'
=	operator	token: '='
b	identifier	token: '*id*'
;	punctuation	token: ';'

These four tokens are then passed on to the parser.

The CFG for an assignment statement (in the limited form, as required here) is written as follows:

$S \rightarrow id \cdot O \cdot id \cdot P$
$O \rightarrow =$
$P \rightarrow ;$

The parser can derive the string '*id O id* ;' using the aforementioned CFG, as follows:

$S \Rightarrow id \cdot \underline{O} \cdot id \cdot P$
$\Rightarrow id \cdot = \cdot id \cdot \underline{P}$
$\Rightarrow id \cdot = \cdot id \cdot ;$

This matches with the sequence of tokens received. Hence, the parser declares the input to be syntactically correct. Thus, syntax checking is done with the help of CFGs for formal (or programming) languages such as *C*.

5.19 BACKUS–NAUR FORM

Backus–Naur form (or Backus-normal Form; abbreviated as BNF) is a notational technique used for context-free grammars. It is mainly used to denote the syntax for the programming languages; it used by most of the tools that consume context-free grammars.

The simple context-free grammar notation that has been used in this chapter so far, and is described in Section 5.4, has limited capabilities. For example, in simple notation, non-terminal symbols are denoted by capital letters. As there are only 26 capital letters, we cannot use this notation for any complex CFG that may require more than 26 non-terminals. In most programming languages, the syntax needs to be defined with much complex CFGs that require more than 1000 grammar rules, and more than 100 non-terminals; and simple CFG notation cannot really suffice this need. Hence, BNF notation is introduced. It is important to remember that by introducing BNF, we are not increasing the power of the CFGs; we are only suggesting a better notation for the CFLs. However, we can safely say that BNF is a more powerful notation for type-2 (and hence, type-3) languages.

In BNF notation, non-terminals can be represented by any word—similar to JAVA identifiers delimited by angular brackets on both the ends, whereas, terminals are represented by words consisting of capital letters. For example, '<ifStatement>', '<program>', and '<postalAddress>' are valid non-terminals in BNF, while 'NEWLINE', 'ID', and 'NUMBER' are valid terminals.

The production rules have a slightly different format compared to simple CFG notation. The productions in BNF are represented as:

<symbol> ::= *expression*

Here, '<symbol>' is any non-terminal, and '*expression*' is any sentential form, that is, any combination of terminals and non-terminals.

For example, consider the following CFG written in BNF format:

<assignmentStatement>	::=	ID <operator> ID <punctuation>
<operator>	::=	'='
<punctuation>	::=	';'

We observe that this CFG is the same as the one discussed in Section 5.15, except for notational changes; and also that any CFG that is expressed in BNF is more readable as well.

SUMMARY

Grammar can be defined as a finite set of formal rules for generating syntactically correct sentences for a particular language for which it is written.

Grammar consists of two types of symbols, namely: Terminals and Non-terminals (also called variables, or auxiliary symbols)

Terminal symbols are those symbols, which are part of the generated sentence. Non-terminal symbols are those symbols, which take part in the formation or generation of the statement, but are not part of the generated statement.

A phrase structure *grammar* is formally denoted by a quadruple of the form:

$G = \{V, T, P, S\}$,

where,

V: Finite set of non-terminals (or variables)

T: Finite set of terminals

S: S is a non-terminal, which is a member of *N*; it is the starting symbol

P: Finite set of productions (or syntactical rules)

Productions have the generic form as, '$\alpha \rightarrow \beta$', where $\alpha, \beta \subset (V \cup T)^*$, and $\alpha \neq \epsilon$. As we know, $V \cap T = \phi$. Note that α and β consist of any number of terminals as well as non-terminals and they are usually termed as *sentential forms*.

If the production rules have the form: '$A \rightarrow \alpha$', where *A* is any non-terminal and α is any sentential form, then such grammar is called *context-free grammar* (CFG).

Context-free language (CFL) is the language generated by a context-free grammar G. This can be described as:

$L(G) = \{w \mid w \subset T^*$, and is derivable from the start symbol S$\}$

In simple CFG notation, non-terminals are usually denoted by capital letters *A, B, C, D*, etc., while terminals are denoted by small case letters *a, b, c*, etc. A string of terminals (i.e., valid words in the

6 Pushdown Stack-memory Machine

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Computational problems that can be solved using a single stack
- Formal model of pushdown stack-memory machine, called pushdown automaton (PDA)
- Comparison of finite automata (FA) and PDA on the basis of their relative computational powers
- Difference between deterministic PDA (DPDA) and non-deterministic PDA (NPDA), and their relative computational powers
- Equivalence between PDA and context-free grammars (CFGs)
- Acceptance of a CFL by an empty stack against acceptance by a final state
- Application of Chomsky normal form (CNF) for controlling stack growth
- Closure properties of context-free languages (CFLs)

6.1 INTRODUCTION

A pushdown stack-memory machine (PDM) is a computational model that is used to solve any problem that has an algorithmic solution and requires a single stack memory (infinite memory, ideally) as storage. Thus, a PDM can remember arbitrarily long input strings, which is not feasible for a finite state machine (FSM) as it does not have any memory at all.

The PDMs can accept or recognize context-free languages (CFLs). A set of all regular languages is a proper subset of CFLs; hence, PDMs can also accept regular languages. Therefore, one can say that a PDM is an FSM having an external stack memory, which makes it a more powerful tool comparatively. A PDM however, is less powerful when compared to a Turing machine (TM). Stacks have limited ability. They can only be used to implement LIFO, whereas a tape is more powerful when compared to a stack in usage. A TM can consume its own output as input using the tape, which is impossible in case of a PDM as the stack is external to the tape.

6.2 ELEMENTS OF A PDM

A PDM is collectively described with the help of the following elements:

1. An input alphabet, (Σ), that is, a finite set of input symbols.
2. An unbounded input tape, bounded only by the input length. The input string is written onto the tape, which is initially assumed to contain blank characters ($\not b$). The left end of the input tape is fixed and the tape is unbounded towards the right end.
3. An alphabet of stack symbols (Γ).
4. A pushdown stack. Initially the stack is empty and is assumed to contain a blank character $\not b$ at the bottom to represent the stack empty position.
5. Start, accept, and reject indicators.
6. Branching state—read.
7. Stack operations, namely push and pop.

The input tape of a PDM is visualized to be divided into cells having one input symbol from Σ written into each—a finite control (head assembly that moves one cell at a time towards the right upon reading a symbol from the tape) and an external stack, which is an external memory. Figure 6.1 shows a pictorial representation of the memory of a PDM.

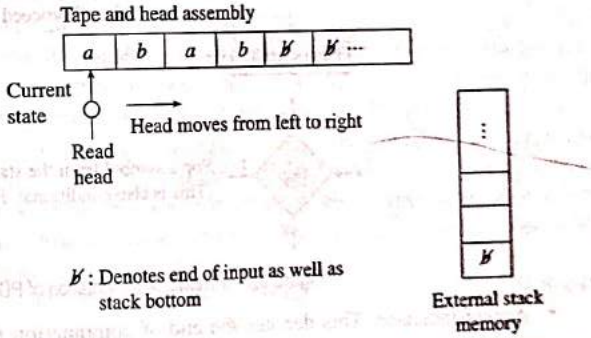


Figure 6.1 Memory of PDM

A PDM can only read from the tape and cannot write onto it; the read head always moves in one direction, that is, from left to right. The stack is external to the reading assembly and acts as an auxiliary memory. Thus, a PDM can be considered as an FSM having an external stack memory.

6.2.1 Pictorial Representation of PDM Elements

Instead of using a state transition diagram that is the usual notation for describing machines, a PDM is represented using flowcharts.

Figure 6.2 depicts the different elements that are used in the flowchart representation of a PDM. The typical elements are as follows:

- *Start* indicator: This is a very common element found in flowcharts. It denotes the beginning of the computation.

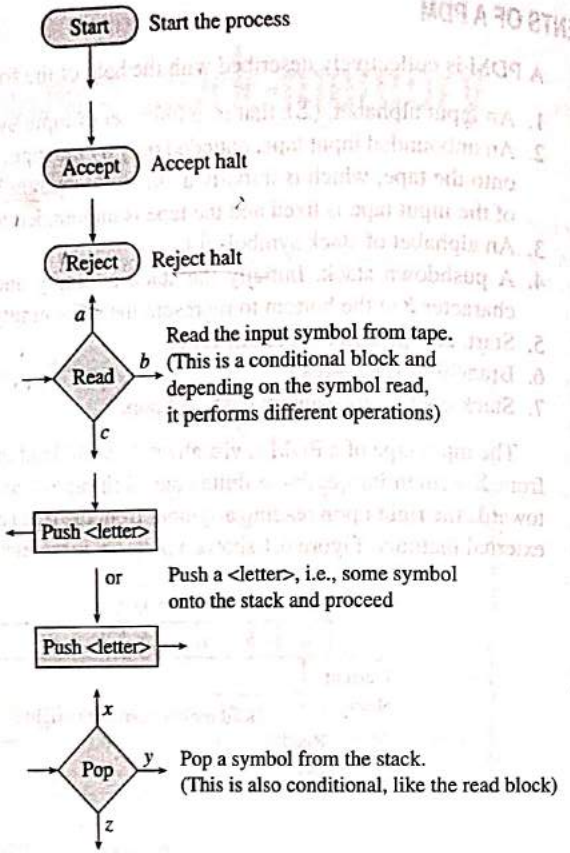


Figure 6.2 Pictorial representation of PDA elements

- *Accept* indicator: This denotes the end of computation that results in accepting the input string. It means that the input string has a valid pattern as defined by the context-free grammar (CFG). We know that a PDM accepts or recognizes CFLs.
- *Reject* indicator: This denotes the end of computation that results in the rejection of the input string. This means that the input is invalid with respect to the CFG rules.
- *Push* action: Push is an operation that is carried over the external stack and is denoted by a rectangular block. One symbol can be pushed at a time.
- *Pop* action: Pop is an operation that results in a decision block. The program flow is specified based on the symbol that is retrieved from the stack.
- *Read* state: Read state is represented as a decision block. The program flow is specified, based on the input symbol that is read.

6.3 PUSHDOWN AUTOMATA

Pushdown automaton (plural—automata; abbreviated as PDA) is the mathematical model (formalism) of the PDM.

Formal Definition

A pushdown automaton M is denoted as

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where,

Q: Finite set of states
Σ: Input alphabet (input strings are composed of symbols from Σ)
Γ: Stack alphabet
q_0: Initial state q_0, which is a member of Q
Z_0: Z_0, which is a member of Γ, is a particular stack symbol called the start symbol. It indicates the bottom of the stack, and is considered as $\not b$ (blank character) in many designs
F: Set of final states, $F \subseteq Q$
δ: $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$ (This defines the transition function δ for a *deterministic PDA* or *DPDA*. The transition function for a non-deterministic PDA or NPDA is different, and is stated later)

The stack is an external storage system. Hence, one can store any symbol onto the stack, though it may not really be the symbol that has been read from the input tape. Moreover, one can push multiple symbols onto the stack, upon reading a single symbol from the input tape. Hence, there is a distinction between input alphabet (Σ) and stack alphabet (Γ). These two sets may overlap or may be completely distinct sets, depending on the problem that we are solving.

Any subset of Q can be marked as the set of final states F, depending on the solution. Upon reading the entire input string, if the machine resides in any of the final states, then the input string is considered as 'accepted' by the machine. Otherwise, the machine resides in any of the non-final states, and the input is considered as 'rejected' by the machine. Usually, in the pictorial representation of the problem solution, rejection is not explicitly shown. The paths that are unspecified in the flowchart are considered rejection paths. We shall discuss more about this in the examples.

A stack is initially assumed to contain the symbol Z_0, that is, the blank character $\not b$. Any PDM solution is based on this assumption.

In case of a DPDA, $\delta(q, a, Z)$ does not contain more than one element for any state q in Q, any symbol Z in Γ on top of the stack, and input symbol a in Σ. Thus, the transition would be of the form

$$\delta(q, a, Z) = (p, \gamma)$$

where, p is the unique next state in Q to which the machine makes the transition (p may or may not be equal to q), and γ is a member of Γ^* (zero or more occurrences of symbols from Γ). The symbol γ can be one of the following:

- Empty, that is, ϵ, if the stack operation performed is pop. This means that Z is popped out of the stack.
- Z, if the stack is not updated—only a state transition is performed.
- $xx\ldots xZ$, if multiple symbols $xx\ldots x$ are pushed onto the stack.
- $xx\ldots x$, if Z is popped out of the stack and multiple symbols $xx\ldots x$ are pushed onto the stack.

Therefore, if the DPDA is in state q, it reads symbol a from the tape cell, and if Z is on top of the stack, it changes the state to p, replaces Z on the stack top by γ (considering the aforementioned four scenarios), and moves the head one cell position to the right on the tape.

For an NPDA, the transition function δ is defined as follows:

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

For example, consider the following transition:

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\},$$

where, $q, p_1, p_2, \ldots, p_m$ are states from Q, a is any input symbol in Σ, Z is any stack symbol in Γ, and all γ_i for $1 \le i \le m$ are members of Γ^*.

The interpretation of this definition is that the NPDA in state q reads input symbol a while Z is the topmost symbol on the stack, possibly makes transition to any state p_i, replaces symbol Z on the stack by any string γ_i (as per the aforementioned four scenarios), and advances the read head one cell position to the right on the input tape. The NPDA, hence, is considered as an unpredictable (non-deterministic or possibilistic) machine as it can perform any of the possible transitions during its execution, though the current machine state, current input symbol being read, and the stack state are the same.

6.4 FINITE AUTOMATA VS PDA

As we have discussed, a PDA can be visualized as an FA having an external stack. Thus, a PDA has infinite storage in the form of a stack, which is missing in the FA. Hence, the FA has lesser computational power compared to the PDA. Furthermore, an FA cannot solve any problem that requires to store intermediate results in the memory for further computation.

As we know, FA are capable of accepting (or recognizing) regular languages (RLs), while PDA can accept CFLs. Since the set of all RLs is a subset of the class of CFLs, we can say that every RL is also a CFL; however, the vice versa may not be true. Hence, PDAs can accept all RLs. This is demonstrated through the following examples.

6.4.1 Examples of PDA Accepting Regular Languages

Example 6.1 Construct a PDA that recognizes the language accepted by the DFA shown in Fig. 6.3.

Solution The DFA given in Fig. 6.3 accepts the regular language represented by the following regular expression:

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

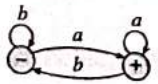


Figure 6.3 Example DFA

We can construct a DPDA equivalent to the given DFA, as shown in Fig. 6.4.

In Fig. 6.4, we see that $Read_1$ state of the DPDA is analogous to the initial state of the DFA and $Read_2$ state is analogous to the final state of the given DFA. Since $Read_2$ state is analogous to the final state, if the input ends in $Read_1$, that is, if we get a blank

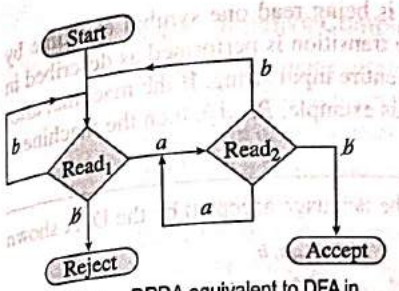


Figure 6.4 DPDA equivalent to DFA in Fig. 6.3

character $\not b$ on the tape in $Read_1$, the machine rejects the input string; else it transits to 'accept' state as shown in the figure.

Please note that the external stack is not required for this example as the language being accepted is a regular language. Hence, the DFA and the DPDA are equivalent machines and the changes are only notational.

In this example, we have demonstrated the PDA as a regular language acceptor.

Let us simulate the working of the PDA for the strings '*bbaaba*' and '*baaabab*'.

1. Simulation for string '*bbaaba*':

Current state	Input symbol	Next state	Head point position
Start	—	$Read_1$	$\underline{b}\,b\,a\,b\,a\,\not b$ ...
$Read_1$	b	$Read_1$	$b\,\underline{b}\,a\,a\,b\,a\,\not b$ ...
$Read_1$	b	$Read_1$	$b\,b\,\underline{a}\,a\,b\,a\,\not b$ ...
$Read_1$	a	$Read_2$	$b\,b\,a\,\underline{a}\,b\,a\,\not b$ ...
$Read_2$	a	$Read_2$	$b\,b\,a\,a\,\underline{b}\,a\,\not b$ ...
$Read_2$	b	$Read_1$	$b\,b\,a\,a\,b\,\underline{a}\,\not b$ ...
$Read_1$	a	$Read_2$	$b\,b\,a\,a\,b\,a\,\underline{\not b}$ ...
$Read_2$	$\not b$	Accept	$b\,b\,a\,a\,b\,a\,\not b\,\underline{\not b}$ ...

Thus, the string '*bbaaba*' is accepted by the PDA.

2. Simulation for string '*baaabab*':

Current state	Input symbol	Next state	Head point position
Start	—	$Read_1$	$\underline{b}\,a\,a\,a\,b\,a\,b\,\not b$ ...
$Read_1$	b	$Read_1$	$b\,\underline{a}\,a\,a\,b\,a\,b\,\not b$ ...
$Read_1$	a	$Read_2$	$b\,a\,\underline{a}\,a\,b\,a\,b\,\not b$ ...
$Read_2$	a	$Read_2$	$b\,a\,a\,\underline{a}\,b\,a\,b\,\not b$ ...
$Read_2$	a	$Read_2$	$b\,a\,a\,a\,\underline{b}\,a\,b\,\not b$ ...
$Read_2$	b	$Read_1$	$b\,a\,a\,a\,b\,\underline{a}\,b\,\not b$ ...
$Read_1$	a	$Read_2$	$b\,a\,a\,a\,b\,a\,\underline{b}\,\not b$ ...
$Read_2$	b	$Read_1$	$b\,a\,a\,a\,b\,a\,b\,\underline{\not b}$ ...
$Read_1$	$\not b$	Reject	$b\,a\,a\,a\,b\,a\,b\,\not b\,\underline{\not b}$ ...

Thus, the string '*baaabab*' is rejected by the PDA

This simulation depicts how the input string is being read one symbol at a time by the machine. Upon reading every symbol, a state transition is performed as described in Fig. 6.4. The machine stops after consuming the entire input string. If the machine, after reading the entire input, is in the final state (in this example, $Read_2$), then the machine is said to accept the string; else it rejects the string.

Example 6.2 Construct a PDA that recognizes the language accepted by the DFA shown in Fig. 6.5.

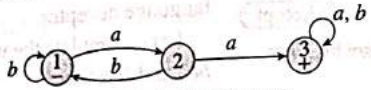


Figure 6.5 Example DFA

Solution The equivalent DPDA can be constructed as shown in Fig. 6.6.

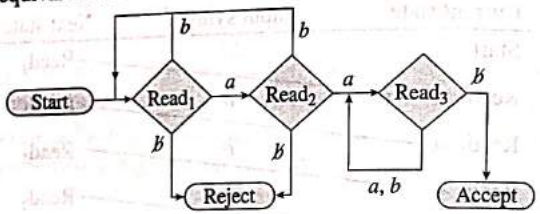


Figure 6.6 DPDA equivalent to DFA in Fig. 6.5

We see that the $Read_1$ state in the PDA is analogous to state 1 in the given DFA. Similarly, $Read_2$ is analogous to state 2, and $Read_3$ is analogous to state 3 in the given DFA. Hence, $Read_1$ and $Read_2$ are non-final states. If the PDA reads a blank character $\not b$ indicating the end of the input string while in these states, the machine rejects the input string. Upon reading the entire input string, if the PDA reaches the final state, that is, $Read_3$, then the input string is accepted by the PDA.

Let us simulate the working of the PDA for the input given as '*abaab*'. The acceptance of the input can be shown as follows:

Current state	Input symbol	Next state	Head point position
Start	—	$Read_1$	$\underset{\uparrow}{a}\,b\,a\,a\,b\,\not b\ldots$
$Read_1$	a	$Read_2$	$a\,\underset{\uparrow}{b}\,a\,a\,b\,\not b\ldots$
$Read_2$	b	$Read_1$	$a\,b\,\underset{\uparrow}{a}\,a\,b\,\not b\ldots$
$Read_1$	a	$Read_2$	$a\,b\,a\,\underset{\uparrow}{a}\,b\,\not b\ldots$
$Read_2$	a	$Read_3$	$a\,b\,a\,a\,\underset{\uparrow}{b}\,\not b\ldots$
$Read_3$	b	$Read_3$	$a\,b\,a\,a\,b\,\not b\,\underset{\uparrow}{\not b}\ldots$
$Read_3$	$\not b$	Accept	$a\,b\,a\,a\,b\,\not b\,\not b\ldots$

6.4.2 Relative Computational Powers of PDA and FA

We have seen in the previous subsection that PDA can accept regular languages just as FA. However, a PDA is much more powerful when compared to the FA, as it has infinite memory in the form of an external stack that is absent in the case of FA. In other words, we can say that FA is a special case of PDA—it is a PDA without an external stack.

For every FA, we can construct an equivalent PDA that accepts the same regular language. Such a PDA does not require a stack and its operations such as push and pop.

We have already discussed earlier that PDA can accept CFLs that are the superset of the class of RLs. In the following section, we shall look at some examples of PDA that accept CFLs.

6.5 PDA ACCEPTING CFLs

We have seen that a PDA accepts not only RLs, but also CFLs. Further, we also know that building a DPDA or an NPDA that accepts RLs does require an external stack. However, this is not possible while designing a PDA solution for the CFLs.

The following examples will illustrate this further.

Example 6.3 Construct a PDA that accepts the following language:

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

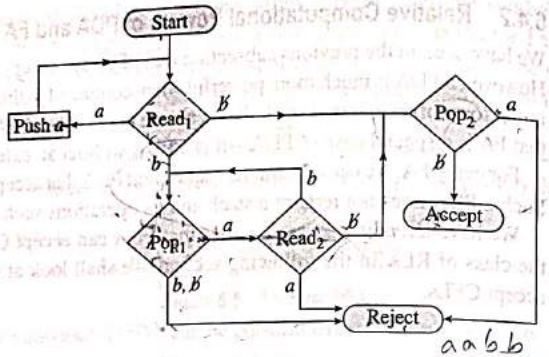
Solution As per the definition, every string in the language contains n number of a's followed by the same number of b's; also, it contains an empty string ϵ, whenever $n = 0$.

Algorithm

1. Keep on pushing all a's onto the stack till the machine reads the first b.
2. Each time it reads b, it pops one a from the stack.
3. If the number of a's are equal to the number of b's, then at the end of the input string, that is, when blank character $\not b$ is read from the tape, the top of the stack should also contain the blank character $\not b$, indicating that the stack is empty. The string is accepted only in this case; else it is rejected.

The required DPDA is constructed as shown in Fig. 6.7.

We see from Fig. 6.7 that $Read_1$ state pushes all a's onto the stack. On reading the first b, it performs the pop operation; $Read_2$ state checks whether or not the same number of b's are following the a's, by repeatedly popping the a's from the stack for every matching b that is read.



$Read_1$: Keep on pushing a's till you get the first b
Pop_1: Pop an a when you read one b
$Read_2$: When you get b while reading b's go to 'Pop_2'
Pop_1: Input is finished, so check whether the stack is empty or not. If empty go to 'Accept'

Figure 6.7 PDA that accepts $\{a^n b^n \mid n \geq 0\}$

Simulation

1. Let us simulate the acceptance of the string '*aabb*':

Current state	Stack contents	Tape and head
Start	▷	a a b b ▷ ▷ ...
↓		↑
$Read_1$	▷	a a b b ▷ ▷ ...
a ↓		↑
Push a	▷ a	a a b b ▷ ▷ ...
↓		↑
$Read_1$	▷ a	a a b b ▷ ▷ ...
a ↓		↑
Push a	▷ a a	a a b b ▷ ▷ ...
↓		↑
$Read_1$	▷ a a	a a b b ▷ ▷ ...
b ↓		↑
Pop_1	▷ a	a a b b ▷ ▷ ...
a ↓		↑
$Read_2$	▷ a	a a b b ▷ ▷ ...
b ↓		↑
Pop_1	▷	a a b b ▷ ▷ ...
a ↓		↑
$Read_2$	▷	a a b b ▷ ▷ ...
▷ ↓		↑
Pop_2	—	a a b b ▷ ▷ ...
▷ ↓		↑
Accept		

The arrows on the left-hand side indicate transitions from one state to another on reading the same symbol. The operations 'push' and 'pop' are performed while processing the input.

In the beginning, the stack is assumed to contain the blank character ▷ to indicate the bottom of the stack. In the aforementioned simulation, the first two characters in the input string are pushed onto the stack as both are a's. Then, for every b read, an a is popped out of the stack. Once the blank character ▷ is read, indicating the end of the input, 'pop' operation is performed to check whether the stack is empty as well. If the stack is empty, that is, if pop operation returns the bottommost element as ▷, then the input string is accepted by the PDA. Thus, the input string '*aabb*' is found to match the expected pattern, $a^n b^n$. This is also referred to as *acceptance by empty stack* (refer to Section 6.5.2).

2. Let us now simulate the rejection of '*abbba*':

Current state	Stack contents	Tape and head
Start	▷	a b b b a ▷ ▷ ...
↓		↑
$Read_1$	▷	a b b b a ▷ ▷ ...
a ↓		↑
Push a	▷ a	a b b b a ▷ ▷ ...
↓		↑
$Read_1$	▷ a	a b b b a ▷ ▷ ...
b ↓		↑
Pop_1	▷	a b b b a ▷ ▷ ...
a ↓		↑
$Read_2$	▷	a b b b a ▷ ▷ ...
b ↓		↑
Pop_1	—	a b b b a ▷ ▷ ...
▷ ↓		↑
Reject		

In this simulation, the first a gets pushed onto the stack. When the second input symbol b is read, the topmost stack symbol a is popped to match the read symbol b. The stack becomes empty after the first pop. The third symbol b is read but the stack is found to be empty—it does not contain any more a's to match with the b that is read. Hence, the input string is rejected.

Notes:
1. The given language $L = \{a^n b^n \mid n \geq 0\}$, is not a regular language; therefore, it cannot be recognized by an FA. It is actually a CFL and the CFG for the same can be written as follows:

$S \rightarrow a S b \mid c$

2. We observe that the PDA in Fig. 6.7 is a DPDA because from every state there is a unique transition for a symbol that is read.
3. We need not restrict ourselves to using the same alphabet for the input strings as well as the stack. Since the stack is an external memory component, the stack alphabet (Γ) is considered to be distinct from the input alphabet (Σ). In this example, it is possible to read an a from the tape and push x onto the stack. In this case, the number of x's represents the count of the number of a's present, as shown in Fig. 6.8.

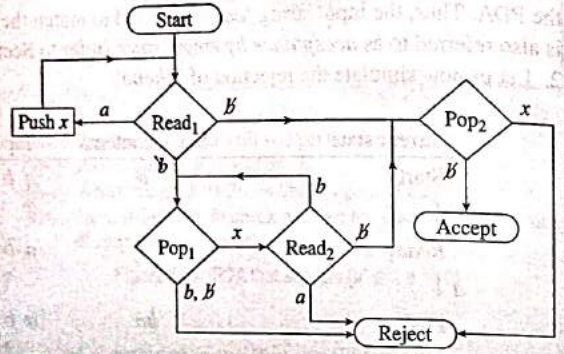


Figure 6.8 DPDA using distinct sets of input and stack alphabets

The DPDA in Fig. 6.7 can also be specified using the following set of equations, which are based on the formal notations discussed in Section 6.3.

The following two equations denote the pushing of symbol a onto the stack if the symbol read is a; the stack is either empty or contains a as its topmost symbol. These two equations collectively signify the process of pushing all the a's onto the stack (till the first b is read). Refer to program block 1 in Fig. 6.9.

$$\delta\,(Read_1, a, Z) = (Read_1, aZ)$$
$$\delta\,(Read_1, a, a) = (Read_1, aa)$$

The following two equations denote the pop operation of a out of the stack for every b that is read. This is indicated by the stack state, which changes from a to ϵ (refer to program block 2 in Fig. 6.9). The two transitions indicate moving from $Read_1$ to $Read_2$ for the first b that is read and $Read_2$ to itself for the subsequent b's that are read.

$$\delta\,(Read_1, b, a) = (Read_2, \epsilon)$$
$$\delta\,(Read_2, b, a) = (Read_2, \epsilon)$$

The following two equations denote the two program paths that lead to the acceptance of the input string, if it matches the required pattern, $a^n b^n$. The first equation represents the case $n = 0$, that is, when the input string is empty (ϵ). The second equation represents the case $n > 0$.

$$\delta\,(Read_1, Z, Z) = \text{Accept}$$
$$\delta\,(Read_2, Z, Z) = \text{Accept}$$

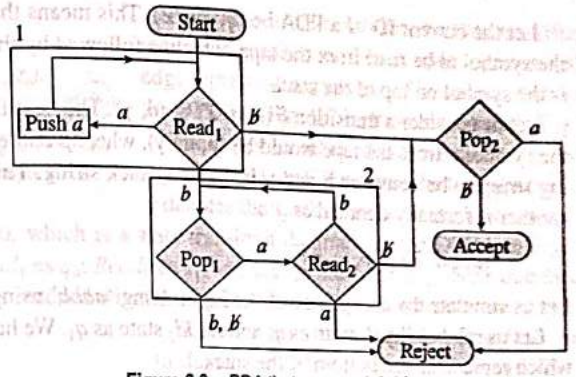


Figure 6.9 PDA that accepts $\{a^n b^n \mid n \geq 0\}$

In principle, the aforementioned six equations are sufficient to denote the DPDA that accepts the given CFL. The following six are additional equations that denote rejections and may not be specified—since anyway, what is not specified is always rejected.

The following two rejection equations represent the presence of additional number of a's.

$$\delta\,(Read_1, Z, a) = \text{Reject}$$
$$\delta\,(Read_2, Z, a) = \text{Reject}$$

The following two rejections represent that the number of b's is more than the number of a's.

$$\delta\,(Read_1, b, Z) = \text{Reject}$$
$$\delta\,(Read_2, b, Z) = \text{Reject}$$

These two equations are reached when the DPDA reads a while in $Read_2$ state, no matter what is on the top of the stack. This is something that is against the expected pattern. These equations represent the case where there are some a's even after all the b's have been read. Hence, they are considered as rejection equations.

$$\delta\,(Read_2, a, Z) = \text{Reject}$$
$$\delta\,(Read_2, a, a) = \text{Reject}$$

6.5.1 Instantaneous Description of PDA

Instantaneous description (ID) of a PDA, as the name suggests, is its description at a given instance, while computing a given input string. It is described with the help of a triple (q, w, z), where q denotes the current state of the machine; w denotes the input string remaining to be read from the tape; and z denotes the topmost stack symbol.

While consuming any given input string (symbol by symbol), the PDA moves from one ID to another. This process continues as long as the input lasts. This means that the simulations we have seen in the examples can also be described more formally with the help of a series of IDs.

Let the current ID of a PDA be (q, aw, Z). This means that q is the current state; a is the symbol to be read from the tape, which is followed by the remaining string w; and Z is the symbol on top of the stack.

Let us consider a transition $\delta(q, a, Z) = (p, \gamma)$. The next ID for the PDA after reading the symbol a from the tape would be (p, w, γ), where p is the next state; w is the remaining string to be consumed; and γ is the new stack string. This transition from one ID to another is formally denoted as

$$(q, aw, Z) \vdash (p, w, \gamma)$$

Let us simulate the acceptance of the input string '*aabb*' using the series of IDs.

Let us relabel $Read_1$ state as q_0 and $Read_2$ state as q_1. We have already labelled Z_0 as $\not b$, which represents the bottom of the stack.

$$\begin{aligned}(q_0, aabb\not b, \not b) &\vdash (q_0, abb\not b, a\not b)\\ &\vdash (q_0, bb\not b, aa\not b)\\ &\vdash (q_1, b\not b, a\not b)\\ &\vdash (q_1, \not b, \not b)\\ &\vdash (\text{Accept}, \epsilon, \epsilon)\end{aligned}$$

6.5.2 Acceptance of CFL by Empty Stack

The acceptance of an input string '*aabb*' as depicted in the previous section is an example of acceptance by an empty stack. Observe that neither $Read_1$ nor $Read_2$ is designated as a final state in the DPDA shown in Fig. 6.7. In this chapter, we have implemented acceptance by empty stack for all the solved examples, as it is the most common way of implementation. Readers may attempt to build the acceptance by final state for these examples, if interested.

A string w is said to be accepted by a PDA, if

$$(q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$$

where, p is any state in Q, and '$\vdash^*$' denotes an ID after reading the entire input string. '$\vdash^*$' denotes multiple steps in reading the string, while '$\vdash$' denotes one step at a time (refer to Fig. 6.10).

6.5.3 Acceptance of CFL by Final State

Acceptance by final state is not very commonly implemented for PDA and is not very different from acceptance by empty stack.

A string w is said to be accepted by a PDA if:

$$(q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$$

where, p is a member of F, which is a designated set of final states (refer to Section 6.3), and γ is any stack string. This means that upon reading the entire string w, if the PDA transits to a final state, then the string w is said to be accepted by the PDA (refer to Fig. 6.11).

6.5.4 State Transition Diagram for PDA

We can also represent PDAs using the state transition diagram, just as we did in the case of FA and Turing machines. However, there is a slight change in the notation, based on whether a given PDA accepts a CFL with an empty stack or a final state.

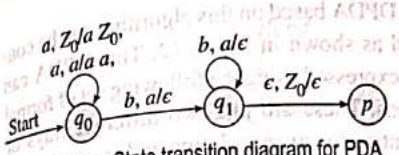


Figure 6.10 State transition diagram for PDA that accepts $\{a^n b^n \mid n \geq 0\}$ by empty stack

As per the usual notations, the state transition diagram is a directed graph, where each node represents a state and an edge represents a transition. The state transition diagram for the DPDA in Fig. 6.7 is drawn as shown in Fig. 6.10.

The edges are labelled in the form '$a, Z/\gamma$' where, a is the input symbol read; Z denotes the topmost stack symbol; and γ denotes the stack string after transition is complete.

In Fig. 6.10, which is a state transition diagram for the DPDA in Fig. 6.7, we have relabelled $Read_1$ as q_0; $Read_2$ as q_1; and accept indicator as p. Note that though p is not a final state, it is reached when the input ends and the stack becomes empty.

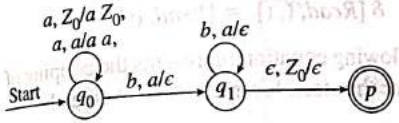


Figure 6.11 State transition diagram for PDA that accepts $\{a^n b^n \mid n \geq 0\}$ by final state

The state transition diagram in Fig. 6.10 can be modified to depict the DPDA accepting the same CFL by the final state, as shown in Fig. 6.11.

We observe that the only change in Fig. 6.11 is that p is now marked with double circles to indicate that it is a final state; and the stack string Z_0 remains unchanged while approaching the state p.

Example 6.4 Design a PDA that checks for well-formed parentheses.

Solution As we know, a string of parentheses that is well-formed should start with the opening bracket '(' and must end with the closing bracket ')'. This is a CFL and can be described with the help of the following grammar:

$$S \rightarrow (\,S\,)\,S \mid \epsilon$$

The language can be expressed as

$$L = \{\epsilon, (\,), (\,(\,)\,), (\,)\,(\,), (\,)\,(\,(\,)\,), (\,(\,)\,)\,(\,), \ldots\}$$

Algorithm

The problem is similar to that in the previous example, in which we have seen that the string should begin with a, and is matched with the subsequent b's; in this example, all '('s are pushed onto the stack to be matched later with the subsequent ')'s. The only difference is that the machine can read opening parenthesis '('even after reading the closing parenthesis ')', rather than all '('s being read first followed by all ')'s. In other words, the language is not of the form $(^n\,)^n$; for example, there may be strings of the form '( ) ( )' or '( ) (( ))' as well. Due to this, the algorithm and the PDA will be a little different, as follows:

1. If you read symbol '(', push '(' onto the stack.
2. On reading every ')' pop one '(' from the stack.
3. Repeat the aforementioned two steps till you finish with all the parentheses in the string.
4. When the input ends, that is, on reading $\not b$, the top of the stack should also contain $\not b$; this indicates that the string consists of well-formed parentheses.

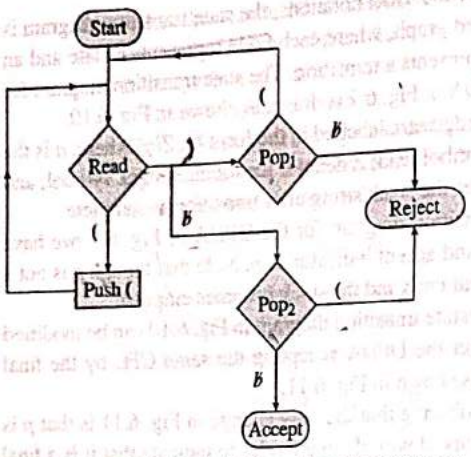


Figure 6.12 DPDA that checks for well-formed parentheses

The DPDA based on this algorithm can be constructed as shown in Fig. 6.12. This DPDA can also be expressed using the following set of formal equations. These are just two different ways of representing the given solution and are equivalent to each other.

When the symbol '(' is read, push it onto the stack. This is expressed using the following two equations:

$$\delta\,[Read, (, \not b] = [Read, (\not b]$$
$$\delta\,[Read, (, (] = [Read, ((]$$

The following equation represents the popping of '(' out of the stack for every ')' that is read:

$$\delta\,[Read,), (] = [Read, \epsilon]$$

The following equation is the only path to acceptance. When the input ends, the stack must be empty as well.

$$\delta\,(Read, \not b, \not b) = \text{Accept}$$

Rejection paths are expressed with the help of the following two equations:

$\delta\,[Read,), \not b]$ = Reject ... number of ')'s is more
$\delta\,[Read, \not b, (]$ = Reject ... number of '('s is more

Simulation

Let us simulate the acceptance of the string '(( ) ( ))' by an empty stack.

Current state	Stack contents	Tape and head
Start	$\not b$	(( ) ( )) $\not b \not b$ ...
↓		↑
Read	$\not b$	(( ) ( )) $\not b \not b$ ...
(↓		↑
Push (	$\not b$ (	(( ) ( )) $\not b \not b$ ...
↓		↑
Read	$\not b$ (	(( ) ( )) $\not b \not b$ ...
(↓		↑
Push (	$\not b$ ((	(( ) ( )) $\not b \not b$ ...
↓		↑
Read	$\not b$ ((	(( ) ( )) $\not b \not b$ ...
)↓		↑
Pop_1	$\not b$ (	(( ) ( )) $\not b \not b$ ...
(↓		↑

Read	$\not b$ (	(( ) ( )) $\not b \not b$ ...
(↓		↑
Push (	$\not b$ ((	(( ) ( )) $\not b \not b$ ...
↓		↑
Read	$\not b$ ((	(( ) ( )) $\not b \not b$ ...
)↓		↑
Pop_1	$\not b$ (	(( ) ( )) $\not b \not b$ ...
(↓		↑
Read	$\not b$ (	(( ) ( )) $\not b \not b$ ...
)↓		↑
Pop_1	$\not b$	(( ) ( )) $\not b \not b$ ...
(↓		↑
Read	$\not b$	(( ) ( )) $\not b \not b$ ...
$\not b$↓		↑
Pop_2	—	(( ) ( )) $\not b \not b$ ...
$\not b$↓		↑
Accept		

Example 6.5 Construct a PDA that accepts the following language:

$$L = \{X, aXa, bXb, aaXaa, abXba, baXab, bbXbb, aaaXaaa, \ldots\}$$

Solution We see that the language consists of all odd length palindrome strings over $\Sigma = \{a, b\}$ having X as their middle symbol. This is a CFL with the grammar expressed as

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid X$$

Algorithm

1. Push all symbols onto the stack till X is read by the machine.
2. Beyond X, for every input symbol that is read, pop a symbol from the stack and check for the equality with the recently-read symbol.
3. If the symbol on the stack matches with the symbol read, then continue step 2; else, reject the string.
4. If the end of the input is reached, that is, the symbol read is $\not b$, pop the symbol onto the top of the stack. If the symbol popped is also $\not b$ (indicates stack empty), then accept the input string; else, reject the string.

The DPDA based on this algorithm is constructed as shown in Fig. 6.13. The set of equations equivalent to the DPDA are given here:

The following six equations are responsible for pushing all a's and b's till the symbol X has been read.

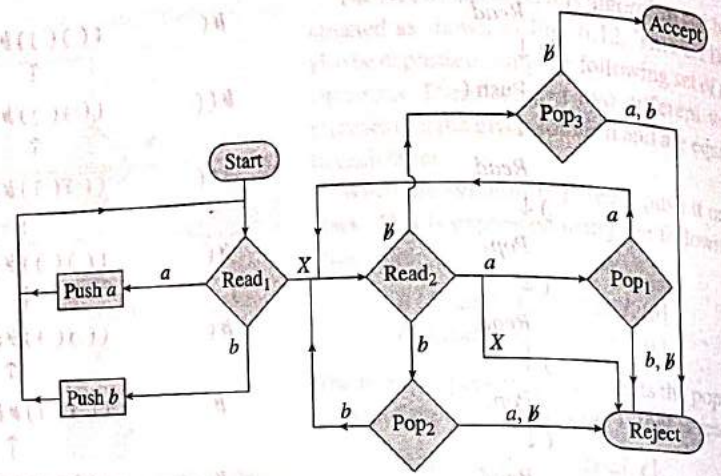


Figure 6.13 PDA that accepts language *L*

$\delta\ (Read_1, a, \not\vdash) = (Read_1, a\not\vdash)$
$\delta\ (Read_1, a, a) = (Read_1, aa)$
$\delta\ (Read_1, a, b) = (Read_1, ab)$

$\delta\ (Read_1, b, \not\vdash) = (Read_1, b\not\vdash)$
$\delta\ (Read_1, b, b) = (Read_1, bb)$
$\delta\ (Read_1, b, a) = (Read_1, ba)$

The following three equations represent the case when the middle symbol *X* is read. A transition is made from $Read_1$ state to the next state, $Read_2$, and the stack remains unchanged.

$\delta\ (Read_1, X, a) = (Read_2, a)$
$\delta\ (Read_1, X, b) = (Read_2, b)$
$\delta\ (Read_1, X, \not\vdash) = (Read_2, \not\vdash)$

Once the symbol *X* is read, the steps are followed to match the remaining half of the input string to the first half, which is already present on the stack. This is expressed with the help of the following three equations:

If the symbol read is the same as the symbol on top of the stack, then it gets popped off.

$\delta\ (Read_2, b, b) = (Read_2, \varepsilon)$
$\delta\ (Read_2, a, a) = (Read_2, \varepsilon)$
$\delta\ (Read_2, \not\vdash, \not\vdash) = \text{Accept}$

The invalid input strings are rejected by the DPDA. This is expressed through the following equations:

$\delta\ (Read_2, a, \not\vdash) = \text{Reject}$
$\delta\ (Read_2, a, b) = \text{Reject}$

$\delta\ (Read_2, b, \not\vdash) = \text{Reject}$
$\delta\ (Read_2, b, a) = \text{Reject}$

$\delta\ (Read_2, \not\vdash, b) = \text{Reject}$
$\delta\ (Read_2, \not\vdash, a) = \text{Reject}$
$\delta\ (Read_2, X, b) = \text{Reject}$
$\delta\ (Read_2, X, a) = \text{Reject}$
$\delta\ (Read_2, X, \not\vdash) = \text{Reject}$

Simulation

Let us simulate the acceptance of the string '*abXba*'.

Current state	Stack contents	Tape and head
Start	⊬	a b X b a ⊬ ⊬ ...
↓		↑
$Read_1$	⊬	a b X b a ⊬ ⊬ ...
a ↓		↑
Push *a*	⊬ a	a b X b a ⊬ ⊬ ...
↓		↑
$Read_1$	⊬ a	a b X b a ⊬ ⊬ ...
b ↓		↑
Push *b*	⊬ a b	a b X b a ⊬ ⊬ ...
↓		↑
$Read_1$	⊬ a b	a b X b a ⊬ ⊬ ...
X ↓		↑
$Read_2$	⊬ a b	a b X b a ⊬ ⊬ ...
b ↓		↑
Pop_2	⊬ a	a b X b a ⊬ ⊬ ...
b ↓		↑
$Read_2$	⊬ a	a b X b a ⊬ ⊬ ...
a ↓		↑
Pop_1	⊬	a b X b a ⊬ ⊬ ...
a ↓		↑
$Read_2$	⊬	a b X b a ⊬ ⊬ ...
⊬ ↓		↑
Pop_3	—	a b X b a ⊬ ⊬ ...
⊬ ↓		↑
Accept		

6.6 DPDA VS NPDA

Consider a CFL that consists of all possible palindrome strings over $\Sigma = \{a, b\}$. Such a CFL can be expressed with the help of the following CFG:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon \qquad (G)$$

Odd length palindrome strings over $\Sigma = \{a, b\}$ can be generated using the following grammar rules:

$$S \rightarrow aSa \mid bSb \mid a \mid b \quad \text{(G1)}$$

This is obtained by replacing the middle symbol X in the CFG we have seen in Example 6.5 by a or b.

Similarly, all even length palindrome strings over $\Sigma = \{a, b\}$ can be generated using the following grammar rules:

$$S \rightarrow aSa \mid bSb \mid \epsilon \quad \text{(G2)}$$

Here, the middle symbol X in Example 6.5 is replaced by ϵ.

We see that the grammar G is a combination of the rules for odd as well as even length palindrome strings over $\Sigma = \{a, b\}$.

Using this grammar, and with the help of the following example, let us attempt to compare DPDA and NPDA.

Example 6.6 Construct a PDA, which accepts the language denoted by the following grammar.

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon \quad \text{(G)}$$

Solution The given language consists of all possible palindrome strings over $\Sigma = \{a, b\}$.

The required PDA can be obtained by making the following changes in the DPDA shown in Fig. 6.13.

In Fig. 6.13, the middle symbol X separates the first half of the string that is pushed onto the stack, from the second half that is used for comparison. If we wish to accept all odd length palindrome strings over $\Sigma = \{a, b\}$, this middle symbol X must be replaced by either a or b, as shown in grammar $G1$. Hence, the transition from $Read_1$ state to $Read_2$ state is changed either to symbol a or b (refer to Fig. 6.14).

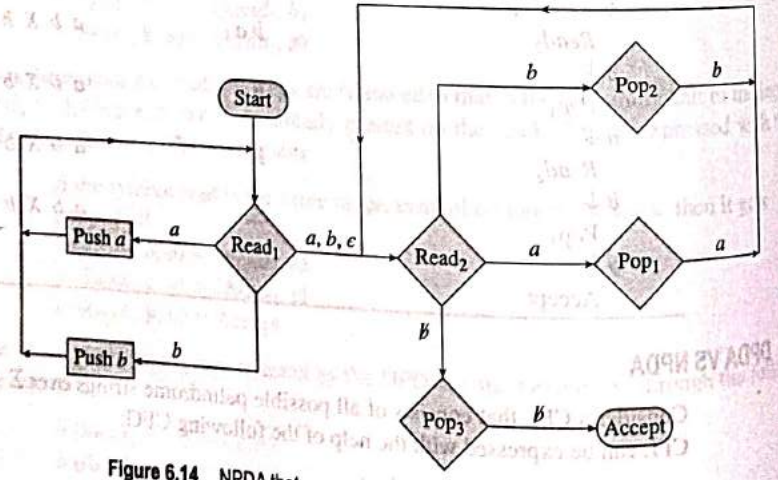


Figure 6.14 NPDA that accepts all palindrome strings over $\Sigma = \{a, b\}$

Similarly, as per grammar $G2$, the middle symbol X becomes ϵ for even length palindrome strings; hence, the transition from $Read_1$ state to $Read_2$ state. Therefore, in the required PDA (refer to Fig. 6.14), we change the symbol X to ϵ.

Thus, there are now three possible transitions from $Read_1$ to $Read_2$ depending on the input symbol read—a, b, or ϵ.

Please note that the PDA that we have obtained in Fig. 6.14 is non-deterministic. Further, the rejection paths are not shown in the figure for further simplification. The NPDA in Fig. 6.14 can also be expressed using the equations mentioned here.

The following set of equations denotes the operation of pushing a's or b's onto the stack, till the mid-point is reached:

$$\left.\begin{aligned} \delta(Read_1, a, ß) &= (Read_1, aß) \\ \delta(Read_1, a, a) &= (Read_1, aa) \\ \delta(Read_1, a, b) &= (Read_1, ab) \\ \delta(Read_1, b, ß) &= (Read_1, bß) \\ \delta(Read_1, b, b) &= (Read_1, bb) \\ \delta(Read_1, b, a) &= (Read_1, ba) \end{aligned}\right\} \quad \text{(6.1)}$$

The following set of six equations ignores the middle symbol. The transition from $Read_1$ state to $Read_2$ state is made without changing the stack string.

$$\left.\begin{aligned} \delta(Read_1, a, a) &= (Read_2, a) \\ \delta(Read_1, a, b) &= (Read_2, b) \\ \delta(Read_1, b, a) &= (Read_2, a) \\ \delta(Read_1, b, b) &= (Read_2, b) \\ \delta(Read_1, \epsilon, a) &= (Read_2, a) \\ \delta(Read_1, \epsilon, b) &= (Read_2, b) \end{aligned}\right\} \quad \text{(6.2)}$$

The following three equations denote the process of matching the second half of the input string with the stacked symbols in order to check whether or not the string is a palindrome, and accept the string in case it is a palindrome.

$$\begin{aligned} \delta(Read_2, b, b) &= (Read_2, \epsilon) \\ \delta(Read_2, a, a) &= (Read_2, \epsilon) \\ \delta(Read_2, ß, ß) &= \text{Accept} \end{aligned}$$

We notice that Eq. sets (6.1) and (6.2) have some conflicting entries, which are mentioned here:

$$\begin{aligned} &\delta(Read_1, a, a) = (Read_1, aa) \text{ from Eqs (6.1)} \\ &\delta(Read_1, a, a) = (Read_2, a) \text{ from Eqs (6.2)} \end{aligned}$$

This is because the PDA is non-deterministic.

We have already discussed in Section 6.3, the differences between the transition function δ in a DPDA and an NPDA.

The conflicting entries can be explained using the transition function δ of the NPDA, which is defined as

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

The aforementioned example of conflicting entries can formally be written as

$$\delta(Read_1, a, a) = \{(Read_1, aa), (Read_2, a)\}$$

Thus, the NPDA in Fig. 6.14 is capable of accepting the language of all palindrome strings over $= \{a, b\}$. We see that it is however not possible to build a DPDA for the same CFL, as we cannot really decide when the mid-point is reached, in order to match the two halves of the input string using a single stack.

Simulation

Let us see a possible simulation for the acceptance of the string '*aba*'.

Current state	Stack contents	Tape and head
Start	♭	a b a ♭ ♭ ... ↑
↓		
$Read_1$	♭	a b a ♭ ♭ ... ↑
a ↓		
Push *a*	♭ a	a b a ♭ ♭ ... ↑
↓		
$Read_1$	♭ a	a b a ♭ ♭ ... ↑
b ↓		
$Read_2$	♭ a	a b a ♭ ♭ ... ↑
a ↓		
Pop_1	♭	a b a ♭ ♭ ... ↑
a ↓		
$Read_2$	♭	a b a ♭ ♭ ... ↑
♭ ↓		
Pop_3	—	a b a ♭ ♭ ... ↑
♭ ↓		
Accept		

Since NPDA is a possibilistic machine, the aforementioned simulation is one possible machine transition that leads to acceptance. This is interpreted considering the fact that *b* is a mid-point of '*aba*'.

Note: No DPDA exists, which can accept the CFL expressed by the given grammar *G*. Thus, we can say that NPDA is more powerful than DPDA; NPDA accepts many more CFLs for which no DPDA can be built. In other words, the class of CFLs accepted by DPDA is a proper subset of the CFLs accepted by NPDA.

6.6.1 Relative Powers of DPDA/NPDA and NFA/DFA

From the aforementioned discussion, we can conclude that a language accepted by an NPDA may not be accepted by a DPDA. As a result, for every NPDA, there may not exist an equivalent DPDA. This is comparatively different from the NFA/DFA scenario.

Figure 6.15 Venn diagram explaining relative powers of DPDA/NPDA and DFA/NFA

Refer to the Venn diagram shown in Fig. 6.15 that depicts the relative powers of DFA, NFA, DPDA, and NPDA. Since, for every NFA there exists an equivalent DFA accepting the same regular language, we can say that DFA and NFA have equal powers. However, this does not hold true for PDAs—NPDA and DPDA have different capabilities. The NPDA can accept any CFL, while DPDA is a special case of NPDA that accepts only a subset of the CFLs accepted by the NPDA. Thus, DPDA is less powerful than NPDA.

6.7 EQUIVALENCE OF CFG AND PDA

So far in the examples, we have not directly used the CFG for building the PDA. Instead, we have used the language properties and a simple algorithm that uses a single stack to achieve the acceptance.

In this section, we shall discuss the algorithm to build an NPDA directly, based on the given CFG. This algorithm treats the NPDA mainly as a syntax analyser (or parser) that validates the input string based on the given grammar or CFG. Precisely, the NPDA is considered as a top-down parser, which uses leftmost derivation to generate the input string to be validated. The NPDA thus constructed uses the CFG to apply one grammar rule at a time and generates the input string, symbol by symbol. Each generated symbol is then matched with the input symbol read. If all the input symbols are matched, then the NPDA considers the input string as valid.

Example 6.7 Construct a PDA that accepts the language generated by the following CFG:

$$S \rightarrow SS \mid (S) \mid ( )$$

Solution The required NPDA is constructed as shown in Fig. 6.16.

The algorithm is based on the leftmost derivation process as mentioned earlier. As we are aware, the leftmost derivation process involves replacing the leftmost non-terminal at every step by the right-hand side of the grammar rule. For example, if the provided grammar rule is of the form '$A \rightarrow \alpha$', then A is replaced by α. Further, whenever a terminal symbol is generated on the stack, an input symbol is read to match it.

Algorithm

1. Start with pushing the start symbol onto the stack.
2. After popping the start symbol, push the right-hand side of the production rule that is applicable in the leftmost derivation of the string onto the stack, but in the reverse way. Please note that the right-hand side of the production rule is pushed in the reverse order so as to receive the leftmost non-terminal symbol onto the top of the stack. In this way, the leftmost non-terminal symbol can be popped out of the stack and expanded further to achieve the leftmost derivation.
3. Whenever a terminal symbol is generated after popping from the stack, read an input symbol from the tape and check whether it matches the currently popped symbol.

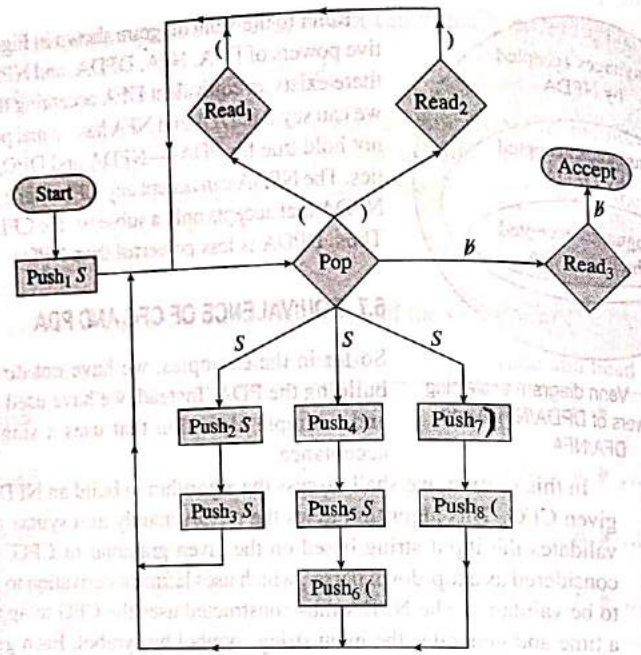


Figure 6.16 NPDA that accepts the language generated by $S \rightarrow SS \mid (S) \mid ( )$

4. Continue the process till the entire input string is consumed.
5. When the stack is empty, that is, the popped symbol is $\not b$, read the input. If the symbol read from the input stream is $\not b$ (indicates end of input), then declare acceptance. This also denotes that the input string is entirely derived using the NPDA and that the input string is syntactically valid.

Simulation

Let us simulate the working of the NPDA in Fig. 6.16 for the string '(( ) ( ))'.

Leftmost derivation for the string '(( ) ( ))' is:

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (SS) \\ &\Rightarrow (( )S) \\ &\Rightarrow (( )( )) \end{aligned}$$

We shall use this derivation for the simulation, as follows:

Current state	Stack contents	Tape and head
Start	$\not b$	(( ) ( )) $\not b \not b$ ...
↓		↑
Push$_1$ S	$\not b S$	(( ) ( )) $\not b \not b$ ...
↓		↑

Pop	$\not b$	(( ) ( )) $\not b \not b$ ...
S ↓		↑
Push$_4$)	$\not b$)	(( ) ( )) $\not b \not b$ ...
↓		↑
Push$_5$ S	$\not b$) S	(( ) ( )) $\not b \not b$ ...
↓		↑
Push$_6$ (	$\not b$) S (	(( ) ( )) $\not b \not b$ ...
↓		↑
Pop	$\not b$) S	(( ) ( )) $\not b \not b$ ...
(↓		↑
Read$_1$	$\not b$) S	(( ) ( )) $\not b \not b$ ...
(↓		↑
Pop	$\not b$)	(( ) ( )) $\not b \not b$ ...
S ↓		↑
Push$_2$ S	$\not b$) S	(( ) ( )) $\not b \not b$ ...
S ↓		↑
Push$_3$ S	$\not b$) SS	(( ) ( )) $\not b \not b$ ...
↓		↑
Pop	$\not b$) S	(( ) ( )) $\not b \not b$ ...
S ↓		↑
Push$_7$)	$\not b$) S)	(( ) ( )) $\not b \not b$ ...
↓		↑
Push$_8$ (	$\not b$) S) (	(( ) ( )) $\not b \not b$ ...
↓		↑
Pop	$\not b$) S)	(( ) ( )) $\not b \not b$ ...
(↓		↑
Read$_1$	$\not b$) S)	(( ) ( )) $\not b \not b$ ...
(↓		↑
Pop	$\not b$) S	(( ) ( )) $\not b \not b$ ...
) ↓		↑
Read$_2$	$\not b$) S	(( ) ( )) $\not b \not b$ ...
) ↓		↑
Pop	$\not b$)	(( ) ( )) $\not b \not b$ ...
S ↓		↑
Push$_7$)	$\not b$))	(( ) ( )) $\not b \not b$ ...
↓		↑
Push$_8$ (	$\not b$)) (	(( ) ( )) $\not b \not b$ ...
↓		↑
Pop	$\not b$))	(( ) ( )) $\not b \not b$ ...
(↓		↑
Read$_1$	$\not b$))	(( ) ( )) $\not b \not b$ ...
(↓		↑
Pop	$\not b$)	(( ) ( )) $\not b \not b$ ...
) ↓		↑

$Read_2$	$\not b)$	$((\,)(\,))\not b\not b\ldots$ ↑
$)\downarrow$		
Pop	$\not b$	$((\,)(\,))\not b\not b\ldots$ ↑
$)\downarrow$		
$Read_2$	$\not b$	$((\,)(\,))\not b\not b\ldots$ ↑
$)\downarrow$		
Pop	—	$((\,)(\,))\not b\not b\ldots$ ↑
$\not b\downarrow$		
$Read_3$	—	$((\,)(\,))\not b\not b\ldots$ ↑
$\not b\downarrow$		
Accept		

Example 6.8 Construct a PDA that accepts the language generated by the grammar

$S \rightarrow S + S \mid S * S \mid 4$

Solution The required NPDA is constructed as shown in Fig. 6.17.

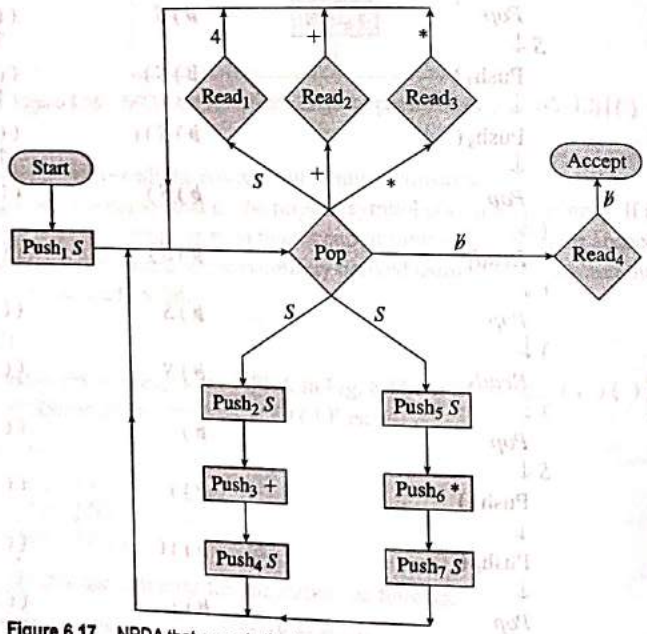


Figure 6.17 NPDA that accepts the language generated by $S \rightarrow S + S \mid S * S \mid 4$

Simulation

Let us simulate the working of the NPDA that we have constructed for the string '4 + 4 * 4'

Leftmost derivation for the string '4 + 4 * 4' is as follows:

$S \Rightarrow S + S$
$\Rightarrow 4 + S$
$\Rightarrow 4 + S * S$
$\Rightarrow 4 + 4 * S$
$\Rightarrow 4 + 4 * 4$

Based on the leftmost derivation, the simulation for the acceptance of the string '4 + 4 * 4' is as follows:

Current state	Stack contents	Tape and head
Start	$\not b$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
$Push_1 S$	$\not b S$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
Pop	$\not b$	4 + 4 * 4 $\not b\not b$ ... ↑
$S\downarrow$		
$Push_2 S$	$\not b S$	4 + 4 * 4 $\not b\not b$ ... ↑
$S\downarrow$		
$Push_3 +$	$\not b S +$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
$Push_4 S$	$\not b S + S$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
Pop	$\not b S +$	4 + 4 * 4 $\not b\not b$ ... ↑
$S\downarrow$		
$Read_1$	$\not b S +$	4 + 4 * 4 $\not b\not b$ ... ↑
$4\downarrow$		
Pop	$\not b S$	4 + 4 * 4 $\not b\not b$ ... ↑
$+\downarrow$		
$Read_2$	$\not b S$	4 + 4 * 4 $\not b\not b$ ... ↑
$+\downarrow$		
Pop	$\not b$	4 + 4 * 4 $\not b\not b$ ... ↑
$S\downarrow$		
$Push_5 S$	$\not b S$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
$Push_6 *$	$\not b S *$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
$Push_7 S$	$\not b S * S$	4 + 4 * 4 $\not b\not b$ ... ↑
↓		
Pop	$\not b S *$	4 + 4 * 4 $\not b\not b$ ... ↑
$S\downarrow$		
$Read_1$	$\not b S *$	4 + 4 * 4 $\not b\not b$ ... ↑
$4\downarrow$		

Pop	♭ S	4 + 4 * 4 ♭ ♭ ...
* ↓ *Read*$_3$	♭ S	4 + 4 * 4 ♭ ♭ ...
* ↓ *Pop*	♭	4 + 4 * 4 ♭ ♭ ...
S ↓ *Read*$_1$	♭	4 + 4 * 4 ♭ ♭ ...
4 ↓ *Pop*	—	4 + 4 * 4 ♭ ♭ ...
♭ ↓ *Read*$_4$	—	4 + 4 * 4 ♭ ♭ ...
♭ ↓ Accept		

6.7.1 NPDA Construction using Chomsky Normal Form

As we know, Chomsky normal form (CNF) mandates only two forms of production rules:

1. $S \rightarrow AB$
2. $S \rightarrow a$

Production rule of type 1 has exactly two non-terminals on the right-hand side, while the production rule of type 2 contains a single terminal symbol at the right-hand side.

In the previous two examples, we observe that even the terminal symbols are pushed onto the stack and again popped to match the input symbols. The complexity of this algorithm can be reduced using CNF. If we express the grammar in CNF, then the production rule of type 1 is used to push the right-hand side onto the stack, while the rule of type 2 is directly used to match the input symbols from the tap. This has two benefits:

1. Terminals are not pushed onto the stack.
2. At most, two non-terminal symbols are pushed onto the stack at every step in the derivation process.

This in turn helps control the stack growth, which as we have seen, is much more without CNF. On the other hand, grammar without CNF may have multiple symbols on the right-hand side of production rules.

Example 6.9 Construct a PDA that accepts the following language:

$$L = \{a^{2n} \mid n > 0\}$$

Solution The CFG for the given language can be written as

$$S \rightarrow SS \mid aa$$

Let us express the grammar in CNF:

$$S \rightarrow SS \mid AA$$
$$A \rightarrow a$$

The NPDA is constructed using the CFG expressed in CNF as shown in Fig. 6.18.

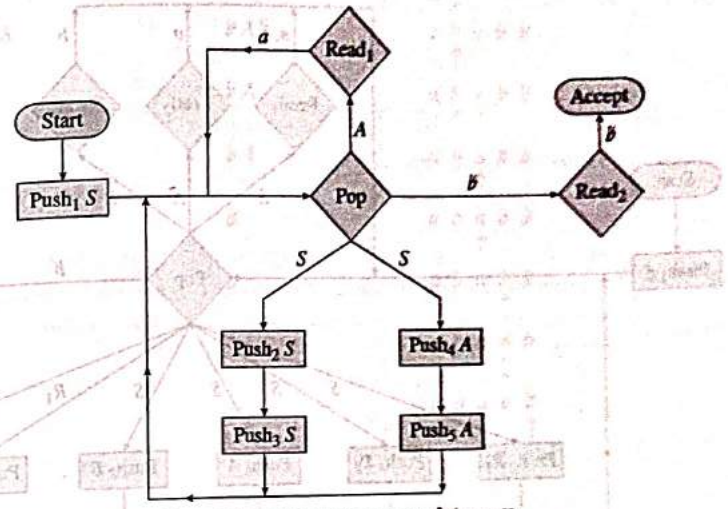


Figure 6.18 NPDA that accepts $\{a^{2n} \mid n > 0\}$

We see from Fig. 6.18 that in the grammar expressed in CNF, no terminal gets pushed onto the stack.

Example 6.10 Construct a PDA that accepts all palindrome strings over $\Sigma = \{a, b\}$.

Solution This is the same as Example 6.6 that we solved earlier. Let us apply the CNF and then compare the efficiency of the resultant PDA.

The CFG for the language is defined as

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon$$

Removing the ϵ-production, the grammar can be written as

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid a\,a \mid b\,b$$

Now, let us convert this grammar to CNF:

$$S \rightarrow A\,S\,A \mid B\,S\,B \mid A\,A \mid B\,B \mid a \mid b$$
$$A \rightarrow a$$
$$B \rightarrow b$$

Therefore, the final CFG expressed in the CNF, which can be considered for constructing the required PDA is as follows:

$$S \rightarrow A\,R_1 \mid B\,R_2 \mid A\,A \mid B\,B \mid a \mid b$$
$$R_1 \rightarrow SA$$
$$R_2 \rightarrow SB$$
$$A \rightarrow a$$
$$B \rightarrow b$$

Using this grammar, the NPDA can be constructed as shown in Fig. 6.19.

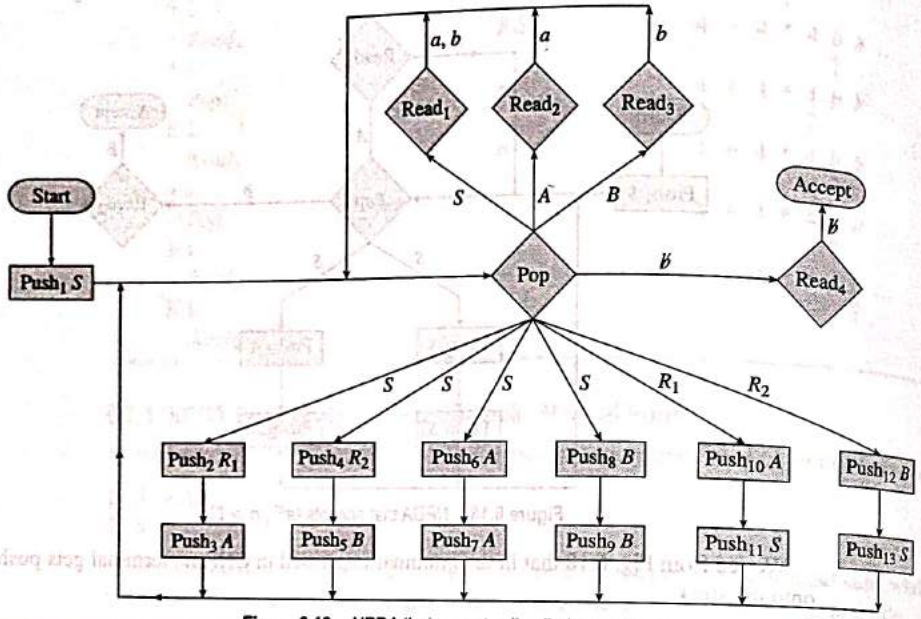


Figure 6.19 NPDA that accepts all palindrome strings

Simulation

Let us simulate the working of the NPDA in Fig. 6.19 for the string '*aba*'.

Current state	Stack contents	Tape and head
Start ↓	$\not b$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Push$_1$ S ↓	$\not b\ S$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Pop S ↓	$\not b$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Push$_2$ R_1 ↓	$\not b\ R_1$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Push$_3$ A ↓	$\not b\ R_1 A$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Pop A ↓	$\not b\ R_1$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Read$_2$ a ↓	$\not b\ R_1$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Pop R_1 ↓	$\not b$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Push$_{10}$ A ↓	$\not b\ A$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Push$_{11}$ S ↓	$\not b\ A S$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Pop S ↓	$\not b\ A$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Read$_1$ b ↓	$\not b\ A$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Pop A ↓	$\not b$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Read$_2$ a ↓	$\not b$	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Pop $\not b$ ↓	—	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Read$_4$ $\not b$ ↓	—	$a\ b\ a\ \not b\ \not b\ \ldots$ ↑
Accept		

This simulation is based on the following leftmost derivation:

$$\begin{aligned} S &\Rightarrow \underline{A}\, R_1 \\ &\Rightarrow a\, \underline{R_1} \\ &\Rightarrow a\, \underline{S}\, A \\ &\Rightarrow a\, b\, \underline{A} \\ &\Rightarrow a\, b\, a \end{aligned}$$

6.8 CLOSURE PROPERTIES OF CFLs

Context-free languages (CFLs), like RLs, are also closed under union, concatenation, and Kleene closure. Let us attempt to prove these facts using CFG and PDA concepts.

Theorem 6.1

If L_1 and L_2 are CFLs, then their union '$L_1 \cup L_2$' is also a CFL. In other words, ***CFLs are closed under union.***

Proof

Let us consider two context-free languages, L_1 and L_2. This means that these languages can be represented using context-free grammars (CFGs).

Let us assume that L_1 is represented by the CFG G_1, which has S_1 as its starting non-terminal (or start symbol). Hence, G_1 begins with a production of the form

$$S_1 \rightarrow \alpha_1$$
$$\ldots$$

Similarly, let us assume that L_2 is represented by G_2, which has S_2 as the start symbol; then G_2 begins with a production of the form

$$S_2 \rightarrow \alpha_2$$
$$\cdots$$

From the definition of a CFG, we know that we can combine G_1 and G_2 into a grammar G, which is defined as follows:

$$S \rightarrow S_1 \mid S_2$$

We see that S is the starting non-terminal for the new grammar G that is built with only one additional grammar rule. The start symbol S can now generate a language that can be derived from S_1 as well as S_2. In other words, we can say that it generates the union of L_1 and L_2, that is, $L_1 \cup L_2$.

For example, let us assume L_1 is represented by the following CFG:

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon$$

and L_2 is represented by the following CFG:

$$S \rightarrow a\,S\,b \mid \epsilon$$

Then, the CFG for '$L_1 \cup L_2$' is defined as

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow a\,S_1\,a \mid b\,S_1\,b \mid a \mid b \mid \epsilon$$
$$S_2 \rightarrow a\,S_2\,b \mid \epsilon$$

We can also prove this closure property by constructing individual PDAs for L_1 and L_2 and combining the two to form an NPDA with a new 'start' state having two outgoing edges, each incident onto the start states of these PDAs. Thus, the resulting NPDA accepts a language, which is also a CFL, and equal to '$L_1 \cup L_2$'.

Theorem 6.2

If L_1 and L_2 are CFLs, then their concatenation '$L_1 \cdot L_2$' is also a CFL. In other words, *CFLs are closed under concatenation.*

Proof

Let us consider two context-free languages, L_1 and L_2 that can be represented using CFGs.

Let us assume that L_1 is represented by the CFG G_1, which has S_1 as its starting non-terminal (or start symbol). Hence, G_1 begins with a production of the form

$$S_1 \rightarrow \alpha_1$$
$$\cdots$$

Similarly, if we assume that L_2 is represented by the CFG G_2, which has S_2 as the start symbol, then G_2 begins with a production of the form

$$S_2 \rightarrow \alpha_2$$
$$\cdots$$

From the definition of a CFG, we know that we can combine G_1 and G_2 into G, which is described as follows:

$$S \rightarrow S_1\,S_2$$

We see that S is the starting non-terminal for the new grammar G that is built with only one additional grammar rule. Now, S can generate a language that is the concatenation of L_1 and L_2, that is, '$L_1 \cdot L_2$'.

Let us assume that L_1 is represented by the CFG

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon$$

and L_2 is represented by the CFG

$$S \rightarrow a\,S\,b \mid \epsilon$$

The CFG for '$L_1 \cdot L_2$' is defined as

$$S \rightarrow S_1\,S_2$$
$$S_1 \rightarrow a\,S_1\,a \mid b\,S_1\,b \mid a \mid b \mid \epsilon$$
$$S_2 \rightarrow a\,S_2\,b \mid \epsilon$$

This can be represented using two individual PDA for L_1 and L_2 respectively such that the final state of the PDA for L_1 is merged with the start state of the PDA for L_2.

Theorem 6.3

If L is a CFL, then L^* (Kleene closure of L) is also a CFL. In other words, *CFLs are closed under Kleene closure.*

Proof

Let us assume that L is represented by a CFG G that has a starting non-terminal S and is defined as follows:

$$S \rightarrow \alpha$$
$$\cdots$$

Let us construct a grammar G_1 having starting symbol S_1 with the following production rules:

$$S_1 \rightarrow S\,S_1 \mid \epsilon$$

Thus, G_1 can generate a language obtained from repetitive concatenation of L. The resultant language is thus obtained by concatenating L to itself zero or more times; this is the same as L^*.

For example, let the CFG for language L be defined as

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon$$

then, the CFG for L^* is

$$S_1 \rightarrow S\,S_1 \mid \epsilon$$
$$S \rightarrow a\,S\,a \mid b\,S\,b \mid a \mid b \mid \epsilon$$

We can represent L by a PDA. If we combine the start state and the final state of this PDA, it will represent the PDA for L^*.

6.9 ADDITIONAL PDA EXAMPLES

In this section, let us discuss some more examples on the construction of PDA.

Example 6.11 Construct a PDA that recognizes the following language:

$\{a^n x \mid n \geq 0, x \in \{a, b\}^* \text{ and } |x| \leq n\}$

Solution We see that for the given language, it is not possible to determine where the string of a's—that is, a^n—ends, and the string x starts. This is because x also consists of a's. Hence, we design an NPDA as shown in Fig. 6.20.

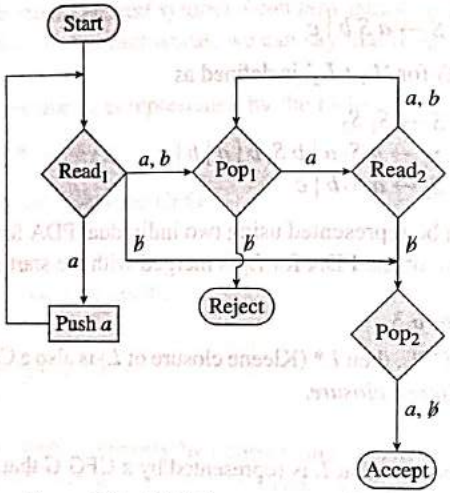


Figure 6.20 NPDA that accepts $\{a^n x \mid n \geq 0, x \in \{a, b\}^* \text{ and } |x| \leq n\}$

Algorithm

1. Push all a's onto the stack till we reach the beginning of x. (Remember that x may start with either a or b. Determinism is impossible to achieve if x starts with a).
2. Read one symbol from x—either a or b—and pop one a from the stack.
3. Continue the process till you reach the end of the input.
4. When the input ends, that is, when you read $\not b$ on the tape, the stack may either be empty (if $|x| = n$) or may have a few a's left (if $|x| < n$).
5. In either case, the stack cannot be empty before the input ends, as $|x| > n$ is not allowed. Hence, in such a case, the input string should be rejected.

Example 6.12 Construct a PDA for the language described as follows:
The set of all strings over alphabet $\{a, b\}$ with exactly equal number of a's and b's.

Solution We need to consider the fact that the string might begin with either a or b. Therefore, pushing only a's or only b's will not work. We need to push whenever the stack

is empty or the top of the stack carries the same symbol as the one just read. Refer to the algorithm that follows. The required DPDA is depicted in Fig. 6.21.

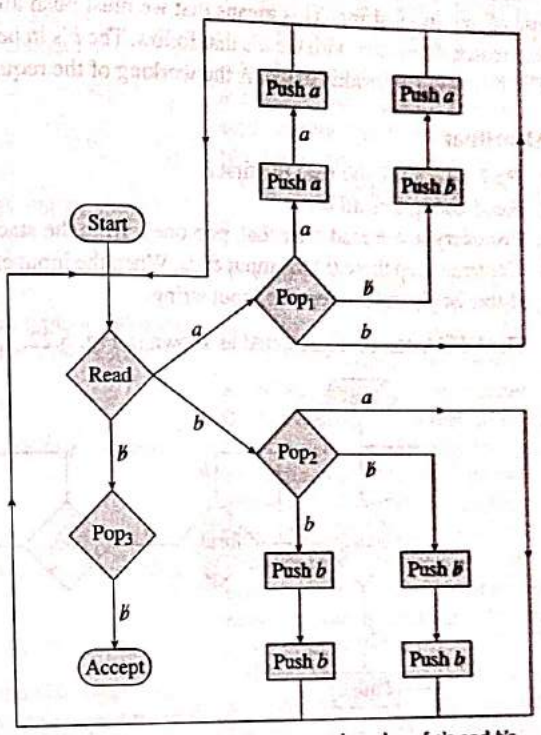


Figure 6.21 DPDA that accepts equal number of a's and b's

Algorithm

1. Read a symbol, either a or b.
2. If the stack is empty, that is, the top of the stack top contains $\not b$, push the symbol that has been read.
3. If the top of the stack contains the same symbol as the symbol that has been read, then also push this symbol.
4. If the top of the stack carries a symbol that is different from what has been read—for example, if the symbol read is a and the top of the stack contains b, or vice versa—pop the symbol onto the top of the stack. This contributes to matching a's with b's, or vice versa.
5. Continue with the aforementioned four steps until the input string ends, that is, until you read $\not b$ on the tape.
6. When the input string ends, then the top of this stack should be $\not b$. If this holds true, then accept the input string.

Example 6.13 Construct a PDA that accepts the language $L = \{a^n b^m a^n \mid m, n \geq 1\}$.

Solution For this language, we need to match the number of a's at the beginning and the end of any input string. This means that we must push all the a's before the string of b's and match them later with the a's that follow. The b's in between can be read and ignored. The following algorithm explains the working of the required PDA.

Algorithm

1. Push all a's till you read the first b.
2. Read and ignore all b's.
3. For every a you read after that, pop one a from the stack.
4. Continue step three till the input ends. When the input ends, the stack should be empty; if this holds true, accept the input string.

The DPDA can be constructed as shown in Fig. 6.22.

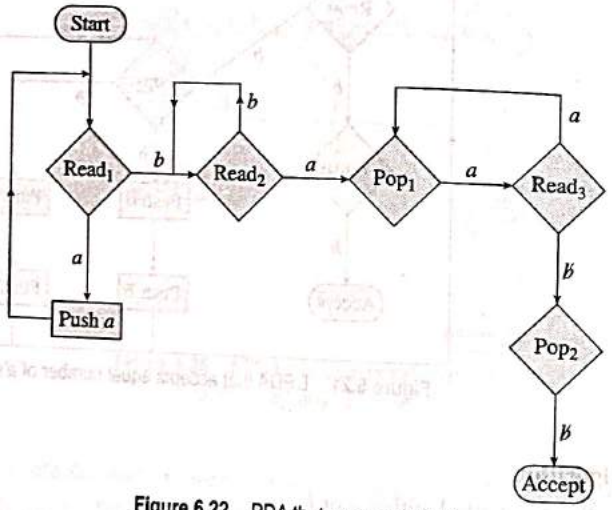


Figure 6.22 PDA that accepts $a^n b^m a^n$

SUMMARY

A pushdown stack-memory machine (PDM) is a computational model that we can use to solve any problem that has an algorithmic solution and requires a single stack memory.

A PDM accepts or recognizes regular languages (RLs) as well as context-free languages (CFLs). Since the set of RLs is a proper subset of the class of CFLs, for every finite state machine (FSM), there exists an equivalent PDM, but not vice versa. Hence, a PDM is more powerful than the FSM due to its infinite memory in the form of an external stack.

Most commonly, a PDM is represented with the help of a flowchart-like diagram. Its formalism is obtained using a pushdown automaton (PDA), which is the mathematical model of a PDM.

A pushdown automaton M is denoted as

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where,

Q: Finite set of states

Σ: Input alphabet (input strings are composed of the symbols from Σ)

Γ: Stack alphabet

q_0: Initial state; $q_0 \in Q$

Z_0: $Z_0 \in \Gamma$ is a particular stack symbol called the start symbol, usually considered as a blank character ß in many designs.

F: Set of final states; $F \subseteq Q$

The δ function for deterministic PDA or DPDA is defined as

$$\delta: Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$$

The δ function for non-deterministic PDA or NPDA is defined as

$$\delta: Q \times \{\Sigma \cup (\varepsilon)\} \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

In the case of a DPDA, the transition function $\delta(q, a, Z)$ does not contain more than one element for any state q in Q, any symbol Z in Γ on top of the stack, and any input symbol a in Σ. Thus, the transition would be of the form

$$\delta(q, a, Z) = (p, \gamma)$$

where, p is the unique next state in Q to which the machine makes the transition, and γ is a member of Γ^*(zero or more occurrences of symbols from Γ), and can be one of the following:

1. Empty, that is, ε, if the stack operation performed is 'pop'. This means that Z is popped out of the stack.
2. Z, if the stack is not updated; only a state transition is performed.
3. 'xx...xxZ', if multiple symbols 'xx...xx' are pushed onto the stack.
4. 'xx...xx', if Z is popped out of the stack and multiple symbols 'xx...xx' are pushed onto the stack.

An NPDA performs any one of the multiple actions available for an input symbol read and the same stack symbol that is on top of the stack. Hence, NPDA is a possibilistic machine.

For example, let us consider the following transition:

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}.$$

where, $q, p_1, p_2, \ldots, p_m$ are states in Q, a is any input symbol in Σ, Z is any stack symbol in Γ, and all γ_i, for $1 \leq i \leq m$, are members of Γ^*.

The NPDA accepts the entire class of CFLs, while DPDA cannot be constructed to accept a few CFLs. Hence, DPDA is less powerful than NPDA. In other words, DPDA can only accept a subset of the class of CFLs. We see that this is slightly different from the finite automata (FA) scenario, where non-deterministic FA (NFA) is equivalent in power to the deterministic FA (DFA). Thus, for every NFA we can construct an equivalent DFA; however, every NPDA cannot be transformed into an equivalent DPDA.

There exists an algorithm to automatically build an NPDA from the given context-free grammar (CFG). The NPDA thus built is more like a simple top-down parser, which uses the leftmost derivation to regenerate the input string and check its validity. If it can generate all the input symbols in the same order as in the input string using the given CFG, it is considered to be a valid input string for the parser.

If a parser is generated using a CFG, which is expressed in Chomsky normal form (CNF), it has the advantage of controlled stack growth. It is also more efficient since no terminal symbol gets unnecessarily pushed onto the stack.

Context-free languages (CFLs) are closed under these operations—union, concatenation, and Kleene closure.

Turing Machines

LEARNING OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Computational problems that can be solved using a Turing machine (TM)
- Formal model of a TM
- Comparison of finite automata (FA) and TM on the basis of their relative computational powers
- Recursively enumerable languages and recursive languages
- Concept of composite TM and iterative TM
- Multi-tape TM, multi-stack TM, and multi-track TM
- Concept of universal Turing machine (UTM)
- Halting problem and limits of computability
- Church's Turing hypothesis
- Solvable, semi-solvable, and unsolvable problems
- Total and partial recursive functions
- Post's correspondence problem (PCP)

4.1 INTRODUCTION

The limitations of finite state machines (FSMs) necessitate the search for a more powerful machine. We have seen that an FSM cannot remember an arbitrarily long sequence of symbols. It has a read head that can move only in one direction—to the right always. It cannot move backwards to retrieve previous information stored onto the tape. Similarly, though the two-way finite automaton (2FA) has a read head that can move in any direction, it cannot store (write) anything onto the tape; hence, it cannot multiply two numbers, as the process requires intermediate results to be stored onto the tape. Furthermore, an FSM cannot check if a set of parentheses are well-formed, or for palindrome sequences. All these limitations arise because the FSMs do not have memory, and hence, they cannot solve problems that need to store data to be used for later computation.

We have already discussed the limitations of FSMs in detail in Chapter 2 (refer to Section 2.14). To overcome these limitations, we require a more powerful machine, which has the following properties:

1. Finite state control similar to that of the FSM—since any program needs a finite number of functions/states.
2. External memory capable of remembering arbitrarily long sequences of inputs—to achieve this, the machine should have unbounded one-dimensional memory from which, it can choose any required part, by moving to the left, right, or staying in one position (i.e., no movement)—the entire unbounded tape acts as an infinite memory.
3. Ability to store (write onto the tape)—the machine head should be a read/write head.
4. Ability to consume its own output as input during execution at a later time (like a complete feedback control system)—for this, all the intermediate information must be stored in the memory. For example, while multiplying numbers (multiplication is a process of repetitive addition), it is required to store the intermediate or partial sums, which are later added to get the final answer. Hence, the distinction between the input and output symbols needs to be dropped. This means that external communication as well as communication within the machine should rest on a common alphabet set.

All the aforementioned characteristics are satisfied by the *Turing machine* (TM), which was proposed by Alan Turing, a decade prior to the designing of the first shared-program computer. This concept of the Turing machine led to the concept of algorithms and finite procedures, since the basis of the TM is to first divide the process into primitive operations (functions/procedures/states) such that they cannot be further divisible, and then execute each operation in sequence.

4.2 ELEMENTS OF A TURING MACHINE

A TM consists of the following:

1. A head, which can read or write a symbol at a time, and move either to the left, right or remain in the same position, depending on the symbol read from the tape cell.
2. An infinite tape, extending on either side of the head, marked-off into square cells, on which the symbols from an alphabet set can be written. The tape represents the unbounded one-dimensional memory (refer to Fig. 4.1), which is considered to be filled with blank characters, *b*'s, unless specified otherwise.
3. A finite set of symbols, called the external alphabet set *I*, which consists of lower-case English letters, digits, usual punctuation marks, and the blank character *b*.
4. A finite set of states denoted by *S*; the machine resides in one of these states.

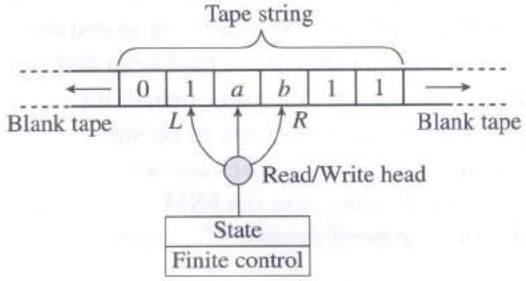


Figure 4.1 Turing machine

The TM thus overcomes all the limitations of the FSM. It has infinite memory in the form of a tape that is unbounded at both the ends; a read/write head that can move in any direction, and helps the machine retrieve information from the tape that is stored earlier—in a way, this can be considered as converting the dumb storage into memory; it can consume its own output as input for use at a later stage in the algorithm execution. All these features make it a completely powerful machine. The TM and FSM are thus two extremes—an FSM is a minimal machine, whereas a TM is the most powerful machine.

4.3 TURING MACHINE FORMALISM

As we have already seen, a TM has a finite set of symbols I, and a finite set of states S (one of which is the halt/final state).

A TM is denoted with the help of three functions, namely, the machine function (MAF), state transition function (STF), and the direction function (DIF), which are defined as:

$$\text{MAF: } I \times S \rightarrow I$$
$$\text{STF: } I \times S \rightarrow S$$
$$\text{DIF: } I \times S \rightarrow D = \{L, R, N\}$$

where, L stands for 'move towards left by one tape cell position', R stands for 'move towards right by one tape cell position', and N stands for 'no movement, that is, stay in the same position or remain at the same tape cell, which has been read last'.

Note: Observe that the MAF for FSM is defined as: $I \times S \rightarrow O$, as there is a distinction between input and output; on the other hand, in case of a TM, it is defined as: $I \times S \rightarrow I$.

Based on the machine's state—which is an element of S—at any given time, and the input symbol read—which is an element of I—at any given time, the machine either takes no action (i.e., halt state) or performs the following three actions:

1. The input symbol read is erased and another symbol (possibly a blank character or the same symbol again) is printed on the tape cell.
2. The internal state of the machine is changed (possibly to the same state as it was initially in).
3. The head either moves one cell towards the right or left, or remains in the same position.

These actions can be described collectively by an ordered set of five elements, that is, a quintuple. For example, (a, α, b, β, L) represents such a quintuple, which exists if the head reads $a \in I$, while the machine is in state $\alpha \in S$, writes b onto the tape cell after erasing a, changes the machine state to $\beta \in S$, and moves the head position to the left by one cell.

To represent these quintuples together, a table called a transition matrix or functional matrix (FM) is used. Such a matrix is a combination of the aforementioned three individual functions—the MAF, STF, and DIF. In this functional matrix, the rows and columns are labelled by symbols forms S and I, respectively (as in FSMs), while the remaining triples, that is, (b, β, L) are placed as the entries of the matrix.

Table 4.1 Example functional matrix

S \ I	0	1	♭
α	$0\alpha R$	$0\beta R$	$\flat\alpha R$
β	—	—	$\flat\gamma N$
γ	—	—	—

If the number of symbols in S, that is, the number of states = $|S| = m$, and the number of symbols in I, that is, number of tape symbols = $|I| = n$, then the size of the functional matrix is $(m \times n)$ number of output triples. For an example, refer to Table 4.1.

We see that in Table 4.1, S has three states and I has three tape symbols; hence, the functional matrix is of size $3 \times 3 = 9$.

If δ is a functional matrix, then an example transition can be expressed as:

$$\delta\,(a, \alpha) \rightarrow (a, \alpha, L)$$

This means, on reading symbol a while in α state, the machine erases a, writes a again, changes the state from α to itself, and moves the head position to the left by one tape cell.

In the aforementioned case, the significant change after transition is just the position of the head, because the symbol on the tape a and the current state of the machine, that is, α, remain the same even after the transition. Therefore, instead of writing the whole triple in the functional matrix, we can save space by writing only the significant changes. Hence, the aforementioned triple can be simplified as:

$$\delta\,(a, \alpha) \rightarrow (L)$$

Table 4.2 Simplified functional matrix

S \ I	0	1	♭
α	R	$0\beta R$	R
β	—	—	γN
γ	—	—	—

The modified functional matrix, in which we only record the significant changes after deleting all other insignificant entries to save space, is called a *simplified functional matrix* (SFM). The SFM equivalent to the functional matrix in Table 4.1 is depicted in Table 4.2.

Note: It is also possible to convert a quintuple representation to a quadruple representation. For this, additional states may have to be introduced in S, so that the given TM is deterministic. For example, the quintuple (a, α, b, β, R) may be written as two quadruples: (a, α, b, β') and (b, β', R, β), where, β' is a new state. This quadruple notation essentially breaks two of the three operations into different transitions. The first transition (a, α, b, β') is about replacing the input symbol a by b and the second transition (b, β', R, β) is about moving the head position one cell to the right. However, this quadruple notation is very rarely used; the quintuple notation discussed here is more popular.

Formal Definition

A Turing machine (TM) is denoted here:

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

where,

Q: Finite set of states

Γ: Finite set of allowable tape symbols, including blank character ♭

Σ: The set of input symbols, which is a subset of Γ, excluding blank character ♭

B: A symbol for blank character ♭

q_0: Start (or initial) state $\in Q$
F: Set of final states (or halt states) $\subseteq Q$
δ: Functional matrix, such that δ: $(Q \times \lceil) \rightarrow (Q \times \lceil \times \{L, R, N\})$

In order to have a deterministic TM, there should be a unique triple (b, β, d) for every state α on some symbol a, for $\delta(\alpha, a)$ in the functional matrix.

Notes:

1. δ may be undefined for some arguments, which are represented in the table as '—', (similar to the transition table of an FSM). Refer to Tables 4.1 and 4.2.
2. FSM is a special case of a TM. If we make the tape length finite and restrict the head movement only to one direction, then the functional matrix corresponding to this FSM will specify: $I \times S \rightarrow S$, which is, nothing but the state transition function (STF).

4.4 INSTANTANEOUS DESCRIPTION

Instantaneous description (ID) of a TM M is denoted by '$\alpha_1\ q\ \alpha_2$', where q is the current state of the TM, that is, $q \in Q$, and '$\alpha_1\ \alpha_2$' is the string in $\lceil^*$—that is, the contents of the tape bounded on both the ends by blank characters ($\not b$'s). Note that $\not b$ may occur within '$\alpha_1\ \alpha_2$' as well.

We define a move (or transition) of M as follows:

Let '$a_0\ a_1 \ldots a_{i-1}\ q\ a_i \ldots a_n$' be an ID of M. This ID indicates that the current state of the machine is q and the machine head is about to read symbol a_i from the tape.

Suppose, $\delta(q, a_i) = (*, p, L)$ is a transition, where

If $i = n + 1$, then a_i is taken to be $\not b$, that is, blank character.

If $i = 0$, then the tape head moves one position to the left of the first symbol that is considered to be blank character $\not b$, in the next ID.

If $i > 1$, then we can write the ID for the aforementioned transition as

$$a_0\ a_1 \ldots a_{i-1}\ q\ a_i \ldots a_n \underset{M}{\vdash} a_0\ a_1 \ldots a_{i-2}\ p\ a_{i-1}\ *\ a_{i+1} \ldots a_n$$

Similarly, for the move $\delta(q, a_i) = (*, p, R)$ we can write

$$a_0\ a_1 \ldots a_{i-1}\ q\ a_i \ldots a_n \underset{M}{\vdash} a_0\ a_1 \ldots a_{i-1}\ *\ p\ a_{i+1} \ldots a_n$$

If the two IDs are related by '$\underset{M}{\vdash}$', we say that the second results from the first by one move, as we have seen here. If one ID results from another by some finite number of moves (including zero number of moves), then they are related by the symbol, '$\overset{*}{\underset{M}{\vdash}}$'.

The language accepted by M is denoted by $L(M)$ and is the set of those words in * that cause M to enter a final state when M is placed in state q_0 and the tape head of M is at the leftmost cell.

Formally, we can define the language accepted by $M = (Q, \Sigma, \lceil, \delta, q_0, B, F)$ as

$$L(M) = \{w \mid w \in \Sigma^*, \text{ and } q_0 W \overset{*}{\underset{M}{\vdash}} \alpha_1\ p\ \alpha_2 \text{ for some } p \text{ in } F \text{ and } \alpha_1 \text{ and } \alpha_2 \text{ in } \lceil^*\}$$

Let us consider an example of a TM.

simplified functional matrix

Example 4.1 Consider an SFM for a TM with the following:

$$I = \{0, 1, \not b\}$$
$$S = \{\alpha, \beta, \gamma = \text{halt}\}$$
$$D = \{L, R, N\}$$

The SFM is given in Table 4.3.

Find the purpose of the aforementioned TM, provided the initial configuration of the TM is as given in Fig. 4.2.

Table 4.3 SFM for example TM

S \ I	0	1	$\not b$
α	R	$0\beta R$	R
β	—	—	γN
γ	—	—	—

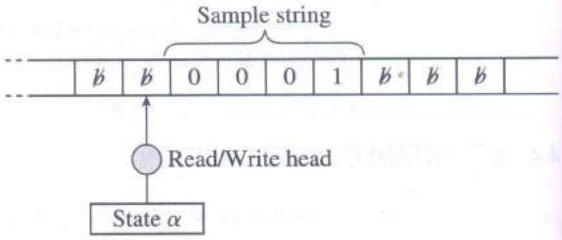


Figure 4.2 Initial configuration of example TM

Solution It is very important to note that the functional matrix gives us only half the information on what the TM does, and unless we know the initial configuration or initial ID of the TM, we cannot determine its purpose. Thus, the interpretation of the TM is highly dependent on its initial configuration, which includes the following: what the TM starts with, how the input string is assumed to be placed onto the tape, and the initial state of the machine. Note that the interpretation may differ for different initial configurations even for the same functional matrix.

Therefore, we can say that the initial configuration or the initial ID is like the assumption that forms the basis for the interpretation of the algorithm described by the functional matrix.

As we can see, initially the machine is the state α, and the head points to the blank character $\not b$ just before the actual string begins; hence, α is the initial or start state of the TM.

We observe that the function of the aforementioned TM is to replace the last (or ending) 1 by 0, whenever a sequence of 0's followed by a 1 is encountered; it then moves to the next available blank, and halts.

Let us simulate the working of the TM using a sample string '0001':

$$\alpha \not b\, 0\, 0\, 0\, 1\, \not b \qquad \text{initial configuration}$$
$$\vdash_M \not b\, \alpha\, 0\, 0\, 0\, 1\, \not b \qquad \delta(\alpha, \not b) = (R)$$
$$\vdash_M \not b\, 0\, \alpha\, 0\, 0\, 1\, \not b \qquad \delta(\alpha, 0) = (R)$$
$$\vdash_M \not b\, 0\, 0\, \alpha\, 0\, 1\, \not b \qquad \delta(\alpha, 0) = (R)$$

4.5 TRANSI

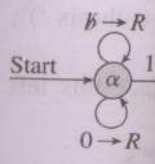


Figure 4.3

$$\vdash_M \quad \flat\,0\,0\,0\,\alpha\,1\,\flat \qquad \delta(\alpha, 0) = (R)$$

$$\vdash_M \quad \flat\,0\,0\,0\,0\,\beta\,\flat \qquad \delta(\alpha, 1) = (0\,\beta\,R)$$

$$\vdash_M \quad \flat\,0\,0\,0\,0\,\gamma\,\flat \qquad \delta(\beta, \flat) = (\gamma\,N)$$

The simulation of a TM for a particular input is a recording of the transition from one ID to another. The simulation given here is a formal notation. However, the IDs can also be represented using a slightly different notation as shown in the following simulation. We observe that in the changed notation, the head is clearly shown as a pointer pointing to the tape cell to be read. This change makes it visibly simple for the reader to understand. In this chapter, we are going to follow the same changed notation for representing the IDs for TM in preference to the formal one. Using the simpler ID notation, the aforementioned simulation can be shown as follows:

$\flat\,0\,0\,0\,1\,\flat$ $\uparrow$ α	initial configuration
$\flat\,0\,0\,0\,1\,\flat$ $\uparrow$ α	moved one position to right using $\delta(\alpha, \flat)$
$\flat\,0\,0\,0\,1\,\flat$ $\uparrow$ α	moved one position to right using $\delta(\alpha, 0)$
$\flat\,0\,0\,0\,1\,\flat$ $\uparrow$ α	moved right position to right using $\delta(\alpha, 0)$
$\flat\,0\,0\,0\,1\,\flat$ $\uparrow$ α	moved one position to right using $\delta(\alpha, 0)$
$\flat\,0\,0\,0\,0\,\flat$ $\uparrow$ β	changed 1 to 0, changed state to β, and moved one position to right using $\delta(\alpha, 1)$
$\flat\,0\,0\,0\,0\,\flat$ $\uparrow$ γ (halt)	changed state to γ with no movement using $\delta(\beta, \flat)$, halt after that

4.5 TRANSITION GRAPH FOR TURING MACHINE

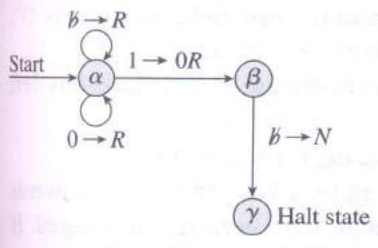


Figure 4.3 TG for example TM

A TM can also be represented pictorially with the help of a transition graph (TG). In such a graph, the nodes denote the internal states from the set S. A pair of nodes is connected by a directed edge (or arc) labelled by the input symbol from the set Γ, followed by an arrow and the new output symbol again from the set Γ— this symbol is to be written after erasing the previous symbol; the direction of the move is selected from the set D. The transition graph for the TM in Example 4.1 is shown in Fig. 4.3.

Note: If a TM starts the run from an initial state and eventually reaches a halt state, the computation is stopped and we say that it has *accepted* the input word.

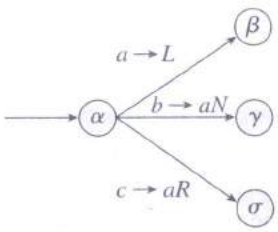


Figure 4.4 Example transition graph

On the other hand, let us consider the transition graph in Fig. 4.4. If the machine is in state α, and if the symbol read is different from a, b, and c, then we cannot proceed with the transition. This, in fact, corresponds to having unspecified entries in the functional matrix. Further, if the machine is in state β and reads a or b, then there is no way to proceed. In such situations, the computation is halted and we say that the machine has *rejected* the input word.

Sometimes, the machine may go into an infinite loop without ever halting. For example, if the triplet corresponding to ($\not b$, β) is ($\not b$, β, R), that is, δ (β, $\not b$) = ($\not b$, β, R), and the remainder of the tape consists of only blank characters, then the machine goes on moving to the right without ever changing the state, creating an infinite loop.

4.6 SOLVED PROBLEMS

Let us try to construct some TMs that solve specific problems.

Example 4.2 Construct a TM for checking if a set of parentheses are well-formed.

Solution Before directly creating the functional matrix let us first decide the initial configuration, that is, we first begin with the assumption on which the algorithm is to be fixed, and then generate the FM.

The initial configuration of this TM is as shown in Fig. 4.5(a).

It is assumed that the input string is to be written onto the tape delimited by semicolons on both the sides. The initial state of the machine is q_0, and the machine head points to the leftmost symbol of the input string. In case the input string is empty, the head points to the right end-marker semicolon.

Algorithm

1. From the initial state, move to the right until you read the first right parenthesis ')', replace this right parenthesis by '*' and move one position to the left.
2. Continue moving left till you read the first left parenthesis '(' ; replace this left parenthesis by '*' and move to the right.
3. Repeat the aforementioned two steps till the whole string is replaced by *'s.
4. While moving right if you came across ';' (right end-marker), then move towards the left to search for any left parenthesis '(' that might have been left unchanged. If the parentheses are well-formed, there will be no more left parentheses '('; and the machine will only read the left end-marker ';'.

For this TM, we have:

$$I = \{(, *,), ;, R_p, L_p, O\}$$
$$S = \{q_0, q_1, q_2, q_3 = \text{halt}\}$$
$$D = \{L, R, N\}$$

Here, q_0 is considered as the start state while q_3 is the halt state.

The SFM for the TM is as given in Table 4.4.

Table 4.4 SFM of a TM that checks if parentheses are well-formed

S \ I	(	*	)	;	R_p	L_p	O
q_0	R	R	$*q_1L$	q_2L	—	—	—
q_1	$*q_0R$	L	L	R_pq_3N	—	—	—
q_2	L_pq_3N	L	L	Oq_3N	—	—	—
q_3	—	—	—	—	—	—	—

If the TM encounters an extra right parenthesis ')', it ends up in halt giving R_p as output. If there is an extra left parenthesis '(', it moves to state q_3, that is, halt state, giving L_p as output. If the entire sequence of parentheses is balanced, then the machine moves to the halt state q_3 giving the output O, indicating acceptable input.

We observe from Table 4.4 that state q_0 is responsible for searching for the first right parenthesis ')'. It moves towards the right, ignoring any left parenthesis '(' that is encountered. Once the right parenthesis is found, it is replaced with '*'; then the machine moves one position to the left and changes to state q_1. State q_1 is responsible for searching towards the left for the matching left parenthesis '('. Once the first left parenthesis '(' is found, it is replaced with '*', and the machine moves back to q_0. Thus, the states q_0 and q_1 represent the first two steps of the algorithm.

While moving towards the left and searching for the left parenthesis '(' while in state q_1, if the machine cannot find '(', that is, the machine reads ';' (left end-marker) instead, then it is an error condition. In such a case, the machine generates output symbol R_p, indicating an extra right parenthesis, and moves to the halt state q_3.

On the other hand, if while moving towards the right in q_0 and searching for the first right parenthesis, if the machine reads ';' (right end-marker), indicating there are no more right parentheses ')', to be changed to '*', the machine moves to state q_2. In state q_2, one needs to check for any unmatched left parentheses, '(', that are left. In state q_2, while moving to the left, if the machine comes across the symbol ';' (left end-marker), indicating that there are no more unmatched left parentheses '(', it is considered as the acceptance condition. Hence, the machine generates the output O, indicating that the input string is accepted, and moves to the halt state q_3.

However, in case while moving left it finds a left parenthesis '(', then it indicates the string is not well-formed and has at least one unmatched extra left parenthesis '('. This is an error condition and the machine generates output symbol L_p and halts.

The transition graph for this TM is shown in Fig. 4.5(b).

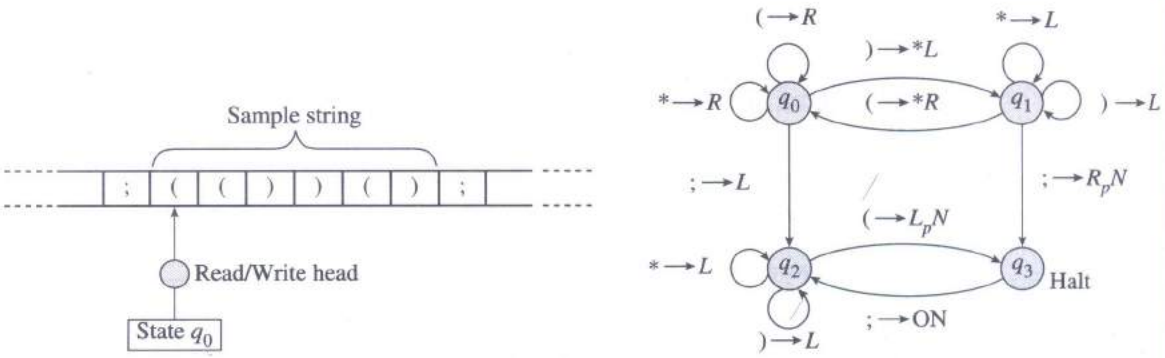
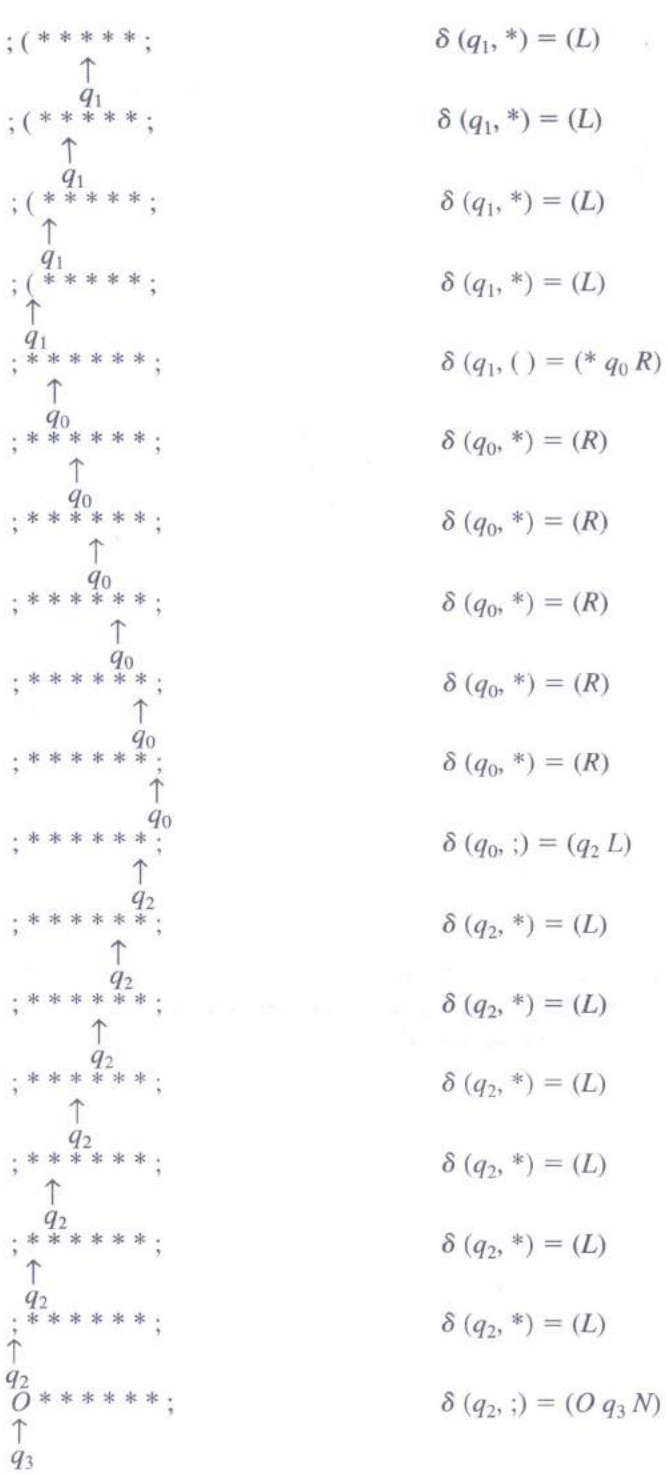


Figure 4.5 TM for checking if a set of parentheses are well-formed (a) Initial configuration (b) TG for TM that checks if parentheses are well-formed

Simulation

1. Let us simulate the working of the aforementioned TM on the sequence '(( ) ( ))'.

; (( ) ( )) ; ↑ q_0	initial configuration
; (( ) ( )) ; ↑ q_0	$\delta\ (q_0, (\ ) = (R)$
; (( ) ( )) ; ↑ q_0	$\delta\ (q_0, (\ ) = (R)$
; ((* ( )) ; ↑ q_1	$\delta\ (q_0,)\ ) = (*\ q_1, L)$
; (* * ( )) ; ↑ q_0	$\delta\ (q_1, (\ ) = (*\ q_0\ R)$
; (* * ( )) ; ↑ q_0	$\delta\ (q_0, *) = (R)$
; (* * ( )) ; ↑ q_0	$\delta\ (q_0, (\ ) = (R)$
; (* * (*) ; ↑ q_1	$\delta\ (q_0,)\ ) = (*\ q_1\ L)$
; (* * * *) ; ↑ q_0	$\delta\ (q_1, (\ ) = (*\ q_0\ R)$
; (* * * *) ; ↑ q_0	$\delta\ (q_0\ *) = (R)$
; (* * * * * ; ↑ q_1	$\delta\ (q_0,)\ ) = (*\ q_1\ L)$



The output symbol O indicates that the string '(( ) ( ))' is accepted by the TM as it is well-formed.

2. Let us now simulate the working of this TM for input sequence '('.

; (; ↑ q_0	initial configuration
; (; ↑ q_0	$\delta\ (q_0, (\ ) = (R)$
; (; ↑ q_2	$\delta\ (q_0, ;) = (q_2\ L)$
; L_p ; ↑ q_3	$\delta\ (q_2, (\ ) = (L_p\ q_3\ N)$

The output symbol L_p indicates that there is at least one left parenthesis '(', which is unmatched, and the input string is not well-formed.

3. Finally, let us simulate the working of this TM for an empty sequence:

; ; ↑ q_0	initial configuration
; ; ↑ q_2	$\delta\ (q_0, ;) = (q_2\ L)$
O ; ↑ q_3	$\delta\ (q_2, ;) = (O\ q_3\ N)$

We observe that the empty string is accepted, as it is always well-formed.

Example 4.3 Design a TM to find the 2's complement of a given binary number.

Solution The initial configuration of the TM is considered as shown in Fig. 4.6(a). The initial state is q_0, and the head points to the first digit, that is, the most significant bit (MSB), of the binary number. The binary number is delimited by semicolons, which are used as the end-markers. The remaining tape is filled with blank characters, that is, ₿'s.

The algorithm we are going to use is the same as that used in Example 2.17 in Chapter 2.

Algorithm

1. Move towards the right till you reach ';', which is the right end-marker of the sequence.
2. Start moving towards the left till you reach the first 1; then move towards the left and replace each 0 by 1, and each 1 by 0, till you reach the left end-marker ';' of the sequence, then halt.

For this TM, we have

$$I = \{0, 1, ;\}$$
$$S = \{q_0, q_1, q_2, q_3 = \text{halt}\}$$
$$D = \{L, R, N\}$$

The SFM for the TM is given in Table 4.5.

Table 4.5 SFM for a TM that finds 2's complement of a binary number

S \ I	0	1	;
q_0	R	R	q_1L
q_1	L	q_2L	q_3N
q_2	$1L$	$0L$	q_3N
q_3	—	—	—

The TG for the TM is as shown in Fig. 4.6(b).

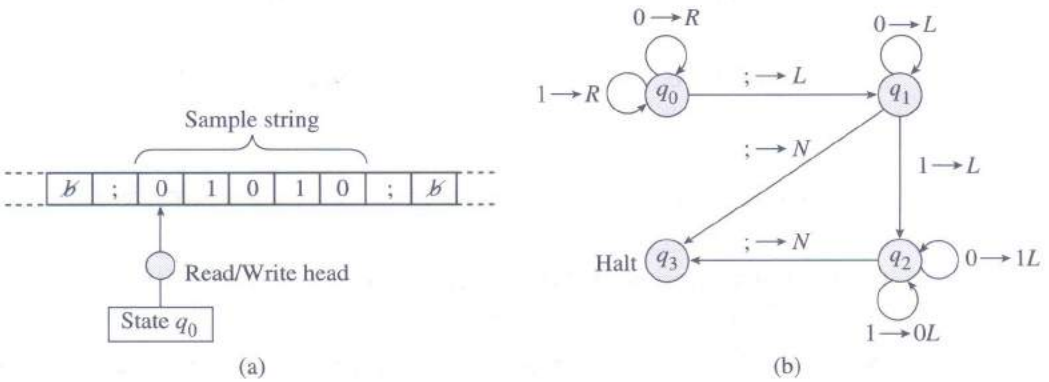


Figure 4.6 TM to find the 2's complement of a given binary number (a) Initial configuration (b) TG for TM that finds 2's complement of a binary number

Simulation

Let us simulate the working of this TM for the string '01010'.

```
;0 1 0 1 0;          initial configuration
 ↑
 q0
;0 1 0 1 0;          δ (q0, 0) = (R)
  ↑
  q0
;0 1 0 1 0;          δ (q0, 1) = (R)
   ↑
   q0
;0 1 0 1 0;          δ (q0, 0) = (R)
    ↑
    q0
;0 1 0 1 0;          δ (q0, 1) = (R)
     ↑
     q0
;0 1 0 1 0;          δ (q0, 0) = (R)
      ↑
      q0
```

```
;0 1 0 1 0;          δ (q0, ;) = (q1 L)
    ↑
   q1
;0 1 0 1 0;          δ (q1, 0) = (L)
   ↑
  q1
;0 1 0 1 0;          δ (q1, 1) = (q2 L)
  ↑
 q2
;0 1 1 1 0;          δ (q2, 0) = (1 L)
  ↑
 q2
;0 0 1 1 0;          δ (q2, 1) = (0 L)
 ↑
q2
;1 0 1 1 0;          δ (q2, 0) = (1 L)
↑
q2
;1 0 1 1 0;          δ (q2, ;) = (q3 N)
↑
q3
```

Thus, the TM generates the 2's complement, for the input string '01010', as '10110'.

Example 4.4 Design a TM that replaces all occurrences of '111' by '101' from a sequence of 0's and 1's.

Solution The required TM is a typical sequence detector machine (Mealy machine) that we have studied in Chapter 2. Let the initial configuration of the TM be as shown in Fig. 4.7(a).

The initial state is assumed to be q_0. The string is delimited at the rightmost end by a semicolon ';', which is used as the end-marker. The TM starts reading from the leftmost symbol in the input string.

Table 4.6 SFM for a TM that replaces each occurrence of '111' by '101'

S \ I	0	1	;
q_0	R	q_1R	q_5N
q_1	q_0R	q_2R	q_5N
q_2	q_0R	q_3L	q_5N
q_3	—	$0q_4R$	—
q_4	—	q_0R	—
q_5	—	—	—

The SFM for the required TM, which converts each occurrence of '111' by '101' is as given in Table 4.6.

For this TM, we have

$$I = \{0, 1, ;\}$$
$$S = \{q_0, q_1, q_2, q_3, q_4, q_5 = \text{halt}\}$$
$$D = \{L, R, N\}$$

We see from Table 4.6 that states q_0, q_1, and q_2 collectively identify the sequence '111'. Once the sequence is identified, state q_2 makes a transition to state q_3 and points to the middle 1, which is replaced by 0 in state q_3; thus, the string is replaced by '101'. Once the replacement is done, the TM moves to state q_4, which resets to state q_0 for repeating the same process again to search for another sequence of '111' that may exist.

The TG for the TM is shown in Fig. 4.7(b).

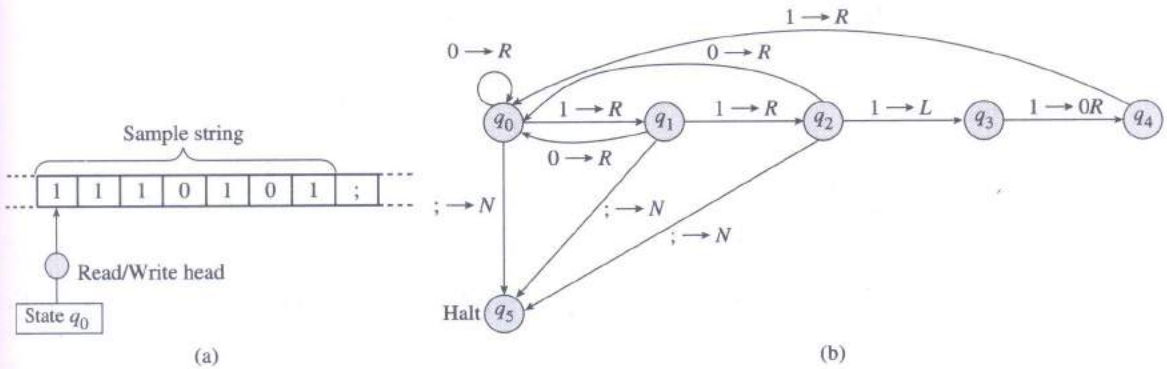


Figure 4.7 TM that replaces all occurrences of '111' by '101' (a) Initial configuration (b) TG for TM that replaces each occurrence of '111' by '101'

Simulation

Let us simulate the working of the TM that we have constructed for the input string '11110111'.

1 1 1 1 0 1 1 1 ; ↑ q_0	initial configuration
1 1 1 1 0 1 1 1 ; ↑ q_1	$\delta\,(q_0, 1) = (q_1\, R)$
1 1 1 1 0 1 1 1 ; ↑ q_2	$\delta\,(q_1, 1) = (q_2\, R)$
1 1 1 1 0 1 1 1 ; ↑ q_3	$\delta\,(q_2, 1) = (q_3\, L)$
1 0 1 1 0 1 1 1 ; ↑ q_4	$\delta\,(q_3, 1) = (0\, q_4\, R)$
1 0 1 1 0 1 1 1 ; ↑ q_0	$\delta\,(q_4, 1) = (q_0\, R)$
1 0 1 1 0 1 1 1 ; ↑ q_1	$\delta\,(q_0, 1) = (q_1\, R)$
1 0 1 1 0 1 1 1 ; ↑ q_0	$\delta\,(q_1, 0) = (q_0\, R)$
1 0 1 1 0 1 1 1 ; ↑ q_1	$\delta\,(q_0, 1) = (q_1\, R)$
1 0 1 1 0 1 1 1 ; ↑ q_2	$\delta\,(q_1, 1) = (q_2\, R)$
1 0 1 1 0 1 1 1 ; ↑ q_3	$\delta\,(q_2, 1) = (q_3\, L)$

$$
\begin{array}{ll}
1\,0\,1\,1\,0\,1\,0\,1\,; & \delta\,(q_3, 1) = (0\,q_4\,R) \\
\quad\uparrow\;\; q_4 & \\
1\,0\,1\,1\,0\,1\,0\,1\,; & \delta\,(q_4, 1) = (q_0\,R) \\
\quad\uparrow\;\; q_0 & \\
1\,0\,1\,1\,0\,1\,0\,1\,; & \delta\,(q_0, ;) = (q_5\,N) \\
\quad\uparrow\;\; q_5 &
\end{array}
$$

Thus, the TM has converted both the occurrences of '111' from the input string to '101'.

***Note*:** As we know, the particular problem we have discussed can also be solved using Mealy machine. Since the TM is a more powerful machine, it can always solve problems that are solvable using an FSM.

Example 4.5 Design a TM that recognizes words of the form $0^n\,1^n$ for $n \geq 0$.

Solution Let us assume the initial configuration as shown in Fig. 4.8(a). The transition graph for the TM is shown in Fig. 4.8(b).

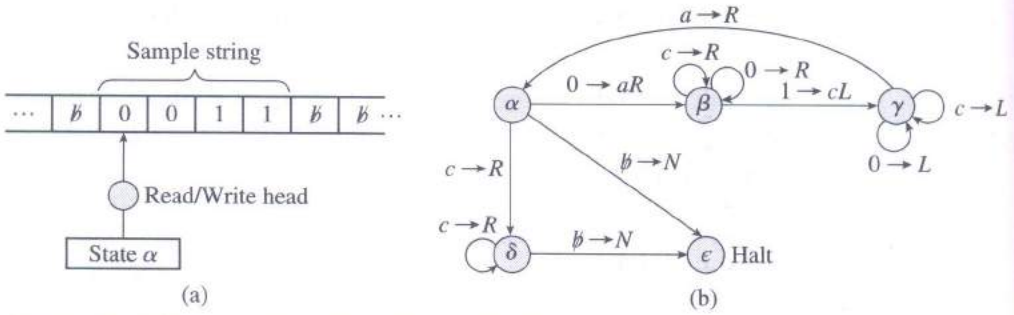


Figure 4.8 TM that recognizes strings of the form $0^n\,1^n$ for $n \geq 0$ (a) Initial configuration (b) Transition graph

Algorithm

1. Replace the first 0 by some other symbol, say a, and move towards the right till you reach the first 1.
2. Replace this first 1 by some other symbol, say c, and move towards the left till you reach the 0 immediately next to the one that was previously replaced by a.
3. Repeat the procedure till all 0's are replaced by a's.

For this TM, we have

$$
\begin{aligned}
I &= \{0, 1, a, c, \not b\} \\
S &= \{\alpha, \beta, \gamma, \delta, \epsilon = \text{halt}\} \\
D &= \{L, R, N\}
\end{aligned}
$$

The SFM for the TM is as shown in Table 4.7.

Table 4.7 SFM for a TM that recognizes string $0^n 1^n$

S \ I	0	1	a	c	b
α	$a\beta R$	—	—	δR	cN (accept)
β	R	$c\gamma L$	—	R	—
γ	L	—	αR	L	—
δ	—	—	—	R	cN (accept)
c	—	—	—	—	—

Here, α is the initial state of the machine and c is the halt state. The transition diagram is drawn as shown in Fig. 4.8(b).

Simulation

1. Let us simulate the working of the machine for the string '0011'.

$0\ 0\ 1\ 1\ \flat$ $\uparrow$ α	initial configuration
$a\ 0\ 1\ 1\ \flat$ $\uparrow$ β	$\delta\ (\alpha, 0) = (a\ \beta\ R)$
$a\ 0\ 1\ 1\ \flat$ $\uparrow$ β	$\delta\ (\beta, 0) = (R)$
$a\ 0\ c\ 1\ \flat$ $\uparrow$ γ	$\delta\ (\beta, 1) = (c\ \gamma\ L)$
$a\ 0\ c\ 1\ \flat$ $\uparrow$ γ	$\delta\ (\gamma, 0) = (L)$
$a\ 0\ c\ 1\ \flat$ $\uparrow$ α	$\delta\ (\gamma, a) = (\alpha\ R)$
$a\ a\ c\ 1\ \flat$ $\uparrow$ β	$\delta\ (\alpha, 0) = (a\ \beta\ R)$
$a\ a\ c\ 1\ \flat$ $\uparrow$ β	$\delta\ (\beta, c) = (R)$
$a\ a\ c\ c\ \flat$ $\uparrow$ γ	$\delta\ (\beta, 1) = (c\ \gamma\ L)$
$a\ a\ c\ c\ \flat$ $\uparrow$ γ	$\delta\ (\gamma, c) = (L)$
$a\ a\ c\ c\ \flat$ $\uparrow$ α	$\delta\ (\gamma, a) = (\alpha\ R)$

$$\begin{array}{ll} a\ a\ c\ c\ \not b & \delta\ (\alpha, c) = (\delta\ R) \\ \quad\uparrow & \\ \quad\delta & \\ a\ a\ c\ c\ \not b & \delta\ (\delta, c) = (R) \\ \quad\quad\uparrow & \\ \quad\quad\delta & \\ a\ a\ c\ c\ \not b\ . & \delta\ (\delta, \not b) = (\epsilon\ N) \\ \quad\quad\quad\uparrow & \\ \quad\quad\quad\epsilon & \end{array}$$

Thus, the string '0011' is accepted by the TM.

2. Let us simulate the working of this TM for the empty string.

$$\begin{array}{ll} \not b & \text{initial configuration} \\ \uparrow & \\ \alpha & \\ \not b & \delta\ (\alpha, \not b) = (\epsilon\ N) \\ \uparrow & \\ \epsilon & \end{array}$$

Empty string is accepted by the machine as it fits the pattern $0^n\ 1^n$ for $n = 0$. In case of empty string, the machine starts reading the trailing blank character, $\not b$.

Example 4.6 Design a TM that recognizes strings containing equal number of 0's and 1's.

Solution Let the initial configuration of the machine be as shown in Fig. 4.9(a).

Algorithm

1. Starting from the initial state, if you reach the first 0 replace it by '*', and move right till you reach the first 1; replace that also by the same symbol, '*'.
2. If you reach 1 first, then replace that with the symbol '*', and move right till you reach the first 0; replace that also by the same symbol, '*'.
3. Repeat the procedure till you finish reading all 1's and 0's. If any 1 or 0 remains, there must be an error, and the machine should reject the input.

For this TM, we have

$I = \{0, 1, ;, *, T, F\}$, where T = accepted; and F = rejected
$S = \{q_0, q_1, q_2, q_3, q_4 = \text{halt}\}$
$D = \{L, R, N\}$

The SFM for the TM is as shown in Table 4.8.

The initial state q_0 makes transition to q_1 after replacing the input symbol 0 by *. State q_1 hence carries the responsibility of finding the matching symbol 1. State q_3 is reached from q_0 on finding the symbol 1; hence, q_3 needs to find the matching 0 so that the number of 0's and 1's are same. The output symbol T is generated if the number of 0's and 1's are equal in the input string; else output F is generated.

Table 4.8 SFM for a TM that accepts strings containing equal number of 0's and 1's

S \ I	0	1	;	*	T	F
q_0	$*q_1R$	$*q_3R$	Tq_4N	R	—	—
q_1	R	$*q_2L$	Fq_4N	R	—	—
q_2	L	L	q_0R	L	—	—
q_3	$*q_2L$	R	Fq_4N	R	—	—
q_4	—	—	—	—	—	—

The transition diagram for the TM is as shown in Fig. 4.9(b).

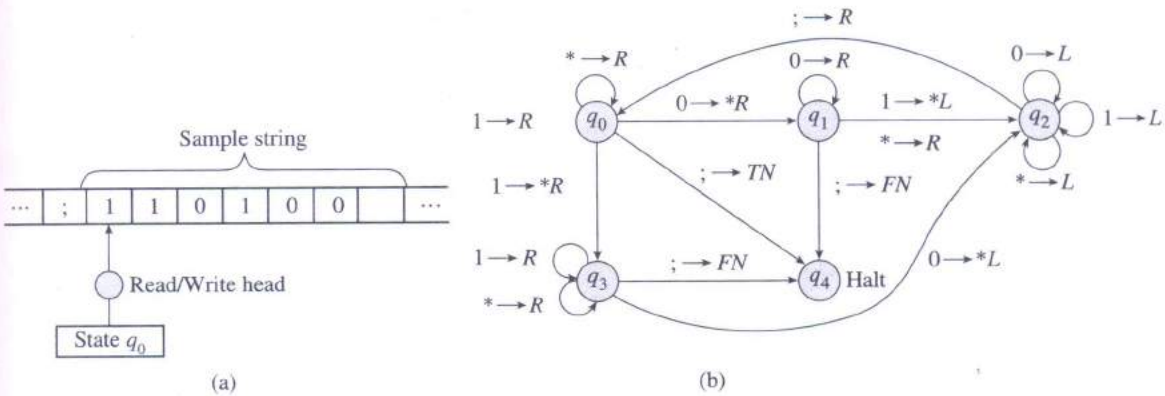


Figure 4.9 TM that recognizes strings containing equal number of 0's and 1's (a) Initial configuration (b) TG for TM that accepts all strings containing equal number of 0's and 1's

Simulation

1. Let us simulate the working of the TM for the string '110100'.

; 1 1 0 1 0 0 ; initial configuration
↑
q_0

; * 1 0 1 0 0 ; $\delta(q_0, 1) = (* q_3 R)$
↑
q_3

; * 1 0 1 0 0 ; $\delta(q_3, 1) = (R)$
↑
q_3

; * 1 * 1 0 0 ; $\delta(q_3, 0) = (* q_2 L)$
↑
q_2

; * 1 * 1 0 0 ; $\delta(q_2, 1) = (L)$
↑
q_2

Example 4.7 Design a TM that recognizes binary palindromes.

Solution Let the initial configuration of the TM be as shown in Fig. 4.10(a).

In this case, we assume that the head initially points to the left end-marker ';', that is, just prior to the actual start of the input. This assumption is the basis for the algorithm we write. Note that if we make a different assumption—for example, we may assume that the head points to the first input symbol—the algorithm as well as the SFM that is generated for the TM will be different.

Algorithm

1. Read the first symbol, which may be 0 or 1. Replace it by some other symbol, say ';'.
2. Move towards the right to find the right end of the sequence, that is, the right end-marker ';'. Then, move one position to the left to point to the last symbol.
3. Read the last symbol. If it is equal to the symbol that we read at the other end, then replace it with ';', and continue the process.
4. If it is not equal, then the sequence is not a palindrome sequence.

For this TM, we have

$$I = \{0, 1, ;, F, T\}$$
$$S = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6 = \text{halt}\}$$
$$D = \{L, R, N\}$$

Here, T indicates that the input sequence is a palindrome and is therefore accepted by the TM; and F indicates that the input is not a palindrome sequence and is therefore rejected by the TM.

The SFM for the TM is as shown in Table 4.9.

Table 4.9 SFM for a TM that recognizes binary palindromes

S \ I	0	1	;	F	T
q_0	L	L	q_1R	—	—
q_1	$;q_2R$	$;q_4R$	Tq_6N	—	—
q_2	R	R	q_3L	—	—
q_3	$;q_0L$	Fq_6N	Tq_6N	—	—
q_4	R	R	q_5L	—	—
q_5	Fq_6N	$;q_0L$	Tq_6N	—	—
q_6	—	—	—	—	—

The transition graph is drawn as shown in Fig. 4.10(b).

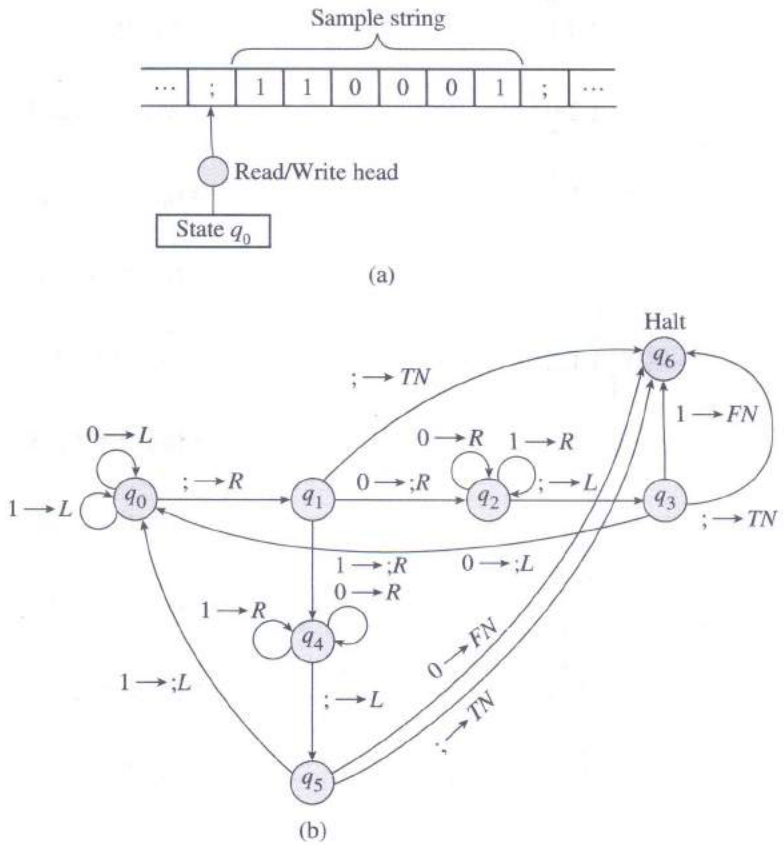


Figure 4.10 TM that recognizes binary palindromes (a) Initial configuration (b) TG for TM that recognizes binary palindromes

Simulation

1. Let us simulate the working of the TM that we have constructed for the string '0110', which is a binary palindrome sequence of even length.

; 0 1 1 0 ; ↑ q_0	initial configuration
; 0 1 1 0 ; ↑ q_1	$\delta\ (q_0, ;) = (q_1\ R)$
; ; 1 1 0 ; ↑ q_2	$\delta\ (q_1, 0) = (;\ q_2\ R)$
; ; 1 1 0 ; ↑ q_2	$\delta\ (q_2, 1) = (R)$

; ; 1 ;
  ↑
  q_3
; ; F ;
  ↑
  q_6

$\delta(q_2, ;) = (q_3\ L)$

$\delta(q_3, 1) = (F\ q_6\ N)$

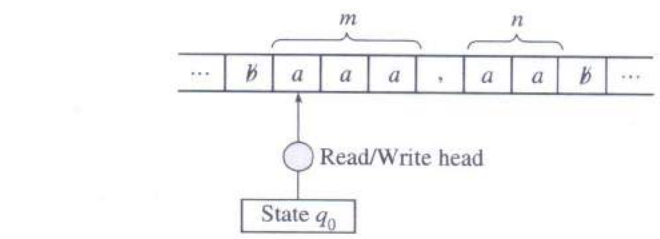
The output F indicates that the string '01' is not a palindrome string, and hence, is rejected by the TM.

Example 4.8 Design a TM, which compares two positive integers m and n and produces output G_t if $m > n$; L_t if $m < n$; and E_q if $m = n$.

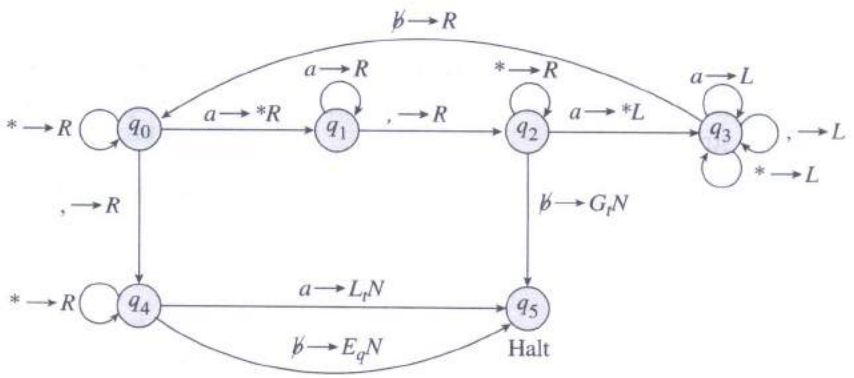
Solution This TM is a symbol manipulation system. For this, we need to represent the numbers in the unary format. If we consider $\Sigma = \{a\}$, then the numbers can be represented using that many a's. For example, '2' in unary format can be written as 'aa' and '5' can be written as '$aaaaa$'.

The initial configuration of the TM is assumed as shown in Fig. 4.11(a). Notice that the two numbers, m and n, are separated by a punctuation symbol ',' (comma).

The transition diagram for the required TM is shown in Fig. 4.11(b).



(a)



(b)

Figure 4.11 TM that compares two positive integers (a) Initial configuration (b) Transition graph

Algorithm

1. Replace one a from m by '*'
2. Move right till you get the first a of n, replace it with '*'
3. If no a is remaining of n that is to be replaced by '*' then $m > n$
4. Move left to find the next a of m; if found repeat the aforementioned two steps
5. If no a is remaining of m then search if any a is remaining that of n, if yes then, $m < n$; if no then, $m = n$

In short, keep replacing each a of m and n by '*', until both or one of them gets fully replaced with *'s. If finally both are totally replaced by *'s, then they must be equal; otherwise, the one with some remaining a's must be greater than the other.

For this TM, we have

$$I = \{a, , , *, \not b, G_t, L_t, E_q\}$$
$$S = \{q_0, q_1, q_2, q_3, q_4, q_5 = \text{halt}\}$$
$$D = \{L, R, N\}$$

The SFM for the TM is shown in Table 4.10.

Table 4.10 SFM for a TM that compares two positive integers

S \ I	a	,	*	$\not b$	G	L	E
q_0	$*q_1R$	q_4R	R	—	—	—	—
q_1	R	q_2R	—	—	—	—	—
q_2	$*q_3L$	—	R	G_tq_5N	—	—	—
q_3	L	L	L	q_0R	—	—	—
q_4	L_tq_5N	—	R	E_qq_5N	—	—	—
q_5	—	—	—	—	—	—	—

Initially, the TM is in state q_0, which replaces one a from m by '*', and makes a transition to a new state q_1. State q_1 is responsible for finding the beginning of the next number n; on accomplishing this, q_1 makes transition to q_2. State q_2 replaces one a from n by '*' to match the one that we have replaced from m earlier. Essentially, we subtract 1 from both the numbers.

Now, if the TM replaces one a from m by a '*', but there is no a left to be replaced in n, then it means that in state q_2 the TM does not find any a; it instead read[s] a blank character $\not b$. In this case, the TM generates the output G_t to indicate that the first n[umber] than the second number n.

Let us consider another scenario. If there is no a left in the numbe[r] reads comma ',' while in state q_0; then, it changes to state q_4. State q_4 [checks] whether there is any a left in the second number n. If the TM finds an [a]

$\not b$, a $\not b$ $\quad\quad \delta(q_0, ,) = (q_4 R)$
↑
q_4

$\not b$, L_t $\not b$ $\quad\quad \delta(q_4, a) = (L_t q_5 N)$
↑
q_5

The output L_t indicates that $m = 0$ is less than $n = 1$.

3. Let us consider the case when both numbers are equal: let $m = n = 1 =$ 'a'.

$\not b$ a , a $\not b$ $\quad\quad$ initial configuration
↑
q_0

$\not b$ * , a $\not b$ $\quad\quad \delta(q_0, a) = (* q_1 R)$
↑
q_1

$\not b$ * , a $\not b$ $\quad\quad \delta(q_1, ,) = (q_2 R)$
↑
q_2

$\not b$ * , * $\not b$ $\quad\quad \delta(q_2, a) = (* q_3 L)$
↑
q_3

$\not b$ * , * $\not b$ $\quad\quad \delta(q_3, ,) = (L)$
↑
q_3

$\not b$ * , * $\not b$ $\quad\quad \delta(q_3, *) = (L)$
↑
q_3

$\not b$ * , * $\not b$ $\quad\quad \delta(q_3, \not b) = (q_0 R)$
↑
q_0

$\not b$ * , * $\not b$ $\quad\quad \delta(q_0, *) = (R)$
↑
q_0

$\not b$ * , * $\not b$ $\quad\quad \delta(q_0, ,) = (q_4 R)$
↑
q_4

$\not b$ * , * $\not b$ $\quad\quad \delta(q_4, *) = (R)$
↑
q_4

$\not b$ * , * E_q $\quad\quad \delta(q_4, \not b) = (E_q q_5 N)$
↑
q_5

Thus, m and n are equal, which is indicated by the output E_q.

Example 4.9 Design a TM that performs the addition of two unary numbers.

Solution Let us consider the pair of numbers expressed in unary form using a symbol 'a', that is, a number x is represented by x consecutive a's. Let the initial configuration of the machine be as shown in Fig. 4.12(a). As both the unary numbers are represented using the letter 'a', they are separated by the delimiter 'c' on the tape. The initial state of the TM is α, as specified in Fig. 4.12(a). The TG for this TM is shown in Fig. 4.12(b).

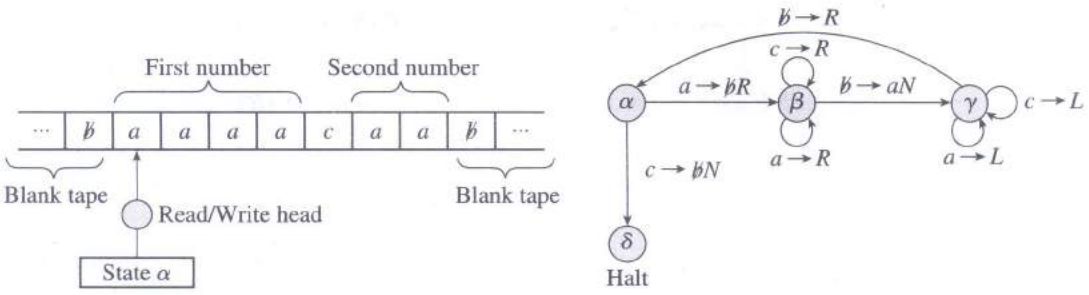


Figure 4.12 TM that performs addition of two unary numbers (a) Initial configuration (b) TG for TM that adds two unary numbers

Algorithm

Addition is nothing but concatenation. This is exactly similar to what is taught in pre-primary level mathematics. To achieve '$m + n$', let us represent m and n by some object such as blue balls and then perform aggregation. Put m blue balls into a bucket and put n blue balls into the same bucket; then count the total number of blue balls in the bucket. It gives the value after addition.

Replace each a from the first number, that is, the string of a's on the left side of c by a blank character b and add one a after the string of a's representing the second number. In this way, after completion of addition, the string of a's equal to the result of addition resides on the right side of c. At the end, replace the c with a blank character b.

Table 4.11 SFM for a TM that adds two unary numbers

I / S	a	b	c
α	bβR	—	bδN
β	R	aγN	R
γ	L	αR	L

For this TM, we have

$$I = \{a, b, c\}$$
$$S = \{\alpha, \beta, \gamma, \delta = \text{halt}\}$$
$$D = \{L, R, N\}$$

The simplified functional matrix for the TM is given in Table 4.11.

Note: In this example, entries for δ state are not shown because they are null entries, as it is a halt state. In all previous examples, we have used '-' (hyphens) to indicate them as null/unspecified entries.

Simulation

Let us simulate the working of the TM for the following example:

First number = 3 = '*aaa*'

Second number = 2 = '*aa*'

... b a a a c a a b ... initial configuration

↑

α

$\ldots \not b \not b a a c a a \not b \ldots$ $\uparrow$ β	$\delta(\alpha, a) = (\not b\ \beta\ R)$
$\ldots \not b \not b a a c a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b a a c a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b a a c a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, c) = (R)$
$\ldots \not b \not b a a c a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b a a c a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\beta, \not b) = (a\ \gamma\ N)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, c) = (L)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b a a c a a a \not b \ldots$ $\uparrow$ α	$\delta(\gamma, \not b) = (\alpha\ R)$
$\ldots \not b \not b \not b a c a a a \not b \ldots$ $\uparrow$ β	$\delta(\alpha, a) = (\not b\ \beta\ R)$
$\ldots \not b \not b \not b a c a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b a c a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, c) = (R)$
$\ldots \not b \not b \not b a c a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$

$\ldots \not b \not b \not b a c a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b a c a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\beta, \not b) = (a\ \gamma\ N)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, c) = (L)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$
$\ldots \not b \not b \not b a c a a a a \not b \ldots$ $\uparrow$ α	$\delta(\gamma, \not b) = (\alpha\ R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ β	$\delta(\alpha, a) = (\not b\ \beta\ R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, c) = (R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ β	$\delta(\beta, a) = (R)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\beta, \not b) = (a\ \gamma\ N)$
$\ldots \not b \not b \not b \not b c a a a a \not b \ldots$ $\uparrow$ γ	$\delta(\gamma, a) = (L)$

Tape configuration	Transition
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on 4th a, state γ)	$\delta(\gamma, a) = (L)$
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on 3rd a, state γ)	$\delta(\gamma, a) = (L)$
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on 2nd a, state γ)	$\delta(\gamma, a) = (L)$
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on 1st a, state γ)	$\delta(\gamma, a) = (L)$
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on c, state γ)	$\delta(\gamma, c) = (L)$
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on 4th b̸, state γ)	$\delta(\gamma, b̸) = (\alpha\ R)$
… b̸ b̸ b̸ b̸ c a a a a a b̸ … (head on c, state α)	$\delta(\alpha, c) = (b̸\ \delta\ N)$
… b̸ b̸ b̸ b̸ b̸ a a a a a b̸ … (head on 5th b̸, state δ)	

Thus, the addition of the two numbers '3 + 2' is completed, and that is indicated by the five *a*'s that are left on the tape before the machine halts.

Example 4.10 Design a TM that multiplies two unary numbers.

Solution Multiplication, as we know, is repetitive addition of multiplicand to itself. We have already discussed unary addition using a TM in the previous example.

If *m* is the multiplier and *n* is the multiplicand, then $n \times m$ can be viewed as:

$$n \times m = n + n + \ldots m \text{ number of times}$$

Thus, we consider addition as the concatenation of *m* number of *n*'s.

Let us consider the initial configuration of the TM as shown in Fig. 4.13(a). The multiplier and multiplicand are both unary representations of the decimal numbers. They are represented here using symbol 1; for example, the decimal number 2 in unary format is written as '11'. The rest of the tape is assumed to be filled with all 0's instead of b̸'s. The

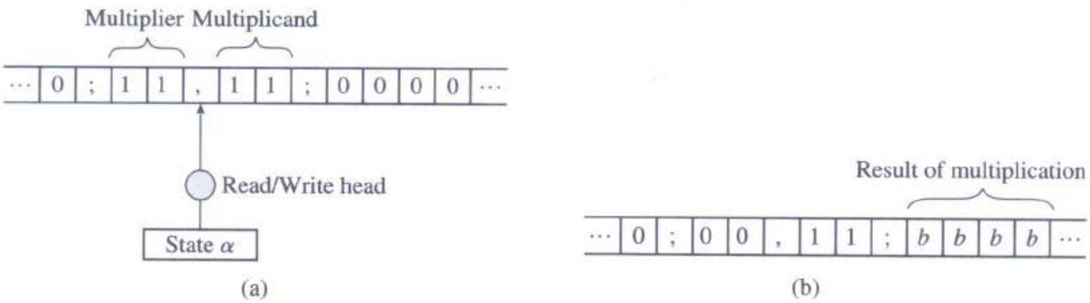


Figure 4.13 TM that multiplies two unary numbers (a) Initial configuration (b) Final configuration of TM (for the operation $2 \times 2 = 4$).

multiplier and multiplicand are separated by a comma ',' and delimited at both the ends by semicolons ';'. The head points to the separator symbol ',' initially.

The result of the multiplication is written after the right end-marker semicolon (;). We have an example final configuration of the TM for a sample multiplication (2×2), as shown in Fig. 4.13(b). Observe how the result is written and where it is written onto the tape. The result is also represented in unary format but using the symbol b. The multiplier at the end gets replaced with 0's. The algorithm thus is a destructive one, as it modifies the parameters sent to it (in this case, the parameter modified is the multiplier).

Algorithm

1. Replace one '1' of the multiplier by '0', that is, subtract one from the multiplier.
2. To add the multiplicand to the result area, that is, beyond the right end-marker ';' replace one '1' of the multiplicand by some symbol, say 'a'
3. Find the end of the result where '0' can be found, replace it with symbol 'b' (result is represented by all 'b's)
4. Repeat the aforementioned steps 2 and 3 till all the '1's in the multiplicand are all replaced by 'a's
5. The multiplicand is added once to the result area; hence, reset the multiplicand to all '1's again and repeat the steps starting from 1.
6. Stop when all the '1's in the multiplier are replaced by all '0's

For this TM, we have:

$I = \{0, 1, a, b, ;, ,\}$
$S = \{\alpha, \beta, \gamma, \delta, \epsilon, f = \text{halt}\}$
$D = \{L, R, N\}$

The SFM for the TM is shown in Table 4.12.

Table 4.12 SFM for a TM that multiplies two unary numbers

S \ I	0	1	a	b	;	,
α	L	$0\beta R$	—	—	ϕN	L
β	R	aR	—	—	γL	R
γ	—	—	$1\,\delta\,R$	—	R	L
δ	$b\epsilon L$	R	—	R	R	—
ϵ	L	L	$1\,\delta\,R$	L	L	αN
ϕ	—	—	—	—	—	—

In state α, the TM starts by replacing one '1' from the multiplier by 0, that is, reducing the multiplier by 1. In state β, the TM moves right by replacing all 1's from the multiplicand by a's, and changes the state to γ once the right end-marker ';' is found. States γ,

δ, and ϵ are responsible for concatenating the multiplicand to the right end once. In this process, all the 1's in the multiplicand that were replaced by all a's are replaced again by 1's. The concatenated result is represented in unary form using the symbol b. The halt state f is entered once all the multiplier 1's are replaced by 0's, which means that the TM halts when the multiplicand is added (concatenated) to itself multiplier number of times.

Let us see a simulation of this TM for a sample multiplication.

Simulation

Let us simulate the working of this TM for the following:

Multiplier = $m = 2 =$ '11'

Multiplicand = $n = 2 =$ '11'.

$\ldots 0\,;1\,1\,,1\,1\,;0\,0\,0\,0\ldots$ initial configuration

$\uparrow$

α

$\ldots 0\,;1\,1\,,1\,1\,;0\,0\,0\,0\ldots$ $\delta\,(\alpha, ,) = (L)$

$\uparrow$

α

$\ldots 0\,;1\,0\,,1\,1\,;0\,0\,0\,0\ldots$ $\delta\,(\alpha, 1) = (0\;\beta\;R)$

$\uparrow$

β

(Decremented multiplier by 1)

$\ldots 0\,;1\,0\,,1\,1\,;0\,0\,0\,0\ldots$ $\delta\,(\beta, ,) = (R)$

$\uparrow$

β

(Concatenation of the multiplicand to the result area begins from here...)

$\ldots 0\,;1\,0\,,a\,1\,;0\,0\,0\,0\ldots$ $\delta\,(\beta, 1) = (a\;R)$

$\uparrow$

β

(Replacing all '1's by 'a's from multiplicand is for performing concatenation of the multiplicand to the result)

$\ldots 0\,;1\,0\,,a\,a\,;0\,0\,0\,0\ldots$ $\delta\,(\beta, 1) = (a\;R)$

$\uparrow$

β

$\ldots 0\,;1\,0\,,a\,a\,;0\,0\,0\,0\ldots$ $\delta\,(\beta, ;) = (\gamma\;L)$

$\uparrow$

γ

$\ldots 0\,;1\,0\,,a\,1\,;0\,0\,0\,0\ldots$ $\delta\,(\gamma, a) = (1\;\delta\;R)$

$\uparrow$

δ

(This step again replaces the 'a's in the multiplicand with 1; therefore, after concatenation in the result area, the multiplicand will be restored as all '1's)

$\ldots 0\,;1\,0\,,a\,1\,;0\,0\,0\,0\ldots$ $\delta\,(\delta, ;) = (R)$

$\uparrow$

δ

$\ldots 0\,;1\,0\,,a\,1\,;b\,0\,0\,0\ldots$ $\delta\,(\delta, 0) = (b\;\epsilon\;L)$

$\uparrow$

ϵ

$\ldots 0\,;1\,0\,,a\,1\,;b\,0\,0\,0\ldots$ $\delta\,(\epsilon, ;) = (L)$

$\uparrow$

ϵ

Note: If we want to design a non-destructive TM, we must ensure that the original arguments (or parameters) are kept intact. Therefore, we must construct the TM such that before entering into the halt state, it replaces all '0's in the multiplier by all '1's again.

Thus, we see that multiplication, which is not possible for an FSM, is achieved using a TM.

Example 4.11 Design a TM that finds the greatest common divisor (GCD) of two given numbers.

Solution Let the initial configuration of the TM be as shown in Fig. 4.14(a). We observe that the two numbers—in this example, we consider the numbers, 4 and 2—are stored onto the tape without any separator in between; further, we observe that both the numbers are represented in unary format using the symbol 1.

The initial state of the TM is α, and the head points to the last 1 of the unary representation of the first number among the given pair of numbers. As the sample pair is 4 and 2, we observe that the numbers in the tape are '1111' and '11'; and hence, the head points to the last 1 in '1111'.

Algorithm

Examine the numbers on the tape to find which of the two is larger. This is achieved by a repetitive subtraction process: First change one 1 in the first number to a; then change one 1 in the second number to b. Then return to the first number to change another 1 to a, and so on. This effectively subtracts the smaller number from the larger one.

If x is the smaller number and y is the larger number, then after the aforementioned processing, the pair x, y is changed to x and 'y - x'. This process is recursively carried out again on the newly-obtained pair. When the machine halts, it leaves one number in the pair as 0, and the other as the GCD.

The SFM for the TM is as shown in Table 4.13.

Table 4.13 SFM for a TM that finds the GCD of two given numbers

S \ I	0	1	a	b
α	δR	$a\beta N$	L	L
β	γL	$b\alpha N$	R	R
γ	αR	αR	$1L$	$0L$
δ	*Halt*	αL	$0R$	$1R$

For this TM, we have

$$I = \{0, 1, a, b\}$$
$$S = \{\alpha, \beta, \gamma, \delta\}$$

Halt state is not explicitly mentioned. One can introduce it if required, as we did for the examples so far. Wherever there is an entry 'Halt' in the transition Table 4.13, one can introduce a transition to such a halt state. The change is done to showcase a variation in the representation.

$$D = \{L, R, N\}$$

The transition diagram is as shown in Fig. 4.14(b).

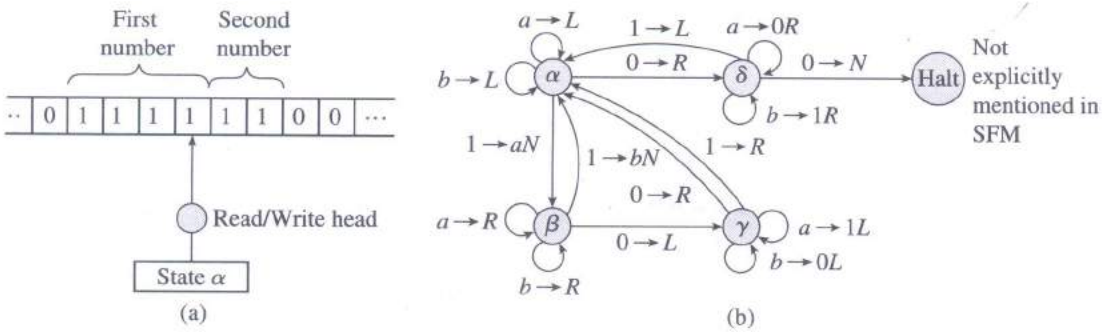


Figure 4.14 TM that finds the GCD of two given numbers (a) Initial configuration (b) TG for TM that finds GCD of two given numbers

Simulation

Let us simulate the working of the TM that we have constructed for the input numbers:

$x = 4 =$ '1111'
$y = 2 =$ '11'

... 0 1 1 1 1 1 1 0 ...	initial configuration
↑ α	
... 0 1 1 1 a 1 1 0 ...	$\delta(\alpha, 1) = (a\ \beta\ N)$
↑ β	
... 0 1 1 1 a 1 1 0 ...	$\delta(\beta, a) = (R)$
↑ β	
... 0 1 1 1 a b 1 0 ...	$\delta(\beta, 1) = (b\ \alpha\ N)$
↑ α	
... 0 1 1 1 a b 1 0 ...	$\delta(\alpha, b) = (L)$
↑ α	
... 0 1 1 1 a b 1 0 ...	$\delta(\alpha, a) = (L)$
↑ α	
... 0 1 1 a a b 1 0 ...	$\delta(\alpha, 1) = (a\ \beta\ N)$
↑ β	
... 0 1 1 a a b 1 0 ...	$\delta(\beta, a) = (R)$
↑ β	
... 0 1 1 a a b 1 0 ...	$\delta(\beta, a) = (R)$
↑ β	
... 0 1 1 a a b 1 0 ...	$\delta(\beta, b) = (R)$
↑ β	

$\ldots 0\,a\,a\,b\,b\,0\,0\,0 \ldots$ $\quad \delta(\alpha, a) = (L)$

$\ldots 0\,a\,a\,b\,b\,0\,0\,0 \ldots$ $\quad \delta(\alpha, 0) = (\delta\, R)$

$\ldots 0\,0\,a\,b\,b\,0\,0\,0 \ldots$ $\quad \delta(\delta, a) = (0\, R)$

$\ldots 0\,0\,0\,b\,b\,0\,0\,0 \ldots$ $\quad \delta(\delta, a) = (0\, R)$

$\ldots 0\,0\,0\,1\,b\,0\,0\,0 \ldots$ $\quad \delta(\delta, b) = (1\, R)$

$\ldots 0\,0\,0\,1\,1\,0\,0\,0 \ldots$ $\quad \delta(\delta, b) = (1\, R)$

GCD

$\ldots 0\,0\,0\,1\,1\,0\,0\,0 \ldots$ $\quad \delta(\delta, 0) =$ Halt

therefore, GCD = "1 1" = 2

As the second number of the newly-obtained pair is 0, the GCD of 4 and 2 is represented by the number of 1's remaining on the tape. Hence, the answer is 2.

Example 4.12 Design a TM that divides one number by the other, and finds the result of the division as well as the remainder if any.

Solution Let us represent the numbers in the unary format using the symbol 1. For example, '3' can be represented as '111' in unary format. Let us assume the initial configuration of the TM as shown in the Fig. 4.15.

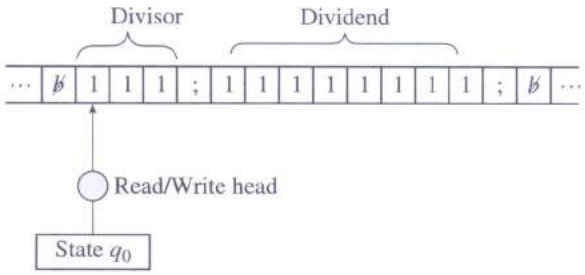


Figure 4.15 Initial configuration of a TM that performs number division

We observe that the initial state of the TM is q_0, and that the divisor and dividend are separated by a comma ',' . The machine head initially points to the start of the divisor.

After the division is performed, when the machine halts, the result of the division, which is also represented in unary format using the symbol 1, is written onto the tape immediately after the right end-marker ';'. The number of a's in the divisor area represents the remainder of the division.

Algorithm

Division is repetitive subtraction of the divisor from the dividend. In the previous example, we have used repetitive subtraction to find the GCD of two numbers. Similarly, in Example 4.8, we designed a TM that compares two positive integers using subtraction. We now use a similar algorithm for division.

Replace one 1 of the divisor by a; and for each a in the divisor, replace one 1 of the dividend by b. Repeat the process until the whole divisor is subtracted from the dividend. Write 1 at the right end indicating that one cycle of subtraction is complete.

Iteratively, subtract the divisor again from the dividend till you have exhausted the dividend completely. At the end of every iteration, write 1 at the right end; after all the iterations are completed, the number of 1's at the right end represents the result of the division.

If the entire divisor cannot be subtracted from the dividend, then the number of a's that replaces the 1's from the divisor represents the remainder of the division.

The SFM for the TM is as shown in Table 4.14.

Table 4.14 SFM for a TM that performs number division

S \ I	1	,	;	⊬	a	b
q_0	aq_1R	q_4R	—	—	—	—
q_1	R	q_2R	—	—	—	—
q_2	bq_3L	—	q_7L	—	—	R
q_3	L	L	—	—	q_0R	L
q_4	R	—	R	$1q_5R$	—	R
q_5	L	q_6L	L	—	—	L
q_6	—	—	—	q_0R	$1L$	—
q_7	—	q_8L	—	—	—	L
q_8	L	—	—	—	$1q_9N$	—
q_9	—	—	—	—	—	—

For this TM, we have:

$I = \{1, , , ; , ⊬, a, b\}$
$S = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9 = \text{halt}\}$
$D = \{L, R, N\}$

Simulation

Let us simulate the working of TM for

Divisor = 3 = '111'
Dividend = 8 = '11111111'

... ⊬ 1 1 1 , 1 1 1 1 1 1 1 1 ; ⊬ ... initial configuration
↑
q_0

... ⊬ a 1 1 , 1 1 1 1 1 1 1 1 ; ⊬ ... $\delta(q_0, 1) = (a\ q_1\ R)$
↑
q_1

(Subtraction of the divisor from dividend begins)

The number of *a*'s in the divisor area represents the remainder, and the number of 1's after the right end-marker ';' represents the result of the division. Hence, the result obtained when 8 is divided by 3 is 2, and the remainder is 2.

Example 4.13 Design a TM to find the value of $\log_2(n)$, where n is any binary number and a perfect power of 2.

Solution It is given that n is a binary number and a perfect power of 2.
We know:

$2^0 = 1 = 1$ (binary)
$2^1 = 2 = 10$ (binary)
$2^2 = 4 = 100$ (binary)
$2^3 = 8 = 1000$ (binary)

From the aforementioned listing, we have

$\log_2(2^0) = \log_2(1) = 0$ (zero number of 0's after first 1 in the binary format)
$\log_2(2^1) = \log_2(2) = 1$ (one 0 after first 1 in the binary format, which is 10)
$\log_2(2^2) = 2\log_2(2) = 2 \times 1 = 2$ (two 0's after first 1 in binary format, which is 100)

The conclusion is that, the value of $\log_2(n)$, where n is a binary number and a perfect power of 2, is equal to the number of 0's after the first 1 in the given number n.

Algorithm

Count of the number of zeros after the beginning '1' of the binary number '*n*'; it gives us the required answer.

1. Read the first digit, that is, 1, and ignore it.
2. Read the following 0 from the string that follows the 1, replace it by a symbol, *a*, and write a symbol, *c*, after the right end-marker.
3. Repeat the procedure till all the 0's that follow 1 get replaced by *a*'s.
4. The number of *c*'s written after the right end-marker gives the required value of $\log_2(n)$.

Thus, the input string is in binary format and the resultant value is in unary format using symbol *c*.

Note: We could use a simpler algorithm as well, which simply replaces the first digit 1 with a blank character ♭; the remaining number of 0's gives the required answer. However, this would destroy the input parameter, and hence we do not use this algorithm.

The initial configuration of the required TM is shown in Fig. 4.16(a), and its transition graph is shown in Fig. 4.16(b).

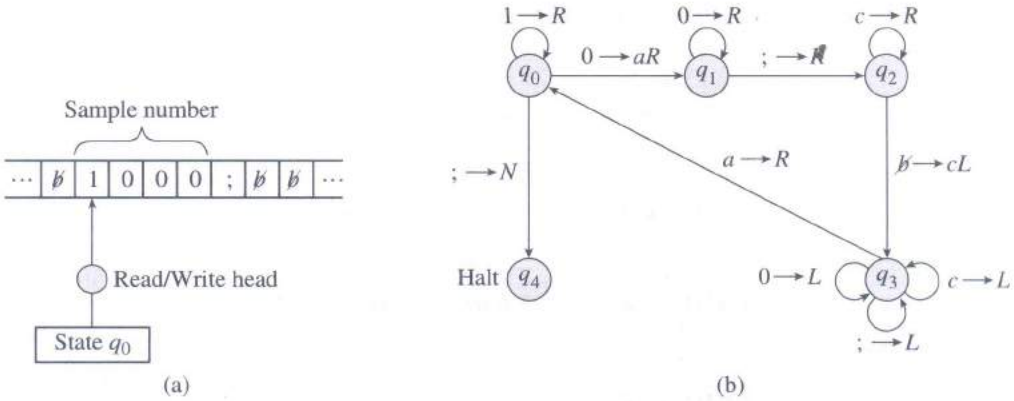


Figure 4.16 Finding the value of $\log_2(n)$ (a) Initial configuration (b) TG for TM that finds $\log_2(n)$

The SFM is created as shown in Table 4.15.

Table 4.15 SFM for a TM that finds $\log_2$ (n)

S \ I	1	0	;	a	c	♭
q_0	R	aq_1R	q_4N	—	—	—
q_1	—	R	q_2R	—	—	—
q_2	—	—	—	—	R	cq_3L
q_3	—	L	L	q_0R	L	—
q_4	—	—	—	—	—	—

Simulation

Let us simulate the working of the TM that we have constructed for $n = 1000$ ($= 8 = 2^3$). We know that for this number n, $\log_2(n) = 3$, which can be represented as ccc in unary form.

1 0 0 0 ; ♭ ♭ … initial configuration
↑
q_0

1 0 0 0 ; ♭ ♭ … $\delta(q_0, 1) = (R)$
↑
q_0

1 a 0 0 ; ♭ ♭ … $\delta(q_0, 0) = (a\, q_1\, R)$
↑
q_1

1 a 0 0 ; ♭ ♭ … $\delta(q_1, 0) = (R)$
↑
q_1

$1\ a\ a\ a\ ;\ c\ c\ \not b\ \not b \ldots$ $\quad \delta\ (q_0, 0) = (a\ q_1\ R)$
$\uparrow$
q_1

$1\ a\ a\ a\ ;\ c\ c\ \not b\ \not b \ldots$ $\quad \delta\ (q_1, ;) = (q_2\ R)$
$\uparrow$
q_2

$1\ a\ a\ a\ ;\ c\ c\ \not b\ \not b \ldots$ $\quad \delta\ (q_2, c) = (R)$
$\uparrow$
q_2

$1\ a\ a\ a\ ;\ c\ c\ \not b\ \not b \ldots$ $\quad \delta\ (q_2, c) = (R)$
$\uparrow$
q_2

$1\ a\ a\ a\ ;\ c\ c\ c\ \not b \ldots$ $\quad \delta\ (q_2, \not b) = (c\ q_3\ L)$
$\uparrow$
q_3

$1\ a\ a\ a\ ;\ c\ c\ c\ \not b \ldots$ $\quad \delta\ (q_3, c) = (L)$
$\uparrow$
q_3

$1\ a\ a\ a\ ;\ c\ c\ c\ \not b \ldots$ $\quad \delta\ (q_3, c) = (L)$
$\uparrow$
q_3

$1\ a\ a\ a\ ;\ c\ c\ c\ \not b \ldots$ $\quad \delta\ (q_3, ;) = (L)$
$\uparrow$
q_3

$1\ a\ a\ a\ ;\ c\ c\ c\ \not b \ldots$ $\quad \delta\ (q_3, a) = (q_0\ R)$
$\uparrow$
q_0

$1\ a\ a\ a\ ;\ c\ c\ c\ \not b \ldots$ $\quad \delta\ (q_0, ;) = (q_4\ N)$
$\uparrow$
q_4

The TM halts with the result *ccc*, which is the expected value, that is, 3.

We could also replace all the *a*'s by 0's again, in order to retain the input parameter, so as to have a non-destructing algorithm.

4.7 COMPLEXITY OF A TURING MACHINE

The complexity of a TM is directly proportional to the size of the functional matrix. In other words, we can say that the complexity of a TM depends on the number of symbols that are being used and the number of states of the TM. Hence

$$\text{Complexity of a TM} = |\Gamma| \times |Q| \qquad (\text{or } |I| \times |S|),$$

where, $|\Gamma|$ = Cardinality of tape alphabet (i.e., number of tape symbols),
and $|Q|$ = Number of states of the TM.

Let us consider Example 4.13 in which we designed a TM that finds $\log_2 (n)$, where n is a perfect power of 2, and is represented in binary format. For this example, we have

$\Gamma = \{1, 0, a, c, ;, \not b\}$
$Q = \{q_0, q_1, q_2, q_3, q_4 = \text{halt}\}$

Therefore, the complexity of the TM = $|\Gamma| \times |Q| = 6 \times 5 = 30$

Thus, while designing a TM, we must ensure that it has minimum complexity, that is, we should design a TM such that it has lesser number of input symbols and states.

4.8 COMPOSITE AND ITERATIVE TURING MACHINES

Two or more Turing machines can be combined to solve a complex problem, such that the output of one TM forms the input to the next TM, and so on. This is called *composition.*

For realizing a composite TM (or a CTM), the functional matrices of the component TMs are combined by re-labeling the symbols, as required, and suitably branching to an appropriate state rather than the halt state at the completion of the performance of each component TM. Figure 4.17(a) depicts a composition of n TMs.

Another way of having a combination TM is by applying its own output as input repetitively. This is called *iteration* or *recursion*, and the TM is said to be an iterative TM (or ITM). For an example, refer to Fig. 4.17(b).

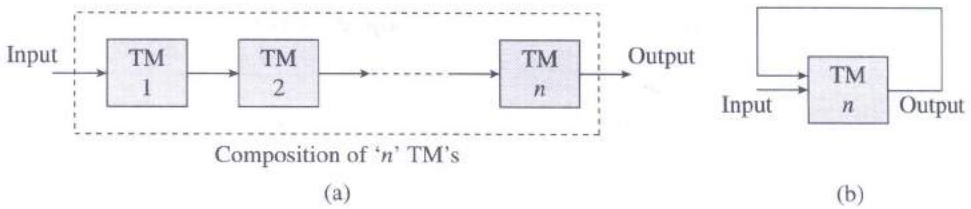


Figure 4.17 Composite and iterative TMs (a) Composite TM (CTM) (b) Iterative TM (ITM)

The idea of a composite TM gives rise to the concept of breaking a complicated job into a number of smaller jobs, implementing each separately, and then combining them together to get the answer for the job required to be done. Therefore, we can divide a problem into simple jobs and design different TMs for each job. This is a typical *separation of concerns* achieved in software development; it is analogous to the function composition that we know from discrete mathematics: $f \circ g\,(x) = f(g\,(x))$. The output of $g\,(x)$ is given as input to function f. In a way, modular programming can be considered to be influenced by CTM.

Functionally, most of the TMs that we have implemented earlier in the chapter, for example, multiplication as repetitive addition, division as repetitive subtraction, and so on, are examples of iterative TMs.

Example 4.14 Design a TM to find the value of n^2, where n is any integer ≥ 0.

Solution Let us consider the initial configuration to be as shown in Fig. 4.18 (a).

The number n is represented in unary form using 0's, as shown in Fig. 4.18(a). We find n^2 is in terms of the multiplication '$n \times n$'. This involves copying n after the comma ',', onto the tape, and using that as the multiplicand—represented in unary form using symbol 1—as shown in Fig. 4.18(b). We can then perform the multiplication '$n \times n$', as we have done in Example 4.10. Figure 4.18(b) is the initial configuration for the TM that performs multiplication as we have seen earlier.

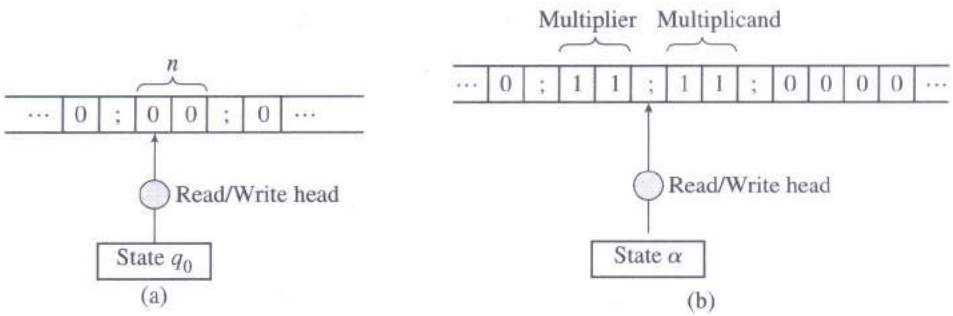


Figure 4.18 TM that computes n^2 ($n \geq 0$) (a) Initial configuration (b) Configuration after copying n onto tape

Observe that the TM that computes n^2 can be considered as a composition of two TMs. The first TM prepares the output in the form suggested in Figure 4.18(b), which can be used as input for the second TM that performs multiplication. The second TM is similar to the one that we have already designed in Example 4.10; the SFM for this TM is described in Table 4.12 with α as the initial state. Hence, we need to design the first TM, which converts the initial configuration shown in Fig. 4.18(a) into the form that can be used as input by the second TM, whose initial configuration is shown in Fig. 4.18(b).

Algorithm

1. Replace one 0 at a time by symbol 1 and copy it after the right end-marker, ';'.
2. Repeat this process till all the 0's from n are replaced by 1's.
3. Thus, another copy of n gets prepared after the right end-marker, ';'. This is done to store n as the multiplier as well as multiplicand onto the tape.
4. Now, replace the end-marker, ';', by a comma, ',', and add another end-marker, ';', at the end of the copy of n created at the right end.

The SFM for the required TM is shown in Table 4.16.

Table 4.16 SFM for the TM that prepares input for multiplication

S \ I	0	;	1	,
q_0	$1q_1R$	$,q_5R$	—	—
q_1	R	q_2R	—	—
q_2	$1q_3L$	—	R	—
q_3	—	q_4L	L	—
q_4	L	—	q_0R	—
q_5	$;q_6L$	—	R	—
q_6	—	—	L	αN

Simulation

Let us simulate the working of the TM for $n = 2$.

... 0 ; 0 0 ; 0 0 ... initial configuration
↑
q_0

... 0 ; 1 0 ; 0 0 ... $\delta(q_0, 0) = (1\ q_1\ R)$
↑
q_1