CHAPTER - 1

Introduction

1.1 Mathematical Preliminaries

A. Sets and Subsets

A set is a well-defined collection of objects. For example: the set {red, yellow, blue) contains the names of primary colors. The individual objects are called member or elements of the set. We use the capital letters (A, B, C …) for denoting the sets and smaller letters (a, b, c …) to denote the elements of any sets. When 'a' is an element of the set 'A', we writes $a \in A$, otherwise we write $a \notin A$. The set can be described in following three ways:

- *By listing its elements:* All the elements of a set are written by enclosing them within braces. For example: $A = \{3, 6, 9, 12, 15\}$
- *By describing the properties of the elements of the set:* The property of elements is called the predicate. For example: $B = \{x: x \text{ is a multiple of } 3\}$ or $B = \{x \mid x \text{ is a multiple of } 3\}$
- *By recursion:* The computational rule is defined for calculating each elements of the set. For example: the set of all natural numbers leaving a reminder 1 when divided by 3 can be described as: $\{a_n \mid a_0 = 1, a_{n+1} = a_n + 3\}$

Let *A* and *B* be sets. We say that *B* is a **subset** of *A*, and write $B \subseteq A$, if every element of *B* is an element of *A*. Two sets A and B are equal (we write A = B) if their members are the same. In practice, to prove that A = B, we prove $A \subseteq B$ and $B \subseteq A$. A set with no element is called an empty set, also called a null set or a void set, and is denoted by ϕ. We define the some **operations** on set as follows:

- *Union of A and B:* $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- *Intersection of A and B:* $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
- *Complement of B in A:* $A - B = \{x \mid x \in A \text{ and } x \notin B\}$

B. Relations

The concept of relation is similar to the real life that compares one with the other. The relation made a pair of objects that is true in some cases and false in others; the statement '*x* is less than *y*' is true if $x = 3$ and $y = 4$, for example, but false if $x = 3$ and $y = 2$.

A relation R in a set S is a collection of ordered pair of elements in S. When (x, y) is in R, we write xRy. When (x, y) is not in R, we write xR'y. For example: the relation R in set Z can be defined by xRy if x>y.

Properties of relations

The relations have generally three properties: reflective, symmetric, and transitive. A relation R in a set S is called an equivalence relation if it is reflective, symmetric, and transitive.

- A relation R in a set S is *reflective* if xRx for every x in S.
- A relation R in a set S is *symmetric* if for x, y in S, yRx whenever xRy.
- A relation R in a set S is *transitive* if for x, y and z in S, xRz whenever xRy and yRz.

For example: Let us define a relation R in {1, 2, 3 …, 10} by aRb if a divides b. R is reflective and transitive but not symmetric (3R6 but 6R'3).

- Reflective relation: 1R1, 2R2, 3R3, 4R4, 5R5, 6R6, 7R7, 8R8, 9R9, and 10R10
- Transitive relation:
  - Let x = 1, y = 2, and z = 4 then 1R4 whenever 1R2 and 2R4
  - Let x = 2, y = 4, and z = 8 then 2R8 whenever 2R4 and 4R8
  - Let x = 1, y = 3, and z = 6 then 1R6 whenever 1R3 and 3R6

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

C. Functions

Let *X* and *Y* are two sets. A *function* or *map* '*f*' from *X* to *Y* is a rule that assigns to each element x of *X* exactly unique element of *Y*, which is denoted by f(x). The element f(x) is called the image of x under '*f*'. The function is denoted by $f : X \rightarrow Y$. The function can be defined in two ways:

- By giving the images (collection of values) of all elements X. For example: $f : \{1, 2, 3, 4\} \rightarrow \{a, b, c\}$can be denoted by $f(1) = a$, $f(2) = c$, $f(3) = a$, $f(4) = b$.
- By computational rule which computes f(x) once x is given. For example: $f : R \rightarrow R$ can be defined by $f(x) = x^2 + 2x + 1$ for every x in R. (R denotes all the real numbers.)

Types of functions

- *One – to – one (or injective) function:* The function $f : X \rightarrow Y$ is said to be one-to-one if different elements in X have different images i.e. $f(x_1) \neq f(x_2)$ when $x_1 \neq x_2$. For example: $f : Z \rightarrow Z$ given by $f(n) = 2n$ is one-to-one but not onto.
- *Onto (subjective) function:* The function $f : X \rightarrow Y$ is said to be onto if every element y in Y is the image of some element x in X i.e. one-to-many relations.
- *One – to – one correspondence (bijection):* The function $f : X \rightarrow Y$ is said to be one–to–one correspondence if *f* is both one-to-one and onto.

1.2 Alphabets and Languages

A. Alphabet

An alphabet is a non-empty set of *symbols* or *letters*, e.g. characters {a, b, . . . , z} or digits{0, 1} or ASCII characters etc. The alphabets are represented by the symbol Σ. Alphabets are important in the use of formal languages, automata and semi-automata. In most cases, for defining instances of automata, such as deterministic finite automata (DFAs), it is required to specify an alphabet from which the input strings for the automaton are built.

For example a common alphabet is {0, 1}, the *binary alphabet*. A finite string is a finite sequence of letters from an alphabet. For example, if we use the binary alphabet {0, 1}, the strings (ε, 0, 1, 00, 01, 10, 11, 000, etc.) would all be in the Kleene closure (i.e. set of alphabets) of the alphabet (where ε represents the empty string). An infinite sequence of letters may be constructed from elements of an alphabet as well.

B. Strings

A string (or word) is a finite sequence of symbols chosen from some alphabet. For example: 01101 and 111 are strings from the binary alphabet $\Sigma = \{0, 1\}$.

- The string with zero occurrences of symbols is called *empty string* (denoted by ε).
- The number of positions for symbols in the string is called *length of a string*. The length of string is noted as: w: |w|. For example: |011| = 3 and | ε | = 0

Generation of strings

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using the exponential notation: Σ^k: the set of strings of length k, each of whose is in Σ.

Examples:

- Σ^0: {ε}, regardless of what alphabet Σ is. That is ε is the only string of length 0

If $\Sigma = \{0, 1\}$, then:

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

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

- Σ^2 = {00, 01, 10, 11}
- Σ^3 = {000, 001, 010, 011, 100, 101, 110, 111}

Note: confusion between Σ and Σ^1
- Σ is an alphabet; its members 0 and 1 are symbols
- Σ^1 is a set of strings; its members are strings (each one of length 1)

Kleen star

Σ^*: The set of all strings over an alphabet Σ
- {0, 1}* = {ε, 0, 1, 00, 01, 10, 11, 000, . . .}
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$

The symbol $*$ is called **Kleene star** and is named after the mathematician and logician Stephen Cole Kleene.

Note:
- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots$
- Thus: $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

String concatenation

Strings can be concatenated yielding another string using the binary operation (.), called *concatenation* on Σ^*. If $a_1a_2a_3 \ldots a_n$ and $b_1b_2 \ldots b_m$ are in Σ^*, then $a_1a_2a_3 \ldots a_n . b_1b_2 \ldots b_m = a_1a_2a_3 \ldots a_n b_1b_2 \ldots b_m$.

If x are y be strings then x.y denotes the concatenation of x and y, that is, the string formed by making a copy of x and following it by a copy of y. For examples:
- x = 01101 and y = 110 then xy = 01101110 and yx = 11001101
- For any string w, the equations εw = wε = w hold.

C. Language

Languages define a set of alphabets and associated meanings. A *formal language L* over an alphabet Σ is a subset of Σ^*, that is, a set of words over that alphabet. Sometimes the sets of words are grouped into expressions, whereas rules and constraints may be formulated for the creation of *'well-formed expressions'*. For example, the expression 0(0 + 1)*1 represents the set of all strings that begin with a 0 and end with a 1. A formal language is often defined by means of a formal grammar such as a regular grammar or context-free grammar, also called its formation rule.

If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a (formal) **language** over Σ. A language (possibly infinite) set of strings all of which are chosen from some Σ^*.

Examples: The language may be Programming language C, English or French and others are:
- The language of all strings consisting of n 0s followed by n 1s ($n \geq 0$): {ε, 01, 0011, 000111, . . .}
- The set of strings of 0s and 1s with an equal number of each: {ε, 01, 10, 0011, 0101, 1001, . . .}
- Σ^* is a language for any alphabet Σ
- ∅, the empty language, is a language over any alphabet

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

- {ε}, the language consisting of only the empty string, is also a language over any alphabet
    - Note: $\emptyset \neq \{\varepsilon\}$ since $\emptyset$ has no strings and {ε} has one
- {w | w consists of an equal number of 0 and 1}
- $\{0^n1^n \mid n \geq 1\}$
- $\{0^i1^j \mid 0 \leq i \leq j\}$

Important operators on languages

- The **union** of two languages L and M, denoted L∪M, is the set of strings that are in either L, or M, or both. For example: If L = {001, 10, 111} and M = {ε, 001} then L∪M = {ε, 001, 10, 111}
- The **concatenation** of languages L and M, denoted L.M or just LM , is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. For example: If L = {001, 10, 111} and M = {ε, 001} then L.M = {001, 10, 111, 001001, 10001, 111001}
- The **closure** of a language L is denoted L* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (*i.e.*, the same string may be selected more than once) and concatenating all of them. For examples:
    - If L = {0, 1} then L* is all strings of 0 and 1
    - If L = {0, 11} then L* consists of strings of 0 and 1 such that the 1 come in pairs, *e.g.*, 011, 11110 and ε. But not 01011 or 101.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

CHAPTER - 2

Regular Expression and Finite Automata

2.1 Chomsky Hierarchy of Grammar

- *Language:* A language is a collection of sentences of finite length, all constructed from a finite alphabet of symbols. Thus, the string or sentences is a sequential occurrence of input alphabets or symbols such as 0011.
- *Grammar:* A grammar can be regarded as a device that enumerates the sentences of a language.
- *Formal Grammar:* Formal grammar is a way to describe a formal language i.e. set of finite length strings over a certain finite alphabet. They are named "formal grammar" by the analogy with the concept of grammar of human language.

It is the containment hierarchy of class of formal grammar which was describe by Noam Chomsky in 1956.Here, the automation of the respective grammar is the mathematical model to operate the language of grammar.

Class	***Grammars***	***Languages***	***Automaton***
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown
Type-3	Regular	Regular	Finite

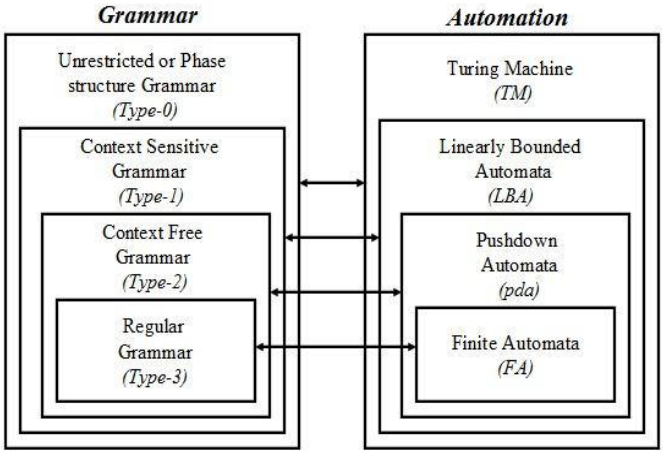


Fig. The Chomsky hierarchy of Grammar

2.2 Regular Expression

The regular expressions are useful for representing certain set of strings in an algebraic fashion. The regular expressions are generated by the ***regular grammar*** as they are the ***language generator***. Actually, the regular expression describes the language accepted by the finite state automata. Any set represented by a regular expression is called a *regular set*. We give a formal definition of regular expression over Σ as follows:

1. Any terminal symbol of the input alphabet such as 0, 1, a, b…$\in$ or $\wedge$ and ϕ are regular expression.
2. The union of two regular expression R1 & R2, written as R1+R2 or R1$\cup$R2 is also RE.
3. The concatenation of two regular expression R1 & R2, written as R1.R2 is also RE.
4. The iteration of any regular expression R, written as R* is also regular expression.
5. If R is a regular expression then (R) is also RE. This have different mean for the case of iteration i.e. $1+0^* \neq (1+0)^*$

Identities for Regular Expression

There are some identities for the regular expression that are helpful for simplifying the regular expressions.

- $\Phi + R = R$
- $\Phi R = R \phi = \phi$
- $\wedge R = R \wedge = R$
- $\wedge^* = \wedge$ and $\phi^* = \wedge$
- $R + R = R$

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

- R*R* = R*
- RR* = R*
- (R*)* = R*
- $\wedge$ + RR* = R* = $\wedge$ + R*R
- (PQ)*P = P(QP)*
- (P + Q)* = (P*Q*)* = (P* + Q*)*
- (P + Q)R = PR + QR and R(P+Q) = RP + RQ

Equivalence of two regular expressions

Let P and Q be two regular expressions then P & Q are equivalent (we write P = Q) iff they represent the same set. Also, P and Q are equivalent iff the corresponding finite automata are equivalent or isomorphic.

Operators of Regular Expression

The regular expression denotes the language. For example, the RE 01* + 10* denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by number of 0's. There are three operations on languages that represent the operators of the regular expression. These operations are:

1. The **union** of two languages L & M denoted by L+M, is the set of string that are in either L or M or both. For example, let L = {001, 10, 111} and M = {$\in$, 001} then L + M = {$\in$, 10, 001, 111}.
2. The **concatenation** of language L & M is the set of string that can be formed by taking any string in L and concatenating it with any string in M and is denoted by L.M. For example, let L = {001, 10, 111} and M = {$\in$, 001} then L.M = {001, 10, 111, 001001, 10001, 111001}.
3. The **closure** (or star or Kleene closure) of language L is denoted by L* and represent the set of those string that can be formed by taking any number of string form L, possibly with repetitions (i.e. the same string may be selected more than once) and concatenating all them. For example, let L = {0,1} then L* is the set of all the string of 0's and 1's with $\in$ or $\wedge$, i.e. L* = {$\wedge$, 0, 1, 01, 10, 11, 000, … }.

Example

Let *L1 = the set of all string of 0's and 1's ending in 00*, and then the equivalent regular expression can be described as:

- Any string in L1 is obtained by concatenating any string over {0, 1} and the string 00. Since, {0, 1} is represented by 0 + 1. Hence L1 is represented by (0 + 1)*00.

L2 = *the set of all string of 0's and 1's beginning with 0 and ending with 1*, and then the equivalent regular expression can be described as:

- Any string in L2 is obtained by concatenating 0, any string over {0, 1} and 1. Thus L2 can be represented by 0(0+1)*1.

L3 = {$\wedge$, 11, 1111, 111111 …}, and then the equivalent regular expression can be described as:

- Any element of L3 is either $\wedge$, or a string of even number of 1's so L3 can be represented by (11)*.

Notes:

- $\Sigma^* = \Sigma^+ + \in$
- (0,1)* = (0+1)* but (0+1) $\neq$ (0,1)
- 0+1 = Either 0 or 1 but not both. Thus, 0* + 1* = Either string of zeros or string of one.
- 01 = First consume 0 then 1. Thus, 0*1* = First string of zeros then the string of one.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Decision properties of regular language

1. Is the language describe is empty?
2. Is the particular string 'w' is in the described language 'L'?
3. Is two described language are equivalent or describe the same language?

A. Testing of emptiness of Regular Language

If there is any path from the start state to some accepting state, then the language is non-empty, while if the accepting states are all separated from the start state, then the language is empty. Thus, if we can compute the set or any one of reachable states or accepting state from the start state by reading the any input string then we answer "NO" (i.e. the language is non-empty) and otherwise we answer "YES" (i.e. the language is empty).

B. Testing membership in a Regular Language

The next important question is that if 'w' is a given string represented explicitly and 'L' be a regular language represented by an automata or regular expression then "Is 'w' is in 'L' or not?" For this, we have to construct a DFA and then simulate the DFA processing the string of input symbol 'w', beginning in the start state. If the DFA end in the accepting state, the answer is "YES", otherwise the answer is "NO".

C. Testing of equivalence of two Regular Languages

Another important question is "whether the different description of two languages indicate the same things or not?" Since, the different description of a language is given by a tool called regular expression. To check the equivalence of regular languages, we construct the DFA of different description. If we get the same DFA for different description, the answer is "YES", otherwise the answer is "NO".

2.3 Automata Theory

- Automata theory: the study of abstract computing devices, or "machines"
- Before computers (1930), **Alan Turing** studied an abstract machine (*Turing machine*) that had all the capabilities of today's computers (concerning what they could compute). His goal was to describe precisely the boundary between what a computing machines could do and what it could not do.
- Simpler kinds of machines (*finite automata*) were studied by a number of researchers and useful for a variety of purposes. Theoretical developments bear directly on what computer scientists do today
  - Finite automata, formal grammars: design/ construction of software
  - Turing machines: help us understand what we can expect from a software
  - Theory of intractable problems: are we likely to be able to write a program to solve a given problem? Or we should try an approximation, a heuristic...

Why Study Automata Theory?

Finite automata are a useful model for many important kinds of software and hardware:

1. Software for designing and checking the behavior of digital circuits
2. The lexical analyzer of a typical compiler, that is, the compiler component that breaks the input text into logical units
3. Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases or other patterns
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols of protocols for secure exchange information

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Finite state Machine (FSM)

A finite state machine or simply a state machine is a particular conceptualization of a sequential circuit composed of inputs & outputs, a finite number of states, transition between those states and actions. Thus, a computer can be viewed as huge FSMs. In other word, a finite state machine defined as a system of automation where energy, material and information are transformed and use for performing some functions without direct participation of man. Examples are: automatic machines, and automatic photo printing machine. In computer science the term 'automation' means 'discrete automation' and is defined in more abstract way as shown in figure.

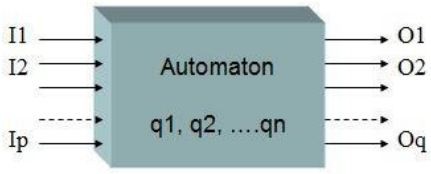
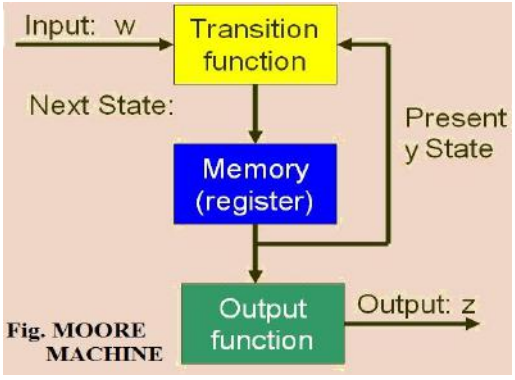


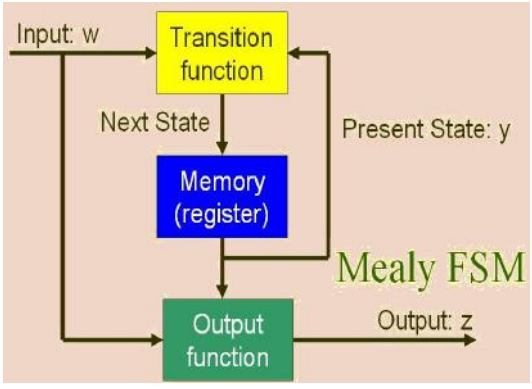
Fig. Model of Discrete Automation

Formally, the finite state machine M = (Q, Σ, O, δ, q_0 g) is a 6-tuple that consists of :

- Q = Finite non-empty set of states.
- Σ = Finite non-empty set of input alphabets.
- O = Finite non-empty set of output alphabets.
- δ = State relation or transition function.
- q_0 = Start state, $q_0 \in Q$.
- g = Output function for all pair of states.

* An automation in which the output depends only on the input is called *automation without a memory*.
* An automation, in which the output depends on input and the state as well, is called *automation with finite memory.*
* An automation in which the output depends only on state of the machine is called *Moore Machine*.
* An automation in which the output depends on the state as well as on the input at any instant of time is called a *Mealy Machine*.





Finite State Automata (FSA)

Finite state automata (singular Automation) are a special kind of finite state machine with no outputs but have a set of final states. They recognize string that take the starting state to final state but no storage mechanism so sometime called the automation without memory. The finite state automata can be used as ***language recognizers***. This application plays a fundamental role in the design and *construction of compilers for programming languages.*

The sets of strings recognized by finite automata are called regular expression. The sets built up from the null set, the empty string, and singleton (a set by single digit) strings by concatenations, unions and Kleene closures, in arbitrary order is called regular sets.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Analytically, a finite automaton *M = (Q, Σ, δ, q0, F)* is quin or 5-tuple which consists of

Q = Finite non-empty set of states

Σ = Finite non-empty set of input alphabet

δ = Transition function that assign a next states and inputs (since, Q x Σ → δ)

q0 = Initial states or start states (q0 ∈ Q)

F = Subset of Q (i.e. F ⊆ Q) consisting of final state (or accepting states). It is assumed that there may more than one final state.

The various component of the FSA or FA are

- **Input tape:** The input tape is divided into squares, each square containing a single symbol from the input alphabet Σ. The absent of the end mark on the input string indicate the infinite length of string.
- **Reading Head:** The reading head examines only one square at a time and can move one square left or right (but generally from left to right).
- **Finite Control:** Here, all the states of the system on reading the input string remain on this set so it is called the finite control. It controls the activities on the system as following manner:
  - Motion of the reading head along the tape to the next square.
  - The next state of the finite automata after the processing of δ (q, a), where q = state of machine, a = input symbol & δ = transition function.

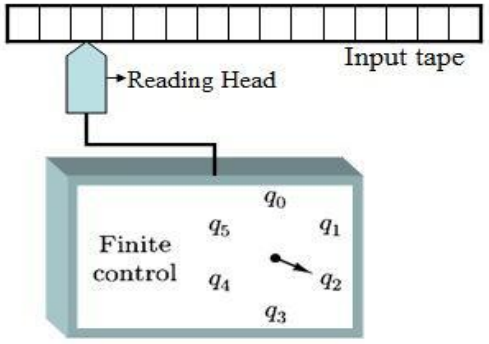


Fig. B.D. of a Finite automata

→ **State:** A uniquely identifiable set of values measured at various points in a digital system.

→ **Next State**

- The state to which the state machine makes the next transition, determined by the inputs present when the device is clocked.
- At the termination of the input string the state of the FA must reach to any one final state among the various possible final states.

Notation for FA

There are two preferred notation for describing the automata, which are:

A. State Diagram or Transition Diagram or graph

A transition diagram or transition system is a finite directed weight graph in which each vertex (or node) represents a state and the directed edge indicate the transition of the state and the edges are labelled with input. The *initial state* is represented by a *circle or bubble with an arrow pointed towards it,* the *final state* by *two concentric* circles and *other states* by just a *circle.*

The state diagram accept the string ***w*** in Σ* if there exists a path which originates from some initial states, goes along the arrows and terminate at some final state with path value ***w***.

Example:

Let we have the transition function as: (q0, 0) → q0, (q0, 1) → q1, (q1, 0) → q0, (q1, 1) → q1, with the initial state q0 and the final state q1, then the corresponding state diagram is given by:

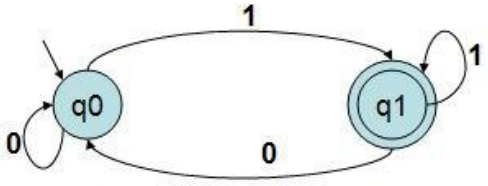


Fig. Transition Diagram

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

B. State Table or Transition Table

A transition table is a conventional tabular representation of a function like δ that take *two arguments and returns a value*. The row of the table corresponds to the status and the column corresponds to the inputs. The entry for the row corresponding to state 'q' and the column correspond to input 'a' is the state δ(q, a).

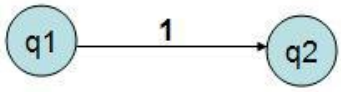
Q/Σ	0	1
→q0	q0	q0
q1	q0	q0

Fig. Transition Table

Types of Finite Automata

A. Deterministic Finite Automata (DFA)

DFA is a FA that is in a single state after reading any sequence of inputs. The term 'deterministic' refer to the fact that on each input, there is only one state to which the automata can transition from its current states.



A deterministic finite automaton *M = (Q, Σ, δ, q0, F)* is quin or 5-tuple which consists of
Q = Finite non-empty set of states
Σ = Finite non-empty set of input alphabet
δ = Transition function (usually called the *direct transition function*) that assign a next states and inputs (since, Q x Σ → δ). In the transition diagram representation of DFA, δ is represented by arcs between states and the label on the arc. If 'q' is a state, and 'a' is an input symbol then δ(q, a) is represent that state 'p' such that there is an arc labelled 'a' from 'q' to 'p'.

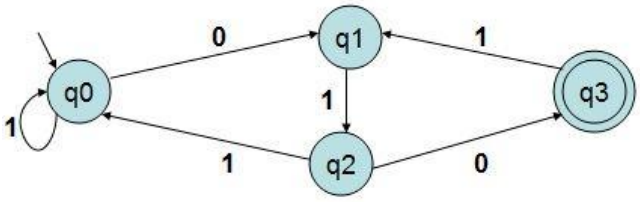


Fig. Transition Diagram for DFA

q0 = Initial states or start states (q0 ∈ Q)
F = Subset of Q (i.e. F ⊆ Q) consisting of final state (or accepting states). It is assumed that there may more than one final state.

Language of DFA

The language of DFA, A = (Q, Σ, δ, q0, F) is defined by L(A) = { W: δ(q0, W) is in F} i.e. the language of 'A' is the set of string 'W' that takes the start state q0 to one of the accepting state. If language 'L' is L(A) for some DFA 'A', then we say 'L' is a regular language.

Acceptability of a string by a DFA

A string 'x' is accepted by a DFA, A = (Q, Σ, δ, q0, F) if δ(q0, x) = q for some q∈F.

Q/Σ	0	1
→q0	q2	q1
q1	q3	q0
q2	q0	q3
q3	q1	q2

Fig. Transition Table

$$\delta(q0, 110101) = \delta(q1, 10101)$$
$$= \delta(q0, 0101)$$
$$= \delta(q2, 101)$$
$$= \delta(q3, 01)$$
$$= \delta(q1, 1)$$
$$= \delta(q0, ^)$$
$$= q0$$

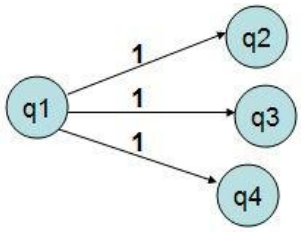
Hence

$$q0 \xrightarrow{1} q0 \xrightarrow{1} q0 \xrightarrow{0} q0 \xrightarrow{1} q0 \xrightarrow{0} q0 \xrightarrow{1} q0$$

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

B. Non-deterministic Finite Automata (NFA)

Non-deterministic means a choice of moves for an automata i.e. rather than prescribing a unique move in each situation of input, we allow a set of possible moves. As shown in figure the state from q1 can change to the possible states q2, q3 & q4 on reading the same input 1. As there are multiple state for the same input so the system sometime may not reach to the final state, hence termed as the non-deterministic automata.



A non-deterministic finite automaton $M = (Q, \Sigma, \delta, q0, F)$ is quin or 5-tuple which consists of
Q = Finite non-empty set of states
Σ = Finite non-empty set of input alphabet
q0 = Initial states or start states (q0 ∈ Q)
F = Subset of Q (i.e. F ⊆ Q) consisting of final state (or accepting states). It is assumed that there may more than one final state.
δ = Transition function that assign a next states and inputs (since, Q x Σ → δ).

Notice that the only difference between NFA & DFA is that the δ returns a set of value in case of NFA and a single value in case of DFA.

Conversation of NFA to DFA

- For every NFA, there exists a DFA which simulate the behavior of NDFA. Alternatively, if L is the accepted by NDFA, then there exists a DFA which also accept L.
- Every state of a DFA will be represented by some subset of set of states of NFA and therefore the transition of NFA to DFA is normally called a ***subset construction***. If there are 'n' states of NFA then the possible subsets are 2^n, in which some states are used for construction of the equivalent DFA. Thus, for two states q0 & q1 for NFA then its possible subsets are $2^2 = 4$ [i.e. {ϕ},{q0}, {q1}, {q0q1}]
- For converting the NFA to DFA,
  - Seek all the transition from start state for every symbol in set of inputs 'Σ'. If you get the different state beside the start state then terms them as a new state.
  - For all the new state, check all the transition for every symbol in Σ. Repeat step-2 till we are getting a new state.
  - All these state which consists at least one accepting or final state of given NFA will be considered as the final state for the equivalent DFA.

EXERCISE – I

NFA with ∈ Transition

If a FA is modified to permit transition without input symbol, along with zero, one or more transition on input symbols, then we get a NFA with ∈- transitions because the transition mode without symbol are called ∈-transitions. The idea of the ∈-NFA come from the concept of closure or iteration.

Formally we represent an ∈-NFA by A = $(Q, \Sigma, \delta, q0, F)$, where all components have their same interpretation as for NFA except δ is now a function that takes as arguments: -

i. All the states lie in Q after transition.
ii. A member of $\Sigma \cup \{\in\}$ i.e. either an input symbol or the symbol for empty string ∈ and it cannot be the member of the alphabet Σ only.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

The figure shows the NFA with ∈- transition because it is possible to make the transition from state q0 to q1 and q1 to q3 without consuming any of the input symbols. This state diagram accept the string 010 as it can be written as 0∈1∈0.

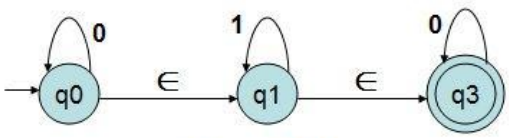
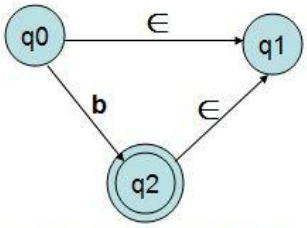


Fig. ∈- NFA

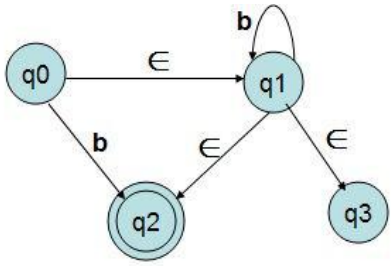
∈-Closure

A string 'w' in Σ will be accepted by ∈-NFA if there exist at least one path corresponding 'w', which start in an initial state and ends in one of the final states. Since this path may be formed by ∈- transitions as well as non-∈- transitions. We may require defining a function ∈- Closure (q), where q is a state of the automata.

The function ∈- Closure (q) is defined as "the set of all those stats of the automata (i.e. ∈-NFA) which can be reached from q on a path labeled by ∈ i.e. without consuming any input symbols. Note that if there is no ∈ for any state to transit then the ∈- Closure (q) function for that state will be the same state.



∈- Closure (q0) = {q0, q1}
∈- Closure (q1) = { q1}
∈- Closure (q2) = { q2, q1}



∈- Closure (q0) = {q0, q1, q2, q3}
∈- Closure (q1) = { q2, q3}
∈- Closure (q2) = { q2}
∈- Closure (q3) = { q3}

Conversion of ∈-NFA to DFA

This conversion process follows two steps:

- Find the ∈- Closure (q) function of each states
- Identify the transition function of each state obtained from ∈- Closure (q) function.

EXERCISE – II

Conversation of Regular Expression (RE) into equivalent Deterministic Finite Automata (DFA)

Theorem: *Every regular expression 'R' can be recognized by a transition system. In other word, for every string 'w' in the set of regular expression 'R', there exists a path from the initial state to a final state with path value 'w'.*

Since, any string generated by regular expression is accepted by finite state automata. Thus, on reading the string from start to finish, it is accepted by the FSA in regular order of path.

It should be noted that in the case of iteration, we can add new state by performing the null or ∧ move and

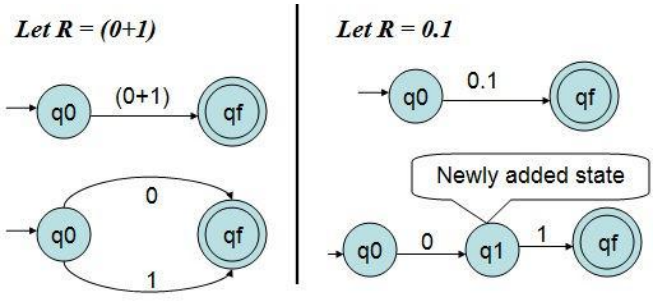


Fig. Conversation of RE in to equivalent FA

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

generally we merge the ∧ - move in final step of conversation process. For the RE to FA conversation, the order of precedence is *, (, +, & . and the associatively moves from left to right.

Note: The representation of a given regular expression may have multiple ***isomorphic graphs*** in conversion of equivalent deterministic finite automata.

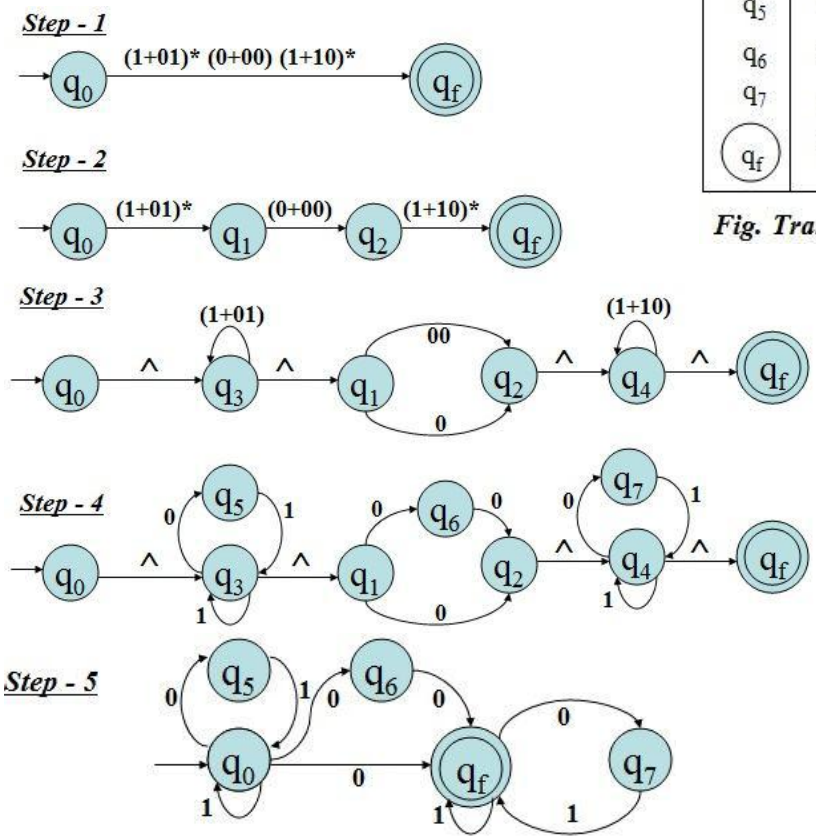
Example

The given regular expression (R) = (1+01)* (0+00) (1+10)*

Now, the equivalent Finite Automata (FA) can be constructed as:

Q/Σ	0	1
→ q_0	q_5, q_6,q_f	q_0
q_5	ϕ	q_0
q_6	q_f	ϕ
q_7	ϕ	q_f
(q_f)	q_7	q_f

Fig. Transition Table



Since, the resulting Finite Automata is NFA, which can be converted in to the DFA as follow. Let δ' be the new transition function for the equivalent DFA.

Thus,

δ' (q_0, 0) = δ (q_0, 0) = {q_5, q_6, q_f} *(New state)* → q_r (Say)

δ' (q_0, 1) = δ (q_0, 1) = {q_0}

δ' [(q_5, q_6, q_f), 0] = δ (q_5, 0) ∪ δ (q_6, 0) ∪δ (q_f, 0) = {ϕ} ∪ {q_f} ∪ {q_7} = {q_7, q_f} *(New state)* → q_s (Say)

δ' [(q_5, q_6, q_f), 1] = δ (q_5, 1) ∪ δ (q_6, 1) ∪δ (q_f, 1) = {q_0} ∪ {ϕ} ∪{q_f} = {q_0, q_f} *(New state)* → q_t (Say)

δ' [(q_7, q_f), 0] = δ (q_7, 0) ∪ δ (q_f, 0) = {ϕ} ∪{q_7} = {q_7} *(New state)*

δ' [(q_7, q_f), 1] = δ (q_7, 1) ∪ δ (q_f, 1) = {q_f} ∪{q_f} = {q_f} *(New state)*

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

$\delta'[(q_0, q_f), 0] = \delta(q_0, 0) \cup \delta(q_f, 0) = \{q_5, q_6, q_f\} \cup \{q_7\} = \{q_5, q_6, q_7, q_f\}$ *(New state)* $\rightarrow q_u$ (Say)

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

$\delta'(q_7, 0) = \delta(q_7, 0) = \{\phi\}$
$\delta'(q_7, 1) = \delta(q_7, 1) = \{q_f\}$

$\delta'(q_f, 0) = \delta(q_f, 0) = \{q_7\}$
$\delta'(q_f, 1) = \delta(q_f, 1) = \{q_f\}$

$\delta'[(q_5, q_6, q_7, q_f), 0] = \delta(q_5, 0) \cup \delta(q_6, 0) \cup \delta(q_7, 0) \cup \delta(q_f, 0) = \{\phi\} \cup \{q_f\} \cup \{\phi\} \cup \{q_7\} = \{q_7, q_f\}$

$\delta'[(q_5, q_6, q_7, q_f), 1] = \delta(q_5, 1) \cup \delta(q_6, 1) \cup \delta(q_7, 1) \cup \delta(q_f, 1) = \{q_0\} \cup \{\phi\} \cup \{q_f\} \cup \{q_f\} = \{q_0, q_f\}$

Thus, the new transition δ' of the DFA can be represented by the transition table as:

Q/Σ	0	1
→q_0	q_r	q_0
q_r	q_s	q_t
q_s	q_7	q_f
q_t	q_u	q_t
q_7	ϕ	q_f
q_f	q_7	q_f
q_u	q_s	q_t

Now,

i. Acceptance of string 0011100

$\delta(q_0, 0011100) = \delta(q_r, 011100)$
$= \delta(q_s, 11100)$
$= \delta(q_f, 1100)$
$= \delta(q_f, 100)$
$= \delta(q_f, 00)$
$= \delta(q_7, 0)$
$= \delta(\phi, \wedge)$

This implies that the string is not acceptable by the Finite Automata.

ii. Acceptance of string 10111001101

$\delta(q_0, 10111001101) = \delta(q_0, 0111001101)$
$= \delta(q_r, 111001101)$
$= \delta(q_t, 11001101)$
$= \delta(q_t, 1001101)$
$= \delta(q_t, 001101)$
$= \delta(q_u, 01101)$
$= \delta(q_s, 1101)$
$= \delta(q_f, 101)$
$= \delta(q_f, 01)$
$= \delta(q_7, 1)$
$= \delta(q_f, \wedge)$
$= \{q_f\}$

This implies that the string is acceptable by the Finite Automata.

EXERCISE – III

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Conversation of Deterministic Finite Automata (DFA) into equivalent Regular Expression (RE)

Arden's Theorem: Let **P** & **Q** be two regular expressions over Σ. If **P** does not contain $\wedge$, then the following equation in **R**, namely **R = Q + RP** has a unique solution (i.e. one and only one solution) given by **R = QP*.**

Steps for DFA to RE conversion

1. Define the equation for each state. The equation of start state must contain the $\wedge$ also.
2. Attempt to find the expression contain only the alphabet (as RE contain the alphabets only) for the final state. If there is more than one final state then identify the expression of alphabets only for all final state and finally merge them into one expression.

Example

For the given Transition table, the Finite Automata M = [{q_0, q_1, q_2, q_3}, {0, 1}, δ, q_0, {q_3}] where δ can be defined as:

Q/Σ	0	1
→q0	q1	q0
q1	q2	q0
q2	q3	q0
(q3)	q3	q3

Fig. Transition Table

Now, FA can be constructed as:

Now, the equivalent Regular Expression (R.E.) of the Finite Automata (F.A.) can be constructed as:

Step1: Defining the equation for each state with reference to the input symbol (Only incoming arrows)

Here,

$q_0 = q_0.1 + q_1.1 + q_2.1 + \wedge$ ---- I

$q_1 = q_0.0$ ---- II

$q_2 = q_1.0$ ---- III

$q_3 = q_2.0 + q_3.0 + q_3.1$ ---- IV

Step2: Finding the expression of symbol only for Final state

Here, From equation IV, we have

$q_3 = q_2.0 + q_3.0 + q_3.1$

$q_3 = q_2.0 + q_3 (0 + 1)$

Using the Arden's theorem, we have

$q_3 = q_2.0 (0 + 1)^*$ ---- V

Now, using equation II & III on the equation V, we have

$q_3 = q_0.000 (0 + 1)^*$ ---- VI

Now, from equation I, II, & III, we have

$q_0 = q_0.1 + q_0.01 + q_0.001 + \wedge$

$q_0 = q_0 (1 + 01 + 001) + \wedge$

Using the Arden's theorem, we have

$q_0 = \wedge (1 + 01 + 001)^* = (1 + 01 + 001)^*$ ---- VII

Now, from equation VI, & VII, we have

$q_3 = (1 + 01 + 001)^* 000 (0 + 1)^*$

This is the R.E. which is accepted by FA.

EXERCISE – IV

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Closure Property of Regular set

The closure property of the regular set is defined with reference to the Finite Automata (FA) i.e. the language acceptor. Class of language accepted by finite automata is closed under: Union, concatenation, Kleene closure or Star, omplementation and Intersection.

A. Union

Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ be NFA that accept a regular language $L = L(M_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ be another NFA that accept a regular language $L = L(M_2)$. Here, we assume that Q1 and Q2 are two disjoint sets.

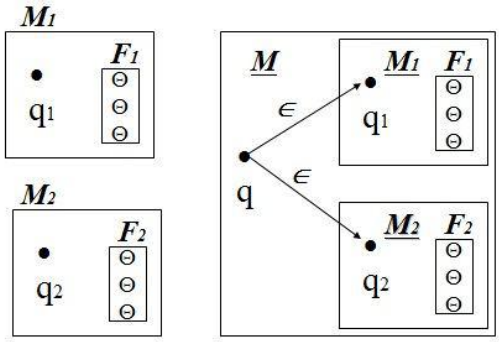


Fig. Regular Language is closed under union

Now, we construct NFA $M = (Q, \Sigma, \delta, q, F)$, such that it can accepts $L = L(M1) \cup L(M2)$, where q is the new state not in Q1 or Q2. Where,

- $Q = Q1 \cup Q2 \cup \{q\}$
- Σ = Set of input states = $\Sigma1 \cup \Sigma2 \cup \{\wedge\}$
- $\delta = \delta1 \cup \delta2 \cup [(q, \wedge) \rightarrow q1, (q, \wedge) \rightarrow q2]$
- q = Start state.
- $F = F1 \cup F2$.

M begins any computation by non-deterministically choosing to either the initial state of M1 or the initial state of M2 and interpreter, M initiates either M1 or M2. Formally, if $w \in \Sigma^*$, then $(q, w) \xrightarrow{*}_{M} (P, \in)$, for some $P \in F$ if and only if $(q, w) \xrightarrow{*}_{M_1} (P, \in)$, for some $P \in F1$, or $(q, w) \xrightarrow{*}_{M_2} (P, \in)$, for some $P \in F2$

Hence, $L(M) = L(M1) \cup L(M2)$.

B. Concatenation

Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ be NFA that accept a regular language $L = L(M_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ be another NFA that accept a regular language $L = L(M_2)$. Now, we construct NFA $M = (Q, \Sigma, \delta, q, F)$, such that it can accepts $L = L(M) = L(M1) \bullet L(M2)$.

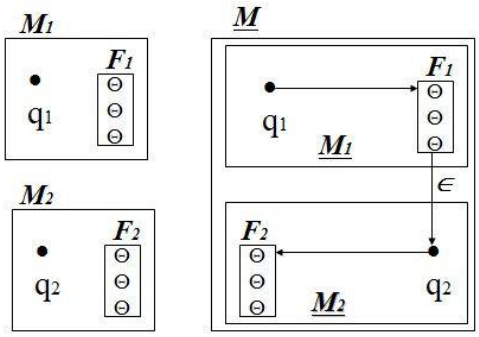


Fig. RL is closed under Concatenation

Where,

- $Q = Q1 \cup Q2$
- Σ = Set of input states = $\Sigma1 \cup \Sigma2 \cup \{\wedge\}$
- $\delta = \delta1 \cup \delta2 \cup [(F1, \wedge) \rightarrow q2]$
- q = Start state = q1
- F = F2.

Formally, $(q, w) \xrightarrow{*}_{M} (P, \in)$, for some $P \in F$ if and only if there are w1, w2 $\in \Sigma^*$ and there is a $P1 \in F1$, such that w = w1w2 and $(q2, w2) \xrightarrow{*}_{M_2} (P2, \in)$ & $(q1, w1) \xrightarrow{*}_{M_1} (P1, \in)$

Hence, $L(M) = L(M1) \bullet L(M2)$.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

C. Kleene closure

Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ be NFA that accept a regular language L = L (M_1). Now, we construct NFA M = *(Q, Σ, δ, q, F)*, such that it can accepts L = L (M) = L (M1)*.

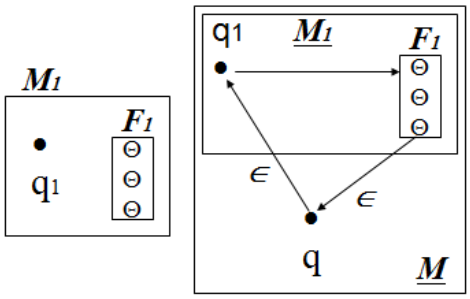


Fig. RL is closed under Kleene Closure

Where,

- Q = Q1 ∪ {q}
- Σ = Set of input states = Σ1 ∪ {∧}
- δ = δ1 ∪ [(q, ∧) → q1, (q1, ∧) → F_1, (F_1, ∧) → q]
- q = Start state
- F = F1∪ {q}.

Here, M consists of the states of M1 and all the transition of M1 and also, any final state of M1 is the final state of M. In addition, M has a new initial state 'q'. The new initial state is also final so that ∈ is accepted. From it, the state q1 can be reaches on the input ∈, so that the operations of M1 can be initiated after M has been started in state 'q'. Finally, the transition on input are added from each final state of M1 back to q1, once a string in L (M1) has been read, computation can resume from the initial state of M1.

It follows the inspection of M that if w ∈ L (M) then either w = ∧ or w = $w_1, w_2 \ldots w_k$, for k ≥ i. for i = 1, 2 …k, there is $F_i \in F$ such that $(q1, w) \xrightarrow[M_1]{*} (F1, \in)$

Hence, w ∈ L (M1)* so L (M) = L (M1)*.

Pumping Lemma for Regular Set or Language

- The pumping lemma for regular languages describes an essential property of all regular languages. Informally, it says that all sufficiently long words in a regular language may be pumped - that is, have a middle section of the word repeated an arbitrary number of times - to produce a new word which also lies within the same language.

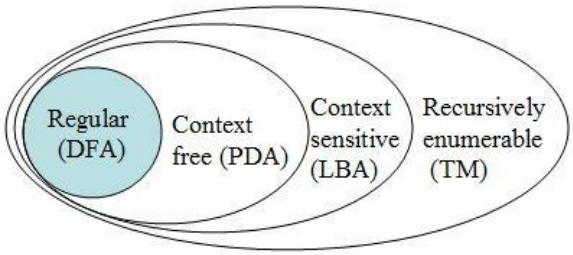


Fig. The Chomsky hierarchy of Grammar

- The pumping lemma gives a method of pumping (generating) many input string from a given string. In general, it gives a necessary condition for an input string to belong to a regular set.
- We have established that the class of language known as regular language for at least four different descriptions. They are the language accepted by DFA, NFA, ∈-NFA, and regular expression. Not every language is regular. The pumping lemma is a powerful technique for showing certain language is not-regular.
- Let Σ = {0, 1} = {0, 1, 00, 11, 01, 001…} be the set of string generated from the given input symbol. These set of the string may fall in different class of language as shown in figure. Thus, the pumping lemma helps us to identify whether the set of string or language fall in the class of regular language or not. We can use proof by contradiction together with the Pumping Lemma to demonstrate a language is not regular.
- Pumping lemma is the relation between the set of string (n) and the length of the string (m or |w|).

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

- Neither the original nor the general version of the pumping lemma gives a sufficient condition for a language to be regular. That is, the lemma holds for some non-regular languages. If the lemma does not hold for a language L, then L is not regular. If the lemma does hold for a language L, L might be regular.

Statement

Let *M = (Q, Σ, δ, q0, F)* be a finite automaton with n states that accepts a regular language L. Let w be any string such that w∈ L and |w| ≥ n then there exists x, y, z such that

1. *w = xyz,*
2. *$y \neq \wedge$* (Given assumption)
3. *$|xy| \leq n$ ($|xy| = n$ when $z = \in$) and*
4. *$xy^iz \in L$, for all $i \geq 0$.*

Here,
- n = Number of states.
- w = Any string w∈ L
- |w|= m= Length of string in w (if w = abcd, |w| = 4.)
- y^i = Read i at infinite times.

Proof

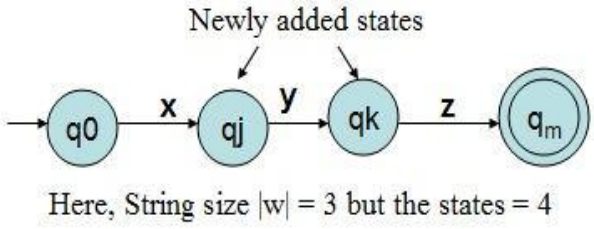
Let w = a1, a2, a3 … a_m, (m is length of string)

δ(q0, a1, a2, a3 … a_i) = qi, (i = 1, 2, 3 … n) = Sequence of states in the path with path value 'w'.

Q' = {q0, q1, q2 … q_n)

Now, the input string 'w' can be decomposed into three substrings as:
- x = a1, a2, a3 … a_j
- y = a_{j+1}, … a_k
- z = a_{k+1}, … a_m

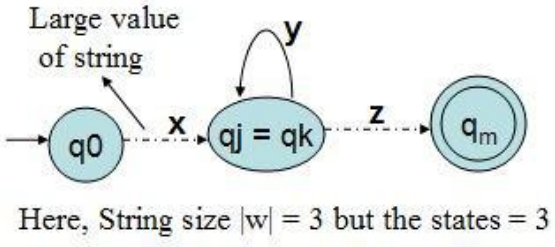
Since, we know that for any regular expression, there exists a path from initial state to final sate with path value the given regular expression. Thus, for 'w' the finite automata can be constructed as shown in figure.



Here, on reading the string xyz (i.e. |w| = m =3), a new states are added on the existing states (i.e. states (n) = 4). But by the definition of pumping lemma, we have, |w| ≥ n. Thus, by using *Pigeon-Hole* principle, there must coincides at least two states in Q, as there are only n distinct states defined but on applying the input string the states becomes n+1. Thus, among various pair of repeated states, we take first as qj and qk (i.e. qj = qk) for merging & hence the path with value w in the transition diagram of M is shown in figure below.

$xy^iz \in L$, for all $i \geq 0$.

- When i =0, w = xy (String is recognized by finite automata)
- When i =1, w = xyz (String is recognized by finite automata)
- When i =2, w = xy^2z (String is recognized by finite automata) and so on. Hence we can conclude that this condition is valid for all regular string.



Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

The figure shows the string accepted by the finite automata M. Here, M starts for the initial state q0 and on applying the string x, it reaches the state qj = qk. Again, on applying y, it comes back to the same state qj = qk until application of y^i, $i \geq 0$ and finally it reaches to q_m on applying z. Thus, the string of 'y' can pump in several numbers of times to generate large string. Hence, $xy^iz \in L$.

For $|xy| \leq n$

The verification of the case is obvious by the help of the given FA defined for the regular string 'w'. Since, $w = xyz = a1, a2 \ldots a_j, a_{j+1} \ldots a_k, a_{k+1} \ldots a_m$, thus we can write $1 \leq j < k \leq m$ and this implies that $|xy| \leq n$.

Hence, we can say that the pumping lemma is proved.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

CHAPTER - 3

Context Free Language and Pushdown Automata

3.1 Context Free Grammar (CFG)

The context free grammar is a ***language generator*** which operates on some set of rules called the production rule. The context-free languages are applied in *parser design*. They are also useful for describing *block structure of programming languages.* In the production rule of context sensitive language, the replacement of any non-terminal symbol have the influence of their respective terminal but in case of the context free language, the non-terminals or variables are replaced independently as we have to *define the production rule for each non-terminals individually*.

A. In Production Rule of Context Free Grammar (CFG)

Non-terminal → Terminals only or Non-terminals only or Combination of terminals and non-terminals

For example: s → a/b/SS/MM, M → p/D/q, D →x/y/z etc. This implies that each non-terminal is separately or individually defined so they are context free on their replacement.

B. In Production Rule of Context Sensitive Grammar (CSG)

Non-terminal only or Combination of non-terminals and terminals → Terminals only or Non-terminals only or Combination of terminals and non-terminals

For example: aaSbc → aab/Sbb/aaS, S → aa/bb etc. In the first production rule, it implies that on replacing the aaSbc, we have to consider the present context of S present ahead and back side also.

Mathematically, a grammar G = (V, Σ, R, S) is a context-free grammar (CFG) if

V = Finite set of non-terminals or variables that are represented by capital letters

Σ = Finite set of terminals that are represented by the small letter or sign or number etc

S = Starting non-terminal symbol and $S \in V$

R = Set of rules called the production rule of the form $\alpha \rightarrow \beta$, where $\alpha \in V$ and $\beta \in (V \cup \Sigma)^*$ (i.e. the LHS of production rule in CFG have only the non-terminals and the RHS may have empty string, terminals, non-terminals or the combination of terminals and non-terminals).

The production rules or productions or rewriting rules is the kernel of any grammar and language specification. The productions are used to derive one string over $V \cup \Sigma$ from another string. In the application of the production rule, the reverse substitution is not permitted i.e. if S →AA then AA →S is not possible.

Note: Is there any relationship between the regular expression and the context free grammar? Define by suitable example.

Solution

We know from the Chomsky hierarchy of grammar, all the regular expressions can be described on the basis of the Context Free Grammar (CFG), but the reverse cannot be true. This can be verified with the following example. Let any regular expression, ***R = a (a*+b*) b***, then the operating rules for R can be generated as:

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

S → aMb (i.e. S = start state = String with M starting with a and ending with b)
M → A/B [i.e. M = (a*+b*) = any number of string of a or b]
A → aA/∧ (i.e. String with any number of a)
B → bB/∧ (i.e. String with any number of b)

Now,

Let G = (V, Σ, R, S) be a context-free grammar (CFG) that can describe the string generated by regular expression, R. Where

V = Finite set of non-terminals or variables = {S, M, A, B}
Σ = Finite set of terminals = {a, b, ∧}
S = Starting non-terminal symbol and S∈V
R = The production rule, which can be describe as: S → aMb, M → A/B, A → aA/∧, & B → bB/∧.

Numerical – 1: ***Write a CFG to generate only the palindrome with the input symbol,Σ = {0, 1}.***

Solution

Let G = (V, Σ, R, S) be a context-free grammar (CFG) that can describe the string of palindrome only.

Where V = Finite set of non-terminals or variables = {S}
Σ = Finite set of terminals = {0, 1, ∧}
S = Starting non-terminal symbol and S∈V
R = Production rule, which can be describe as: S → 0S0/1S1/∧.

Note: The production rule will be S → 0S/1S/∧ or S → SS/1/0/∧ for the palindrome and non-palindrome string generation but not only the palindrome as above.

Derivations

The process of deriving the required string over (V∪Σ)* using the given production rule on the existing string is called the derivation. The string generated by the most recent application of production is called the *working string*. The derivation of a string completed when the working string cannot be modified. The different derivations results are quite different in different sentential form such as *context sensitive grammar*, but for a context free grammar, it really doesn't make much difference in what order you expand the variable.

Suppose that $\alpha_1, \alpha_2 \ldots \alpha_m$ are strings over $(V \cup \Sigma)^*$ and $\alpha_1 \xrightarrow[G]{} \alpha_2 \quad \alpha_2 \xrightarrow[G]{} \alpha_3 \quad \alpha_3 \xrightarrow[G]{} \alpha_4 \quad \cdots \quad \alpha_{(m-1)} \xrightarrow[G]{} \alpha_m$ Then, we say that α_1 derives α_m in grammar G or this can be represented as $\alpha_1 \xrightarrow[G]{*} \alpha_m$

Hence, we call the sequence of the derivation in G of α_m from α_1 in the following manner as: $\alpha_1 \xrightarrow[G]{} \alpha_2 \xrightarrow[G]{} \alpha_3 \xrightarrow[G]{} \alpha_4 \quad \cdots \xrightarrow[G]{} \alpha_m$

The derivations are described as following manner.

- If we describe a string by applying the production rule at once (i.e. one time application) then the string is called ***directly derivable string*** and is denoted by $\alpha \xrightarrow[G]{} \beta$
- If we derive the string by more than one sequence of operations using given production rules then the string is called ***derivable string*** and is denoted by $\alpha \xrightarrow[G]{*} \beta$ or $\alpha \xrightarrow{*} \beta$

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Type of derivation

A. Leftmost derivation

A derivation $A \xrightarrow{*} w$ is called a leftmost derivation if we apply a production only to the leftmost variable at every step. For example, if the given production rules of any CFG are as: S → 0B/1A, A → 0/0S/1AA, & B → 1/1S/0BB then any given string w = 00110101 can be derived using the leftmost derivation in following manner:

S → 0B
→ 00BB (As, B → 0BB)
→ 001B (As, B → 1)
→ 0011S (As, B → 1S)
→ 00110B (As, B → 0B)
→ 001101S (As, B → 1S)
→ 0011010B (As, B → 0B)
→ 00110101(As, B → 1)

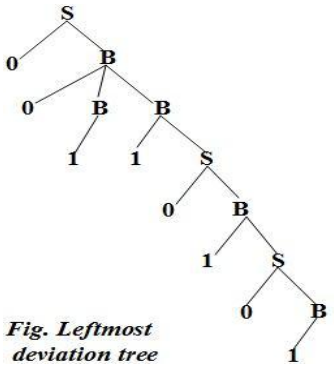


Fig. Leftmost deviation tree

Here, the tree is a derivation tree with yield 00110101.

B. Rightmost derivation

A derivation $A \xrightarrow{*} w$ is called a rightmost derivation if we apply a production only to the rightmost variable at every step. For example, if the given production rules of any CFG are as: S → 0B/1A, A → 0/0S/1AA, & B → 1/1S/0BB then any given string w = 00110101 can be derived using the rightmost derivation in following manner:

S → 0B
→ 00BB (As, B → 0BB)
→ 00B1 (As, B → 1)
→ 001S1 (As, B → 1S)
→ 0011A1 (As, S → 1A)
→ 00110S1 (As, A → 0S)
→ 001101A1 (As, S → 1A)
→ 00110101(As, A → 0)

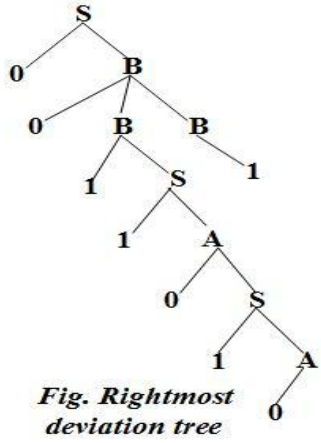


Fig. Rightmost deviation tree

3.2 Representation of CFG

The Context Free Grammar (CFG) can be represented in two ways:

- Derivation tree or Parse tree or Production tree
- Backus Naur Form (BNF)

A. Derivation Tree

It is easy to visualize derivation in context-free languages as we can represent derivations using tree structure. Such tree representing derivations are called derivation trees or *parse trees*. A parse tree is an ordered tree in which nodes are labeled with the left side of production (i.e. non-terminals only) and the children of the nodes (i.e. leaves) represent its corresponding right-sides (i.e. the terminals or non-terminals or both).

A derivation tree for a CFG, G = (V, Σ, R, S) is a tree satisfying the following conditions:

1. Every vertex has label, which is a variable or terminal or empty string (∧).
2. The root has label 'S' (i.e. start symbol).
3. The label of an internal vertex is a variable.
4. Each vertex of variable is extended towards the leaf-node or terminals using the production rule (R).

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

For example, if the given production rules of any CFG are as: S → 0B/1A, A → 0/0S/1AA, & B → 1/1S/0BB then any given string w = 00110101 can be derived using the leftmost derivation in following manner: -

S → 0B
→ 00BB (As, B → 0BB)
→ 001B (As, B → 1)
→ 0011S (As, B → 1S)
→ 00110B (As, B → 0B)
→ 001101S (As, B → 1S)
→ 0011010B (As, B → 0B)
→ 00110101(As, B → 1)

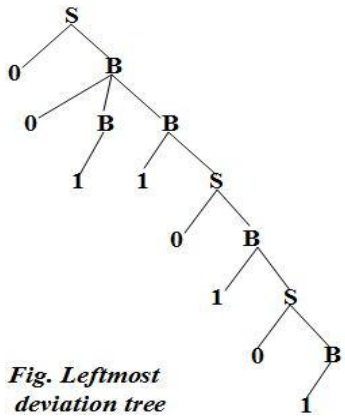


Fig. Leftmost deviation tree

Here, the tree is a derivation tree with yield 00110101. The yield of a derivation tree is the concatenation of the labels of the leaves without repetition in the left-to-right ordering.

Exercise

- Consider a CFG, S → XX, & X → XXX/bX/Xb/a, then find the parse tree for any given string w = bbaaaab.
- Consider the grammar G, with production S → aXY, X → bYb, & Y → X/c, then find the parse tree for any string w = abbbb.

Note: The derivation tree does not specify the order in which we apply the production for getting the required string. So, same derivation tree can include several derivations. But, in general we use leftmost derivation than that of the rightmost derivation.

3.3 Ambiguity in Context-Free Grammar

Grammars are used to put structures on programs or documents. The assumption was that a grammar uniquely determines a structure for each string in the language. However, not every grammar does provide unique structure. Thus, when a grammar fails to provide unique structure, it is called the ambiguous grammar i.e. in this case grammar puts more than one structure for same string in the language.

A Context Free Grammar, G is ambiguous if there exists some terminal string w ∈ L (G) is ambiguous. The terminal string w is ambiguous if there exists two or more leftmost derivations for single w. In other word, the single terminal string w is ambiguous if it may be the yield of two derivation trees.

For example: Consider G = ({S}, {a, b, +, ∗}, R, S), where R consists of S → S+S/S*S/a/b. Now, we have two derivation trees for the terminal string w = a + a ∗ b as given below: -

Leftmost derivation – I

S → S+S
→ a + S∗S
→ a + a ∗ S
→ a + a ∗ b

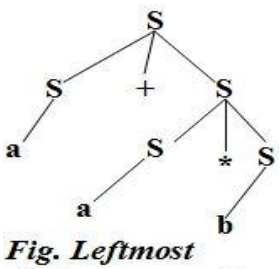


Fig. Leftmost deviation tree -I

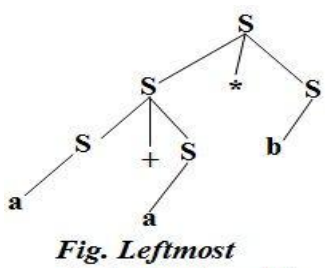


Fig. Leftmost deviation tree -II

Leftmost derivation – II

S → S∗S
→ S + S∗S
→ a + S ∗ S

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

→ a + a ∗ S
→ a + a ∗ b

Since, here exists two leftmost derivation trees for a same terminal string w = a + a ∗ b (i.e. w is ambiguous). Hence, we can conclude that the given grammar G is ambiguous.

Exercise

- Consider any grammar G, with the production rule S → SbS/a. Show that the given grammar is ambiguous. Assume the terminal string w = abababa.

3.4 Normal Form

Since, in CFG, the production rule is of the form $\alpha \rightarrow \beta$, where $\alpha \in V$ and $\beta \in (V \cup \Sigma)^*$ (i.e. the LHS of production rule in CFG have only the non-terminals and the RHS may have empty string, terminals, non-terminals or the combination of terminals and non-terminals). Thus, when we apply some restriction on the R.H.S. of the Context Free Grammar G for defining new production then G is said to be in "normal form". Among several normal forms, we deal with the following two normal forms:

- Chomsky Normal Form (CNF)
- Greibach Normal Form (GNF)

A. Chomsky Normal Form (CNF)

In the Chomsky normal form, we have restrictions on the length of R.H.S. and the nature of symbols in the R.H.S. of production. Here, the restriction is that every node has either two internal vertices (i.e. two non-terminals only) or a single leaf (i.e. exactly one terminal). When a grammar is in CNF, some of the proofs and constrains are simpler.

A context-free grammar G is in Chomsky normal form if every production is of the form A → a and A →BC. Here, A, B and C are non-terminal symbols, 'a' is a terminal symbol, S is the start symbol, and ∧ is the empty string. Also, neither B nor C may be the start symbol but A may be. For **example:** consider G whose productions are S →AB, A → a, & B → c. Then G is in CNF.

Note: Any Context Free Grammar (CFG) can be reduced into the Chomsky normal form (CNF) but the converse is not true as CNF is a restricted form of CFG.

Reduction to Chomsky Normal Form

Any context-free language is generated by a context-free grammar in Chomsky normal form. For every context-free grammar (CFG), there is an equivalent grammar G2 in Chomsky normal form (CNF). Here, we put the non-terminals in place of terminals in the same principle of the *bottom-up parsing*.

Proof idea:

1. Show that any CFG G can be converted into a CFG G′ in Chomsky normal form;
2. Conversion procedure has several stages where the rules that violate Chomsky normal form conditions are replaced with equivalent rules that satisfy these conditions;
3. Order of transformations: (1) add a new start variable, (2) eliminate all null production, (3) eliminate unit-productions, (4) Elimination of terminals on R.H.S.
4. Check that the obtained CFG G′ define the same language as the initial CFG G by restricting the number of variables on R.H.S.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

For example: If the given production rules of context-free grammar, G is given as: S →aAD, A → aB/bAB, B → b, & D → d, then construct the equivalent CNF.

Solution

Since in CNF, the restriction is that every nodes on R.H.S. has either two internal vertices (i.e. two non-terminals only) or a single leaf (i.e. exactly one terminal). Also, there are no null productions or unit production. Thus, the production rule can be constructed as: -

- B → b & D → d are in R'.
- S →aAD gives S →C_aAD, where C_a → a and also S →C_aAD, gives S →C_aC_1, where C_1 → AD.
- A → aB gives A → C_aB, where C_a → a
- A → bAB gives A → BAB, where B → b and also A → BAB gives A → BC_b, where C_b→ AB.

Hence, Let G' = (V', Σ, R', S') be newly constructed CNF equivalent to the given CFG, where

V' = Set of non-terminals only = {S, A, B, D, C_a, C_b, C_1}

Σ = Set of terminals only = {a, b, d}

S' = Start state = S

R' = Set of production rule that can be defined as: S →C_aC_1, A→ C_aB, A → BC_b, C_a → a, C_b→ AB, C_1 → AD, B → b & D → d.

For example: If the given production rules of context-free grammar, G is given as: S →aBASA/aBA, A → aAA/a, & B → bBB/b then construct the equivalent CNF.

Solution

We can construct the CNF for the given CFG by defining the production rule as:

- B → b & A → a.
- S →aBASA gives S →C_aBASA, where C_a → a
  S →C_aBASA, gives S →C_aC_1SA, where C_1 → BA.
  S →C_aC_1SA, gives S →$C_aC_1C_2$, where C_2 → SA.
  S →$C_aC_1C_2$, gives S →C_aC_3, where C_3 → C_1C_2.
  - S →aBA gives S →C_aBA, where C_a → a

  S →C_aBA gives S →C_aC_1, where C_1 → BA.
  - A → aAA gives A → C_aAA, where C_a → a.

  A → C_aAA gives A → C_aC_4, where C_4 → AA.
  - B → bBB gives B → C_bBB, where C_b→ b.

  B → C_bBB gives B → C_bC_5, where C_5→ BB.

Hence, Let G' = (V', Σ, R', S') be newly constructed CNF equivalent to the given CFG, where

V' = Set of non-terminals only = {S, A, B, C_a, C_b, C_1, C_2, C_3, C_4, C_5}

Σ = Set of terminals only = {a, b}

S' = Start state = S

R' = Set of production rule that can be defined as: S →C_aC_3/C_aC_1, C_a → a, C_1 → BA, C_2 → SA, C_3 → C_1C_2, S →C_aC_1, A → C_aC_4, C_4 → AA, B → C_bC_5, C_b→ b & C_5→ BB.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

B. Greibach Normal Form (GNF)

GNF is another normal form quite useful in some proofs and constructions. A context free grammar generating the set accepted by a pushdown is in GNF. A grammar in GNF is a natural generalization of a regular grammar as the production of the regular grammar are of the form $A \rightarrow a\alpha$, where $\alpha \in V^*$ and $a \in \Sigma$.

A context-free grammar is in Greibach normal form if every production is in the form $A \rightarrow a\alpha$, where $\alpha \in V^*$ and 'a' is only one terminal, $a \in \Sigma$. Hence, any CFG will be in GNF if it is in the form "non-terminal → exactly one terminal followed by any number of non-terminals including null (∧).For example, the grammar G with productions S → aAB, A →bC, B→b, C →c is in GNF.

Conversation of CFG into GNF

Example: If the given production rules of context-free grammar, G is given as: S →abaSa/aba, then construct the equivalent GNF.

Solution

We can construct the GNF for the given CFG by defining the production rule.
Here, we have

S →abaSa/aba

Now, let us introduce new variables A and B and productions A →a & B→b and substitute into the given grammar as

S →aBASA/aBA
A →a &
B→b

Hence, Let G' = (V', Σ, R', S') be newly constructed GNF equivalent to the given CFG, where

V' = Set of non-terminals only = {S, A, B}
Σ = Set of terminals only = {a, b}
S' = Start state = S
R' = Set of production rule that can be defined as: S →aBASA/aBA, A →a & B→b.

Example: If the given production rules of context-free grammar, G is given as: S →AB, A →aA/bB/b & B→b then construct the equivalent GNF.

Solution

We can construct the GNF for the given CFG by defining the production rule as: -

- S →AB gives S →aAB, (Since A →aA)
- S →AB gives S →bBB, (Since A →bB)
- S →AB gives S →bB, (Since A →b)
- A →aA/bB/b
- B → b

Now, Let G' = (V', Σ, R', S') be newly constructed GNF equivalent to the given CFG.
Where

V' = Set of non-terminals only = {S, A, B}
Σ = Set of terminals only = {a, b}
S' = Start state = S
R' = Set of production rule that can be defined as: S →aAB/bBB/bB, A → aA/bB/b & B→b.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

3.5 Simplification of CFG

The unit production, useless productions & the useless production in CFG makes it ambiguous & thus there is not any unique structure for any language generated by using the grammar and also the cost of production will increase. Thus, for simplification of CFG, we attempt to do following procedures:

- Removal of unit productions
- Removal of null productions
- Removal of useless productions

A. Elimination of Unit Productions

A Context Free Grammar G may have production of the form A → B, where A & B are variable in G, (i.e. one non-terminal → one non-terminal) is called unit production. On removing the unit productions, we analyze the non-terminals by substitution of the terminals. The unit production increases the cost of the derivation in grammar.

Example: If the given production rules of context-free grammar, G is given as: S →AB, A →a, B→C/b, C→D, D →E & E→a, then remove the unit productions.

Solution

The given grammar contain the following unit productions

B→C,
C→D, &
D →E

Also, the terminals given by the non-terminals are

A →a,
B →b, gives C →b and hence it generates B →b/b or B →b which is given
E →a, gives D →a & hence C → a and finally we get B →a

Now, the useless production for the given CFG can be distinguished by analyzing the not reachable non-terminals from the start symbol. Here, the non-terminals are A & B are only reachable from start symbol and others are useless productions i.e. C →b, E →a & D →a are useless or never be used. Hence, we can eliminate them.

Now, Let G' = (V', Σ, R', S') be newly constructed CFG, which is completely reachable grammar.

Where

V' = Set of non-terminals only = {S, A, B}
Σ = Set of terminals only = {a, b}
S' = Start state = S
R' = Set of production rule that can be defined as: S →AB, A → a & B→a/b.

Example: If the given production rules of context-free grammar, G is given as: S →A/bb, A →B/b, & B→S/a, then remove the unit productions.

Solution

The given grammar contain the following unit productions

S→A,
A→B, &
B →S

Also, the terminals given by the non-terminals are

B →a, gives S →a & hence A →a
A →b, gives S →b & hence B →b

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

S →bb, gives B →bb & hence A → bb

Now, Let G' = (V', Σ, R', S') be newly constructed CFG, which holds no unit productions.

Where

V' = Set of non-terminals only = {S, A, B}

Σ = Set of terminals only = {a, b}

S' = Start state = S

R' = Set of production rule that can be defined as: S →a/b/bb, A → a/b/bb & B→a/b/bb.

Here, the productions of the non-terminals A and B are useless as they are not included on the start symbol or we can say that we can generate the entire possible string only by using the start symbol so we can eliminate the production A & B. Hence the final production rule is only S →a/b/bb.

B. Elimination of Null Productions

A CFG may have production of the form A → ∧, where A is any variable, is called a null production. The production A → ∧ is just used to erase A.

To eliminate the null production, we use the following procedure. If A →∧ is a production to be eliminate then we look for all productions, whose right side contains 'A' and replace each occurrence of 'A' in each of these productions to obtain the non-null productions only. Now, these resultant non-null productions must be added to the grammar.

Example: If the given production rules of context-free grammar, G is given as: S →aA, & A →b/∧, then remove the null productions.

Solution

The given grammar has null productions A→∧. So, put null (∧) in place of 'A' at the right side of productions and add the resultant productions to the grammar.

Thus,

S →aA, gives S →a/∧ & hence S →a

Hence, Let G' = (V', Σ, R', S') be newly constructed null-free CFG.

Where

V' = Set of non-terminals only = {S, A}

Σ = Set of terminals only = {a, b}

S' = Start state = S

R' = Set of production rule that can be defined as: S →a, & A → b.

Here, the production of the non-terminals A is useless as they are not included on the start symbol. Hence the final production rule is only S →a.

Example: If the given production rules of context-free grammar, G is given as: S →ABAC, A →aA/∧, B →bB/∧, & C →c then remove the null productions.

Solution

The given grammar contains the following null productions:

A→∧

B→∧

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

So, put null (∧) in place of 'A' at the right side of productions, we get

S →BAC, S →ABC, S →BC, and A →a

Thus, the simplified grammar become: S →ABAC/BAC/ABC/BC, and A →a

Again, put null (∧) in place of 'B' at the right side of productions, we get

S →AAC, S →AC, S →C, and B →b

Thus, the final simplified grammar become: S →ABAC/BAC/ABC/BC/AAC/AC/C, A →a and B →b

Hence, Let G' = (V', Σ, R', S') be newly constructed null-free CFG.

Where

V' = Set of non-terminals only = {S, A, C}

Σ = Set of terminals only = {a, b}

S' = Start state = S

R' = Set of production rule as: S →ABAC/BAC/ABC/BC/AAC/AC/C, A →a and B →b

Exercise: Consider the following grammar and remove the null productions.

- S →aSa /bSb/∧
- S →a /Xb/aYa/∧, X → Y/∧, Y →b/∧.

C. Elimination of Useless Productions

For the identification of the useful production, the following two points should be noted: -

a. Can a production generate a string or terminal symbol? This means that not generating is useless productions.
b. Can a non-terminal symbols (or Variables) are reachable from the start symbol? This means that non reachable are useless productions.

Example: Eliminate the useless productions from context-free grammar, G where V = {S, A, B, C} and Σ = {a, b} with the productions S →aS/A/C, A →a, B →aa, & C →aCb.

Solution

→ First identify the set of variables that can lead to a terminal string.

i.e. A →a,

B →aa, and

S →aS/A

Since, C cannot generate any string so we remove C. Now, we get a new context-free grammar, G1 having V1 = {S, A, B} and Σ1 = {a} with the production rule R1 defined as: S →aS/A, A →a, B →aa.

→ Next step is the elimination of the variables that cannot be reached from the start variable or symbol. For this, we draw a dependency graph (or transition diagram) where its vertices are labelled with non-terminals and an edge between C and D is connected if and only if there is a production of the form C →xDy.

Here, the non-terminal B is useless. Hence the specified grammar is G' = (V', Σ', R', S')

Where

V' = Set of non-terminals only = {S, A}

Σ' = Set of terminals only = {a}

S' = Start state = S

R' = Set of production rule as: S →aS/A, A →a.

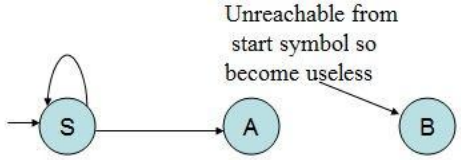


Fig. Dependency graph for Useless production

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

3.6 Closure Properties of CFL

The closure property of the CFL is defined itself by the CFL i.e. the language generator so it can be distinguish from the closure property of the regular set as defined earlier. The class of language of context-free language is closed under: Union, Concatenation, and Kleene closure.

A. Union

Let L_1 and L_2 be two context-free languages generated by the Context-free Grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ respectively. Now, we construct new language 'L (G)' using the grammar $G = (V, \Sigma, R, S)$, such that it can accepts $L(G_1) \cup L(G_2)$.

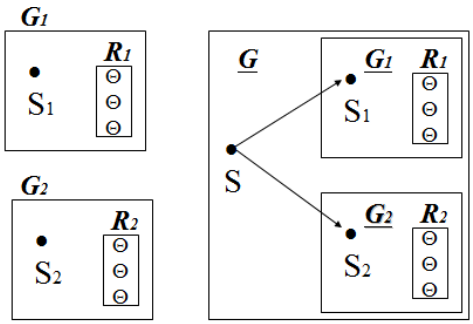


Fig. CFG is closed under union

Where,

- $V = V_1 \cup V_2 \cup \{S\}$
- Σ = Set of input states = $\Sigma_1 \cup \Sigma_2$
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1/S_2\}$
- S = Start state.

Now, let us choose a string $w \in \{\Sigma_1 \cup \Sigma_2\}^*$ if $S_1 \xrightarrow{*} w$ and $S_2 \xrightarrow{*} w$ and in our grammar $S \rightarrow S_1/S_2$, so S will lead the string w.

Hence, G is a context-free grammar that can generate L (G) such that $L(G) = L(G_1) \cup L(G_2)$.

B. Concatenation

Let L_1 and L_2 be two context-free languages generated by the Context-free Grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ respectively. Now, we construct new language 'L (G)' using the grammar $G = (V, \Sigma, R, S)$, such that it can accepts $L(G_1) \cup L(G_2)$.

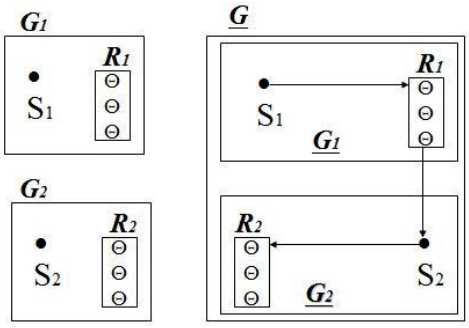


Fig. CFL is closed under Concatenation

Where,

- $V = V_1 \cup V_2 \cup \{S\}$
- Σ = Set of input states = $\Sigma_1 \cup \Sigma_2$
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1.S_2\}$
- S = Start state.

Now, let us choose string $w_1 \in \Sigma_1$ and $w_1 \in \Sigma_2$. We know that $S_1 \xrightarrow{*} w$ and $S_2 \xrightarrow{*} w$, but in the above grammar G, $S \rightarrow S_1S_2$, so S will lead the concatenation of the string w_1 & w_2 i.e. $w_1\, w_2$ and the language will be L_1L_2. Since L_1 & L_2 are CFL so L_1L_2 is also CFL.

C. Kleene closure

Let L_1 be a context-free languages generated by the Context-free Grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$. Now, we construct new language 'L (G)' using the grammar $G = (V, \Sigma, R, S)$, such that it can accepts Kleene Star of the language L (or L*).

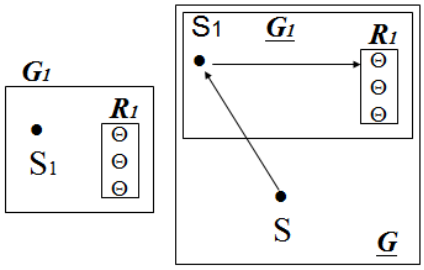


Fig. CFL is closed under Kleene Closure

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Where,

- $V = V_1 \cup \{S\}$
- Σ = Set of input states = $\Sigma_1 \cup \{\wedge\}$
- $R = R_1 \cup \{S \rightarrow S_1, S \rightarrow \wedge, S \rightarrow S_1S_1\}$
- S = Start state.

Here, R follows all the properties of CFG as R1 is a production of given CFG and $S \rightarrow S_1$, $S \rightarrow \wedge$, & $S \rightarrow S_1S_1$ also fulfill the requirement so we say that G is a CFG that can generate the context-free language L*.

Decision Algorithm for Context-Free Language

- Algorithm for deciding whether a CFL is empty (*emptiness*).
- Algorithm for deciding whether a CFL is finite (*finiteness*).
- Algorithm for deciding whether any string 'w' can be generated by the same CFG (Membership).

3.7 Pumping Lemma for Context-Free Languages

The pumping lemma for the context-free language (called *Bar-Hillel lemma* or just "*the pumping lemma*") gives a method of generating an infinite number of strings from a given sufficiently long string in a context-free language L. It is used to prove that certain languages are not context-free.

While the pumping lemma is often a useful tool to prove that a given language is not context-free, it does not give a complete characterization of the context-free languages. If a language does not satisfy the condition given by the pumping lemma, we have established that it is not context-free. On the other hand, there are languages that are not context-free, but still satisfy the condition given by the pumping lemma. There are more powerful proof techniques available, such as ***Ogden's lemma***, but also these techniques do not give a complete characterization of the context-free languages.

Statement

Let L be a context-free language and n be the length of the string or *pumping length* such that:

i. **Every $z \in L$ with $|z| = n$ can be written as uvwxy for some u, v, w, x & y.** (*i.e. any string z can be decompose into five sub-strings u, v, w, x & y*)
ii. **$|vx| \geq 1$** (*i.e. may one null at a time but not both or must have at least one string because we have to pump at least one sub-string to generate the new infinite number of string*)
iii. **$|vwx| \leq n$** (*i.e. if u & y are $\wedge$, then vwx = n*)
iv. **$uv^kwx^ky \in L$ for all $k \geq 0$** (*i.e. generate infinite number of string by setting any value of k*)

Proof

To prove the theorem, we consider a CFG whose productions are given by: $S \rightarrow AB$, $A \rightarrow aB/a$, $B \rightarrow bA/b$.

Now, let any string z = ababb such that $z \in L$. Thus we can decompose z as u = a, v = ba, w = b, x = $\wedge$ and y = b.

Since, the string z and z_1 are the yield of the tree T. Thus, we can write: **$z = uz_1vy$**

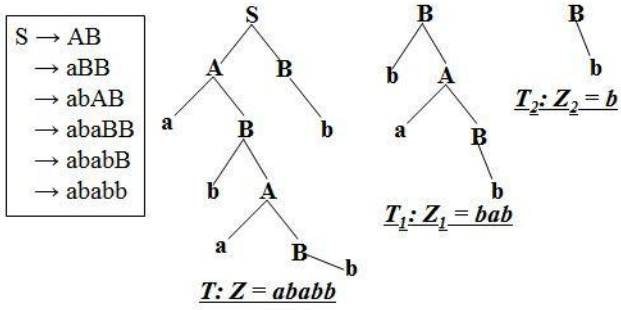


Fig. Derivation & tree T and it's sub-trees T_1 & T_2

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Again, the string z_1 & z_2 are the yield of the tree T1. Thus, we can write: $\mathbf{z_1 = vwx}$. Also, | vwx | > |w| so $|vx| \geq 1$
Hence, we have
z = uvwxy with $|vwx| \leq n$ & $|vx| \geq 1$
As T is an S-tree and T_1 & T_2 are the B-tree, we get $S \xrightarrow{*} uBy, B \xrightarrow{*} vBx, \& \; B \xrightarrow{*} w$
Now,

$S \rightarrow uBy$
$\rightarrow uvBxy$ (since, $B \rightarrow uBx$)
$\rightarrow uvwxy$ (since, $B \rightarrow w$)
Thus, $S \rightarrow uv^1wx^1y$, where k = 1

Similarly,

$S \rightarrow uBy$
$\rightarrow uvBxy$ (since, $B \rightarrow uBx$)
$\rightarrow uvuBxxy$ (since, $B \rightarrow uBx$)
$\rightarrow uvvwxxy$ (since, $B \rightarrow w$)
Thus, $S \rightarrow uv^2wx^2y$, where k = 2 and so on.

Hence, we can conclude that $S \rightarrow uBy$ gives $S \xrightarrow{*} uv^kwx^ky \in L$ This proves the theorem.

Example: Prove that the language $L = \{a^nb^nc^n \mid n \geq 0\}$ is not context-free language.

Solution
Let us consider 'L' is a context-free language. Also, let $\mathbf{z = a^pb^pc^p}$, $\mathbf{z \in L}$.
Now, according to the pumping lemma of the CFL, the string 'z' can be decomposes into u, v, w, x, & y as follow:

$\mathbf{u = a^r}$
$\mathbf{v = a^s \; (s > 0)}$
$\mathbf{w = a^{p-(r+s)}}$
$\mathbf{x = b^t \; (t > 0)}$
$\mathbf{y = b^{(p-t)}c^p}$

Now, using the pumping lemma, $z = uv^kwx^ky$; $k \geq 0$, we have
$\mathbf{z = uv^2wx^2y = a^r\,(a^s)^2\,a^{p-(r+s)}\,(b^t)^2\,b^{(p-t)}c^p = a^r\,a^{2s}\,a^{p-(r+s)}\,b^{2t}\,b^{(p-t)}c^p = a^{(p+s)}\,b^{(p+t)}c^p \neq a^pb^pc^p}$
Here, our assumption $\mathbf{z \in L}$, contradict with our result.
Hence, we can conclude that the language $L = \{a^nb^nc^n \mid n \geq 0\}$ is not context-free language.

Example: Prove that the language $L = \{a^n \mid n \geq 0\}$ is not context-free language.

Solution
Let us consider 'L' is a context-free language. Also, let $\mathbf{z = a^p}$, $\mathbf{z \in L}$.
Now, according to the pumping lemma of the CFL, the string 'z' can be decomposes into u, v, w, x, & y as follow:

$\mathbf{u = a^\alpha, \; v = a^\beta \; (\beta > 0), \; w = a^{q-(\alpha+\beta)}, \; x = a^\gamma \; (\gamma > 0), \; \& \; y = a^{p-(q+\gamma)}}$

Now, using the pumping lemma, $z = uv^kwx^ky$; $k \geq 0$, we have
$\mathbf{z = uv^2wx^2y = a^\alpha\,(a^\beta)^2\,a^{q-(\alpha+\beta)}\,(a^\gamma)^2\,a^{p-(q+\gamma)} = a^\alpha\,a^{2\beta}\,a^{q-(\alpha+\beta)}\,a^{2\gamma}\,a^{p-(q+\gamma)} = a^{p+\alpha+\beta} \neq a^p}$
Here, our assumption $\mathbf{z \in L}$, contradict with our result.
Hence, we can conclude that the language $L = \{a^n \mid n \geq 0\}$ is not context-free language.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

3.8 Pushdown Automata

Finite automata provide only finite amount of memory, thus preventing us from recognizing language that require remembering an infinite amount of information during the processing of a string. For example a finite automat cannot accept the string of the form $L = a^n b^n$ $(n \geq 1)$ as it has to remember the number of a's in a string and so it will require infinite number of states. This difficulty can be avoided by adding memory capability to finite automata. The addition of infinite stack to the finite automata leads to what we call the *push down automata* (pda).

Thus, the push down automata is essentially a finite automaton with control of both input tape and a stack to store what it has read. Hence, the pushdown automata consist of the following three things:

- An input tape
- A finite control
- A stack

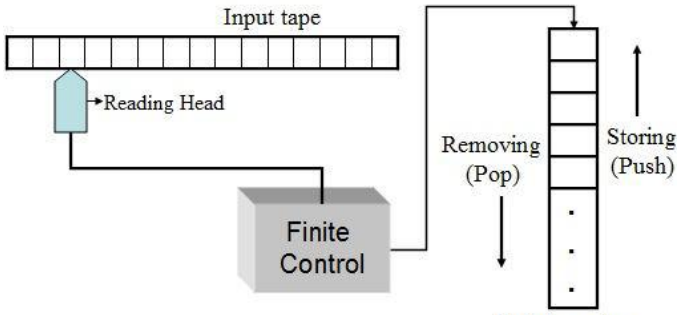


Fig. B.D. of a Pushdown automata

Here is an input tape from which the finite control reads the input and same time it reads the symbol from the stack. Now it depends upon the finite control that what is the next state and what will happen with the stack. In one transition, the pushdown automata do the following:

- Consume the input symbol that it uses in the transition. If the input is $\wedge$, then there is no consumption of input symbol.
- Goes to the new state, which may or may not be the same as the previous state.
- Replace the symbol at the top of the stack by any string. It could be the same symbol that appeared at the top of the stack.

Mathematically, the pushdown automata, is six tuple, can be defined as $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Where

- Q is the non-empty finite set of states.
- Σ is the non-empty finite set of input symbol.
- Γ is a finite non-empty set of pushdown or stack symbol
- q_0 is the start state, $q_0 \in Q$.
- F is a set of final states, $F \subseteq Q$.
- δ is a transition function which maps $(Q \times \Sigma^* \times \Gamma^*) \rightarrow (Q \times \Gamma^*)$. Formally δ takes an argument, a triple as $\delta(q, a, x)$ where,
  - q is a state in Q.
  - a is an input symbol.
  - x is a stack symbol that is a member of Γ.

  The output of δ is a finite set of pair (p, r), where 'p' is the new state and r is the string of the stack symbol that replaces x at the top of the stack.

Move of PDA

The pushdown automata consist of following moves:

* $\delta(q, a, x) \rightarrow (p, y)$ means that whenever PDA is in state q with x, the top of stack, may read 'a' from the input tape, replace 'x' by 'y' on the top of stack and enter state p.
* $\delta(q, a, \wedge) \rightarrow (p, y)$, indicates the pushes 'y' on the top of the stack.
* $\delta(q, a, y) \rightarrow (p, \wedge)$, indicates the pops a symbol 'y' from the top of the stack.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Example: Design a PDA which accepts a language L = {w = $a^n b^n$: n≥1}

Solution

To solve the problem, we have to first analyze the given language so that we can generate the PDA for it. As we can see, the string must have equal number of 'a' and 'b' and the order of placement is number of 'a' followed by number of 'b'. Thus, to design such PDA, we have to read the number of 'a' and then the same number of 'b' and finally, the input string and the stack must be empty for the accepting condition. To do so, we have to push first the whole string of 'a', one by one, into the empty stack and when the string of 'b' start to read, pop one by one the string of 'a' on stack in each 'b' read. Hence, when the input string is consumed, there is nothing on the stack also and that we termed as the accepting condition of the PDA.

Now, let the required PDA for the given string be: P = (Q, Σ, Γ, δ, q_0, F), Where

- Q = finite set of states = {q_0, q, q_f }
- Σ = finite set of input symbol = {a, b}
- Γ = finite set of stack symbol = {a}
- q_0 = start state
- F = set of final states = { q_f }
- δ = transition function which can be defined as
  i. δ (q_0, a, ∧) → (q_1, a)
  ii. δ (q_1, a, a) → (q_1, aa)
  iii. δ (q_1, b, a) → (q_1, ∧)
  iv. δ (q_1, ∧, ∧) → (q_f, ∧)

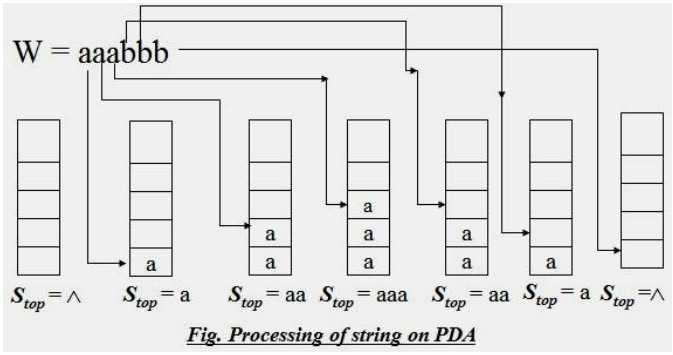


Fig. Processing of string on PDA

For example, let w = aaabbb is any string then we can process it using PDA defined above in tabular form:

Present States	*Unread Input*	*Present Stack symbols*	*Next States*	*Next Stack symbols*	*Transition Used*
q_0	→aaabbb	∧	q_1	a	I
q_1	→aabbb	a	q_1	aa	II
q_1	→abbb	aa	q_1	aaa	II
q_1	→bbb	aaa	q_1	aa	III
q_1	→bb	aa	q_1	a	III
q_1	→b	a	q_1	∧	III
q_1	→∧	∧	q_f		IV

Example: Design a PDA which accepts a language L = (w ∈ {a, b}* where w has equal number of a & b).

Solution

As we can see, the string must have equal number of 'a' and 'b' with any order of placement. Hence, let us consider P = (Q, Σ, Γ, δ, q_0, F) be the required PDA for the given string where

- Q = finite set of states = {q_0, q_f }
- Σ = finite set of input symbol = {a, b}
- Γ = finite set of stack symbol = {a, b}
- q_0 = start state
- F = set of final states = { q_f }
- δ = transition function which can be defined as
  i. δ (q_0, a, ∧) → (q_0, a)
  ii. δ (q_0, b, ∧) → (q_0, b)

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

iii. $\delta(q_0, a, a) \rightarrow (q_0, aa)$
iv. $\delta(q_0, b, b) \rightarrow (q_0, bb)$
v. $\delta(q_0, a, b) \rightarrow (q_0, \wedge)$
vi. $\delta(q_0, b, a) \rightarrow (q_0, \wedge)$
vii. $\delta(q_0, \wedge, \wedge) \rightarrow (q_f, \wedge)$

For example, let w = abbbaaba is any string then we can process it using PDA defined above in tabular form:

Present States	***Unread Input***	***Stack top***	***Next States***	***New Stack-top***	***Transition Used***
q_0	→abbbaaba	$\wedge$	q_0	a	I
q_0	→bbbaaba	a	q_0	$\wedge$	VI
q_0	→bbaaba	$\wedge$	q_0	b	II
q_0	→baaba	b	q_0	bb	IV
q_0	→aaba	bb	q_0	b	V
q_0	→aba	b	q_0	$\wedge$	V
q_0	→ba	$\wedge$	q_0	b	II
q_0	→a	b	q_0	$\wedge$	V
q_0	→$\wedge$	$\wedge$	q_f	$\wedge$	VII

This can also be done in the following manner:

- $\delta(q_0, \boldsymbol{a}bbbaaba, \wedge) \rightarrow (q_0, a)$ *by rule I.*
- $\delta(q_0, \boldsymbol{b}bbaaba, a) \rightarrow (q_0, \wedge)$ *by rule VI.*
- $\delta(q_0, \boldsymbol{b}baaba, \wedge) \rightarrow (q_0, b)$ *by rule II.*
- $\delta(q_0, \boldsymbol{b}aaba, b) \rightarrow (q_0, bb)$ *by rule IV.*
- $\delta(q_0, \boldsymbol{a}aba, bb) \rightarrow (q_0, b)$ *by rule V.*
- $\delta(q_0, \boldsymbol{a}ba, b) \rightarrow (q_0, \wedge)$ *by rule V.*
- $\delta(q_0, \boldsymbol{b}a, \wedge) \rightarrow (q_0, b)$ *by rule II.*
- $\delta(q_0, \boldsymbol{a}, b) \rightarrow (q_0, \wedge)$ *by rule V.*
- $\delta(q_1, \wedge, \wedge) \rightarrow (q_f, \wedge)$, *by rule VII*, which is the accepting state.

Example: Design a PDA which accepts a language L = (wcw^T, w ∈ {a, b}* i.e. w belong to any strings of a & b).

Solution

As we can see, the string of any number of 'a' or 'b' must be complement on both side of 'c'. Hence, let us consider that P = (Q, Σ, Γ, δ, q_0, F) be the required PDA for the given string where

- Q = finite set of states = {q_0, q_1, q_f}
- Σ = finite set of input symbol = {a, c, b}
- Γ = finite set of stack symbol = {a, b}
- q_0 = start state
- F = set of final states = { q_f }
- δ = transition function which can be defined as
  i. $\delta(q_0, a, \wedge) \rightarrow (q_0, a)$
  ii. $\delta(q_0, a, a) \rightarrow (q_0, aa)$
  iii. $\delta(q_0, a, b) \rightarrow (q_0, ab)$
  iv. $\delta(q_0, b, b) \rightarrow (q_0, bb)$
  v. $\delta(q_0, b, a) \rightarrow (q_0, ba)$
  vi. $\delta(q_0, c, a) \rightarrow (q_1, a)$ i.e. only state change.
  vii. $\delta(q_0, c, b) \rightarrow (q_1, b)$ i.e. only state change.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

viii. $\delta(q_1, a, a) \rightarrow (q_1, \wedge)$
ix. $\delta(q_1, b, a) \rightarrow (q_1, \wedge)$
x. $\delta(q_1, a, b) \rightarrow (q_1, \wedge)$
xi. $\delta(q_0, b, \wedge) \rightarrow (q_0, b)$
xii. $\delta(q_1, b, b) \rightarrow (q_1, \wedge)$
xiii. $\delta(q_1, \wedge, \wedge) \rightarrow (q_f, \wedge)$, which is the accepting state.

For example, let w = aabcbaa is any string then we can process it using PDA defined above in following tabular form:

Present States	***Unread Input***	***Stack top***	***Next States***	***New Stack-top***	***Transition Used***
q_0	→ aabcbaa	$\wedge$	q_0	a	I
q_0	→ abcbaa	a	q_0	aa	II
q_0	→ bcbaa	aa	q_0	baa	VIII
q_0	→ cbaa	baa	q_1	baa	IX
q_1	→ baa	baa	q_1	aa	X
q_1	→aa	aa	q_1	a	V
q_1	→a	a	q_1	$\wedge$	V
q_1	→$\wedge$	$\wedge$	q_f	$\wedge$	XI

Example: Design a PDA which accepts a language L = $\{a^n b^{2n}, n > 0\}$.

Solution

As we can see, the string must have just half number of 'a' than that of 'b' with order of placement is 'a' followed by 'b'. Hence, let us consider P = $(Q, \Sigma, \Gamma, \delta, q_0, F)$ be the required PDA for the given string where

- Q = finite set of states = $\{q_0, q_1, q_f\}$
- Σ = finite set of input symbol = {a, b}
- Γ = finite set of stack symbol = {a}
- q_0 = start state
- F = set of final states = $\{q_f\}$
- δ = transition function which can be defined as
  i. $\delta(q_0, a, \wedge) \rightarrow (q_1, aa)$
  ii. $\delta(q_1, a, a) \rightarrow (q_1, aaa)$
  iii. $\delta(q_1, b, a) \rightarrow (q_1, \wedge)$
  iv. $\delta(q_1, \wedge, \wedge) \rightarrow (q_f, \wedge)$

For example, let w = aabbbb is any string then we can process it using PDA defined above in following tabular form:

Present States	***Unread Input***	***Present Stack symbols***	***Next States***	***Next Stack symbols***	***Transition Used***
q_0	→aabbbb	$\wedge$	q_1	aa	I
q_1	→abbb	aa	q_1	aaaa	II
q_1	→bbbb	aaaa	q_1	aaa	II
q_1	→bbb	aaa	q_1	aa	III
q_1	→bb	aa	q_1	a	III
q_1	→b	a	q_1	$\wedge$	III
q_1	→$\wedge$	$\wedge$	q_f		IV

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Exercise: -Design a PDA which accepts a language L = $\{a^n b^{n+1}, n>0\}$.
Hind: - We can construct the string of the form L = a^n ***b*** b^n

3.9 Context-Free Grammar (CFG) to Pushdown Automata (PDA) Conversation

The CFG generates the language accepted by the PDA so we can construct the PDA equivalent to the given Context-Free Grammar.

Example: Construct a PDA equivalent to the following CFG: S → 0BB, B → 0S/1S/0. Also, test whether o10000 is accepted by PDA or not.

Solution

We can define the PDA as: P = (Q, Σ, Γ, δ, q_0, F) where

- Q = finite set of states = {q_0, q_f }
- Σ = finite set of input symbol = {0, 1}
- Γ = finite set of stack symbol = {S, B, 0, 1}
- q_0 = start state
- F = set of final states = { q_f }
- δ = transition function which can be defined by the following rules:
  i. δ (q_0, ∧, ∧) → (q_f, S)
  ii. δ (q_f, ∧, S) → (q_f, 0BB)
  iii. δ (q_f, ∧, B) →{(q_f, 0S), (q_f, 1S), (q_f, 0)}
  iv. δ (q_f, 0, 0) → (q_f, ∧)
  v. δ (q_f, 1, 1) → (q_f, ∧) For the given string w = 010000, we can process it using the defined transition rules.

- **On the basis of stack processing**

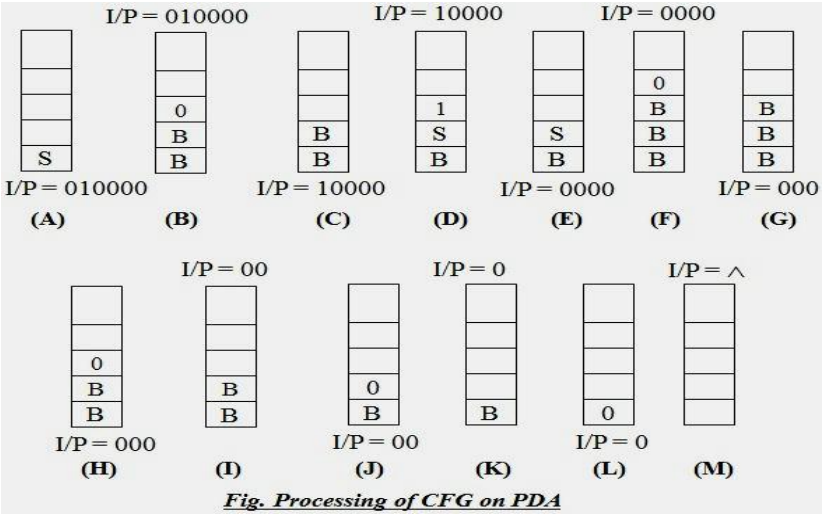


Fig. Processing of CFG on PDA

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

- **Tabular form**

Present States	***Unread Input***	***Stack top***	***Next States***	***New Stack-top***	***Transition Used***
q_0	$\rightarrow \wedge 010000$	$\wedge$	q_f	S	I
q_f	$\rightarrow \wedge 010000$	S	q_f	0BB	II
q_f	$\rightarrow 010000$	0BB	q_f	BB	IV
q_f	$\rightarrow \wedge 10000$	BB	q_f	1SB	III
q_f	$\rightarrow 10000$	1SB	q_f	SB	V
q_f	$\rightarrow \wedge 0000$	SB	q_f	0BBB	II
q_f	$\rightarrow 0000$	0BBB	q_f	BBB	IV
q_f	$\rightarrow \wedge 000$	BBB	q_f	0BB	III
q_f	$\rightarrow 000$	0BB	q_f	BB	IV
q_f	$\rightarrow \wedge 00$	BB	q_f	0B	III
q_f	$\rightarrow 00$	0B	q_f	B	IV
q_f	$\rightarrow \wedge 0$	B	q_f	0	III
q_f	$\rightarrow 0$	0	q_f	$\wedge$	IV
	$\rightarrow \wedge$	$\wedge$	q_f		

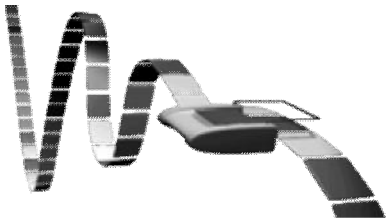
- This can also be done in the following manner: -
  - $\delta(q_0, 010000, \wedge) \rightarrow \delta(q_f, 010000, 0BB) \rightarrow \delta(q_f, 10000, BB)$
    $\rightarrow \delta(q_f, 10000, 1SB) \rightarrow \delta(q_f, 0000, SB)$
    $\rightarrow \delta(q_f, 0000, 0BBB) \rightarrow \delta(q_f, 000, BBB)$
    $\rightarrow \delta(q_f, 000, 0BB) \rightarrow \delta(q_f, 00, BB)$
    $\rightarrow \delta(q_f, 00, 0B) \rightarrow \delta(q_f, 0, B)$
    $\rightarrow \delta(q_f, 0, 0)$
    $\rightarrow \delta(q_f, \wedge, \wedge)$, which is the accepting condition.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

CHAPTER - 4

Turing Machine and Undecidability

4.1 Introduction to Turing Machine



Turing machine provides an ideal theoretical model (as it has infinite memory on tape) of a computer that accepts type-0 language. It is used for computing functions & turns out to be a mathematical model of partial recursive functions. Turing machines are also used for determining the un-decidability of certain language and measuring the space & time complexity.

A Turing machine is a mathematical model of a general computing machine. It is a theoretical device that manipulates symbols contained on a strip of tape. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. It is believed that if a problem can be solved by an algorithm, there exist a Turing machine that solves the problem. The Turing machine is the most commonly used model in complexity theory to define different complexity class problems such as class P-problems, class NP-problems, and class EXPSPACE-problems etc.

Many types of Turing machines are used to define complexity classes, such as deterministic Turing machines, probabilistic Turing machines, non-deterministic Turing machines, quantum Turing machines, symmetric Turing machines and alternating Turing machines. They are all equally powerful in principle, but when resources (such as time or space) are bounded, some of these may be more powerful than others.

Definition

Turing machine can be thought as a finite state automata connected to an R/W (read/write) head. It has one tape which is divided into a number of squares or cells. The block diagram of the basic model for Turing Machine is given as shown in figure:

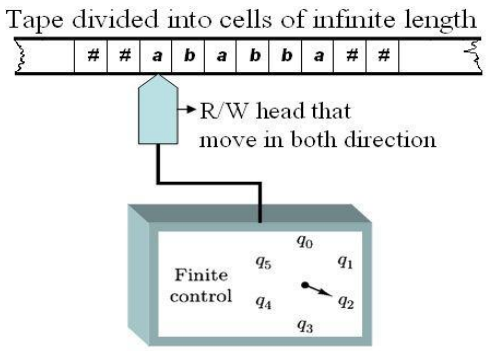


Fig. B.D. of a Turing Machine

- The machine consists of a finite control that can be in any of a finite set of states. The tape is divided into numbers of cells each can holds any one of a finite number of symbols.
- Initially, the input, which is a finite length string of symbols chosen from the input alphabet, is placed on the tape. All other tape cells, extending infinitely to the left & right, initially holds the special symbols called the ***blank*** (#). The blank is a tape symbol but not the input symbol.
- There is a tape head that is always positioned at one of the tape cell where the Turing Machine is said to be scanning that cell. Initially, the tape head is at the leftmost cell that holds the input string.

Mathematically, the Turing Machine, $M = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ can be described by the 7-touple, where

Q = Finite non-empty set of states

Σ = Finite non-empty set of input alphabet or symbols

Γ = Complete set of tape symbols (Σ + #*)

q_0 = Initial states or start states ($q0 \in Q$)

= Blank symbol (# $\in \Gamma$)

F = Subset of Q (i.e. $F \subseteq Q$) consisting of final state (i.e. one or more accepting states).

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

δ = Transition function, which define the move of Turing machine of the form δ(q, x) → (p, y, D).

Here,

q = Current state in Q
x = Tape symbol
p = Next state in Q defined after the current state
y = Symbol in Γ written in the cell being scanned, replacing whatever symbol was there
D = Direction either Left (L) or Right (R) or No-move (N), in which the head move.

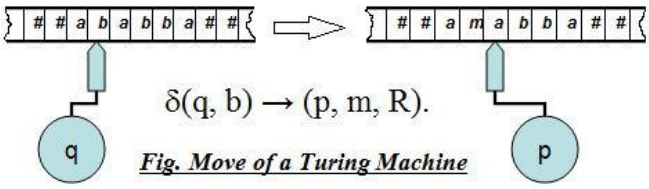


Fig. Move of a Turing Machine

4.2 Representation of Turing Machine

A. Representation of TM by Instantaneous Descriptions (ID)

An ID of an Turing Machine 'M' is a string ***α q b β,*** where 'q' is the present state of M, the entire string is split as ***αβ+b,*** the symbol 'b' is the current symbol under the R/W head, the 'α' represent the left sub-string of the input string & 'β' represent the right sub-string of the input string.

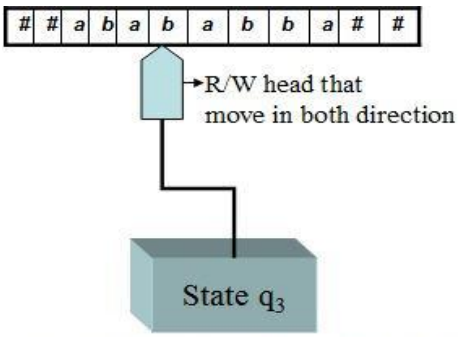


Fig. A Snapshot of Turing Machine

The ID shows the action for one *instant* of time or 'Snapshots' of a Turing Machine. Here, we use the term instant because it explain the state information, direction of move & write facility, thus we cannot use the same terminology for the Finite Automata or PDA.

Here, the present symbol under R/W head is ***'b'*** & the present state is ***'q_3'***. Thus, ***'b'*** must be written to the right of ***'q_3'***. The non-blank symbol to the left of 'b' is ***'aba'*** that must be written to the left of ***'q_3'***. The sequence of non-blank symbols to the right of 'b' is ***'abba'***. Hence, the ID can be shown in the following figure. The $\delta(q_3, b)$ induces a change in ID of the TM. We call this change in ID a ***move***.

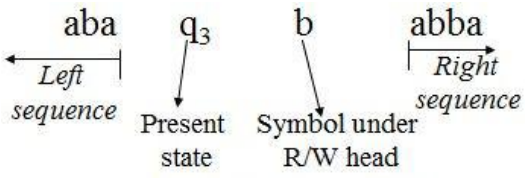


Fig. Representation of ID

- Suppose, $\delta(q, x_i) = (p, y, L)$ is a move of TM, where the input string can be represented as $x_1, x_2 \ldots x_n$. Here, the present symbol under R/W head is 'x_i'.
  Thus,

  ID before processing

  $x_1, x_2 \ldots x_{i-1}$ **q** $x_i, x_{i+1} \ldots x_n$

  ID after processing

  $x_1, x_2 \ldots x_{i-2,}$ **p** x_{i-1} y, $x_{i+1} \ldots x_n$

  This change of ID is represented by

  $x_1, x_2 \ldots x_{i-1}$ **q** $x_i, x_{i+1} \ldots x_n \rightarrow x_1, x_2 \ldots x_{i-2,}$ **p** x_{i-1} y, $x_{i+1} \ldots x_n$

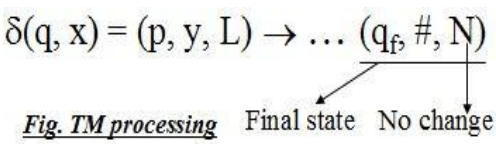


Fig. TM processing

- If $\delta(q, x_i) = (p, y, R)$ then the change of ID is represented by

  $x_1, x_2 \ldots x_{i-1}$ **q** $x_i, x_{i+1} \ldots x_n \rightarrow x_1, x_2 \ldots x_{i-2,} x_{i-1}$ y, **p** $x_{i+1} \ldots x_n$

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

B. Representation of TM by Transition Table

We give the definition of δ in the form of a table called transition table. If δ(q, x) = (p, y, β), it means that 'x' is the current symbol under the R/W head that will replace by 'y' after transition, β gives the movement of the head (L or R or N) and 'p' denotes the new state into which the Turing Machine enters. An example can be seen in figure.

Q/Σ	0	1	#
→q_0	(q_0, 0, R)	(q_1, 0, R)	-
q_1	-	(q_2, 1, R)	(q_0, #, L)
(q_2)	(q_1, 1, R)	(q_2, 0, L)	(q_0, 1, R)

Where, Σ= Tape symbol
Q = Present state

Fig. Transition Table of a TM

C. Representation of TM by Transition Diagram

A transition diagram of a TM is a graphical representation consisting of a finite number of nodes and directed labeled arcs between the nodes. Each node represents a state of the TM and a label on arc from one state to another state represents the information about the required input for transition. The label on arc is generally written as 0 / (1, β) where β gives the movement of the head in L or R or N.

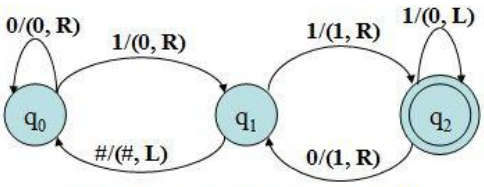


Fig. Transition Diagram for TM

4.3 Design of Turing machine

The Turing machines are designed to play following different roles:

- TM can use as language accepter similar to the role played by the Finite Automata & PDAs.
- TM can be used as computing function. In this role, a TM represents a particular function. The initial input is treated as representing an argument of the function and the final symbol on the tape when the TM enters the halt state treated as representative of the value obtained by an application of the function to the argument represented by the initial string.
- TM can be used as enumerator of the strings of a language that outputs the string of a language, one at a time, in some systematic order that is on a list.
- TM can able to differentiate any language whether it is decidable or un-decidable.
- TM can able to differentiate any problem whether it is in class-P or class-NP.

Basic guidelines for designing a Turing machine

a. The fundamental objective in scanning a symbol by the R/W head is to 'know' what to do in the future. The machine must remember the past symbol scanned. The Turing machine can remember this by going to the next unique state.
b. The number of state must be minimized. This can be achieved by changing the state only when there is a change in the written symbol or when there is a change in moment of the R/W head.

A. Turing machine as Language acceptor

A TM works as a language acceptor for a language if it is able to tell us whether a string 'w' belongs to the language L or not.

Let us consider the Turing machine M = (Q, Σ, Γ, δ, q_0, #, F). A string 'w' in Σ* is said to be accepted by M if $q_0 w \text{-}^* \rightarrow \alpha_1 p \alpha_2$ for some $p \in F$ and $\alpha_1, \alpha_2 \in \Gamma^*$. Here, 'p' is any final state determined after reading the whole input string. The Turing machine 'M' does

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

not accept 'w' if the machine 'M' either halts in a non-accepting state or reach to any non-accepting state after read the whole input string.

There are two ways for acceptance of a string by Turing machine:

a. Acceptance by Final State

Here, the TM must reach to the any one final state after processing the whole input string.

For example

Let a Turing Machine, M = (Q, Σ, Γ, δ, q_0, #, F) can be described by the 7-touple, where

Q = Set of states = {q_0, q_1}
Σ = Set of input alphabet or symbols = {0, 1}
Γ = Set of tape symbols (Σ + #*) = {0, 1, #}
q_0 = Initial states or start states ($q_0 \in Q$)
= Blank symbol (# ∈ Γ)
q_1 = Final state
δ = Transition function, which define by the given transition diagram.
Now, check whether M accept the given string (w) = 00011 or not.

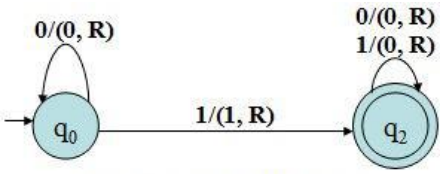


Fig. Transition Diagram

Solution

Here, transition function can be defined as:
$\delta(q_0, 0) \rightarrow (q_0, 0, R)$
$\delta(q_0, 1) \rightarrow (q_1, 1, R)$
$\delta(q_1, 0) \rightarrow (q_1, 0, R)$
$\delta(q_1, 1) \rightarrow (q_1, 1, R)$

Now, the given string (w) = 00011 can be process as:
$q_0 00011 \rightarrow 0q_0 0011 \rightarrow 00q_0 011 \rightarrow 000q_0 11 \rightarrow 0001q_1 1 \rightarrow 00011q_1 \#$ or $\rightarrow 00011q_1$

Since the TM reach to the final state after processing all input string so we say that it is accepted by TM.

b. Acceptance by Halting State

Here, we first declare the list of all the halt states, and then we examine that the resulting state (beside any final state) after processing all the input symbols is whether the defined halting state or not for accepting state. Thus, we can conclude that if the TM goes into the halting state before reading the whole input symbol & further move is not defined then we say that the TM cannot accept the given string. This can be represented as: $(q_0, \#) \rightarrow (h, \#, N)$, where 'h' is any one defined halting state & 'N' is no direction for move.

Example-1: Design a TM that erases all non-blank symbols on the tape.

Solution

Let the required Turing machine, M = (Q, Σ, Γ, δ, q_0, #, F), where
Q = Set of states = {q_0, h}, where h = halting state
Σ = Set of input alphabet or symbols = {a, b}
Γ = Set of tape symbols (Σ + #*) = {a, b, #}
q_0 = Initial states or start states ($q_0 \in Q$)

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

= Blank symbol (# ∈ Γ)
δ = Transition function, which defines as:

$\delta(q_0, a) \rightarrow (q_0, \#, R)$
$\delta(q_0, b) \rightarrow (q_1, \#, R)$
$\delta(q_0, \#) \rightarrow (h, \#, N)$
$\delta(h, \#) \rightarrow$ Accepting state

Present state	***Tape symbol***		
	a	***b***	***#***
$\rightarrow q_0$	$(q_0, \#, R)$	$(q_0, \#, R)$	$(h, \#, N)$
h			Accepting state

Now, let any string (w) = abbbaa, which can be process as:
q_0 abbbaa $\rightarrow$ $\#q_0$bbbaa $\rightarrow$ $\#\#0q_0$bbaa $\rightarrow$ $\#\#\#q_0$baa $\rightarrow$ $\#\#\#\#q_1$aa $\rightarrow$ $\#\#\#\#\#q_1$a $\rightarrow$ $\#\#\#\#\#\#q_1\# \rightarrow$ ######h

Since the TM reach to the defined halting state after processing all input string so we say that it is accepted by TM. Hence, the defined TM is capable for handling our problem.

Present state	***Tape symbol***	
	a	***b***
$\rightarrow q_0$	$(q_1, \#, R)$	$(q_1, \#, R)$
$*q_0$	$(q_1, \#, R)$	$(q_1, \#, R)$

Note: This problem also can be treated by defining the final state rather than the halting state as:

Example-2: Design a TM that recognizes the language of all the string of even length over the alphabet {a, b}.

Solution

Let the required Turing machine, M = (Q, Σ, Γ, δ, q_0, #, F), where
Q = Set of states = {q_0, q_1, h}, where h = halting state
Σ = Set of input alphabet or symbols = {a, b}
Γ = Set of tape symbols (Σ + #*) = {a, b, #}
q_0 = Initial states or start states ($q_0 \in Q$)
= Blank symbol (# ∈ Γ)
δ = Transition function, which defines as:

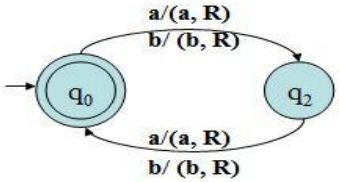


Fig. Transition Diagram

$\delta(q_0, a) \rightarrow (q_1, a, R)$
$\delta(q_0, b) \rightarrow (q_1, b, R)$
$\delta(q_1, a) \rightarrow (q_0, a, R)$
$\delta(q_1, b) \rightarrow (q_0, b, R)$
$\delta(q_0, \#) \rightarrow (h, \#, N)$
$\delta(h, \#) \rightarrow$ Accepting state

Present state	***Tape symbol***		
	a	***b***	***#***
$\rightarrow q_0$	(q_1, a, R)	(q_1, b, R)	$(h, \#, N)$
q_1	(q_0, a, R)	(q_0, b, R)	
h			Accepting state

Now, let any string (w) = abbbaa, which can be process as:
q_0 abbbaa $\rightarrow$ aq_1bbbaa $\rightarrow$ abq_0bbaa $\rightarrow$ $abbq_1$baa $\rightarrow$ $abbbq_0$aa $\rightarrow$ $abbbaq_1$a $\rightarrow$ $abbbaaq_0\# \rightarrow$ ######h

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

Since the TM reach to the defined halting state after processing all input string so we say that it is accepted by TM. Hence, the defined TM is capable for handling our problem.

B. Turing machine for computing function

The Turing machine can be used to compute some functions that are called the *Turing computable function.*

Let Σ_0, Σ_1 be any alphabets not containing the blank symbol #. Also, let 'f' be a function from Σ_0^* to Σ_1^*. A Turing machine 'M' is said to compute 'f' if Σ_0, $\Sigma_1 \subseteq \Sigma$ and for any string $w \in \Sigma_0^*$ if f (w) = u then

$$(S, \# w \underline{\#}) \xrightarrow[M]{*} (h, \# u \underline{\#})$$

Where, h is the halting state.

If some such Turing machine 'M' exists, then the function 'f' is said to be a Turing computable function.

Note:

a. Here, the TM must be halt when it find the resulting functional value during processing from the start state's'.
b. Σ is the input symbol for the defined TM.
c. $\subseteq$ is proper subset which indicates subset or same set
d. U is any resulting value of function 'f'
e. Function 'f' from Σ_0^* to Σ_1^* shows that the both set of input symbols are fall on the same domain i.e. real number to real number etc.
f. The # shows the current position of the reading head.

Example-1: Design a Turing machine which compute the function f (n) = n+1, for $n \in N$ (natural number).

Solution

Here, we design a Turing machine 'M' which computes any function 'f' by writing an 'I' in the tape square in which its head is initially located, moving its head one square right and halting.

Formally, let the required Turing machine, M = (Q, Σ, Γ, δ, q_0, #, F), where
Q = Set of states = $\{q_0, h\}$, where h = halting state
Σ = Set of input alphabet or symbols = {I}
Γ = Set of tape symbols (Σ + #*) = {I, #}
q_0 = Initial states or start states ($q_0 \in Q$)
= Blank symbol (# $\in \Gamma$)
δ = Transition function, which defines as:

$\delta(q_0, \#) \rightarrow (h, I, R)$
OR
$\delta(q_0, \#) \rightarrow (q_0, I, N)$
$\delta(q_0, I) \rightarrow (h, \#, R)$
$\delta(h, \#) \rightarrow$ Accepting state

Present state	***Tape symbol***	
	I	***#***
$\rightarrow q_0$	(h, #, R)	(q_0, I, N)
h		Accepting state

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

In case of $\delta(q_0, \#) \rightarrow (h, I, R)$

For n=0
- $\delta(q_0, \underline{\#}\#) \rightarrow (h, I\underline{\#})$ i.e. Accepted

For n=1
- $\delta(q_0, \#I\underline{\#}\#) \rightarrow (h, \#II\underline{\#})$ i.e. Accepted

For n=2
- $\delta(q_0, \#II\underline{\#}\#) \rightarrow (h, \#III\underline{\#})$ i.e. Accepted

Hence, in similar manner we can conclude that
For $n=I^n$
- $\delta(q_0, \# I^n\underline{\#}\#) \rightarrow (h, \# I^{n+1}\underline{\#})$ i.e. Accepted

In case second

For n=0
- $\delta(q_0, \underline{\#}\#) \rightarrow (q_0, \underline{I}\#) \rightarrow (h, I\underline{\#})$ i.e. Accepted

For n=1
- $\delta(q_0, I\underline{\#}\#) \rightarrow (q_0, I\underline{I}\#) \rightarrow (h, II\underline{\#})$ i.e. Accepted

For n=2
- $\delta(q_0, II\underline{\#}\#) \rightarrow (q_0, III\#) \rightarrow (h, III\underline{\#})$ i.e. Accepted

Hence, in similar manner we can conclude that
For n=0
- $\delta(q_0, I^n\underline{\#}\#) \rightarrow (q_0, I^nI\#)$ or $(h, I^{n+1}\underline{\#})$ i.e. Accepted

Hence, the Turing machine 'M' computes the given function.

Example-2: Design a Turing machine which compute the function f (n) = n+2, for n∈N (natural number).

Solution

Here, we design a Turing machine 'M' which computes any function 'f' by writing an 'I' in the tape square in which its head is initially located, moving its head one square right and halting.

Formally, let the required Turing machine, M = (Q, Σ, Γ, δ, q_0, #, F), where
Q = Set of states = {q_0, h}, where h = halting state
Σ = Set of input alphabet or symbols = {I}
Γ = Set of tape symbols (Σ + #*) = {I, #}
q_0 = Initial states or start states ($q_0 \in Q$)
= Blank symbol (# ∈ Γ)
δ = Transition function, which defines as:
$\delta(q_0, \#) \rightarrow (q_1, I, R)$
$\delta(q_1, \#) \rightarrow (h, I, R)$
$\delta (h, \#) \rightarrow$ Accepting state

Present state	***Tape symbol***	
	I	***#***
$\rightarrow q_0$	(q_1, I, R)	
q_1	(h, I, R)	
h		Accepting state

Here,

For n=0
- $\delta(q_0, \underline{\#}\#\#) \rightarrow (q_1, I\underline{\#}\#) \rightarrow (h, II\underline{\#})$ i.e. Accepted

For n=1
- $\delta(q_0, I\underline{\#}\#\#) \rightarrow (q_1, II\underline{\#}\#) \rightarrow (h, III\underline{\#})$ i.e. Accepted

For n=2
- $\delta(q_0, II\underline{\#}\#\#\#) \rightarrow (q_1, III\underline{\#}\#) \rightarrow (h, IIII\underline{\#})$ i.e. Accepted

Hence, in similar manner we can conclude that
For n=0
- $\delta(q_0, I^n\underline{\#}\#\#) \rightarrow (q_1, I^nI\underline{\#}\#) \rightarrow (h, I^nII\underline{\#})$ or $(h, I^{n+2}\underline{\#})$ i.e. Accepted

Hence, the Turing machine 'M' computes the given function.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

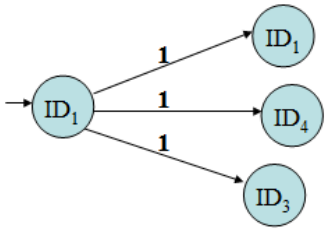
4.4 Deterministic & Non-deterministic Turing machine

Here, we define the deterministic & non-deterministic approach of the TM on the basis of the *Instantaneous Descriptions (ID)*.

- In case of the Deterministic Turing machine (DTM), the processing of an input symbol results in the transition to a unique ID, so the processing of an input string results in a chain of ID's such as: $\mathbf{ID_1 \rightarrow ID_5 \rightarrow ID_3 \rightarrow \ldots ID_{n-1} \rightarrow ID_n}$

  The transition function $\mathbf{\delta(q_0, a)}$ for the DTM is defined for some elements of set of states (Q) x Set of tape symbols (Γ).

- In case of the Non-deterministic Turing machine (NTM), the processing of an input symbol result in the transition to several IDs.



  The transition function $\mathbf{\delta(q_0, a)}$ for the NTM is defined for some elements of set of states (Q) x Set of tape symbols (Γ) x set of direction of moves {L, R}.

  i.e. $\mathbf{\delta(q_0, a) \rightarrow Q \times \Gamma \times \{L, R\}}$

4.5 Universal Turing Machine

Designing of TM is complex task. We must design a machine that accept two inputs i.e. the input data and a description of algorithm (computation). This is precisely a general purpose computer does. It accepts a data and a program. A general purpose TM is usually called universal TM which is powerful enough to simulate the behavior of any computer, including a TM itself. More precisely, UTM can simulate the behavior of an arbitrary TM over ∑.

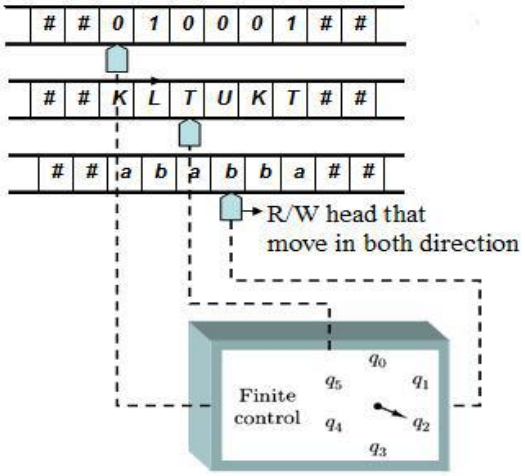


Fig. B.D. of a Turing Machine

Extension of Turing machine

- There may be several tapes instead of only one tape, each having its own dependent need.
- The tape may be allowed to be infinite in both directions.
- There may be more than one head scanning various cells of the tape. Two or more heads simultaneously read the same cell or may attempt to write in the same cell.

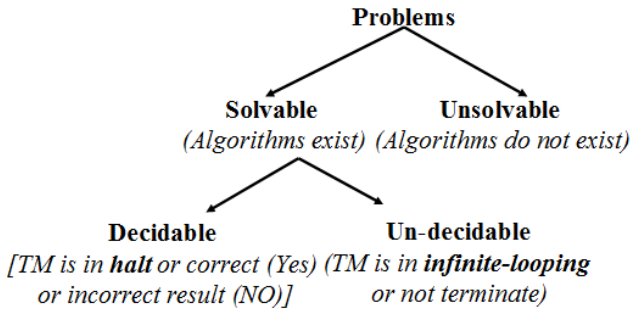
4.6 Church's Thesis

Turing machine can be carry out any computation that can be carried out by similar type of automata because these automata seems to capture the essential features of real computing machines, we take TM to be a precise formal equivalent of the intuitive notion of algorithm. Nothing will be considered an algorithm if it cannot be rendered as a TM. The principle that "TMs are formal version of algorithm and that no computational procedure will be considered an algorithm unless it can be represented as a TM" is known as Church's thesis. It is a thesis, not a theorem because it is not a mathematical result.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

4.7 Types of Problems

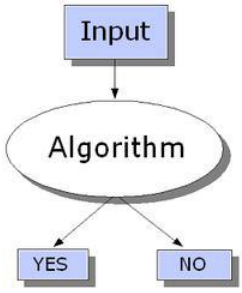
If we compare the language as a problem then it indicates all the solvable problems that include both the decidable & un-decidable problems.



A. Decision problem

In computability theory and computational complexity theory, a **decision problem** is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters. For example, the problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of *x* and *y*.

Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'. A corresponding function problem is "given two numbers *x* and *y*, what is *x* divided by *y*?". They are also related to optimization problems, which are concerned with finding the *best* answer to a particular problem.



A method for solving a decision problem given in the form of an algorithm is called a **decision procedure** for that problem. A decision problem which can be solved by an algorithm, such as this example, is called **decidable**.

B. Function problem

In computational complexity theory, a **function problem** is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just YES or NO. Notable examples include the travelling salesman problem, which asks for the route taken by the salesman, and the integer factorization problem, which asks for the list of factors.

Function problems can be sorted into complexity classes in the same way as decision problems. For example FP is the set of function problems which can be solved by a deterministic Turing machine in polynomial time, and FNP is the set of function problems which can be solved by a non-deterministic Turing machine in polynomial time.

C. Search problem

In computational complexity theory and computability theory, a **search problem** is a type of computational problem represented by a binary relation. We need the search problem to select multiple solution from the given large data items but in case of function problem we only select single result of data.

For instance, such problems occurs very frequently in graph theory, where searching graphs for structures such as particular matching, cliques, independent set, etc. are subjects of interest.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

4.8 Recursive Language

A language 'L' is recursive language if there exists some Turing machine 'M' that satisfy the following two conditions:

- If $w \in L$, then M accept 'w' *(i.e. reaches on halting state on processing w)* and halts.
- If $w \notin L$, then M eventually halts, without reaching an accepting state.

A TM of this type corresponds to the notation of an algorithm, a well defined sequence of stapes that always finishes and produces an answer. If we think of the language 'L' as a problem, then the problem 'L' is called **decidable** *(if it is a recursive language)* and **un-decidable** *(if it is not a recursive language).*

Recursively Enumerable Language

The recursively enumerable language indicates the set of language that are accepted by using a Turing machine. The terms recursively enumerable means that a function could list all the member of a language in some order.

4.9 Recursive function Theory

A. Partial function

A partial function **f** from X to Y is a rule which assigns to every element of X almost one element of Y. For example, if R denotes the set of all real numbers, then $f(r) = +\sqrt{r}$ is a partial function since f(r) is not defined as a real number when r is negative.

B. Total function

A total function **f** from X to Y is a rule which assigns to every element of X, a unique element of Y. For example, if R denotes the set of all real numbers, then f(r) = 2r is a total function since f(r) is defined for both positive and negative.

C. Primitive Recursive function

The set of primitive function is obtained by three types of initial functions and three structuring rules for constructing more complex function from already constructed functions.

- The three type of initial functions are:
  - **Zero function** defined by Z(x) = 0 e.g. Z(7) = 0
  - **Successor function** defined by S(x) = x + 1, e.g. S(7) = 5
  - **Projection function** defined by $U_i^n(x_1, \ldots x_n) = x$

- The three type of structuring rules are:
  - **Composition**: For example if f_1, f_2 and f_3 are partial functions of two variables and g is a partial function of three variable, then the composition of g with f_1, f_2 and f_3 is given by: g $\{f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2)\}$
  - **Recursion**: It is the process of repeating items in a self-similar way. It is related to mathematical induction. For e.g. Fibonacci sequence: $F_n = F_{n-1} + F_{n-2}$, initial condition $F_0=0$, $F_1= 1$, the factorial function: n! etc.
  - **μ –recursive function**: It is a partial function that can be constructed from the initial functions by a finite number of applications of the compositions, primitive recursions and minimization to regular functions.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

CHAPTER - 5

Computational Complexity Theory

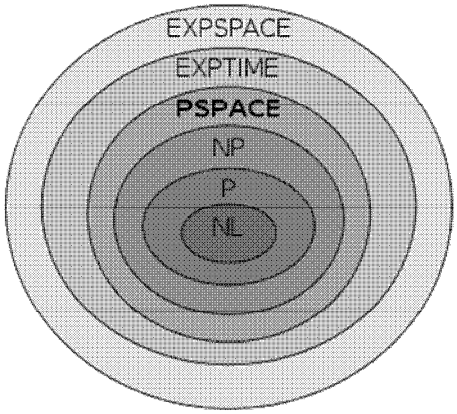
5.1 Introduction

Complexity theory considers not only whether a problem can be solved at all on a computer (as computability theory), but also how efficiently the problem can be solved. Thus, the complexity theory classifies problems according to their degree of "difficulty". Two major aspects are considered: time complexity and space complexity, which are respectively how many steps does it take to perform a computation, and how much memory is required to perform that computation. Although the time and space complexity are dependent of input problem but it is more complexity issue, so computer scientists have adopted Big O notation that compare the asymptotic behavior of large problems.

Complexity classes

Complexity class	Model of computation	Resource constraint
P	Deterministic Turing machine	Time poly(n)
EXPTIME	Deterministic Turing machine	Time $2^{poly(n)}$
NP	Non-deterministic Turing machine	Time poly(n)
NEXPTIME	Non-deterministic Turing machine	Time $2^{poly(n)}$
L	Deterministic Turing machine	Space O(log n)
PSPACE	Deterministic Turing machine	Space poly(n)
EXPSPACE	Deterministic Turing machine	Space $2^{poly(n)}$
NL	Non-deterministic Turing machine	Space O(log n)
NPSPACE	Non-deterministic Turing machine	Space poly(n)
NEXPSPACE	Non-deterministic Turing machine	Space $2^{poly(n)}$

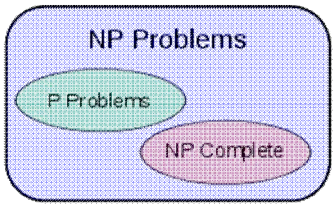
5.2 Class P and NP problems



a. **P = Polynomial** (*Problems that solve by deterministic algorithm i.e. by TM that halts*)
b. **NP = Non-deterministic polynomial** (*Problems that solve by non-deterministic algorithm i.e. by NTM*)

- The ***time complexity,*** **T(n)** of Turing machine (M) is the maximum moves made by M in passing any input string of length 'n' i.e. *M halts after making at most T(n) moves.*
- For defining class P and class NP, we require the time complexity of a Turing machine.
- A language L is said to be in class P if there exists a deterministic Turing machine M such that M is of time complexity P(n) for some polynomial P and M accept L. E.g. Kruskal's Algorithm
- A language L is said to be in class NP if there exists a non-deterministic Turing machine M such that M is of time complexity P(n) for some polynomial P and M accept L. E.g. Traveling sales man problem

P versus NP problem



- The complexity class P is often seen as a mathematical abstraction modeling those computational tasks that admit an efficient algorithm.
- The complexity class NP, on the other hand, contains many problems that people would like to solve efficiently, but for

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)

which no efficient algorithm is known. Diagram of complexity classes provided that $P \neq NP$.

- A deterministic machine, at each point in time, executes an instruction. Depending on the outcome of executing the instruction, it then executes some next instruction, which is unique. A non-deterministic machine on the other hand has a choice of next steps. It is free to choose any that it wishes. For example, it can always choose a next step that leads to the best solution for the problem. A non-deterministic machine thus has the power of extremely good, optimal guessing.

The **P** versus **NP** problem is to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time. To define the problem precisely it is necessary to give a formal model of a computer. The standard computer model in computability theory is the Turing machine, introduced by Alan Turing in 1936. Although the model was introduced before physical computers were built, it nevertheless continues to be accepted as the proper computer model for the purpose of defining the notion of *computable function*.

Informally the class **P** is the class of decision problems solvable by some algorithm within a number of steps bounded by some fixed polynomial in the length of the input. Turing was not concerned with the efficiency of his machines, but rather his concern was whether they can simulate arbitrary algorithms given sufficient time. However it turns out Turing machines can generally simulate more efficient computer models (for example machines equipped with many tapes or an unbounded random access memory) by at most squaring or cubing the computation time. Thus **P** is a robust class, and has equivalent definitions over a large class of computer models.

NP-completeness

NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; *NP* may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine.

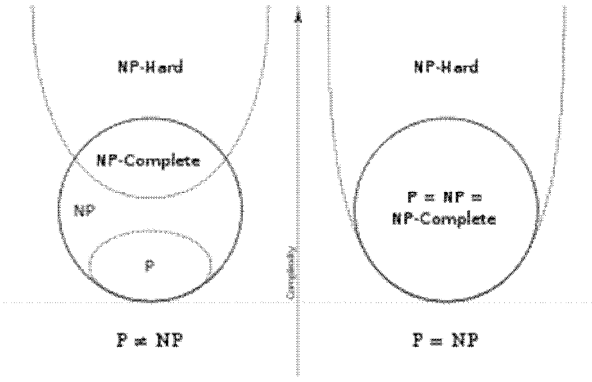
A decision problem C is NP-complete if:

1. C is in NP, and
2. Every problem in NP is reducible to C in polynomial time.

NP-hard problems

Some problem that proves the condition – (2) of NP-completeness problem but not the condition – (1) is called the NP-hard problem. They are intractable problems.

Euler diagram for P, NP, NP-complete, and NP-hard set of problems can be seen here.



Note: Here we define the language as problem in case of all above complexity class problems.

Print to PDF without this message by purchasing novaPDF (http://www.novapdf.com/)