INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

Lecture Notes:

THEORY OF COMPUTATION(AITC04)

Drafted by :
Dr. U Sivaji (iare 10671)
Assistant Professor

Department Of Information Technology
Institute of Aeronautical Engineering

February 21, 2023

Contents

List of Figures

Chapter 1

FINITE AUTOMATA

Course Outcomes

After successful completion of this module, students should be able to:

CO 1	Make use of deterministic finite automata and non deterministic finite automata for modeling lexical analysis and text editors.	Apply

1.1 Fundamental

In theoretical computer science, the theory of computation is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory. In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine.

1.2 Automata theory

In theoretical computer science, automata theory is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational

problems that can be solved using these machines. These abstract machines are called automata which is sihown in fig 1.1 This automaton consists of

1. states (represented in the figure by circles),
2. transitions (represented by arrows).

As the automaton sees a symbol of input, it makes a transition (or jump) to another state, according to its transition function (which takes the current state and the recent symbol as its inputs).
Uses of Automata: compiler design and parsing

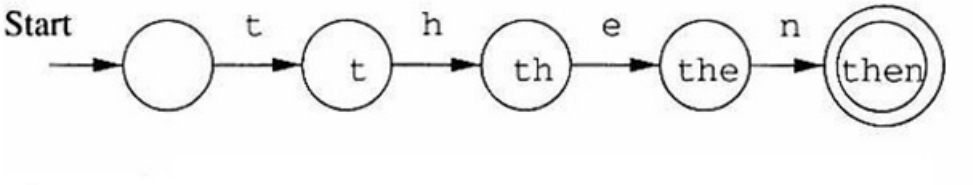


Figure 1.1: Finite Automaton modelling recognition of then

Languages:
The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal definitions. Programming languages belong to this category.

Symbols:

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as a, 0, 1.

Alphabets:

An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted by $\sum$.
Example which is shiown in fig 1.2:

$\Sigma = \{0, 1\}$
$\Sigma = \{a, b, c\}$
$\Sigma = \{a, b, c, \&, z\}$
$\Sigma = \{\#, \nabla, \spadesuit, \beta\}$

Figure 1.2: Examples for Alphabets

Strings:

A string or word over an alphabet $\sum$ is a finite sequence of concatenated symbols of $\sum$

Example: 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } . aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alphabet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

Length of a string:

The number of symbols in a string w is called its length, denoted by |w|.

Example: | 011 | = 4, |11| = 2, | b | = 1

Convention: We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end to Denote strings over an alphabet. That is,U,VW,X,Y,Z (Symbols) are strings.

Some String Operations:

Let $x = a_1a_2a_3 and y = b_1b_2b_3$ be two strings. The concatenation of x and y Denoted by xy, is the string$a_1a_2a_3b_1b_2b_3$. That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

Example: Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.

Prefixes: ε, 0, 01, 011.

Suffixes: ε, 1, 11, 011.

Substrings: ε, 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and $\varepsilon$is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and $x \neq y$. In the above example, all prefixes except 011 are proper prefixes.

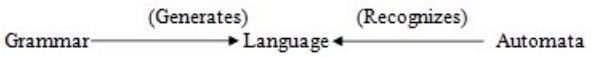


Figure 1.3: Grammar to Automata

Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string which is shiown in fig 1.3

An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

• Every automaton consists of some essential features as in real computers. It has a mech anism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left)and change the input which is shiown in fig 1.4

• The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an output sequence in response to an input sequence, then it is called a transducer(or automaton

with output).

• The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet (whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.

• The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some de- fined manner determined by a transition function.

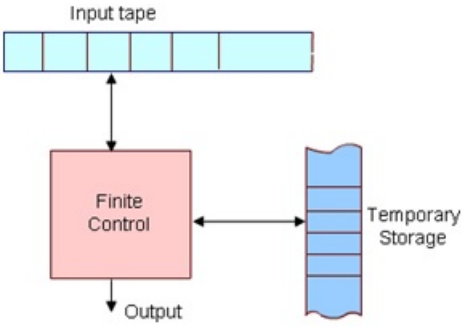


Figure 1.4: Finite Automaton model

Operation of the automation is defined as follows:

At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automa- ton then moves to some other integral (or remain in the same state) as defined by the transition function. The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the con- tent of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition. The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next (as defined by the transition function) is called a move. Finite state machine or Finite Automation is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and

hence a restricted model of computation.

Finite Automata:

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

States, Transitions and Finite-State Transition System :

Let us first give some intuitive idea about a state of a system and state transitions before describing finite automata.

Informally, a state of a system is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

Transitions are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A system containing only a finite number of states and transitions among them is called a finite-state transition system.

Finite-state transition systems can be modeled abstractly by a mathematical model called finite automation

1.3 Deterministic Finite (state) Automata

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an in- put string – one symbol at a time – and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the

tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A tape to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from .
2. A tape head for reading symbols from the tape
3. A control , which itself consists of 3 things:

- finite number of states that the machine is allowed to be in (zero or more states are designated as accept or final states),
- a current state, initially set to a start state,
- a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there tothe control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition functionto get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined. If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summarizing all the above we can formulate the following formal definition: Deterministic Finite State Automaton (DFA)is a 5-tuple : M = {Q,$\sum, \delta, q_0$,F}

- Q is a finite set of states.
- $\sum$ is a finite set of input symbols or alphabet
- δ is the “next state” transition function (which is total). Intuitively, a function that tells which state to move to in response to an input, i.e., if M isin state q and sees input a, it moves to state .

$\delta : QX\sum \rightarrow Q$

- $q_0$is the start state.
- F is the set of accept or final states.

Acceptance of Strings :

A DFA accepts a string W $= x = a_1a_2.....a_n if there is a sequence of states q_0q_1.....q_n$ in Q such that

1. q_0 is the start state
2. $\delta(q_i, a_i) = q_{i+1} \forall 0 < i <$ n
3. $q_n \in F$

Language Accepted or Recognized by a DFA :
The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by L(M) i.e.L(M) =$\{$w $\in \sum^*$ | M accepts w $\}$. The notion of acceptance can also be made more precise by extending the transition function δ .
Extended transition function :
Extend $\delta : QX\sum \rightarrow$Q (which is function on symbols) to a function on strings, i.e . $\hat{\delta}$: $QX\sum^* \rightarrow Q$ That is, $\hat{\delta}(q,w)$ is the state the automation reaches when it starts from the state q andfinish processing the string w. Formally, we can give an inductive definition as follows: The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.L(M) =$\{$w $\in \sum^*|$ M accepts w $\}$ means $\hat{\delta}(q_0,w) \in$ F.

It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over 0, 1 having at least one 1.
We can describe the same DFA by transition table or state transition diagram as following:

Transition Table which is shiown in fig 1.5

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the “next state”).

- Rows correspond to states,

- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

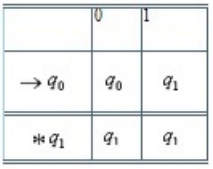


Figure 1.5: Transition Table

(State) Transition diagram :which is shiown in fig 1.6 , 1.7 and 1.8

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.
2. There is a directed edge from node q to node p labeleda iff . (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.

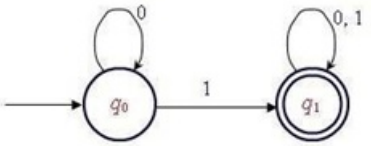


Figure 1.6: Transition Diagram

Design FA with $\sum = \{0, 1\}$ accepts even number of 0's and even number of 1's.

Here q0 is a start state and the final state also. Note carefully that a symmetry of 0's and 1's is maintained. We can associate meanings to each state as:

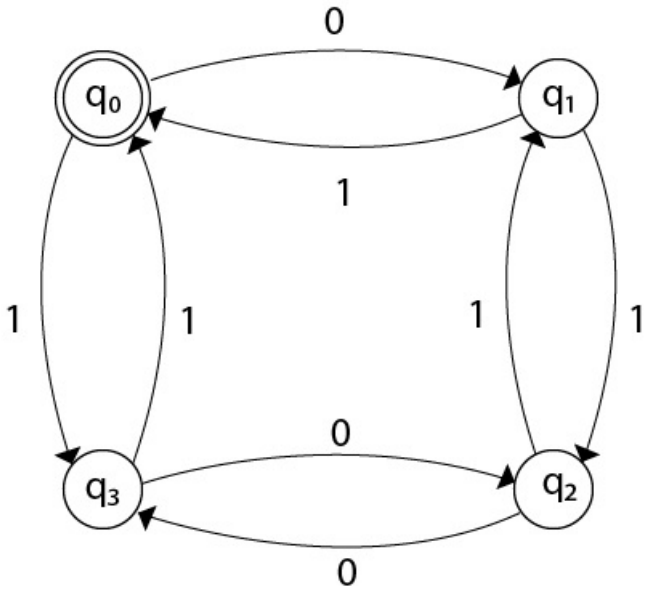


Figure 1.7: DFA Transition Diagram for even number of 0's and 1's

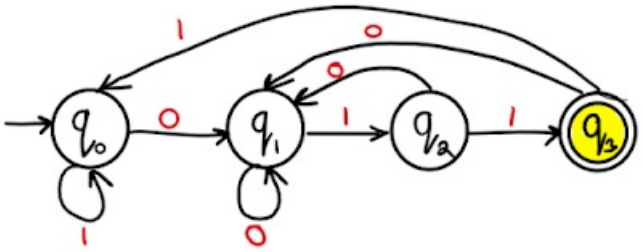


Figure 1.8: DFA Transition Diagram for strings ending with 011

q0: state of even number of 0's and even number of 1's.

q1: state of odd number of 0's and even number of 1's.

q2: state of odd number of 0's and odd number of 1's.

q3: state of even number of 0's and odd number of 1's.

Draw a DFA for the language accepting strings ending with '011' over input alphabets $\sum$ = {0, 1}

1.4 Non-Deterministic Finite Automata

Nondeterminism is an important abstraction in computer science. Importance of nondeterminism is found in the design of algorithms. For examples, there are many problems with efficient nondeterministic solutions but no known efficient deterministic solutions. (Travelling salesman, Hamiltonean cycle, clique, etc). Behaviour of a process is in a distributed system is also a good example of nondeterministic situation. Because the behaviour of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.

It is easy to construct and comprehend an NFA than DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in this subject.

In the context of FA nondeterminism can be incorporated naturally. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

1. multiple next state.
2. ε- transitions.

Multiple Next State :

In contrast to a DFA, the next state is not necessarily uniquely determined by the current state and input symbol in case of an NFA. (Recall that, in a DFA there is exactly one start state and exactly one transition out of every state for each symbol in). This means that - in a state q and with input symbol a - there could be one, more than one or zero next state to go, i.e. the value of is a subset of Q. Thus = which means that any one of could be the next state.

The zero next state case is a special one giving = , which means that there is no next state on input symbol when the automata is in state q. In such a case, we may think that the automata ”hangs” and the input will be rejected.

ε- transitions :

In an ε-transition, the tape head doesn't do anything- it doesnot read and it doesnot move.

However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as implying that the next state could by any one of w o consuming the next input symbol.

a 5-tuple : M = {Q,$\sum, \delta, q_0$,F}

- Q is a finite set of states.
- $\sum$ is a finite set of input symbols or alphabet
- δ is the "next state" transition function (which is total). Intuitively, a function that tells which state to move to in response to an input, i.e., if M isin state q and sees input a, it moves to state .

$\delta : QX\sum \rightarrow 2^Q$

- $q_0$is the start state.
- F is the set of accept or final states.

Acceptance of Strings :

A NFA accepts a string W = $x = a_1a_2.....a_n if there is a sequence of states q_0q_1.....q_n$ in Q such that

1. q_0 is the start state
2. $\delta(q_i, a_i) = q_{i+1} \forall 0 < i <$ n
3. $q_n \in F$

Language Accepted or Recognized by a NFA :

The language accepted or recognized by a NFA M is the set of all strings accepted by M , and is denoted by L(M) i.e.L(M) ={w $\in \sum^*$ | M accepts w }. The notion of acceptance can also be made more precise by extending the transition function δ .

Extended transition function :

Extend $\delta : QX\sum \rightarrow 2^Q$ (which is function on symbols) to a function on strings, i.e . $\hat{\delta}$: $QX\sum^* \rightarrow 2^Q$ That is, $\hat{\delta}(q, w)$ is the state the automation reaches when it starts from the state q andfinish processing the string w. Formally, we can give an inductive definition as

follows: The language of the NFA M is the set of strings that can take the start state to one of the accepting states i.e.L(M) ={w $\in \sum^*$| M accepts w } means $\hat{\delta}(q_0, w) \in$ F.

Acceptance :

Informally, an NFA is said to accept its input if it is possible to start in some start state and process , moving according to the transition rules and making choices along the way whenever the next state is not uniquely defined, such that when is completely processed (i.e. end of is reached), the automata is in an accept state. There may be several possible paths through the automation in response to an input since the start state is not determined and there are choices along the way because of multiple next states. Some of these paths may lead to accpet states while others may not. The

automation is said to accept if at least one computation path on input starting from at least one start state leads to an accept state- otherwise, the automation rejects input . Alternatively, we can say that, is accepted iff there exists a path with label from some start state to some accept state. Since there is no mechanism for determining which state to start in or which of the possible next moves to take (including the - transitions) in response to an input symbol we can think that the automation is having some ”guessing” power to chose the correct one in case the input is accepted

Transition diagram:

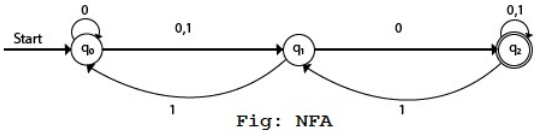


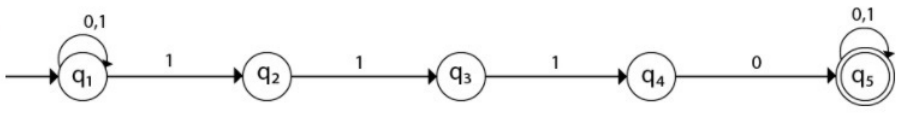
Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q0, q1	q1
q1	q2	q0
*q2	q2	q1, q2

Figure 1.9: NFA Example

Design an NFA in which all the string contain a substring 1110 .



Transition table for the above transition diagram can be given below:

Present State	0	1
→q1	q1	q1, q2
q2	ø	q3
q3	ø	q4
q4	q5	ø
*q5	q5	q5

Figure 1.10: NFA Example for substring 1110

[h]

Consider a string 111010,

$$\begin{aligned}\delta(q1, 111010) &= \delta(q1, 1100)\\ &= \delta(q1, 100)\\ &= \delta(q2, 00)\end{aligned}$$

Got stuck! As there is no path from q2 for input symbol 0. We can process string 111010 in another way.

$$\begin{aligned}\delta(q1, 111010) &= \delta(q2, 1100)\\ &= \delta(q3, 100)\\ &= \delta(q4, 00)\\ &= \delta(q5, 0)\\ &= \delta(q5, \varepsilon)\end{aligned}$$

As state q5 is the accept state. We get the complete scanned, and we reached to the final state.

Figure 1.11: NFA accepting string 1110

1.5 Non-Deterministic Finite Automata with ε- transitions.

Nondeterministic finite automaton with ε-moves (NFA-ε) is a further generalization to NFA. This automaton replaces the transition function with the one that allows the empty

string εas a possible input. The transitions without consuming an input symbol are called ε-transitions.

a 5-tuple : M = {Q,$\sum, \delta, q_0$,F}

- Q is a finite set of states.
- $\sum$ is a finite set of input symbols or alphabet
- δ is the "next state" transition function (which is total). Intuitively, a function that tells which state to move to in response to an input, i.e., if M isin state q and sees input a, it moves to state .

$\delta : QX\{\sum U\varepsilon\} \rightarrow 2^Q$

- $q_0$is the start state.
- F is the set of accept or final states.

Acceptance of Strings :

A NFA with ε- transanctions accepts a string W = $x = a_1a_2.....a_n$ if there is a sequence of states $q_0q_1.....q_n$ in Q such that

1. q_0 is the start state
2. $\delta(q_i, a_i) = q_{i+1} \forall 0 < i <$ n
3. $q_n \in F$

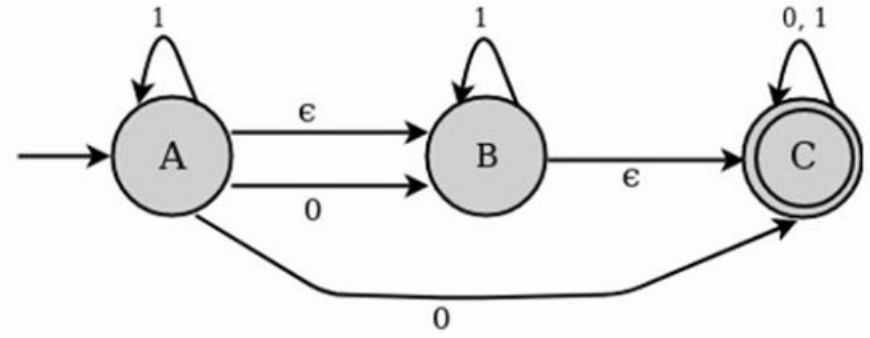
Epsilon (ε) - closure Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or εmoves including the state X itself.

In other words, ε-closure for a state can be obtained by union operation of the ε-closure of the states which can be reached from X with a single εmove in a recursive manner.

Removing ε-transition:

ε- transitions do not increase the power of an NFA . That is, any ε- NFA (NFA with εtransition), we can always construct an equivalent NFA without ε-transitions. The equivalent NFA must keep track where the ε- NFA goes at every step during computation. This can

Consider the following figure of NFA with ε move −



The transition state table for the above NFA is as follows −

State	0	1	ϵ
A	B,C	A	B
B	ø	B	C
C	C	C	ø

For the above example, ε closure are as follows −

- E closure(A) : {A, B,C}
- E closure(B) :{B,C}
- E closure(C) : {C}

Figure 1.12: NFA with epsilon Example

be done by adding extra transitions for removal of every ε- transitions from the ε-NFA as follows.

If we removed the ε- transition from theε- NFA , then we need to moves from state p to all the state on input symbol q which are reachable from state q (in the ε- NFA) on same input symbol q. This will allow the modified NFA to move from state p to all states on some input symbols which were possible in case of ε-NFA on the same input symbol.

Convert NFA with $\varepsilon$to without ε:
In this method, we try to remove all the -transitions from the given Non-deterministic finite automata (NFA) −ε- transition
The method is mentioned below stepwise −

- Step 1 − Find out all the ε-transitions from each state from Q. That will be called as ε-closure(qi) where, qi ∈ Q.

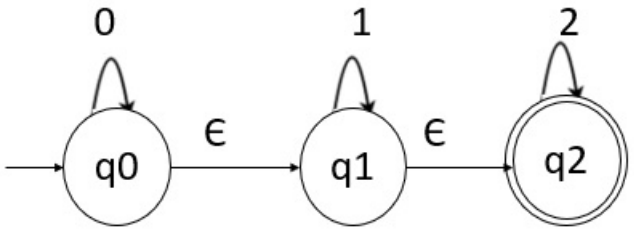


Figure 1.13: NFA with epsilon problem

- Step 2 − Then, δtransitions can be obtained. Finally ε-closure on δmoves.
- Step 3 − Step 2 is repeated for each input symbol and for each state of given NFA.
- Step 4 − By using the resultant status, the transition table for equivalent NFA without can be built.

NFA with $\varepsilon$to without $\varepsilon$is as follows − δ(q,a) = ε- closure ($\delta(\hat{\delta}(q,\varepsilon),a)$) where, $\hat{\delta}(q,\varepsilon)$ = ε-closure(q)

We will first obtain ε-closure of each state i.e., we will find ε-reachable states from the current state. Hence,

- ε-closure(q0) = {q0,q1,q2}
- ε-closure(q1) = {q1,q2}
- ε-closure(q2) = {q2}

ε-closure(q0) means with null input (no input symbol) we can reach q0, q1, q2. In a similar manner for q1 and q2 ε-closure are obtained.
Now we will obtain δtransitions for each state on each input symbol as shown below −
δ'(q0, 0) = ε-closure($\delta(\hat{\delta}$ (q0, ε),0))
= ε-closure(δ(ε-closure(q0),0))
= ε-closure(δ(q0,q1,q2), 0))
= ε-closure(δ(q0, 0) U δ(q1, 0) U δ(q2, 0))
= ε-closure(q0 U ϕU ϕ)
= ε-closure(q0)

= {q0,q1, q2}

δ'(q0, 1) = ε-closure($\delta(\hat{\delta}$ (q0, ε),1))
= ε-closure(δ(q0,q1,q2), 1))
= ε-closure(δ(q0, 1) U δ(q1, 1) U δ(q2, 1))
= ε-closure(ϕU q1 U ϕ)
= ε-closure(q1)
= {q1, q2}

δ'(q0, 2) = ε-closure($\delta(\hat{\delta}$ (q0, ε),2))
= ε-closure(δ(q0,q1,q2), 2))
= ε-closure(δ(q0, 2) U δ(q1, 2) U δ(q2, 2))
= ε-closure(ϕU ϕU q2)
= ε-closure(q2)
= {q2}

δ'(q1, 0) = ε-closure($\delta(\hat{\delta}$ (q1, ε),0))
= ε-closure(δ(q1,q2), 0))
= ε-closure(δ(q1, 0) U δ(q2, 0))
= ε-closure(ϕU ϕ)
= ε-closure(ϕ)
= ϕ

δ'(q1,1) = ε-closure($\delta(\hat{\delta}$ (q1, ε),1))
= ε-closure(δ(q1,q2), 1))
= ε-closure(δ(q1, 1) U δ(q2, 1))
= ε-closure(q1 U ϕ)
= ε-closure(q1)
= {q1,q2}

$\delta'(q1, 2) = \varepsilon\text{-closure}(\delta(\hat{\delta}\ (q1, \varepsilon),2))$

$= \varepsilon\text{-closure}(\delta(q1,q2), 2))$

$= \varepsilon\text{-closure}(\delta(q1, 2)\ U\ \delta(q2, 2)\ )$

$= \varepsilon\text{-closure}(\phi U\ q2)$

$= \varepsilon\text{-closure}(q2)$

$= \{q2\}$

$\delta'(q2, 0) = \varepsilon\text{-closure}(\delta(\hat{\delta}\ (q2, \varepsilon),0))$

$= \varepsilon\text{-closure}(\delta(q2), 0))$

$= \varepsilon\text{-closure}(\delta(q2, 0))$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta'(q2, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}\ (q2, \varepsilon),1))$

$= \varepsilon\text{-closure}(\delta(q2), 1)$

$= \varepsilon\text{-closure}(\delta(q2, 1))$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta'(q2, 2) = \varepsilon\text{-closure}(\delta(\hat{\delta}\ (q2, \varepsilon),))$

$= \varepsilon\text{-closure}(\delta(q2), 2))$

$= \varepsilon\text{-closure}(\delta(q2, 2))$

$= \varepsilon\text{-closure}(q2)$

$= \{q2\}$

Now, we will summarize all the computed δ' transitions as given below −

$\delta'(q0,0)=\{q0,q1,q2\}$

$\delta'(q0,1)=\{q1,q2\}$

$\delta'(q0,2)=\{q2\}$

$\delta'(q1,0)= \{\ \phi\}$

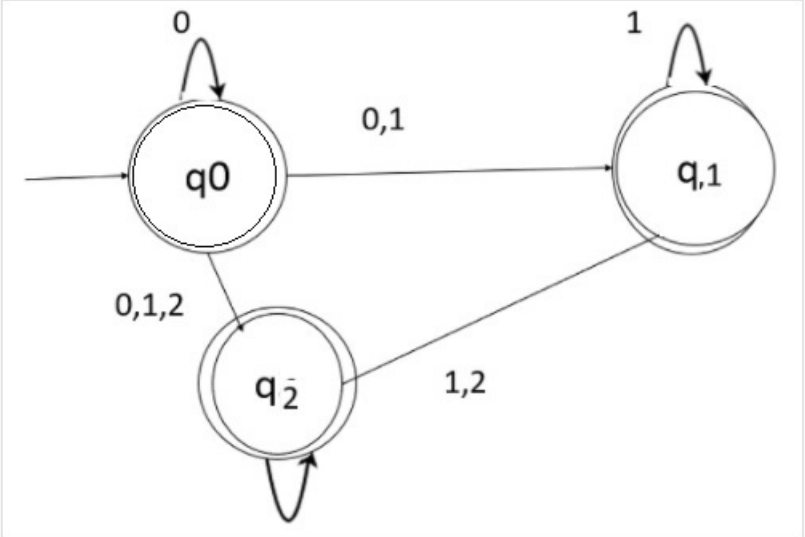
$\delta'(q1,1)=\{q1,q2\}$

The **transition table** is given below –

States\inputs	0	1	2
q0	{q0,q1,q2}	{q1,q2}	{q2}
q1	Φ	{q1,q2}	{q2}
q2	Φ	Φ	{q2}

NFA without epsilon

The NFA without epsilon is given below –



Here, q0, q1, q2 are final states because ε-closure(q0), ε-closure(q1) and ε-closure(q2) contain a final state q2.

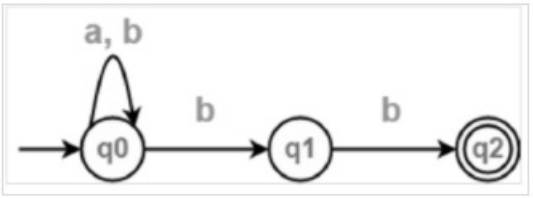
Figure 1.14: NFA with out epsilon after conversion

δ'(q1,2)={q2}

δ'(q2,0)={ ϕ}
δ'(q2,1)={ ϕ}
δ'(q2,2)={q2}

Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA):

The transition diagram is as follows −



The transition table of NFA is as follows −

State	a	b
->q0	q0	q0,q1
q1	-	*q2
*q2	-	-

The DFA table cannot have multiple states. So, make q0q1 as a single state.

Let's convert the given NFA to DFA by considering two states as a single state.

The transition table of DFA is as follows −

State	a	b
->q0	q0	[q0,q1]
[q0q1]	[q0]	[q0q1q2]
*[q0q1q2]	[q0]	[q0q1q2]

In the above transition table, q2 is the final state. Wherever, q2 is present that becomes the final state.

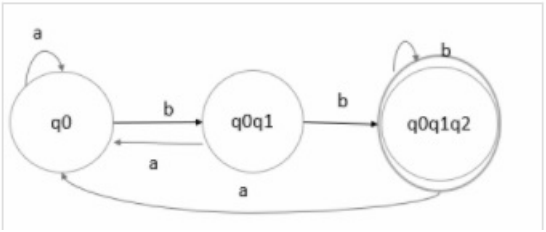


Figure 1.15: NFA to DFA conversion

- After conversion the number of states in the final DFA may or may not be the same as in NFA.
- The maximum number of states present in DFA may be 2^Q, (Q in NFA)
- The final DFA all states that contain the final states of NFA are treated as final states.

Chapter 2

REGULAR LANGUAGES

Course Outcomes

After successful completion of this module, students should be able to:

CO 2	Extend regular expressions and regular grammars for parsing and designing programming languages.	Understand
CO 3	Illustrate the pumping lemma on regular and context free languages for performing negative test .	Understand

2.1 Regular Expressions: Formal Definition

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

Definition : Let S be an alphabet. The regular expressions are defined recursively as follows.

1. ϕ is a RE
2. ε is a RE
3. $\forall a \in S$, a is RE

If r1 and r2 are REs over$\sum$, then so are

1. r1 + r2 is a RE
2. r1r2 is a RE
3. r1* , r2* are RE
4. (r1) is a RE

ε is a regular expression, and L(ε) is { ε}, that is, the language whose sole member is the empty string.
Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,
• (r)|(s) is a regular expression denoting the language L(r) U L(s).
• (r)(s) is a regular expression denoting the language L(r)L(s).
• (r)* is a regular expression denoting (L(r))*.
• (r) is a regular expression denoting L(r).

Language described by REs : Each describes a language (or a language is associated with every RE). We will see later that REs are used to attribute regular languages.
Notation : If r is a RE over some alphabet then L(r) is the language associate with r . We can define the language L(r) associated with (or described by) a REs as follows.

1. ϕ is a regular expression denoting the empty set language {} i.e. L(ϕ)= ϕ
2. ε is a regular expression denoting the set that contains only the empty string {ε} as language, i.e., L(ε)= ε
3. For every symbol a in $\sum$, a is a regular expression and {a} is a language. L(a)={a}

These are called as Primitive Regular Expressions or Building blocks.

Regular Expression	Regular Language	Remarks
∅	{}	Empty Language
ε	{ε}	Language containing empty string.
a	{a}	a is a regular expression
a*	{ε, a,aa,aaa, aaaa,.....} $\{a^n \mid n \geq 0\}$	Set of all strings over 'a' including null string. (a+ε)*, zero or more a's
a^+	a.a* or a*.a {a, aa, aaa, ...} $\{a^n \mid n \geq 1\}$	Set of all strings over 'a' without null string. Positive Closure. One or more a's
a+b	Either a or b	Alternation {a,b} Similarly a+ε = {a, ε}
(a+b)*	{ε, a, b, aa, ab, ba, bb, aaa,.....}	Set of all strings possible over a,b (a+b)* = (a*b*)* $\{a^m b^n \mid m, n \geq 0\}$
$(a+b)^+$	{a, b, aa, ab, ba, bb, ...}	Set of all strings possible over a, b 1 or more occurrences of either a or b or both.
(ab)	{ab}	Not a or b, **together ab**
(ab)*	{ε, ab, abab, ababab, ...}	Set of strings with only ab in order. $\{(ab)^n \mid n \geq 0\}$
a+b*	{a, ε, b, bb, bbb, ...}	a or 0 or more number of b's
(a+b)ab*	{aa, ba, aab,babb, aabbbb,....}	Set of all strings start with either a or b, second symbol is a and followed by 0 or more number of b's
a*ba*	{b,aba,aab,baa,..}	Contains single b
Σ*bΣ*	{b,abab,aab,aabaa,..}	At least one b if Σ={a,b}
a*b*c*	L={ε, a,b,c, abccc, aaaabccc,aabbcc.....}	Any number of a's followed by any number of b's followed by any number of c's.
aa*bb*cc*	$a^+b^+c^+$	At least one of each symbol.

Figure 2.1: Examples for Regular Expressions and Regular Languages

Regular Expressions - Identity Rules:There are many identities for the regular expression. Let p, q and r are regular expressions.

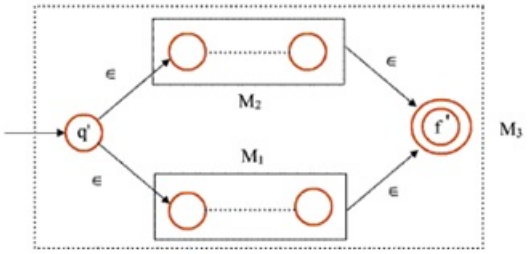
- $\emptyset + r = r$
- $\emptyset.r = r.\emptyset = \emptyset$
- $\in.r = r.\in = r$
- $\in^* = \in$ and $\emptyset^* = \in$
- $r + r = r$
- $r^*.r^* = r^*$
- $r.r^* = r^*.r = r^+$.
- $(r^*)^* = r^*$
- $\in + r.r^* = r^* = \in + r.r^*$
- $(p.q)^*.p = p.(q.p)^*$
- $(p + q)^* = (p^*.q^*)^* = (p^* + q^*)^*$
- $(p+ q).r = p.r + q.r$ and $r.(p+q) = r.p + r.q$

Figure 2.2: Regular Expressions - Identity Rules

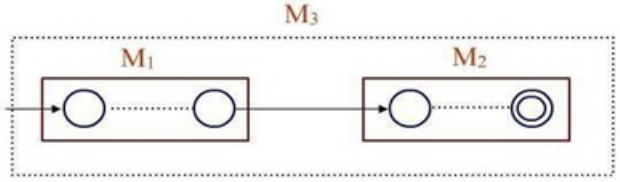
2.2 Regular Expressions to Finite Automata.

The Thompson's Construction Algorithm is used to build an NFA from Regular Expression(RE), and Subset construction algorithm can be applied to convert the NFA into a DFA.

Case (i) : Consider the RE r3 = r1 + r2 denoting the language L(r1) UL(r2) . We construct FA M3 from M1 and M2 to accept the language denoted by RE r3 as follows :



Case (ii) : Consider the RE r3 = r1r2 denoting the language L(r1) L(r2) . We construct FA M3 from M1 and M2 to accept the language denoted by RE r3 as follows :



Case (iii) : Consider the RE r = r1* denoting the language L(r1)* . We construct FA M from M1 to accept the language denoted by RE r as follows :

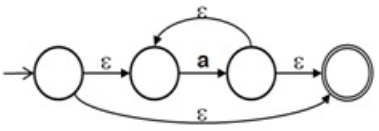


Figure 2.3: Regular Expressions to Finite Automata rules

Construct Finite Automata for the given Regular Expression: a (b | c)*

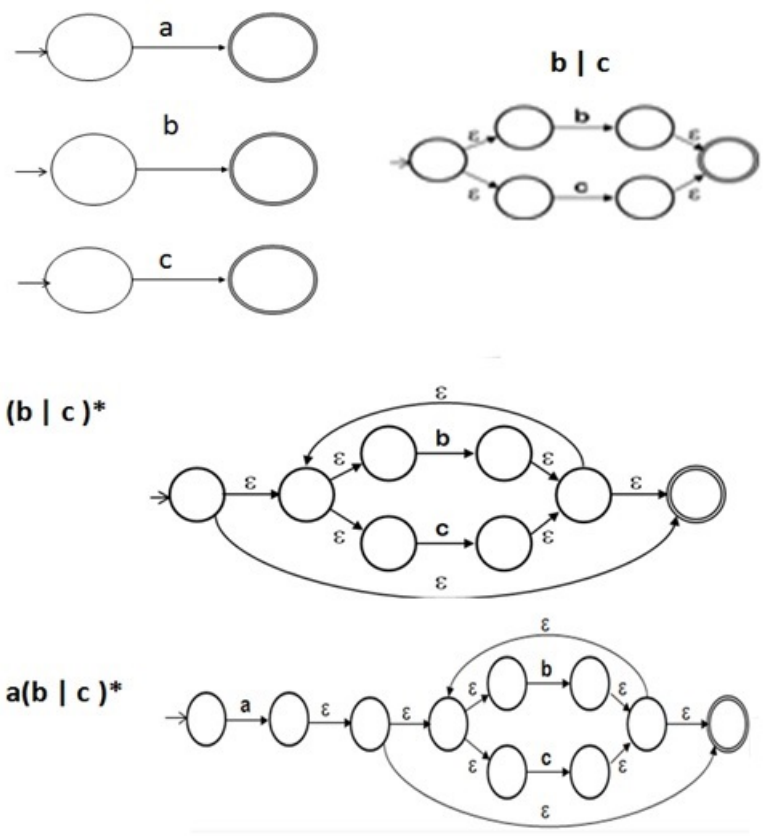


Figure 2.4: Finite Automata for a (b | c)*

Construct Finite Automata for the given Regular Expression: (ab + a)*

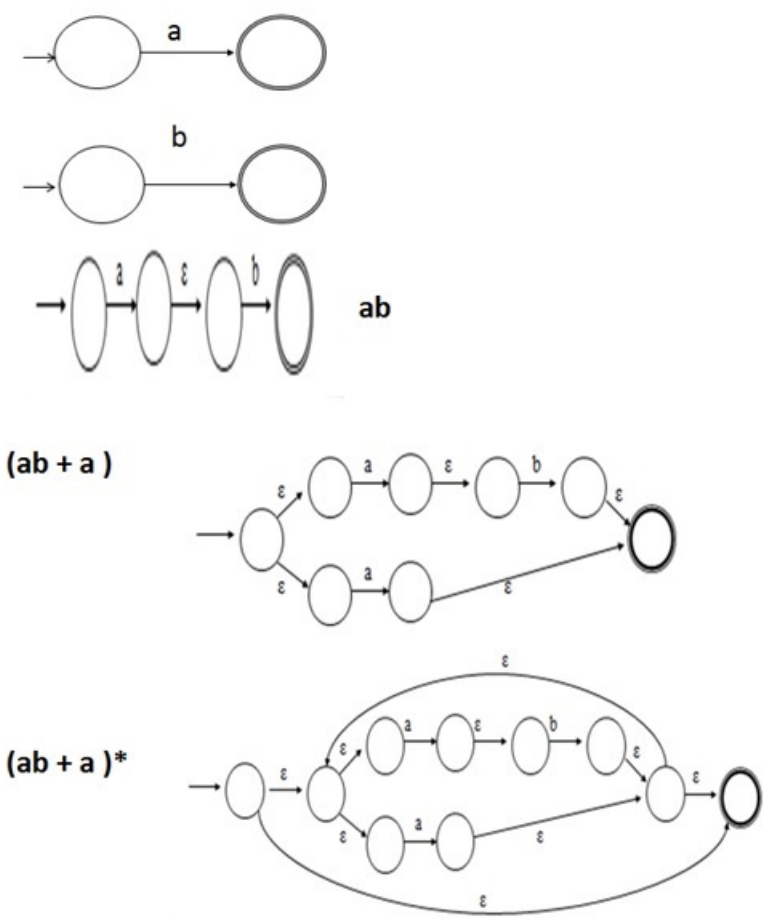


Figure 2.5: Finite Automata for (ab + a)*

2.3 Finite Automata to Regular Expressions.

Arden's Theorem:

The Arden's Theorem states that R = Q + RP, and gets reduced to R = QP*.

The Arden's Theorem is useful for the conversion of DFA to a regular expression. Following algorithm is used to build the regular expression form given DFA.

1. Let q1 be the initial state.

2. There are q2, q3, q4 ....qn number of states. The final state may be some qj where j $\leq$n.

3. Let ji represents the transition from qj to qi.

4. Calculate qi such that
   qi = ji * qj
   If qj is a start state then we have: qi = ji * qj + ε

5. Similarly, compute the final state which ultimately gives the regular expression 'r'.

Construct the regular expression for the given DFA

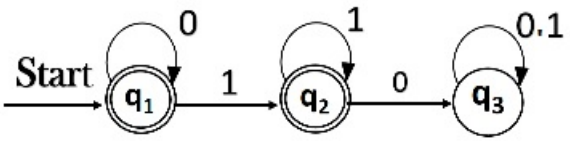


Figure 2.6: Finite Automata to Regular Expressiong by Arden's theorem

q1 = q1 0 + ε
q2 = q1 1 + q2 1
q3 = q2 0 + q3 (0+1)

q1 = q1 0 + ε
q1 = ε+ q1 0
The Arden's Theorem states that R = Q + RP, and gets reduced to R = QP*.
Assuming R = q1, Q = ε, P = 0
We get

q1 = .(0)* q1 = 0* (.R*= R*)

q2 = 0* 1 + q2 1

q2 = 0* 1 (1)* (R = Q + RP →Q P*)

The regular expression is given by

r = q1 + q2

= 0* + 0* 1.1*

r = 0* + 0* 1^+ ($1.1^* = 1^+$)

Find regular expression for the following DFA using Arden's Theorem-

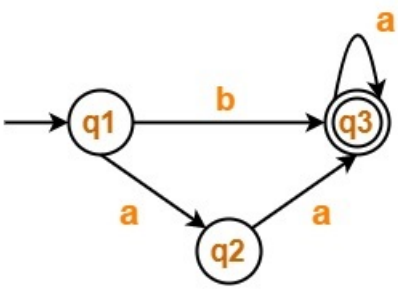


Figure 2.7: Finite Automata to Regular Expressiong by Arden's theorem

Form a equation for each state-

q1 = ε……(1)

q2 = q1.a ……(2)

q3 = q1.b + q2.a + q3.a …….(3)

Using (1) in (2), we get-

q2 = ε.a

q2 = a …….(4)

Using (1) and (4) in (3), we get-

q3 = q1.b + q2.a + q3.a

q3 = .b + a.a + q3.a

q3 = (b + a.a) + q3.a …….(5)

Using Arden's Theorem in (5), we get-
q3 = (b + a.a)a*

Thus, Regular Expression for the given DFA = (b + aa)a*

2.4 Pumping Lemma for Regular Expressions.

Pumping Lemma:
Let L be a regular language. Then there exists a constant 'c' such that for every string w in L −
|w| ≥c
We can break w into three strings, w = xyz, such that −
|y| > 0
|xy| ≤c
For all k ≥0, the string xy^kz is also in L.
Applications of Pumping Lemma
Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If L is regular, it satisfies Pumping Lemma.
- If L does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular: At first, we have to assume that L is regular.
So, the pumping lemma should hold for L.
Use the pumping lemma to obtain a contradiction −

1. Select w such that |w| ≥c

2. Select y such that |y| ≥1

3. Select x such that |xy| ≤c

4. Assign the remaining string to z.

5. Select k such that the resulting string is not in L.

Hence L is not regular.

Prove that L = { $a^i b^i$ | i ≥0 } is not regular.

- At first, we assume that L is regular and n is the number of states.
- Let w = $a^n b^n$. Thus |w| = 2n ≥n.
- By pumping lemma, let w = xyz, where |xy| ≤n.
- Let x = a^p, y = a^q, and z = $a^r b^n$, where p + q + r = n, p ≠0, q ≠0, r ≠0. Thus |y| ≠0.
- Let k = 2. Then $xy^2z = a^p a^{2q} a^r b^n$
- Number of as = (p + 2q + r) = (p + q + r) + q = n + q

Hence,$xy^2z = a^{n+q} b^n$. Since q ≠0, xy^2z is not of the form $a^n b^n$.
Thus, xy^2z is not in L. Hence L is not regular.

2.5 Closure Properties of Regular Languages -Automata

Closure Properties used in Regular Languages are as follows:
- Union - If L1 and L2 are regular languages, then their union L1 U L2 is also a regular language.
- Concatenation-The Concatenation operation of two regular languages is also regular.
- Complementation-The complement of two regular language is also regular
- Intersection-The set of regular languages are closed under intersection.
- Reversal-The set of regular languages are closed under reversal.
- Difference- If L1 and L2 are regular languages then L1-L2 is also regular
- Homomorphism:

The Homomorphism theorem depicts that a single letter is replaced with a string. If h is a homomorphism on alphabet ∑ and w = a1a2……..an is a string of symbols in ∑, then

h(w) = h(a1)h(a2)……..h(an)

If L is a language over alphabet ∑ then its homomorphism h is defined as:

h(L) = h(w) : w ∈ L

• Inverse Homomorphism-The inverse homomorphism theorem states that if h is a homomorphism from alphabet ∑ to alphabet A and L is a regular language on A then h'(L) is also a regular language.

2.6 Regular Grammars

Grammar:

• A Grammar is a 4-tuple such that-G = (V , T , P , S)where-

• V = Finite non-empty set of non-terminal symbols

• T = Finite set of terminal symbols

• P = Finite non-empty set of production rules

• S = Start symbol

A Grammar is mainly composed of two basic elements-Terminal symbols,Non-terminal symbols

1. Terminal Symbols-

• Terminals are the basic symbols from which strings are formed.

• Terminal symbols are denoted by using small case letters such as a, b, c..etc.

2. Non-Terminal Symbols-

• Non-terminals are syntactic variables that denote sets of strings.

• The non-terminals define sets of strings that help define the language generated by the grammar.

• Non-Terminal symbols are also called as auxiliary symbolsor variables.

• Non-Terminal symbols are denoted by using capital letters such as A, B, C ..etc

• A regular grammaris a formal grammar that is right-regular or left-regular.

• Every regular grammar describes a regular language.

TYPES

1.Right linear regular grammar:

• A right regular grammar(also called right linear grammar) is a formal grammar (N, $\sum$, P, S) such that all the production rules in P are of one of the following forms:

• A $\rightarrow$a, where Ais a non-terminal in N and a is a terminal in $\sum$

• A $\rightarrow$aB, where A and B are non-terminals in N and a is in $\sum$

• A $\rightarrow\varepsilon$, where Ais in Nand εdenotes the empty string, i.e. the string of length 0 2.Left linear regular grammar:

• In a left regular grammar(also called left linear grammar), all rules obey the forms:

• A$\rightarrow$a, where A is a non-terminal in N and a is a terminal in $\sum$

• A$\rightarrow$Ba, where A and B are in N and a is in $\sum$.

• A$\rightarrow\varepsilon$, where A is in N and $\varepsilon$is the empty string.

Example for Right linear regular grammar: A $\rightarrow$a, A $\rightarrow$aB, A$\rightarrow\varepsilon$

where, A and B are non-terminals,

a is terminal, and $\varepsilon$is empty string

S $\rightarrow$00B | 11S

B $\rightarrow$0B | 1B | 0 | 1

where, S and B are non-terminals, and

0 and 1 are terminals

Example for Left linear regular grammar:

A $\rightarrow$a, A$\rightarrow$Ba, A $\rightarrow\varepsilon$

where, A and B are non-terminals,

a is terminal, and $\varepsilon$is empty string

S$\rightarrow$B00 | S11

B $\rightarrow$B0 | B1 | 0 | 1

where S and B are non-terminals, and 0 and 1 are terminals

2.7 Regular Grammars and Finite Automata conversions

Convert the following Finite Automata to Regular Grammars.

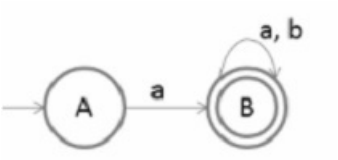


Figure 2.8: Finite Automata to Regular Grammar example1

Pick the start state A and output is on symbol 'a' going to state B

A→aB and A →ε, since B is final state

Now we will pick state B and then we will go on each output

i.e B →aB , B →bB , B →εsince B is final state

Therefore, Final right linear grammar is as follows −

A →aB/ε, B→aB/bB/ε

Convert the following Finite Automata to Regular Grammars.

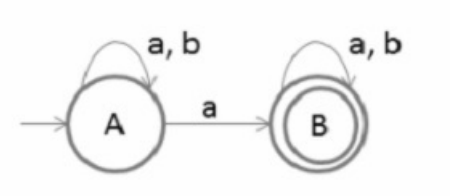


Figure 2.9: Finite Automata to Regular Grammar example2

Start with state A which is an initial state the output on symbol 'a' goes to A and B and the output on symbol 'b' goes to A. Now the production rule of right linear grammar is −

A →aA/bA/aB/εsince B is final state

Now pick state B and then go on each output the right linear grammar is −

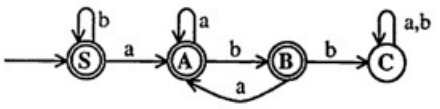
B →aB/bB/εsince B is final state

The final right linear grammar for the given finite automata is −

A →aA/bA/aB/ε

B →aB/bB/ε

Convert the following Finite Automata to Regular Grammars.



Transitions	Productions
$\delta(S, a) = A$	S → aA
$\delta(S,b)=S$	S → bS
$\delta(A,a)=A$	A → aA
$\delta(A, b) = B$	A → bB
$\delta(B,a)=A$	B → aA
$\delta(B,b)=C$	B → bC
$\delta(C, a) = C$	C → aC
$\delta(C, b) = C$	C → bC

Grammar G = (V, T, P, S) where
V = { S, A, B, C }
T = { a, b }
P = {
S → aA | bS | ∈
A → aA | bB | ∈
B → aA | bC | ∈
C → aC | bC
}
S is the start symbol

Figure 2.10: Finite Automata to Regular Grammar example3

Construct a DFA to accept the language generated by the following grammar
S →01A , A →10B , B →0A/11

Note that for each production of the form A →wB, the corresponding transition will be δ(A,w)=B and for each production of the form A →w, the corresponding transition will be δ(A,w)=qf (final state).

Productions			Transitions
S	→	01A	$\delta(S,\ 01) = A$
A	→	10B	$\delta(A,\ 10) = B$
B	→	0A	$\delta(B,\ 0) = A$
B	→	11	$\delta(B,\ 11) = q_f$

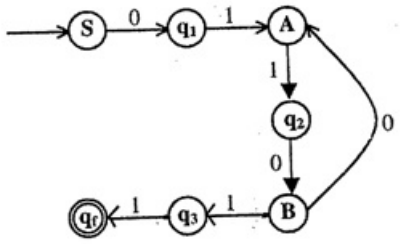
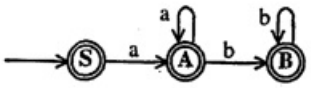


Figure 2.11: Regular Grammar to Finite Automata example1

Construct a DFA to accept the language generated by the following grammar
S →aA/ ε, A →aA/bB/ ε, B →bB/ ε

Productions			Transitions
S	→	aA	$\delta(S,a) = A$
S	→	$\in$	S is the final state
A	→	aA	$\delta(A,a) = A$
A	→	bB	$\delta(A,b) = B$
A	→	$\in$	A is the final state
B	→	bB	$\delta(B, b) = B$
B	→	$\in$	B is the final state.



So, the DFA $M = (Q, \Sigma, \delta, q_0, A)$ where
$Q = \{S, A, B\}$, $\Sigma = \{a, b\}$
$q_0 = S$, $A = \{S, A, B\}$
δ is as obtained from the above table .

Figure 2.12: Regular Grammar to Finite Automata example 2

Chapter 3

CONTEXT FREE GRAMMARS

Course Outcomes

After successful completion of this module, students should be able to:

CO 3	Illustrate the pumping lemma on regular and context free languages on regular and context free languages for performing negative test.	Understand
CO 4	Demonstrate context free grammars, normal forms for generating patterns of strings and minimize the ambiguity in parsing the given strings.	Understand

3.1 Introduction CFG

A Context-Free Grammar (CFG) consisting of a finite set of grammar rules is a quadruple (V, T, P, S) or (V, $\sum$, P, S) where

- V is a finite set of non-terminal symbols / variables.(Each variable represents a set of strings)
- T is a finite set of terminals where V $\cap$T = NULL. (i.e., the symbols that form the strings of the language being defined)
- P is a finite set of rules or productions, P: V $\rightarrow$(V U T)* (represent the recursive definition of the language.)

• S is the start symbol that represents the language being defined. (Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol.)

Each production is in the form of head body, where head is a variable, and body is a string of zero or more terminals and variables.

1. G= ({S}, {a, b}, P, S), where

P: S →aSa

S →bSb or S →aSa | bSb | ε

S →ε

2. G= ({S, F}, {0, 1}, P, S), where

P: S →00S | 11F

F →00F | ε

3.2 Derivation Tree

Derivations:

The productions of a grammar are used to derive strings.

One Step Derivation or Direct Derivation

v is one-step derivable from u, written u →v, if:

u = x αz

v = x βz if $\alpha \rightarrow \beta$in P

and say than x αz derives x βz or x αz yields x βz

Example:

A →aAa | B

B →bB | ε

Sample Derivation:

A →aAa

→aaAaa

→aaaAaaa

→aaaBaaa

→aaabBaaa

→aaabbBaaa

→aaabbaaa

Zero or more derivation steps:

u ⇒* v (read: u derives v) if u = v or there is a sequence u1, u2, … uk, k ≥0, such that

u →u1 →u2 →…. →uk →v

A →aAa | B

B →bB |ε

A ⇒* aaabbaaaa

⇒(single step derivations)

⇒k (k step derivations)

⇒* (derivations of 0 or more steps)

Derivation of aaabbaaa:

A →aAa

→aaAaa

→aaaAaaa

→aaaBaaa

→aaabBaaa

→aaabbBaaa

→aaabbaaa

Language specified by G

If G = (V, T, P, S) is a CFG then the language specified by G (or the language of G) is a Context Free Language denoted as

L(G) = { w ∈T* | S ⇒* w }

i.e., the set of strings over T derivable from the start symbol S, and the elements of L(G) are called sentence.

Examples:

G = ({S}, {0,1}. {S →0S1 | ε}, S)

$L(G) = \{0^n 1^n \mid n > 0\}$

All strings of balanced parentheses

$P \rightarrow \varepsilon \mid (P) \mid PP$

$P \rightarrow \varepsilon \mid \{ P \} \mid PP$

Parse Tree or Derivation Tree:

- The sequence of substitutions used to obtain a string using a CFG is called a derivation and may be represented by a tree called derivation tree or a parse tree
- A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information of a string derived from a context-free grammar.
- Root label = start node
- Each interior label = variable.
- Each parent/child relation = derivation step.
- Each leaf label = terminal or ε.
- The leaves of a parse tree when read left to right, are called the frontier or yield of the tree.

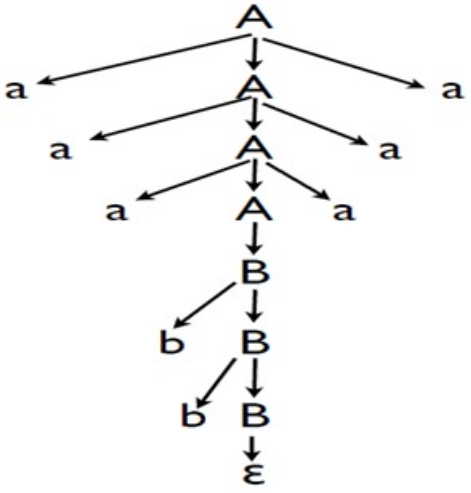


Figure 3.1: Derivation Tree

Sentential Form and Partial Derivation Tree:A partial derivation tree is a sub-tree of a parse tree such that either all its children are in the sub-tree or none of them are in the

sub-tree.

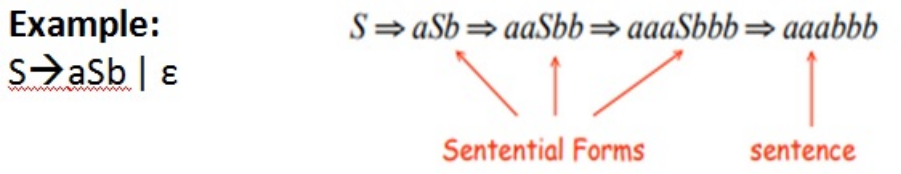


Figure 3.2: Sentential Form

Derivation or Yield of a Tree:The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls.

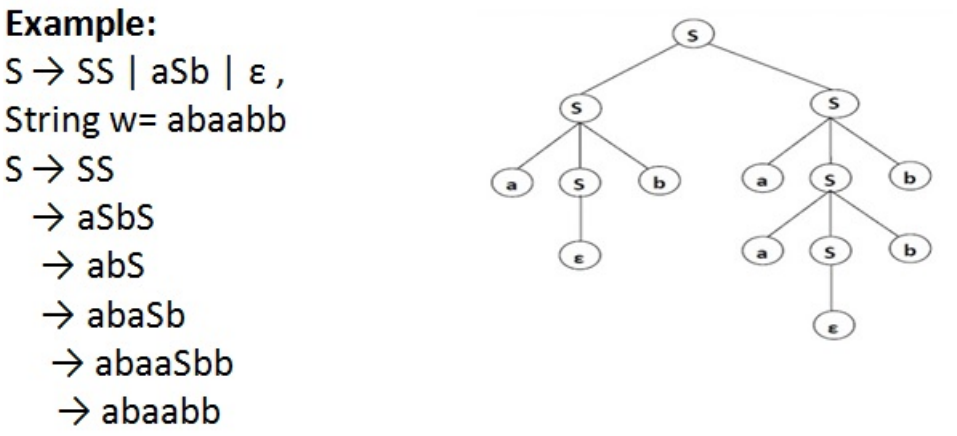


Figure 3.3: Derivation or Yield of a Tree

Leftmost and Rightmost derivation:

- A derivation is basically a sequence of production rules, in order to get the input string.
- During derivation, we may have many choices to replace the nonterminal in the body.

Example:

Production rules:

X →X + X

X →X * X

X →a

String: a + a * a

Leftmost derivation:-Choose first X from Left Side X →X * X

Rightmost derivationChoose first X from Right Side X →X * X

To decide which non-terminal to be replaced with production rule, we can have two options.

- Leftmost Derivation →chooses the leftmost nonterminal to expand
- Rightmost Derivation→chooses the rightmost nonterminal to expand

The left-most derivation is:

X →X * X

X →X + X* X

X →a + X * X

X →a + a * X

X →a + a * a

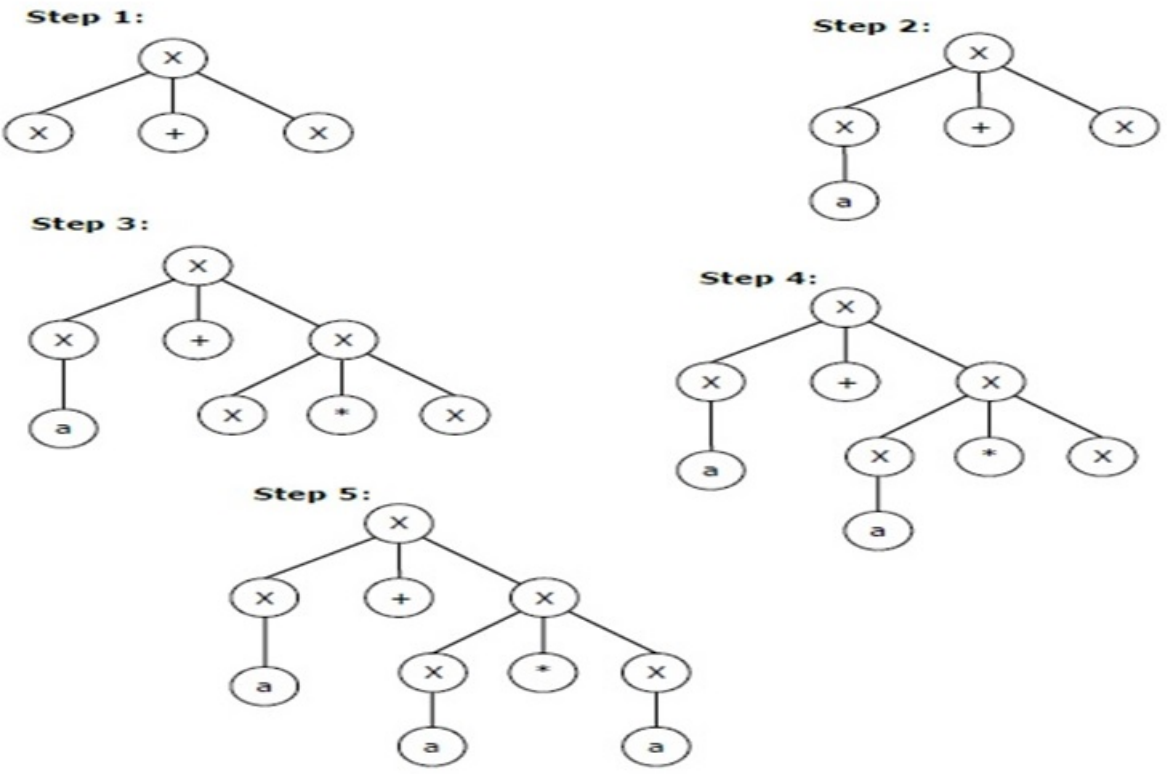


Figure 3.4: Derivation Tree for leftmost derivation

The right-most derivation is:

X →X * X

X →X * a

X →X + X * a

X →X + a * a

X →a + a * a

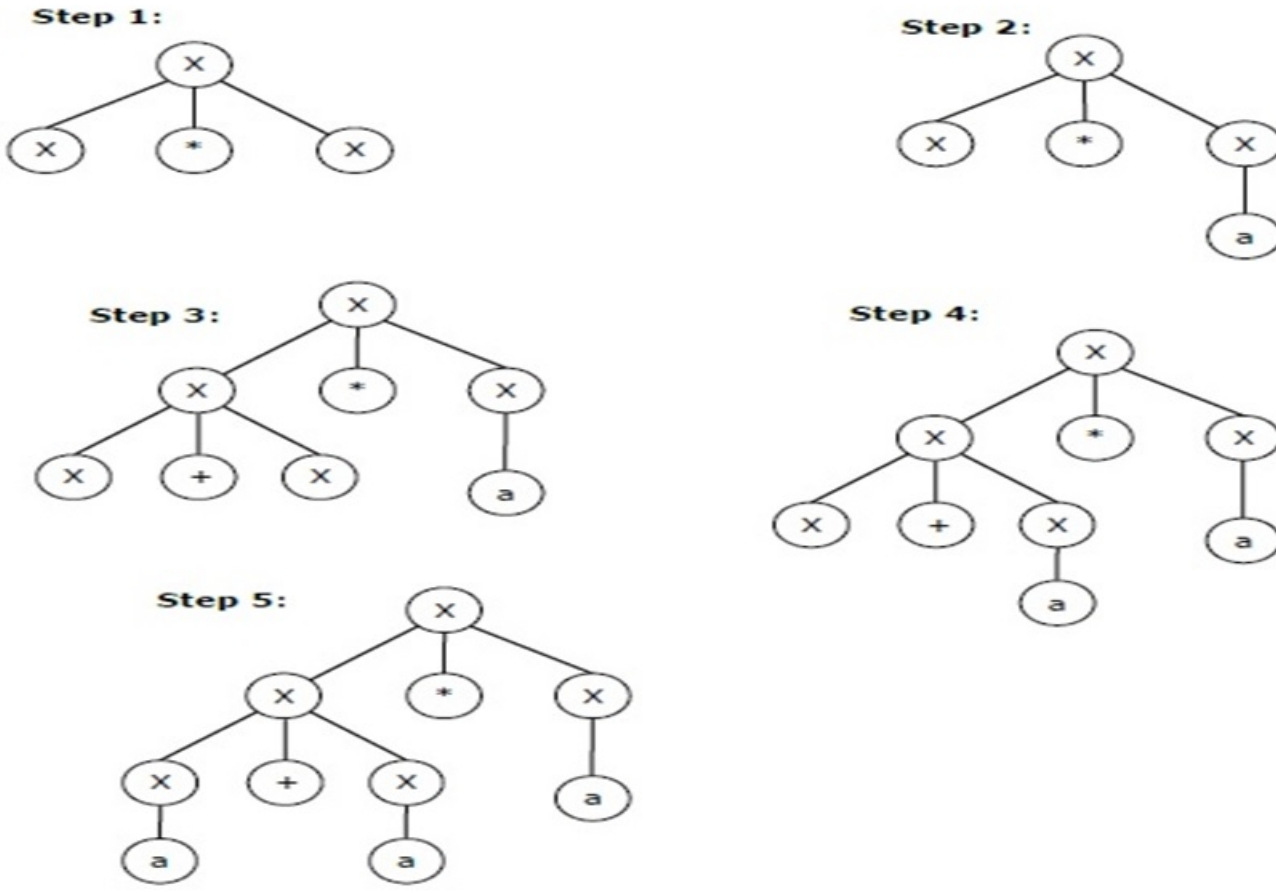


Figure 3.5: Derivation Tree for rightmost derivation

Graphical representation for a derivation -

- Filters out choice regarding replacement order
- Rooted by the start symbol S
- Interior nodes represent nonterminals in N
- Leaf nodes are terminals in T or ε
- Node A can have children X1 X2 ... Xn if a rule A→X1 X2 ... Xn exists

Ambiguity in context free grammars:-

- A grammar is said to be ambiguous, if It permits a terminal string to have more than one parse tree i.e. more than one leftmost derivation or more than one rightmost derivative for the same string.
- If the grammar is not ambiguous then it is called unambiguous

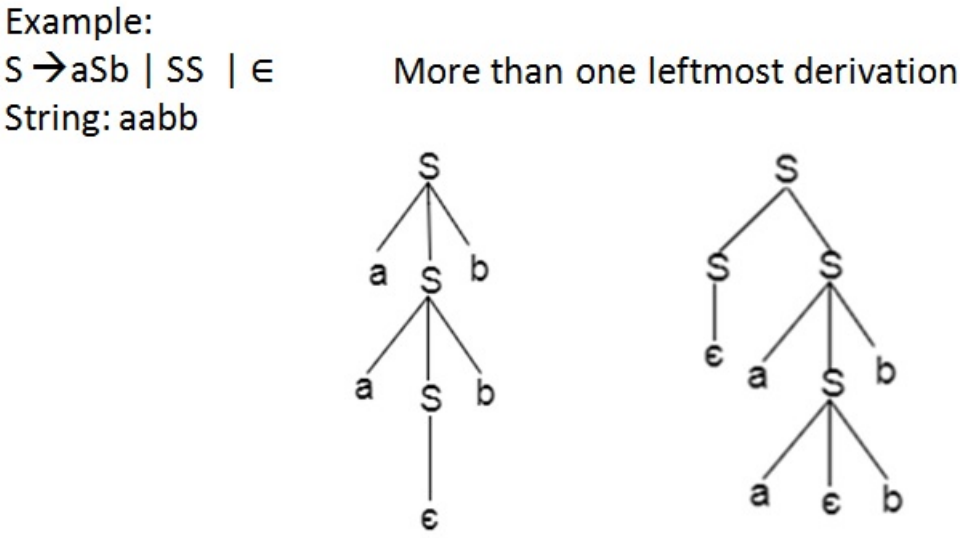


Figure 3.6: Ambiguity in context free grammars example 1

Example: E →E+E | E*E | id , Prove that grammar is ambiguous or not?
consider String: id+id*id

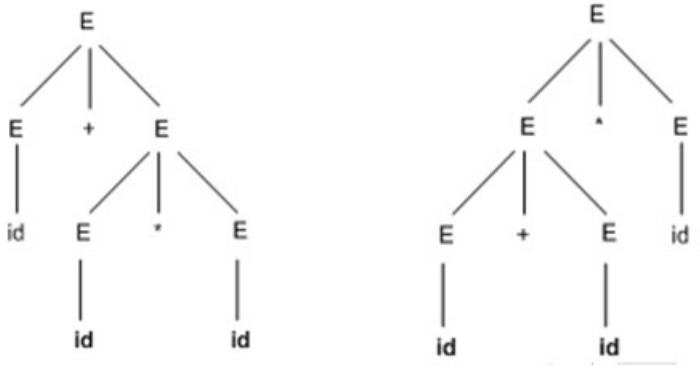


Figure 3.7: Ambiguity in context free grammars example 2

Two left most derivations possible, so this grammar is ambiguous grammar.

3.3 Minimization

Reduction of CFG:

Phase 1 − Derivation of an equivalent grammar, G', from the CFG, G, such that each variable derives some terminal string.

Derivation Procedure −

Step 1 − Include all symbols, W1, that derive some terminal and initialize i=1.

Step 2 − Include all symbols, Wi+1, that derive Wi.

Step 3 − Increment i and repeat Step 2, until Wi+1 = Wi.

Step 4 − Include all production rules that have Wi in it.

Phase 2 − Derivation of an equivalent grammar, G", from the CFG, G', such that each symbol appears in a sentential form.

Derivation Procedure −

Step 1 − Include the start symbol in Y1 and initialize i = 1.

Step 2 − Include all symbols, Yi+1, that can be derived from Yi and include all production rules that have been applied.

Step 3 − Increment i and repeat Step 2, until Yi+1 = Yi.

Find a reduced grammar equivalent to the grammar G, having production rules, P: S → AC | B, A → a, C → c | BC, E → aA | e Solution

Phase 1 −

T = { a, c, e }

W1 = { A, C, E } from rules A → a, C → c and E → aA

W2 = { A, C, E } U { S } from rule S → AC

W3 = { A, C, E, S} U ϕ

Since W2 = W3, we can derive G' as −

G' = { { A, C, E, S }, { a, c, e}, P, {S} }

where P: S → AC, A → a, C → c , E → aA | e

Phase 2 −

Y1 = { S }

Y2 = { S, A, C } from rule S → AC

Y3 = { S, A, C, a, c } from rules A → a and C → c

Y4 = { S, A, C, a, c }

Since Y3 = Y4, we can derive G" as −

G" = { { A, C, S }, { a, c }, P, {S} }

where P: S → AC, A → a, C → c

Removal of Unit Productions:

Any production rule in the form A → B where A, B Non-terminal is called unit production.

Removal Procedure −

Step 1 − To remove A → B, add production A → x to the grammar rule whenever B → x occurs in the grammar. [x Terminal, x can be Null]

Step 2 − Delete A → B from the grammar.

Step 3 − Repeat from step 1 until all unit productions are removed.

Problem:Remove unit production from the following −

S → XY, X → a, Y → Z | b, Z → M, M → N, N → a

Solution −

There are 3 unit productions in the grammar −

Y → Z, Z → M, and M → N

At first, we will remove M → N.

As N → a, we add M → a, and M → N is removed.

The production set becomes

S → XY, X → a, Y → Z | b, Z → M, M → a, N → a

Now we will remove Z → M.

As M → a, we add Z→ a, and Z → M is removed.

The production set becomes

S → XY, X → a, Y → Z | b, Z → a, M → a, N → a

Now we will remove Y → Z.

As Z → a, we add Y→ a, and Y → Z is removed.

The production set becomes

S → XY, X → a, Y → a | b, Z → a, M → a, N → a

Now Z, M, and N are unreachable, hence we can remove those.

The final CFG is unit production free − S → XY, X → a, Y → a | b

Removal of Null Productions

In a CFG, a non-terminal symbol 'A' is a nullable variable if there is a production A → εor there is a derivation that starts at A and finally ends up withε: A → ..... → ε

Removal Procedure

Step 1 − Find out nullable non-terminal variables which derive ε.

Step 2 − For each production A → a, construct all productions A → x where x is obtained from 'a' by removing one or multiple non-terminals from Step 1.

Step 3 − Combine the original productions with the result of step 2 and remove ε- productions.

Problem: Remove null production from the following −

S → ASA | aB | b, A → B, B → b | ε

Solution −

There are two nullable variables − A and B

At first, we will remove B → ε.

After removing B →ε, the production set becomes −

S→ASA | aB | b | a, A → B| b | ε, B → b

Now we will remove A → ε.

After removing A → ε, the production set becomes −

S→ASA | aB | b | a | SA | AS | S, A → B| b, B → b

This is the final production set without null transition

3.4 Chomsky Normal Form

A CFG is in Chomsky Normal Form if the Productions are in the following forms −

A → a

A → BC

S → ε

where A, B, and C are non-terminals and a is terminal.

Algorithm to Convert into Chomsky Normal Form −

Step 1 − If the start symbol S occurs on some right side, create a new start symbol S' and a new production S'→ S.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Replace each production A → B1…Bn where n > 2 with A → B1C where C → B2 …Bn. Repeat this step for all productions having two or more symbols in the right side.

Step 5 − If the right side of any production is in the form A → aB where a is a terminal and A, B are non-terminal, then the production is replaced by A → XB and X → a. Repeat this step for every production which is in the form A → aB.

Problem

Convert the following CFG into CNF

S → ASA | aB, A → B | S, B → b | ε

Solution

(1) Since S appears in R.H.S, we add a new state S0 and S0→S is added to the production set and it becomes −

S0→S, S→ ASA | aB, A → B | S, B → b | ε

(2) Now we will remove the null productions −

B → $\varepsilon$and A → ε

After removing B → ε, the production set becomes −

S0→S, S→ ASA | aB | a, A → B | S | ε, B → b

After removing A → ε, the production set becomes −

S0→S, S→ ASA | aB | a | AS | SA | S, A → B | S, B → b

(3) Now we will remove the unit productions.

After removing S → S, the production set becomes −

S0→S, S→ ASA | aB | a | AS | SA, A → B | S, B → b

After removing S0→ S, the production set becomes −

S0→ ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → B | S, B → b

After removing A→ B, the production set becomes −

S0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → S | b

B → b

After removing A→ S, the production set becomes −

S0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → b |ASA | aB | a | AS | SA, B → b

(4) Now we will find out more than two variables in the R.H.S

Here, S0→ ASA, S → ASA, A→ ASA violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF −

S0→ AX | aB | a | AS | SA

S→ AX | aB | a | AS | SA

A → b |AX | aB | a | AS | SA

B → b

X → SA

(5) We have to change the productions S0→ aB, S→ aB, A→ aB

And the final production set becomes −

S0→ AX | YB | a | AS | SA

S→ AX | YB | a | AS | SA

A → b A → b |AX | YB | a | AS | SA

B → b

X → SA

Y → a

3.5 Greibach Normal Form

A CFG is in Greibach Normal Form if the Productions are in the following forms −

A → b

A → bD1…Dn

S → ε

where A, D1,....,Dn are non-terminals and b is a terminal.

Algorithm to Convert a CFG into Greibach Normal Form

Step 1 − If the start symbol S occurs on some right side, create a new start symbol S' and a new production S' → S.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Remove all direct and indirect left-recursion.

Step 5 − Do proper substitutions of productions to convert it into the proper form of GNF.

Problem

Convert the following CFG into CNF

S → XY | Xn | p

X → mX | m

Y → Xn | o

Solution

Here, S does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Step 4

Now after replacing

X in S → XY | Xo | p

with

mX | m

we obtain

S → mXY | mY | mXo | mo | p.

And after replacing

X in Y → Xn | o

with the right side of

X → mX | m

we obtain

Y → mXn | mn | o.

Two new productions O → o and P → p are added to the production set and then we came to the final GNF as the following −

S → mXY | mY | mXC | mC | p

X → mX | m

Y → mXD | mD | o

O → o

P → p

3.6 Pumping Lemma for CFG

Lemma

If L is a context-free language, there is a pumping length p such that any string w ∈L of length p can be written as w = uvxyz, where vy ≠ε, |vxy| p, and for all i 0, uvixyiz ∈L.

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language L ={$x^n y^n z^n$ | n ≥1} is context free or not.

Solution

Let L is context free. Then, L must satisfy pumping lemma.

At first, choose a number n of the pumping lemma. Then, take z as $0^n 1^n 2^n$.

Break z into uvwxy, where

|vwx| ≤n and vx ≠ε.

Hence vwx cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − vwx has no 2s. Then vx has only 0s and 1s. Then uwy, which would have to be in L, has n 2s, but fewer than n 0s or 1s.

Case 2 − vwx has no 0s.

Here contradiction occurs.

Hence, L is not a context-free language.

Closure Properties of Context Free Languages:

Union : If L1 and L2 are two context free languages, their union L1 U L2 will also be context free

Concatenation : If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free.

Kleene Closure : If L1 is context free, its Kleene closure L1* will also be context free

Intersection and complementation : If L1 and If L2 are two context free languages, their intersection L1 L2 need not be context free, complementation of context free language L1 which is $\sum^*$ – L1, need not be context free.

Chapter 4

PUSHDOWN AUTOMATA

Course Outcomes

After successful completion of this module, students should be able to:

CO 5	Construct push down automata for context free languages for developing parsing phase of a compiler.	Apply

4.1 Pushdown Automata Introduction

Basic Structure of PDA:

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is − "Finite state machine" + "a stack"

A pushdown automaton has three components −

- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations −

- Push − a new symbol is added at the top.
- Pop − the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

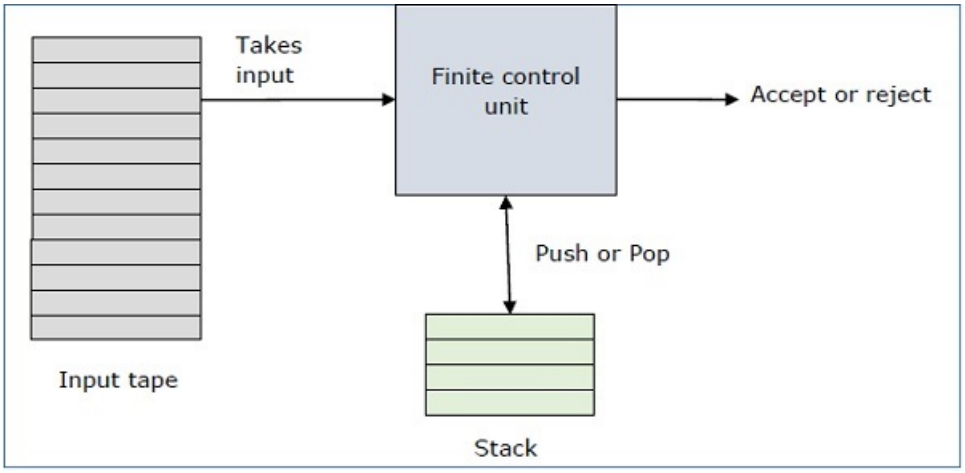


Figure 4.1: Basic Structure of PDA

A PDA can be formally described as a 7-tuple (Q, ∑, S, δ, q0, I, F) −

- Q is the finite number of states
- ∑ is input alphabet
- S is stack symbols
- $\delta$is the transition function: Q X (∑ U { ε}) X S → Q X S*
- q0 is the initial state (q0 ∈Q)
- I is the initial stack top symbol (I ∈S)
- F is a set of accepting states (F ∈Q)

The following diagram shows a transition in a PDA from a state q1 to state q2, labeled as a,b → c −

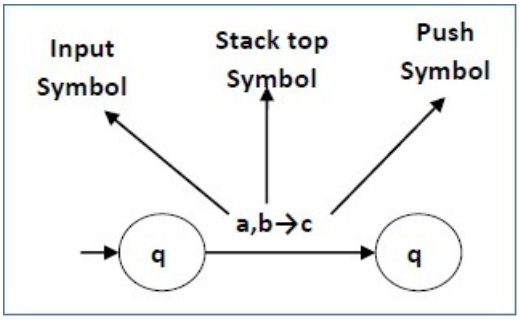


Figure 4.2: Transition in a PDA

This means at state q1, if we encounter an input string 'a' and top symbol of the stack is 'b', then we pop 'b', push 'c' on top of the stack and move to state q2.

Terminologies Related to PDA:

Instantaneous Description:

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where • q is the state

- w is unconsumed input
- s is the stack contents

Turnstile Notation:

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol " ⊢".

Consider a PDA (Q, ∑, S, δ, q0, I, F) . A transition can be mathematically represented by the following turnstile notation −

(p, aw, Tβ) ⊢(q, w, b)

This implies that while taking a transition from state p to state q, the input symbol 'a' is consumed, and the top of the stack 'T' is replaced by a new string ' '.

Note − If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

There are two different ways to define PDA acceptability.

Final State Acceptability:
In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA (Q, $\sum$, S, δ, q0, I, F), the language accepted by the set of final states F is −
L(PDA) = {w | (q0, w, I) $\vdash^*$ (q,ε, x), q ∈F }
for any input stack string x.

Empty Stack Acceptability:
Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.
For a PDA (Q, $\sum$, S, δ, q0, I, F), the language accepted by the empty stack is −
L(PDA) = w | (q0, w, I) $\vdash^*$ (q, ε, ε), q ∈Q

4.2 Pushdown Automata Problems

Example 1 :Construct PDA for 0^n1^n, n≥0

M = ({q1, q2}, {0, 1}, {L, #}, δ, q1, #, ϕ)

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

δ:

(1) $\delta(q_1, 0, \#) = \{(q_1, L\#)\}$ // stack order: L on top, then # below

(2) $\delta(q_1, 1, \#) = \emptyset$ // illegal, string rejected, *When will it happen?*

(3) $\delta(q_1, 0, L) = \{(q_1, LL)\}$

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

(5) $\delta(q_2, 1, L) = \{(q_2, \varepsilon)\}$

(6) $\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$ //if ε read & stack hits bottom, accept

(7) $\delta(q_2, \varepsilon, L) = \emptyset$ // illegal, string rejected

(8) $\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$ // n=0, accept

- **0011**
- ***(q1, 0 011, #) |-***
  (q1, 0 11, L#) |-
  (q1, 1 1, LL#) |-
  (q2, 1, L#) |-
  (q2, e, #) |-
  (q2, e, e): accept
- ***011***
- ***(q1, 0 11, #) |-***
  (q1, 1 1, L#) |-
  (q2, 1, #) |-
  Ø : reject

Figure 4.3: PDA example1

Example 2 :Construct PDA for balanced parentheses

M = ({q1}, {"(", ")" }, {L, #},δ, q1, #, ϕ)

δ:

(1) $\delta(q_1, (, \#) = \{(q_1, L\#)\}$ // stack order: L-on top-then-#lower

(2) $\delta(q_1,), \#) = \emptyset$ //illegal, string rejected

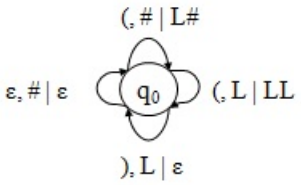
(3) $\delta(q_1, (, L) = \{(q_1, LL)\}$

(4) $\delta(q_1,), L) = \{(q_1, \varepsilon)\}$

(5) $\delta(q_1, \varepsilon, \#) = \{(q_1, \varepsilon)\}$ //if ε read & stack hits bottom, accept

(6) $\delta(q_1, \varepsilon, L) = \emptyset$ //illegal, string rejected

- Transition Diagram:



- Example Computation:

Current Input	Stack	Transition	
(())	#	-- initial status	
())	L#	(1)	- Could have applied rule (5), but
))	LL#	(3)	it would have done no good
)	L#	(4)	
ε	#	(4)	
ε	-	(5)	

Figure 4.4: PDA example2

Example 3:Construct PDA for the language {x | x = wcw^r and w in{0,1}*,but sigma={0,1,c}}

M = ({q1, q2}, {0, 1, c}, {#, B, G}, δ, q1, #, ϕ)

δ:

(1)	$\delta(q_1, 0, \#) = \{(q_1, B\#)\}$	(9)	$\delta(q_1, 1, \#) = \{(q_1, G\#)\}$
(2)	$\delta(q_1, 0, B) = \{(q_1, BB)\}$	(10)	$\delta(q_1, 1, B) = \{(q_1, GB)\}$
(3)	$\delta(q_1, 0, G) = \{(q_1, BG)\}$	(11)	$\delta(q_1, 1, G) = \{(q_1, GG)\}$
(4)	$\delta(q_1, c, \#) = \{(q_2, \#)\}$		
(5)	$\delta(q_1, c, B) = \{(q_2, B)\}$		
(6)	$\delta(q_1, c, G) = \{(q_2, G)\}$		
(7)	$\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$	(12)	$\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(8)	$\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$		

State	Input	Stack	Rule Applied	Rules Applicable
q_1	**0**1c10	#		(1)
q_1	**1**c10	B#	(1)	(10)
q_1	**c**10	GB#	(10)	(6)
q_2	**1**0	GB#	(6)	(12)
q_2	**0**	B#	(12)	(7)
q_2	**ε**	#	(7)	(8)
q_2	ε	ε	(8)	-

Figure 4.5: PDA example3

Example 4:Construct PDA for the language {x | x = ww^r and w in{0,1}* }

M = ({q1, q2}, { 0, 1}, {#, B, G}, δ, q1, #, ϕ)

(1)	$\delta(q_1, 0, \#) = \{(q_1, B\#)\}$	(6)	$\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$
(2)	$\delta(q_1, 1, \#) = \{(q_1, G\#)\}$	(7)	$\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$
(3)	$\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$	(8)	$\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(4)	$\delta(q_1, 0, G) = \{(q_1, BG)\}$	(9)	$\delta(q_1, \varepsilon, \#) = \{(q_2, \varepsilon)\}$
(5)	$\delta(q_1, 1, B) = \{(q_1, GB)\}$	(10)	$\delta(q_2, \varepsilon, \#) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied	Rules Applicable
q_1	000000	#		(1), (9)
q_1	00000	B#	(1)	(3), both options
q_1	0000	BB#	(3) option #1	(3), both options
q_1	000	BBB#	(3) option #1	(3), both options
q_2	00	BB#	(3) option #2	(7)
q_2	0	B#	(7)	(7)
q_2	ε	#	(7)	(10)
q_2	ε	ε	(10)	

Figure 4.6: PDA example4

4.3 Pushdown Automata and Context Free Grammars Conversions

Context Free Grammars to Pushdown Automata :

Let G = (V, T, P, S) be a CFL. If every production in P is of the form

A → aγ,Where A is in V, a is in T, and is in V*, then G is said to be in Greibach Normal Form (GNF) and there exist an equivalent Pushdown Automata.

δ(q, a, A) = {(q, γ) | A → aγ is in P and $\gamma$is in Γ*}, for all a in Σ and A in Γ, M = (Q, $\sum$, Γ, δ, q, z, ϕ)

Example 1: Consider the following CFG in GNF convert to PDA.

S → aS G is in GNF

S → a L(G) = a+

Construct M as:

Q = {q}

Σ = T = {a}

Γ= V = {S}

z = S

δ(q, a, S) = {(q, S), (q, ε)}

δ(q, ε, S) = Ø

Example 2: Consider the following CFG in GNF convert to PDA. (1) S → aA

(2) S → aB

(3) A → aA

(4) A → aB

(5) B → bB

(6) B → b

Construct M as:

Q = {q}

Σ = T = {a, b}

Γ= V = {S, A, B}

z = S

(1) δ(q, a, S) = {(q, A), (q, B)} From productions #1 and 2, S→aA, S→aB

(2) δ(q, a, A) = {(q, A), (q, B)} From productions #3 and 4, A→aA, A→aB

(3) δ(q, a, B) = Ø

(4) δ(q, b, S) = Ø

(5) δ(q, b, A) = Ø

(6) δ(q, b, B) = {(q, B), (q, ε)} From productions #5 and 6, B→bB, B→b

(7) δ(q, ε, S) = Ø

(8) δ(q, ε, A) = Ø

(9) δ(q, ε, B) = Ø

Algorithm to find PDA corresponding to a given CFG:

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, $\sum$, S, δ, q0, I, F)

Step 1 − Convert the productions of the CFG into GNF.

Step 2 − The PDA will have only one state {q}.

Step 3 − The start symbol of CFG will be the start symbol in the PDA.

Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 − For each production in the form A → aX where a is terminal and A, X are combination of terminal and non-terminals, make a transition δ(q, a, A).

Problem -Construct a PDA from the following CFG.

G = ({S, X}, {a, b}, P, S)

where the productions are −

S → XS | ε, A → aXb | Ab | ab

Solution

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ–

δ(q, ε, S) = {(q, XS), (q, ε)}

δ(q, ε, X) = {(q, aXb), (q, Xb), (q, ab)}

δ(q, a, a) = {(q, ε)}

δ(q, 1, 1) ={(q, ε)}

Pushdown Automata to Context Free Grammars:

Step 1 : Writing the production function for the starting symbol

s $\rightarrow$ (q0,z0,q) forall q $\in$Q

Step 2 : δ(q,a,z0) = (q1,z1,z2,..zm)

(q,z0,q') $\rightarrow$ a(q1,z1,q1)(q1,z2,q2)(q2,z3,q3)....(qm,zm,q')

Step 3:δ(q1,a,z) =(q2,ε)

(q1,z,q2) $\rightarrow$ a

Example :

M = ({q1, q2}, { 0, 1}, {x,z0}, δ, q0, z0, ϕ)

δ(q0,0,z0) = (q0,xz0)

δ(q0,0,x) = (q0,xx)

δ(q0,1,x) = (q1,ε)

δ(q1,1,x) = (q1,ε)

δ(q1,ε,x) = (q1,ε)

δ(q1,ε,z0) = (q1,ε)

(q0,0,z0) , (q0,0,x) , (q0,1,z0) , (q0,1,x) , (q1,0,z0) , (q1,0,x) , (q1,1,z0) , (q1,1,x)

Step 1 :

S $\rightarrow$ (q0,z0,q0) S $\rightarrow$ (q0,z0,q1)

Step 2 :

δ(q0,0,z0) = (q0,xz0)

(q0,z0,q0) = 0(q0,x,q0)(q0,z0,q0)

(q0,z0,q0) = 0(q0,x,q1)(q1,z0,q0)

(q0,z0,q1) = 0(q0,x,q0)(q0,z0,q1)

(q0,z0,q1) = 0(q0,x,q1)(q1,z0,q1)

δ(q0,0,x) = (q0,xx)

(q0,x,q0) = 0(q0,x,q0)(q0,x,q0)

(q0,x,q0) = 0(q0,x,q1)(q1,x,q0)

(q0,x,q1) = 0(q0,x,q0)(q0,x,q1)

(q0,x,q1) = 0(q0,x,q1)(q1,x,q1)

Step 3 :

δ(q0,1,x) = (q1,ε)

(q0,x,q1)=1

δ(q1,1,x) = (q1,ε)

(q1,x,q1)=1

δ(q1,ε,x) = (q1,ε)

(q1,x,q1)=ε

δ(q1,ε,z0) = (q1,ε)

By simiplification of above productions will result as :

S $\rightarrow$ (q0,z0,q1)

(q0,z0,q1) = 0(q0,x,q1)(q1,z0,q1)

(q0,x,q1) = 0(q0,x,q1)(q1,x,q1)

(q0,x,q1)=1

(q1,x,q1)=1

(q1,x,q1)=ε

(q1,z0,q1)=ε

4.4 Deterministic context free languages and Deterministic pushdown automata.

Deterministic pushdown automata:

Two transitions ((p1, u1, β1), (q1, γ1)) and ((p2, u2, β2), (q2, γ2)) are compatible if

1. p1 = p2,

2. u1 and u2 are compatible (which means that u1 is a prefix of u2 or that u2 is a prefix of u1),

3. β1 and β2 are compatible.

A pushdown automaton is deterministic if for every pair of compatible transitions, theses transitions are identical.

Deterministic pushdown automaton (DPDA or DPA) is a variation of the pushdown automaton. The class of deterministic pushdown automata accepts the deterministic context-free languages, a proper subset of context-free languages. A deterministic pushdown automaton has at most one legal transition for the same combination of input symbol, state, and top stack symbol.

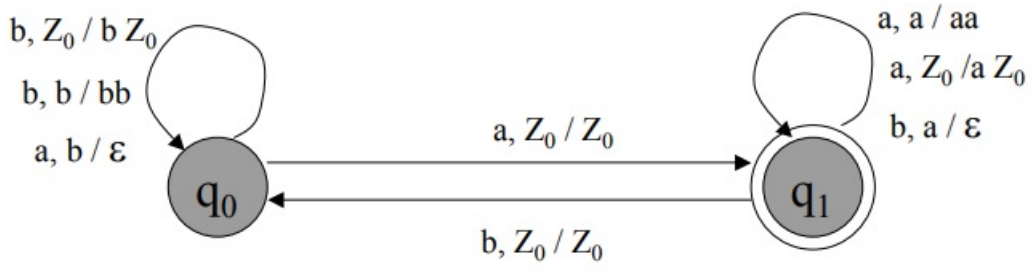


Figure 4.7: Deterministic pushdown automaton

Non deterministic pushdown automaton (NPDA) is a variation of the pushdown automaton. The class of non deterministic pushdown automata accepts the non deterministic

context-free languages, a proper subset of context-free languages. A deterministic pushdown automaton has one or more legal transition for the same combination of input symbol, state, and top stack symbol.

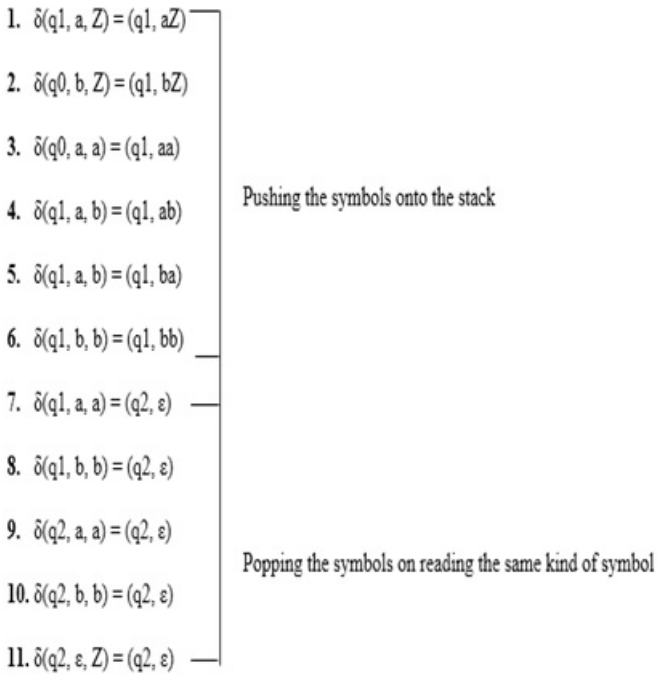


Figure 4.8: Non Deterministic pushdown automaton

Deterministic context-free languages:

Let L be a language defined over the alphabet $\sum$, the language L is deterministic context-free if and only if it is accepted by a deterministic pushdown automaton.

- Not all context-free languages are deterministic context-free.
- L1 = {wcw^R | w ∈{a, b}* } is deterministic context-free.
- L2 = {ww^R| w ∈{a, b}* } is context-free, but not deterministic context-free.

Chapter 5

TURING MACHINE

Course Outcomes

After successful completion of this module, students should be able to:

CO 6	Apply Turing machines and Linear bounded automata for recognizing the languages,complex problems.	Apply

5.1 TURING MACHINE MODEL

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition:

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple (Q, X, $\sum$,δ, q0, B, F) where −

- Q is a finite set of states

- X is the tape alphabet
- $\sum$ is the input alphabet
- $\delta$is a transition function; δ: Q $\varkappa$X $\rightarrow$ Q $\varkappa$X $\varkappa${Left shift, Right shift}.
- q0 is the initial state
- B is the blank symbol
- F is the set of final states
- A Turing machine then, or a computing machine as Turing called it, in Turing's original definition is a machine capable of a finite set of configurations q1,…,qn (the states of the machine, called m configurations by Turing). It is supplied with a one-way infinite and one-dimensional tape divided into squares each capable of carrying exactly one symbol. At any moment, the machine is scanning the content of one square r which is either blank (symbolized by S0) or contains a symbol S1,…,Sm with S1=0 and S2=1.The machine is an automatic machine (a-machine) which means that at any given moment, the behavior of the machine is completely determined by the current state and symbol (called the configuration) being scanned. This is the so-called determinacy condition . These a-machines are contrasted with the so-called choice machines for which the next state depends on the decision of an external device or operator . A Turing machine is capable of three types of action:

- Print Si, move one square to the left (L) and go to state qj
- Print Si, move one square to the right (R) and go to state qj
- Print Si, do not move (N) and go to state qj.

The 'program' of a Turing machine can then be written as a finite set of quintuples of the form: qiSjSi,jMi,jqi,j

Instantaneous description (ID) is a triple α1qα2

Suppose the following is the current ID of a TM

x1x2…xi-1qxixi+1…xn

Case 1)δ(q, xi) = (p, y, L)

(a) if i = 1 then qx1x2…xi-1xixi+1…xn $\vdash$pByx2…xi-1xixi+1…xn

(b) else x1x2…xi-1qxixi+1…xn $\vdash$x1x2…xi-2pxi-1yxi+1…xn

If any suffix of xi-1yxi+1…xn is blank then it is deleted.

Case 2) δ(q, xi) = (p, y, R)

x1x2…xi-1qxixi+1…xn $\vdash$x1x2…xi-1ypxi+1…xn

If i>n then the ID increases in length by 1 symbol

x1x2...xnq $\vdash$x1x2...xnyp

Definition: Let M = (Q, $\sum$, Γ, δ, q0, B, F) be a TM, and let w be a string in $\sum^*$. Then w is accepted by M iff

q0w $\vdash^*$ α1pα2

where p is in F and α1 and α2 are in Γ^*

Definition: Let M = (Q, $\sum$, Γ, δ, q0, B, F) be a TM. The language accepted by M, denoted L(M), is the set

{w | w is in $\sum^*$ and w is accepted by M}

Notes:

- In contrast to FA and PDAs, if a TM simply passes through a final state then the string is accepted.
- Given the above definition, no final state of a TM need to have any transitions. Henceforth, this is our assumption.
- If x is NOT in L(M) then M may enter an infinite loop, or halt in a non-final state.
- Some TMs halt on ALL inputs, while others may not. In either case the language defined by TM is still well defined.which is shiown in fig 5.1

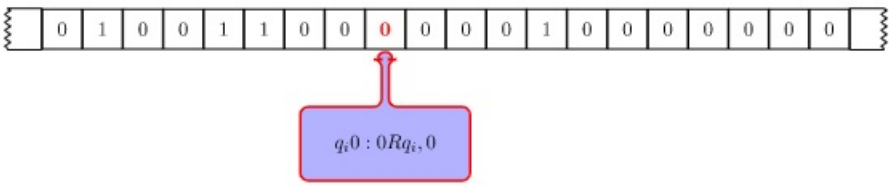


Figure 5.1: Turing Machine model

5.2 TURING MACHINE Examples

Example 1 :Construct a Turing Machine for language L= { 0^n1^n | n 1}

Logic: cross 0's with X's, scan right to look for corresponding 1, on finding it cross it with Y, and scan left to find next leftmost 0, keep iterating until no more 0's, then scan right looking for B.which is shiown in fig 5.2 and 5.3

The TM matches up 0's and 1's

- q1 is the "scan right" state, looking for 1
- q2 is the "scan left" state, looking for X
- q3 is "scan right", looking for B
- q4 is the final state

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

Figure 5.2: Turing Machine model for equal number of 0's , 1's

$q_0$0011BB.. |— X$q_1$011
|— X0$q_1$11
|— X$q_2$0Y1
|— q_2X0Y1
|— X$q_0$0Y1
|— XXq_1Y1
|— XXY$q_1$1
|— XXq_2YY
|— Xq_2XYY
|— XXq_0YY
|— XXYq_3YB...
|— XXYYq_3BB...
|— XXYYBq_4

Figure 5.3: Turing Machine model for equal number of 0's , 1's string computation

- q0 is on 0, replaces with the character to X, changes state to q1, moves right
- q1 sees next 0, ignores (both 0's and X's) and keeps moving right
- q1 hits a 1, replaces it with Y, state to q2, moves left
- q2 sees a Y or 0, ignores, continues left
- when q2 sees X, moves right, returns to q0 for looping step 1 through 5
- when finished, q0 sees Y (no more 0's), changes to pre-final state q3
- q3 scans over all Y's to ensure there is no extra 1 at the end (to crash on seeing any 0 or 1)
- when q3 sees B, all 0's matched 1's, done, changes to final state q4
- blank line for final state q4

Example 2 :Construct a Turing Machine for language L= { $0^n1^n2^n$ | n 1}

Prerequisite – Turing Machine

The language L = { $0^n1^n2^n$ | n 1} represents a kind of language where we use only 3 character, i.e., 0, 1 and 2. In the beginning language has some number of 0's followed by equal number of 1's and then followed by equal number of 2's. Any such string which falls in this category will be accepted by this language. The beginning and end of string is marked by $ sign.

Examples –

Input : 0 0 1 1 2 2 Output:Accepted

Input : 000 1 1 1 2 2 2 2 Output : Notaccepted

Assumption: We will replace 0 by X, 1 by Y and 2 by Z

Approach used –

First replace a 0 from front by X, then keep moving right till you find a 1 and replace this 1 by Y. Again, keep moving right till you find a 2, replace it by Z and move left. Now keep moving left till you find a X. When you find it, move a right, then follow the same procedure as above.A condition comes when you find a X immediately followed by a Y. At this point we keep moving right and keep on checking that all 1's and 2's have been converted to Y and Z. If not then string is not accepted. If we reach $ then string is accepted.

- Step-1: Replace 0 by X and move right, Go to state Q1.
- Step-2: Replace 0 by 0 and move right, Remain on same state Replace Y by Y and move right, Remain on same state Replace 1 by Y and move right, go to state Q2.
- Step-3: Replace 1 by 1 and move right, Remain on same state Replace Z by Z and move right, Remain on same state Replace 2 by Z and move right, go to state Q3.
- Step-4: Replace 1 by 1 and move left, Remain on same state Replace 0 by 0 and move left, Remain on same state Replace Z by Z and move left, Remain on same state Replace Y by Y and move left, Remain on same state Replace X by X and move right, go to state Q0.
- Step-5: If symbol is Y replace it by Y and move right and Go to state Q4 Else go to step 1
- Step-6: Replace Z by Z and move right, Remain on same state Replace Y by Y and move right, Remain on same state If symbol is $ replace it by $ and move left, STRING

IS ACCEPTED, GO TO FINAL STATE Q5 which is shiown in fig 5.4 and 5.5

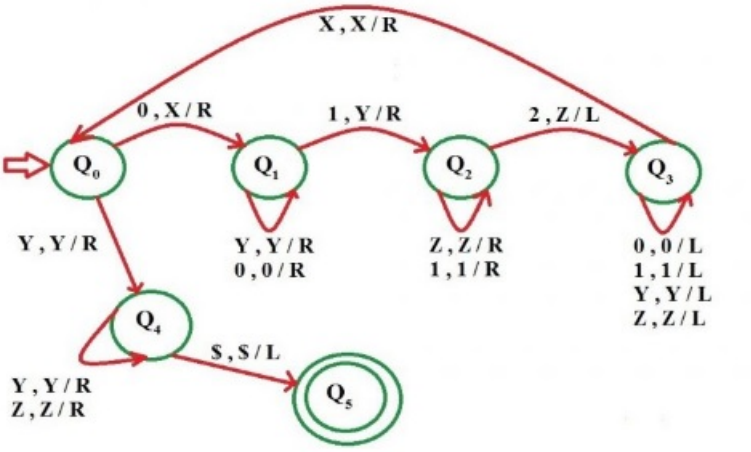


Figure 5.4: Turing Machine model for equal number of 0's , 1's , 2's

Example 3 :Construct a Turing Machine for language L= { $a^n b^n c^n$ | n 1}

	a	b	c	X	Y	Z	B
q0	q1,X,R				q4,Y,R		
q1	q1,a,R	q2,Y,R			q1,Y,R		
q2		q2,b,R	q3,Z,L			q2,Z,R	
q3	q3,a,L	q3,b,L		q0,X,R	q3,Y,L	q3,Z,L	
q4					q4,Y,R	q5,Z,R	
q5						q5,Z,R	qf,B,R
qf							

Figure 5.5: Turing Machine model for equal number of a's , b's , c's

5.3 The Chomsky Hierarchy

Type3 < Type 2 < Type 1 < Type 0

5.4 Types of Turing Machine

Types of Turing Machine : which is shiown in fig 5.6 and 5.7

1. Multiple track Turing Machine:

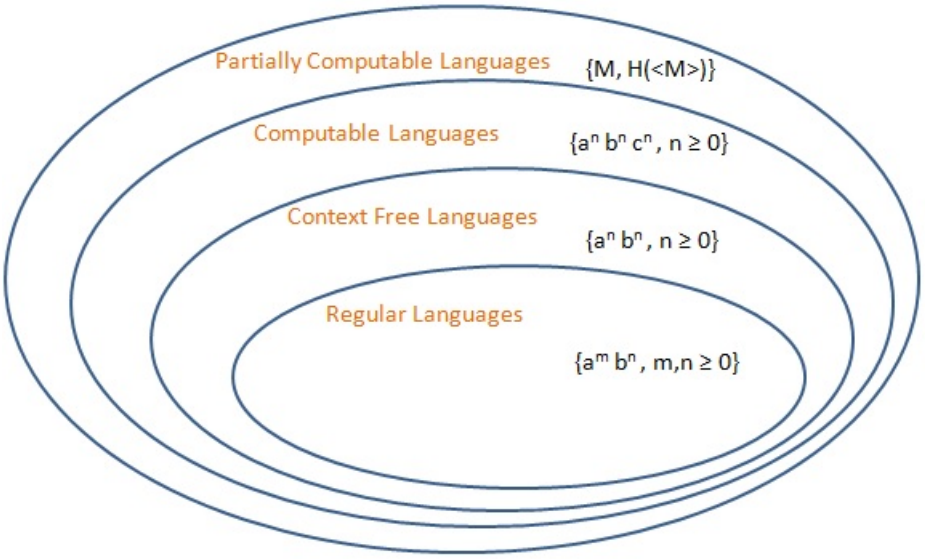


Figure 5.6: Chomsky Hierarch of Languages

Type	Language	Grammar	Automaton
0	Partially Computable	Unrestricted	DTM - NTM
1	Context Sensitive	Context Sensitive	Linearly Bounded Automaton
2	Context Free	Context Free	NPDA
3	Regular	right regular, left regular	DFA, NFA

Figure 5.7: Chomsky Hierarch of Languages

• A k-tack Turing machine(for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.

• A k-track Turing Machine can be simulated by a single track Turing machine

2. Two-way infinite Tape Turing Machine:

• Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.

• Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine(standard Turing machine).

3. Multi-tape Turing Machine:

• It has multiple tapes and controlled by a single head.

• The Multi-tape Turing machine is different from k-track Turing machine but expressive power is same.

- Multi-tape Turing machine can be simulated by single-tape Turing machine.

4. Multi-tape Multi-head TuringMachine:

- The multi-tape Turing machine has multiple tapes and multipleheads
- Each tape controlled by separatehead
- Multi-Tape Multi-head Turing machine can be simulated by standard Turing machine.

5. Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where head can move any direction that is left, right, up or down.
- Multi dimensional tape Turing machine can be simulated by one-dimensional Turing machine

6. Multi-head Turing Machine:

- A multi-head Turing machine contain two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by single head Turing machine.

7. Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one way infinite tape.
- For a given state and input symbol has atleast one choice to move (finite number of choices for the next move), each choice several choices of path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to deterministic Turing machine.

Post Correspondence Problem:

we will discuss the undecidability of string and not of Turing machines. The undecidability of the string is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

"The Post's correspondence problem consists of two lists of string that are of equal length over the input. The two lists are A = w1, w2, w3, ... , wn and B = x1, x2, x3, ... xn then there exists a non empty set of integers i1, i2, i3, ... , in such that, w1, w2, w3, ... wn = x1, x2, x3, ... xn".

To solve the post correspondence problem we try all the combinations of i1, i2, i3, ... , in to find the w1 = x1 then we say that PCP has a solution.
Find the solution for A = (b, bab^3,ba)$andB = (b^3$, ba, a). The input set is $\sum = \{0, 1\}$.

A solution is 2, 1, 1, 3. That means w2w1w1w3 = x2x1x1x3.The constructed string from both lists is bab3b3a.

Does PCP with two lists x = (b, a, aba, bb) and y = (ba, ba, ab, b) have a solution

Solution: Now we have to find out such a sequence that strings formed by x and y are identical. Such a sequence is 1, 2, 1, 3, 3, 4.
b a b aba aba bb = ba ba ba ab ab b

Bibliography

[1] John C Martin "Introduction to Languages and Automata Theory", . Tata McGraw-Hill, 3rd Edition, 2007.

[2] Daniel I.A. Cohen, "Introduction to Computer Theory", . John Wiley Sons, 2nd Edition, 2004.