Semester – V

		L	T	P	C
16MMU502B	**BOOLEAN ALGEBRA AND AUTOMATA THEORY**	**6**	**2**	**0**	**6**

Scope: On successful completion of course the learners gain about finite Automata and regular languages.

Objectives: To enable the students to learn and gain knowledge about context free grammars and pushdown automata.

UNIT I

Definition of ordered set with examples and basic properties of ordered sets, maps between ordered sets , duality principle, lattices as ordered sets- lattices as algebraic structures, sublattices, products and homomorphisms, modular and distributive lattices.
Boolean algebras: Boolean polynomials, minimal forms of Boolean polynomials, Quinn-McCluskey method, Karnaugh diagrams, switching circuits and applications of switching circuits.

UNIT II

The central concept of Automata: Alphabets, strings, and languages. Finite Automata and Regular Languages: deterministic and non-deterministic finite automata, regular expressions, regular languages and their relationship with finite automata, pumping lemma and closure properties of regular languages.

UNIT III

Context Free Grammars and Pushdown Automata: Context free grammars (CFG), parse trees, ambiguities in grammars and languages, pushdown automaton (PDA) and the language accepted by PDA, deterministic PDA, Non- deterministic PDA, properties of context free languages, normal forms, pumping lemma, closure properties, decision properties.

UNIT IV

Turing Machines: Turing machine as a model of computation, programming with a Turing machine, variants of Turing machine and their equivalence.

UNIT V

Undecidability: Recursively enumerable and recursive languages, undecidable problems about Turing machines: halting problem, Post Correspondence Problem, and undecidability problems About CFGs.

SUGGESTED READINGS

TEXT BOOKS

1. Davey B A., and Priestley H. A., (2002). Introduction to Lattices and Order, Cambridge University Press, Cambridge. **(For Unit-I)**

2. Hopcroft J. E., Motwani R., and Ullman J.D., (2001). Introduction to Automata Theory, Languages, and Computation, Second Edition, Addison-Wesley, USA. **(For Unit-II to V)**

REFERENCES

1. Edgar G. Goodaire and Michael M. Parmenter, (2003). Discrete Mathematics with Graph Theory, Second Edition, Pearson Education P.Ltd., Singapore.

2. Rudolf Lidl and Günter Pilz, (2004). Applied Abstract Algebra, Second Edition , Undergraduate Texts in Mathematics, Springer (SIE).

3. Lewis H.R., Papadimitriou C.H.,and Papadimitriou C.,(1997). Elements of the Theory of Computation, Second Edition ,Prentice-Hall. New Delhi**.**

4. Anderson J.A., (2006). Automata Theory with Modern Applications, Cambridge University Press, Cambridge.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

LECTURE PLAN

DEPARTMENT OF MATHEMATICS

STAFF NAME: Dr. K.KALIDASS
SUBJECT NAME: BOOLEAN ALGEBRA AND AUTOMATA THEORY
UB.CODE:16MMU502B SEMESTER: V

CLASS: III B. Sc. MATHEMATICS

S. No	Lecture Duration Hour	Topics To Be Covered	Support Materials
UNIT-I			
1	1	Introduction to ordered set	T1: Ch 1,1-3
2	1	Basic properties of ordered sets	R1: Ch 2, 63-67
3	1	Maps between ordered sets	T1: Ch 1, 23-24
4	1	**Tutorial**	
5	1	Duality principle	T1: Ch 1, 25
6	1	lattices as ordered sets	T1: Ch 2, 33-38
7	1	lattices as algebraic structures	T1: Ch 2, 39-40
8	1	**Tutorial**	
9	1	Sublattices	T1: Ch 2, 41-43
10	1	Products and homomorphisms	T1: Ch 2, 44-46
11	1	Modular and distributive lattices	T1: Ch 2, 46-49
12	1	**Tutorial**	
13	1	Theorem Boolean polynomials	T1: Ch 4, 83-88
14	1	Quinn-McCluskey method	T1: Ch 4, 89
15	1	Problems on Karnaugh diagrams	T1: Ch 4, 90-91
16	1	**Tutorial**	
17	1	Problems on minimal forms of Boolean polynomials	R1: Ch 1, 40-50
18	1	Applications of switching circuits	R1: Ch 2, 55-62
19	1	Recapitulation and discussion of possible questions	
Total number of hours planed for unit I 18 hours			
. UNIT-II			
1	1	Alphabets, strings, and languages./**Tutorial**	T2: Ch 1, 28-37
2	1	Finite Automata Languages	R3: Ch 2,28-37

3	1	Regular Languages	T2: Ch 2, 40-45
4	1	non-deterministic finite automata	T2: Ch 2, 55-57
5	1	**Tutorial**	
6	1	non-deterministic finite automata	T2: Ch 2. 58-60
7	1	regular expressions	T2: Ch 3,83-91
8	1	regular expressions	T2: Ch 3, 92-106
9	1	**Tutorial**	
10	1	regular languages & their relationship with finite automata	T2: Ch 3, 107-113
11	1	regular languages & their relationship with finite automata	T2: Ch 3, 114-120
12	1	regular languages & their relationship with finite automata	T2: Ch 4, 126
13	1	**Tutorial**	
14	1	pumping lemma	T2: Ch 4, 127
15	1	pumping lemma	T2: Ch 4, 131-133
16	1	closure properties of regular languages	T2: Ch 4, 133-137
17	1	**Tutorial**	
18	1	closure properties of regular languages	T2: Ch 4, 133-137
19	1	Recapitulation and discussion of possible questions	
Total number of hours planed for unit II 19 hours			
. UNIT-III			
1	1	Context free grammars	T2: Ch 5, 169-179
2	1	parse trees **Tutorial**	T2: Ch 5, 181-204
3	1	parse trees	T2: Ch 5, 205-213
4	1	ambiguities in grammars and languages	
5	1	pushdown automaton	T2: Ch 6, 219-228
6	1	**Tutorial**	
7	1	the language accepted by PDA	T2: Ch 6, 229-236
8	1	deterministic PDA	T2: Ch 6, 247-248
9	1	Non- deterministic PDA	T2: Ch 6, 249-250
10	1	**Tutorial**	
11	1	Non- deterministic PDA	T2: Ch 6, 250-253
12	1	properties of context free languages	T2: Ch 7, 255-266
13	1	properties of context free languages	T2: Ch 7, 266-268
14	1	**Tutorial**	
15	1	normal forms	T2: Ch 7, 274-280
16	1	pumping lemma	T2: Ch 7, 281-302
17	1	closure properties and decision properties	T2: Ch 7, 281-302
18	1	Recapitulation and discussion of possible questions	

. UNIT-IV			
1	1	Turing Machines	T2: Ch 8, 307- 316
2	1	Turing Machines	T2: Ch 8, 318-320
3	1	Turing Machines	T2: Ch 8, 320-328
4	1	**Tutorial**	
5	1	Turing machine as a model of computation	T2: Ch 8, 329-332
6	1	programming with a Turing machine	
7	1	programming with a Turing machine	T2: Ch 8, 333-337
8	1	**Tutorial**	
9	1	programming with a Turing machine	T2: Ch 8, 338-340
10	1	variants of Turing machine	T2: Ch 8, 341-345
11	1	variants of Turing machine	T2: Ch 8, 346-350
12	1	**Tutorial**	
13	1	variants of Turing machine	T2: Ch 8, 351-353
14	1	variants of Turing machine	T2: Ch 8, 354
15	1	Turing machine equivalence	T2: Ch 8, 355
16	1	**Tutorial**	
17	1	Turing machine equivalence	T2: Ch 8, 356
18	1	Turing machine equivalence	
19	1	Recapitulation and discussion of possible questions	
Total number of hours planed for unit IV 9 hours			
UNIT-V			
1	1	Recursively enumerable and recursive languages	T2: Ch 9, 367-370
2	1	Recursively enumerable and recursive languages	T2: Ch 9, 371-373
3	1	Recursively enumerable and recursive languages	T2: Ch 9, 374-376
4	1	Recursively enumerable and recursive languages	T2: Ch 9, 374-376
5	1	**Tutorial**	
6	1	undecidable problems about Turing machines	T2: Ch 9, 378-380
7	1	undecidable problems about Turing machines	T2: Ch 9, 381-386
8	1	undecidable problems about Turing machines	T2: Ch 9, 387-340
9	1	**Tutorial**	
10	1	halting problem	T2: Ch 9, 341-343
11	1	Post Correspondence Problem	T2: Ch 9, 344-346
12	1	undecidability problems about CFGs	T2: Ch 9, 344-346
13	1	**Tutorial**	
14	1	undecidability problems about CFGs	T2: Ch 9, 347-350
15	1	Recapitulation and discussion of possible questions	T2: Ch 9, 352-363
16	1	Discusion of ESE qns	T2: Ch 9, 364-373
17	1	**Tutorial**	
18	1	Discusion of ESE qns	

19	1	Discusion of ESE qns	
20	1	Discusion of ESE qns	
21	1	Discusion of ESE qns/ **Tutorial**	
Total number of hours planed for unit V 21 Hours			

Unit	**Hours(L+T)**
I	**18(14+4)**
II	**20(15+5)**
III	**18(14+4)**
IV	**19(15+4)**
V	**21(15+6)**
Total	**96(73+23)**

UNIT I

Definition of ordered set with examples and basic properties of ordered sets, maps between ordered sets , duality principle, lattices as ordered sets- lattices as algebraic structures, sublattices, products and homomorphisms, modular and distributive lattices. Boolean algebras: Boolean polynomials, minimal forms of Boolean polynomials, Quinn-McCluskey method, Karnaugh diagrams, switching circuits and applications of switching circuits.

Introduction

Partial order and lattice theory now play an important role in many disciplines of computer science and engineering. For example, they have applications in distributed computing (vector clocks, global predicate detection), concurrency theory (pomsets, occurrence nets), programming language semantics (fixed-point semantics), and data mining (concept analysis). They are also useful in other disciplines of mathematics such as combinatorics, number theory and group theory. This book differs from earlier books written on the subject in two aspects. First, the present book takes a computational perspective — the emphasis is on algorithms and their complexity. While mathematicians generally identify necessary and sufficient conditions to characterize a property, this book focuses on efficient algorithms to test the property. As a result of this bias, much of the book concerns itself only with finite sets. Second, existing books do not dwell on applications of lattice theory. This book treats applications at par with the theory. In particular, applications of lattice theory to distributed computing are treated in extensive detail. I have also shown many applications to combinatorics because the theory of partial orders forms a core topic in combinatorics. This chapter covers the basic definitions of partial orders.

A partial order is simply a relation with certain properties. A **relation** R over any set X is a subset of $X \times X$. For example, let

$$X = \{a, b, c\}.$$

Then, one possible relation is

$$R = \{(a, c), (a, a), (b, c), (c, a)\}.$$

It is sometimes useful to visualize a relation as a graph on the vertex set X such that there is a directed edge from x to y iff $(x, y) \in R$. The graph corresponding to the relation R in the previous example is shown in Figure 1.1.

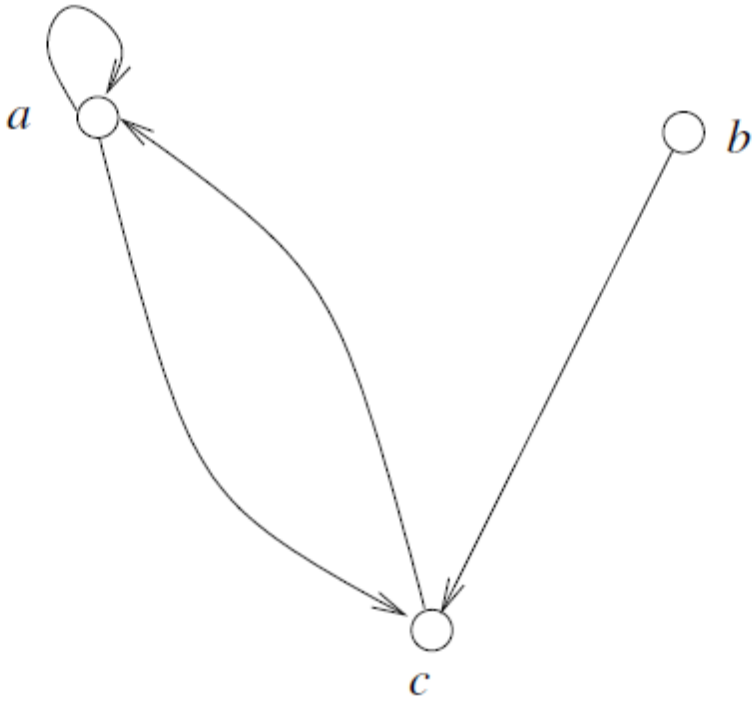


Figure 1.1: The graph of a relation

$$R = \{(x, y) \mid x \ mod\ 5 = y \ mod\ 5\}. \tag{1.1}$$

A symmetric relation can be represented using an undirected graph. R is **antisymmetric** if for all $x, y \in X$, $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$. For example, the relation *less than or equal to* defined on $\mathcal{N}$ is anti-symmetric. A relation R is **asymmetric** if for all $x, y \in X$, $(x, y) \in R$ implies $(y, x) \notin R$. The relation *less than* on $\mathcal{N}$ is asymmetric. Note that an asymmetric relation is always irreflexive.

A relation R is **transitive** if for all $x, y, z \in X$, $(x, y) \in R$ and $(y, z) \in R$ implies $(x, z) \in R$. The relations *less than* and *equal to* on $\mathcal{N}$ are transitive.

A relation is **reflexive** if for each $x \in X$, $(x, x) \in R$. In terms of a graph, this means that there is a self-loop on each node. If X is the set of natural numbers, $\mathcal{N}$, then "x divides y" is a reflexive relation. R is **irreflexive** if for each $x \in X$, $(x, x) \notin R$. In terms of a graph, this means that there are no self-loops. An example on the set of natural numbers, $\mathcal{N}$, is the relation "x less than y." Note that a relation may be neither reflexive nor irreflexive.

A relation R is **symmetric** if for all $x, y \in X$, $(x, y) \in R$ implies $(y, x) \in R$. An example of a symmetric relation on $\mathcal{N}$ is

A relation R is an **equivalence** relation if it is reflexive, symmetric, and transitive. When R is an equivalence relation, we use $x \equiv_R y$ (or simply $x \equiv y$ when R is clear from the context) to denote that $(x, y) \in R$. Furthermore, for each $x \in X$, we use $[x]_R$, called the **equivalence class of** x, to denote the set of all $y \in X$ such that $y \equiv_R x$. It can be seen that the set of all such equivalence classes forms a **partition** of X. The relation on $\mathcal{N}$ defined in (1.1) is an example of an equivalence relation. It partitions the set of natural numbers into five equivalence classes.

Given any relation R on a set X, we define its irreflexive transitive closure, denoted by R^+, as follows. For all $x, y \in X : (x, y) \in R^+$ iff there exists a sequence $x_0, x_1, ..., x_j, j \geq 1$ with $x_0 = x$ and $x_j = y$ such that

$$\forall i : 0 \leq i < j : (x_i, x_{i+1}) \in R.$$

Thus $(x, y) \in R^+$ iff there is a nonempty path from x to y in the graph of the relation R. We define the reflexive transitive closure, denoted by R^*, as

$$R^* = R^+ \cup \{(x, x) \mid x \in X\}$$

A relation is a **total order** if R is a partial order and for all distinct $x, y \in X$, either $(x, y) \in R$ or $(y, x) \in R$. The natural order on the set of integers is a total order, but the "divides" relation is only a partial order.

A relation R is a **reflexive partial order** (or, a **non-strict partial order**) if it is reflexive, antisymmetric, and transitive. The *divides* relation on the set of natural numbers is a reflexive partial order. A relation R is an **irreflexive partial order**, or a **strict partial order** if it is irreflexive and transitive. The *less than* relation on the set of natural numbers is an irreflexive partial order. When R is a reflexive partial order, we use $x \leq_R y$ (or simply $x \leq y$ when R is clear from the context) to denote that $(x, y) \in R$. A reflexive partially ordered set, **poset** for short, is denoted by $(X, \leq)$. When R is an irreflexive partial order, we use $x <_R y$ (or simply $x < y$ when R is clear from the context) to denote that $(x, y) \in R$. The set X together with the partial order is denoted by $(X, <)$. We use $P = (X, <)$ to denote the poset.

The two versions of partial orders — reflexive and irreflexive — are essentially the same. Given an irreflexive partial order, we can define $x \leq y$ as $x < y$ or $x = y$ which gives us a reflexive partial order. Similarly, given a reflexive partial order $(X, \leq)$, we can define an irreflexive partial order $(X, <)$ by defining $x < y$ as $x \leq y$ and $x \neq y$. In this book, we use a *poset* to mean a set X with either an irreflexive partial order or a reflexive partial order.

Finite posets are often depicted graphically using **Hasse diagrams**. To define Hasse diagrams, we first define a relation **covers** as follows. For any two elements $x, y \in X$, y covers x if $x < y$ and $\forall z \in X : x \leq z < y$ implies $z = x$. In other words, there should not be any element z with $x < z < y$. We use $x \prec y$ to denote that y covers x (or x is covered by y). We also say that y is an **upper cover** of x and x is a **lower cover** of y. A Hasse diagram of a poset is a graph with the property that there is an edge from x to y iff $x \prec y$. Furthermore, when drawing the graph on a Euclidean plane, x is drawn lower than y when y covers x. This allows us to suppress the directional arrows in the edges. For example, consider the following poset $(X, \leq)$,

$$X \stackrel{\text{def}}{=} \{p, q, r, s\}; \quad \leq \stackrel{\text{def}}{=} \{(p, q), (q, r), (p, r), (p, s)\}. \tag{1.2}$$

Its Hasse diagram is shown in Figure 1.2. Note that we will sometimes use directed edges in Hasse diagrams if the context demands it. In general, in this book, we switch between the directed graph and undirected graph representations of Hasse diagrams.

Given a poset $(X, \leq_X)$ a **subposet** is simply a poset $(Y, \leq_Y)$ where $Y \subseteq X$, and

$$\forall x, y \in Y : x \leq_Y y \stackrel{\text{def}}{=} x \leq_X y$$

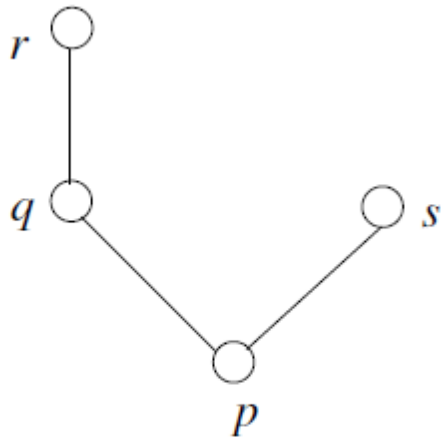


Figure 1.2: Hasse diagram

Let $x, y \in X$ with $x \neq y$. If either $x < y$ or $y < x$, we say x and y are **comparable**. On the other hand, if neither $x < y$ nor $x > y$, then we say x and y are incomparable, and write $x||y$. A poset $(Y, \leq)$ (or a subposet $(Y, \leq)$ of $(X, \leq)$) is called a **chain** if every distinct pair of elements from Y is comparable. Similarly, we call a poset an **antichain** if every distinct pair of elements from Y is incomparable. For example, for the poset represented in Figure 1.2, $\{q, r\}$ is a chain, and $\{q, s\}$ is an antichain.

A chain C of a poset $(X, \leq)$ is a **maximum chain** if no other chain contains more elements than C. We use a similar definition for **maximum antichain**. The **height** of the poset is the number of elements in the maximum chain, and the **width** of the poset is the number of elements in a maximum antichain.For example, the poset in Figure 1.2 has height equal to 3 (the maximum chain is $\{p, q, r\}$) and width equal to 2 (a maximum antichain is $\{q, s\}$).

We now define two operators on subsets of the set X—**meet or infimum** (or **inf**) and **join** or **supremum** (or **sup**). Let $Y \subseteq X$, where $(X, \leq)$ is a poset. For any $m \in X$, we say that $m = inf\ Y$ iff

1. $\forall y \in Y : m \leq y$, and
2. $\forall m' \in X : (\forall y \in Y : m' \leq y) \Rightarrow m' \leq m$.

The condition (1) says that m is a lower bound of the set Y. The condition (2) says that if m' is another lower bound of Y, then it is less than m. For this reason, m is also called the **greatest lower bound** (*glb*) of the set Y. It is easy to check that the infimum of Y is unique whenever it exists. Observe that m is not required to be an element of Y.

The definition of *sup* is similar. For any $s \in X$, we say that $s = sup\ Y$ iff

1. $\forall y \in Y : y \leq s$
2. $\forall s' \in X : (\forall y \in Y : y \leq s') \Rightarrow s \leq s'$

Again, s is also called the **least upper bound** (*lub*) of the set Y. We denote the **glb** of $\{a, b\}$ by $a \sqcap b$, and **lub** of $\{a, b\}$ by $a \sqcup b$. In the set of natural numbers ordered by the *divides* relation, the *glb* corresponds to finding the greatest common divisor (gcd) and the *lub* corresponds to finding the least common multiple of two natural numbers. The greatest lower bound or the least upper bound may not always exist. In Figure 1.3, the set $\{e, f\}$ does not have any upper bound. In the third poset in Figure 1.4, the set $\{b, c\}$ does not have any least upper bound (although both d and e are upper bounds).

The following lemma relates $\leq$ to the meet and join operators.

Lemma 1.1 *[Connecting Lemma]*

1. $x \leq y \equiv (x \sqcup y) = y$, *and*
2. $x \leq y \equiv (x \sqcap y) = x$.

Proof:

$x \leq y$ implies that y is an upper bound on $\{x, y\}$. y is also the least upper bound because any upper bound of $\{x, y\}$ is greater than both x and y. Therefore, $(x \sqcup y) = y$. Conversely, $(x \sqcap y) = y$ means y is an upper bound on $\{x, y\}$. Therefore, $x \leq y$.

The proof for the second part is the dual of this proof.

■

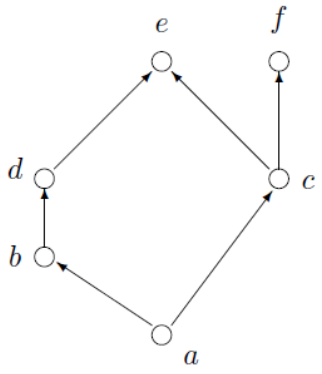
Definition 1.2 (Lattice) *A poset $(X, \leq)$ is a* **lattice** *iff $\forall x, y \in X : x \sqcup y$ and $x \sqcap y$ exist.*

Definition 1.3 (Distributive Lattice) *A lattice L is distributive if*

$$\forall a, b, c \in L : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

$$X = \{a, b, c, d, e, f\}$$

$$R = \left\{ \begin{array}{l} (a,b), (a,c), (b,d), \\ (c,f), (c,e), (d,e) \\ (a,d), (a,e), (a,f), \\ (b,e) \end{array} \right\}$$



Generalizing the notation for intervals on the real-line, we define an **interval** $[x, y]$ in a poset $(X, \leq)$ as

$$\{z | x \leq z \leq y\}$$

The meaning of (x, y) and $[x, y)$ and $(x, y]$ is similar. A poset is **locally finite** if all intervals are finite. Most posets in this book will be locally finite if not finite.

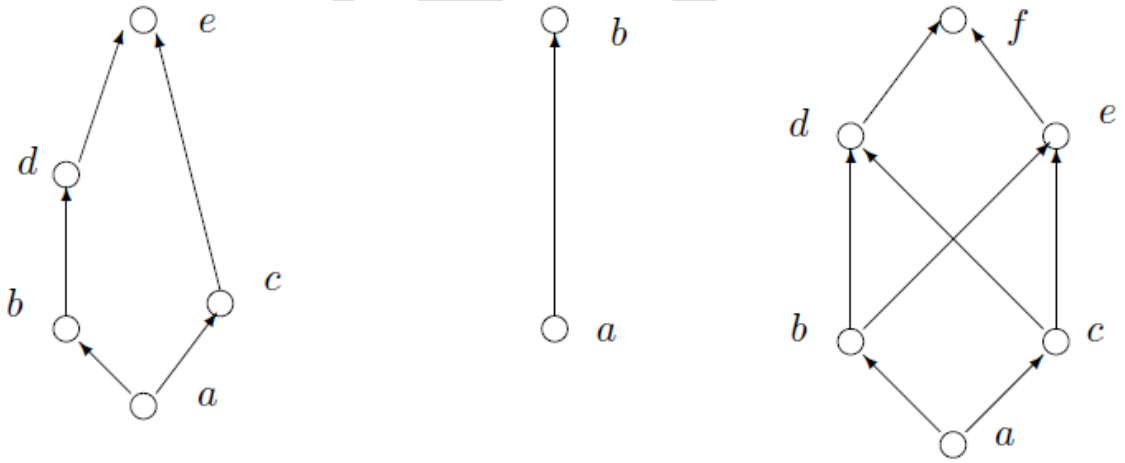


Figure 1.4: Various posets.

A poset is **well-founded** iff it has no infinite decreasing chain. The set of natural numbers under the usual relation is well-founded but the set of integers is not well-founded. A poset corresponding to a distributed computation may be infinite but is usually well-founded.

Poset $Q = (X, \leq_Q)$ extends the poset $P = (X, \leq_P)$ if

$$\forall x, y \in X : x \leq_P y \Rightarrow x \leq_Q y$$

If Q is a total order, then we call Q a linear extension of P. For example, for the poset $(X, \leq)$ defined in Figure (1.2), a possible **linear extension** Q is

$$X \stackrel{\text{def}}{=} \{p, q, r, s\}; \quad \leq_Q \stackrel{\text{def}}{=} \{(p, q), (q, r), (p, r), (p, s), (q, s), (r, s)\}.$$

Let $(X, \leq)$ be any poset. We call a subset $Y \subseteq X$ a **down-set** if

$$\forall y, z \in X : z \in Y \wedge y \leq z \Rightarrow y \in Y.$$

Down-sets are also called **order ideals**. It is easy to see that for any x, $D[x]$ defined below is an order ideal. Such order ideals are called **principal order ideals**,

$$D[x] = \{y \in X | y \leq x\}.$$

For example, in Figure 1.2, $D[r] = \{p, q, r\}$. Similarly, we call $Y \subseteq X$ an **up-set** if

$$y \in Y \wedge y \leq z \Rightarrow z \in Y.$$

Up-sets are also called **order filters**. We also use the notation below to denote **principal order filters**

$$U[x] = \{y \in X | x \leq y\}.$$

In Figure 1.2, $U[p] = \{p, q, r, s\}$.

The following lemmas provides a convenient relationship between principal order filters and other operators defined earlier.

Lemma 1.4 1. $x \leq y \equiv U[y] \subseteq U[x]$

2. $x = sup(Y) \equiv U[x] = \cap_{y \in Y} U[y]$

In some applications, the following notation is also useful:

$$U(x) = \{y \in X | x < y\}$$

$$D(x) = \{y \in X | y < x\}.$$

We will call $U(x)$ the **upper-holdings** of x, and $D(x)$, the **lower-holdings** of x. We extend the definitions of $D[x], D(x), U[x], U(x)$ to sets of elements, A. For example,

$$U[A] = \{y \in X | \exists x \in A : x \leq y\}.$$

An element x is a **bottom** element or a **minimum** element of a poset P if $x \in P$, and

$$\forall y \in P : x \leq y$$

For example, 0 is the bottom element in the poset of whole numbers, and $\emptyset$ is the bottom element in the poset of all subsets of a given set W. Similarly, an element x is a **top** element, or a **maximum** element of a poset P if $x \in P$, and

$$\forall y \in P : y \leq x$$

A bottom element of the poset is denoted by $\perp$ and the top element by $\top$. It is easy to verify that if bottom and top elements exist, they are unique.

An element x is a **minimal** element of a poset P if

$$\forall y \in P : y \not< x.$$

The minimum element is also a minimal element. However, a poset may have more than one minimal element. Similarly, an element x is a **maximal** element of a poset P if

$$\forall y \in P : y \not> x.$$

Definition 1.5 *An element x is join-irreducible in P if it cannot be expressed as join of other elements of P. Formally, x is* **join-irreducible** *if*

$$\forall Y \subseteq P : x = sup(Y) \Rightarrow x \in Y$$

Theorem 1.6 *Let P be a finite poset. Then, for any $x \in P$*

$$x = sup(D[x] \cap \mathcal{J}(P))$$

Proof: We use $I[x]$ to denote the set $D[x] \cap \mathcal{J}(P)$. We need to show that $x = sup(I[x])$ for any x. The proof is by induction on the cardinality of $D[x]$.

(Base case) $D[x]$ is singleton.
This implies that x is a minimal element of the poset. If x is the unique minimum, $x \notin \mathcal{J}(P)$ and $I[x]$ is empty; therefore, $x = sup(I[x]$. Otherwise, $x \in \mathcal{J}(P)$ and we get that $x = sup(I[x])$.

(Induction case) $D[x]$ is not singleton.
First assume that x is join-irreducible. Then, $x \in I[x]$ and all other elements in $I[x]$ are smaller than x. Therefore, x equals $sup(I[x])$. If x is not join-irreducible, then $x = sup(D(x))$. By induction, each $y \in D(x)$ satisfies $y = sup(I[y])$. Therefore, we have

$$\forall y \in D(x) : U[y] = \cap_{z \in I[y]} U[z] \tag{1.3}$$

We now have,

$x = sup(D(x))$
$\equiv$ { property of sup }
$U[x] = \cap_{y \in D(x)} U[y]$
$\equiv$ { Equation 1.3 }
$U[x] = \cap_{y \in D(x)} \cap_{z \in I[y]} U[z]$
$\equiv$ { definition of $I[y]$ }
$U[x] = \cap_{y \in D(x)} \cap_{z \in (D[y] \cap \mathcal{J}(P))} U[z]$
$\equiv$ { definition of $I[x]$ and simplification }
$U[x] = \cap_{z \in I[x]} U[z]$
$\equiv$ { property of sup }
$x = sup(I[x])$

■

Theorem 1.7 *For a finite poset P, $x \in \mathcal{J}(P)$ iff $\exists y \in P : x \in minimal(P - D[y])$*

Proof: First assume that $x \in \mathcal{J}(P)$.

Let $LC(x)$ be the set of elements covered by x. If $LC(x)$ is singleton, then choose that element as y. It is clear that $x \in minimal(P - D[y])$.

Now consider the case when $LC(x)$ is not singleton (it is empty or has more than one element). Let Q be the set of upper bounds for $LC(x)$. Q is not empty because $x \in Q$. Further, x is not the minimum element in Q because x is join-irreducible. Pick any element $y \in Q$ that is incomparable to x. Since $D[y]$ includes $LC(x)$ and not x, we get that x is minimal in $P - D[y]$.

The converse is left as an exercise.

■

Definition 1.8 *For a poset P, $x \in P$ is an upper dissector if there exists $y \in P$ such that*

$$P - U[x] = D[y]$$

Theorem 1.9 *x is a dissector implies that x is join-irreducible.*

Proof: If x is an upper dissector, then there exists y such that x is *minimum* in $P - D[y]$. This implies that x is minimal in $P - D[y]$.

S is a sublattice of a given lattice $L = (X, \leq)$ iff it is non-empty, and:

$$\forall a, b \in S : sup(a, b) \in S \wedge inf(a, b) \in S.$$

Note that the *sup* and inf of any two elements in the sublattice S must be the same as the *sup* and inf of those elements in the original lattice L.

For S to be a sublattice, S being a subset of a lattice is not sufficient. In addition to S being a subset of a lattice, *sup* and inf operations must be inherited from the lattice.

We have so far looked at a lattice as a special type of poset, $P = (X, \leq)$, where the operator $\leq$ is reflexive, antisymmetric and transitive. We have defined the operations of $\sqcup$ and $\sqcap$ on lattices based on the given $\leq$ relation.

An alternative method of studying lattices is to start with a set equipped with $\sqcup$ and $\sqcap$ operator and define $\leq$ relation based on these operations. Consider any set X with two algebraic operators $\sqcup$ and $\sqcap$. Assume that the operators satisfy the following properties:

$(L1)$	$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$	(associativity)
$(L1)^{\delta}$	$a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$	
$(L2)$	$a \sqcup b = b \sqcup a$	(commutativity)
$(L2)^{\delta}$	$a \sqcap b = b \sqcap a$	
$(L3)$	$a \sqcup (a \sqcap b) = a$	(absorption)
$(L3)^{\delta}$	$a \sqcap (a \sqcup b) = a$	

Theorem 5.2 *Let $(X, \sqcup, \sqcap)$ be a nonempty set with the operators $\sqcup$ and $\sqcap$ which satisfy $(L1)-(L3)$ above. We define the operator $\leq$ on X by:*

$$a \leq b \equiv (a \sqcup b = b)$$

Then,

	a	b	c	d	e	f	g
a	a	a ⊓ b	a ⊓ c	a ⊓ d	a ⊓ e	a ⊓ f	a ⊓ g
b	b ⊔ a	b	b ⊓ c	b ⊓ d	b ⊓ e	b ⊓ f	b ⊓ g
c	c ⊔ a	c ⊔ b	c	c ⊓ d	c ⊓ e	c ⊓ f	c ⊓ g
d	d ⊔ a	d ⊔ b	d ⊔ c	d	d ⊓ e	d ⊓ f	d ⊓ g
e	e ⊔ a	e ⊔ b	e ⊔ c	e ⊔ d	e	e ⊓ f	e ⊓ g
f	f ⊔ a	f ⊔ b	f ⊔ c	f ⊔ d	f ⊔ e	f	f ⊓ g
g	g ⊔ a	g ⊔ b	g ⊔ c	g ⊔ d	g ⊔ e	g ⊔ f	g

- $\leq$ *is reflexive, antisymmetric and transitive*
- $sup(a,b) = a \sqcup b$
- $inf(a,b) = a \sqcap b$.

Proof: Left as an exercise.

■

Theorem 5.3 *For a lattice L,*

$$n - 1 \leq e_< \leq n^{3/2}$$

We prove that $e_< \leq n^{3/2}$.

Proof: The lower bound is clear because every element in the lattice has at least one lower cover (except the smallest element). We show the upper bound.

Let L be an inf-semilattice. Consider two distinct elements x, $y \in L$. Let $B(x)$ denote the set of elements covered by x. $B(x) \bigcap B(y)$ cannot have more than one element because that would violate inf-semilattice property. Let $B'(x) = B(x) \bigcup \{x\}$. Hence,

$$|B'(x) \bigcap B'(y)| \leq 1$$

Let

$$L = \{x_0, \ldots, x_{n-1}\}$$

$$b_i = |B(x_i)|$$

Because there is no pair in common between $B'(x)$ and $B'(y)$ for distinct x and y, we get

$$\sum_{i=0}^{n-1} \binom{b_i + 1}{2} \leq \binom{n}{2}$$

Simplifying, we get

$$\sum_{i=0}^{n-1} (b_i^2 + b_i) \leq n^2 - n$$

Dropping b_i from the left hand side and $-n$ from the right hand side, we get

$$\sum_{i=0}^{n-1} b_i^2 < n^2$$

Since $e_< = \sum b_i$, we get

$$\left(\frac{e_<}{n}\right)^2 = \left(\frac{\sum b_i}{n}\right)^2 \leq \frac{\sum b_i^2}{n} \leq \frac{n^2}{n} = n$$

The first inequality follows from Cauchy-Schwarz inequality.

Therfore,

$$e_<^2 \leq n^3$$

■

Example 1: In the lattice shown in Figure 5.4, x and y are the only join-irreducible elements.

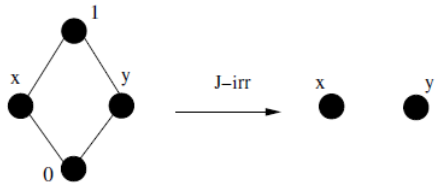


Figure 5.4: Join-irreducible elements: $J(L) = \{x, y\}$

Lemma 5.4 *Let L be a finite lattice. The following two are equivalent.*

1. $\forall a, b \in L : x = (a \sqcup b) \Rightarrow (x = a) \vee (x = b)$
2. $\forall a, b \in L : (a < x) \wedge (b < x) \Rightarrow (a \sqcup b < x)$

Proof: ($\Rightarrow$) Assume x is join-irreducible. In general, we know:
$\forall a, b \in L : (a < x) \wedge (b < x) \Rightarrow (a \sqcup b \leq x) \Rightarrow (a \sqcup b < x) \vee (a \sqcup b = x)$

If $(a \sqcup b = x)$, then since x is join-irreducible, $x = a$ or $x = b$ must be true, which is clearly a contradiction to our assumption in (2). Therefore, $(a \sqcup b < x)$ has to be true.

($\Leftarrow$) Assume $(x = a \sqcup b)$. This means, $x \geq a \wedge x \geq b$. It is given that $(x > a) \wedge (x > b) \Rightarrow (x > a \sqcup b)$,
which is a contradiction to our assumption in (1). So either $(x = b)$ or $(x = a)$ must be true.

■

Lemma 3 *For a finite lattice L, $a \leq b$ is equivalent to $\forall x : x \in J(L) : x \leq a \Rightarrow x \leq b$.*

Proof: For the forward direction, $a \leq b$ and $x \leq a$ implies, for any x, $x \leq b$ by transitivity. For the reverse direction, denote by $h(z)$ the *height* of z, i.e. the length (number of edges) of the *longest* path from z to $\inf L$ in the cover graph (well-defined, since L is a finite lattice). We will prove the property

$$P(a) \;:=\; \forall b : ((\forall x : x \in J(L) : x \leq a \Rightarrow x \leq b) \Rightarrow a \leq b)$$

for all $a \in L$ by *strong* induction on $h(a)$. Given a, consider an arbitrary b and assume

(LHS) $\forall x : x \in J(L) : x \leq a \Rightarrow x \leq b$ and (IH) $P(c)$ holds for all c with $h(c) < h(a)$.

(1) If $h(a) = 0$, then $a = \inf L \leq b$, and $P(a)$ is vacuously true.
(2) If a is join-irreducible, then, using $x := a$ in (LHS), $a \leq b$ follows, and $P(a)$ is again vacuously true.
(3) Now assume $h(a) > 0$ and a not join-irreducible. Then there exist $c \neq a, d \neq a$ such that $a = c \sqcup d$. Since $c \neq a$, we can conclude that $h(a) \geq h(c) + 1$ (h measures *longest* paths!). By (IH), $P(c)$ holds, i.e. $\forall b : ((\forall x : x \in J(L) : x \leq c \Rightarrow x \leq b) \Rightarrow c \leq b)$. We will use $P(c)$ to show $c \leq b$: assume $x \in J(L)$ with $x \leq c$, then $x \leq c \sqcup d = a$, hence $x \leq a$, thus, by (LHS), $x \leq b$. Property $P(c)$ delivers that $c \leq b$. Similarly, one can derive $d \leq b$, hence $c \sqcup d \leq b$, and with $a = c \sqcup d$ we obtain $a \leq b$.

■

Lemma 5.5 *For a finite lattice L and any $a \in L$,*

$$a = \bigsqcup \{x \in J(L) : x \leq a\}.$$

Proof: Let $T = \{x \in J(L) : x \leq a\}$. We have to show that $a = \text{lub}(T)$.
Since any $x \in T$ satisfies $x \leq a$, a is an upper bound on T. Consider *any* upper bound u:

u is an upper bound on T
$\Leftrightarrow$ ⟨Definition of upper bound⟩
$x \in T \Rightarrow x \leq u$
$\Leftrightarrow$ ⟨Definition of T⟩
$(x \in J(L) \wedge x \leq a) \Rightarrow x \leq u$
$\Leftrightarrow$ ⟨Elementary propositional logic: $(a \wedge b) \Rightarrow c \equiv a \Rightarrow (b \Rightarrow c)$⟩
$x \in J(L) \Rightarrow (x \leq a \Rightarrow x \leq u)$
$\Leftrightarrow$ ⟨Lemma 3⟩
$a \leq u$,

so a is in fact the least upper bound on T.

■

Lemma 5.6 *In a finite distributive lattice (FDL) L, an element x is join-irreducible if and only if*

$$x \neq \inf L \quad \textit{and} \quad \forall a, b \in L : x \leq a \sqcup b \Rightarrow (x \leq a \vee x \leq b). \tag{5.1}$$

Proof: If x is join-irreducible, then $x \neq \inf L$ by definition, and

$x \leq a \sqcup b$
$\Leftrightarrow$ ⟨property of $\leq$ and $\sqcap$⟩
$x = x \sqcap (a \sqcup b)$
$\Leftrightarrow$ ⟨L distributive⟩
$x = (x \sqcap a) \sqcup (x \sqcap b)$
$\Rightarrow$ ⟨x join-irreducible⟩
$x = x \sqcap a \vee x = x \sqcup b$
$\Leftrightarrow$ ⟨property of $\leq$ and $\sqcap$⟩
$x \leq a \vee x \leq b$.

Conversely, assume (9.2), and let $x = a \sqcup b$. Then $x \leq a \sqcup b$, hence $x \leq a \vee x \leq b$. On the other hand, $x = a \sqcup b$ implies $x \geq a \wedge x \geq b$. From the last two, since $\vee$ distributes over $\wedge$, it follows that $x = a \vee x = b$.

■

UNIT II

The central concept of Automata: Alphabets, strings, and languages. Finite Automata and Regular Languages: deterministic and non-deterministic finite automata, regular expressions, regular languages and their relationship with finite automata, pumping lemma and closure properties of regular languages.

Formal language

The alphabet of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed; frequently it is required to be finite. The strings formed from this alphabet are called words, and the words that belong to a particular formal language are sometimes called *well-formed words* or *well-formed formulas*. 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.

The field of **formal language theory** studies the purely syntactical aspects of such languages—that is, their internal structural patterns. Formal language theory sprang out of linguistics, as a way of understanding the syntactic regularities of natural languages. In computer science, formal languages are often used as the basis for defining programming languages and other systems in which the words of the language are associated with particular meanings or semantics.

A formal languageL over an alphabet Σ is a subset of Σ^*, that is, a set of words over that alphabet.

In computer science and mathematics, which do not usually deal with natural languages, the adjective "formal" is often omitted as redundant.

While formal language theory usually concerns itself with formal languages that are described by some syntactical rules, the actual definition of the concept "formal language" is only as above: a (possibly infinite) set of finite-length strings, no more nor less. In practice, there are many languages that can be described by rules, such as regular languages or context-free languages. The notion of a formal grammar may be closer to the intuitive concept of a "language," one described by syntactic rules.

Formal language

A formal grammar (sometimes simply called a grammar) is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are

valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form.

A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting must start. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer"—a function in computing that determines whether a given string belongs to the language or is grammatically incorrect. To describe such recognizers, formal language theory uses separate formalisms, known as automata theory. One of the interesting results of automata theory is that it is not possible to design a recognizer for certain formal languages.

Alphabet

An alphabet, in the context of formal languages, can be any set, although it often makes sense to use an alphabet in the usual sense of the word, or more generally a character set such as ASCII. Alphabets can also be infinite; e.g. first-order logic is often expressed using an alphabet which, besides symbols such as^, ¬, □ and parentheses, contains infinitely many elements x0, x1, x2, … that play the role of variables. The elements of an alphabet are called its letters.

word

A word over an alphabet can be any finite sequence, or string, of letters. The set of all words over an alphabet Σ is usually denoted by Σ* (using the Kleene star). For any alphabet there is only one word of length 0, the empty word, which is often denoted by e, ε or λ. By concatenation one can combine two words to form a new word, whose length is the sum of the lengths of the original words. The result of concatenating a word with the empty word is the original word.

Operations on languages

Certain operations on languages are common. This includes the standard set operations, such as union, intersection, and complement. Another class of operation is the element-wise application of string operations.

Examples: suppose L1 and L2 are languages over some common alphabet.

- The concatenation L1L2 consists of all strings of the form vw where v is a string from L1 and w is a string from L2.
- The intersection L1 ∩ L2 of L1 and L2 consists of all strings which are contained in both languages
- The complement ¬L of a language with respect to a given alphabet consists of all strings over the alphabet that are not in the language.
- The Kleene star: the language consisting of all words that are concatenations of 0 or more words in the original language;
- Reversal:
  - Let e be the empty word, then eR = e, and
  - for each non-empty word w = x1…xn over some alphabet, let wR = xn…x1,
  - then for a formal language L, LR = {wR | w □ L}.
- String homomorphism

Such string operations are used to investigate closure properties of classes of languages. A class of languages is closed under a particular operation when the operation, applied to languages in the class, always produces a language in the same class again. For instance, the context-free languages are known to be closed under union, concatenation, and intersection with regular languages, but not closed under intersection or complement. The theory of trios and abstract families of languages studies the most common closure properties of language families in their own right.

Language

"A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols.In general, if $\Sigma$is an alphabet and L is a subset of Σ^*, then L is said to be a *language*over Σ, or simply a language if$\Sigma$is understood. Each element of L is said to be a *sentence*or a *word* or a *string*of the language.

Example 1 ***{0, 11, 001}, {ε, 10}, and {0, 1}* are subsets of {0, 1}*, and so they are languages over the alphabet {0, 1}.***

The empty set Ø and the set {ε} are languages over every alphabet. Ø is a language that contains no string. {ε} is a language that contains just the empty string.

The *union* of two languages L_1 and L_2, denoted $L_1 \cup L_2$, refers to the language that consists of all the strings that are either in L_1 or in L_2, that is, to { x | x is in L_1 or x is in L_2 }. The *intersection* of L_1 and L_2, denoted $L_1 \cap L_2$, refers to the language that consists of all the strings that are both in L_1 and L_2, that is, to { x | x is in L_1 and in L_2 }. The *complementation* of a language L over Σ, or just the complementation of L when Σ is understood, denoted $\overline{L}$, refers to the language that consists of all the strings over Σ that are not in L, that is, to { x | x is in Σ^* but not in L }.

Example 2 ***Consider the languages L_1 = {ε, 0, 1} and L_2 = {ε, 01, 11}. The union of these languages is $L_1 \cup L_2$ = {ε, 0, 1, 01, 11}, their intersection is $L_1 \cap L_2$ = {ε}, and the complementation of L_1 is $\overline{L_1}$ = {00, 01, 10, 11, 000, 001, . . . }.***

$\emptyset \cup L = L$ for each language L. Similarly, $\emptyset \cap L = \emptyset$ for each language L. On the other hand, $\overline{\emptyset} = \Sigma^*$ and $\overline{\Sigma^+} = \emptyset$ for each alphabet Σ.

The *difference* of L_1 and L_2, denoted $L_1 - L_2$, refers to the language that consists of all the strings that are in L_1 but not in L_2, that is, to { x | x is in L_1 but not in L_2 }. The *crossproduct* of L_1 and L_2, denoted $L_1 \times L_2$, refers to the set of all the pairs (x, y) of strings such that x is in L_1 and y is in L_2, that is, to the relation { (x, y) | x is in L_1 and y is in L_2 }. The *composition* of L_1 with L_2, denoted L_1L_2, refers to the language { xy | x is in L_1 and y is in L_2 }.

Example 3 *If L_1 = {λ, 1, 01, 11} and L_2 = {1, 01, 101} then $L_1 - L_2$ = {λ, 11} and $L_2 - L_1$ = {101}.*

On the other hand, if $L_1 = \{\lambda, 0, 1\}$ and $L_2 = \{01, 11\}$, then the cross product of these languages is $L_1 \times L_2 = \{(\lambda, 01), (\lambda, 11), (0, 01), (0, 11), (1, 01), (1, 11)\}$, and their composition is $L_1L_2 = \{01, 11, 001, 011, 101, 111\}$.

$L - \emptyset = L$, $\emptyset - L = \emptyset$, $\emptyset L = \emptyset$, and $\{\lambda\}L = L$ for each language L.

L^i will also be used to denote the composing of i copies of a language L, where L^0 is defined as $\{\varepsilon\}$. The set $L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$, called the *Kleeneclosure* or just the *closure* of L, will be denoted by L*. The set $L^1 \cup L^2 \cup L^3 \ldots$, called the *positiveclosure* of L, will be denoted by L^+.

L^i consists of those strings that can be obtained by concatenating i strings from L. L* consists of those strings that can be obtained by concatenating an arbitrary number of strings from L.

Example 4 ***Consider the pair of languages $L_1 = \{\varepsilon, 0, 1\}$ and $L_2 = \{01, 11\}$. For these languages $L_1^2 = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$, and $L_2^3 = \{010101, 010111, 011101, 011111, 110101, 110111, 111101, 111111\}$. In addition, ε is in L_1^*, in L_1^+, and in L_2^* but not in L_2^+.***

The operations above apply in a similar way to relations in $\Sigma^* \times \Delta^*$, when Σ and Δ are alphabets. Specifically, the *union* of the relations R_1 and R_2, denoted $R_1 \cup R_2$, is the relation $\{ (x, y) \mid (x, y)$ is in R_1 or in $R_2 \}$. The *intersection* of R_1 and R_2, denoted $R_1 \cap R_2$, is the relation $\{ (x, y) \mid (x, y)$ is in R_1 and in $R_2 \}$. The *composition* of R_1 with R_2, denoted R_1R_2, is the relation $\{ (x_1x_2, y_1y_2) \mid (x_1, y_1)$ is in R_1 and (x_2, y_2) is in $R_2 \}$.

Grammar

It is often convenient to specify languages in terms of grammars. The advantage in doing so arises mainly from the usage of a small number of rules for describing a language with a large number of sentences. For instance, the possibility that an English sentence consists of a subject phrase followed by a predicate phrase can be expressed by a grammatical rule of the form <sentence>→<subject><predicate>. (The names in angular brackets are assumed to belong to the grammar metalanguage.) Similarly, the possibility that the subject phrase consists of a noun phrase can be expressed by a grammatical rule of the form <subject>→<noun>.

G is defined as a mathematical system consisting of a quadruple <N, Σ, P, S>, where

N : is an alphabet, whose elements are called nonterminalsymbols.

Σ : is an alphabet disjoint from N, whose elements are called terminalsymbols.

P :is a relation of finite cardinality on $(N \cup \Sigma)^*$, whose elements are called productionrules.

Moreover, each production rule (α, β) in P, denoted $\alpha \to \beta$, must have at least one nonterminal

symbol in α. In each such production rule, $\alpha$is said to be the left-handside of the production rule, and $\beta$is said to be the right-handside of the production rule.

S is a symbol in N called the start, or sentence, symbol.

Types of grammars

Prescriptive: prescribes authoritative norms for a language

Descriptive: attempts to describe actual usage rather thanenforce arbitrary rules

Formal: a precisely defined grammar, such as context-free

Generative: a formal grammar that can generate naturallanguage expressions

Chomsky hierarchy of languages.

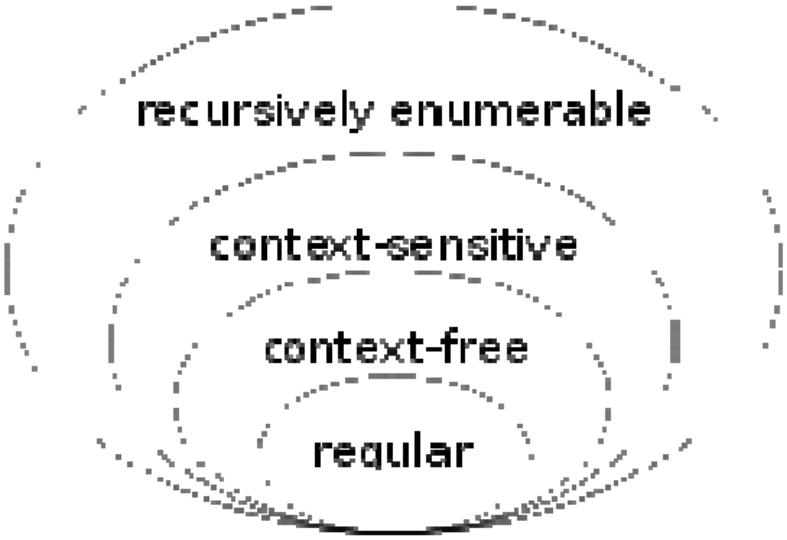
The Chomsky hierarchy consists of the following levels:

Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.



Examples:

1. The language consists of all strings begin with 0.

{0}{0, 1}*

2. The language consists of all strings begin with 0, and end with 1.

{0}{0, 1}*{1}

3. The language consists of all strings with odd lengths.

{0, 1}2n−1, n = 1, 2, . . .

4. The language consists of all strings with substring of three consecutive 0.

{0, 1}*000{0, 1}*

5. The language consists of all strings without substring of three consecutive 0.

{001, 01, 1}*

Regular grammar

A regular grammar is any right-linear or left-linear grammar.

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

B → a - where B is a non-terminal in N and a is a terminal in Σ

B → aC - where B and C are in N and a is in Σ

B → ε - where B is in N and ε denotes the empty string, i.e. the string of length 0.

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

A → a - where A is a non-terminal in N and a is a terminal in Σ

A → Ba - where A and B are in N and a is in Σ

A → ε - where A is in N and ε is the empty string.

An example of a right regular grammar G with N = {S, A}, Σ = {a, b, c}, P consists of the following rules

S → aS

S → bA

A → ε

A → cA

andS is the start symbol. This grammar describes the same language as the regular expression a*bc*.

Extended regular grammars

An extended right regular grammar is one in which all rules obey one of

1. B → a - where B is a non-terminal in N and a is a terminal in Σ
2. A → wB - where A and B are in N and w is in Σ*
3. A → ε - where A is in N and ε is the empty string.

Some authors call this type of grammar a right regular grammar (or right linear grammar) and the type above a strictly right regular grammar (or strictly right linear grammar).

An extended left regular grammar is one in which all rules obey one of

1. A → a - where A is a non-terminal in N and a is a terminal in Σ
2. A → Bw - where A and B are in N and w is in Σ*
3. A → ε - where A is in N and ε is the empty string.

Regular expression

A regular expression (or regexp, or pattern, or RE) is a text string that describes some (mathematical) set of strings. A RE r matches a string s if s is in the set of strings described by r. Regular Expressions have their own notation. Characters are single letters for example 'a', ' '(single blank space), '1' and '-' (hyphen). Operators are entries in a RE that match one or more characters.

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory. Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (**empty set)**∅ denoting the set ∅.
- (**empty string)** ε denoting the set containing only the "empty" string, which has no characters at all.
- (**literal character)** a in Σ denoting the set containing only the character a.

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (**concatenation**) RS denoting the set { αβ | α in set described by expression R and β in set described by S }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (**alternation**) R | S denoting the set union of sets described by R and S. For example, if R describes {"ab", "c"} and S describes {"ab", "d", "ef"}, expression R | S describes {"ab", "c", "d", "ef"}.
- (**Kleene star**) R* denoting the smallest superset of set described by R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by R. For

- (**Kleene star**) R* denoting the smallest superset of set described by R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by R. For

example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity then parentheses may be omitted. For example, (ab)c can be written as abc, and a|(b(c*)) can be written as a|bc*.

Examples:

- a|b* denotes {ε, "a", "b", "bb", "bbb", ...}
- (a|b)* denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}
- ab*(c|ε) denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", ...}

Deterministic finite automaton (D.F.A)

- In the automata theory, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as deterministic finite state machine—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. 'Deterministic' refers to the uniqueness of the computation.
- A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of accept states (denoted graphically by a double circle) which help define when a computation is successful.
- A DFA is defined as an abstract mathematical concept, but due to the deterministic nature of a DFA, it is implementable in hardware and software for solving various specific problems. For example, a DFA can model a software that decides whether or not online user-input such as email addresses are valid.

- DFAs recognize exactly the set of regular languageswhich are, among other things, useful for doing lexical analysis and pattern matching. DFAs can be built from nondeterministic finite automata through the powerset construction.

Formal definition

A deterministic finite automaton M is a 5-tuple, **$(Q, \Sigma, \delta, q_0, F)$** consisting of

- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of accept states ($F \in Q$)

  Let w = a1a2 ... an be a string over the alphabet Σ. The automaton M accepts the string w if a sequence of states, $r_0, r_1, \ldots, r_n$, exists in Q with the following conditions:

- $r_0 = q_0$
- $r_{i+1} = \delta(r_i, a_{i+1})$, for $i = 0, \ldots, n-1$
- $r_n \in F$.

  In words, the first condition says that the machine starts in the start state q_0. The second condition says that given each character of string w, the machine will transition from state to state according to the transition function δ. The last condition says that the machine accepts w if the last input of w causes the machine to halt in one of the accepting states. Otherwise, it is said that the automaton rejects the string. The set of strings M accepts is the language recognized by M and this language is denoted by L(M).

Transition Function Of DFA

A deterministic finite automaton without accept states and without a starting state is known as a transition system or semi automaton.

Given an input symbol $a \in \Sigma$, one may write the transition function as $\delta_a : Q \rightarrow Q$, using the simple trick of currying, that is, writing $\delta(q, a) = \delta_a(q)$ for all $q \in Q$. This way, the transition function can be seen in simpler terms: it's just something that "acts" on a state in Q,

yielding another state. One may then consider the result of function composition repeatedly applied to the various functions δ_a, δ_b, and so on. Using this notion we define

$\hat{\delta} : Q \times \Sigma^* \to Q$. Given a pair of letters $a, b \in \Sigma$, one may define a new function $\hat{\delta}$, by insisting that $\hat{\delta}_{ab} = \delta_a \circ \delta_b$, where $\circ$ denotes function composition. Clearly, this process can be recursively continued. So, we have following recursive definition

$\hat{\delta}(q, \epsilon) = q$ where ϵ is empty string and

$\hat{\delta}(q, wa) = \delta_a(\hat{\delta}(q, w))$ where $w \in \Sigma^*, a \in \Sigma$ and $q \in Q$.

- $\hat{\delta}$ is defined for all words $w \in \Sigma^*$

Advantages and disadvantages

- DFAs were invented to model real world finite state machines in contrast to the concept of a Turing machine, which was too general to study properties of real world machines.
- DFAs are one of the most practical models of computation, since there is a trivial linear time, constant-space, online algorithm to simulate a DFA on a stream of input. Also, there are efficient algorithms to find a DFA recognizing:

1. the complement of the language recognized by a given DFA.
2. the union/intersection of the languages recognized by two given DFAs.

- Because DFAs can be reduced to a canonical form (minimal DFAs), there are also efficient algorithms to determine:

1. whether a DFA accepts any strings
2. whether a DFA accepts all strings
3. whether two DFAs recognize the same language
4. the DFA with a minimum number of states for a particular regular language

- DFAs are equivalent in computing power to nondeterministic finite automata (NFAs). This is because, firstly any DFA is also an NFA, so an NFA can do what a DFA can do. Also, given an NFA, using the powerset construction one can build a DFA that

UNIT III

Context Free Grammars and Pushdown Automata: Context free grammars (CFG), parse trees, ambiguities in grammars and languages, pushdown automaton (PDA) and the language accepted by PDA, deterministic PDA, Non- deterministic PDA, properties of context free languages, normal forms, pumping lemma, closure properties, decision properties..

Context free grammars

A context-free grammar G is defined by the 4-tuple:

G=(V,T,P,S) where

1. V is a finite set; each element $v \in V$ is called a non-terminal character or a variable. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G.
2. T is a finite set of terminals, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G.
3. P is a set of production rule.
4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

Example:

1. $S \rightarrow x$
2. $S \rightarrow y$
3. $S \rightarrow z$
4. $S \rightarrow S + S$
5. $S \rightarrow S - S$
6. $S \rightarrow S * S$
7. $S \rightarrow S / S$
8. $S \rightarrow (S)$

This grammar can, for example, generate the string

(x + y) * x - z * y / (x + x)

as follows:

S (the start symbol)

→ S - S (by rule 5)

→ S * S - S (by rule 6, applied to the leftmost S)

→ S * S - S / S (by rule 7, applied to the rightmost S)

→ (S) * S - S / S (by rule 8, applied to the leftmost S)

→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)

→ (S + S) * S - S / (S) (etc.)

→ (S + S) * S - S * S / (S)

→ (S + S) * S - S * S / (S + S)

→ (x + S) * S - S * S / (S + S)

→ (x + y) * S - S * S / (S + S)

→ (x + y) * x - S * y / (S + S)

→ (x + y) * x - S * y / (x + S)

→ (x + y) * x - z * y / (x + S)

→ (x + y) * x - z * y / (x + x)

Problem 1. Give a context-free grammar for the language

$L = \{a^n b^m : n \neq 2m\}$.

Is your grammar ambiguous?

Ans:

A grammar for the language is

S →aaSb | A | B | ab

A →aA | a

B →bB | b

This grammar is unambiguous; it is not hard to prove that every string in the langauge has one

and only one parse tree.

Problem 2. Give a context-free grammar for the language

$L = \{x \in \{0, 1\}_ : x$ has the same number of 0's and 1's$\}$

Is your grammar ambiguous?

Ans:

A grammar for the language is

S →S0S1S | S1S0S | ε

This grammar is ambiguous; for example, 0101 has two different parse trees

Problem 3. Prove that L = {aibjck : j = max{i, k}} is not context free.

Ans:

Suppose for contradiction that L were context free. Let N be the "N" of the pumping lemma

for context-free languages. Consider the string $w = a^N b^N c^N$. Suppose w = uvxyz, where $|vxy| \leq N$ and $|vy| \geq 1$. If vy contains only a's or vy contains only c's, then pump up: the string $uv^2xy^2z \notin L$. Suppose vy contains only b's. Then we can pump either way to get a string not in L. Suppose v contains two different letters or y contains two different letters. Then uv^2xy^2z is not even of the form a*b*c*, so certainly it is not in L. Finally, suppose ($v \in a+$ and) $y \in b+$, or $v \in b+$ (and $y \in c+$). Then we can pump down and there will be too few b's. By $|vwy| \leq N$, these are all the possible cases. So in all cases there is some i for which $uv^ixy^iz \notin L$, a contradiction.

Theorem : L ⊆A* is CF iff∃NPDA M that accepts L.

Proof ->: Given CFL L, consider any grammar G(L) for L. Construct NPDA M that simulates all possible derivations of G. M is essentially a single-state FSM, with a state q that applies one of G's rules at a time. The start state q0 initializes the stack with the content S ¢, where S is the start symbol of G, and ¢ is the bottom of stack symbol. This initial stack content means that M aims to read an input that is an instance of S. In general, the current stack content is a sequence of symbols that represent tasks to be accomplished in the characteristic LIFO order (last-in first-out). The task on top of the stack, say a non-terminal X, calls for the next characters of the input string to be an instance of X. When these characters have been read and verified to be an instance of X, X is popped from the stack, and the new task on top of the stack is started. When ¢ is on top of the stack, i.e. thestack is empty, all tasks generated by the first instance of S have been successfully met, i.e. the input string read so far is an instance of S. M moves to the accept state and stops.

The following transitions lead from q to q:

1) □, X-> w for each rule X -> w. When X is on top of the stack, replace X by a right-hand side for X.

2) a, a -> □ for each a ∈ A. When terminal a is read as input and a is also on top of the stack, pop the stack.

Rule 1 reflects the following fact: one way to meet the task of finding an instance of X as a prefix of the inputstring not yet read, is to solve all the tasks, in the correct order, present in the right-hand side w of the productionX -> w. M can be considered to be a non-deterministic parser for G. A formal proof that M accepts precisely Lcan be done by induction on the length of the derivation of any w ∈ L.

Ambiguous Grammar;

A grammar is said to be ambiguous if more than two parse trees can be constructed from it.

Example 1:

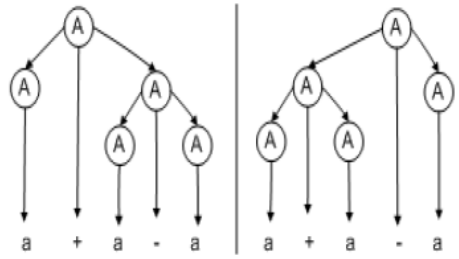
The context free grammar

$A \rightarrow A+A \mid A-A \mid a$

is ambiguous since there are two leftmost derivations for the string a + a + a:

$A \rightarrow A+A$	$A \rightarrow A+A$
$\rightarrow a+A$	$\rightarrow A+A+A$ (First A is replaced by A+A. Replacement of the second A would yield a similar derivation)
$\rightarrow a+A+A$	$\rightarrow a+A+A$
$\rightarrow a+a+A$	$\rightarrow a+a+A$
$\rightarrow a+a+a$	$\rightarrow a+a+a$

As another example, the grammar is ambiguous since there are two parse trees for the string a + a − a:



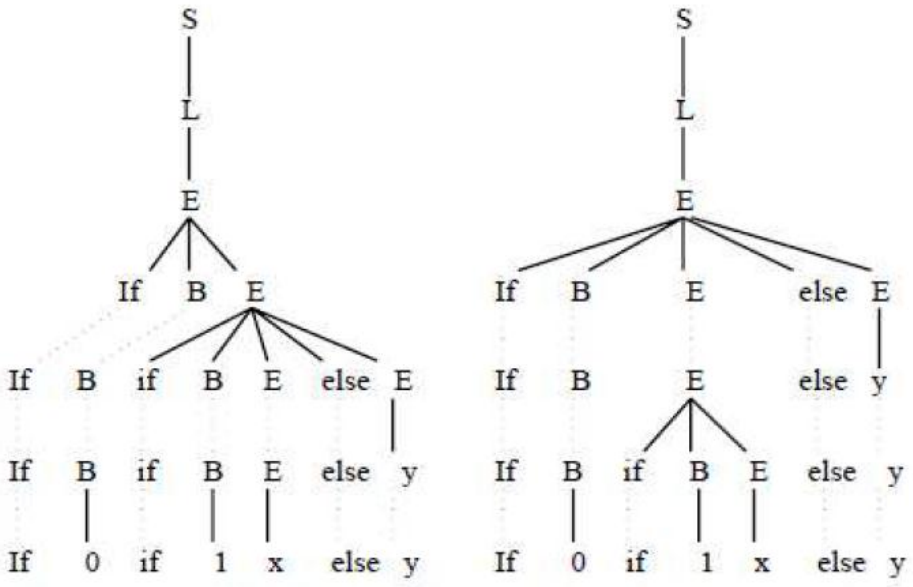
The language that it generates, however, is not inherently ambiguous; the following is a non-ambiguous grammar generating the same language:

$A \rightarrow A+a \mid A-a \mid a$

Example 2 : Show that the following grammar is ambigious

$$
\begin{aligned}
S &\longrightarrow L \\
L &\longrightarrow E \\
L &\longrightarrow LE \\
E &\longrightarrow \text{if B E else E} \\
E &\longrightarrow \text{if B E} \\
E &\longrightarrow x \\
E &\longrightarrow y \\
B &\longrightarrow 0 \\
B &\longrightarrow 1
\end{aligned}
$$

Answer :



As you can see we can have two corresponding parse trees for the above grammar, so the grammar is ambiguous.

Removal of Ambiguity

For compiling applications we need to design unambiguous grammar, or to use ambiguous grammar with additional rules to resolve the ambiguity.

1. Associativity of operators.
2. Precedence of operators.
3. Separate rules or Productions.

1. Associativity of Operators:

If operand has operators on both side then by connection, operand should be associated with the operator on the left.

In most programming languages arithmetic operators like addition, subtraction, multiplication, and division are left associative.

- Token string: 9 - 5 + 2
- Production rules

  list → list - digit | digit

  digit → 0 | 1 | 2 | . . . | 9

In the C programming language the assignment operator, =, is right associative. That is, token string a = b = c should be treated as a = (b = c).

- Token string: a = b =c.
- Production rules:

  right → letter = right | letter

  letter → a | b | . . . | z

2. Precedence of Operators:

An expression 9 + 5 * 2 has two possible interpretation:

(9 + 5) * 2 and 9 + (5 * L)

The associativity of '+' and '*' do not resolve this ambiguity. For this reason, we need to know the relative precedence of operators.

The convention is to give multiplication and division higher precedence than addition and subtraction.

Only when we have the operations of equal precedence, we apply the rules of associative.

So, in the example expression: 9 + 5 * 2.

We perform operation of higher precedence i.e., * before operations of lower precedence i.e., +.

Therefore, the correct interpretation is 9 + (5 *).

3. Separate Rule:

Consider the following grammar and language again.

S → IF b THEN S ELSE S
| IF b THEN S
| a

An ambiguity can be removed if we arbitrary decide that an ELSE should be attached to the last preceding THEN.

We can revise the grammar to have two nonterminals S_1 and S_2. We insist that S_2 generates IF-THEN-ELSE, while S_1 is free to generate either kind of statements.

The rules of the new grammar are:

S_1 → IF b THEN S_1
| IF b THEN S_2 THEN S_1
| a

S_2 → IF b THEN S_2 ELSE S_2
| a

Although there is no general algorithm that can be used to determine if a given grammar is ambiguous, it is certainly possible to isolate rules which leads to ambiguity or ambiguous grammar.

Example:

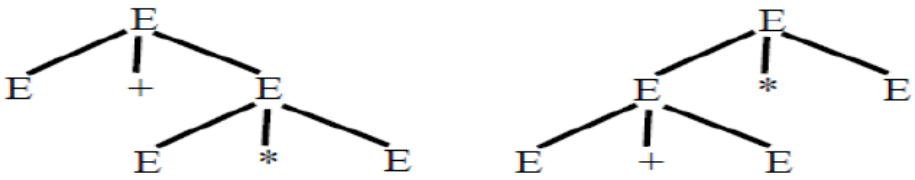
Show that the given grammar is ambiguous and also remove the ambiguity.

$$E \rightarrow I/E+E/E*E/(E)$$
$$I \rightarrow a/b/Ia/Ib/I0/I1$$

Answer :

Consider the sentential form $E+E*E$. It has two derivations from E.

1. $E \Rightarrow E+E \Rightarrow E+E*E.$
2. $E \Rightarrow E*E \Rightarrow E+E*E.$



(a) (b)

As two parse trees can be possible so the above given grammar is ambigious.

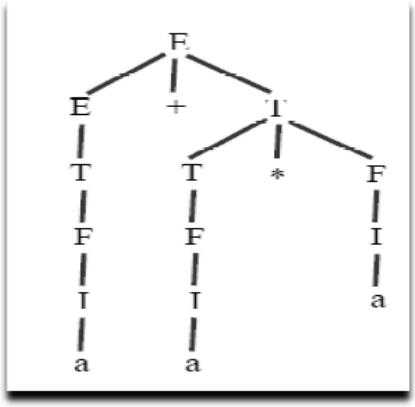
The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of **binding strength**. Specially:

1. A **factor** is an expression that can not be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.
2. A **term** is an expression that cannot be broken by the + operator. In our example, where + and * are the only operators, a term is a product of one or more factors. For instance, the term $a * b$ can be **broken** if we use left associativity and place $a1*$ to its left. That is, $a1 * a * b$ is grouped $(a1 * a) * b$, which breaks apart the $a * b$. However, placing additive term, such as $a1+$ to its left or $+a1$ to its right cannot break $a*b$. The proper grouping of $a1 + a * b$ is $a1 + (a * b)$, and the proper grouping of $a * b + a1$ is (a*b)+a1.
3. An **expression** will henceforth refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +. Thus, an expression for our example is a sum of one or more terms.

An unambiguous expression grammar

$$E \rightarrow T/E+T$$
$$T \rightarrow F/T*F$$
$$F \rightarrow I/(E)$$
$$I \rightarrow a/b/Ia/Ib/I0/I1$$

Now the same parse tree can be drawn as follows.



Inherent Ambiguity

A context-free language L is said to be inherently ambiguous if all its grammars are ambiguous.

If even one grammar for L is unambiguous, then L is an unambiguous language.

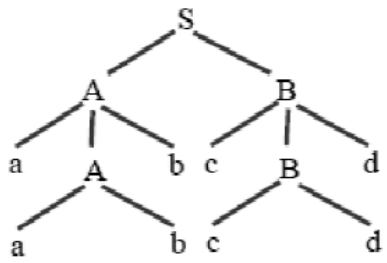
Example :

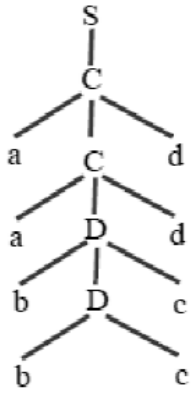
A grammar for an inherently ambiguous language

$$\begin{array}{|l} S \rightarrow AB/C \\ A \rightarrow aAb/ab \\ B \rightarrow cBd/cd \\ C \rightarrow aCd/aDd \\ D \rightarrow bDc/bc \end{array}$$

This grammar is ambiguous. For example, the string $aabbccdd$ has the two leftmost derivations:

1. $S \Rightarrow_{lm} AB \Rightarrow_{lm} aAbB \Rightarrow_{lm} aabbB \Rightarrow_{lm} aabbcBd \Rightarrow_{lm} aabbccdd.$
2. $S \Rightarrow_{lm} C \Rightarrow_{lm} aCd \Rightarrow_{lm} aaDdd \Rightarrow_{lm} aabDdd \Rightarrow_{lm} aabbccdd$





Pumping Lemma For CFL

For every CFL L there is a constant n such that every $z \in L$ of length $|z| \geq n$ can be written as $z = u\ v\ w\ x\ y$ such that the following holds:

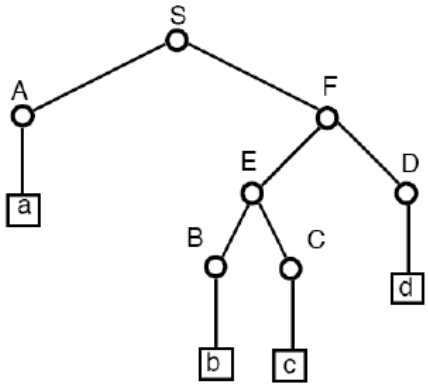
1) $v\ x \neq \square$

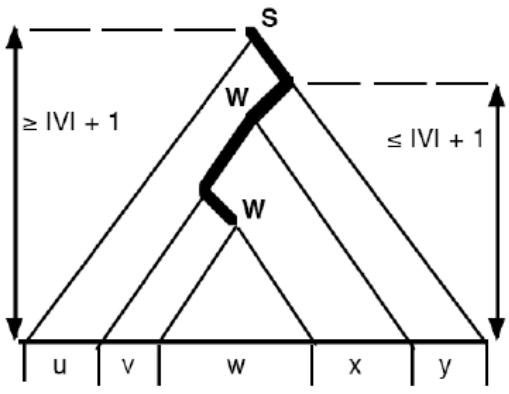
2) $|v\ w\ x| \leq n$, and

3) $uv^k\ w\ x^k\ y \in L$ for all $k \geq 0$.

Proof:

Given CFL L, choose any G = G(L) in Chomsky NF. This implies that the parse tree of any $z \in L$ is a binary tree, as shown in the figure below at left. The length n of the string at the leaves and the height h of a binary tree are related by $h \geq \log n$, i.e. a long string requires a tall parse tree. By choosing the critical length $n = 2^{|V|} + 1$ we force the height of the parse trees considered to be $h \geq |V| + 1$. On a root-to-leaf path of length $\geq |V| + 1$ we encounter at least $|V| + 1$ nodes labeled by non-terminals. Since G has only $|V|$ distinct non-terminals, this implies that on some long root-to-leaf path we must encounter 2 nodes labeled with the same non-terminal, say W, as shown at right.





For two such occurrences of W (in particular, the two lowest ones), and for some u, v, y, x, w $\in A^*$, we have: S ->* u W y, W ->* v W x and W ->* w. But then we also have W ->* v2 W x2, and in general, W ->* $v^k W x^k$, and S ->* u v^k W x^k y and S ->* u v^k w x^k y for all $k \geq 0$,

Example-1 : Let G be a CFG in Chomsky normal form that contains b variables. Show that, if G generates some string with a derivation having at least 2b steps, L(G) is infinite.

Answer:

Since G is a CFG in Chomsky normal form, every derivation can generate at most two non-terminals, so that in any parse tree using G, an internal node can have at most two children. This implies that every parse tree with height k has at most 2^k–1 internal nodes.If G generates some string with a derivation having at least 2^b steps, the parse tree of that string will have at least 2^b internal nodes. Based on the above argument, this parse tree has height is at least b + 1, so that there exists a path from root to leaf containing b + 1 variables. By pigeonhole principle, there is one variable occurring at least twice. So, we can use the technique in the proof of the pumping lemma to construct infinitely many strings which are all in L(G).

Example-2 :

Let C = {xy | x, y 2 {0, 1}_, |x| = |y|, and x 6= y}. Show that C is a context-free language.

Answer:

We observe that a string is in C if and only if it can be written as xywith |x| = |y| such that for some i, the i^{th} character of x is different from the i^{th} character of y. To obtain such a string, we start generating the corresponding i^{th}characters, and fill up the remaining characters. Based on the above idea, we define the CFG for C is as follows:

$S \rightarrow AB \mid BA$

$A \rightarrow XAX \mid 0$

$B \rightarrow XBX \mid 1$

$X \rightarrow 0 \mid 1$

Let A =

Example-3 :

Let A= $\{wtw^R \mid w, t \in \{0, 1\}^*$ and $|w| = |t|\}$ Prove that A is not a context-free language.

Answer:

Suppose on the contrary that A is context-free. Then, let p be the pumping length for A, such that any string in A of length at least p will satisfy the pumping lemma. Now, we select a string s in A with $s = 0^{2p}0^p1^p0^{2p}$. For s to satisfy the pumping lemma,

there is a way that s can be written as uvxyz, with $|vxy| \leq p$ and $|vy| \geq 1$, and for any i, uv^ixy^iz is a string in A. There are only three cases to write s with the above conditions:

Case 1: vy contains only 0s and these 0s are chosen from the last 0^{2p} of s. Let i be a number with $7p > |vy| \times (i + 1) \geq 6p$. Then, either the length of $uv^ixy^iz$is not a multiple of 3, or this string is of the form wtw0 such that $|w| = |t| = |w'|$ with w′is all 0s and w is not all 0s (this is, $w' = w^R$).

Case 2: vy does not contain any 0s in the last 0^2p of s. Then, either the length of uv^2xy^2z is not a multiple of 3, or this string is of the form wtw′ such that $|w| = |t| = |w'|$ with w is all 0s and w′ is not all 0s (that is, $w' = w^R$).

Case 3: vy is not all 0s, and some 0s are from the last 0^2p of s. As $|vxy| \_ p$, vxy in this case must be a substring in 1p′p. Then, either the length of uv2xy2z is not a multiple of 3, or this string is of the form wtw′such that $|w| = |t| = |w'|$ with w is all 0s and w′ is not all 0s (that is, w′ 6= w^R).

In summary, we observe that there is no way s can satisfy the pumping lemma. Thus, a contradiction occurs (where?), and we conclude that A is not a context-free language.

Theorem : L1 = { 0k 1k 2k / k ≥ 0 } is not context free.

Pf (by contradiction): Assume L is CF, let n be the constant asserted by the pumping lemma. Consider z = 0n 1n 2n = u v w x y. Although we don't know where vwx is positioned within z, the assertion $|vw x| \leq n$ implies that v w x contains at most two distinct letters among 0, 1, 2. In other words, one or two of thethree letters 0, 1, 2 is missing in vwx. Now consider u v2 w x2 y. By the pumping lemma, it must be in L. Theassertion $|v x| \geq 1$ implies that u v2 w x2 y is longer than u v w x y. But u v w x y had an equal number of 0s, 1s,and 2s, whereas u v2 w x2 y cannot, since only one or two of the three distinct symbols increased in number. Thiscontradiction proves the theorem.

Theorem : L2 = { w w / w ∈ {0, 1} } is not context free.

Proof (by contradiction): Assume L is CF, let n be the constant asserted by the pumping lemma. Consider z = 0n+1 1n+1 0n+1 1n+1 = u v w x y. Using k = 0, the lemma asserts z0 = u w y ∈ L, but we showthat z0 cannot have the form t t, for any string t, and thus that z0 ∈ L, leading to a contradiction. Recall that $|v wx| \leq n$, and thus, when we delete v and x, we delete symbols that

are within a distance of at most n from eachother. By analyzing three cases we show that, under this restriction, it is impossible to delete symbols in such away as to retain the property that the shortened string z0 = u w x has the form t t. We illustrate this using theexample n = 3, but the argument holds for any n.Given z = 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1, slide a window of length n = 3 across z, and delete any characters youwant from within the window. Observe that the blocks of 0s and of 1s within z are so long that the truncated z,call it z', still has the form "0s 1s 0s 1s". This implies that if z' can be written as z' = t t, then t must have theform t = "0s 1s". Checking the three cases: the window of length 3 lies entirely within the left half of z; thewindow straddles the center of z; and the window lies entirely within the right half of z, we observe that in noneof these cases z' has the form z' = t t, and thus that z0 = u w y $\notin$ L.

Context sensitive grammars and languages

The rewriting rules B -> w of a CFG imply that a non-terminal B can be replaced by a word w $\in$ (V □ A)* "in any context". In contrast, a context sensitive grammar (CSG) has rules of the form: u B v -> u w v, where u, v, w $\in$ (V □ A)*, implying that B can be replaced by w only in the context "u on the left, v on the right". It turns out that this definition is equivalent (apart from the nullstring□) to requiring that any CSG rule be of the form v -> w, where v, w $\in$ (V □ A)*, and |v| $\leq$ |w|. This monotonicity property (in any derivation, the current string never gets shorter) implies that the word problem for CSLs: "given CSG G and given w, is w $\in$ L(G)?" is decidable. An exhaustive enumeration of all derivations up to the length |w| settles the issue. As an example of the greater power of CSGs over CFGs, recall that we used the pumping lemma to prove that the language $0^k 1^k 2^k$ is not CF.

Parikh's theorem

Parikh's theorem in theoretical computer science says that if we look only at the relative number of occurrences of terminal symbols in a context-free language, without regard to their order, then the language is indistinguishable from a regular language. It is useful for deciding whether or not a string with given number of some terminals is accepted by a context-free grammar. It was first proved by Rohit Parikh in 1961 and republished in 1966.

Reg. No.....

16MMU502B

Karpagam Academy of Higher Education
Karpagam University
Coimbatore-21
Department of Mathematics
V Semester- I Internal test
Boolean Algebra and Automata theory

Date: Time: 2 hours
Class: III B.Sc Mathematics Max Marks: 50

Answer ALL questions
PART - A ($20 \times 1 = 20$ marks)

1. Hasse diagrams are drawn for
   a. poset b. lattice
   c. both a and b d. neither a nor b

2. The idempotent is defined as
   a. $a \vee a = a$ b. $a \wedge a = a$
   c. both a and b d. neither a nor b

3. In a poset $(P, \leq)$, $a, b \in P$ are called caomparable if
   a. $a \leq b$ b. $b \leq a$
   c. both a and b d. either a or b

4. The relation $\leq$ is a —— order on the set of real numbers $\mathbb{R}$.
   a. partial
   b. total
   c. both a and b d. either a or b

5. The set $\{5, 15, 25, 35\}$ with usual $\leq$, is a —— set
   a. partial b. total
   c. both a and b d. either a or b

6. In a poset $(\mathbb{N}, \leq)$, the zero element is
   a. 0 b. 1
   c. both a and b d. either a or b

7. Number of zero elements in a poset
   a. 0 b. 1
   c. 3 d. 5

8. In $\mathbb{N}$ with $a \leq b$ iff $a|b$, $\leq$ is —— order
   a. partial b. total
   c. both a and b d. either a or b

9. A lattice is a poset in which every two elements have
   a. g.l.b b. l.u.b
   c. both a and b d. either a or b

10. Suppose $a \leq b$. Then
    a. $a \vee b = b$ b. $a \wedge b$
    c. both a and b d. either a or b

11. Suppose L is a lattice with 0 and 1. Then $0' =$
    a. 0 b. 1
    c. both a and b d. either a or b

12. The relation $\{(1,2), (1,3), (3,1), (1,1), (3,3), (3,2), (1,4), (4,2), (3,4)\}$ is
    a.reflexive b. symmetric
    c. antisymmetric d. transitive

13. Hasse diagram are drawn for
    a. poset b. lattice
    c. Boolean algebera d. all the above

14. Let R be a non-empty relation on a collection of sets defined by $A \leq B$ if and only if $A \cap B = \emptyset$
    a. $\leq$ is relexive and transitive
    b. $\leq$ is an equivalence relation

c. $\leq$ is symmetric and not transitive
d. $\leq$ is not relexive and not symmetric

15. Let A be a set with n elements. Then number of relations defined on A is
a. n b. n^2
c. 2^n d. 2^{n^2}

16. A self-complemented, distributive lattice is called
a. Boolean algebra b. Modular lattice
c. Complete lattice d. Self dual lattice

17. Let $X = \{2, 3, 6, 12, 24\}$, and = be the partial order defined by $a \leq b$ if a divides b. Number of edges in the Hasse diagram of $(X, \leq)$ is
a. 2 b. 4
c. 6 d. 8

18. The absorption law is defined as
a. $a \vee (a \wedge b) = a$ b. $a \wedge (a \vee b) = a$
c. both a and b d. neither a nor b

19. Power set of empty set has exactly —- subset.
a. 0 b. 1
c.2 d. 3

20. What is the cardinality of the set of odd positive integers less than 10?
a. 10 b. 15
c. 5 d. 20

Part B-($3 \times 2 = 6$ marks)

21. Define partially oredered set

22. Give an example for a poset which is not a lattice

23. Define comparable elements

Part C-($3 \times 8 = 24$ marks)

24. a) Show that in a poset the l.u.b and g.l.b of a subset need not exist

OR

b) Let L be a lattice and $a, b.c \in L$. Then $a \leq b$ and $c \leq d$ implies
i) $a \vee c \leq b \vee d$
ii) $a \wedge c \leq b \wedge d$

25. a) State and prove associative law for a lattice

OR

b) Prove that in a lattice
i $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
i $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

26. a) Prove that the lattice of normal subgroups of any group is modular lattice

OR

b) Let G be a group. Let L be the set of all subgroups of G. Define $A \leq B$ iff $A \subset B$. Prove that $(L, \leq)$ is a lattice