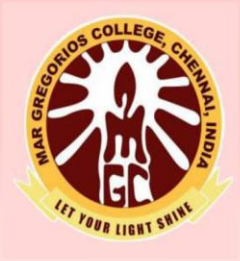
MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution



PG DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: THEORETICAL FOUNDATIONS OF COMPUTER SCIENCE

SUBJECT CODE: PED1A

SEMESTER: I

PREPARED BY: PROF. S.JAMES BENEDICT FELIX

THEORETICAL FOUNDATIONS OF COMPUTER SCIENCE

Unit – I

1.Propositions and Compound Propositions

A proposition (or statement) is a declarative statement which is true or false, but not both. Consider, for example, the following six sentences. .

- Ice floats in water.
- China is in Europe.
- 2 + 2 = 4
- 2 + 2 = 5
- Where are you going?
- Do your homework.

The first four are propositions, the last two are not. Also, (i) and (iii) are true, but (ii) and (iv) are false.

Compound Propositions

Many propositions are *composite*, that is, composed of *sub propositions* and various connectives discussed subsequently. Such composite propositions are called *compound propositions*. Aproposition is said to be *primitive* if it cannot be broken down into simpler propositions, that is, if it is not composite.

For example, the above propositions (i) through (iv) are primitive propositions. On the other hand, the following two propositions are composite:

"Roses are red and violets are blue." and "John is smart or he studies every night."

2. Logical Operations

The three basic logical operations of conjunction, disjunction, and negation which correspond, respectively, to the English words "and," "or," and "not."

Conjunction, p ∧q

Any two propositions can be combined by the word "and" to form a compound proposition called the *conjunction* of the original propositions. Symbolically,

p ∧q

read "*p* and *q*," denotes the conjunction of *p* and *q*. Since p ∧q is a proposition it has a truth value, and this truth value depends only on the truth values of *p* and *q*. Specifically:

If *p* and *q* are true, then p ∧q is true; otherwise p ∧q is false.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

(*a*) "*p* and *q*"

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

(*b*) "*p* or *q*"

p	$\neg p$
T	F
F	T

(*c*) "not *p*"

Disjunction, p ∨q

Any two propositions can be combined by the word "or" to form a compound proposition called the *disjunction* of the original propositions. Symbolically,

p ∨q

read "*p* or *q*," denotes the disjunction of *p* and *q*. The truth value of p ∨q depends only on the truth values of *p* and *q* as follows.

If *p* and *q* are false, then p ∨q is false; otherwise p ∨q is true.

Negation, ¬p

Given any proposition *p*, another proposition, called the *negation* of *p*, can be formed by writing "It is not true that ..." or "It is false that ..." before *p* or, if possible, by inserting in *p* the word "not." Symbolically, the negation of *p*, read "not *p*," is denoted by

¬p

The truth value of ¬p depends on the truth value of *p* as follows:

If *p* is true, then ¬p is false; and if *p* is false, then ¬p is true.

3. Truth Tables

Let P(p,q,...) denote an expression constructed from logical variables p,q,..., which take on the valueTRUE (T) or FALSE (F), and the logical connectives ∧, ∨, and ¬ (and others discussed subsequently). Such an expression P(p,q,...) will be called a *proposition*.

The main property of a proposition P(p,q,...) is that its truth value depends exclusively upon the truth values of its variables, that is, the truth value of a proposition is known once the truth value of each of its variables is known. A simple concise way to show this relationship is through a *truth table*. We describe a way to obtain such a truth table below.

Consider, for example, the proposition ¬(p ∧¬q). Figure 4-2(*a*) indicates how the truth table of ¬(p ∧¬q) is constructed

p	q	$\neg q$	$p \wedge \neg q$	$\neg(p \wedge \neg q)$
T	T	F	F	T
T	F	T	T	F
F	T	F	F	T
F	F	T	F	T

(*a*)

p	q	$\neg(p \wedge \neg q)$
T	T	T
T	F	F
F	T	T
F	F	T

(*b*)

4. Tautologies and Contradictions

Some propositions P(p,q,...) contain only *T* in the last column of their truth tables or, in other words, they are true for any truth values of their variables. Such propositions are called *tautologies*.Analogously, a proposition P(p,q,...) is called a *contradiction* if it contains only *F* in the last column of its truth table or, in other words, if it is false for any truth values of its variables. For example, the proposition "*p* or not *p*," that is, p ∨ ¬p, is a tautology, and the proposition "*p* and not *p*," that is, p∧¬p, is a contradiction. This is verified by looking at their truth tables in Fig. 4-5. (The truth tables have only two rows since each proposition has only the one variable *p*.)

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

(*a*) $p \vee \neg p$

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

(*b*) $p \wedge \neg p$

5. Logical Equivalence

Two propositions P(p,q,...) and Q(p,q,...) are said to be *logically equivalent*, or simply *equivalent* or *equal*, denoted by

$$P(p, q, ...) \equiv Q(p, q, ...)$$

if they have identical truth tables. Consider, for example, the truth tables of ¬(p ∧q) and ¬p ∨¬q appearing in Fig. 4-6. Observe that both truth tables are the same, that is, both propositions are false in the first case and true in the other three cases. Accordingly, we can write

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

Let *p* be "Roses are red" and *q* be "Violets are blue." Let *S*be the statement:

"It is not true that roses are red and violets are blue."

Then *S* can be written in the form ¬(p ∧q). However, as noted above, ¬(p ∧q) ≡ ¬p ∨ ¬q. Accordingly, *S* has the same meaning as the statement:

"Roses are not red, or violets are not blue."

p	q	$p \wedge q$	$\neg(p \wedge q)$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

(*a*) $\neg(p \wedge q)$

p	q	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T	F	F	F
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

(*b*) $\neg p \vee \neg q$

6. Algebra of Propositions

Propositions satisfy various laws which are listed in Table 4-1. (In this table, *T* and *F* are restricted to the truth values "True" and "False," respectively.) We state this result formally

Table1. Laws of the algebra of propositions

Idempotent laws:	(1a) p ∨p ≡ p	(1b) p ∧p ≡ p
Associative laws:	(2a) (p ∨q) ∨r ≡ p ∨(q ∨r)	(2b) (p ∧q) ∧r ≡ p ∧(q ∧r)
Commutative laws:	(3a) p ∨q ≡ q ∨p	(3b) p ∧q ≡ q ∧p
Distributive laws:	(4a) p ∨(q ∧r) ≡ (p ∨q) ∧(p ∨r)	(4b) p ∧(q ∨r) ≡ (p ∧q) ∨(p ∧r)
Identity laws:	(5a) p ∨F ≡ p (6a) p ∨T ≡ T	(5b) p ∧T ≡ p (6b) p ∧F ≡ F
Involution law:	(7) ¬¬p ≡ p	
Complement laws:	(8a) p ∨ ¬p ≡ T (9a) ¬T ≡ F	(8b) p ∧ ¬p ≡ T (9b) ¬F ≡ T
DeMorgan's laws:	(10a) (p q) p q	(10b) (p q) p q

7. Conditional and Biconditional Statements

Many statements, particularly in mathematics, are of the form "If p then q." Such statements are called *conditional* statements and are denoted by

$$p \rightarrow q$$

The conditional $p \rightarrow q$ is frequently read "p implies q" or "p only if q."

Another common statement is of the form "p if and only if q." Such statements are called *biconditional*statements and are denoted by

$$p \leftrightarrow q$$

The truth values of $p \rightarrow q$ and $p \leftrightarrow q$ are defined by the tables in Fig. 4-7(a) and (b). Observe that:

(a) The conditional $p \rightarrow q$ is false only when the first part p is true and the second part q is false.Accordingly, when p is false, the conditional $p \rightarrow q$ is true regardless of the truth value of q.

(b) The biconditional $p \leftrightarrow q$ is true whenever p and q have the same truth values and false otherwise.

The truth table of $\neg p \wedge q$ appears in Fig. 4-7(c). Note that the truth table of $\neg p \vee q$ and $p \rightarrow q$ are identical, that is, they are both false only in the second case. Accordingly, $p \rightarrow q$ is logically equivalent to $\neg p \vee q$; that is,

$$p \rightarrow q \equiv \neg p \vee q$$

In other words, the conditional statement "If p then q" is logically equivalent to the statement "Not p or q" which only involves the connectives $\vee$and $\neg$ and thus was already a part of our language. We may regard $p \rightarrow q$ as an abbreviation for an oft-recurring statement.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

(a) $p \rightarrow q$

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

(b) $p \leftrightarrow q$

p	q	$\neg p$	$\neg p \vee q$
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

(c) $\neg p \vee q$

8. Arguments

An *argument* is an assertion that a given set of propositions $P_1, P_2, ..., P_n$, called *premises*, yields (has a consequence) another proposition Q, called the *conclusion*. Such an argument is denoted by

$$P_1, P_2, ..., P_n \$ Q$$

An argument $P_1, P_2, ..., P_n\$ Q$ is said to be *valid* if *Q* is true whenever all the premises P1,P2,...,$P_n$are true.An argument which is not valid is called *fallacy*.

9. Quantifiers

Universal Quantifier

Let p(x) be a propositional function defined on a set *A*. Consider the expression

$$(\forall x \in A)p(x) \quad \text{or} \quad \forall x\, p(x)$$

which reads "For every *x* in *A*, p(x) is a true statement" or, simply, "For all *x*, p(x)."

The symbol ∀which reads "for all" or "for every" is called the *universal quantifier*. The statement (*4.1*) is equivalent to the statement

$$T_p = \{x \mid x \in A, p(x)\} = A$$

that is, that the truth set of p(x) is the entire set *A*.

The expression p(x) by itself is an open sentence or condition and therefore has no truth value. However, ∀x p(x), that is p(x) preceded by the quantifier ∀, does have a truth value which follows from the equivalence of (*4.1*) and (*4.2*). Specifically:

Q_1: If $\{x \mid x \in A, p(x)\} = A$ then ∀x p(x) is true; otherwise, ∀x p(x) is false.

Existential Quantifier

Let p(x) be a propositional function defined on a set *A*. Consider the expression

$$(\exists x \in A)p(x) \quad \text{or} \quad \exists x, p(x)$$

which reads "There exists an *x* in *A* such that p(x) is a true statement" or, simply, "For some *x*, p(x)." The symbol ∃
which reads "there exists" or "for some" or "for at least one" is called the *existential quantifier*. Statement (*4.3*) is equivalent to the statement

$$T_p = \{x \mid x \in A, p(x)\} = \qquad (4.4)$$

i.e., that the truth set of p(x) is not empty. Accordingly, ∃x p(x), that is, p(x) preceded by the quantifier ∃, does have a truth value. Specifically:

Q_2: If $\{x \mid p(x)\} =$ then∃x p(x) is true; otherwise, ∃x p(x) is false.

10. Negation of Quantified Statements

Consider the statement: "All math majors are male." Its negation reads:

"It is not the case that all math majors are male" or, equivalently, "There exists at least one math major who is a female (not male)"

Symbolically, using *M* to denote the set of math majors, the above can be written as

$$\neg(\forall x \in M)(x \text{ is male}) \equiv (\exists x \in M)\ (x \text{ is not male})$$

or, when p(x) denotes "*x* is male,"

$$\neg(\forall x \in M)p(x) \equiv (\exists x \in M)\neg p(x) \text{ or} \neg\forall x p(x) \equiv \exists x \neg p(x)$$

The above is true for any proposition p(x).

(DeMorgan): $\neg(\forall x \in A)p(x) \equiv (\exists x \in A)\neg p(x)$.

In other words, the following two statements are equivalent:
(1) It is not true that, for all a ∈A, p(a) is true. (2) There exists an a ∈A such that p(a) is false.

There is an analogous theorem for the negation of a proposition which contains the existential quantifier.

(DeMorgan): $\neg(\exists x \in A)p(x) \equiv (\forall x \in A)\neg p(x)$.

That is, the following two statements are equivalent:
(1) It is not true that for some a ∈A, p(a) is true. (2) For all a ∈A, p(a) is false.

11. Basic Counting Principles

There are two basic counting principles used throughout this chapter. The first one involves addition and the second one multiplication.

Sum Rule Principle:

Suppose some event *E* can occur in *m* ways and a second event *F* can occur in *n* ways, and suppose both events cannot occur simultaneously. Then *E* or *F* can occur in $m + n$ ways.

Product Rule Principle:

Suppose there is an event *E* which can occur in *m* ways and, independent of this event, there is a second event *F* which can occur in *n* ways. Then combinations of *E* and *F* can occur in mn ways.

The above principles can be extended to three or more events. That is, suppose an event E_1 can occur in n_1 ways, a second event E_2 can occur in n_2 ways, and, following E_2; a third event E_3 can occur in n_3 ways, and so on. Then:

Sum Rule:

If no two events can occur at the same time, then one of the events can occur in:

$$n1 + n2 + n3 + \cdots \text{ ways.}$$

Product Rule:

If the events occur one after the other, then all the events can occur in the order indicated in:

$$n_1 \cdot n_2 \cdot n_3 \cdot \ldots \text{ ways.}$$

12. Factorial

The product of the positive integers from 1 to *n*inclusive is denoted by *n*!, read "*n* factorial." Namely:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2)(n-1)n = n(n-1)(n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Accordingly, 1! = 1 and n! = n(n − l)!. It is also convenient to define 0! = 1.

13. Binomial Coefficient

The symbol $\binom{n}{r}$, read "*nCr*" or "*n* Choose *r*," where *r* and *n* are positive integers with $r \leq n$, is defined asfollows:

$$\binom{n}{r} = \frac{n(n-1)\cdots(n-r+1)}{r(r-1)\ldots 3\cdot 2\cdot 1} \quad \text{or equivalently} \quad \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Note that n − (n − r) = r. This yields the following important relation.

$$\binom{n}{n-r} = \binom{n}{r} \text{ or equivalently, } \binom{n}{a} = \binom{n}{b}$$

where $a + b = n$.Motivated by that fact that we defined 0! = 1, we define:

$$\binom{n}{0} = \frac{n!}{0!n!} = 1 \quad \text{and} \quad \binom{0}{0} = \frac{0!}{0!\,0!} = 1$$

14. Permutation

Any arrangement of a set of *n* objects in a given order is called a *permutation* of the object (taken all at a time).

Any arrangement of any r ≤ n of these objects in a given order is called an "*r*-permutation" or "a permutation of the *n* objects taken *r* at a time." Consider, for example, the set of letters *A*, *B*, *C*, *D*. Then:

(i) *BDCA*, *DCBA*, and *ACDB* are permutations of the four letters (taken all at a time).

(ii) *BAD*, *ACB*, *DBC* are permutations of the four letters taken three at a time.

(iii) *AD*, *BC*, *CA* are permutations of the four letters taken two at a time.

We usually are interested in the number of such permutations without listing them. The number of permutations of *n* objects taken *r* at a time will be denoted by

P(n,r) (other texts may use ${}_nP_r$, $P_{n,r}$, or $(n)_r$).

The following theorem applies.

$$P(n, r) = n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!}$$

We emphasize that there are *r* factors in n(n − 1)(n − 2)···(n − r + 1).

15. Combinations

Let *S*be a set with *n* elements. A *combination* of these *n* elements taken *r* at a time is any selection of *r* of the elements where order does not count. Such a selection is called an *r-combination*; it is simply a subset of *S* with *r* elements. The number of such combinations will be denoted by

C(n,r) (other texts may use ${}_nC_r$, $C_{n,r}$, or C_r^n).

Before we give the general formula for C(n, r), we consider a special case.

16. Pigeonhole Principle

Many results in combinational theory come from the following almost obvious statement.

Pigeonhole Principle:

If *n* pigeonholes are occupied by n + 1 or more pigeons, then at least one pigeonhole is occupied by more than one pigeon.

This principle can be applied to many problems where we want to show that a given situation can occur.

EXAMPLE

Suppose a department contains 13 professors, then two of the professors (pigeons) were born in the same month (pigeonholes).

(a) Find the minimum number of elements that one needs to take from the set S = {1, 2, 3,..., 9} to be sure that two of the numbers add up to 10.
Here the pigeonholes are the five sets {1, 9}, {2, 8}, {3, 7}, {4, 6}, {5}. Thus any choice of six elements (pigeons) of *S* will guarantee that two of the numbers add up to ten.

The Pigeonhole Principle is generalized as follows.

Generalized Pigeonhole Principle:

If *n* pigeonholes are occupied by *kn*+ 1 or more pigeons, where *k* is a positive integer, then at least one pigeonhole is occupied by k + 1 or more pigeons.

17. Ordered and Unordered Partitions.

Suppose a set has 7 elements. We want to find the number *m* of ordered partitions of *S* into three cells, say $[A_1, A_2, A_3]$, so they contain 2, 3, and 2 elements, respectively.

Since *S* has 7 elements, there are *C*(7, 2) ways of choosing the first two elements for A_1. Following this, there are *C*(5, 3) ways of choosing the 3 elements for A_2. Lastly, there are *C*(2, 2) ways of choosing the 2 elements for A_3 (or, the last 2 elements form the cell A_3). Thus:

$$m = C(7,2)C(5,3)C(2,2) = \binom{7}{2}\binom{5}{3}\binom{2}{2} = \frac{7\cdot 6}{2\cdot 1}\cdot\frac{5\cdot 4\cdot 3}{3\cdot 2\cdot 1}\cdot\frac{2\cdot 1}{2\cdot 1} = 210$$

Observe that

$$m = \binom{7}{2}\binom{5}{3}\binom{2}{2} = \frac{7!}{2!5!}\cdot\frac{5!}{3!2!}\cdot\frac{2!}{2!0!} = \frac{7!}{2!3!2!}$$

since each numerator after the first is cancelled by a term in the denominator of the previous factor. The above discussion can be shown to be true in general. Namely:

Theorem

The number *m* of ordered partitions of a set *S* with *n* elements into *r* cells $[A_1, A_2, ..., A_r]$ where, for each *i*, $n(A_i) = n_i$, follows:

$$m = \frac{n!}{n_1!n_2!\ldots n_r!}$$

Unordered Partitions

Frequently, we want to partition a set S into cellswhere the cells $[A_1, A_2, \ldots, A_r]$ are now unordered. The number m of such unordered partitions is obtained m' from the number of ordered partitions by dividing m by each k! where k of the cells have the same number of element.

EXAMPLE

Find the number m of ways to partition 10 students into four teams $[A_1,A_2,A_3,A_4]$ so that two teams contain 3 students and two teams contain 2 students.

By Theorem 6.2, there are m= 10!/(3!3!2!2!) = 25 200 such ordered partitions.
Since the teams form an unordered partition, we divide m by 2! because of the two cells with 3 elements each and 2! because of the two cells with 2 elements each. Thus m = 25 200/(2!2!) = 6300.

Unit – II

1.Order and Inequalities

An inequality is a mathematical statement which shows that two values are not equal.

$a \neq b$ means that a is not equal to b. It means that either a is less or greater than b.

There are special symbols that show how things are not equal.

- $a < b$ means that a is less than b
- $a > b$ means that a is greater than b
- $a \leq b$ means that a is less than or equal to b
- $a \geq b$ means that a is greater than or equal to b

Example

To compare two numbers, we use the symbols, or =.

Use <, >, or = to compare the numbers given below

18 21

72 38

109 163

Solution

Step 1:Since 18 is less than 21 we write 18 < 21

Step 2: As 72 is greater than 38, we write 72 > 38

Step 3: As 109 is less than 163, we write 109 < 163

2. Mathematical Induction

Mathematical induction, is a technique for proving results or establishing statements for natural numbers. This part illustrates the method through a variety of examples.

Definition

Mathematical Induction is a mathematical technique which is used to prove a statement, a formula or a theorem is true for every natural number.

The technique involves two steps to prove a statement, as stated below −

Step 1(Base step): It proves that a statement is true for the initial value.

Step 2(Inductive step): It proves that if the statement is true for the nth iteration (or number n), then it is also true for (n+1)th iteration (or number n+1).

How to Do It

Step 1: Consider an initial value for which the statement is true. It is to be shown that the statement is true for n = initial value.

Step 2: Assume the statement is true for any value of n = k. Then prove the statement is true for n = k+1. We actually break n = k+1 into two parts, one part is n = k (which is already proved) and try to prove the other part.

Problem 1

3n−13n−1 is a multiple of 2 for n = 1, 2, ...

Solution

Step 1: For n=1,31−1=3−1=2n=1,31−1=3−1=2 which is a multiple of 2

Step 2: Let us assume 3n−13n−1 is true for n=kn=k, Hence, 3k−13k−1 is true (It is an assumption)

We have to prove that 3k+1−13k+1−1 is also a multiple of 2

3k+1−1=3×3k−1=(2×3k)+(3k−1)3k+1−1=3×3k−1=(2×3k)+(3k−1)

The first part $(2\times3k)(2\times3k)$ is certain to be a multiple of 2 and the second part $(3k-1)(3k-1)$ is also true as our previous assumption.

Hence, $3k+1-13k+1-1$ is a multiple of 2.

So, it is proved that $3n-13n-1$ is a multiple of 2.

3. Division Algorithm

The following fundamental property of arithmetic is essentially a restatement of the result of long division.

Theorem

- Let a and b be integers with $b = 0$.
- Then there exists integers q and r such that $a = bq + r$ and $0 \leq r < |b|$.
- Also, the integers q and r are unique.
- The number q in the above theorem is called the quotient, and r is called the remainder.
- We stress the fact that r must be non-negative.
- The theorem also states that $r = a - bq$
- This equation will be used subsequently If a and b are positive, then q is non negative.
- If b is positive, then Fig. 11-2 gives a geometrical interpretation of this theorem.
- That is, the positive and negative multiples of b will be evenly distributed throughout the number line R, and a will fall between some multiples qb and $(q + 1)b$.
- The distance between qb and a is then the remainder r.

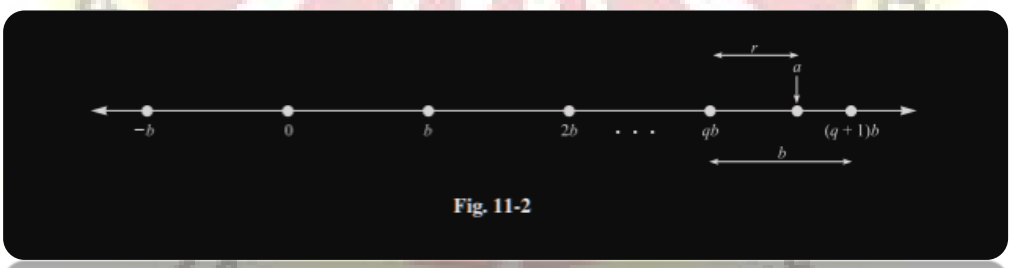


Fig. 11-2

4. DIVISIBILITY

Let a and b be integers with $a \neq 0$. Suppose $ac = b$ for some integer c. We then say that a divides b or b is divisible by a, and we denote this by writing $a \mid b$

We also say that b is a multiple of a or that a is a factor or divisor of b. If a does not divide b, we will write $a \neq \mid b$.

EXAMPLE

(a) Clearly, $3 \mid 6$ since $3 \cdot 2 = 6$, and $-4 \mid 28$ since $(-4)(-7) = 28$.

(b) The divisors of 4 are $\pm1, \pm2, \pm4$ and the divisors of 9 are $\pm1, \pm3, \pm9$.

(c) If $a = 0$, then $a \mid 0$ since $a \cdot 0 = 0$.

(d) Every integer a is divisible by ± 1 and $\pm a$. These are sometimes called the trivial divisors of a. The basic properties of divisibility is stated in the next theorem

Theorem

Suppose a, b, c are integers.

(i) If $a \mid b$ and $b \mid c$, then $a \mid c$.

(ii) If $a \mid b$ then, for any integer x, $a \mid bx$.

(iii) If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$ and $a \mid (b - c)$.

(iv) If $a \mid b$ and $b = 0$, then $a = \pm b$ or $|a| < |b|$.

(v) If $a \mid b$ and $b \mid a$, then $|a| = |b|$, i.e., $a = \pm b$.

(vi) If $a \mid 1$, then $a = \pm 1$

Putting (ii) and (iii) together, we obtain the following important result.

Note: Suppose $a \mid b$ and $a \mid c$. Then, for any integers x and y, $a \mid (bx + cy)$. The expression $bx + cy$ will be called a linear combination of b and c. Primes A positive integer $p > 1$ is called a prime number or a prime if its only divisors are ± 1 and $\pm p$, that is, if p only has trivial divisors. If $n > 1$ is not prime, then n is said to be composite. We note (Problem 11.13) that if $n > 1$ is composite then $n = ab$ where $1 < a, b < n$.

5. Euclidean Algorithm

Let a and b be integers, and let $d = \gcd(a, b)$. One can always find d by listing all the divisors of a and then all the divisors of b and then choosing the largest common divisor. The complexity of such an algorithm is $f(n) = 0(\sqrt{n})$ where $n = |a| + |b|$. Also, we have given no method to find the integers x and y such that $d = ax + by$.

This subsection gives a very efficient algorithm, called the Euclidean algorithm, with complexity $f(n) = O(\log n)$, for finding $d = \gcd(a, b)$ by applying the division algorithm to a and b and then repeatedly applying it to each new quotient and remainder until obtaining a nonzero remainder. The last nonzero remainder is $d = \gcd(a, b)$.

Then we give an "unraveling" algorithm which reverses the steps in the Euclidean algorithm to find the integers x and y such that $d = xa + yb$.

We illustrate the algorithms with an example.

Example

Let $a = 540$ and $b = 168$. We apply the Euclidean algorithm to a and b. These steps, which repeatedly apply the division algorithm to each quotient and remainder until obtaining a zero remainder, are pictured using long division and also where the arrows indicate the quotient and remainder in the next step. The last nonzero remainder is 12.

Thus 12 = gcd(540, 168)

This follows from the fact that

gcd(540, 168) = gcd(168, 36) = gcd(36, 24) = gcd(24, 12) = 12

Next we find x and y such that 12 = 540x+168y by "unraveling" the above steps in the Euclidean algorithm. Specifically, the first three quotients in Fig. 11-3 yield the following equations:

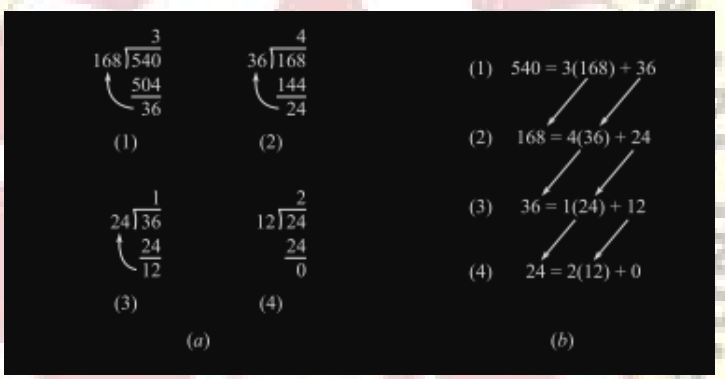
(1) 36 = 540 − 3(168)

(2) 24 = 168 − 4(36)

(3) 12 = 36 − 1(24)

Equation (3) tells us that d = gcd(a, b) = 12 is a linear combination of 36 and 24. Now we use the preceding equations in reverse order to eliminate the other remainders. That is, first we use equation (2) to replace 24 in equation (3) so we can write 12 as a linear combination of 168 and 36 as follows:

(4) 12 = 36 − 1[168 − 4(36)] = 36 − 1(168) + 4(36) = 5(36) − 1(168)



Next we use equation (1) to replace 36 in (4) so we can write 12 as a linear combination of 168 and 540 as follows:

12 = 5[540 − 3(168)] − 1(168) = 5(54) − 15(168) − 1(168) = 5(540) − 16(168)

This is our desired linear combination. In other words, x = 5 and y = −16.

6. CONGRUENCE RELATION

Let m be a positive integer. We say that a is congruent to b modulo m. written a ≡ b (modulo m) or simply a ≡ b (mod m) if m divides the difference a −b .

The integer m is called the modulus. The negation of a ≡ b (mod m) is written a ≡ b (mod m).

Example

(i)87 ≡ 23 (mod 4) since 4 divides 87 − 23 = 64.

(ii) 67 ≡ 1 (mod 6) since 6 divides 67 − 1 = 66.

(iii) 72 ≡ −5 (mod 7) since 7 divides 72 − (−5) = 77.

7. Congruence Equations

A polynomial congruence equation or, simply, a congruence equation (in one unknown x) is an equation of the form

$$\mathbf{anxn + an-1xn-1 + ... + a1x + a0 \equiv 0 \ (mod\ m)}$$

Such an equation is said to be of degree n if a ≡ 0 (mod m). Suppose s ≡ t (mod m). Then s is a solution of (11.2) if and only if t is a solution of (11.2). Thus the number of solutions of (11.2) is defined to be the number of incongruent solutions or, equivalently, the number of solutions in the set

$$\mathbf{\{0, 1, 2,...,m - 1\}}$$

Of course, these solutions can always be found by testing, that is, by substituting each of the m numbers into equation to see if it does indeed satisfy the equation.

The complete set of solutions of the equation is a maximum set of incongruent solutions whereas the general solution of (11.2) is the set of all integral solutions of the general solution of the equation can be found by adding all the multiples of the modulus m to any complete set of solutions.

8. Semi-groups

Let S be a nonempty set with an operation. Then S is called a semigroup if the operation is associative. If the operation also has an identity element, then S is called a monoid.

Example

(a) Consider the positive integers N. Then (N, +) and (N, ×) are semigroups since addition and multiplication on N are associative. In particular, (N, ×) is a monoid since it has the identity element 1. However, (N, +) is not a monoid since addition in N has no zero element. **(b)** Let S be a finite set, and let F (S) be the collection of all functions f : S → S under the operation of composition of functions. Since the composition of functions is associative, F (S) is a semigroup. In fact, F (S) is a monoid since the identity function is an identity element for F (S).

(c) Let S = {a, b, c, d}. The multiplication tables in Fig. B-1 define operations ∗ and · on S. Note that ∗ can be defined by the formula x ∗ y = x for any x and y in S. Hence

$$\mathbf{(x * y) * z = x * z = x \text{ and } x * (y * z) = x * y = x}$$

Therefore, ∗ is associative and hence (S, ∗) is a semigroup. On the other hand, · is not associative since, for example,

$$\mathbf{(b \cdot c) \cdot c = a \cdot c = c \text{ but } b \cdot (c \cdot c) = b \cdot a = b}$$

Thus (S, ·) is not a semi-group.

9. Groups

Let G be a nonempty set with a binary operation (denoted by juxtaposition). Then G is called a group if the following axioms hold:

[G1] Associative Law: For any a, b, c in G, we have (ab)c = a(bc).

[G2] Identity element: There exists an element e in G such that ae = ea = a for every a in G.
[G3] Inverses: For each a in G, there exists an element a−1 in G (the inverse of a) such that

aa−1 = a−1a = e

A group G is said to be abelian (or commutative) if ab = ba for every a, b ∈ G, that is, if G satisfies the Commutative Law.

When the binary operation is denoted by juxtaposition as above, the group G is said to be written multiplicatively. Sometimes, when G is abelian, the binary operation is denoted by + and G is said to be written additively. In such a case the identity element is denoted by 0 and it is called the zero element; and the inverse is denoted by −a and it is called the negative of a.

The number of elements in a group G, denoted by |G|, is called the order of G. In particular, G is called a finite group if its order is finite.

Suppose A and B are subsets of a group G. Then we write:

AB = {ab | a ∈ A, b ∈ B} or A + B = {a + b | a ∈ A, b ∈ B}

10. Sub Groups

Let H be a subset of a group G. Then H is called a subgroup of G if H itself is a group under the operation of G. Simple criteria to determine subgroups follow.

Proposition: A subset H of a group G is a subgroup of G if:

(i) The identity element e ∈ H.

(ii) H is closed under the operation of G, i.e. if a, b ∈ H, then ab ∈ H.

(iii) H is closed under inverses, that is, if a ∈ H, then a−1 ∈ H.

Every group G has the subgroups {e} and G itself. Any other subgroup of G is called a nontrivial subgroup.

Cosets

Suppose H is a subgroup of G and a ∈ G. Then the set H a = {ha | h ∈ H} is called a right coset of H.

Theorem: Let H be a subgroup of a group G. Then the right cosets H a form a partition of G.
Theorem: Let H be a subgroup of a finite group G. Then the order of H divides the order of G.

The number of right cosets of H in G, called the index of H in G, is equal to the number of left cosets of H in G; and both numbers are equal to |G| divided by |H|.

11. Normal Subgroups

Definition: A subgroup H of G is a normal subgroup if $a^{-1}Ha \subseteq H$, for every $a \in G$, or, equivalently, if $aH = Ha$, i.e., if the right and left cosets coincide.

Note that every subgroup of an abelian group is normal.

The importance of normal subgroups comes from the following result

Theorem: Let H be a normal subgroup of a group G. Then the cosets of H form a group under coset multiplication:

$$(aH)(bH) = abH$$

This group is called the quotient group and is denoted by G/H.

Suppose the operation in G is addition or, in other words, G is written additively. Then the cosets of a subgroup H of G are of the form a + H. Moreover, if H is a normal subgroup of G, then the cosets form a group under coset addition, that is,

$$(a + H) + (b + H) = (a + b) + H.$$

12. Homomorphisms

A mapping f from a group G into a group G is called a homomorphism if, for every a, $b \in G$,

$$f(ab) = f(a)f(b)$$

In addition, if f is one-to-one and onto, then f is called an isomorphism; and G and G are said to be isomorphic, written $G \sim= G$.

If $f : G \rightarrow G$ is a homomorphism, then the kernel of f , written Ker f , is the set of elements whose image is the identity element e of G ; that is,

$$\text{Ker } f = \{a \in G \mid f(a) = e\}$$

Recall that the image of f , written f (G) or Im f , consists of the images of the elements under f ; that is,

$$\text{Im } f = \{b \in G \mid \text{there exists } a \in G \text{ for which } f(a) = b\} .$$

The following theorem is fundamental to group theory

Theorem : Suppose $f : G \rightarrow G$ is a homomorphism with kernel K. Then K is a normal subgroup of G, and the quotient group G/K is isomorphic to f (G).

Example:

(a) Let G be the group of real numbers under addition, and let G be the group of positive real numbers under multiplication. The mapping $f : G \rightarrow G$ defined by $f(a) = 2a$ is a homomorphism because

$$f(a + b) = 2^{a+b} = 2^a 2^b = f(a)f(b)$$

In fact, f is also one-to-one and onto; hence G and G are isomorphic.

(b) Let a be any element in a group G. The function f : Z → G defined by f (n) = an is a homomorphism since

$$\mathbf{f(m + n) = a^{m+n} = a^m \cdot a^n = f(m) \cdot f(n)}$$

The image of f is gp(a), the cyclic subgroup generated,

$$\mathbf{gp(a) \sim= Z/K}$$

where K is the kernel of f. If K = {0}, then gp(a) = Z. On the other hand, if m is the order of a, then K ={multiples of m}, and so gp(a) ~= Zm. In other words, any cyclic group is isomorphic to either the integers Z under addition, or to Zm, the integers under addition modulo m.

13. Graph Theory

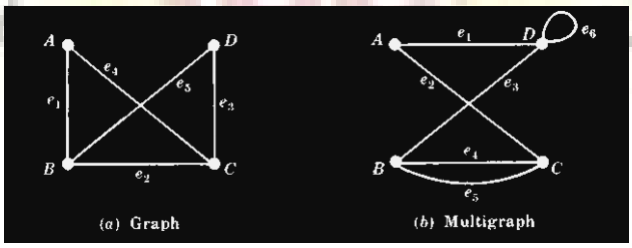
GRAPHS AND MULTIGRAPHS

A graph G consists of two things:

(i) A set V = V (G) whose elements are called vertices, points, or nodes of G.

(ii) A set E = E(G) of unordered pairs of distinct vertices called edges of G.

We denote such a graph by G(V , E) when we want to emphasize the two parts of G. Vertices u and v are said to be adjacent or neighbors if there is an edge e = {u, v}. In such a case, u and v are called the endpoints of e, and e is said to connect u and v. Also, the edge e is said to be incident on each of its endpoints u and v. Graphs are pictured by diagrams in the plane in a natural way. Specifically, each vertex v in V is represented by a dot (or small circle), and each edge e = {v1, v2} is represented by a curve which connects its endpoints v1 and v2 For example, The Fig below represents the graph G(V , E) where:



(i) V consists of vertices A, B, C, D.

(ii) E consists of edges e1 = {A, B}, e2 = {B,C}, e3 = {C, D}, e4 = {A, C}, e5 = {B,D}.

In fact, we will usually denote a graph by drawing its diagram rather than explicitly listing its vertices and edges.

Multigraphs:

Consider the above diagram. The edges e4 and e5 are called multiple edges since they connect the same endpoints, and the edge e6 is called a loop since its endpoints are the same vertex. Such a diagram is called a multigraph; the formal definition of a graph permits neither multiple edges nor loops. Thus a graph may be defined to be a multigraph without multiple edges or loops.

PATHS, CONNECTIVITY

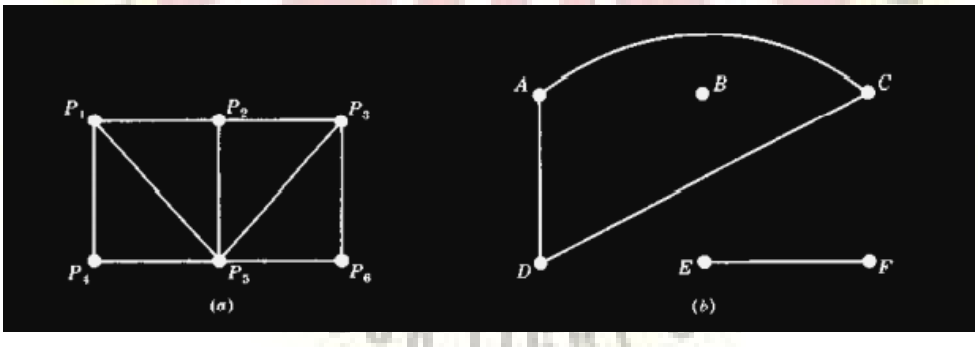
A path in a multigraph G consists of an alternating sequence of vertices and edges of the form

v0, e1, v1, e2, v2, ..., en−1, vn−1, en, vn

where each edge ei contains the vertices vi−1 and vi (which appear on the sides of ei in the sequence). The number n of edges is called the length of the path. When there is no ambiguity, we denote a path by its sequence of vertices (v0, v1,...,vn). The path is said to be closed if v0 = vn. Otherwise, we say the path is from v0, to vn or between v0 and vn, or connects v0 to vn.

A simple path is a path in which all vertices are distinct. (A path in which all edges are distinct will be called a trail.) A cycle is a closed path of length 3 or more in which all vertices are distinct except v0 = vn. A cycle of length k is called a k-cycle.

EXAMPLE



Consider the graph G given above.

Consider the following sequences:

α = (P4, P1, P2, P5, P1, P2, P3, P6), β = (P4, P1, P5, P2, P6),

γ = (P4, P1, P5, P2, P3, P5, P6), δ = (P4, P1, P5, P3, P6).

The sequence α is a path from P4 to P6; but it is not a trail since the edge {P1, P2} is used twice. The sequence β is not a path since there is no edge {P2, P6}. The sequence γ is a trail since no edge is used twice; but it is not a simple path since the vertex P5 is used twice. The sequence δ is a simple path from P4 to P6; but it is not the shortest path (with respect to

length) from P4 to P6. The shortest path from P4 to P6 is the simple path (P4, P5, P6) which has length 2.

By eliminating unnecessary edges, it is not difficult to see that any path from a vertex u to a vertex v can be replaced by a simple path from u to v. We state this result formally.

Unit- IV

1. Properties of Regular sets

Any set that represents the value of the Regular Expression is called a Regular Set.

Properties of Regular Sets

Property 1

The union of two regular set is regular.

Proof:

- Let us take two regular expressions
- RE1 = a (aa)* and RE2 = (aa)*
- So, L1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)
  and L2 ={ ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)
- L1 ∪ L2 = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.....}
- (Strings of all possible lengths including Null)
- RE (L1 ∪ L2) = a* (which is a regular expression itself)
- Hence, proved.

Property 2

The intersection of two regular set is regular.

Proof:

- Let us take two regular expressions
- RE1 = a(a*) and RE2 = (aa)*
- So, L1 = { a,aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)
- L2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)
- L1 ∩ L2 = { aa, aaaa, aaaaaa,.....} (Strings of even length excluding Null)
- RE (L1 ∩ L2) = aa(aa)* which is a regular expression itself.
- Hence, proved.

Property 3

The complement of a regular set is regular.

Proof:

- Let us take a regular expression
- RE = (aa)*

- So, L = {ε, aa, aaaa, aaaaaa, ...} (Strings of even length including Null)
- Complement of L is all the strings that is not in L.
- So, L' = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)
- RE (L') = a(aa)* which is a regular expression itself.
- Hence, proved.

Property 4

The difference of two regular set is regular.

Proof:

- Let us take two regular expressions.
- RE1 = a (a*) and RE2 = (aa)*
- So, L1 = {a, aa, aaa, aaaa, ...} (Strings of all possible lengths excluding Null)
- L2 = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)
- L1 – L2 = {a, aaa, aaaaa, aaaaaaa, ...}
- (Strings of all odd lengths excluding Null)
- RE (L1 – L2) = a (aa)* which is a regular expression.
- Hence, proved.

Property 5

The reversal of a regular set is regular.

Proof:

- We have to prove LR is also regular if L is a regular set.
- Let, L = {01, 10, 11, 10}
- RE (L) = 01 + 10 + 11 + 10
- LR = {10, 01, 11, 01}
- RE (LR) = 01 + 10 + 11 + 10 which is regular
- Hence, proved.

Property 6

The closure of a regular set is regular.

Proof:

- If L = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)
- i.e., RE (L) = a (aa)*
- L* = {a, aa, aaa, aaaa , aaaaa,……} (Strings of all lengths excluding Null)
- RE (L*) = a (a)*
- Hence, proved.

Property 7

The concatenation of two regular sets is regular.

Proof:

- Let RE1 = (0+1)*0 and RE2 = 01(0+1)*
- Here, L1 = {0, 00, 10, 000, 010, ...} (Set of strings ending in 0)
  and L2 = {01, 010,011,.....} (Set of strings beginning with 01)
- Then, L1 L2 = {001,0010,0011,0001,00010,00011,1001,10010,.....}
- Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*
- Hence, proved.

Identities Related to Regular Expressions

Given R, P, L, Q as regular expressions, the following identities hold −

- Ø* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*
- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*
- R + Ø = Ø + R = R (The identity for union)
- R ε = ε R = R (The identity for concatenation)
- Ø L = L Ø = Ø (The annihilator for concatenation)
- R + R = R (Idempotent law)
- L (M + N) = LM + LN (Left distributive law)
- (M + N) L = ML + NL (Right distributive law)
- ε + RR* = ε + R*R = R*

2. Pumping lemma

Theorem

Let L be a regular language. Then there exists a constant 'c' such that for every string w in L.

$$|w| \geq c$$

We can break w into three strings, w = xyz, such that

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xykz is also in L.

Applications of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

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

Method to prove that a language L is not regular

- At first, we have to assume that L is regular.
- So, the pumping lemma should hold for L.
- Use the pumping lemma to obtain a contradiction

- Select w such that $|w| \geq c$
- Select y such that $|y| \geq 1$
- Select x such that $|xy| \leq c$
- Assign the remaining string to z.
- Select k such that the resulting string is not in L.
- Hence L is not regular.

Problem

Prove that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

Solution

- At first, we assume that L is regular and n is the number of states.
- Let $w = a^n b^n$. Thus $|w| = 2n \geq n$.
- By pumping lemma, let $w = xyz$, where $|xy| \leq n$.
- Let $x = a^p$, $y = a^q$, and $z = a^r b^n$, where $p + q + r = n$, $p \neq 0$, $q \neq 0$, $r \neq 0$. Thus $|y| \neq 0$.
- Let $k = 2$. Then $xy^2z = a^p a^{2q} a^r b^n$.
- Number of as = $(p + 2q + r) = (p + q + r) + q = n + q$
- Hence, $xy^2z = a^{n+q} b^n$. Since $q \neq 0$, xy^2z is not of the form $a^n b^n$.
- Thus, xy^2z is not in L. Hence L is not regular.

3.Closure Properties

Closure properties on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language. Closure refers to some operation on a language, resulting in a new language that is of same "type" as originally operated on i.e., regular.

Regular languages are closed under following operations.

Consider L and M are regular languages:

1.KleenClosure:

RS is a regular expression whose language is L, M. R* is a regular expression whose language is L*.

2.Positive closure:

RS is a regular expression whose language is L, M. R+ is a regular expression whose language is L+.

3. Complement:

The complement of a language L (with respect to an alphabet **E** such that **E*** contains L) is E*–L. Since E* is surely regular, the complement of a regular language is always regular.

4. Reverse Operator:

Given language L, LR is the set of strings whose reversal is in L.

Example:

L = {0, 01, 100};
={0, 10, 001}.

Proof:

Let E be a regular expression for L. We show how to reverse E, to provide a regular expression ER for LR.

5. Union:

Let L and M be the languages of regular expressions R and S, respectively. Then R+S is a regular expression whose language is(L U M).

6. Intersection:

Let L and M be the languages of regular expressions R and S, respectively then it a regular expression whose language is L intersection M.

Proof:

Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs consisting of final states of both A and B.

7. Set Difference operator:

If L and M are regular languages, then so is L – M = strings in L but not M.

Proof:

Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs, where A-state is final but B-state is not.

8.Homomorphism:

A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet.

Example:

- h(0) = ab; h(1) = E. Extend to strings by h(a1…an) =h(a1)…h(an).
- h(01010) = abababab.

If L is a regular language, and h is a homomorphism on its alphabet, then h(L)= {h(w) | w is in L} is also a regular language.

Proof:

Let E be a regular expression for L. Apply h to each symbol in E. Language of resulting R, E is h(L).

10. Inverse Homomorphism:

Let h be a homomorphism and L a language whose alphabet is the output language of h. h-1 (L) = {w | h(w) is in L}.

Note: There are few more properties like symmetric difference operator, prefix operator, substitution which are closed under closure properties of regular language.

4.DecisionProperties

Approximately all the properties are decidable in case of finite automaton.

(i) Emptiness

(ii) Non-emptiness

(iii) Finiteness

(iv) Infiniteness

(v) Membership

(vi) Equality

These are explained as following below.

(i) Emptiness and Non-emptiness:

Step-1: Select the state that cannot be reached from the initial states & delete them (remove unreachable states).

Step 2: If the resulting machine contains at least one final states, so then the finite automata accepts the non-empty language.

Step 3: If the resulting machine is free from final state, then finite automata accepts empty language.

(ii) Finiteness and Infiniteness:

Step-1: Select the state that cannot be reached from the initial state & delete them (remove unreachable states).

Step-2: Select the state from which we cannot reach the final state & delete them (remove dead states).

Step-3: If the resulting machine contains loops or cycles then the finite automata accepts infinite language.

Step-4: If the resulting machine do not contain loops or cycles then the finite automata accepts infinite language.

(iii)Membership:

Membership is a property to verify an arbitrary string is accepted by a finite automaton or not i.e. it is a member of the language or not.

Let M is a finite automata that accepts some strings over an alphabet, and let 'w' be any string defined over the alphabet, if there exist a transition path in M, which starts at initial state & ends in anyone of the final state, then string 'w' is a member of M, otherwise 'w' is not a member of M.

(iv)Equality:

Two finite state automata M1 & M2 is said to be equal if and only if, they accept the same language. Minimise the finite state automata and the minimal DFA will be unique.

Attention reader! Don't stop learning now. Get hold of all the important DSA concepts with the DSA Self-Paced Course at a student-friendly price and become industry ready.

6. My hill - Nerode Theorem

DFA Minimization using Myhill-Nerode Theorem Algorithm

Input: DFA

Output: Minimized DFA

Step 1: Draw a table for all pairs of states (Qi ,Qj) not necessarily connected directly [All are unmarked initially]

Step 2: Consider every state pair (Qi ,Qj) in the DFA where Qi ∈ F and Qj∉ F or vice versa and mark them. [Here F is the set of final states].

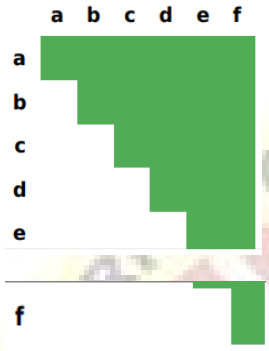
Step 3: Repeat this step until we cannot mark anymore states − If there is an unmarked pair (Qi ,Qj), mark it if the pair {δ(Qi , A), δ (Qi , A)} is marked for some input alphabet.

Step 4: Combine all the unmarked pair (Qi ,Qj) and make them a single state in the reduced DFA.

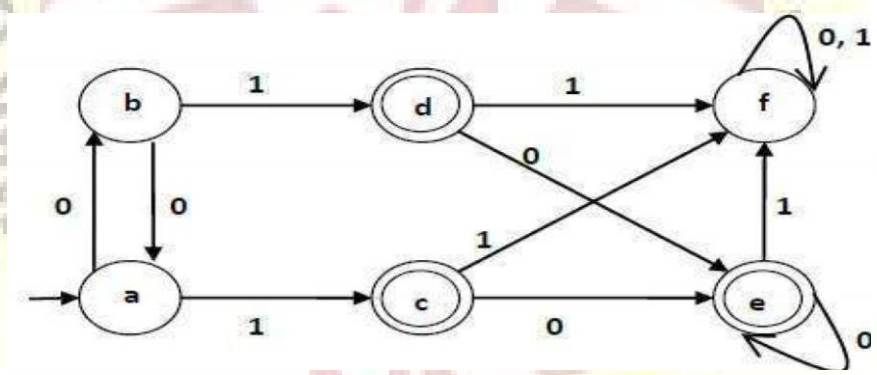
Example:

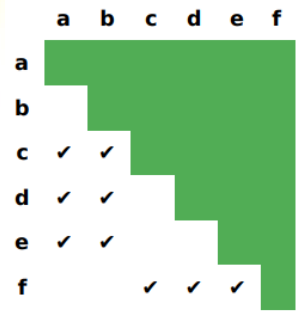
Let us use above algorithm to minimize the DFA shown below

Step 1: We draw a table for all pair of state

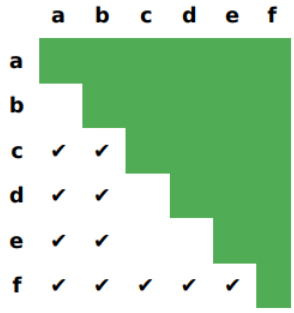


Step 2: We mark the state pairs

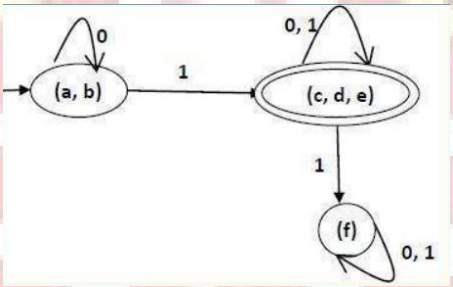




Step 3: We will try to mark the state pairs, with green colored check mark, transitively. If we input 1 to state 'a' and 'f', it will go to state 'c' and 'f' respectively. c, f is already marked, hence we will mark pair a, f. Now, we input 1 to state 'b' and 'f'; it will go to state'd' and 'f' respectively. d, f is already marked, hence we will mark pair b, f.



- After step 3, we have got state combinations {a, b} {c, d} {c, e} {d, e} that are unmarked.
- We can recombine {c, d} {c, e} {d, e} into {c, d, e} Hence we got two combined states as − {a, b} and {c, d, e}.
- So the final minimized DFA will contain three states {f}, {a, b} and {c, d, e}.



7. Context Free Grammars

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

G - Describes the grammar

T - Describes a finite set of terminal symbols.

V - Describes a finite set of non-terminal symbols

P - Describes a set of production rules

S - Is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcwR | w € (a, b)*}

Production rules:

$S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow c$

Now check that abbcbba string can be derived from the given CFG.

$S \Rightarrow aSa$

$S \Rightarrow abSba$

$S \Rightarrow abbSbba$

$S \Rightarrow abbcbba$

By applying the production $S \rightarrow aSa$, $S \rightarrow bSb$ recursively and finally applying the production $S \rightarrow c$, we get the string abbcbba.

Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficientparser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

We have to decide the non-terminal which is to be replaced.

We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

Production rules:

S = S + S

S = S - S

S = a | b |c

Input:

a - b + c

The left-most derivation is:

S = S + S

S = S - S + S

S = a - S + S

S = a - b + S

S = a - b + c

Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

S = S + S

S = S - S

S = a | b |c

Input:

a - b + c

The right-most derivation is:

S = S - S

S = S - S + S

S = S - S + c

S = S - b + c

S = a - b + c

Parse tree

Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.

In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.

It is the graphical representation of symbol that can be terminals or non-terminals.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

Production rules:

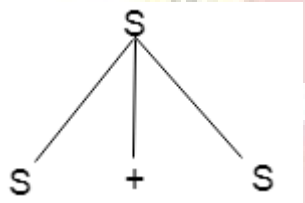
T= T + T | T * T

T = a|b|c

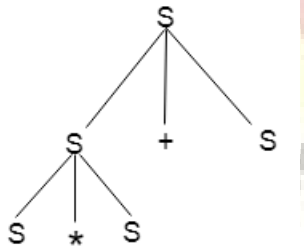
Input:

a * b + c

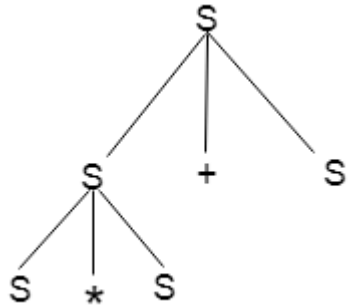
Step 1:



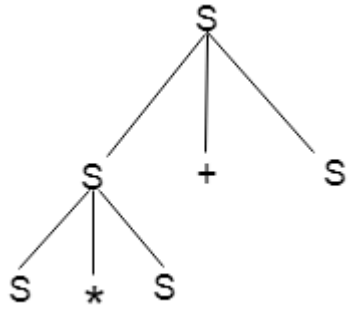
Step 2:



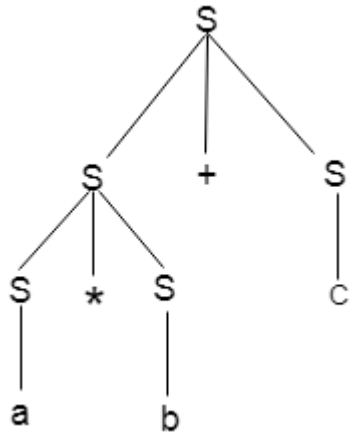
Step 3:



Step 4:



Step 5:



Ambiguity

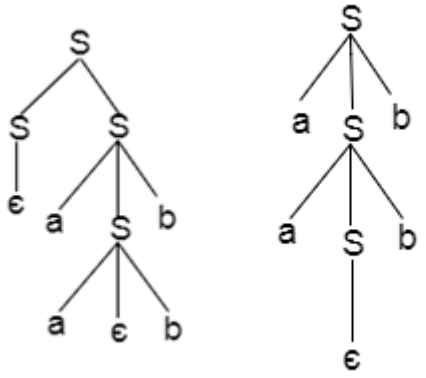
A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

S = aSb | SS

S = ∈

For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

Derivation Trees.

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG. It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules. The derivation tree is also called a parse tree.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

A parse tree contains the following properties:

1. The root node is always a node indicating start symbols.
2. The derivation is read from left to right.
3. The leaf node is always terminal nodes.
4. The interior nodes are always the non-terminal nodes.

Example 1:

Production rules:

E = E + E

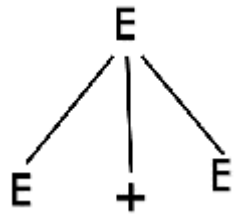
E = E * E

E = a | b | c

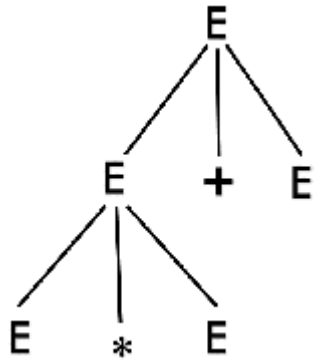
Input

a * b + c

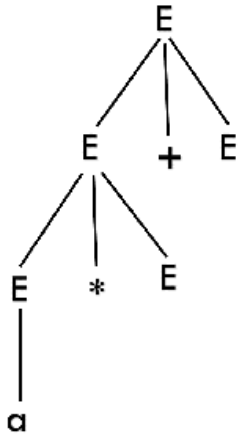
Step 1:



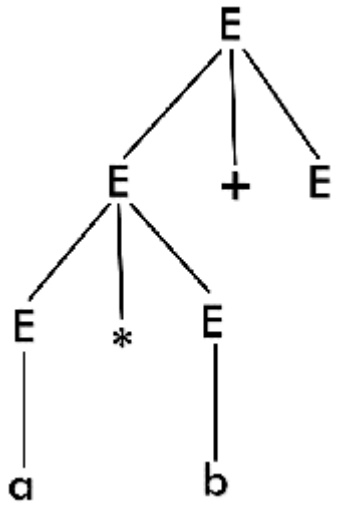
Step 2:



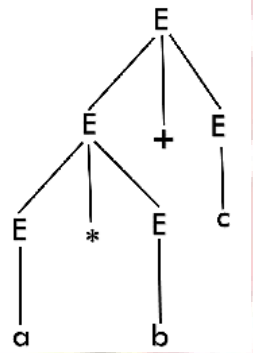
Step 3:



Step 4:



Step 5:



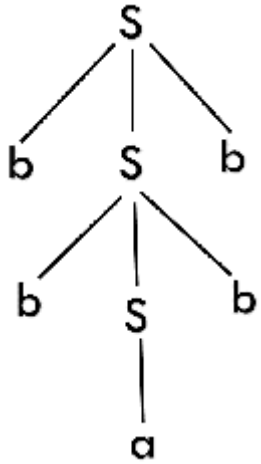
Example 2:

Draw a derivation tree for the string "bab" from the CFG given by

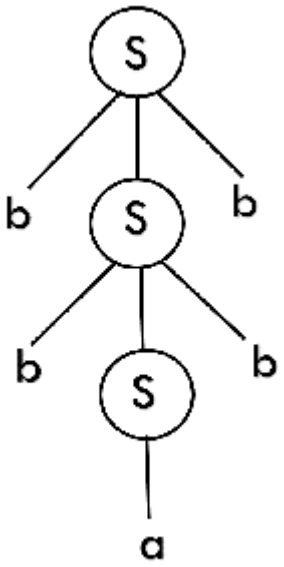
S → bSb | a | b

Solution:

Now, the derivation tree for the string "bbabb" is as follows:



The above tree is a derivation tree drawn for deriving a string bbabb. By simply reading the leaf nodes, we can obtain the desired string. The same tree can also be denoted by,



Example 3:

Construct a derivation tree for the string aabbabba for the CFG given by,

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

Solution:

To draw a tree, we will first try to obtain derivation for the string aabbabba

S

aB

a aBB

aa bS B

aab bA B

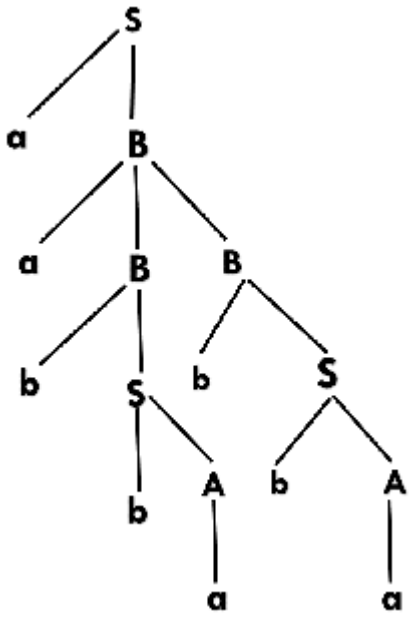
aabb a B

aabba bS

aabbab bA

aabbabb a

Now, the derivation tree is as follows:



Ambiguity in Grammar

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

Example 1:

Let us consider a grammar G with the production rule

$E \rightarrow I$

$E \rightarrow E + E$

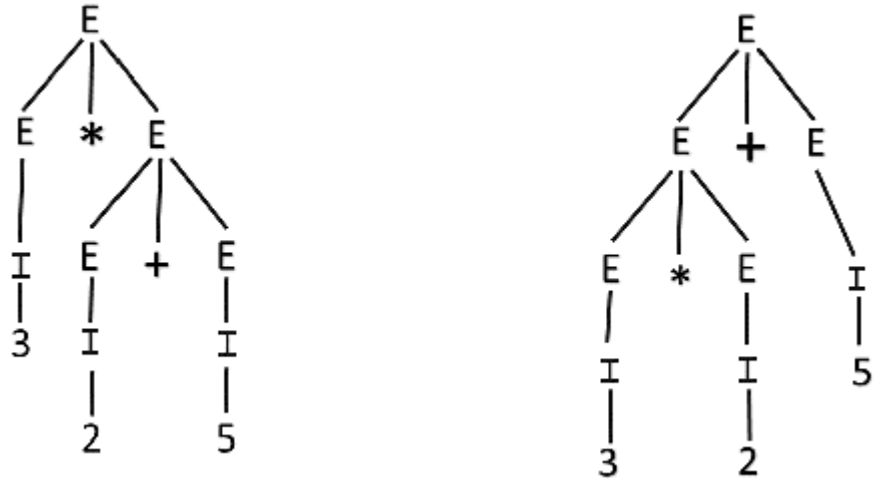
$E \rightarrow E * E$

$E \rightarrow (E)$

$I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid ... \mid 9$

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example 2:

Check whether the given grammar G is ambiguous or not.

E → E + E

E → E - E

E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

E → E + E

→ id + E

→ id + E - E

→ id + id - E

→ id + id- id

Second Leftmost derivation

E → E - E

→ E + E - E

→ id + E - E

→ id + id - E

→ id + id - id

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

Unambiguous Grammar

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

$$\mathbf{X \rightarrow Xa}$$

2. If the right associative operates (^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

$$\mathbf{X \rightarrow aX}$$

Example 1:

Consider a grammar G is given as follows:

$S \rightarrow AB \mid aaB$

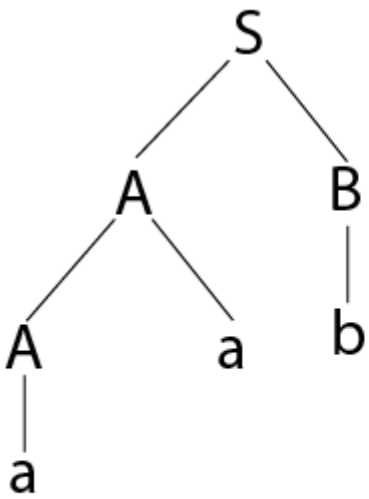
$A \rightarrow a \mid Aa$

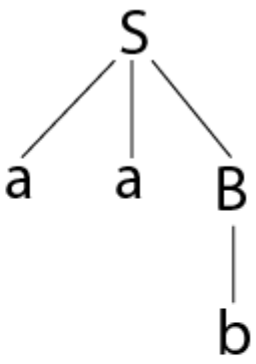
$B \rightarrow b$

Determine whether the grammar G is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G.

Solution:

Let us derive the string "aab"





Parse tree 1 Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

S → AB

A → Aa | a

B → b

Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

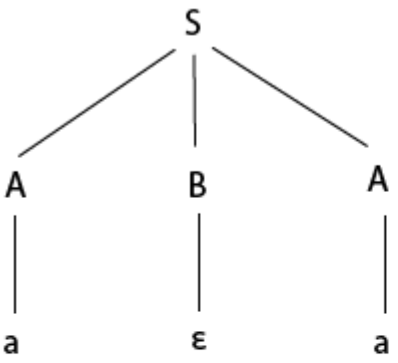
S → ABA

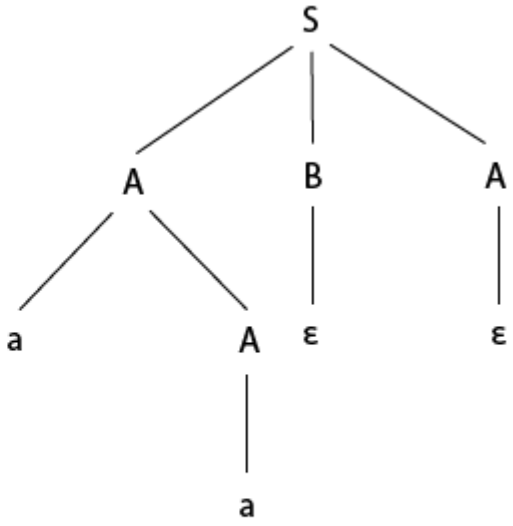
A → aA | ε

B → bB | ε

Solution:

The given grammar is ambiguous because we can derive two different parse tree for string aa.





Parse tree 1 Parse tree 2

The unambiguous grammar is:

S → aXY | bYZ | ε

Z → aZ | a

X → aXY | a | ε

Y → bYZ | b | ε

Unit –V

1. Simplifying Context Free Grammars

The definition of context free grammars (CFGs) allows us to develop a wide variety of grammars. Most of the time, some of the productions of CFGs are not useful and are redundant. This happens because the definition of CFGs does not restrict us from making these redundant productions.

By simplifying CFGs we remove all these redundant productions from a grammar , while keeping the transformed grammar equivalent to the original grammar. Two grammars are called equivalent if they produce the same language. Simplifying CFGs is necessary to later convert them into Normal forms.

Types of redundant productions and the procedure of removing them are mentioned below.

1. **Useless productions**

The productions that can never take part in derivation of any string , are called useless productions. Similarly, a variable that can never take part in derivation of any string is called a useless variable.

For Example

S ->abS | abA | abB

A -> cd

B ->aB

C -> dc

In the example above , production 'C -> dc' is useless because the variable 'C' will never occur in derivation of any string. The other productions are written in such a way that variable 'C' can never reached from the starting variable 'S'.

Production 'B ->aB' is also useless because there is no way it will ever terminate . If it never terminates, then it can never produce a string. Hence the production can never take part in any derivation.

To remove useless productions, we first find all the variables which will never lead to a terminal string such as variable 'B'. We then remove all the productions in which variable 'B' occurs.

So the modified grammar becomes

S ->abS | abA

A -> cd

C -> dc

We then try to identify all the variables that can never be reached from the starting variable such as variable 'C'. We then remove all the productions in which variable 'C' occurs.

The grammar below is now free of useless productions

S ->abS | abA

A -> cd

2. λ productions

The productions of type 'A -> λ' are called λ productions (also called lambda productions and null productions) . These productions can only be removed from those grammars that do not generate λ (an empty string). It is possible for a grammar to contain null productions and yet not produce an empty string.

To remove null productions, we first have to find all the nullable variables. A variable 'A' is called nullable if λ can be derived from 'A'. For all the productions of type 'A ->λ' ,

'A' is a nullable variable. For all the productions of type 'B -> A1A2…An', where all 'Ai's are nullable variables, 'B' is also a nullable variable.

After finding all the nullable variables, we can now start to construct the null production free grammar. For all the productions in the original grammar, we add the original production as well as all the combinations of the production that can be formed by replacing the nullable variables in the production by λ. If all the variables on the RHS of the production are nullable, then we do not add 'A -> λ' to the new grammar. An example will make the point clear.

Consider the grammar

S ->ABCd (1)

A -> BC (2)

B ->bB | λ (3)

C ->cC | λ (4)

Let's first find all the nullable variables. Variables 'B' and 'C' are clearly nullable because they contain 'λ' on the RHS of their production. Variable 'A' is also nullable because in (2), both variables on the RHS are also nullable. Similarly, variable 'S' is also nullable. So variables 'S', 'A', 'B' and 'C' are nullable variables.

Let's create the new grammar. We start with the first production. Add the first production as it is. Then we create all the possible combinations that can be formed by replacing the nullable variables with λ. Therefore line (1) now becomes 'S ->ABCd | ABd | ACd | BCd | Ad | Bd |Cd | d '.

We apply the same rule to line (2) but we do not add 'A -> λ' even though it is a possible combination. We remove all the productions of type 'V -> λ'. The new grammar now becomes

S ->ABCd | ABd | ACd | BCd | Ad |Bd |Cd | d

A -> BC | B | C

B ->bB | b

C ->cC | c

3. Unit productions

The productions of type 'A -> B' are called unit productions. To create a unit production free grammar 'Guf' from the original grammar 'G', we follow the procedure mentioned below.

First add all the non-unit productions of 'G' in 'Guf'. Then for each variable 'A' in grammar 'G' , find all the variables 'B' such that 'A *=> B'. Now, for all variables like 'A' and 'B', add 'A -> x1 | x2 | …xn' to 'Guf' where 'B -> x1 | x2 | …xn 'is in 'Guf'. None of

the x1, x2 … xn are single variables because we only added non-unit productions in 'Guf'. Hence the resultant grammar is unit production free.

Example.

S ->Aa | B

A -> b | B

B -> A | a

Let's add all the non-unit productions of 'G' in 'Guf'. 'Guf' now becomes

S ->Aa

A -> b

B -> a

Now we find all the variables that satisfy 'X *=> Z'. These are 'S *=>A', 'S*=>B', 'A *=> B' and 'B *=> A'. For 'A *=>B', we add 'A -> a' because 'B ->a' exists in 'Guf'. 'Guf' now becomes

S ->Aa

A -> b | a

B -> a

For 'B *=> A', we add 'B -> b' because 'A -> b' exists in 'Guf'. The new grammar becomes

S ->Aa

A -> b | a

B -> a | b

We follow the same step for 'S *=> A' and 'S*=>B' and finally get the following grammar

S ->Aa | b | a

A -> b | a

B -> a | b

Now remove B ->a|b , since it doesn't occur in the production 'S', then the following grammar becomes,

S->Aa|b|a

A->b|a

To remove all kinds of productions mentioned above, first remove the null productions, then the unit productions and finally, remove the useless productions. Following this order is very important to get the correct result.

2. Chomsky Normal Form (CNF)

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

$$\mathbf{A \rightarrow a}$$

$$\mathbf{A \rightarrow BC}$$

$$\mathbf{S \rightarrow \varepsilon}$$

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

Algorithm to Convert into Chomsky Normal Form

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

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

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

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

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

Problem

Convert the following CFG into CNF

$$\mathbf{S \rightarrow ASA \mid aB,\ A \rightarrow B \mid S,\ B \rightarrow b \mid \varepsilon}$$

Solution

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

$$\mathbf{S0 \rightarrow S,\ S \rightarrow ASA \mid aB,\ A \rightarrow B \mid S,\ B \rightarrow b \mid \in}$$

(2) Now we will remove the null productions −

$$\mathbf{B \rightarrow \in \text{ and } A \rightarrow \in}$$

After removing B → ε, the production set becomes −

$$\mathbf{S0 \rightarrow S,\ S \rightarrow ASA \mid aB \mid a,\ A \rightarrow B \mid S \mid \in,\ B \rightarrow b}$$

After removing A → ∈, the production set becomes −

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

(3) Now we will remove the unit productions.

After removing S → S, the production set becomes −

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

After removing S0→ S, the production set becomes −

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

A → B | S, B → b

After removing A→ B, the production set becomes −

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

A → S | b

B → b

After removing A→ S, the production set becomes −

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

A → b |ASA | aB | a | AS | SA, B → b

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

Here,

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

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

S0→ AX | aB | a | AS | SA

S→ AX | aB | a | AS | SA

A → b |AX | aB | a | AS | SA

B → b

X → SA

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

And the final production set becomes

S0→ AX | YB | a | AS | SA

S→ AX | YB | a | AS | SA

A → b A → b |AX | YB | a | AS | SA

B → b

X → SA

Y → a

3. Greibach Normal Form (GNF)

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

A → b

A → bD1…Dn

S → ε

Where

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

Algorithm to Convert a CFG into Greibach Normal Form

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

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

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

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

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

Problem

Convert the following CFG into CNF

S → XY | Xn | p

X → mX | m

Y → Xn | o

Solution

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

Step 4

Now after replacing

X in S → XY | Xo | p

With

mX | m

We obtain

S → mXY | mY | mXo | mo | p.

And after replacing

X in Y → Xn | o

With the right side of

X → mX | m

We obtain

Y → mXn | mn | o.

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

S → mXY | mY | mXC | mC | p

X → mX | m

Y → mXD | mD | o

O → o

P → p

4. Pushdown Automta

Basic Structure of PDA

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

Basically a pushdown automaton is

"Finite state machine" + "a stack"

A pushdown automaton has three components

An input tape,

A control unit, and

A stack with infinite size.

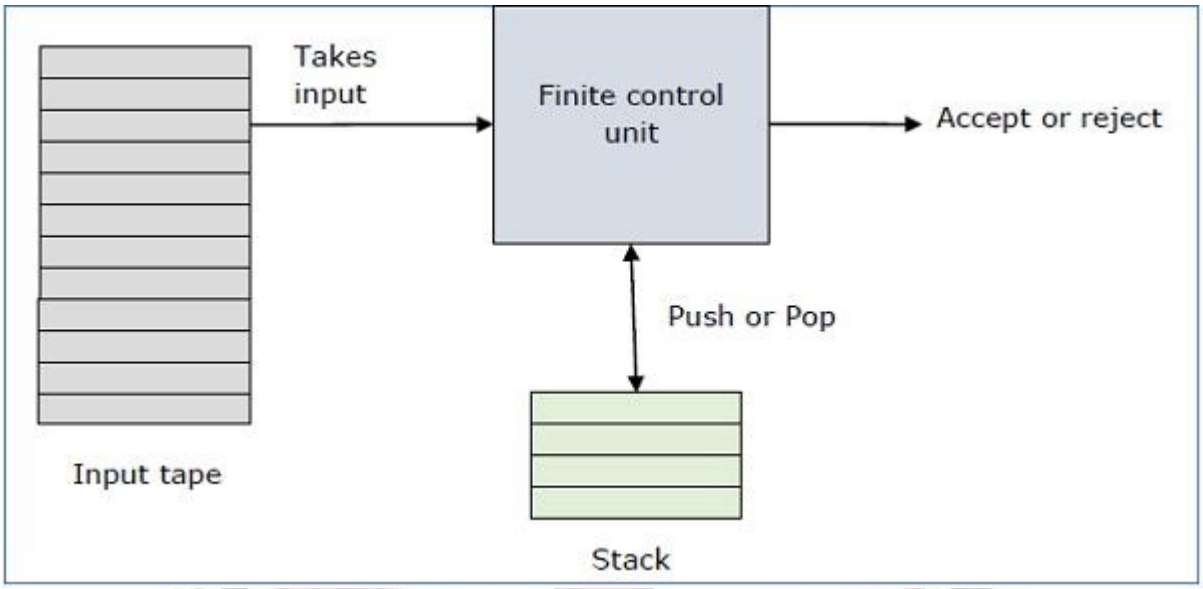
The stack head scans the top symbol of the stack.

A stack does two operations −

Push − a new symbol is added at the top.

Pop − the top symbol is read and removed.

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



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

Q is the finite number of states

∑ is input alphabet

S is stack symbols

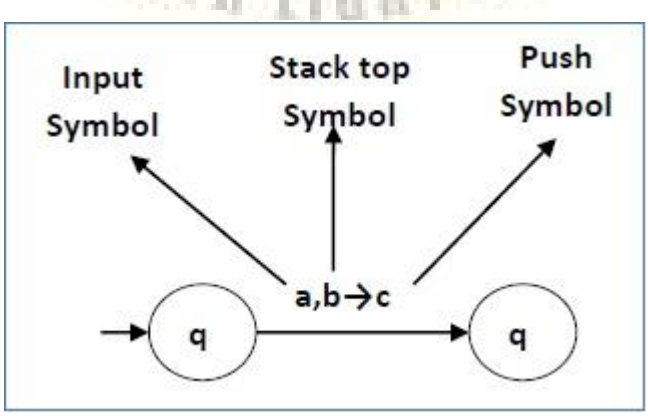
δ is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*

q0 is the initial state (q0 ∈ Q)

I is the initial stack top symbol (I ∈ S)

F is a set of accepting states (F ∈ Q)

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



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

Terminologies Related to PDA

Instantaneous Description

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

q is the state

w is unconsumed input

s is the stack contents

Turnstile Notation

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

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

$$(p, aw, T\beta) \vdash (q, w, \alpha b)$$

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

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

There are two different ways to define PDA acceptability.

Final State Acceptability

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

For a PDA (Q, ∑, S, δ, q0, I, F), the language accepted by the set of final states F is

$$L(PDA) = \{w \mid (q0, w, I) \vdash^* (q, \varepsilon, x), q \in F\}$$

For any input stack string x.

Empty Stack Acceptability

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

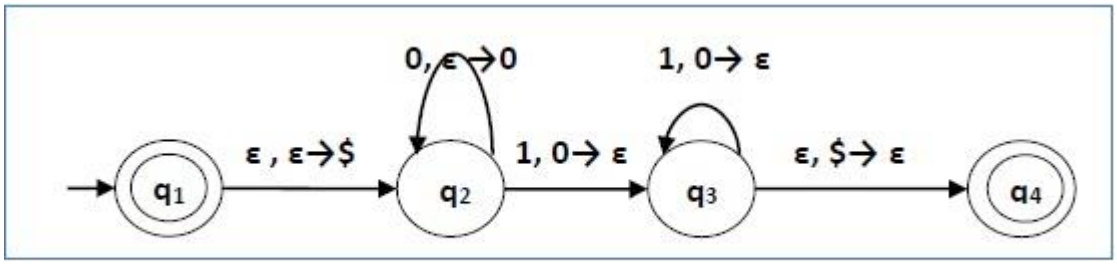
For a PDA (Q, ∑, S, δ, q0, I, F), the language accepted by the empty stack is −

$$L(PDA) = \{w \mid (q0, w, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$$

Example

Construct a PDA that accepts L = {0n 1n | n ≥ 0}

Solution



PDA for L= {0ⁿ 1ⁿ | n≥0}

This language accepts L = {ε, 01, 0011, 000111,,,,}

Here, in this example, the number of 'a' and 'b' have to be same.

Initially we put a special symbol '$' into the empty stack.

Then at state q2, if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0.

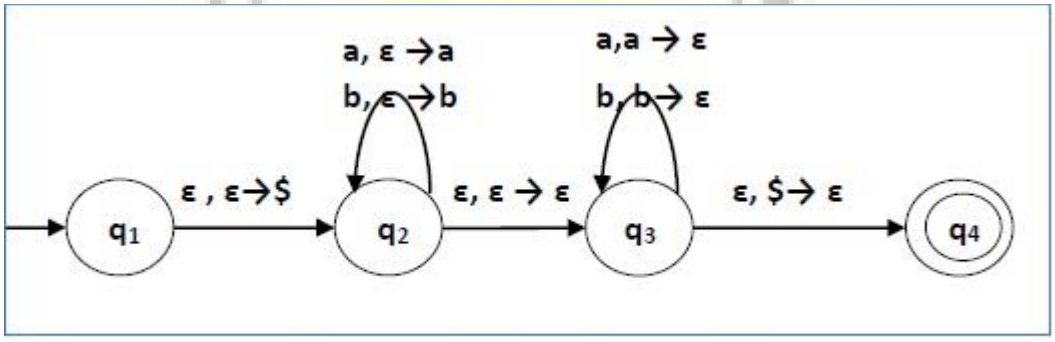
Then at state q3, if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element.

If the special symbol '$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state q4.

Example

Construct a PDA that accepts L = { wwR | w = (a+b)* }

Solution



PDA for L= {wwᴿ | w = (a+b)*}

Initially we put a special symbol '$' into the empty stack. At state q2, the w is being read. In state q3, each 0 or 1 is popped when it matches the input. If any other input is given,

the PDA will go to a dead state. When we reach that special symbol '$', we go to the accepting state q4.

5. Pushdown automata and context-free languages

If a grammar G is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar G. A parser can be built for the grammar G.

Also, if P is a pushdown automaton, an equivalent context-free grammar G can be constructed where

L(G) = L(P)

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F)

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

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

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

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

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

Problem

Construct a PDA from the following CFG.

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

Where the productions are −

S → XS | ε , A → aXb | Ab | ab

Solution

Let the equivalent PDA,

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

where δ −

δ(q, ε , S) = {(q, XS), (q, ε)}

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

$\delta(q, a, a) = \{(q, \varepsilon)\}$

$\delta(q, 1, 1) = \{(q, \varepsilon)\}$

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F) such that the non- terminals of the grammar G will be {Xwx | w,x∈ Q} and the start state will be Aq0,F.

Step 1: For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule Xwx → a Xyzb in grammar G.

Step 2: For every w, x, y, z ∈ Q, add the production rule Xwx → XwyXyx in grammar G.

Step 3: For w ∈ Q, add the production rule Xww → ε in grammar G.

*****