Eastern Economy Edition

Theory of Computer Science

Automata, Languages and Computation

Third Edition

K.L.P. Mishra
N. Chandrasekaran

PHI

http://engineeringbooks.net

THEORY OF COMPUTER SCIENCE

Automata, Languages and Computation

THIRD EDITION

K.L.P. MISHRA
Formerly Professor
Department of Electrical and Electronics Engineering
and Principal, Regional Engineering College
Tiruchirapalli

N. CHANDRASEKARAN
Professor
Department of Mathematics
St. Joseph's College
Tiruchirapalli

Prentice-Hall of India Private Limited
New Delhi - 110 001
2008

http://engineeringbooks.net

Contents

Preface *ix*

Notations *xi*

1. PROPOSITIONS AND PREDICATES **1–35**

1.1 Propositions (or Statements) *1*

1.1.1 Connectives (Propositional Connectives or Logical Connectives) *2*

1.1.2 Well-formed Formulas *6*

1.1.3 Truth Table for a Well-formed Formula *7*

1.1.4 Equivalence of Well-formed Formulas *9*

1.1.5 Logical Identities *9*

1.2 Normal Forms of Well-formed Formulas *11*

1.2.1 Construction to Obtain a Disjunctive Normal Form of a Given Formula *11*

1.2.2 Construction to Obtain the Principal Disjunctive Normal Form of a Given Formula *12*

1.3 Rules of Inference for Propositional Calculus (Statement Calculus) *15*

1.4 Predicate Calculus *19*

1.4.1 Predicates *19*

1.4.2 Well-formed Formulas of Predicate Calculus *21*

1.5 Rules of Inference for Predicate Calculus *23*

1.6 Supplementary Examples *26*

Self-Test *31*

Exercises *32*

http://engineeringbooks.net

2. MATHEMATICAL PRELIMINARIES **36–70**

2.1 Sets, Relations and Functions *36*
2.1.1 Sets and Subsets *36*
2.1.2 Sets with One Binary Operation *37*
2.1.3 Sets with Two Binary Operations *39*
2.1.4 Relations *40*
2.1.5 Closure of Relations *43*
2.1.6 Functions *45*
2.2 Graphs and Trees *47*
2.2.1 Graphs *47*
2.2.2 Trees *49*
2.3 Strings and Their Properties *54*
2.3.1 Operations on Strings *54*
2.3.2 Terminal and Nonterminal Symbols *56*
2.4 Principle of Induction *57*
2.4.1 Method of Proof by Induction *57*
2.4.2 Modified Method of Induction *58*
2.4.3 Simultaneous Induction *60*
2.5 Proof by Contradiction *61*
2.6 Supplementary Examples *62*
Self-Test *66*
Exercises *67*

3. THE THEORY OF AUTOMATA **71–106**

3.1 Definition of an Automaton *71*
3.2 Description of a Finite Automaton *73*
3.3 Transition Systems *74*
3.4 Properties of Transition Functions *75*
3.5 Acceptability of a String by a Finite Automaton *77*
3.6 Nondeterministic Finite State Machines *78*
3.7 The Equivalence of DFA and NDFA *80*
3.8 Mealy and Moore Models *84*
3.8.1 Finite Automata with Outputs *84*
3.8.2 Procedure for Transforming a Mealy Machine into a Moore Machine *85*
3.8.3 Procedure for Transforming a Moore Machine into a Mealy Machine *87*
3.9 Minimization of Finite Automata *91*
3.9.1 Construction of Minimum Automaton *92*
3.10 Supplementary Examples 97
Self-Test *103*
Exercises *104*

http://engineeringbooks.net

4. FORMAL LANGUAGES **107–135**

4.1 Basic Definitions and Examples *107*
4.1.1 Definition of a Grammar *109*
4.1.2 Derivations and the Language Generated by a Grammar *110*
4.2 Chomsky Classification of Languages *120*
4.3 Languages and Their Relation *123*
4.4 Recursive and Recursively Enumerable Sets *124*
4.5 Operations on Languages *126*
4.6 Languages and Automata *128*
4.7 Supplementary Examples 129
Self-Test *132*
Exercises *134*

5. REGULAR SETS AND REGULAR GRAMMARS **136–179**

5.1 Regular Expressions *136*
5.1.1 Identities for Regular Expressions *138*
5.2 Finite Automata and Regular Expressions *140*
5.2.1 Transition System Containing Λ-moves *140*
5.2.2 NDFAs with Λ-moves and Regular Expressions *142*
5.2.3 Conversion of Nondeterministic Systems to Deterministic Systems *146*
5.2.4 Algebraic Method Using Arden's Theorem *148*
5.2.5 Construction of Finite Automata Equivalent to a Regular Expression *153*
5.2.6 Equivalence of Two Finite Automata *157*
5.2.7 Equivalence of Two Regular Expressions *160*
5.3 Pumping Lemma for Regular Sets *162*
5.4 Application of Pumping Lemma *163*
5.5 Closure Properties of Regular Sets *165*
5.6 Regular Sets and Regular Grammars *167*
5.6.1 Construction of a Regular Grammar Generating $T(M)$ for a Given DFA M *168*
5.6.2 Construction of a Transition System M Accepting $L(G)$ for a Given Regular Grammar G *169*
5.7 Supplementary Examples *170*
Self-Test *175*
Exercises *176*

6. CONTEXT-FREE LANGUAGES **180–226**

6.1 Context-free Languages and Derivation Trees *180*
6.1.1 Derivation Trees *181*
6.2 Ambiguity in Context-free Grammars *188*

http://engineeringbooks.net

6.3 Simplification of Context-free Grammars *189*
6.3.1 Construction of Reduced Grammars *190*
6.3.2 Elimination of Null Productions *196*
6.3.3 Elimination of Unit Productions *199*
6.4 Normal Forms for Context-free Grammars *201*
6.4.1 Chomsky Normal Form *201*
6.4.2 Greibach Normal Form *206*
6.5 Pumping Lemma for Context-free Languages *213*
6.6 Decision Algorithms for Context-free Languages *217*
6.7 Supplementary Examples *218*
Self-Test *223*
Exercises *224*

7. PUSHDOWN AUTOMATA **227–266**
7.1 Basic Definitions *227*
7.2 Acceptance by pda *233*
7.3 Pushdown Automata and Context-free Languages *240*
7.4 Parsing and Pushdown Automata *251*
7.4.1 Top-down Parsing *252*
7.4.2 Top-down Parsing Using Deterministic pda's *256*
7.4.3 Bottom-up Parsing *258*
7.5 Supplementary Examples *260*
Self-Test *264*
Exercises *265*

8. LR(k) GRAMMARS **267–276**
8.1 LR(k) Grammars *267*
8.2 Properties of LR(k) Grammars *270*
8.3 Closure Properties of Languages *272*
8.4 Supplementary Examples *272*
Self-Test *273*
Exercises *274*

9. TURING MACHINES AND LINEAR BOUNDED AUTOMATA **277–308**
9.1 Turing Machine Model *278*
9.2 Representation of Turing Machines *279*
9.2.1 Representation by Instantaneous Descriptions *279*
9.2.2 Representation by Transition Table *280*
9.2.3 Representation by Transition Diagram *281*
9.3 Language Acceptability by Turing Machines *283*
9.4 Design of Turing Machines *284*
9.5 Description of Turing Machines *289*

http://engineeringbooks.net

9.6 Techniques for TM Construction *289*
9.6.1 Turing Machine with Stationary Head *289*
9.6.2 Storage in the State *290*
9.6.3 Multiple Track Turing Machine *290*
9.6.4 Subroutines *290*
9.7 Variants of Turing Machines *292*
9.7.1 Multitape Turing Machines *292*
9.7.2 Nondeterministic Turing Machines *295*
9.8 The Model of Linear Bounded Automaton *297*
9.8.1 Relation Between LBA and Context-sensitive Languages *299*
9.9 Turing Machines and Type 0 Grammars *299*
9.9.1 Construction of a Grammar Corresponding to *TM* *299*
9.10 Linear Bounded Automata and Languages *301*
9.11 Supplementary Examples *303*
Self-Test *307*
Exercises *308*

10. DECIDABILITY AND RECURSIVELY ENUMERABLE LANGUAGES **309–321**

10.1 The Definition of an Algorithm *309*
10.2 Decidability *310*
10.3 Decidable Languages *311*
10.4 Undecidable Languages *313*
10.5 Halting Problem of Turing Machine *314*
10.6 The Post Correspondence Problem *315*
10.7 Supplementary Examples *317*
Self-Test *319*
Exercises *319*

11. COMPUTABILITY **322–345**

11.1 Introduction and Basic Concepts *322*
11.2 Primitive Recursive Functions *323*
11.2.1 Initial Functions *323*
11.2.2 Primitive Recursive Functions Over N *325*
11.2.3 Primitive Recursive Functions Over $\{a, b\}$ *327*
11.3 Recursive Functions *329*
11.4 Partial Recursive Functions and Turing Machines *332*
11.4.1 Computability *332*
11.4.2 A Turing Model for Computation *333*
11.4.3 Turing-computable Functions *333*
11.4.4 Construction of the Turing Machine That Can Compute the Zero Function Z *334*
11.4.5 Construction of the Turing Machine for Computing—The Successor Function *335*

http://engineeringbooks.net

11.4.6 Construction of the Turing Machine for Computing the Projection U_i^m *336*
11.4.7 Construction of the Turing Machine That Can Perform Composition *338*
11.4.8 Construction of the Turing Machine That Can Perform Recursion *339*
11.4.9 Construction of the Turing Machine That Can Perform Minimization *340*
11.5 Supplementary Examples *340*
Self-Test *342*
Exercises *343*

12. COMPLEXITY **346–371**
12.1 Growth Rate of Functions *346*
12.2 The Classes **P** and **NP** *349*
12.3 Polynomial Time Reduction and *NP*-completeness *351*
12.4 Importance of *NP*-complete Problems *352*
12.5 SAT is *NP*-complete *353*
12.5.1 Boolean Expressions *353*
12.5.2 Coding a Boolean Expression *353*
12.5.3 Cook's Theorem *354*
12.6 Other *NP*-complete Problems *359*
12.7 Use of *NP*-completeness *360*
12.8 Quantum Computation *360*
12.8.1 Quantum Computers *361*
12.8.2 Church–Turing Thesis *362*
12.8.3 Power of Quantum Computation *363*
12.8.4 Conclusion *364*
12.9 Supplementary Examples *365*
Self-Test *369*
Exercises *370*

Answers to Self-Tests ***373–374***
Solutions (or Hints) to Chapter-end Exercises ***375–415***
Further Reading ***417–418***
Index ***419–422***

http://engineeringbooks.net

Preface

The enlarged third edition of *Theory of Computer Science* is the result of the enthusiastic reception given to earlier editions of this book and the feedback received from the students and teachers who used the second edition for several years.

The new edition deals with all aspects of theoretical computer science, namely **automata, formal languages, computability** and **complexity**. Very few books combine all these theories and give adequate examples. This book provides numerous examples that illustrate the basic concepts. It is profusely illustrated with diagrams. While dealing with theorems and algorithms, the emphasis is on constructions. Each construction is immediately followed by an example and only then the formal proof is given so that the student can master the technique involved in the construction before taking up the formal proof.

The key feature of the book that sets it apart from other books is the provision of detailed solutions (at the end of the book) to chapter-end exercises.

The chapter on Propositions and Predicates (Chapter 10 of the second edition) is now the first chapter in the new edition. The changes in other chapters have been made without affecting the structure of the second edition. The chapter on Turing machines (Chapter 7 of the second edition) has undergone major changes.

A novel feature of the third edition is the addition of objective type questions in each chapter under the heading Self-Test. This provides an opportunity to the student to test whether he has fully grasped the fundamental concepts. Besides, a total number of 83 additional solved examples have been added as Supplementary Examples which enhance the variety of problems dealt with in the book.

http://engineeringbooks.net

The sections on pigeonhole principle and the principle of induction (both in Chapter 2) have been expanded. In Chapter 5, a rigorous proof of Kleene's theorem has been included. The chapter on LR(k) grammars remains the same Chapter 8 as in the second edition.

Chapter 9 focuses on the treatment of Turing machines (TMs). A new section on high-level description of TM has been added and this is used in later examples and proofs. Some techniques for the construction of TMs have been added in Section 9.6. The multitape Turing machine and the nondeterministic Turing machine are discussed in Section 9.7.

A new chapter (Chapter 10) on decidability and recursively enumerable languages is included in this third edition. In the previous edition only a sketchy introduction to these concepts was given. Some examples of recursively enumerable languages are given in Section 10.3 and undecidable languages are discussed in Section 10.4. The halting problem of TM is discussed in Section 10.5. Chapter 11 on computability is Chapter 9 of the previous edition without changes.

Chapter 12 is a new chapter on complexity theory and *NP*-complete problems. Cook's theorem is proved in detail. A section on Quantum Computation is added as the last section in this chapter. Although this topic does not fall under the purview of theoretical computer science, this section is added with a view to indicating how the success of Quantum Computers will lead to dramatic changes in complexity theory in the future.

The book fulfils the curriculum needs of undergraduate and postgraduate students of computer science and engineering as well as those of MCA courses. Though designed for a one-year course, the book can be used as a one-semester text by a judicious choice of the topics presented.

Special thanks go to all the teachers and students who patronized this book over the years and offered helpful suggestions that have led to this new edition. In particular, the critical comments of Prof. M. Umaparvathi, Professor of Mathematics, Seethalakshmi College, Tiruchirapalli are gratefully acknowledged.

Finally, the receipt of suggestions, comments and error reports for further improvement of the book would be welcomed and duly acknowledged.

K.L.P. Mishra
N. Chandrasekran

http://engineeringbooks.net

Notations

Symbol	*Meaning*	*Section in which the symbol appears first and is explained*
T	Truth value	1.1
F	False value	1.1
$\neg$	The logical connective NOT	1.1
$\wedge$	The logical connective AND	1.1
$\vee$	The logical connective OR	1.1
$\Rightarrow$	The logical connective IF . . . THEN . . .	1.1
$\Leftrightarrow$	The logical connective If and Only If	1.1
$\mathbf{T}$	Any tautology	1.1
$\mathbf{F}$	Any contradiction	1.1
$\forall$	For every	1.4
$\exists$	There exists	1.4
$\equiv$	Equivalence of predicate formulas	1.4
$a \in A$	The element a belongs to the set A	2.1.1
$A \subseteq B$	The set A is a subset of set B	2.1.1
$\emptyset$	The null set	2.1.1
$A \cup B$	The union of the sets A and B	2.1.1
$A \cap B$	The intersection of the sets A and B	2.1.1
$A - B$	The complement of B in A	2.1.1
A^c	The complement of A	2.1.1
2^A	The power set of A	2.1.1
$A \times B$	The cartesian product of A and B	2.1.1

http://engineeringbooks.net

Symbol	*Meaning*	*Section in which the symbol appears first and is explained*
$\bigcup_{i=1}^{n} A_i$	The union of the sets $A_1, A_2, \ldots, A_n$	2.1.2
$*$, o	Binary operations	2.1.2, 2.1.3
xRy	x is related to y under the relation	2.1.4
$xR'y$	x is not related to y under the relation R	2.1.4
$i = j \mod n$	i is congruent to j modulo n	2.1.4
C_a	The equivalence class containing a	2.1.4
R^+	The transitive closure of R	2.1.5
R^*	The reflexive-transitive closure of R	2.1.5
$R_1 \circ R_2$	The composite of the relations R_1 and R_2	2.1.5
$f: X \to Y$	Map/function from a set X to a set Y	2.1.6
$f(x)$	The image of x under f	2.1.6
$\lceil x \rceil$	The smallest integer $\geq x$	2.2.2
Σ^*	The set of all strings over the alphabet set Σ	2.3
$\wedge$	The empty string	2.3
Σ^+	The set of all nonempty strings over Σ	2.3
$\lvert x \rvert$	The length of the string x	2.3
$(Q, \Sigma, \delta, q_0, F)$	A finite automaton	3.2
¢	Left endmarker in input tape	3.2
\$	Right endmarker in input tape	3.2
$(Q, \Sigma, \delta, Q_0, F)$	A transition system	3.3
$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$	A Mealy/Moore machine	3.8
π	Partition corresponding to equivalence of states	3.9
π_k	Partition corresponding to k-equivalence of states	3.9
(V_N, Σ, P, S)	A grammar	4.1.1
$\alpha \underset{G}{\Rightarrow} \beta$	α directly derives β in grammar G	4.1.2
$\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$	α derives β in grammar G	4.1.2
$\alpha \overset{n}{\underset{G}{\Rightarrow}} \beta$	α derives β in n steps in grammar G	4.1.2
$L(G)$	The language generated by G	4.1.2
$\mathscr{L}_0$	The family of type 0 languages	4.3
$\mathscr{L}_{\text{csl}}$	The family of context-sensitive languages	4.3
$\mathscr{L}_{\text{cfl}}$	The family of context-free languages	4.3
$\mathscr{L}_{\text{rl}}$	The family of regular languages	4.3
$\mathbf{R}_1 + \mathbf{R}_2$	The union of regular expressions $\mathbf{R}_1$ and $\mathbf{R}_2$	5.1
$\mathbf{R}_1\mathbf{R}_2$	The concatenation of regular expressions $\mathbf{R}_1$ and $\mathbf{R}_2$	5.1
$\mathbf{R}^*$	The iteration (closure) of $\mathbf{R}$	5.1

http://engineeringbooks.net

Symbol	*Meaning*	*Section in which the symbol appears first and is explained*
a	The regular expression corresponding to $\{a\}$	5.1
$\|V_N\|$	The number of elements in V_N	6.3
$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$	A pushdown automaton	7.1
ID	An instantaneous description of a pda	7.1
$\vdash_A^*$	A move relation in a pda A	7.1
$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$	A Turing machine	9.1
$\vdash^*$	The move relation in a TM	9.2.2
$T(n)$	Maximum of the running time of M	9.7.1
A_{DFA}	Sets	10.3
A_{CFG}	Sets	10.3
A_{CSG}	Sets	10.3
A_{TM}	Sets	10.4
$HALT_{\text{TM}}$	Sets	10.5
$Z(x)$	The image of x under zero function	11.2
$S(x)$	The image of x under successor function	11.2
U_1^n	The projection function	11.2
nil (x)	The image of x under nil function	11.2
cons $a(x)$	The concatenation of a and x	11.2
cons $b(x)$	The concatenation of b and x	11.2
$p(x)$	The image of x under predecessor function	11.2
$\dot{-}$	The proper subtraction function	Example 11.8
$\|x\|$	The absolute value of x	11.2
id	Identity function	11.2
μ_y	The minimization function	11.3
χ_A	The characteristic function of the set A	Exercise 11.8
$O(g(n))$	The order of $g(n)$	12.1
$O(n^k)$	The order of n^k	12.1
P and **NP**	Classes	12.2
$\|\psi>$	quantum bit (qubit)	12.8.1
PSPACE	Classes	12.8.3
EXP	Classes	12.8.3
BPP	Classes	12.8.3
BQP	Classes	12.8.3
NPI	Classes	12.8.3

http://engineeringbooks.net

1 Propositions and Predicates

Mathematical logic is the foundation on which the proofs and arguments rest. Propositions are statements used in mathematical logic, which are either true or false but not both and we can definitely say whether a proposition is true or false.

In this chapter we introduce propositions and logical connectives. Normal forms for well-formed formulas are given. Predicates are introduced. Finally, we discuss the rules of inference for propositional calculus and predicate calculus.

1.1 PROPOSITIONS (OR STATEMENTS)

A proposition (or a statement) is classified as a declarative sentence to which only one of the truth values, i.e. true or false, can be assigned. When a proposition is true, we say that its truth value is *T*. When it is false, we say that its truth value is *F*.

Consider, for example, the following sentences in English:

1. New Delhi is the capital of India.
2. The square of 4 is 16.
3. The square of 5 is 27.
4. Every college will have a computer by 2010 A.D.
5. Mathematical logic is a difficult subject.
6. Chennai is a beautiful city.
7. Bring me coffee.
8. No, thank you.
9. This statement is false.

The sentences 1–3 are propositions. The sentences 1 and 2 have the truth value *T*. The sentence 3 has the truth value *F*. Although we cannot *know* the truth value of 4 at present, we definitely know that it is true or false, but not both.

http://engineeringbooks.net

So the sentence 4 is a proposition. For the same reason, the sentences 5 and 6 are propositions. To sentences 7 and 8, we cannot assign truth values as they are not declarative sentences. The sentence 9 looks like a proposition. However, if we assign the truth value T to sentence 9, then the sentence asserts that it is false. If we assign the truth value F to sentence 9, then the sentence asserts that it is true. Thus the sentence 9 has either both the truth values (or none of the two truth values). Therefore, the sentence 9 is not a proposition.

We use capital letters to denote propositions.

1.1.1 Connectives (Propositional Connectives or Logical Connectives)

Just as we form new sentences from the given sentences using words like 'and', 'but', 'if', we can get new propositions from the given propositions using 'connectives'. But a new sentence obtained from the given propositions using connectives will be a proposition only when the new sentence has a truth value either T or F (but not both). The truth value of the new sentence depends on the (logical) connectives used and the truth value of the given propositions.

We now define the following connectives. There are five basic connectives.

(i) Negation (NOT)
(ii) Conjunction (AND)
(iii) Disjunction (OR)
(iv) Implication (IF ... THEN ...)
(v) If and Only If.

Negation (NOT)

If P is a proposition then the negation P or NOT P (read as 'not P') is a proposition (denoted by $\neg P$) whose truth value is T if P has the truth value F, and whose truth value is F if P has the truth value T. Usually, the truth values of a proposition defined using a connective are listed in a table called the **truth table** for that connective (Table 1.1).

TABLE 1.1 Truth Table for Negation

P	$\neg P$
T	F
F	T

Conjunction (AND)

If P and Q are two propositions, then the conjunction of P and Q (read as 'P and Q') is a proposition (denoted by $P \wedge Q$) whose truth values are as given in Table 1.2.

http://engineeringbooks.net

TABLE 1.2 Truth Table for Conjunction

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction (OR)

If P and Q are two propositions, then the disjunction of P and Q (read as 'P or Q') is a proposition (denoted by $P \vee Q$) whose truth values are as given in Table 1.3.

TABLE 1.3 Truth Table for Disjunction

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

It should be noted that $P \vee Q$ is true if P is true or Q is true or both are true. This OR is known as *inclusive* OR, i.e. either P is true or Q is true or both are true. Here we have defined OR in the inclusive sense. We will define another connective called *exclusive* OR (either P is true or Q is true, but not both, i.e. where OR is used in the exclusive sense) in the Exercises at the end of this chapter.

EXAMPLE 1.1

If P represents 'This book is good' and Q represents 'This book is cheap', write the following sentences in symbolic form:

(a) This book is good and cheap.
(b) This book is not good but cheap.
(c) This book is costly but good.
(d) This book is neither good nor cheap.
(e) This book is either good or cheap.

Solution

(a) $P \wedge Q$
(b) $\neg P \wedge Q$
(c) $\neg Q \wedge P$
(d) $\neg P \wedge \neg Q$
(e) $P \vee Q$

Note: The truth tables for $P \wedge Q$ and $Q \wedge P$ coincide. So $P \wedge Q$ and $Q \wedge P$ are equivalent (for the definition, see Section 1.1.4). But in natural languages this need not happen. For example, the two sentences,

http://engineeringbooks.net

namely 'I went to the railway station and boarded the train', and 'I boarded the train and went to the railway station', have different meanings. Obviously, we cannot write the second sentence in place of the first sentence.

Implication (IF ... THEN ...)

If P and Q are two propositions, then 'IF P THEN Q' is a proposition (denoted by $P \Rightarrow Q$) whose truth values are given Table 1.4. We also read $P \Rightarrow Q$ as 'P implies Q'.

TABLE 1.4 Truth Table for Implication

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

We can note that $P \Rightarrow Q$ assumes the truth value F only if P has the truth value T and Q has the truth value F. In all the other cases, $P \Rightarrow Q$ assumes the truth value T. In the case of natural languages, we are concerned about the truth values of the sentence 'IF P THEN Q' only when P is true. When P is false, we are not concerned about the truth value of 'IF P THEN Q'. But in the case of mathematical logic, we have to definitely specify the truth value of $P \Rightarrow Q$ in all cases. So the truth value of $P \Rightarrow Q$ is defined as T when P has the truth value F (irrespective of the truth value of Q).

EXAMPLE 1.2

Find the truth values of the following propositions:

(a) If 2 is not an integer, then 1/2 is an integer.
(b) If 2 is an integer, then 1/2 is an integer.

Solution

Let P and Q be '2 is an integer', '1/2 is an integer', respectively. Then the proposition (a) is true (as P is false and Q is false) and the proposition (b) is false (as P is true and Q is false).

The above example illustrates the following: 'We can prove anything if we start with a false assumption.' We use $P \Rightarrow Q$ whenever we want to 'translate' any one of the following: 'P only if Q', 'P is a sufficient condition for Q', 'Q is a necessary condition for P', 'Q follows from P', 'Q whenever P', 'Q provided P'.

If and Only If

If P and Q are two statements, then 'P if and only if Q' is a statement (denoted by $P \Leftrightarrow Q$) whose truth value is T when the truth values of P and Q are the same and whose truth value is F when the statements differ. The truth values of $P \Leftrightarrow Q$ are given in Table 1.5.

http://engineeringbooks.net

TABLE 1.5 Truth Table for If and Only If

P	Q	$P \Leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

Table 1.6 summarizes the representation and meaning of the five logical connectives discussed above.

TABLE 1.6 Five Basic Logical Connectives

Connective	*Resulting proposition*	*Read as*
Negation $\neg$	$\neg P$	Not P
Conjunction $\wedge$	$P \wedge Q$	P and Q
Disjunction $\vee$	$P \vee Q$	P or Q (or both)
Implication $\Rightarrow$	$P \Rightarrow Q$	If P then Q (P implies Q)
If and only if $\Leftrightarrow$	$P \Leftrightarrow Q$	P if and only if Q

EXAMPLE 1.3

Translate the following sentences into propositional forms:

(a) If it is not raining and I have the time, then I will go to a movie.
(b) It is raining and I will not go to a movie.
(c) It is not raining.
(d) I will not go to a movie.
(e) I will go to a movie only if it is not raining.

Solution

Let P be the proposition 'It is raining'.
Let Q be the proposition 'I have the time'.
Let R be the proposition 'I will go to a movie'.

Then

(a) $(\neg P \wedge Q) \Rightarrow R$
(b) $P \wedge \neg R$
(c) $\neg P$
(d) $\neg R$
(e) $R \Rightarrow \neg P$

EXAMPLE 1.4

If P, Q, R are the propositions as given in Example 1.3, write the sentences in English corresponding to the following propositional forms:

http://engineeringbooks.net

(a) $(\neg P \wedge Q) \Leftrightarrow R$
(b) $(Q \Rightarrow R) \wedge (R \Rightarrow Q)$
(c) $\neg (Q \vee R)$
(d) $R \Rightarrow \neg P \wedge Q$

Solution

(a) I will go to a movie if and only if it is not raining and I have the time.
(b) I will go to a movie if and only if I have the time.
(c) It is not the case that I have the time or I will go to a movie.
(d) I will go to a movie, only if it is not raining or I have the time.

1.1.2 WELL-FORMED FORMULAS

Consider the propositions $P \wedge Q$ and $Q \wedge P$. The truth tables of these two propositions are identical irrespective of any proposition in place of P and any proposition in place of Q. So we can develop the concept of a propositional variable (corresponding to propositions) and well-formed formulas (corresponding to propositions involving connectives).

Definition 1.1 A propositional variable is a symbol representing any proposition. We note that usually a real variable is represented by the symbol x. This means that x is not a real number but can take a real value. Similarly, a propositional variable is not a proposition but can be replaced by a proposition.

Usually a mathematical object can be defined in terms of the property/condition satisfied by the mathematical object. Another way of defining a mathematical object is by recursion. Initially some objects are declared to follow the definition. The process by which more objects can be constructed is specified. This way of defining a mathematical object is called a *recursive* definition. This corresponds to a function calling itself in a programming language.

The factorial $n!$ can be defined as $n(n - 1) \ldots 2.1$. The recursive definition of $n!$ is as follows:

$$0! = 1, \qquad n! = n(n - 1)!$$

Definition 1.2 A well-formed formula (wff) is defined recursively as follows:

(i) If P is a propositional variable, then it is a wff.
(ii) If α is a wff, then $\neg \alpha$ is a wff.
(iii) If α and β are well-formed formulas, then $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \Rightarrow \beta)$, and $(\alpha \Leftrightarrow \beta)$ are well-formed formulas.
(iv) A string of symbols is a wff if and only if it is obtained by a finite number of applications of (i)–(iii).

http://engineeringbooks.net

Notes: (1) A wff is not a proposition, but if we substitute a proposition in place of a propositional variable, we get a proposition. For example:

(i) $\neg (P \vee Q) \wedge (\neg Q \wedge R) \Rightarrow Q$ is a wff.
(ii) $(\neg P \wedge Q) \Leftrightarrow Q$ is a wff.

(2) We can drop parentheses when there is no ambiguity. For example, in propositions we can remove the outermost parentheses. We can also specify the hierarchy of connectives and avoid parentheses.

For the sake of convenience, we can refer to a wff as a formula.

1.1.3 Truth Table for a Well-formed Formula

If we replace the propositional variables in a formula α by propositions, we get a proposition involving connectives. The table giving the truth values of such a proposition obtained by replacing the propositional variables by arbitrary propositions is called the truth table of α.

If α involves n propositional constants, then we have 2^n possible combinations of truth values of propositions replacing the variables.

EXAMPLE 1.5

Obtain the truth table for $\alpha = (P \vee Q) \wedge (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Solution

The truth values of the given wff are shown in Table 1.7.

TABLE 1.7 Truth Table of Example 1.5

P	Q	$P \vee Q$	$P \Rightarrow Q$	$(P \vee Q) \wedge (P \Rightarrow Q)$	$(Q \Rightarrow P)$	α
T	T	T	T	T	T	T
T	F	T	F	F	T	F
F	T	T	T	T	F	F
F	F	F	T	F	T	F

EXAMPLE 1.6

Construct the truth table for $\alpha = (P \vee Q) \Rightarrow ((P \vee R) \Rightarrow (R \vee Q))$.

Solution

The truth values of the given formula are shown in Table 1.8.

http://engineeringbooks.net

TABLE 1.8 Truth Table of Example 1.6

P	Q	R	$P \vee R$	$R \vee Q$	$(P \vee R) \Rightarrow (R \vee Q)$	$(P \vee Q)$	α
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	T	T	T	T
T	F	F	T	F	F	T	F
F	T	T	T	T	T	T	T
F	T	F	F	T	T	T	T
F	F	T	T	T	T	F	T
F	F	F	F	F	T	F	T

Some formulas have the truth value T for all possible assignments of truth values to the propositional variables. For example, $P \vee \neg P$ has the truth value T irrespective of the truth value of P. Such formulas are called *tautologies.*

Definition 1.3 A tautology or a universally true formula is a well-formed formula whose truth value is T for all possible assignments of truth values to the propositional variables.

For example, $P \vee \neg P$, $(P \wedge Q) \Rightarrow P$, and $((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)$ are tautologies.

Note: When it is not clear whether a given formula is a tautology, we can construct the truth table and verify that the truth value is T for all combinations of truth values of the propositional variables appearing in the given formula.

EXAMPLE 1.7

Show that $\alpha = (P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$ is a tautology.

Solution

We give the truth values of α in Table 1.9.

TABLE 1.9 Truth Table of Example 1.7

P	Q	R	$Q \Rightarrow R$	$P \Rightarrow (Q \Rightarrow R)$	$P \Rightarrow Q$	$P \Rightarrow R$	$(P \Rightarrow Q) \Rightarrow (P \Rightarrow R)$	α
T	T	T	T	T	T	T	T	T
T	T	F	F	F	T	F	F	T
T	F	T	T	T	F	T	T	T
T	F	F	T	T	F	F	T	T
F	T	T	T	T	T	T	T	T
F	T	F	F	T	T	T	T	T
F	F	T	T	T	T	T	T	T
F	F	F	T	T	T	T	T	T

Definition 1.4 A contradiction (or absurdity) is a wff whose truth value is F for all possible assignments of truth values to the propositional variables. For example,

$$P \wedge \neg P \qquad \text{and} \qquad (P \wedge Q) \wedge \neg Q$$

are contradictions.

Note: α is a contradiction if and only if $\neg \alpha$ is a tautology.

http://engineeringbooks.net

1.1.4 Equivalence of Well-formed Formulas

Definition 1.5 Two wffs α and β in propositional variables $P_1, P_2, \ldots, P_n$ are equivalent (or logically equivalent) if the formula $\alpha \Leftrightarrow \beta$ is a tautology. When α and β are equivalent, we write $\alpha \equiv \beta$.

Notes: (1) The wffs α and β are equivalent if the truth tables for α and β are the same. For example,

$$P \wedge Q \equiv Q \wedge P \qquad \text{and} \qquad P \wedge P \equiv P$$

(2) It is important to note the difference between $\alpha \Leftrightarrow \beta$ and $\alpha \equiv \beta$. $\alpha \Leftrightarrow \beta$ is a formula, whereas $\alpha \equiv \beta$ is not a formula but it denotes the relation between α and β.

EXAMPLE 1.8

Show that $(P \Rightarrow (Q \vee R)) \equiv ((P \Rightarrow Q) \vee (P \Rightarrow R))$.

Solution

Let $\alpha = (P \Rightarrow (Q \vee R))$ and $\beta = ((P \Rightarrow Q) \vee (P \Rightarrow R))$. We construct the truth values of α and β for all assignments of truth values to the variables P, Q and R. The truth values of α and β are given in Table 1.10.

TABLE 1.10 Truth Table of Example 1.8

P	Q	R	$Q \vee R$	$P \Rightarrow (Q \vee R)$	$P \Rightarrow Q$	$P \Rightarrow R$	$(P \Rightarrow Q) \vee (P \Rightarrow R)$
T	T	T	T	T	T	T	T
T	T	F	T	T	T	F	T
T	F	T	T	T	F	T	T
T	F	F	F	F	F	F	F
F	T	T	T	T	T	T	T
F	T	F	T	T	T	T	T
F	F	T	T	T	T	T	T
F	F	F	F	T	T	T	T

As the columns corresponding to α and β coincide, $\alpha \equiv \beta$.

As the truth value of a tautology is T, irrespective of the truth values of the propositional variables, we denote any tautology by **T**. Similarly, we denote any contradiction by **F**.

1.1.5 Logical Identities

Some equivalences are useful for deducing other equivalences. We call them identities and give a list of such identities in Table 1.11.

The identities I_1–I_{12} can be used to simplify formulas. If a formula β is part of another formula α, and β is equivalent to β', then we can replace β by β' in α and the resulting wff is equivalent to α.

http://engineeringbooks.net

TABLE 1.11 Logical Identities

I_1	Idempotent laws: $P \vee P \equiv P, \quad P \wedge P \equiv P$
I_2	Commutative laws: $P \vee Q \equiv Q \vee P, \quad P \wedge Q \equiv Q \wedge P$
I_3	Associative laws: $P \vee (Q \vee R) \equiv (P \vee Q) \vee R, \quad P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$
I_4	Distributive laws: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R), \quad P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
I_5	Absorption laws: $P \vee (P \wedge Q) \equiv P, \quad P \wedge (P \vee Q) \equiv P$
I_6	DeMorgan's laws: $\neg (P \vee Q) \equiv \neg P \wedge \neg Q, \quad \neg (P \wedge Q) \equiv \neg P \vee \neg Q$
I_7	Double negation: $P \equiv \neg (\neg P)$
I_8	$P \vee \neg P \equiv \mathbf{T}, \quad P \wedge \neg P \equiv \mathbf{F}$
I_9	$P \vee \mathbf{T} \equiv \mathbf{T}, \quad P \wedge \mathbf{T} \equiv P, \quad P \vee \mathbf{F} \equiv P, \quad P \wedge \mathbf{F} \equiv \mathbf{F}$
I_{10}	$(P \Rightarrow Q) \wedge (P \Rightarrow \neg Q) \equiv \neg P$
I_{11}	Contrapositive: $P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$
I_{12}	$P \Rightarrow Q \equiv (\neg P \vee Q)$

EXAMPLE 1.9

Show that $(P \wedge Q) \vee (P \wedge \neg Q) \equiv P$.

Solution

L.H.S. $= (P \wedge Q) \vee (P \wedge \neg Q)$

$\equiv P \wedge (Q \vee \neg Q)$ by using the distributive law (i.e. I_4)

$\equiv P \wedge \mathbf{T}$ by using I_8

$\equiv P$ by using I_9

$=$ R.H.S.

EXAMPLE 1.10

Show that $(P \Rightarrow Q) \wedge (R \Rightarrow Q) \equiv (P \vee R) \Rightarrow Q$

Solution

L.H.S. $= (P \Rightarrow Q) \wedge (R \Rightarrow Q)$

$\equiv (\neg P \vee Q) \wedge (\neg R \vee Q)$ by using I_{12}

$\equiv (Q \vee \neg P) \wedge (Q \vee \neg R)$ by using the commutative law (i.e. I_2)

$\equiv Q \vee (\neg P \wedge \neg R)$ by using the distributive law (i.e. I_4)

$\equiv Q \vee (\neg (P \vee R))$ by using the DeMorgan's law (i.e. I_6)

$\equiv (\neg (P \vee R)) \vee Q$ by using the commutative law (i.e. I_2)

$\equiv (P \vee R) \Rightarrow Q$ by using I_{12}

$=$ R.H.S.

http://engineeringbooks.net

1.2 NORMAL FORMS OF WELL-FORMED FORMULAS

We have seen various well-formed formulas in terms of two propositional variables, say, P and Q. We also know that two such formulas are equivalent if and only if they have the same truth table. The number of distinct truth tables for formulas in P and Q is 2^4. (As the possible combinations of truth values of P and Q are *TT, TF, FT, FF,* the truth table of any formula in P and Q has four rows. So the number of distinct truth tables is 2^4.) Thus there are only 16 distinct (nonequivalent) formulas, and any formula in P and Q is equivalent to one of these 16 formulas.

In this section we give a method of reducing a given formula to an equivalent form called the 'normal form'. We also use 'sum' for disjunction, 'product' for conjunction, and 'literal' either for P or for $\neg P$, where P is any propositional variable.

Definition 1.6 An elementary product is a product of literals. An elementary sum is a sum of literals. For example, $P \wedge \neg Q$, $\neg P \wedge \neg Q$, $P \wedge Q$, $\neg P \wedge Q$ are elementary products. And $P \vee \neg Q$, $P \vee \neg R$ are elementary sums.

Definition 1.7 A formula is in disjunctive normal form if it is a sum of elementary products. For example, $P \vee (Q \wedge R)$ and $P \vee (\neg Q \wedge R)$ are in disjunctive normal form. $P \wedge (Q \vee R)$ is not in disjunctive normal form.

1.2.1 CONSTRUCTION TO OBTAIN A DISJUNCTIVE NORMAL FORM OF A GIVEN FORMULA

Step 1 Eliminate $\Rightarrow$ and $\Leftrightarrow$ using logical identities. (We can use I_{12}, i.e. $P \Rightarrow Q \equiv (\neg P \vee Q)$.)

Step 2 Use DeMorgan's laws (I_6) to eliminate $\neg$ before sums or products. The resulting formula has $\neg$ only before the propositional variables, i.e. it involves sum, product and literals.

Step 3 Apply distributive laws (I_4) repeatedly to eliminate the product of sums. The resulting formula will be a sum of products of literals, i.e. sum of elementary products.

EXAMPLE 1.11

Obtain a disjunctive normal form of

$$P \vee (\neg P \Rightarrow (Q \vee (Q \Rightarrow \neg R)))$$

Solution

$$\begin{aligned} & P \vee (\neg P \Rightarrow (Q \vee (Q \Rightarrow \neg R))) \\ & \quad \equiv P \vee (\neg P \Rightarrow (Q \vee (\neg Q \vee \neg R))) && \text{(step 1 using } I_{12}) \\ & \quad \equiv P \vee (P \vee (Q \vee (\neg Q \vee \neg R))) && \text{(step 1 using } I_{12} \text{ and } I_7) \end{aligned}$$

http://engineeringbooks.net

$\equiv P \vee P \vee Q \vee \neg Q \vee \neg R$ by using I_3

$\equiv P \vee Q \vee \neg Q \vee \neg R$ by using I_1

Thus, $P \vee Q \vee \neg Q \vee \neg R$ is a disjunctive normal form of the given formula.

EXAMPLE 1.12

Obtain the disjunctive normal form of

$$(P \wedge \neg (Q \wedge R)) \vee (P \Rightarrow Q)$$

Solution

$(P \wedge \neg (Q \wedge R)) \vee (P \Rightarrow Q)$

$\equiv (P \wedge \neg (Q \wedge R) \vee (\neg P \vee Q)$ (step 1 using I_{12})

$\equiv (P \wedge (\neg Q \vee \neg R)) \vee (\neg P \vee Q)$ (step 2 using I_7)

$\equiv (P \wedge \neg Q) \vee (P \wedge \neg R) \vee \neg P \vee Q$ (step 3 using I_4 and I_3)

Therefore, $(P \wedge \neg Q) \vee (P \wedge \neg R) \vee \neg P \vee Q$ is a disjunctive normal form of the given formula.

For the same formula, we may get different disjunctive normal forms. For example, $(P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R)$ and $P \wedge Q$ are disjunctive normal forms of $P \wedge Q$. So, we introduce one more normal form, called *the principal disjunctive normal form* or the *sum-of-products canonical form* in the next definition. The advantages of constructing the principal disjunctive normal form are:

(i) For a given formula, its principal disjunctive normal form is unique.
(ii) Two formulas are equivalent if and only if their principal disjunctive normal forms coincide.

Definition 1.8 A minterm in n propositional variables $P_1, \ldots, P_n$ is $Q_1 \wedge Q_2 \ldots \wedge Q_n$, where each Q_i is either P_i or $\neg P_i$.

For example, the minterms in P_1 and P_2 are $P_1 \wedge P_2$, $\neg P_1 \wedge P_2$, $P_1 \wedge \neg P_2$, $\neg P_1 \wedge \neg P_2$. The number of minterms in n variables is 2^n.

Definition 1.9 A formula α is in principal disjunctive normal form if α is a sum of minterms.

1.2.2 Construction to Obtain the Principal Disjunctive Normal Form of a Given Formula

Step 1 Obtain a disjunctive normal form.

Step 2 Drop the elementary products which are contradictions (such as $P \wedge \neg P$).

Step 3 If P_i and $\neg P_i$ are missing in an elementary product α, replace α by $(\alpha \wedge P_i) \vee (\alpha \wedge \neg P_i)$.

http://engineeringbooks.net

Step 4 Repeat step 3 until all the elementary products are reduced to sum of minterms. Use the idempotent laws to avoid repetition of minterms.

EXAMPLE 1.13

Obtain the canonical sum-of-products form (i.e. the principal disjunctive normal form) of

$$\alpha = P \vee (\neg P \wedge \neg Q \wedge R)$$

Solution

Here α is already in disjunctive normal form. There are no contradictions. So we have to introduce the missing variables (step 3). $\neg P \wedge \neg Q \wedge R$ in α is already a minterm. Now,

$$\begin{aligned} P &\equiv (P \wedge Q) \vee (P \wedge \neg Q) \\ &\equiv ((P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R)) \vee (P \wedge \neg Q \wedge R) \vee (P \wedge \neg Q \wedge \neg R) \\ &\equiv ((P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R)) \vee ((P \wedge \neg Q \wedge R) \vee (P \wedge \neg Q \wedge \neg R)) \end{aligned}$$

Therefore, the canonical sum-of-products form of α is

$$\begin{aligned} &(P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge R) \\ &\quad \vee (P \wedge \neg Q \wedge \neg R) \vee (\neg P \wedge \neg Q \wedge R) \end{aligned}$$

EXAMPLE 1.14

Obtain the principal disjunctive normal form of

$$\alpha = (\neg P \vee \neg Q) \Rightarrow (\neg P \wedge R)$$

Solution

$$\begin{aligned} \alpha &= (\neg P \vee \neg Q) \Rightarrow (\neg P \wedge R) \\ &\equiv (\neg(\neg P \vee \neg Q)) \vee (\neg P \wedge R) && \text{by using } I_{12} \\ &\equiv (P \wedge Q) \vee (\neg P \wedge R) && \text{by using DeMorgan's law} \\ &\equiv ((P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R)) \vee ((\neg P \wedge R \wedge Q) \vee (\neg P \wedge R \wedge \neg Q)) \\ &\equiv (P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R) \vee (\neg P \wedge Q \wedge R) \vee (\neg P \wedge \neg Q \wedge R) \end{aligned}$$

So, the principal disjunctive normal form of α is

$$(P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R) \vee (\neg P \wedge Q \wedge R) \vee (\neg P \wedge \neg Q \wedge R)$$

A minterm of the form $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$ can be represented by $a_1 a_2 \ldots a_n$, where $a_i = 0$ if $Q_i = \neg P_i$ and $a_i = 1$ if $Q_i = P_i$. So the principal disjunctive normal form can be represented by a 'sum' of binary strings. For example, $(P \wedge Q \wedge R) \vee (P \wedge Q \wedge \neg R) \vee (\neg P \wedge \neg Q \wedge R)$ can be represented by $111 \vee 110 \vee 001$.

The minterms in the two variables P and Q are 00, 01, 10, and 11. Each wff is equivalent to its principal disjunctive normal form. Every principal disjunctive normal form corresponds to the minterms in it, and hence to a

http://engineeringbooks.net

subset of {00, 01, 10, 11}. As the number of subsets is 2^4, the number of distinct formulas is 16. (Refer to the remarks made at the beginning of this section.)

The truth table and the principal disjunctive normal form of α are closely related. Each minterm corresponds to a particular assignment of truth values to the variables yielding the truth value T to α. For example, $P \wedge Q \wedge \neg R$ corresponds to the assignment of T, T, F to P, Q and R, respectively. So, if the truth table of α is given, then the minterms are those which correspond to the assignments yielding the truth value T to α.

EXAMPLE 1.15

For a given formula α, the truth values are given in Table 1.12. Find the principal disjunctive normal form.

TABLE 1.12 Truth Table of Example 1.15

P	Q	R	α
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	T

Solution

We have T in the α-column corresponding to the rows 1, 4, 5 and 8. The minterm corresponding to the first row is $P \wedge Q \wedge R$.

Similarly, the minterms corresponding to rows 4, 5 and 8 are respectively $P \wedge \neg Q \wedge \neg R$, $\neg P \wedge Q \wedge R$ and $\neg P \wedge \neg Q \wedge \neg R$. Therefore, the principal disjunctive normal form of α is

$$(P \wedge Q \wedge R) \vee (P \wedge \neg Q \wedge \neg R) \vee (\neg P \wedge Q \wedge R) \vee (\neg P \wedge \neg Q \wedge \neg R)$$

We can form the 'dual' of the disjunctive normal form which is termed the conjunctive normal form.

Definition 1.10 A formula is in conjunctive normal form if it is a product of elementary sums.

If α is in disjunctive normal form, then $\neg \alpha$ is in conjunctive normal form. (This can be seen by applying the DeMorgan's laws.) So to obtain the conjunctive normal form of α, we construct the disjunctive normal form of $\neg \alpha$ and use negation.

Definition 1.11 A maxterm in n propositional variables $P_1, P_2, \cdots, P_n$ is $Q_1 \vee Q_2 \vee \cdots \vee Q_n$, where each Q_i is either P_i or $\neg P_i$.

http://engineeringbooks.net

Definition 1.12 A formula α is in principal conjunctive normal form if α is a product of maxterms. For obtaining the principal conjunctive normal form of α, we can construct the principal disjunctive normal form of $\neg \alpha$ and apply negation.

EXAMPLE 1.16

Find the principal conjunctive normal form of $\alpha = P \vee (Q \Rightarrow R)$.

Solution

$$\begin{aligned}
\neg \alpha &= \neg (P \vee (Q \Rightarrow R)) \\
&\equiv \neg (P \vee (\neg Q \vee R)) && \text{by using } I_{12} \\
&\equiv \neg P \wedge (\neg (\neg Q \vee R)) && \text{by using DeMorgan's law} \\
&\equiv \neg P \wedge (Q \wedge \neg R) && \text{by using DeMorgan's law and } I_7
\end{aligned}$$

$\neg P \wedge Q \wedge \neg R$ is the principal disjunctive normal form of $\neg \alpha$. Hence, the principal conjunctive normal form of α is

$$\neg (\neg P \wedge Q \wedge \neg R) = P \vee \neg Q \vee R$$

The logical identities given in Table 1.11 and the normal forms of well-formed formulas bear a close resemblance to identities in Boolean algebras and normal forms of Boolean functions. Actually, the propositions under $\vee$, $\wedge$ and $\neg$ form a Boolean algebra if the equivalent propositions are identified. **T** and **F** act as bounds (i.e. 0 and 1 of a Boolean algebra). Also, the statement formulas form a Boolean algebra under $\vee$, $\wedge$ and $\neg$ if the equivalent formulas are identified.

The normal forms of well-formed formulas correspond to normal forms of Boolean functions and we can 'minimize' a formula in a similar manner.

1.3 RULES OF INFERENCE FOR PROPOSITIONAL CALCULUS (STATEMENT CALCULUS)

In logical reasoning, a certain number of propositions are assumed to be true, and based on that assumption some other propositions are derived (deduced or inferred). In this section we give some important rules of logical reasoning or rules of inference. The propositions that are assumed to be true are called *hypotheses* or *premises*. The proposition derived by using the rules of inference is called a *conclusion*. The process of deriving conclusions based on the assumption of premises is called a *valid argument*. So in a valid argument we are concerned with the process of arriving at the conclusion rather than obtaining the conclusion.

The rules of inference are simply tautologies in the form of implication (i.e. $P \Rightarrow Q$). For example, $P \Rightarrow (P \vee Q)$ is such a tautology, and it is a rule of inference. We write this in the form $\dfrac{P}{\therefore P \vee Q}$. Here P denotes a premise. The proposition below the line, i.e. $P \vee Q$ is the conclusion.

http://engineeringbooks.net

We give in Table 1.13 some of the important rules of inference. Of course, we can derive more rules of inference and use them in valid arguments.

For valid arguments, we can use the rules of inference given in Table 1.13. As the logical identities given in Table 1.11 are two-way implications, we can also use them as rules of inference.

TABLE 1.13 Rules of Inference

Rule of inference	*Implication form*
RI_1: Addition $\frac{P}{\therefore P \vee Q}$	$P \Rightarrow (P \vee Q)$
RI_2: Conjunction $\frac{P \quad Q}{\therefore P \wedge Q}$	$P \wedge Q \Rightarrow P \wedge Q$
RI_3: Simplification $\frac{P \wedge Q}{\therefore P}$	$(P \wedge Q) \Rightarrow P$
RI_4: Modus ponens $\frac{P \quad P \Rightarrow Q}{\therefore Q}$	$(P \wedge (P \Rightarrow Q)) \Rightarrow Q$
RI_5: Modus tollens $\frac{\neg Q \quad P \Rightarrow Q}{\therefore \neg P}$	$(\neg Q \wedge (P \Rightarrow Q)) \Rightarrow \neg Q$
RI_6: Disjunctive syllogism $\frac{\neg P \quad P \vee Q}{\therefore Q}$	$(\neg P \wedge (P \vee Q)) \Rightarrow Q$
RI_7: Hypothetical syllogism $\frac{P \Rightarrow Q \quad Q \Rightarrow R}{\therefore P \Rightarrow R}$	$((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)$
RI_8: Constructive dilemma $\frac{(P \Rightarrow Q) \wedge (R \Rightarrow S) \quad P \vee R}{\therefore Q \vee S}$	$((P \Rightarrow Q) \wedge (R \Rightarrow S) \wedge (P \vee R)) \Rightarrow (Q \vee S)$
RI_9: Destructive dilemma $\frac{(P \Rightarrow Q) \wedge (R \Rightarrow S) \quad \neg Q \vee \neg S}{\therefore P \vee R}$	$((P \Rightarrow Q) \wedge (R \Rightarrow S) \wedge (\neg Q \vee \neg S)) \Rightarrow (\neg P \vee \neg R)$

http://engineeringbooks.net

EXAMPLE 1.17

Can we conclude S from the following premises?

(i) $P \Rightarrow Q$
(ii) $P \Rightarrow R$
(iii) $\neg (Q \wedge R)$
(iv) $S \vee P$

Solution

The valid argument for deducing S from the given four premises is given as a sequence. On the left, the well-formed formulas are given. On the right, we indicate whether the proposition is a premise (hypothesis) or a conclusion. If it is a conclusion, we indicate the premises and the rules of inference or logical identities used for deriving the conclusion.

1. $P \Rightarrow Q$	Premise (i)
2. $P \Rightarrow R$	Premise (ii)
3. $(P \Rightarrow Q) \wedge (P \Rightarrow R)$	Lines 1, 2 and RI_2
4. $\neg (Q \wedge R)$	Premise (iii)
5. $\neg Q \vee \neg R$	Line 4 and DeMorgan's law (I_6)
6. $\neg P \vee \neg P$	Lines 3, 5 and destructive dilemma (RI_9)
7. $\neg P$	Idempotent law I_1
8. $S \vee P$	Premise (iv)
9. S	Lines 7, 8 and disjunctive syllogism RI_6

Thus, we can conclude S from the given premises.

EXAMPLE 1.18

Derive S from the following premises using a valid argument:

(i) $P \Rightarrow Q$
(ii) $Q \Rightarrow \neg R$
(iii) $P \vee S$
(iv) R

Solution

1. $P \Rightarrow Q$	Premise (i)
2. $Q \Rightarrow \neg R$	Premise (ii)
3. $P \Rightarrow \neg R$	Lines 1, 2 and hypothetical syllogism RI_7
4. R	Premise (iv)
5. $\neg (\neg R)$	Line 4 and double negation I_7
6. $\neg P$	Lines 3, 5 and modus tollens RI_5
7. $P \vee S$	Premise (iii)
8. S	Lines 6, 7 and disjunctive syllogism RI_6

Thus, we have derived S from the given premises.

http://engineeringbooks.net

EXAMPLE 1.19

Check the validity of the following argument:

If Ram has completed B.E. (Computer Science) or MBA, then he is assured of a good job. If Ram is assured of a good job, he is happy. Ram is not happy. So Ram has not completed MBA.

Solution

We can name the propositions in the following way:

P denotes 'Ram has completed B.E. (Computer Science)'.
Q denotes 'Ram has completed MBA'.
R denotes 'Ram is assured of a good job'.
S denotes 'Ram is happy'.

The given premises are:

(i) $(P \vee Q) \Rightarrow R$
(ii) $R \Rightarrow S$
(iii) $\neg S$

The conclusion is $\neg Q$.

1.	$(P \vee Q) \Rightarrow R$	Premise (i)
2.	$R \Rightarrow S$	Premise (ii)
3.	$(P \vee Q) \Rightarrow S$	Lines 1, 2 and hypothetical syllogism RI_7
4.	$\neg S$	Premise (iii)
5.	$\neg (P \vee Q)$	Lines 3, 4 and modus tollens RI_5
6.	$\neg P \wedge \neg Q$	DeMorgan's law I_6
7.	$\neg Q$	Line 6 and simplification RI_3

Thus the argument is valid.

EXAMPLE 1.20

Test the validity of the following argument:

If milk is black then every cow is white. If every cow is white then it has four legs. If every cow has four legs then every buffalo is white and brisk. The milk is black.

Therefore, the buffalo is white.

Solution

We name the propositions in the following way:

P denotes 'The milk is black'.
Q denotes 'Every cow is white'.
R denotes 'Every cow has four legs'.
S denotes 'Every buffalo is white'.
T denotes 'Every buffalo is brisk'.

http://engineeringbooks.net

The given premises are:

(i) $P \Rightarrow Q$
(ii) $Q \Rightarrow R$
(iii) $R \Rightarrow S \wedge T$
(iv) P

The conclusion is S.

1. P	Premise (iv)
2. $P \Rightarrow Q$	Premise (i)
3. Q	Modus ponens RI_4
4. $Q \Rightarrow R$	Premise (ii)
5. R	Modus ponens RI_4
6. $R \Rightarrow S \wedge T$	Premise (iii)
7. $S \wedge T$	Modus ponens RI_4
8. S	Simplification RI_3

Thus the argument is valid.

1.4 PREDICATE CALCULUS

Consider two propositions 'Ram is a student', and 'Sam is a student'. As propositions, there is no relation between them, but we know they have something in common. Both Ram and Sam share the property of being a student. We can replace the two propositions by a single statement 'x is a student'. By replacing x by Ram or Sam (or any other name), we get many propositions. The common feature expressed by 'is a student' is called a *predicate*. In predicate calculus we deal with sentences involving predicates. Statements involving predicates occur in mathematics and programming languages. For example, '$2x + 3y = 4z$', 'IF (D. GE. 0.0) GO TO 20' are statements in mathematics and FORTRAN, respectively, involving predicates. Some logical deductions are possible only by 'separating' the predicates.

1.4.1 PREDICATES

A part of a declarative sentence describing the properties of an object or relation among objects is called a predicate. For example, 'is a student' is a predicate.

Sentences involving predicates describing the property of objects are denoted by $P(x)$, where P denotes the predicate and x is a variable denoting any object. For example, $P(x)$ can denote 'x is a student'. In this sentence, x is a variable and P denotes the predicate 'is a student'.

The sentence 'x is the father of y' also involves a predicate 'is the father of'. Here the predicate describes the relation between two persons. We can write this sentence as $F(x, y)$. Similarly, $2x + 3y = 4z$ can be described by $S(x, y, z)$.

http://engineeringbooks.net

Note: Although $P(x)$ involving a predicate looks like a proposition, it is not a proposition. As $P(x)$ involves a variable x, we cannot assign a truth value to $P(x)$. However, if we replace x by an individual object, we get a proposition. For example, if we replace x by Ram in $P(x)$, we get the proposition 'Ram is a student'. (We can denote this proposition by P(Ram).) If we replace x by 'A cat', then also we get a proposition (whose truth value is F). Similarly, $S(2, 0, 1)$ is the proposition $2 \cdot 2 + 3 \cdot 0 = 4 \cdot 1$ (whose truth value is T). Also, $S(1, 1, 1)$ is the proposition $2 \cdot 1 + 3 \cdot 1 = 4 \cdot 1$ (whose truth value is F).

The following definition is regarding the possible 'values' which can be assigned to variables.

Definition 1.13 For a declarative sentence involving a predicate, the universe of discourse, or simply the universe, is the set of all possible values which can be assigned to variables.

For example, the universe of discourse for $P(x)$: 'x is a student', can be taken as the set of all human names; the universe of discourse for $E(n)$: 'n is an even integer', can be taken as the set of all integers (or the set of all real numbers).

Note: In most examples, the universe of discourse is not specified but can be easily given.

Remark We have seen that by giving values to variables, we can get propositions from declarative sentences involving predicates. Some sentences involving variables can also be assigned truth values. For example, consider 'There exists x such that $x^2 = 5$', and 'For all x, $x^2 = (-x)^2$'. Both these sentences can be assigned truth values (T in both cases). 'There exists' and 'For all' quantify the variables.

Universal and Existential Quantifiers

The phrase 'for all' (denoted by $\forall$) is called the universal quantifier. Using this symbol, we can write 'For all x, $x^2 = (-x)^2$' as $\forall x\, Q(x)$, where $Q(x)$ is '$x^2 = (-x)^2$'.

The phrase 'there exists' (denoted by $\exists$) is called the existential quantifier.

The sentence 'There exists x such that $x^2 = 5$' can be written as $\exists x\, R(x)$, where $R(x)$ is '$x^2 = 5$'.

$P(x)$ in $\forall x\, P(x)$ or in $\exists x\, P(x)$ is called the scope of the quantifier $\forall$ or $\exists$.

Note: The symbol $\forall$ can be read as 'for every', 'for any', 'for each', 'for arbitrary'. The symbol $\exists$ can be read as 'for some', for 'at least one'.

When we use quantifiers, we should specify the universe of discourse. If we change the universe of discourse, the truth value may change. For example, consider $\exists x\, R(x)$, where $R(x)$ is $x^2 = 5$. If the universe of discourse is the set of all integers, then $\exists x\, R(x)$ is false. If the universe of discourse is the set of all real numbers, then $\exists x\, R(x)$ is true (when $x = \pm\sqrt{5}$, $x^2 = 5$).

http://engineeringbooks.net

The logical connectives involving predicates can be used for declarative sentences involving predicates. The following example illustrates the use of connectives.

EXAMPLE 1.21

Express the following sentences involving predicates in symbolic form:

1. All students are clever.
2. Some students are not successful.
3. Every clever student is successful.
4. There are some successful students who are not clever.
5. Some students are clever and successful.

Solution

As quantifiers are involved, we have to specify the universe of discourse. We can take the universe of discourse as the set of all students.

Let $C(x)$ denote 'x is clever'.
Let $S(x)$ denote 'x is successful'.

Then the sentence 1 can be written as $\forall x\, C(x)$. The sentences 2–5 can be written as

$$\exists x\, (\neg S(x)), \qquad \forall x\, (C(x) \Rightarrow S(x)),$$
$$\exists x\, (S(x) \wedge \neg C(x)), \qquad \exists x\, (C(x) \neg S(x))$$

1.4.2 Well-formed Formulas of Predicate Calculus

A well-formed formula (wff) of predicate calculus is a string of variables such as $x_1, x_2, \ldots, x_n$, connectives, parentheses and quantifiers defined recursively by the following rules:

(i) $P(x_1, \ldots, x_n)$ is a wff, where P is a predicate involving n variables $x_1, x_2, \ldots, x_n$.
(ii) If α is a wff, then $\neg \alpha$ is a wff.
(iii) If α and β are wffs, then $\alpha \vee \beta$, $\alpha \wedge \beta$, $\alpha \Rightarrow \beta$, $\alpha \Leftrightarrow \beta$ are also wffs.
(iv) If α is a wff and x is any variable, then $\forall x\, (\alpha)$, $\exists x\, (\alpha)$ are wffs.
(v) A string is a wff if and only if it is obtained by a finite number of applications of rules (i)–(iv).

Note: A proposition can be viewed as a sentence involving a predicate with 0 variables. So the propositions are wffs of predicate calculus by rule (i).

We call wffs of predicate calculus as predicate formulas for convenience. The well-formed formulas introduced in Section 1.1 can be called proposition formulas (or statement formulas) to distinguish them from predicate formulas.

http://engineeringbooks.net

Definition 1.14 Let α and β be two predicate formulas in variables $x_1, \ldots, x_n$, and let U be a universe of discourse for α and β. Then α and β are equivalent to each other over U if for every possible assignment of values to each variable in α and β the resulting statements have the same truth values. We can write $\alpha = \beta$ over U.

We say that α and β are equivalent to each other ($\alpha \equiv \beta$) if $\alpha \equiv \beta$ over U for every universe of discourse U.

Remark In predicate formulas the predicate variables may or may not be quantified. We can classify the predicate variables in a predicate formula, depending on whether they are quantified or not. This leads to the following definitions.

Definition 1.15 If a formula of the form $\exists x\, P(x)$ or $\forall x\, P(x)$ occurs as part of a predicate formula α, then such part is called an x-bound part of α, and the occurrence of x is called a bound occurrence of x. An occurrence of x is free if it is not a bound occurrence. A predicate variable in α is free if its occurrence is free in any part of α.

In $\alpha = (\exists x_1\, P(x_1, x_2)) \wedge (\forall x_2\, Q(x_2, x_3))$, for example, the occurrence of x_1 in $\exists x_1\, P(x_1, x_2)$ is a bound occurrence and that of x_2 is free. In $\forall x_2\, Q(x_2, x_3)$, the occurrence of x_2 is a bound occurrence. The occurrence of x_3 in α is free.

Note: The quantified parts of a predicate formula such as $\forall x\, P(x)$ or $\exists x\, P(x)$ are propositions. We can assign values from the universe of discourse only to the free variables in a predicate formula α.

Definition 1.16 A predicate formula is valid if for all possible assignments of values from any universe of discourse to free variables, the resulting propositions have the truth value T.

Definition 1.17 A predicate formula is satisfiable if for some assignment of values to predicate variables the resulting proposition has the truth value T.

Definition 1.18 A predicate formula is unsatisfiable if for all possible assignments of values from any universe of discourse to predicate variables the resulting propositions have the truth value F.

We note that valid predicate formulas correspond to tautologies among proposition formulas and the unsatisfiable predicate formulas correspond to contradictions.

http://engineeringbooks.net

1.5 RULES OF INFERENCE FOR PREDICATE CALCULUS

Before discussing the rules of inference, we note that: (i) the proposition formulas are also the predicate formulas; (ii) the predicate formulas (where all the variables are quantified) are the proposition formulas. Therefore, all the rules of inference for the proposition formulas are also applicable to predicate calculus wherever necessary.

For predicate formulas not involving connectives such as $A(x)$, $P(x, y)$, we can get equivalences and rules of inference similar to those given in Tables 1.11 and 1.13. For Example, corresponding to I_6 in Table 1.11 we get $\neg (P(x) \vee Q(x)) \equiv \neg (P(x)) \wedge \neg (Q(x))$. Corresponding to RI_3 in Table 1.13 $P \wedge Q \Rightarrow P$, we get $P(x) \wedge Q(x) \Rightarrow P(x)$. Thus we can replace propositional variables by predicate variables in Tables 1.11 and 1.13.

Some necessary equivalences involving the two quantifiers and valid implications are given in Table 1.14.

TABLE 1.14 Equivalences Involving Quantifiers

I_{13}	Distributivity of $\exists$ over $\vee$: $\exists x (P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$ $\exists x (P \vee Q(x)) \equiv P \vee (\exists x Q(x))$
I_{14}	Distributivity of $\forall$ over $\wedge$: $\forall x (P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$ $\forall x (P \wedge Q(x)) \equiv P \wedge (\forall x Q(x))$
I_{15}	$\neg (\exists x P(x)) \equiv \forall x \neg (P(x))$
I_{16}	$\neg (\forall x P(x)) \equiv \exists x \neg (P(x))$
I_{17}	$\exists x (P \wedge Q(x)) \equiv P \wedge (\exists x Q(x))$
I_{18}	$\forall x (P \vee Q(x)) \equiv P \vee (\forall x Q(x))$
RI_{10}	$\forall x P(x) \Rightarrow \exists x P(x)$
RI_{11}	$\forall x P(x) \vee \forall x Q(x) \Rightarrow \forall x (P(x) \vee Q(x))$
RI_{12}	$\exists x (P(x) \wedge Q(x)) \Rightarrow \exists x P(x) \wedge \exists x Q(x)$

Sometimes when we wish to derive some conclusion from a given set of premises involving quantifiers, we may have to eliminate the quantifiers before applying the rules of inference for proposition formulas. Also, when the conclusion involves quantifiers, we may have to introduce quantifiers. The necessary rules of inference for addition and deletion of quantifiers are given in Table 1.15.

http://engineeringbooks.net

TABLE 1.15 Rules of Inference for Addition and Deletion of Quantifiers

RI_{13}:	Universal instantiation $\frac{\forall x\, P(x)}{\therefore P(c)}$ c is some element of the universe.
RI_{14}:	Existential instantiation $\frac{\exists x\, P(x)}{\therefore P(c)}$ c is some element for which $P(c)$ is true.
RI_{15}:	Universal generalization $\frac{P(x)}{\forall x\, P(x)}$ x should not be free in any of the given premises.
RI_{16}:	Existential generalization $\frac{P(c)}{\therefore \exists x\, P(x)}$ c is some element of the universe.

EXAMPLE 1.22

Discuss the validity of the following argument:

All graduates are educated.
Ram is a graduate.
Therefore, Ram is educated.

Solution

Let $G(x)$ denote 'x is a graduate'.

Let $E(x)$ denote 'x is educated'.

Let R denote 'Ram'.

So the premises are (i) $\forall x\,(G(x) \Rightarrow E(x))$ and (ii) $G(R)$. The conclusion is $E(R)$.

$\forall x\,(G(x) \Rightarrow E(x))$	Premise (i)
$G(R) \Rightarrow E(R)$	Universal instantiation RI_{13}
$G(R)$	Premise (ii)
$\therefore E(R)$	Modus ponens RI_4

Thus the conclusion is valid.

http://engineeringbooks.net

EXAMPLE 1.23

Discuss the validity of the following argument:

All graduates can read and write.
Ram can read and write.
Therefore, Ram is a graduate.

Solution

Let $G(x)$ denote 'x is a graduate'.
Let $L(x)$ denote 'x can read and write'.
Let R denote 'Ram'.
The premises are: $\forall x\,(G(x) \Rightarrow L(x))$ and $L(R)$.
The conclusion is $G(R)$.
$((G(R) \Rightarrow L(R)) \wedge L(R)) \Rightarrow G(R)$ is not a tautology.
So we cannot derive $G(R)$. For example, a school boy can read and write and he is not a graduate.

EXAMPLE 1.24

Discuss the validity of the following argument:

All educated persons are well behaved.
Ram is educated.
No well-behaved person is quarrelsome.
Therefore, Ram is not quarrelsome.

Solution

Let the universe of discourse be the set of all educated persons.
Let $P(x)$ denote 'x is well-behaved'.
Let y denote 'Ram'.
Let $Q(x)$ denote 'x is quarrelsome'.

So the premises are:

(i) $\forall x\,P(x)$.
(ii) y is a particular element of the universe of discourse.
(iii) $\forall x\,(P(x) \Rightarrow \neg Q(x))$.

To obtain the conclusion, we have the following arguments:

1. $\forall x\,P(x)$	Premise (i)
2. $P(y)$	Universal instantiation RI_{13}
3. $\forall x\,(P(x) \Rightarrow \neg Q(x))$	Premise (iii)
4. $P(y) \Rightarrow \neg Q(y)$	Universal instantiation RI_{13}
5. $P(y)$	Line 2
6. $\neg Q(y)$	Modus ponens RI_4

$\neg Q(y)$ means that 'Ram is not quarrelsome'. Thus the argument is valid.

http://engineeringbooks.net

1.6 SUPPLEMENTARY EXAMPLES

EXAMPLE 1.25

Write the following sentences in symbolic form:

(a) This book is interesting but the exercises are difficult.
(b) This book is interesting but the subject is difficult.
(c) This book is not interesting, the exercises are difficult but the subject is not difficult.
(d) If this book is interesting and the exercises are not difficult then the subject is not difficult.
(e) This book is interesting means that the subject is not difficult, and conversely.
(f) The subject is not difficult but this book is interesting and the exercises are difficult.
(g) The subject is not difficult but the exercises are difficult.
(h) Either the book is interesting or the subject is difficult.

Solution

Let P denote 'This book is interesting'.
Let Q denote 'The exercises are difficult'.
Let R denote 'The subject is difficult'.

Then:

(a) $P \wedge Q$
(b) $P \wedge R$
(c) $\neg P \wedge Q \wedge \neg R$
(d) $(P \wedge \neg Q) \Rightarrow \neg R$
(e) $P \Leftrightarrow \neg R$
(f) $(\neg R) \wedge (P \wedge Q)$
(g) $\neg R \wedge Q$
(h) $\neg P \vee R$

EXAMPLE 1.26

Construct the truth table for $\alpha = (\neg P \Leftrightarrow \neg Q) \Leftrightarrow Q \Leftrightarrow R$

Solution

The truth table is constructed as shown in Table 1.16.

http://engineeringbooks.net

TABLE 1.16 Truth Table of Example 1.26

P	Q	R	$Q \Leftrightarrow R$	$\neg P$	$\neg Q$	$\neg P \Leftrightarrow \neg Q$	α
T	T	T	T	F	F	T	T
T	T	F	F	F	F	T	F
T	F	T	F	F	T	F	T
T	F	F	T	F	T	F	F
F	T	T	T	T	F	F	F
F	T	F	F	T	F	F	T
F	F	T	F	T	T	T	F
F	F	F	T	T	T	T	T

EXAMPLE 1.27

Prove that: $\alpha = ((P \Rightarrow (Q \vee R)) \wedge (\neg Q)) \Rightarrow (P \Rightarrow R)$ is a tautology.

Solution

Let $\beta = (P \Rightarrow (Q \vee R)) \wedge (\neg Q)$

The truth table is constructed as shown in Table 1.17. From the truth table, we conclude that α is a tautology.

TABLE 1.17 Truth Table of Example 1.27

P	Q	R	$\neg Q$	$Q \vee R$	$P \Rightarrow (Q \vee R)$	β	$P \Rightarrow R$	α
T	T	T	F	T	T	F	T	T
T	T	F	F	T	T	F	F	T
T	F	T	T	T	T	T	T	T
T	F	F	T	F	F	F	F	T
F	T	T	F	T	T	F	T	T
F	T	F	F	T	T	F	T	T
F	F	T	T	T	T	T	T	T
F	F	F	T	F	T	T	T	T

EXAMPLE 1.28

State the converse, opposite and contrapositive to the following statements:

(a) If a triangle is isoceles, then two of its sides are equal.

(b) If there is no unemployment in India, then the Indians won't go to the USA for employment.

Solution

If $P \Rightarrow Q$ is a statement, then its converse, opposite and contrapositive statements are, $Q \Rightarrow P$, $\neg P \Rightarrow \neg Q$ and $\neg Q \Rightarrow \neg P$, respectively.

(a) Converse—If two of the sides of a triangle are equal, then the triangle is isoceles.

http://engineeringbooks.net

Opposite—If the triangle is not isoceles, then two of its sides are not equal.

Contrapositive—If two of the sides of a triangle are not equal, then the triangle is not isoceles.

(b) Converse—If the Indians won't go to the USA for employment, then there is no unemployment in India.

Opposite—If there is unemployment in India, then the Indians will go to the USA for employment.

(c) Contrapositive—If the Indians go to the USA for employment, then there is unemployment in India.

EXAMPLE 1.29

Show that:

$$(\neg P \wedge (\neg Q \wedge R)) \vee (Q \wedge R) \vee (P \wedge R) \Leftrightarrow R$$

Solution

$(\neg P \wedge (\neg Q \wedge R)) \vee (Q \wedge R) \vee (P \wedge R)$

$\Leftrightarrow ((\neg P \wedge \neg Q) \wedge R) \vee (Q \wedge R) \vee (P \wedge R)$ by using the associative law

$\Leftrightarrow (\neg (P \vee Q) \wedge R) \vee (Q \wedge R) \vee (P \wedge R)$ by using the DeMorgan's law

$\Leftrightarrow (\neg (P \vee Q) \wedge R) \vee (Q \vee P) \wedge R)$ by using the distributive law

$\Leftrightarrow (\neg (P \vee Q) \vee (P \vee Q) \wedge R$ by using the commutative and distributive laws

$\Leftrightarrow \mathbf{T} \wedge R$ by using I_8

$\Leftrightarrow R$ by using I_9

EXAMPLE 1.30

Using identities, prove that:

$Q \vee (P \wedge \neg Q) \vee (\neg P \wedge \neg Q)$ is a tautology

Solution

$Q \vee (P \wedge \neg Q) \vee (\neg P \wedge \neg Q)$

$\Leftrightarrow ((Q \vee P) \wedge (Q \vee \neg Q)) \vee (\neg P \wedge \neg Q)$ by using the distributive law

$\Leftrightarrow ((Q \vee P) \wedge \mathbf{T}) \vee (\neg P \wedge \neg Q)$ by using I_8

$\Leftrightarrow (Q \vee P) \vee \neg (P \vee Q)$ by using the DeMorgan's law and I_9

$\Leftrightarrow (P \vee Q) \vee \neg (P \vee Q)$ by using the commutative law

$\Leftrightarrow \mathbf{T}$ by using I_8

Hence the given formula is a tautology.

http://engineeringbooks.net

EXAMPLE 1.31

Test the validity of the following argument:

If I get the notes and study well, then I will get first class.
I didn't get first class.
So either I didn't get the notes or I didn't study well.

Solution

Let P denote 'I get the notes'.
Let Q denote 'I study well'.
Let R denote 'I will get first class.'
Let S denote 'I didn't get first class.'

The given premises are:

(i) $P \wedge Q \Rightarrow R$
(ii) $\neg R$

The conclusion is $\neg P \vee \neg Q$.

1. $P \wedge Q \Rightarrow R$	Premise (i)
2. $\neg R$	Premise (ii)
3. $\neg (P \wedge Q)$	Lines 1, 2 and modus tollens.
4. $\neg P \vee \neg Q$	DeMorgan's law

Thus the argument is valid.

EXAMPLE 1.32

Explain (a) the conditional proof rule and (b) the indirect proof.

Solution

(a) If we want to prove $A \Rightarrow B$, then we take A as a premise and construct a proof of B. This is called the conditional proof rule. It is denoted by CP.
(b) To prove a formula α, we construct a proof of $\neg \alpha \Rightarrow \mathbf{F}$. In particular, to prove $A \Rightarrow B$, we construct a proof of $A \wedge \neg B \Rightarrow \mathbf{F}$.

EXAMPLE 1.33

Test the validity of the following argument:

Babies are illogical.
Nobody is despised who can manage a crocodile.
Illogical persons are despised.
Therefore babies cannot manage crocodiles.

http://engineeringbooks.net

Solution

Let $B(x)$ denote 'x is a baby'.
Let $I(x)$ denote 'x is illogical'.
Let $D(x)$ denote 'x is despised'.
Let $C(x)$ denote 'x can manage crocodiles'.

Then the premises are:

(i) $\forall x\ (B(x) \Rightarrow I(x))$
(ii) $\forall x\ (C(x) \Rightarrow \neg D(x))$
(iii) $\forall x\ (I(x) \Rightarrow D(x))$

The conclusion is $\forall x\,(B(x) \Rightarrow \neg C(x))$.

1.	$\forall x\ (B(x) \Rightarrow I(x))$	Premise (i)
2.	$\forall x\ (C(x) \Rightarrow \neg D(x))$	Premise (ii)
3.	$\forall x\ (I(x) \Rightarrow D(x))$	Premise (iii)
4.	$B(x) \Rightarrow I(x)$	1, Universal instantiation
5.	$C(x) \Rightarrow \neg D(x)$	2, Universal instantiation
6.	$I(x) \Rightarrow D(x)$	3, Universal instantiation
7.	$B(x)$	Premise of conclusion
8.	$I(x)$	4,7 Modus pollens
9.	$D(x)$	6,8 Modus pollens
10.	$\neg C(x)$	5,9 Modus tollens
11.	$B(x) \Rightarrow \neg C(x)$	7,10 Conditional proof
12.	$\forall x\ (B(x) \Rightarrow \neg C(x))$	11, Universal generalization.

Hence the conclusion is valid.

EXAMPLE 1.34

Give an indirect proof of

$$(\neg Q,\ P \Rightarrow Q,\ P \vee S) \Rightarrow S$$

Solution

We have to prove S. So we include (iv) $\neg S$ as a premise.

1.	$P \vee S$	Premise (iii)
2.	$\neg S$	Premise (iv)
3.	P	1,2, Disjunctive syllogism
4.	$P \Rightarrow Q$	Premise (ii)
5.	Q	3,4, Modus ponens
6.	$\neg Q$	Premise (i)
7.	$Q \wedge \neg Q$	5,6, Conjuction
8.	**F**	I_8

We get a contradiction. Hence $(\neg Q,\ P \Rightarrow Q,\ P \vee S) \Rightarrow S$.

http://engineeringbooks.net

EXAMPLE 1.35

Test the validity of the following argument:

All integers are irrational numbers.
Some integers are powers of 2.
Therefore, some irrational number is a power of 2.

Solution

Let $Z(x)$ denote 'x is an integer'.
Let $I(x)$ denote 'x is an irrational number'.
Let $P(x)$ denote 'x is a power of 2'.

The premises are:

(i) $\forall x\,(Z(x) \Rightarrow I(x))$
(ii) $\exists x\,(Z(x) \wedge P(x))$

The conclusion is $\exists x\,(I(x) \wedge P(x))$.

1. $\exists x\,(Z(x) \wedge P(x))$	Premise (ii)
2. $Z(b) \wedge P(b)$	1, Existential instantiation
3. $Z(b)$	2, Simplification
4. $P(b)$	2, Simplification
5. $\forall x\,(Z(x) \Rightarrow I(x))$	Premise (i)
6. $Z(b) \Rightarrow I(b)$	5, Universal instantiation
7. $I(b)$	3,6, Modus ponens
8. $I(b) \wedge P(b)$	7,4 Conjunction
9. $\exists x\,(I(x) \wedge P(x))$	8, Existential instantiation.

Hence the argument is valid.

SELF-TEST

Choose the correct answer to Questions 1–5:

1. The following sentence is not a proposition.
(a) George Bush is the President of India.
(b) $\sqrt{-1}$ is a real number.
(c) Mathematics is a difficult subject.
(d) I wish you all the best.

2. The following is a well-formed formula.
(a) $(P \wedge Q) \Rightarrow (P \vee Q)$
(b) $(P \wedge Q) \Rightarrow (P \vee Q) \wedge R)$
(c) $(P \wedge (Q \wedge R)) \Rightarrow (P \wedge Q))$
(d) $\neg (Q \wedge \neg (P \vee \neg Q)$

http://engineeringbooks.net

3. $\dfrac{P \wedge Q}{\therefore P}$ is called:
   (a) Addition
   (b) Conjunction
   (c) Simplification
   (d) Modus tollens
4. Modus ponens is
   (a) $\neg Q$

   $$\frac{P \Rightarrow Q}{\therefore \neg P}$$

   (b) $\neg P$

   $$\frac{P \vee Q}{\therefore Q}$$

   (c) P

   $$\frac{P \Rightarrow Q}{\therefore Q}$$

   (d) none of the above
5. $\neg P \wedge \neg Q \wedge R$ is a minterm of:
   (a) $P \vee Q$
   (b) $\neg P \wedge \neg Q \wedge R$
   (c) $P \wedge Q \wedge R \wedge S$
   (d) $P \wedge R$
6. Find the truth value of $P \Rightarrow Q$ if the truth values of P and Q are F and T respectively.
7. For what truth values of P, Q and R, the truth value of $(P \Rightarrow Q) \Rightarrow R$ is F?
   (P, Q, R have the truth values F, T, F or F, T, T)
8. If P, Q, R have the truth values F, T, F, respectively, find the truth value of $(P \Rightarrow Q) \vee (P \Rightarrow R)$.
9. State universal generalization.
10. State existential instantiation.

EXERCISES

1.1 Which of the following sentences are propositions?
   (a) A triangle has three sides.
   (b) 11111 is a prime number.
   (c) Every dog is an animal.

http://engineeringbooks.net

(d) Ram ran home.
(e) An even number is a prime number.
(f) 10 is a root of the equation $x^2 - 1002x + 10000 = 0$
(g) Go home and take rest.

1.2 Express the following sentence in symbolic form: For any two numbers a and b, only one of the following holds: $a < b$, $a = b$, and $a > b$.

1.3 The truth table of a connective called Exclusive OR (denoted by $\bar{\vee}$) is shown in Table 1.18.

TABLE 1.18 Truth Table for Exclusive OR

P	Q	$P \bar{\vee} Q$
T	T	F
T	F	T
F	T	T
F	F	F

Give an example of a sentence in English (i) in which Exclusive OR is used, (ii) in which OR is used. Show that $\bar{\vee}$ is associative, commutative and distributive over $\wedge$.

1.4 Find two connectives, using which any other connective can be described.

1.5 The connective NAND denoted by $\uparrow$ (also called the Sheffer stroke) is defined as follows: $P \uparrow Q = \neg (P \wedge Q)$. Show that every connective can be expressed in terms of NAND.

1.6 The connective NOR denoted by $\downarrow$ (also called the Peirce arrow) is defined as follows: $P \downarrow Q = \neg (P \vee Q)$. Show that every connective can be expressed in terms of NOR.

1.7 Construct the truth table for the following:
(a) $(P \vee Q) \Rightarrow ((P \vee R) \Rightarrow (R \vee Q))$
(b) $(P \vee (Q \Rightarrow R)) \Leftrightarrow ((P \vee \neg R) \Rightarrow Q)$

1.8 Prove the following equivalences:
(a) $(\neg P \Rightarrow (\neg P \Rightarrow (\neg P \wedge Q))) \equiv P \vee Q$
(b) $P \equiv (P \vee Q) \wedge (P \vee \neg Q)$
(c) $\neg (P \Leftrightarrow Q) \equiv (P \wedge \neg Q) \vee (\neg P \wedge Q)$

1.9 Prove the logical identities given in Table 1.11 using truth tables.

1.10 Show that $P \Rightarrow (Q \Rightarrow (R \Rightarrow (\neg P \Rightarrow (\neg Q \Rightarrow \neg R))))$ is a tautology.

1.11 Is $(P \Rightarrow \neg P) \Rightarrow \neg P$ (i) a tautology, (ii) a contradiction, (iii) neither a tautology nor a contradiction?

http://engineeringbooks.net

1.12 Is the implication $(P \wedge (P \Rightarrow \neg Q)) \vee (Q \Rightarrow \neg Q) \Rightarrow \neg Q$ a tautology?

1.13 Obtain the principal disjunctive normal form of the following:
(a) $P \Rightarrow (P \Rightarrow Q \wedge (\neg (\neg Q \vee \neg P)))$
(b) $(Q \wedge \neg R \wedge \neg S) \vee (R \wedge S)$.

1.14 Simplify the formula whose principal disjunctive normal form is $110 \vee 100 \vee 010 \vee 000$.

1.15 Test the validity of the following arguments:
(a) $P \Rightarrow Q$

$$\frac{R \Rightarrow \neg Q}{\therefore P \Rightarrow \neg R}$$

(b) $R \Rightarrow \neg Q$
$P \Rightarrow Q$

$$\frac{\neg R \Rightarrow S}{\therefore P \Rightarrow S}$$

(c) P
Q
$\neg Q \Rightarrow R$

$$\frac{Q \Rightarrow \neg R}{\therefore R}$$

(d) $P \Rightarrow Q \wedge R$
$Q \vee S \Rightarrow T$

$$\frac{S \vee P}{\therefore T}$$

1.16 Test the validity of the following argument:
If Ram is clever then Prem is well-behaved.
If Joe is good then Sam is bad and Prem is not well-behaved.
If Lal is educated then Joe is good or Ram is clever.
Hence if Lal is educated and Prem is not well-behaved then Sam is bad.

1.17 A company called for applications from candidates, and stipulated the following conditions:
(a) The applicant should be a graduate.
(b) If he knows Java he should know C++.
(c) If he knows Visual Basic he should know Java.
(d) The applicant should know Visual Basic.

Can you simplify the above conditions?

1.18 For what universe of discourse the proposition $\forall x\,(x \geq 5)$ is true?

http://engineeringbooks.net

1.19 By constructing a suitable universe of discourse, show that

$$\exists x\,(P(x) \Rightarrow Q(x)) \Leftrightarrow (\exists x\,P(x) \Rightarrow \exists x\,Q(x))$$

is not valid.

1.20 Show that the following argument is valid:
All men are mortal.
Socrates is a man.
So Socrates is mortal.

1.21 Is the following sentence true? If philosophers are not money-minded and some money-minded persons are not clever, then there are some persons who are neither philosphers nor clever.

1.22 Test the validity of the following argument:
No person except the uneducated are proud of their wealth.
Some persons who are proud of their wealth do not help others.
Therefore, some uneducated persons cannot help others.

http://engineeringbooks.net

2 Mathematical Preliminaries

In this chapter we introduce the concepts of set theory and graph theory. Also, we define strings and discuss the properties of strings and operations on strings. In the final section we deal with the principle of induction, which will be used for proving many theorems throughout the book.

2.1 SETS, RELATIONS AND FUNCTIONS

2.1.1 SETS AND SUBSETS

A set is a well-defined collection of objects, for example, the set of all students in a college. Similarly, the collection of all books in a college library is also a set. The individual objects are called *members* or *elements* of the set.

We use the capital letters $A, B, C, \ldots$ for denoting sets. The small letters $a, b, c, \ldots$ are used to denote the elements of any set. When a is an element of the set A, we write $a \in A$. When a is not an element of A, we write $a \notin A$.

Various Ways of Describing a Set

(i) *By listing its elements.* We write all the elements of the set (without repetition) and enclose them within braces. We can write the elements in any order. For example, the set of all positive integers divisible by 15 and less than 100 can be written as $\{15, 30, 45, 60, 75, 90\}$.

(ii) *By describing the properties of the elements of the set.* For example, the set $\{15, 30, 45, 60, 75, 90\}$ can be described as: $\{n \mid n$ is a positive integer divisible by 15 and less than $100\}$. (The description of the property is called *predicate*. In this case the set is said to be implicitly specified.)

http://engineeringbooks.net

(iii) *By recursion.* We define the elements of the set by a computational rule for calculating the elements. For example, the set of all natural numbers leaving a remainder 1 when divided by 3 can be described as

$$\{a_n \mid a_o = 1,\ a_{n+1} = a_n + 3\}$$

When the computational rule is clear from the context, we simply specify the set by some initial elements. The previous set can be written as {1, 4, 7, 10, ...}. The four elements given suggest that the computational rule is: $a_{n+1} = a_n + 3$.

Subsets and Operations on Sets

A set A is said to be a subset of B (written as $A \subseteq B$) if every element of A is also an element of B.

Two sets A and B are equal (we write $A = B$) if their members are the same. In practice, to prove that $A = B$, we prove $A \subseteq B$ and $B \subseteq A$.

A set with no element is called an empty set, also called a null set or a void set, and is denoted by $\emptyset$.

We define some operations on sets.

$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$, called the union of A and B.
$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$, called the intersection of A and B.
$A - B = \{x \mid x \in A \text{ and } x \notin B\}$, called the complement of B in A.

A^c denotes $U - A$, where U is the universal set, the set of all elements under consideration.

The set of all subsets of a set A is called the *power set* of A. It is denoted by 2^A.

Let A and B be two sets. Then $A \times B$ is defined as $\{(a, b) \mid a \in A \text{ and } b \in B\}$. $((a, b)$ is called an ordered pair and is different from (b, a).)

Definition 2.1 Let S be a set. A collection $(A_1, A_2, \ldots, A_n)$ of subsets of S is called a partition if $A_i \cap A_j = \emptyset\ (i \neq j)$ and $S = \bigcup_{i=1}^{n} A_i$ (i.e. $A_1 \cup A_2 \cup \ldots \cup A_n$).

For example, if $S = \{1, 2, 3, \ldots, 10\}$, then $\{\{1, 3, 5, 7, 9\}, \{2, 4, 6, 8, 10\}\}$ is a partition of S.

2.1.2 Sets with One Binary Operation

A binary operation $*$ on a set S is a rule which assigns, to every ordered pair (a, b) of elements from S, a unique element denoted by $a * b$.

Addition, for example, is a binary operation on the set Z of all integers. (Throughout this book, Z denotes the set of all integers.)

Union is a binary operation on 2^A, where A is any nonempty set. We give below five postulates on binary operations.

Postulate 1: *Closure.* If a and b are in S, then $a * b$ is in S.

http://engineeringbooks.net

Postulate 2: *Associativity.* If a, b, c are in S, then $(a * b) * c = a * (b * c)$.

Postulate 3: *Identity element.* There exists a unique element (called the identity element) e in S such that for any element x in S, $x * e = e * x = x$.

Postulate 4: *Inverse.* For every element x in S there exists a unique element x' in S such that $x * x' = x' * x = e$. The element x' is called the inverse of x w.r.t. $*$.

Postulate 5: *Commutativity.* If $a, b \in S$, then $a * b = b * a$.

It may be noted that a binary operation may satisfy none of the above five postulates. For example, let $S = \{1, 2, 3, 4, \ldots\}$, and let the binary operation be subtraction (i.e. $a * b = a - b$). The closure postulate is not satisfied since $2 - 3 = -1 \notin S$. Also, $(2 - 3) - 4 \neq 2 - (3 - 4)$, and so associativity is not satisfied. As we cannot find a positive integer such that $x - e = e - x = x$, the postulates 3 and 4 are not satisfied. Obviously, $a - b \neq b - a$. Therefore, commutativity is not satisfied.

Our interest lies in sets with a binary operation satisfying the postulates.

Definitions (i) A set S with a binary operation $*$ is called a *semigroup* if the postulates 1 and 2 are satisfied.

(ii) A set S with a binary operation $*$ is called a *monoid* if the postulates 1–3 are satisfied.

(iii) A set S with $*$ is called a *group* if the postulates 1–4 are satisfied.

(iv) A semigroup (monoid or group) is called a *commutative* or an *abelian* semigroup (monoid or group) if the postulate 5 is satisfied.

Figure 2.1 gives the relationship between semigroups, monoids, groups, etc. where the numbers refer to the postulate number.

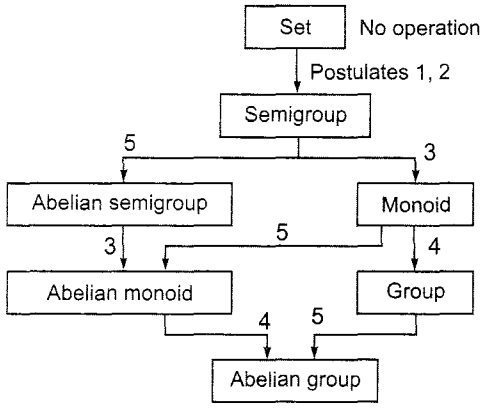


Fig. 2.1 Sets with one binary operation.

We interpret Fig. 2.1 as follows: A monoid satisfying postulate 4 is a group. A group satisfying postulate 5 is an abelian group, etc.

We give below a few examples of sets with one binary operation:

(i) Z with addition is an abelian group.
(ii) Z with multiplication is an abelian monoid. (It is not a group since it does not satisfy the postulate 4.)
(iii) $\{1, 2, 3, \ldots\}$ with addition is a commutative semigroup but not a monoid. (The identity element can be only 0, but 0 is not in the set.)
(iv) The power set 2^A of $A (A \neq \emptyset)$ with union is a commutative monoid. (The identity element is $\emptyset$.)
(v) The set of all 2×2 matrices under multiplication is a monoid but not an abelian monoid.

2.1.3 Sets with Two Binary Operations

Sometimes we come across sets with two binary operations defined on them (for example, in the case of numbers we have addition and multiplication). Let S be a set with two binary operations $*$ and o. We give below 11 postulates in the following way:

(i) Postulates 1–5 refer to $*$ postulates.
(ii) Postulates 6, 7, 8, 10 are simply the postulates 1, 2, 3, 5 for the binary operation o.
(iii) *Postulate* 9: If S under $*$ satisfies the postulates 1–5 then for every x in S, with $x \neq e$, there exists a unique element x' in S such that $x' \text{ o } x = x \text{ o } x' = e'$, where e' is the identity element corresponding to o.
(iv) *Postulate* 11: *Distributivity*. For a, b, c, in S

$$a \text{ o } (b * c) = (a \text{ o } b) * (a \text{ o } c)$$

A set with one or more binary operations is called an algebraic system. For example, groups, monoids, semigroups are algebraic systems with one binary operation.

We now define some algebraic systems with two binary operations.

Definitions (i) A set with two binary operations $*$ and o is called a *ring* if (a) it is an abelian group w.r.t. $*$, and (b) o satisfies the closure, associativity and distributivity postulates (i.e. postulates 6, 7 and 11).

(ii) A ring is called a commutative ring if the commutativity postulate is satisfied for o.
(iii) A commutative ring with unity is a commutative ring that satisfies the identity postulate (i.e. postulate 8) for o.
(iv) A *field* is a set with two binary operations $*$ and o if it satisfies the postulates 1–11.

We now give below a few examples of sets with two binary operations:

(i) Z with addition and multiplication (in place of $*$ and o) is a commutative ring with identity. (The identity element w.r.t. addition is 0, and the identity element w.r.t. multiplication is 1.)

(ii) The set of all rational numbers (i.e. fractions which are of the form a/b, where a is any integer and b is an integer different from zero) is a field. (The identity element w.r.t. multiplication is 1. The inverse of a/b, $a/b \neq 0$ is b/a.)

(iii) The set of all 2×2 matrices with matrix addition and matrix multiplication is a ring with identity, but not a field.

(iv) The power set 2^A ($A \neq \emptyset$) is also a set with two binary operations $\cup$ and $\cap$. The postulates satisfied by $\cup$ and $\cap$ are 1, 2, 3, 5, 6, 7, 8, 10 and 11. The power set 2^A is not a group or a ring or a field. But it is an abelian monoid w.r.t. both the operations $\cup$ and $\cap$.

Figure 2.2 illustrates the relation between the various algebraic systems we have introduced. The interpretation is as given in Fig. 2.1. The numbers refer to postulates. For example, an abelian group satisfying the postulates 6, 7 and 11 is a ring.

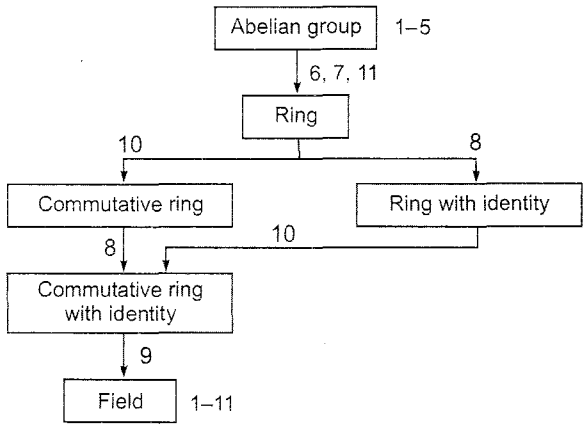


Fig. 2.2 Sets with two binary operations.

2.1.4 RELATIONS

The concept of a relation is a basic concept in computer science as well as in real life. This concept arises when we consider a pair of objects and compare one with the other. For example, 'being the father of' gives a relation between two persons. We can express the relation by ordered pairs (for instance, 'a is the father of b' can be represented by the ordered pair (a, b)).

While executing a program, comparisons are made, and based on the result, different tasks are performed. Thus in computer science the concept of relation arises just as in the case of data structures.

Definition 2.2 A relation R in a set S is a collection of ordered pairs of elements in S (i.e. a subset of $S \times S$). When (x, y) is in R, we write xRy. When (x, y) is not in R, we write $xR'y$.

EXAMPLE 2.1

A relation R in Z can be defined by xRy if $x > y$.

Properties of Relations

(i) A relation R in S is *reflexive* if xRx for every x in S.
(ii) A relation R in S is *symmetric* if for x, y in S, yRx whenever xRy.
(iii) A relation R in S is *transitive* if for x, y and z in S, xRz whenever xRy and yRz.

We note that the relation given in Example 2.1 is neither reflexive nor symmetric, but transitive.

EXAMPLE 2.2

A relation R in $\{1, 2, 3, 4, 5, 6\}$ is given by

$$\{(1, 2), (2, 3), (3, 4), (4, 4), (4, 5)\}$$

This relation is not reflexive as $1R'1$. It is not symmetric as $2R3$ but $3R'2$. It is also not transitive as $1R2$ and $2R3$ but $1R'3$.

EXAMPLE 2.3

Let us define a relation R in $\{1, 2, \ldots, 10\}$ by aRb if a divides b. R is reflexive and transitive but not symmetric ($3R6$ but $6R'3$).

EXAMPLE 2.4

If i, j, n are integers we say that i is congruent to j modulo n (written as $i \equiv j$ modulo n or $i \equiv j \bmod n$) if $i - j$ is divisible by n. The 'congruence modulo n' is a relation which is reflexive and symmetric (if $i - j$ is divisible by n, so is $j - i$). If $i \equiv j \bmod n$ and $j \equiv k \bmod n$, then we have $i - j = an$ for some a and $j - k = bn$ for some b. So,

$$i - k = i - j + j - k = an + bn$$

which means that $i \equiv k \bmod n$. Thus this relation is also transitive.

Definition 2.3 A relation R in a set S is called an equivalence relation if it is reflexive, symmetric and transitive.

Example 2.5 gives an equivalence relation in Z.

EXAMPLE 2.5

We can define an equivalence relation R on any set S by defining aRb if $a = b$. (Obviously, $a = a$ for every a. So, R is reflexive. If $a = b$ then $b = a$. So R is symmetric. Also, if $a = b$ and $b = c$, then $a = c$. So R is transitive.)

EXAMPLE 2.6

Define a relation R on the set of all persons in New Delhi by aRb if the persons a and b have the same date of birth. Then R is an equivalence relation.

Let us study this example more carefully. Corresponding to any day of the year (say, 4th February), we can associate the set of all persons born on that day. In this way the set of all persons in New Delhi can be partitioned into 366 subsets. In each of the 366 subsets, any two elements are related. This leads to one more property of equivalence relations.

Definition 2.4 Let R be an equivalence relation on a set S. Let $a \in S$. Then C_a is defined as

$$\{b \in S \mid aRb\}$$

The C_a is called an equivalence class containing a. In general, the C_a's are called equivalence classes.

EXAMPLE 2.7

For the congruence modulo 3 relation on $\{1, 2, \ldots, 7\}$,

$$C_2 = \{2, 5\}, \qquad C_1 = \{1, 4, 7\}, \qquad C_3 = \{3, 6\}$$

For the equivalence relation 'having the same birth day' (discussed in Example 2.6), the set of persons born on 4th February is an equivalence class, and the number of equivalence classes is 366. Also, we may note that the union of all the 366 equivalence classes is the set of all persons in Delhi. This is true for any equivalence relation because of the following theorem.

Theorem 2.1 Any equivalence relation R on a set S partitions S into disjoint equivalence classes.

Proof Let $\bigcup_{a \in S} C_a$ denote the union of distinct equivalence classes. We have to prove that:

(i) $S = \bigcup_{a \in S} C_a$,

(ii) $C_a \cap C_b = \emptyset$ if C_a and C_b are different, i.e. $C_a \neq C_b$.

Let $s \in S$. Then $s \in C_s$ (since sRs, R being reflexive). But $C_s \subseteq \bigcup_{a \in S} C_a$. So $S \subseteq \bigcup_{a \in S} C_a$. By definition of C_a, $C_a \subseteq S$ for every a in S. So $\bigcup_{a \in S} C_a \subseteq S$. Thus we have proved (i).

Before proving (ii), we may note the following:

$$C_a = C_b \qquad \text{if } aRb \tag{2.1}$$

As aRb, we have bRa because R is symmetric. Let $d \in C_a$. By definition of C_a, we have aRd. As bRa and aRd, by transitivity of R, we get bRd. This means $d \in C_b$. Thus we have proved $C_a \subseteq C_b$. In a similar way we can show that $C_b \subseteq C_a$. Therefore, (2.1) is proved.

Now we prove (ii) by the method of contradiction (refer to Section 2.5). We want to prove that $C_a \cap C_b = \emptyset$ if $C_a \neq C_b$. Suppose $C_a \cap C_b \neq \emptyset$. Then there exists some element d in S such that $d \in C_a$ and $d \in C_b$. As $d \in C_a$, we have aRd. Similarly, we have bRd. By symmetry of R, dRb. As aRd and dRb, by transitivity of R, we have aRb. Now we can use (2.1) to conclude that $C_a = C_b$. But this is a contradiction (as $C_a \neq C_b$). Therefore, $C_a \cap C_b = \emptyset$. Thus (ii) is proved. ∎

If we apply Theorem 2.1 to the equivalence relation congruence modulo 3 on $\{1, 2, 3, 4, 5, 6, 7\}$, we get

$$C_1 = C_4 = C_7 = \{1, 4, 7\}$$
$$C_2 = C_5 = \{2, 5\}$$
$$C_3 = C_6 = \{3, 6\}$$

and therefore,

$$\{1, 2, \ldots, 7\} = C_1 \cup C_2 \cup C_3$$

EXERCISE Let S denote the set of all students in a particular college. Define aRb if a and b study in the same class. What are the equivalence classes? In what way does R partition S?

2.1.5 Closure of Relations

A given relation R may not be reflexive or transitive. By adding more ordered pairs to R we can make it reflexive or transitive. For example, consider a relation $R = \{(1, 2), (2, 3), (1, 1), (2, 2)\}$ in $\{1, 2, 3\}$. R is not reflexive as $3R'3$. But by adding $(3, 3)$ to R, we get a reflexive relation. Also, R is not transitive as $1R2$ and $2R3$ but $1R'3$. By adding the pair $(1, 3)$, we get a relation $T = \{(1, 2), (2, 3), (1, 1), (2, 2), (1, 3)\}$ which is transitive. There are many transitive relations T containing R. But the smallest among them is interesting.

Definition 2.5 Let R be a relation in a set S. Then the transitive closure of R (denoted by R^+) is the smallest transitive relation containing R.

Note: We can define reflexive closure and symmetric closure in a similar way.

Definition 2.6 Let R be a relation in S. Then the reflexive-transitive closure of R (denoted by R^*) is the smallest reflexive and transitive relation containing R.

For constructing R^+ and R^*, we define the composite of two relations. Let R_1 and R_2 be the two relations in S. Then,

(i) $R_1 \circ R_2 = \{(a, c) \in S \times S \mid aR_1b \text{ and } bR_2c \text{ for some } b \in S\}$

(ii) $R_1^2 = R_1 \circ R_1$

(iii) $R_1^n = R_1^{n-1} \circ R_1$ for all $n \geq 2$

Note: For getting the elements of R_1 o R_2, we combine (a, b) in R_1 and (b, c) in R_2 to get (a, c) in R_1 o R_2.

Theorem 2.2 Let S be a finite set and R be a relation in S. Then the transitive closure R^+ of R exists and $R^+ = R \cup R^2 \cup R^3 \ldots$.

EXAMPLE 2.8

Let $R = \{(1, 2), (2, 3), (2, 4)\}$ be a relation in $\{1, 2, 3, 4\}$. Find R^+.

Solution

$$R = \{(1, 2), (2, 3), (2, 4)\}$$

$$R^2 = \{(1, 2), (2, 3), (2, 4)\} \text{ o } \{(1, 2), (2, 3), (2, 4)\}$$

$$= \{(1, 3), (1, 4)\}$$

(We combine (a, b) and (b, c) in R to get (a, c) in R^2.)

$$R^3 = R^2 \text{ o } R = \{(1, 3), (1, 4)\} \text{ o } \{(1, 2), (2, 3), (2, 4)\} = \emptyset$$

(Here no pair (a, b) in R^2 can be combined with any pair in R.)

$$R^4 = R^5 = \cdots = \emptyset$$

$$R^+ = R \cup R^2 = \{(1, 2), (2, 3), (2, 4), (1, 3), (1, 4)\}$$

EXAMPLE 2.9

Let $R = \{(a, b), (b, c), (c, a)\}$. Find R^+.

Solution

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

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

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

(This is obtained by combining the pairs: (a, b) and (b, c), (b, c) and (c, a), and (c, a) and (a, b).)

$$R^3 = R^2 \text{ o } R = \{(a, c), (b, a), (c, b)\} \text{ o } \{(a, b), (b, c), (c, a)\}$$

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

$$R^4 = R^3 \text{ o } R = \{(a, a), (b, b), (c, c)\} \text{ o } \{(a, b), (b, c), (c, a)\}$$

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

So,

$$R^5 = R^4 \text{ o } R = R \text{ o } R = R^2, \qquad R^6 = R^5 \text{ o } R = R^2 \text{ o } R = R^3$$

$$R^7 = R^6 \text{ o } R = R^3 \text{ o } R = R^4 = R$$

Then any R^n is one of R, R^2 or R^3. Hence,

$$R^+ = R \cup R^2 \cup R^3$$

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

Note: $R^* = R^+ \cup \{(a, a) \mid a \in S\}$.

EXAMPLE 2.10

If $R = \{(a, b), (b, c), (c, a)\}$ is a relation in $\{a, b, c\}$, find R^*.

Solution

From Example 2.9,

$$R^* = R^+ \cup \{(a, a), (b, b), (c, c)\}$$
$$= \{(a, b), (b, c), (c, a), (a, c), (b, a), (c, b), (a, a), (b, b), (c, c)\}$$

EXAMPLE 2.11

What is the symmetric closure of relation R in a set S?

Solution

Symmetric closure of $R = R \cup \{(b, a) \mid aRb\}$.

2.1.6 FUNCTIONS

The concept of a function arises when we want to associate a unique value (or result) with a given argument (or input).

Definition 2.7 A function or map f from a set X to a set Y is a rule which associates to every element x in X a unique element in Y, which is denoted by $f(x)$. The element $f(x)$ is called the image of x under f. The function is denoted by $f: X \to Y$.

Functions can be defined either (i) by giving the images of all elements of X, or (ii) by a computational rule which computes $f(x)$ once x is given.

EXAMPLES (a) $f: \{1, 2, 3, 4\} \to \{a, b, c\}$ can be defined by $f(1) = a$, $f(2) = c$, $f(3) = a$, $f(4) = b$.

(b) $f: R \to R$ can be defined by $f(x) = x^2 + 2x + 1$ for every x in R. (R denotes the set of all real numbers.)

Definition 2.8 $f : X \to Y$ is said to be one-to-one (or injective) if different elements in X have different images, i.e. $f(x_1) \neq f(x_2)$ when $x_1 \neq x_2$.

Note: To prove that f is one-to-one, we prove the following: Assume $f(x_1) = f(x_2)$ and show that $x_1 = x_2$.

Definition 2.9 $f : X \to Y$ is onto (surjective) if every element y in Y is the image of some element x in X.

Definition 2.10 $f : X \to Y$ is said to be a one-to-one correspondence or bijection if f is both one-to-one and onto.

EXAMPLE 2.12

$f : Z \to Z$ given by $f(n) = 2n$ is one-to-one but not onto.

Solution

Suppose $f(n_1) = f(n_2)$. Then $2n_1 = 2n_2$. So $n_1 = n_2$. Hence f is one-to-one. It is not onto since no odd integer can be the image of any element in Z (as any image is even).

The following theorem distinguishes a finite set from an infinite set.

Theorem 2.3 Let S be a finite set. Then $f : S \to S$ is one-to-one iff it is onto.

Note: The above result is not true for infinite sets as Example 2.12 gives a one-to-one function $f : Z \to Z$ which is not onto.

EXAMPLE 2.13

Show that $f : R \to R - \{1\}$ given by $f(x) = (x + 1)/(x - 1)$ is onto.

Solution

Let $y \in R$. Suppose $y = f(x) = (x + 1)/(x - 1)$. Then $y(x - 1) = x + 1$, i.e. $yx - x = 1 + y$. So, $x = (1 + y)/(y - 1)$. As $(1 + y)/(y - 1) \in R$ for all $y \neq 1$, y is the image of $(1 + y)/(y - 1)$ in $R - \{1\}$. Thus, f is onto.

The Pigeonhole Principle†

Suppose a postman distributes 51 letters in 50 mailboxes (pigeonholes). Then it is evident that some mailbox will contain at least two letters. This is enunciated as a mathematical principle called the pigeonhole principle.

If n objects are distributed over m places and $n > m$, then some place receives at least two objects.

EXAMPLE 2.14

If we select 11 natural numbers between 1 to 380, show that there exist at least two among these 11 numbers whose difference is at most 38.

Solution

Arrange the numbers 1, 2, 3, . . ., 380 in 10 boxes, the first box containing 1, 2, 3, . . ., 38, the second containing 39, 40, . . ., 76, etc. There are 11 numbers to be selected. Take these numbers from the boxes. By the pigeonhole principle, at least one box will contain two of these eleven numbers. These two numbers differ by 38 or less.

† The pigeonhole principle is also called the Dirichlet drawer principle, named after the French mathematician G. Lejeune Dirichlet (1805–1859).

2.2 GRAPHS AND TREES

The theory of graphs is widely applied in many areas of computer science—formal languages, compiler writing, artificial intelligence (AI), to mention only a few. Also, the problems in computer science can be phrased as problems in graphs. Our interest lies mainly in trees (special types of graphs) and their properties.

2.2.1 GRAPHS

Definition 2.11 A graph (or undirected graph) consists of (i) a nonempty set V called the set of vertices, (ii) a set E called the set of edges, and (iii) a map Φ which assigns to every edge a unique unordered pair of vertices.

Representation of a Graph

Usually a graph, namely the undirected graph, is represented by a diagram where the vertices are represented by points or small circles, and the edges by arcs joining the vertices of the associated pair (given by the map Φ).

Figure 2.3, for example, gives an undirected graph. Thus, the unordered pair $\{v_1, v_2\}$ is associated with the edge e_1; the pair (v_2, v_2) is associated with e_6. (e_6 is a self-loop. In general, an edge is called a self-loop if the vertices in its associated pair coincide.)

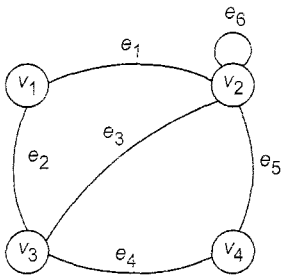


Fig. 2.3 An undirected graph.

Definition 2.12 A directed graph (or digraph) consists of (i) a nonempty set V called the set of vertices, (ii) a set E called the set of edges, and (iii) a map Φ which assigns to every edge a unique ordered pair of vertices.

Representation of a Digraph

The representation is as in the case of undirected graphs except that the edges are represented by directed arcs.

Figure 2.4, for example, gives a directed graph. The ordered pairs (v_2, v_3), (v_3, v_4), (v_1, v_3) are associated with the edges e_3, e_4, e_2, respectively.

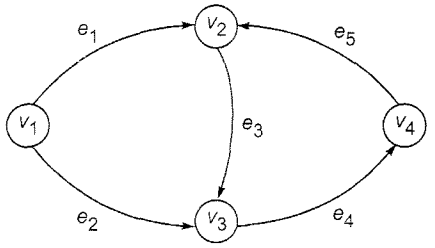


Fig. 2.4 A directed graph.

Definitions (i) If (v_i, v_j) is associated with an edge e, then v_i and v_j are called the end vertices of e; v_i is called a predecessor of v_j which is a successor of v_i.

In Fig. 2.3, v_2 and v_3 are the end vertices of e_3. In Fig. 2.4, v_2 is a predecessor of v_3 which is a successor of v_2. Also, v_4 is a predecessor of v_2 and successor of v_3.

(ii) If G is a digraph, the undirected graph corresponding to G is the undirected graph obtained by considering the edges and vertices of G, but ignoring the 'direction' of the edges. For example, the undirected graph corresponding to the digraph given in Fig. 2.4 is shown in Fig. 2.5.

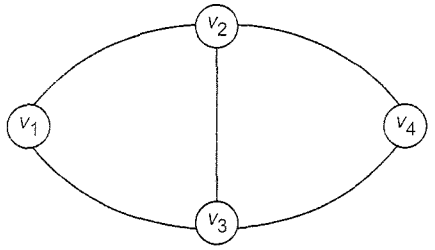


Fig. 2.5 A graph.

Definition 2.13 The degree of a vertex in a graph (directed or undirected) is the number of edges with v as an end vertex. (A self-loop is counted twice while calculating the degree.) In Fig. 2.3, $\deg(v_1) = 2$, $\deg(v_3) = 3$, $\deg(v_2) = 5$. In Fig. 2.4, $\deg(v_2) = 3$, $\deg(v_4) = 2$.

We now mention the following theorem without proof.

Theorem 2.4 The number of vertices of odd degree in any graph (directed or undirected) is even.

Definition 2.14 A path in a graph (undirected or directed) is an alternating sequence of vertices and edges of the form $v_1e_1v_2e_2 \ldots v_{n-1}e_{n-1}v_n$, beginning and ending with vertices such that e_i has v_i and v_{i+1} as its end vertices and no edge or vertex is repeated in the sequence. The path is said to be a path from v_1 to v_n.

For example, $v_1e_2v_3e_3v_2$ is a path in Fig. 2.3. It is a path from v_1 to v_2. In Fig. 2.4, $v_1e_2v_3e_3v_2$ is a path from v_1 to v_2. $v_1e_1v_2$ is also a path from v_1 to v_2.

And $v_3e_4v_4e_5v_2$ is a path from v_3 to v_2. We call $v_3e_4v_4e_5v_2$ a directed path since the edges e_4 and e_5 have the forward direction. (But $v_1e_2v_3e_3v_2$ is not a directed path as e_2 is in the forward direction and e_3 is in the backward direction.)

Definition 2.15 A graph (directed or undirected) is connected if there is a path between every pair of vertices.

The graphs given by Figs. 2.3 and 2.4, for example, are connected.

Definition 2.16 A circuit in a graph is an alternating sequence $v_1e_1v_2e_2 \ldots e_{n-1}v_1$ of vertices and edges starting and ending in the same vertex such that e_i has v_i and v_{i+1} as the end vertices and no edge or vertex other than v_1 is repeated.

In Fig. 2.3, for example, $v_3e_3v_2e_5v_4e_4v_3$, $v_1e_2v_3e_4v_4e_5v_2e_1v_1$ are circuits. In Fig. 2.4, $v_1e_2v_3e_3v_2e_1v_1$ and $v_2e_3v_3e_4v_4e_5v_2$ are circuits.

2.2.2 TREES

Definition 2.17 A graph (directed or undirected) is called a tree if it is connected and has no circuits.

The graphs given in Figs. 2.6 and 2.7, for example, are trees. The graphs given in Figs. 2.3 and 2.4 are not trees.

Note: A directed graph G is a tree iff the corresponding undirected graph is a tree.

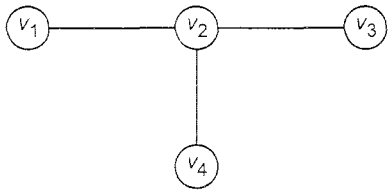


Fig. 2.6 A tree with four vertices.

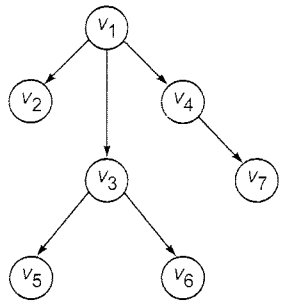


Fig. 2.7 A tree with seven vertices.

We now discuss some properties of trees (both directed and undirected) used in developing transition systems and studying grammar rules.

Property 1 A tree is a connected graph with no circuits or loops.

Property 2 In a tree there is one and only one path between every pair of vertices.

Property 3 If in a graph there is a unique (i.e. one and only one) path between every pair of vertices, then the graph is a tree.

Property 4 A tree with n vertices has $n - 1$ edges.

Property 5 If a connected graph with n vertices has $n - 1$ edges, then it is a tree.

Property 6 If a graph with no circuits has n vertices and $n - 1$ edges, then it is a tree.

A leaf in a tree can be defined as a vertex of degree one. The vertices other than leaves are called internal vertices.

In Fig. 2.6, for example, v_1, v_3, v_4 are leaves and v_2 is an internal vertex. In Fig. 2.7, v_2, v_5, v_6, v_7 are leaves and v_1, v_3, v_4 are internal vertices.

The following definition of ordered trees will be used for representing derivations in context-free grammars.

Definition 2.18 An ordered directed tree is a digraph satisfying the following conditions:

T_1: There is one vertex called the root of the tree which is distinguished from all the other vertices and the root has no predecessors.

T_2: There is a directed path from the root to every other vertex.

T_3: Every vertex except the root has exactly one predecessor.

T_4: The successors of each vertex are ordered 'from the left'.

Note: The condition T_4 of the definition becomes evident once we have the diagram of the graph.

Figure 2.7 is an ordered tree with v_1 as the root. Figure 2.8 also gives an ordered directed tree with v_1 as the root. In this figure the successors of v_1 are ordered as v_2v_3. The successors of v_3 are ordered as v_5v_6.

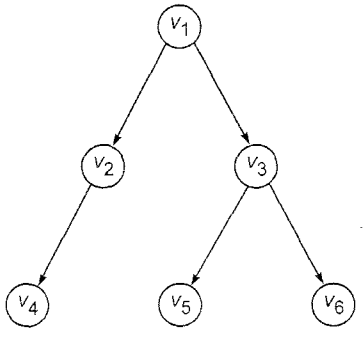


Fig. 2.8 An ordered directed tree.

By adopting the following convention, we can simplify Fig. 2.8. The root is at the top. The directed edges are represented by arrows pointing downwards. As all the arrows point downwards, the directed edges can be simply represented by lines sloping downwards, as illustrated in Fig. 2.9.

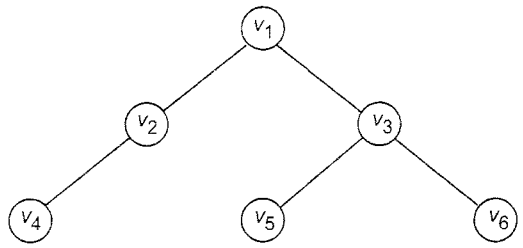


Fig. 2.9 Representation of an ordered directed tree.

Note: An ordered directed tree is connected (which follows from T_2). It has no circuits (because of T_3). Hence an ordered directed tree is a tree (see Definition 2.17).

As we use only the ordered directed trees in applications to grammars, we refer to ordered directed trees as simply trees.

Definition 2.19 A binary tree is a tree in which the degree of the root is 2 and the remaining vertices are of degree 1 or 3.

Note: In a binary tree any vertex has at most two successors. For example, the trees given by Figs. 2.11 and 2.12 are binary trees. The tree given by Fig. 2.9 is not a binary tree.

Theorem 2.5 The number of vertices in a binary tree is odd.

Proof Let n be the number of vertices. The root is of degree 2 and the remaining $n-1$ vertices are of odd degree (by Definition 2.19). By Theorem 2.4, $n-1$ is even and hence n is odd. ∎

We now introduce some more terminology regarding trees:

(i) A son of a vertex v is a successor of v.
(ii) The father of v is the predecessor of v.
(iii) If there is a directed path from v_1 to v_2, v_1 is called an ancestor of v_2, and v_2 is called a descendant of v_1. (*Convention:* v_1 is an ancestor of itself and also a descendant of itself.)
(iv) The number of edges in a path is called the length of the path.
(v) The height of a tree is the length of a longest path from the root. For example, for the tree given by Fig. 2.9, the height is 2. (Actually there are three longest paths, $v_1 \to v_2 \to v_4$, $v_1 \to v_3 \to v_5$, $v_1 \to v_2 \to v_6$. Each is of length 2.)
(vi) A vertex v in a tree is at level k if there is a path of length k from the root to the vertex v (the maximum possible level in a tree is the height of the tree).

Figure 2.10, for example, gives a tree where the levels of vertices are indicated.

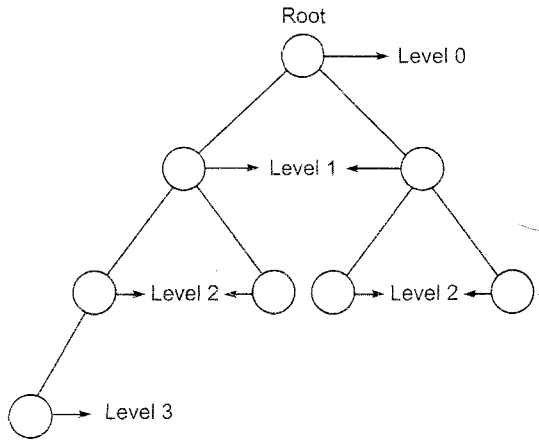


Fig. 2.10 Illustration of levels of vertices.

EXAMPLE 2.15

For a binary tree T with n vertices, show that the minimum possible height is $\lceil \log_2(n + 1) - 1 \rceil$, where $\lceil k \rceil$ is the smallest integer $\geq k$, and the maximum possible height is $(n - 1)/2$.

Solution

In a binary tree the root is at level 0. As every vertex can have at most two successors, we have at most two vertices at level 1, at most 4 vertices at level 2, etc. So the maximum number of vertices in a binary tree of height k is $1 + 2 + 2^2 + \cdots + 2^k$. As T has n vertices, $1 + 2 + 2^2 + \cdots + 2^k \geq n$, i.e. $(2^{k+1} - 1)/(2 - 1) \geq n$, so $k \geq \log_2(n + 1) - 1$. As k is an integer, the smallest possible value for k is $\lceil \log_2(n + 1) - 1 \rceil$. Thus the minimum possible height is $\lceil \log_2(n + 1) - 1 \rceil$.

To get the maximum possible height, we proceed in a similar way. In a binary tree we have the root at zero level and at least two vertices at level 1, 2, ... When T is of height k, we have at least $1 + 2 + \cdots + 2$ (2 repeated k times) vertices. So, $1 + 2k \leq n$, i.e. $k \leq (n - 1)/2$. But, n is odd by Theorem 2.4. So $(n - 1)/2$ is an integer. Hence the maximum possible value for k is $(n - 1)/2$.

EXAMPLE 2.16

When $n = 9$, the trees with minimum and maximum height are shown in Figs. 2.11 and 2.12 respectively. The height of the tree in Fig. 2.11 is $\lceil \log_2(9 + 1) - 1 \rceil = 3$. For the tree in Fig. 2.12, the height $= (9 - 1)/2 = 4$.

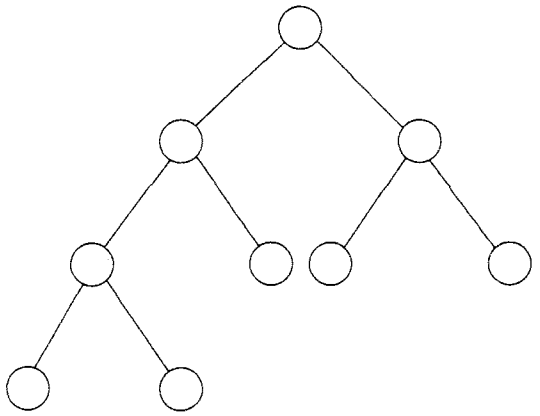


Fig. 2.11 Binary tree of minimum height with 9 vertices.

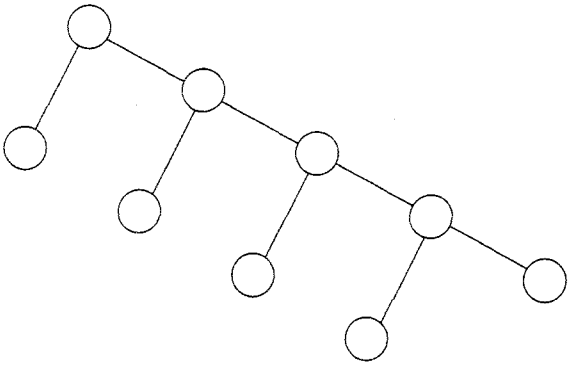


Fig. 2.12 Binary tree of maximum height with 9 vertices.

EXAMPLE 2.17

Prove that the number of leaves in a binary tree T is $(n + 1)/2$, where n is the number of vertices.

Solution

Let m be the number of leaves in a tree with n vertices. The root is of degree 2 and the remaining $n - m - 1$ vertices are of degree 3. As T has n vertices, it has $n - 1$ edges (by Property 4). As each edge is counted twice while calculating the degrees of its end vertices. $2(n - 1)$ = the sum of degrees of all vertices = $2 + m + 3(n - m - 1)$. Solving for m, we get $m = (n + 1)/2$.

EXAMPLE 2.18

For the tree shown in Fig. 2.13, answer the following questions:

(a) Which vertices are leaves and which internal vertices?

(b) Which vertices are the sons of 5?
(c) Which vertex is the father of 5?
(d) What is the length of the path from 1 to 9?
(e) What is the left-right order of leaves?
(f) What is the height of the tree?

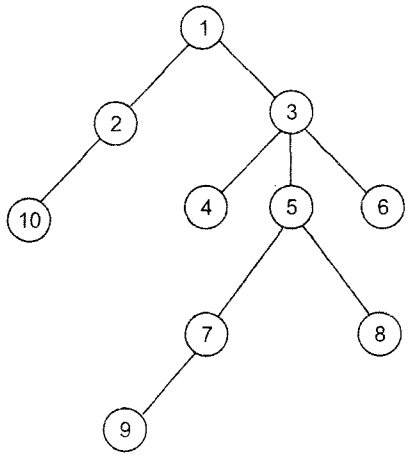


Fig. 2.13 The directed tree for Example 2.18.

Solutions

(a) 10, 4, 9, 8, 6 are leaves. 1, 2, 3, 5, 7 are internal vertices.
(b) 7 and 8 are the sons of 5.
(c) 3 is the father of 5.
(d) Four (the path is $1 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 9$).
(e) 10 – 4 – 9 – 8 – 6.
(f) Four ($1 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 9$ is the longest path).

2.3 STRINGS AND THEIR PROPERTIES

A string over an alphabet set Σ is a finite sequence of symbols from Σ.

NOTATION: Σ^* denotes the set of all strings (including Λ, the empty string) over the alphabet set Σ. That is, $\Sigma^+ = \Sigma^* - \{\Lambda\}$.

2.3.1 OPERATIONS ON STRINGS

The basic operation for strings is the binary concatenation operation. We define this operation as follows: Let x and y be two strings in Σ^*. Let us form a new string z by placing y after x, i.e. $z = xy$. The string z is said to be obtained by concatenation of x and y.

EXAMPLE 2.19

Find xy and yx, where

(a) $x = 010, \quad y = 1$
(b) $x = a\Lambda, \quad y = \text{ALGOL}$

Solution

(a) $xy = 0101$, $yx = 1010$.
(b) $xy = a \wedge \text{ALGOL}$
$yx = \text{ALGOL}\ a\Lambda$.

We give below some basic properties of concatenation.

Property 1 Concatenation on a set Σ^* is associative since for each x, y, z in Σ^*, $x(yz) = (xy)z$.

Property 2 *Identity element.* The set Σ^* has an identity element Λ w.r.t. the binary operation of concatenation as

$$x\Lambda = \Lambda x = x \qquad \text{for every } x \text{ in } \Sigma^*$$

Property 3 Σ^* has left and right cancellations. For x, y, z in Σ^*,

$zx = zy$ implies $x = y$ (left cancellation)

$xz = yz$ implies $x = y$ (right cancellation)

Property 4 For x, y in Σ^*, we have

$$|xy| = |x| + |y|$$

where $|x|, |y|, |xy|$ denote the lengths of the strings x, y, xy, respectively.

We introduce below some more operations on strings.

Transpose Operation

We extend the concatenation operation to define the transpose operation as follows:

For any x in Σ^* and a in Σ,

$$(xa)^T = a(x)^T$$

For example, $(aaabab)^T$ is $babaaa$.

Palindrome. A palindrome is a string which is the same whether written forward or backward, e.g. Malayalam. A palindrome of even length can be obtained by concatenation of a string and its transpose.

Prefix and suffix of a string. A prefix of a string is a substring of leading symbols of that string. For example, w is a prefix of y if there exists y' in Σ^* such that $y = wy'$. Then we write $w < y$. For example, the string 123 has four prefixes, i.e. Λ, 1, 12, 123.

Similarly, a suffix of a string is a substring of trailing symbols of that string, i.e. w is a suffix of y if there exists $y' \in \Sigma^*$ such that $y = y'w$. For example, the string 123 has four suffixes, i.e. Λ, 3, 23, 123.

Theorem 2.6 (Levi's theorem) Let v, w, x and $y \in \Sigma^*$ and $vw = xy$. Then:

(i) there exists a unique string z in Σ^* such that $v = xz$ and $y = zw$ if $|v| > |x|$;

(ii) $v = x$, $y = w$, i.e. $z = \Lambda$ if $|v| = |x|$;

(iii) there exists a unique string z in Σ^* such that $x = vz$, and $w = zy$ if $|v| < |x|$.

Proof We shall give a very simple proof by representing the strings by a diagram (see Fig. 2.14). ∎

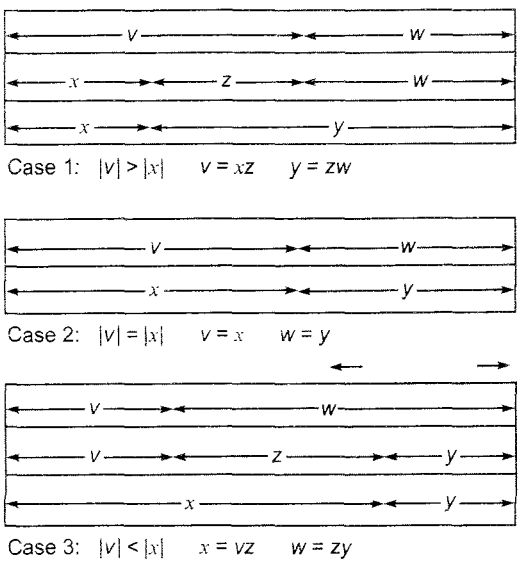


Fig. 2.14 Illustration of Levi's theorem.

2.3.2 Terminal and Nonterminal Symbols

The definitions in this section will be used in subsequent chapters.

A terminal symbol is a unique indivisible object used in the generation of strings.

A nonterminal symbol is a unique object but divisible, used in the generation of strings. A nonterminal symbol will be constructed from the terminal symbols; the number of terminal symbols in a nonterminal symbol may vary; it is also called a variable. In a natural language, e.g. English, the letters a, b, A, B, etc. are terminals and the words boy, cat, dog, go are nonterminal symbols. In programming languages, A, B, C, . . ., Z, :, =, begin, and, if, then, etc. are terminal symbols.

The following will be a variable in Pascal:

< For statement > → for < control variable > : =
< for list > do < statement >

2.4 PRINCIPLE OF INDUCTION

The process of reasoning from general observations to specific truths is called *induction*.

The following properties apply to the set N of natural numbers and the principle of induction.

Property 1 Zero is a natural number.

Property 2 The successor of any natural number is also a natural number.

Property 3 Zero is not the successor of any natural number.

Property 4 No two natural numbers have the same successor.

Property 5 Let a property $P(n)$ be defined for every natural number n. If (i) $P(0)$ is true, and (ii) P(successor of n) is true whenever $P(n)$ is true, then $P(n)$ is true for all n.

A proof by complete enumeration of all possible combinations is called *perfect induction*, e.g. *proof by truth table*.

The method of proof by induction can be used to prove a property $P(n)$ for all n.

2.4.1 METHOD OF PROOF BY INDUCTION

This method consists of three basic steps:

Step 1 Prove $P(n)$ for $n = 0/1$. This is called the *proof for the basis*.

Step 2 Assume the result/properties for $P(n)$. This is called the *induction hypothesis*.

Step 3 Prove $P(n + 1)$ using the induction hypothesis.

EXAMPLE 2.20

Prove that $1 + 3 + 5 + \cdots + r = n^2$, for all $n > 0$, where r is an odd integer and n is the number of terms in the sum. (*Note:* $r = 2n - 1$.)

Solution

(a) *Proof for the basis.* For $n = 1$, L.H.S. $= 1$ and R.H.S. $= 1^2 = 1$. Hence the result is true for $n = 1$.

(b) By induction hypothesis, we have $1 + 3 + 5 + \cdots + r = n^2$. As $r = 2n - 1$,

$$\text{L.H.S.} = 1 + 3 + 5 + \cdots + (2n - 1) = n^2$$

(c) We have to prove that $1 + 3 + 5 + \cdots + r + r + 2 = (n + 1)^2$:

$$\text{L.H.S.} = (1 + 3 + 5 + \cdots + r + (r + 2))$$

$$= n^2 + r + 2 = n^2 + 2n - 1 + 2 = (n + 1)^2 = \text{R.H.S.}$$

EXAMPLE 2.21

Prove the following theorem by induction:

$$1 + 2 + 3 + \cdots + n = n(n + 1)/2$$

Solution

(a) *Proof for the basis.* For $n = 1$, L.H.S. = 1 and R.H.S. = $1(1 + 1)/2 = 1$

(b) Assume $1 + 2 + 3 + \cdots + n = n(n + 1)/2$.

(c) We have to prove:

$$1 + 2 + 3 + \cdots + (n + 1) = (n + 1)(n + 2)/2$$

$$1 + 2 + 3 + \cdots + n + (n + 1)$$

$$= n(n + 1)/2 + (n + 1) \quad \text{(by induction hypothesis)}$$

$$= (n + 1)(n + 2)/2 \quad \text{(on simplification)}$$

The proof by induction can be modified as explained in the following section.

2.4.2 Modified Method of Induction

Three steps are involved in the modified proof by induction.

Step 1 Proof for the basis ($n = 0/1$).

Step 2 Assume the result/properties for all positive integers $< n + 1$.

Step 3 Prove the result/properties using the induction hypothesis (i.e. step 2), for $n + 1$.

Example 2.22 below illustrates the modified method of induction. The method we shall apply will be clear once we mention the induction hypothesis.

EXAMPLE 2.22

Prove the following theorem by induction: A tree with n vertices has $(n - 1)$ edges.

Solution

For $n = 1, 2$, the following trees can be drawn (see Fig. 2.15). So the theorem is true for $n = 1, 2$. Thus, there is basis for induction.

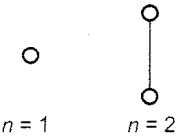


Fig. 2.15 Trees with one or two vertices.

Consider a tree T with $(n + 1)$ vertices as shown in Fig. 2.16. Let e be an edge connecting the vertices v_i and v_j. There is a unique path between v_i and v_j through the edge e. (Property of a tree: There is a unique path between every pair of vertices in a tree.) Thus, the deletion of e from the graph will divide the graph into two subtrees. Let n_1 and n_2 be the number of vertices in the subtrees. As $n_1 \leq n$ and $n_2 \leq n$, by induction hypothesis, the total number of edges in the subtrees is $n_1 - 1 + n_2 - 1$, i.e. $n - 2$. So, the number of edges in T is $n - 2 + 1 = n - 1$ (by including the deleted edge e). By induction, the result is true for all trees.

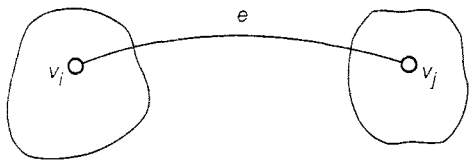


Fig. 2.16 Tree T with $(n + 1)$ vertices.

EXAMPLE 2.23

Two definitions of palindromes are given below. Prove by induction that the two definitions are equivalent.

Definition 1 A palindrome is a string that reads the same forward and backward.

Definition 2 (i) Λ is a palindrome.
(ii) If a is any symbol, the string a is a palindrome.
(iii) If a is any symbol and x is a palindrome, then axa is a palindrome.
(iv) Nothing is a palindrome unless it follows from (i)–(iii).

Solution

Let x be a string which satisfies the Definition 1, i.e. x reads the same forward and backward. By induction on the length of x we prove that x satisfies the Definition 2.

If $|x| \leq 1$, then $x = a$ or Λ. Since x is a palindrome by Definition 1, Λ and a are also palindromes (hence (i) and (ii)), i.e. there is basis for induction. If $|x| > 1$, then $x = awa$, where w, by Definition 1, is a palindrome; hence the rule (iii). Thus, if x satisfies the Definition 1, then it satisfies the Definition 2.

Let x be a string which is constructed using the Definition 2. We show by induction on $|x|$ that it satisfies the Definition 1. There is basis for induction by rule (ii). Assume the result for all strings with length $< n$. Let x be a string of length n. As x has to be constructed using the rule (iii), $x = aya$, where y is a palindrome. As y is a palindrome by Definition 2 and $|y| < n$, it satisfies the Definition 1. So, $x = aya$ also satisfies the Definition 1.

EXAMPLE 2.24

Prove the pigeonhole principle.

Proof We prove the theorem by induction on m. If $m = 1$ and $n > 1$, then all these n items must be placed in a single place. Hence the theorem is true for $m = 1$.

Assume the theorem for m. Consider the case of $m + 1$ places. We prove the theorem for $n = m + 2$. (If $n > m + 2$, already one of the $m + 1$ places will receive at least two objects from $m + 2$ objects, by what we are going to prove.) Consider a particular place, say, P.

Three cases arise:

(i) P contains at least two objects.
(ii) P contains one object
(iii) P contains no object.

In case (i), the theorem is proved for $n = m + 2$. Consider case (ii). As P contains one object, the remaining m places should receive $m + 1$ objects. By induction hypothesis, at least one place (not the same as P) contains at least two objects. In case (iii), $m + 2$ objects are distributed among m places. Once again, by induction hypothesis, one place (other than P) receives at least two objects. Hence, in all the cases, the theorem is true for $(m + 1)$ places. By the principle of induction, the theorem is true for all m.

2.4.3 SIMULTANEOUS INDUCTION

Sometimes we may have a pair of related identities. To prove these, we may apply two induction proofs simultaneously. Example 2.25 illustrates this method.

EXAMPLE 2.25

A sequence $F_0, F_1, F_2, \ldots$ called the sequence of Fibonacci numbers (named after the Italian mathematician Leonardo Fibonacci) is defined recursively as follows:

$$F_{n+1} = F_n + F_{n-1}, \qquad F_0 = 0, \qquad F_1 = 1$$

Prove that:

$$P_n : F_n^2 + F_{n-1}^2 = F_{2n-1} \tag{2.2}$$

$$Q_n : F_{n+1}F_n + F_nF_{n-1} = F_{2n} \tag{2.3}$$

Proof We prove the two identities (2.2) and (2.3) simultaneously by simultaneous induction. P_1 and Q_1 are $F_1^2 + F_0^2 = F_1$ and $F_2F_1 + F_1F_0 = F_2$

respectively. As $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, these are true. Hence there is basis for induction. Assume P_n and Q_n. So

$$F_n^2 + F_{n-1}^2 = F_{2n-1} \tag{2.2}$$

$$F_{n+1}F_n + F_nF_{n-1} = F_{2n} \tag{2.3}$$

Now,

$$\begin{aligned}
F_{n+1}^2 + F_n^2 &= (F_{n-1} + F_n)^2 + F_n^2 \\
&= F_{n-1}^2 + F_n^2 + 2F_{n-1}F_n + F_n^2 \\
&= (F_{n-1}^2 + F_n^2) + F_{n-1}F_n + F_n^2 + F_{n-1}F_n \\
&= F_{n-1}^2 + F_n^2 + (F_{n-1} + F_n)F_n + F_nF_{n-1} \\
&= (F_{n-1}^2 + F_n^2) + F_{n+1}F_n + F_nF_{n-1} \\
&= F_{2n-1} + F_{2n} \qquad \text{(by (2.3))} \\
&= F_{2n+1}
\end{aligned}$$

This proves P_{n+1}.

Also,

$$\begin{aligned}
F_{n+2}F_{n+1} + F_{n+1}F_n &= (F_{n+1} + F_n)F_{n+1} + (F_n + F_{n-1})F_n \\
&= F_{n+1}^2 + F_{n+1}F_n + F_nF_{n-1} + F_n^2 \\
&= (F_{n+1}^2 + F_n^2) + (F_{n+1}F_n + F_nF_{n-1}) \\
&= F_{2n+1} + F_{2n} \qquad \text{(By } P_{n+1} \text{ and (2.3))} \\
&= F_{2n+2}
\end{aligned}$$

This proves Q_{n+1}.

So, by induction (2.2) and (2.3) are true for all n.

We conclude this chapter with the method of proof by contradiction.

2.5 PROOF BY CONTRADICTION

Suppose we want to prove a property P under certain conditions. The method of proof by contradiction is as follows:

Assume that property P is not true. By logical reasoning get a conclusion which is either absurd or contradicts the given conditions.

The following example illustrates the use of proof by contradiction and proof by induction.

EXAMPLE 2.26

Prove that there is no string x in $\{a, b\}^*$ such that $ax = xb$. (For the definition of strings, refer to Section 2.3.)

Proof We prove the result by induction on the length of x. When $|x| = 1$, $x = a$ or $x = b$. In both cases $ax \neq xb$. So there is basis for induction. Assume the result for any string whose length is less than n. Let x be any string of length n. We prove that $ax \neq xb$ through proof by contradiction. Suppose $ax = xb$. As a is the first symbol on the L.H.S., the first symbol of x is a. As b is the last symbol on R.H.S., the last symbol of x is b. So, we can write x as ayb with $|y| = n - 2$. This means $aayb = aybb$ which implies $ay = yb$. This contradicts the induction hypothesis. Thus, $ax \neq xb$. By induction the result is true for all strings.

2.6 SUPPLEMENTARY EXAMPLES

EXAMPLE 2.27

In a survey of 600 people, it was found that:

250 read the *Week*
260 read the *Reader's Digest*
260 read the *Frontline*
90 read both *Week* and *Frontline*
110 read both *Week* and *Reader's Digest*
80 read both *Reader's Digest* and *Frontline*
30 read all the three magazines.

(a) Find the number of people who read at least one of the three magazines.
(b) Find the number of people who read none of these magazines.
(c) Find the number of people who read exactly one magazine.

Solution

Let W, R, F denote the set of people who read *Week*, *Reader's Digest* and *Frontline*, respectively. We use the Venn diagram to represent these sets (see Fig. 2.17).

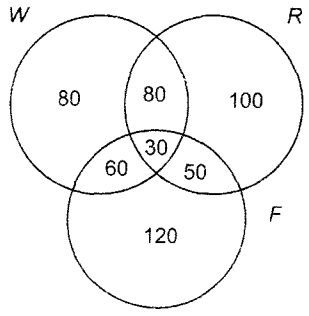


Fig. 2.17 Venn diagram for Example 2.27.

It is given that:

$$|W| = 250, \quad |R| = 260, \quad |F| = 260, \quad |W \cap F| = 90,$$
$$|W \cap R| = 110, \quad |R \cap F| = 80, \quad |W \cap R \cap F| = 30$$

$\therefore$ $|W \cup R \cup F|$

$= |W| + |R| + |F| - |W \cap F| - |W \cap R| - |R \cap F| + |W \cap R \cap F|$

$= 250 + 260 + 260 - 90 - 110 - 80 + 30 = 520$

So the solution for (a) is 520.

(b) The number of people who read none of the magazines

$$= 600 - 520 = 80$$

Using the data, we fill up the various regions of the Venn diagram.

(c) The number of people who read only one magazine

$$= 80 + 100 + 120 = 300$$

EXAMPLE 2.28

Prove that $(A \cup B \cup C)^c = (A \cup B)^c \cap (A \cup C)^c$

Solution

$$(A \cup B \cup C) = (A \cup A) \cup B \cup C$$
$$= A \cup (A \cup B) \cup C = (A \cup B) \cup (A \cup C)$$

Hence $(A \cup B \cup C)^c = (A \cup B)^c \cap (A \cup C)^c$ (by DeMorgan's law)

EXAMPLE 2.29

Define aRb if $b = a^k$ for some positive integer k; $a, b \in z$. Show that R is a partial ordering. (A relation is a partial ordering if it is reflexive, antisymmetric and transitive.)

Solution

As $a = a^1$, we have aRa. To prove that R is antisymmetric, we have to prove that aRb and $bRa \Rightarrow a = b$. As aRb, we have $b = a^k$. As bRa, we have $a = b^l$. Hence $a = b^l = (a^k)^l = a^{kl}$. This is possible only when one of the following holds good:

(i) $a = 1$
(ii) $a = -1$
(iii) $kl = 1$

In case (i), $b = a^k = 1$. So $a = b$.

In case (ii), $a = -1$ and so kl is odd. This implies that both k and l are odd. So

$$b = a^k = (-1)^k = -1 = a$$

In case (iii), $kl = 1$. As k and l are positive integers, $k = l = 1$. So

$$b = a^k = a.$$

If aRb and bRc, then $b = a^k$ and $c = b^l$ for some k, l. Therefore, $c = a^{kl}$. Hence aRc.

EXAMPLE 2.30

Suppose $A = \{1, 2, \ldots, 9\}$ and $\sim$ relation on $A \times A$ is defined by $(m, n) \sim (p, q)$ if $m + q = n + p$, then prove that $\sim$ is an equivalence relation.

Solution

$(m, n) \sim (m, n)$ since $m + n = n + m$. So $\sim$ is reflexive. If $(m, n) \sim (p, q)$, then $m + q = n + p$; thus $p + n = q + m$. Hence $(p, q) \sim (m, n)$. So $\sim$ is symmetric.

If $(m, n) \sim (p, q)$ and $(p, q) \sim (r, s)$, then

$$m + q = n + p \quad \text{and} \quad p + s = q + r$$

Adding these,

$$m + q + p + s = n + p + q + r$$

That is,

$$m + s = n + r$$

which proves $(m, n) \sim (r, s)$.
Hence $\sim$ is an equivalence relation.

EXAMPLE 2.31

If $f : A \to B$ and $g : B \to C$ are one-to-one, prove that $g \circ f$ is one-to-one.

Solution

Let us assume that $g \circ f(a_1) = g \circ f(a_2)$. Then, $g(f(a_1)) = g(f(a_2))$. As g is one-to-one, $f(a_1) = f(a_2)$. As f is one-to-one, $a_1 = a_2$. Hence $g \circ f$ is one-to-one.

EXAMPLE 2.32

Show that a connected graph G with n vertices and $n - 1$ edges ($n \geq 3$) has at least one leaf.

Solution

G has n vertices and $n - 1$ edges. Every edge is counted twice while computing the degree of each of its end vertices. Hence

$$\Sigma \deg(v) = 2(n - 1)$$

where summation is taken over all vertices of G.

So, $\Sigma \deg(v)$ is the sum of n positive integers. If $\deg(v) \geq 2$ for every vertex v of G, then

$$2n \leq \Sigma \deg(v) = 2(n - 1)$$

which is not possible.

Hence $\deg(v) = 1$ for at least one vertex v of G and this vertex v is a leaf.

EXAMPLE 2.33

Prove Property 5 stated in Section 2.2.2.

Solution

We prove the result by induction on n. Obviously, there is basis for induction. Assume the result for connected graphs with $n - 1$ vertices. Let T be a connected graph with n vertices and $n - 1$ edges. By Example 2.32, T has at least one leaf v (say).

Drop the vertex v and the (single) edge incident with v. The resulting graph G' is still connected and has $n - 1$ vertices and $n - 2$ edges. By induction hypothesis, G' is a tree. So G' has no circuits and hence G also has no circuits. (Addition of the edge incident with v does not create a circuit in G.) Hence G is a tree. By the principle of induction, the property is true for all n.

EXAMPLE 2.34

A person climbs a staircase by climbing either (i) two steps in a single stride or (ii) only one step in a single stride. Find a formula for $S(n)$, where $S(n)$ denotes the number of ways of climbing n stairs.

Solution

When there is a single stair, there is only one way of climbing up. Hence $S(1) = 1$. For climbing two stairs, there are two ways, viz. two steps in a single stride or two single steps. So $S(2) = 2$. In reaching n steps, the person can climb either one step or two steps in his last stride. For these two choices, the number of ways are $S(n - 1)$ and $S(n - 2)$.

So,

$$S(n) = S(n - 1) + S(n - 2)$$

Thus, $S(n) = F(n)$, the nth Fibonacci number (refer to Exercise 2.20, at the end of this chapter).

EXAMPLE 2.35

How many subsets does the set $\{1, 2, \ldots, n\}$ have that contain no two consecutive integers?

Solution

Let S_n denote the number of subsets of $\{1, 2, \ldots, n\}$ having the desired property. If $n = 1$, $S_1 = |\{\emptyset, \{1\}\}| = 2$. If $n = 2$, then $S_2 = |\{\emptyset, \{1\}, \{2\}\}| = 3$.

Consider a set A with n elements. If a subset having the desired property contains n, it cannot contain $n - 1$. So there are S_{n-2} such subsets. If it does not contain n, there are S_{n-1} such subsets. So $S_n = S_{n-1} + S_{n-2}$. As $S_1 = 2 = F_3$ and $S_2 = 3 = F_4$,

$$S_n = F_{n+2}$$

the $(n + 2)$th Fibonacci number.

EXAMPLE 2.36

If $n \geq 1$, show that

$$1 \cdot 1! + 2 \cdot 2! + \cdots + n \cdot n! = (n + 1)! - 1$$

Solution

We prove the result by induction on n. If $n = 1$, then $1 \cdot 1! = 1 = (1 + 1)! - 1$. So there is basis for induction.

Assume the result for n, i.e.

$$1 \cdot 1! + 2 \cdot 2! + \cdots + n \cdot n! = (n + 1)! - 1$$

Then,

$$\begin{aligned} &1 \cdot 1! + 2 \cdot 2! + \cdots + n \cdot n! + (n + 1) \cdot (n + 1)! \\ &= (n + 1)! - 1 + (n + 1) \cdot (n + 1)! \\ &= (n + 1)! \, (1 + n + 1) - 1 = (n + 2)! - 1 \end{aligned}$$

Hence the result is true for $n + 1$ and by the principle of induction, the result is true for all $n \geq 1$.

EXAMPLE 2.37

Using induction, prove that $2^n < n!$ for all $n \geq 4$.

Solution

For $n = 4$, $2^4 < 4!$. So there is basis for induction. Assume $2^n < n!$. Then,

$$2^{n+1} = 2^n \cdot 2 < n! \cdot 2 < (n + 1)n! = (n + 1)!$$

By induction, the result is true for all $n \geq 4$.

SELF-TEST

Choose the correct answer to Questions 1–10:

1. $(A \cup A) \cap (B \cap B)$ is

(a) A (b) $A \cap B$ (c) B (d) none of these

2. The reflexive–transitive closure of the relation $\{(1, 2), (2, 3)\}$ is

(a) $\{(1, 2), (2, 3), (1, 3)\}$

(b) $\{(1, 2), (2, 3), (1, 3), (3, 1)\}$

(c) $\{(1, 1), (2, 2), (3, 3), (1, 3), (1, 2), (2, 3)\}$

(d) $\{(1, 1), (2, 2), (3, 3), (1, 3)\}$

3. There exists a function

$$f : \{1, 2, \ldots, 10\} \to \{2, 3, 4, 5, 6, 7, 9, 10, 11, 12\}$$

which is
(a) one-to-one and onto
(b) one-to-one but not onto
(c) onto but not one-to-one
(d) none of these

4. A tree with 10 vertices has
(a) 10 edges (b) 9 edges (c) 8 edges (d) 7 edges.

5. The number of binary trees with 7 vertices is
(a) 7 (b) 6 (c) 2 (d) 1

6. Let $N = \{1, 2, 3, \ldots\}$. Then $f : N \to N$ defined by $f(n) = n + 1$ is
(a) onto but not one-to-one
(b) one-to-one but not onto
(c) both one-to-one and onto
(d) neither one-to-one nor onto

7. *QST* is a substring of
(a) *PQRST*
(b) *QRSTU*
(c) *QSPQSTUT*
(d) *QQSSTT*

8. If $x = 01$, $y = 101$ and $z = 011$, then $xyzy$ is
(a) 01011011
(b) 01101101011
(c) 01011101101
(d) 01101011101

9. A binary tree with seven vertices has
(a) one leaf
(b) two leaves
(c) three leaves
(d) four leaves

10. A binary operation o on $N = \{1, 2, 3, \ldots\}$ is defined by a o $b = a + 2b$. Then:
(a) o is commutative
(b) o is associative
(c) N has an identity element with respect to o
(d) none of these

EXERCISES

2.1 If $A = \{a, b\}$ and $B = \{b, c\}$, find:
(a) $(A \cup B)^*$
(b) $(A \cap B)^*$
(c) $A^* \cup B^*$

(d) $A^* \cap B^*$
(e) $(A - B)^*$
(f) $(B - A)^*$

2.2 Let $S = \{a, b\}^*$. For $x, y \in S$, define $x \circ y = xy$, i.e. $x \circ y$ is obtained by concatenating x and y.
(a) Is S closed under o?
(b) Is o associative?
(c) Does S have the identity element with respect to o?
(d) Is o commutative?

2.3 Let $S = 2^X$, where X is any nonempty set. For $A, B \subseteq X$, let $A \circ B = A \cup B$.
(a) Is o commutative and associative?
(b) Does S have the identity element with respect to o?
(c) If $A \circ B = A \circ C$, does it imply that $B = C$?

2.4 Test whether the following statements are true or false. Justify your answer.
(a) The set of all odd integers is a monoid under multiplication.
(b) The set of all complex numbers is a group under multiplication.
(c) The set of all integers under the operation o given by $a \circ b = a + b - ab$ is a monoid.
(d) 2^S under symmetric difference $\bar{\vee}$ defined by $A \bar{\vee} B = (A - B) \cup (B - A)$ is an abelian group.

2.5 Show that the following relations are equivalence relations:
(a) On a set S, aRb if $a = b$.
(b) On the set of all lines in the plane, $l_1 R l_2$ if l_1 is parallel to l_2.
(c) On $N = \{0, 1, 2, \ldots\}$, mRn if m differs from n by a multiple of 3.

2.6 Show that the following are not equivalence relations:
(a) On a set S, aRb if $a \neq b$.
(b) On the set of lines in the plane, $l_1 R l_2$ if l_1 is perpendicular to l_2.
(c) On $N = \{0, 1, 2, \ldots\}$, mRn if m divides n.
(d) On $S = \{1, 2, \ldots, 10\}$, aRb if $a + b = 10$.

2.7 For x, y in $\{a, b\}^*$, define a relation R by xRy if $|x| = |y|$. Show that R is an equivalence relation. What are the equivalence classes?

2.8 For x, y in $\{a, b\}^*$, define a relation R by xRy if x is a substring of y (x is a substring of y if $y = z_1 x z_2$ for some string z_1, z_2). Is R an equivalence relation?

2.9 Let $R = \{(1, 2), (2, 3), (1, 4), (4, 2), (3, 4)\}$. Find R^+, R^*.

2.10 Find R^* for the following relations:
(a) $R = \{(1, 1), (1, 2), (2, 1), (2, 3), (3, 2)\}$
(b) $R = \{(1, 1), (2, 3), (3, 4), (3, 2)\}$

(c) $R = \{(1, 1), (2, 2), (3, 3), (4, 4)\}$
(d) $R = \{(1, 2), (2, 3), (3, 1), (4, 4)\}$

2.11 If R is an equivalence relation on S, what can you say about R^+, R^*?

2.12 Let $f:\{a, b\}^* \rightarrow \{a, b\}^*$ be given by $f(x) = ax$ for every $x \in \{a, b\}^*$. Show that f is one-to-one but not onto.

2.13 Let $g : \{a, b\}^* \rightarrow \{a, b\}^*$ be given by $g(x) = x^T$. Show that g is one-to-one and onto.

2.14 Give an example of (a) a tree with six vertices and (b) a binary tree with seven vertices.

2.15 For the tree T given in Fig. 2.18, answer the following questions:
(a) Is T a binary tree?
(b) Which vertices are the leaves of T?
(c) How many internal vertices are in T?
(d) What is the height of T?
(e) What is the left-to-right ordering of leaves?
(f) Which vertex is the father of 5?
(g) Which vertices are the sons of 3?

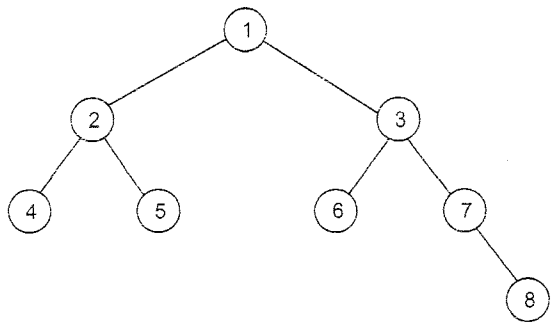


Fig. 2.18 The tree for Exercise 2.15.

2.16 In a get-together, show that the number of persons who know an odd number of persons is even.
[*Hint:* Use a graph.]

2.17 If X is a finite set, show that $|2^X| = 2^{|X|}$.

2.18 Prove the following by the principle of induction:

(a) $\displaystyle\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$

(b) $\displaystyle\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{(n+1)}$

(c) $10^{2n} - 1$ is divisible by 11 for all $n > 1$.

2.19 Prove the following by the principle of induction:

(a) $1 + 4 + 7 + \cdots + (3n - 2) = \dfrac{n(3n - 1)}{2}$

(b) $2^n > n$ for all $n > 1$

(c) If $f(2) = 2$ and $f(2^k) = 2f(2^{k-1}) + 3$, then $f(2^k) = (5/2) \cdot 2^k - 3$.

2.20 The Fibonacci numbers are defined in the following way:

$$F(0) = 1, \qquad F(1) = 1, \qquad F(n + 1) = F(n) + F(n - 1)$$

Prove by induction that:

(a) $F(2n + 1) = \displaystyle\sum_{k=0}^{n} F(2k)$

(b) $F(2n + 2) = \displaystyle\sum_{k=1}^{n} F(2k + 1) + 1$

2.21 Show that the maximum number of edges in a simple graph (i.e. a graph having no self-loops or parallel edges) is $\dfrac{n(n - 1)}{2}$.

2.22 If $w \in \{a, b\}^*$ satisfies the relation $abw = wab$, show that $|w|$ is even.

2.23 Suppose there are an infinite number of envelopes arranged one after another and each envelope contains the instruction 'open the next envelope'. If a person opens an envelope, he has to then follow the instruction contained therein. Show that if a person opens the first envelope, he has to open all the envelopes.

3 The Theory of Automata

In this chapter we begin with the study of automaton. We deal with transition systems which are more general than finite automata. We define the acceptability of strings by finite automata and prove that nondeterministic finite automata have the same capability as the deterministic automata as far as acceptability is concerned. Besides, we discuss the equivalence of Mealy and Moore models. Finally, in the last section, we give an algorithm to construct a minimum state automaton equivalent to a given finite automaton.

3.1 DEFINITION OF AN AUTOMATON

We shall give the most general definition of an automaton and later modify it to computer applications. An automaton is defined as a system where energy, materials and information are transformed, transmitted and used for performing some functions without direct participation of man. Examples are automatic machine tools, automatic packing machines, and automatic photo printing machines.

In computer science the term 'automaton' means 'discrete automaton' and is defined in a more abstract way as shown in Fig. 3.1.

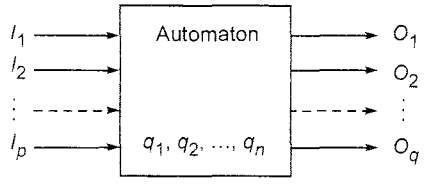


Fig. 3.1 Model of a discrete automaton.

The characteristics of automaton are now described.

(i) *Input.* At each of the discrete instants of time $t_1, t_2, \ldots, t_m$ the input values $I_1, I_2, \ldots, I_p$, each of which can take a finite number of fixed values from the input alphabet Σ, are applied to the input side of the model shown in Fig. 3.1.

(ii) *Output.* $O_1, O_2, \ldots, O_q$ are the outputs of the model, each of which can take a finite number of fixed values from an output O.

(iii) *States.* At any instant of time the automaton can be in one of the states $q_1, q_2, \ldots, q_n$.

(iv) *State relation.* The next state of an automaton at any instant of time is determined by the present state and the present input.

(v) *Output relation.* The output is related to either state only or to both the input and the state. It should be noted that at any instant of time the automaton is in some state. On 'reading' an input symbol, the automaton moves to a next state which is given by the state relation.

Note: An automaton in which the output depends only on the input is called an automaton without a memory. An automaton in which the output depends on the states as well, is called automaton with a finite memory. An automaton in which the output depends only on the states of the machine is called a *Moore machine.* An automaton in which the output depends on the state as well as on the input at any instant of time is called a *Mealy machine.*

EXAMPLE 3.1

Consider the simple shift register shown in Fig. 3.2 as a finite-state machine and study its operation.

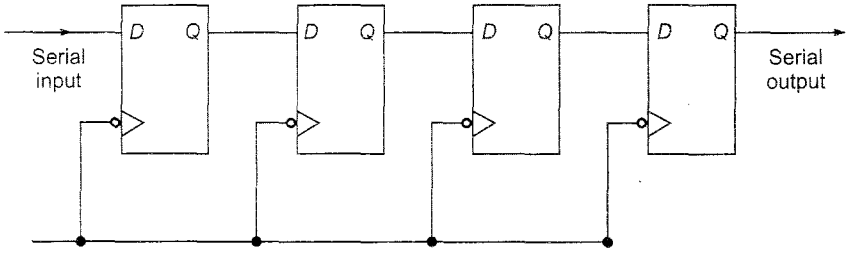


Fig. 3.2 A 4-bit serial shift register using D flip-flops.

Solution

The shift register (Fig. 3.2) can have $2^4 = 16$ states (0000, 0001, ..., 1111), and one serial input and one serial output. The input alphabet is $\Sigma = \{0, 1\}$, and the output alphabet is $O = \{0, 1\}$. This 4-bit serial shift register can be further represented as in Fig. 3.3.

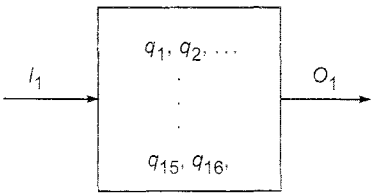


Fig. 3.3 A shift register as a finite-state machine.

From the operation, it is clear that the output will depend upon both the input and the state and so it is a Mealy machine.

In general, any sequential machine behaviour can be represented by an automaton.

3.2 DESCRIPTION OF A FINITE AUTOMATON

Definition 3.1 Analytically, a finite automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(i) Q is a finite nonempty set of states.
(ii) Σ is a finite nonempty set of inputs called the *input alphabet.*
(iii) δ is a function which maps $Q \times \Sigma$ into Q and is usually called the *direct transition function.* This is the function which describes the change of states during the transition. This mapping is usually represented by a transition table or a transition diagram.
(iv) $q_0 \in Q$ is the initial state.
(v) $F \subseteq Q$ is the set of final states. It is assumed here that there may be more than one final state.

Note: The transition function which maps $Q \times \Sigma^*$ into Q (i.e. maps a state and a string of input symbols including the empty string into a state) is called the *indirect transition function.* We shall use the same symbol δ to represent both types of transition functions and the difference can be easily identified by the nature of mapping (symbol or a string), i.e. by the argument. δ is also called the next state function. The above model can be represented graphically by Fig. 3.4.

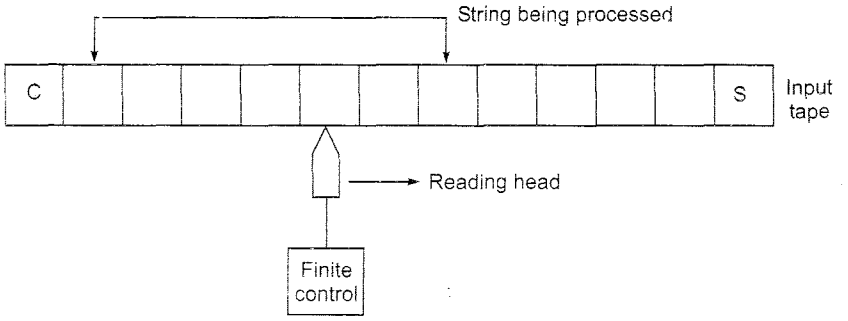


Fig. 3.4 Block diagram of a finite automaton.

Figure 3.4 is the block diagram for a finite automaton. The various components are explained as follows:

(i) *Input tape.* The input tape is divided into squares, each square containing a single symbol from the input alphabet Σ. The end squares of the tape contain the endmarker ¢ at the left end and the endmarker $ at the right end. The absence of endmarkers indicates that the tape is of infinite length. The left-to-right sequence of symbols between the two endmarkers is the input string to be processed.

(ii) *Reading head.* The head examines only one square at a time and can move one square either to the left or to the right. For further analysis, we restrict the movement of the R-head only to the right side.

(iii) *Finite control.* The input to the finite control will usually be the symbol under the R-head, say a, and the present state of the machine, say q, to give the following outputs: (a) A motion of R-head along the tape to the next square (in some a null move, i.e. the R-head remaining to the same square is permitted); (b) the next state of the finite state machine given by $\delta(q, a)$.

3.3 TRANSITION SYSTEMS

A transition graph or a transition system is a finite directed labelled graph in which each vertex (or node) represents a state and the directed edges indicate the transition of a state and the edges are labelled with input/output.

A typical transition system is shown in Fig. 3.5. In the figure, the initial state is represented by a circle with an arrow pointing towards it, the final state by two concentric circles, and the other states are represented by just a circle. The edges are labelled by input/output (e.g. by 1/0 or 1/1). For example, if the system is in the state q_0 and the input 1 is applied, the system moves to state q_1 as there is a directed edge from q_0 to q_1 with label 1/0. It outputs 0.

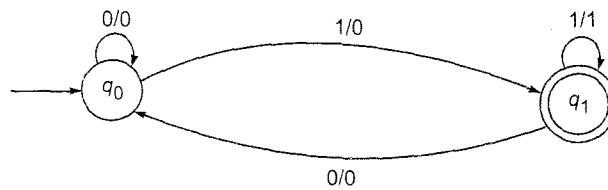


Fig. 3.5 A transition system.

We now give the (analytical) definition of a transition system.

Definition 3.2 A transition system is a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$, where

(i) Q, Σ and F are the finite nonempty set of states, the input alphabet, and the set of final states, respectively, as in the case of finite automata;

(ii) $Q_0 \subseteq Q$, and Q_0 is nonempty; and

(iii) δ is a finite subset of $Q \times \Sigma^* \times Q$.

In other words, if (q_1, w, q_2) is in δ, it means that the graph starts at the vertex q_1, goes along a set of edges, and reaches the vertex q_2. The concatenation of the label of all the edges thus encountered is w.

Definition 3.3 A transition system accepts a string w in Σ^* if

(i) there exists a path which originates from some initial state, goes along the arrows, and terminates at some final state; and

(ii) the path value obtained by concatenation of all edge-labels of the path is equal to w.

Example 3.2

Consider the transition system given in Fig. 3.6.

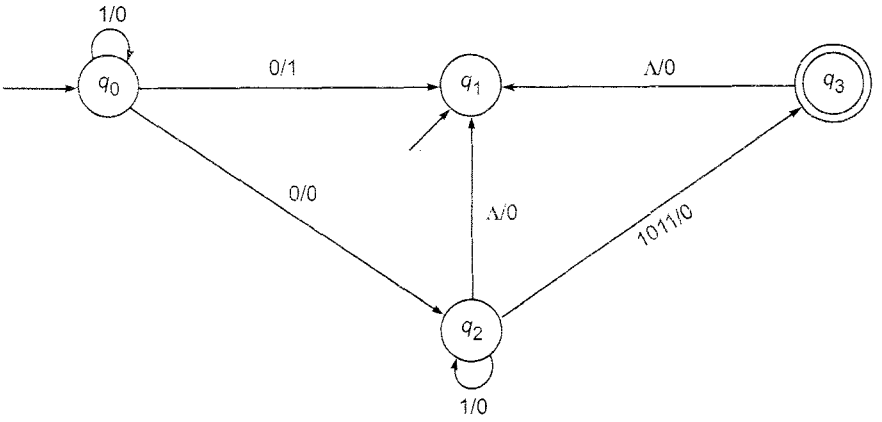


Fig. 3.6 Transition system for Example 3.2.

Determine the initial states, the final states, and the acceptability of 101011, 111010.

Solution

The initial states are q_0 and q_1. There is only one final state, namely q_3.

The path-value of $q_0q_0q_2q_3$ is 101011. As q_3 is the final state, 101011 is accepted by the transition system. But, 111010 is not accepted by the transition system as there is no path with path value 111010.

Note: Every finite automaton $(Q, \Sigma, \delta, q_0, F)$ can be viewed as a transition system $(Q, \Sigma, \delta' Q_0, F)$ if we take $Q_0 = \{q_0\}$ and $\delta' = \{(q, w, \delta(q, w))|q \in Q, w \in \Sigma^*\}$. But a transition system need not be a finite automaton. For example, a transition system may contain more than one initial state.

3.4 PROPERTIES OF TRANSITION FUNCTIONS

Property 1 $\delta(q, \Lambda) = q$ is a finite automaton. This means that the state of the system can be changed only by an input symbol.

Property 2 For all strings w and input symbols a,

$$\delta(q, aw) = \delta(\delta(q, a), w)$$

$$\delta(q, wa) = \delta(\delta(q, w), a)$$

This property gives the state after the automaton consumes or reads the first symbol of a string aw and the state after the automaton consumes a prefix of the string wa.

EXAMPLE 3.3

Prove that for any transition function δ and for any two input strings x and y,

$$\delta(q, xy) = \delta(\delta(q, x), y) \tag{3.1}$$

Proof By the method of induction on $|y|$, i.e. length of y.
Basis: When $|y| = 1$, $y = a \in \Sigma$

$$\begin{aligned} \text{L.H.S. of (3.1)} &= \delta(q, xa) \\ &= \delta(\delta(q, x), a) \quad \text{by Property 2} \\ &= \text{R.H.S. of (3.1)} \end{aligned}$$

Assume the result, i.e. (3.1) for all strings x and strings y with $|y| = n$. Let y be a string of length $n + 1$. Write $y = y_1 a$ where $|y_1| = n$.

$$\begin{aligned} \text{L.H.S. of (3.1)} &= \delta(q, xy_1a) = \delta(q, x_1a), \quad x_1 = xy_1 \\ &= \delta(\delta(q, x_1), a) \quad \text{by Property 2} \\ &= \delta(\delta(q, xy_1), a) \\ &= \delta(\delta(\delta(q, x), y_1), a) \quad \text{by induction hypothesis} \\ \text{R.H.S. of (3.1)} &= \delta(\delta(q, x), y_1a) \\ &= \delta(\delta(\delta(q, x), y_1), a) \quad \text{by Property 2} \end{aligned}$$

Hence, L.H.S. = R.H.S. This proves (3.1) for any string y of length $n + 1$. By the principle of induction, (3.1) is true for all strings. ∎

EXAMPLE 3.4

Prove that if $\delta(q, x) = \delta(q, y)$, then $\delta(q, xz) = \delta(q, yz)$ for all strings z in Σ^+.

Solution

$$\begin{aligned} \delta(q, xz) &= \delta(\delta(q, x), z) \quad \text{by Example 3.3} \\ &= \delta(\delta(q, y), z) \end{aligned} \tag{3.2}$$

By Example 3.3,

$$\begin{aligned} \delta(q, yz) &= \delta(\delta(q, y), z) \\ &= \delta(q, xz) \end{aligned} \tag{3.3}$$

3.5 ACCEPTABILITY OF A STRING BY A FINITE AUTOMATON

Definition 3.4 A string x is accepted by a finite automaton

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

if $\delta(q_0, x) = q$ for some $q \in F$.

This is basically the acceptability of a string by the final state.

Note: A final state is also called an accepting state.

EXAMPLE 3.5

Consider the finite state machine whose transition function δ is given by Table 3.1 in the form of a transition table. Here, $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$. Give the entire sequence of states for the input string 110001.

TABLE 3.1 Transition Function Table for Example 3.5

State	Input	
	0	1
$\rightarrow$ (q_0)	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Solution

$$\begin{aligned}
\delta(q_0, 110101) &= \delta(q_1, 10101) \\
&= \delta(q_0, 0101) \\
&= \delta(q_2, 101) \\
&= \delta(q_3, 01) \\
&= \delta(q_1, 1) \\
&= \delta(q_0, \Lambda) \\
&= q_0
\end{aligned}$$

Hence,

$$q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_0 \xrightarrow{0} q_2 \xrightarrow{1} q_3 \xrightarrow{0} q_1 \xrightarrow{1} q_0$$

The symbol $\downarrow$ indicates that the current input symbol is being processed by the machine.

3.6 NONDETERMINISTIC FINITE STATE MACHINES

We explain the concept of nondeterministic finite automaton using a transition diagram (Fig. 3.7).

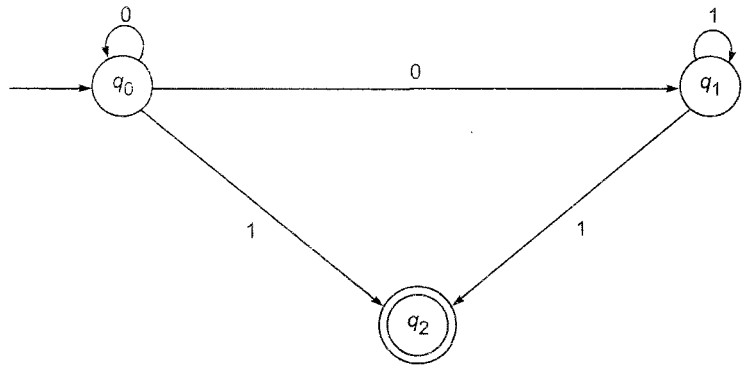


Fig. 3.7 Transition system representing nondeterministic automaton.

If the automaton is in a state $\{q_0\}$ and the input symbol is 0, what will be the next state? From the figure it is clear that the next state will be either $\{q_0\}$ or $\{q_1\}$. Thus some moves of the machine cannot be determined uniquely by the input symbol and the present state. Such machines are called nondeterministic automata, the formal definition of which is now given.

Definition 3.5 A nondeterministic finite automaton (NDFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(i) Q is a finite nonempty set of states;

(ii) Σ is a finite nonempty set of inputs;

(iii) δ is the transition function mapping from $Q \times \Sigma$ into 2^Q which is the power set of Q, the set of all subsets of Q;

(iv) $q_0 \in Q$ is the initial state; and

(v) $F \subseteq Q$ is the set of final states.

We note that the difference between the deterministic and nondeterministic automata is only in δ. For deterministic automaton (DFA), the outcome is a state, i.e. an element of Q; for nondeterministic automaton the outcome is a subset of Q.

Consider, for example, the nondeterministic automaton whose transition diagram is described by Fig. 3.8.

The sequence of states for the input string 0100 is given in Fig. 3.9. Hence,

$$\delta(q_0, 0100) = \{q_0, q_3, q_4\}$$

Since q_4 is an accepting state, the input string 0100 will be accepted by the nondeterministic automaton.

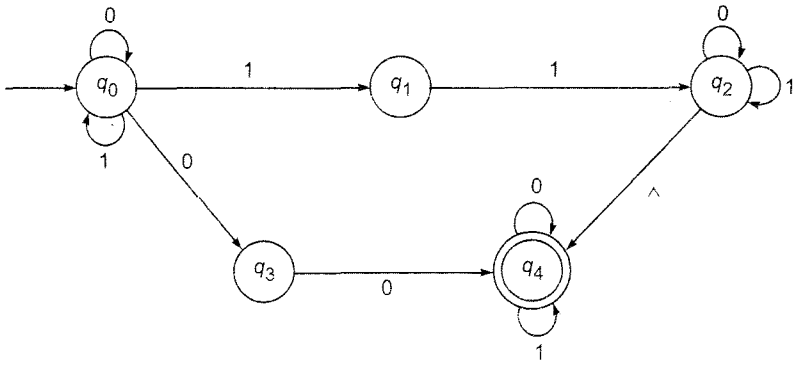


Fig. 3.8 Transition system for a nondeterministic automaton.

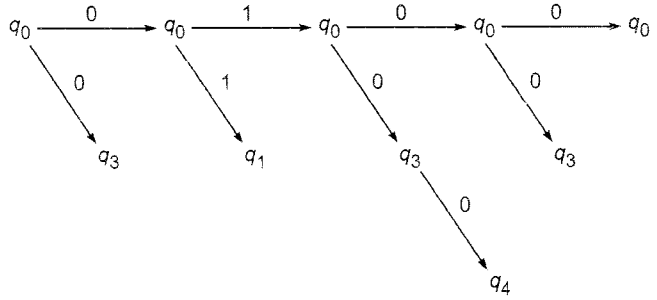


Fig. 3.9 States reached while processing 0100.

Definition 3.6 A string $w \in \Sigma^*$ is accepted by NDFA M if $\delta(q_0, w)$ contains some final state.

Note: As M is nondeterministic, $\delta(q_0, w)$ may have more than one state. So w is accepted by M if a final state is *one* among the possible states that M can reach on application of w.

We can visualize the working of an NDFA M as follows: Suppose M reaches a state q and reads an input symbol a. If $\delta(q, a)$ has n elements, the automaton splits into n identical copies of itself: each copy pursuing one choice determined by an element of $\delta(q, a)$. This type of parallel computation continues. When a copy encounters (q, a) for which $\delta(q, a) = \emptyset$, this copy of the machine 'dies'; however the computation is pursued by the other copies. If any one of the copies of M reaches a final state after processing the entire input string w, then we say that M accepts w. Another way of looking at the computation by an NDFA M is to assign a tree structure for computing $\delta(q, w)$. The root of the tree has the label q. For every input symbol in w, the tree branches itself. When a leaf of the tree has a final state as its label, then M accepts w.

Definition 3.7 The set accepted by an automaton M (deterministic or nondeterministic) is the set of all input strings accepted by M. It is denoted by $T(M)$.

3.7 THE EQUIVALENCE OF DFA AND NDFA

We naturally try to find the relation between DFA and NDFA. Intuitively we now feel that:

(i) A DFA can simulate the behaviour of NDFA by increasing the number of states. (In other words, a DFA $(Q, \Sigma, \delta, q_0, F)$ can be viewed as an NDFA $(Q, \Sigma, \delta', q_0, F)$ by defining $\delta'(q, a) = \{\delta(q, a)\}$.)

(ii) Any NDFA is a more general machine without being more powerful.

We now give a theorem on equivalence of DFA and NDFA.

Theorem 3.1 For every NDFA, there exists a DFA which simulates the behaviour of NDFA. Alternatively, if L is the set accepted by NDFA, then there exists a DFA which also accepts L.

Proof Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NDFA accepting L. We construct a DFA M' as:

$$M' = (Q', \Sigma, \delta, q'_0, F')$$

where

(i) $Q' = 2^Q$ (any state in Q' is denoted by $[q_1, q_2, \ldots, q_j]$, where $q_1, q_2, \ldots, q_j \in Q$);

(ii) $q'_0 = [q_0]$; and

(iii) F' is the set of all subsets of Q containing an element of F.

Before defining δ', let us look at the construction of Q', q'_0 and F'. M is initially at q_0. But on application of an input symbol, say a, M can reach any of the states $\delta(q_0, a)$. To describe M, just after the application of the input symbol a, we require all the possible states that M can reach after the application of a. So, M' has to remember all these possible states at any instant of time. Hence the states of M' are defined as subsets of Q. As M starts with the initial state q_0, q'_0 is defined as $[q_0]$. A string w belongs to $T(M)$ if a final state is one of the possible states that M reaches on processing w. So, a final state in M' (i.e. an element of F') is any subset of Q containing some final state of M.

Now we can define δ':

(iv) $\delta'([q_1, q_2, \ldots, q_i], a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \ldots \cup \delta(q_i, a)$.

Equivalently,

$$\delta'([q_1, q_2, \ldots, q_i], a) = [p_1, \ldots, p_j]$$

if and only if

$$\delta(\{q_1, \ldots, q_i\}, a) = \{p_1, p_2, \ldots, p_j\}.$$

Before proving $L = T(M')$, we prove an auxiliary result

$$\delta'(q'_0, x) = [q_1, \ldots, q_i], \tag{3.4}$$

if and only if $\delta(q_0, x) = \{q_1, \ldots, q_i\}$ for all x in Σ^*.

We prove by induction on $|x|$, the 'if' part, i.e.

$$\delta'(q'_0, x) = [q_1, q_2, \ldots, q_i] \tag{3.5}$$

if $\delta(q_0, x) = \{q_1, \ldots, q_i\}$.

When $|x| = 0$, $\delta(q_0, \Lambda) = \{q_0\}$, and by definition of δ', $\delta'(q_0', \Lambda) = q_0' = [q_0]$. So, (3.5) is true for x with $|x| = 0$. Thus there is basis for induction.

Assume that (3.5) is true for all strings y with $|y| \leq m$. Let x be a string of length $m + 1$. We can write x as ya, where $|y| = m$ and $a \in \Sigma$. Let $\delta(q_0, y) = \{p_1, \ldots, p_j\}$ and $\delta(q_0, ya) = \{r_1, r_2, \ldots, r_k\}$. As $|y| \leq m$, by induction hypothesis we have

$$\delta'(q_0', y) = [p_1, \ldots, p_j] \tag{3.6}$$

Also,

$$\{r_1, r_2, \ldots, r_k\} = \delta(q_0, ya) = \delta(\delta(q_0, y), a) = \delta(\{p_1, \ldots, p_j\}, a)$$

By definition of δ',

$$\delta'([p_1, \ldots, p_j], a) = [r_1, \ldots, r_k] \tag{3.7}$$

Hence,

$$\begin{aligned} \delta'(q_0', ya) &= \delta'(\delta'(q_0', y), a) = \delta'([p_1, \ldots, p_j], a) && \text{by (3.6)} \\ &= [r_1, \ldots, r_k] && \text{by (3.7)} \end{aligned}$$

Thus we have proved (3.5) for $x = ya$.

By induction, (3.5) is true for all strings x. The other part (i.e. the 'only if' part), can be proved similarly, and so (3.4) is established.

Now, $x \in T(M)$ if and only if $\delta(q, x)$ contains a state of F. By (3.4), $\delta(q_0, x)$ contains a state of F if and only if $\delta'(q_0', x)$ is in F'. Hence, $x \in T(M)$ if and only if $x \in T(M')$. This proves that DFA M' accepts L. ∎

Note: In the construction of a deterministic finite automaton M_1 equivalent to a given nondeterministic automaton M, the only difficult part is the construction of δ' for M_1. By definition,

$$\delta'([q_1 \ldots q_k], a) = \bigcup_{i=1}^{k} \delta(q_i, a)$$

So we have to apply δ to (q_i, a) for each $i = 1, 2, \ldots, k$ and take their union to get $\delta'([q_1 \ldots q_k], a)$.

When δ for M is given in terms of a state table, the construction is simpler. $\delta(q_i, a)$ is given by the row corresponding to q_i and the column corresponding to a. To construct $\delta'([q_1 \ldots q_k], a)$, consider the states appearing in the rows corresponding to $q_1, \ldots, q_k$, and the column corresponding to a. These states constitute $\delta'([q_1 \ldots q_k], a)$.

Note: We write δ' as δ itself when there is no ambiguity. We also mark the initial state with $\rightarrow$ and the final state with a circle in the state table.

EXAMPLE 3.6

Construct a deterministic automaton equivalent to

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

where δ is defined by its state table (see Table 3.2).

TABLE 3.2 State Table for Example 3.6

State/Σ	0	1
$\rightarrow (q_0)$	q_0	q_1
q_1	q_1	q_0, q_1

Solution

For the deterministic automaton M_1,

(i) the states are subsets of $\{q_0, q_1\}$, i.e. $\emptyset$, $[q_0]$, $[q_0, q_1]$, $[q_1]$;
(ii) $[q_0]$ is the initial state;
(iii) $[q_0]$ and $[q_0, q_1]$ are the final states as these are the only states containing q_0; and
(iv) δ is defined by the state table given by Table 3.3.

TABLE 3.3 State Table of M_1 for Example 3.6

State/Σ	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$[q_0]$	$[q_0]$	$[q_1]$
$[q_1]$	$[q_1]$	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

The states q_0 and q_1 appear in the rows corresponding to q_0 and q_1 and the column corresponding to 0. So, $\delta([q_0, q_1], 0) = [q_0, q_1]$.

When M has n states, the corresponding finite automaton has 2^n states. However, we need not construct δ for all these 2^n states, but only for those states that are reachable from $[q_0]$. This is because our interest is only in constructing M_1 accepting $T(M)$. So, we start the construction of δ for $[q_0]$. We continue by considering only the states appearing earlier under the input columns and constructing δ for such states. We halt when no more new states appear under the input columns.

EXAMPLE 3.7

Find a deterministic acceptor equivalent to

$$M = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$$

where δ is as given by Table 3.4.

TABLE 3.4 State Table for Example 3.7

State/Σ	a	b
$\rightarrow q_0$	q_0, q_1	q_2
q_1	q_0	q_1
(q_2)		q_0, q_1

Solution

The deterministic automaton M_1 equivalent to M is defined as follows:

$$M_1 = (2^Q, \{a, b\}, \delta, [q_0], F')$$

where

$$F = \{[q_2], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$$

We start the construction by considering $[q_0]$ first. We get $[q_2]$ and $[q_0, q_1]$. Then we construct δ for $[q_2]$ and $[q_0, q_1]$. $[q_1, q_2]$ is a new state appearing under the input columns. After constructing δ for $[q_1, q_2]$, we do not get any new states and so we terminate the construction of δ. The state table is given by Table 3.5.

TABLE 3.5 State Table of M_1 for Example 3.7

State/Σ	*a*	*b*
$[q_0]$	$[q_0, q_1]$	$[q_2]$
$[q_2]$	$\emptyset$	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_0]$	$[q_0, q_1]$

EXAMPLE 3.8

Construct a deterministic finite automaton equivalent to

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

where δ is given by Table 3.6.

TABLE 3.6 State Table for Example 3.8

State/Σ	*a*	*b*
$\rightarrow q_0$	q_0, q_1	q_0
q_1	q_2	q_1
q_2	q_3	q_3
(q_3)		q_2

Solution

Let $Q = \{q_0, q_1, q_2, q_3\}$. Then the deterministic automaton M_1 equivalent to M is given by

$$M_1 = (2^Q, \{a, b\}, \delta, [q_0], F)$$

where F consists of:

$$[q_3], [q_0, q_3], [q_1, q_3], [q_2, q_3], [q_0, q_1, q_3], [q_0, q_2, q_3], [q_1, q_2, q_3]$$

and

$$[q_0, q_1, q_2, q_3]$$

and where δ is defined by the state table given by Table 3.7.

TABLE 3.7 State Table of M_1 for Example 3.8

State/Σ	a	b
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_1]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_3]$
$[q_0, q_1, q_3]$	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_1, q_2, q_3]$

3.8 MEALY AND MOORE MODELS

3.8.1 FINITE AUTOMATA WITH OUTPUTS

The finite automata which we considered in the earlier sections have binary output, i.e. either they accept the string or they do not accept the string. This acceptability was decided on the basis of reachability of the final state by the initial state. Now, we remove this restriction and consider the model where the outputs can be chosen from some other alphabet. The value of the output function $Z(t)$ in the most general case is a function of the present state $q(t)$ and the present input $x(t)$, i.e.

$$Z(t) = \lambda(q(t), x(t))$$

where λ is called the output function. This generalized model is usually called the *Mealy machine*. If the output function $Z(t)$ depends only on the present state and is independent of the current input, the output function may be written as

$$Z(t) = \lambda(q(t))$$

This restricted model is called the *Moore machine*. It is more convenient to use Moore machine in automata theory. We now give the most general definitions of these machines.

Definition 3.8 A Moore machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

(i) Q is a finite set of states;
(ii) Σ is the input alphabet;
(iii) Δ is the output alphabet;
(iv) δ is the transition function $\Sigma \times Q$ into Q;
(v) λ is the output function mapping Q into Δ; and
(vi) q_0 is the initial state.

Definition 3.9 A Mealy machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where all the symbols except λ have the same meaning as in the Moore machine. λ is the output function mapping $\Sigma \times Q$ into Δ.

For example, Table 3.8 describes a Moore machine. The initial state q_0 is marked with an arrow. The table defines δ and λ.

TABLE 3.8 A Moore Machine

Present state	*Next state* δ		*Output*
	$a = 0$	$a = 1$	λ
$\rightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

For the input string 0111, the transition of states is given by $q_0 \rightarrow q_3 \rightarrow q_0 \rightarrow q_1 \rightarrow q_2$. The output string is 00010. For the input string Λ, the output is $\lambda(q_0) = 0$.

Transition Table 3.9 describes a Mealy machine.

TABLE 3.9 A Mealy Machine

Present state	*Next state*			
	$a = 0$		$a = 1$	
	state	*output*	*state*	*output*
$\rightarrow q_1$	q_3	0	q_2	0
q_2	q_4	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0

Note: For the input string 0011, the transition of states is given by $q_1 \rightarrow q_3 \rightarrow q_2 \rightarrow q_4 \rightarrow q_3$, and the output string is 0100. In the case of a Mealy machine, we get an output only on the application of an input symbol. So for the input string Λ, the output is only Λ. It may be observed that in the case of a Moore machine, we get $\lambda(q_0)$ for the input string Λ.

Remark A finite automaton can be converted into a Moore machine by introducing $\Delta = \{0, 1\}$ and defining $\lambda(q) = 1$ if $q \in F$ and $\lambda(q) = 0$ if $q \notin F$.

For a Moore machine if the input string is of length n, the output string is of length $n + 1$. The first output is $\lambda(q_0)$ for all output strings. In the case of a Mealy machine if the input string is of length n, the output string is also of the same length n.

3.8.2 Procedure for Transforming a Mealy Machine into a Moore Machine

We develop procedures for transforming a Mealy machine into a Moore machine and vice versa so that for a given input string the output strings are the same (except for the first symbol) in both the machines.

EXAMPLE 3.9

Consider the Mealy machine described by the transition table given by Table 3.10. Construct a Moore machine which is equivalent to the Mealy machine.

TABLE 3.10 Mealy Machine of Example 3.9

Present state	Next state			
	Input $a = 0$		Input $a = 1$	
	state	output	state	output
$\rightarrow q_1$	q_3	0	q_2	0
q_2	q_1	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0

Solution

At the first stage we develop the procedure so that both machines accept exactly the same set of input sequences. We look into the next state column for any state, say q_i, and determine the number of different outputs associated with q_i in that column.

We split q_i into several different states, the number of such states being equal to the number of different outputs associated with q_i. For example, in this problem, q_1 is associated with one output 1 and q_2 is associated with two different outputs 0 and 1. Similarly, q_3 and q_4 are associated with the outputs 0 and 0, 1, respectively. So, we split q_2 into q_{20} and q_{21}. Similarly, q_4 is split into q_{40} and q_{41}. Now Table 3.10 can be reconstructed for the new states as given by Table 3.11.

TABLE 3.11 State Table for Example 3.9

Present state	Next state			
	Input $a = 0$		Input $a = 1$	
	state	output	state	output
$\rightarrow q_1$	q_3	0	q_{20}	0
q_{20}	q_1	1	q_{40}	0
q_{21}	q_1	1	q_{40}	0
q_3	q_{21}	1	q_1	1
q_{40}	q_{41}	1	q_3	0
q_{41}	q_{41}	1	q_3	0

The pair of states and outputs in the next state column can be rearranged as given by Table 3.12.

TABLE 3.12 Revised State Table for Example 3.9

Present state	*Next state*		*Output*
	$a = 0$	$a = 1$	
$\rightarrow q_1$	q_3	q_{20}	1
q_{20}	q_1	q_{40}	0
q_{21}	q_1	q_{40}	1
q_3	q_{21}	q_1	0
q_{40}	q_{41}	q_3	0
q_{41}	q_{41}	q_3	1

Table 3.12 gives the Moore machine. Here we observe that the initial state q_1 is associated with output 1. This means that with input Λ we get an output of 1, if the machine starts at state q_1. Thus this Moore machine accepts a zero-length sequence (null sequence) which is not accepted by the Mealy machine. To overcome this situation, either we must neglect the response of a Moore machine to input Λ, or we must add a new starting state q_0, whose state transitions are identical with those of q_1 but whose output is 0. So Table 3.12 is transformed to Table 3.13.

TABLE 3.13 Moore Machine of Example 3.9

Present state	*Next state*		*Output*
	$a = 0$	$a = 1$	
$\rightarrow q_0$	q_3	q_{20}	0
q_1	q_3	q_{20}	1
q_{20}	q_1	q_{40}	0
q_{21}	q_1	q_{40}	1
q_3	q_{21}	q_1	0
q_{40}	q_{41}	q_3	0
q_{41}	q_{41}	q_3	1

From the foregoing procedure it is clear that if we have an m-output, n-state Mealy machine, the corresponding m-output Moore machine has no more than $mn + 1$ states.

3.8.3 Procedure for Transforming a Moore Machine into a Mealy Machine

We modify the acceptability of input string by a Moore machine by neglecting the response of the Moore machine to input Λ. We thus define that Mealy Machine M and Moore Machine M' are equivalent if for all input strings w, $bZ_M(w) = Z_{M'}(w)$, where b is the output of the Moore machine for its initial state. We give the following result: Let $M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Moore machine. Then the following procedure may be adopted to construct an equivalent Mealy machine M_2.

Construction

(i) We have to define the output function λ' for the Mealy machine as a function of the present state and the input symbol. We define λ' by

$$\lambda'(q, a) = \lambda(\delta(q, a)) \quad \text{for all states } q \text{ and input symbols } a.$$

(ii) The transition function is the same as that of the given Moore machine.

EXAMPLE 3.10

Construct a Mealy Machine which is equivalent to the Moore machine given by Table 3.14.

TABLE 3.14 Moore Machine of Example 3.10

Present state	Next state		Output
	$a = 0$	$a = 1$	
$\rightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

Solution

We must follow the reverse procedure of converting a Mealy machine into a Moore machine. In the case of the Moore machine, for every input symbol we form the pair consisting of the next state and the corresponding output and reconstruct the table for the Mealy Machine. For example, the states q_3 and q_1 in the next state column should be associated with outputs 0 and 1, respectively. The transition table for the Mealy machine is given by Table 3.15.

TABLE 3.15 Mealy Machine of Example 3.10

Present state	Next state			
	$a = 0$		$a = 1$	
	state	output	state	output
$\rightarrow q_0$	q_3	0	q_1	1
q_1	q_1	1	q_2	0
q_2	q_2	0	q_3	0
q_3	q_3	0	q_0	0

Note: We can reduce the number of states in any model by considering states with identical transitions. If two states have identical transitions (i.e. the rows corresponding to these two states are identical), then we can delete one of them.

EXAMPLE 3.11

Consider the Moore machine described by the transition table given by Table 3.16. Construct the corresponding Mealy machine.

TABLE 3.16 Moore Machine of Example 3.11

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

Solution

We construct the transition table as in Table 3.17 by associating the output with the transitions.

In Table 3.17, the rows corresponding to q_2 and q_3 are identical. So, we can delete one of the two states, i.e. q_2 or q_3. We delete q_3. Table 3.18 gives the reconstructed table.

TABLE 3.17 Transition Table for Example 3.11

Present state	*Next state*			
	$a = 0$		$a = 1$	
	state	*output*	*state*	*output*
$\rightarrow q_1$	q_1	0	q_2	0
q_2	q_1	0	q_3	1
q_3	q_1	0	q_3	1

TABLE 3.18 Mealy Machine of Example 3.11

Present state	*Next state*			
	$a = 0$		$a = 1$	
	state	*output*	*state*	*output*
$\rightarrow q_1$	q_1	0	q_2	0
q_2	q_1	0	q_2	1

In Table 3.18, we have deleted the q_3-row and replaced q_3 by q_2 in the other rows.

EXAMPLE 3.12

Consider a Mealy machine represented by Fig. 3.10. Construct a Moore machine equivalent to this Mealy machine.

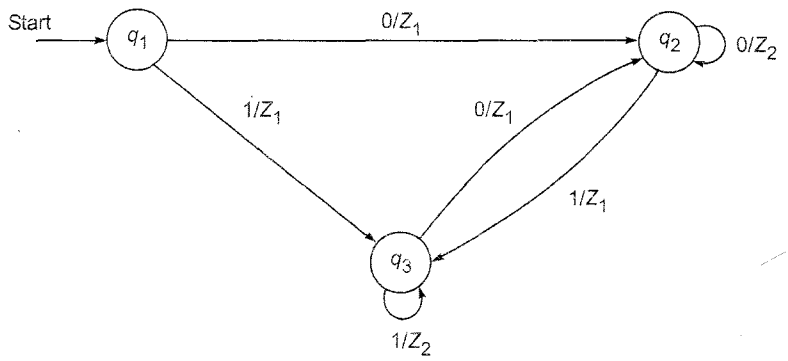


Fig. 3.10 Mealy machine of Example 3.12.

Solution

Let us convert the transition diagram into the transition Table 3.19. For the given problem: q_1 is not associated with any output; q_2 is associated with two different outputs Z_1 and Z_2; q_3 is associated with two different outputs Z_1 and Z_2. Thus we must split q_2 into q_{21} and q_{22} with outputs Z_1 and Z_2, respectively and q_3 into q_{31} and q_{32} with outputs Z_1 and Z_2, respectively. Table 3.19 may be reconstructed as Table 3.20.

TABLE 3.19 Transition Table for Example 3.12

Present state	*Next state*			
	$a = 0$		$a = 1$	
	state	*output*	*state*	*output*
$\rightarrow q_1$	q_2	Z_1	q_3	Z_1
q_2	q_2	Z_2	q_3	Z_1
q_3	q_2	Z_1	q_3	Z_2

TABLE 3.20 Transition Table of Moore Machine for Example 3.12

Present state	*Next state*		*Output*
	$a = 0$	$a = 1$	
$\rightarrow q_1$	q_{21}	q_{31}	
q_{21}	q_{22}	q_{31}	Z_1
q_{22}	q_{22}	q_{31}	Z_2
q_{31}	q_{21}	q_{32}	Z_1
q_{32}	q_{21}	q_{32}	Z_2

Figure 3.11 gives the transition diagram of the required Moore machine.

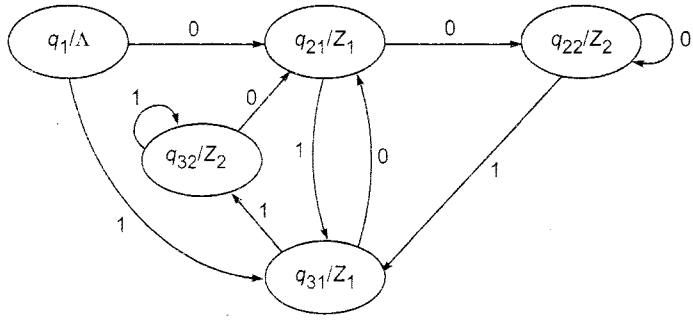


Fig. 3.11 Moore machine of Example 3.12.

3.9 MINIMIZATION OF FINITE AUTOMATA

In this section we construct an automaton with the minimum number of states equivalent to a given automaton M.

As our interest lies only in strings accepted by M, what really matters is whether a state is a final state or not. We define some relations in Q.

Definition 3.10 Two states q_1 and q_2 are equivalent (denoted by $q_1 \equiv q_2$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states, or both of them are nonfinal states for all $x \in \Sigma^*$.

As it is difficult to construct $\delta(q_1, x)$ and $\delta(q_2, x)$ for all $x \in \Sigma^*$ (there are an infinite number of strings in Σ^*), we give one more definition.

Definition 3.11 Two states q_1 and q_2 are k-equivalent ($k \geq 0$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both nonfinal states for all strings x of length k or less. In particular, any two final states are 0-equivalent and any two nonfinal states are also 0-equivalent.

We mention some of the properties of these relations.

Property 1 The relations we have defined, i.e. equivalence and k-equivalence, are equivalence relations, i.e. they are reflexive, symmetric and transitive.

Property 2 By Theorem 2.1, these induce partitions of Q. These partitions can be denoted by π and π_k, respectively. The elements of π_k are k-equivalence classes.

Property 3 If q_1 and q_2 are k-equivalent for all $k \geq 0$, then they are equivalent.

Property 4 If q_1 and q_2 are $(k + 1)$-equivalent, then they are k-equivalent.

Property 5 $\pi_n = \pi_{n+1}$ for some n. (π_n denotes the set of equivalence classes under n-equivalence.)

The following result is the key to the construction of minimum state automaton.

RESULT Two states q_1 and q_2 are $(k + 1)$-equivalent if (i) they are k-equivalent; (ii) $\delta(q_1, a)$ and $\delta(q_2, a)$ are also k-equivalent for every $a \in \Sigma$.

Proof We prove the result by contradiction. Suppose q_1 and q_2 are not $(k + 1)$-equivalent. Then there exists a string $w = aw_1$ of length $k + 1$ such that $\delta(q_1, aw_1)$ is a final state and $\delta(q_2, aw_1)$ is not a final state (or vice versa; the proof is similar). So $\delta(\delta(q_1, a), w_1)$ is a final state and $\delta(\delta(q_2, a), w_1)$ is not a final state. As w_1 is a string of length k, $\delta(q_1, a)$ and $\delta(q_2, a)$ are not k-equivalent. This is a contradiction, and hence the result is proved. ∎

Using the previous result we can construct the $(k + 1)$-equivalence classes once the k-equivalence classes are known.

3.9.1 Construction of Minimum Automaton

Step 1 (Construction of π_0). By definition of 0-equivalence, $\pi_0 = \{ Q_1^0, Q_2^0 \}$ where Q_1^0 is the set of all final states and $Q_2^0 = Q - Q_1^0$.

Step 2 (Construction of π_{k+1} from π_k). Let Q_i^k be any subset in π_k. If q_1 and q_2 are in Q_i^k, they are $(k + 1)$-equivalent provided $\delta(q_1, a)$ and $\delta(q_2, a)$ are k-equivalent. Find out whether $\delta(q_1, a)$ and $\delta(q_2, a)$ are in the same equivalence class in π_k for every $a \in \Sigma$. If so, q_1 and q_2 are $(k + 1)$-equivalent. In this way, Q_i^k is further divided into $(k + 1)$-equivalence classes. Repeat this for every Q_i^k in π_k to get all the elements of π_{k+1}.

Step 3 Construct π_n for $n = 1, 2, \ldots$ until $\pi_n = \pi_{n+1}$.

Step 4 (Construction of minimum automaton). For the required minimum state automaton, the states are the equivalence classes obtained in step 3, i.e. the elements of π_n. The state table is obtained by replacing a state q by the corresponding equivalence class $[q]$.

Remark In the above construction, the crucial part is the construction of equivalence classes; for, after getting the equivalence classes, the table for minimum automaton is obtained by replacing states by the corresponding equivalence classes. The number of equivalence classes is less than or equal to $|Q|$. Consider an equivalence class $[q_1] = \{q_1, q_2, \ldots, q_k\}$. If q_1 is reached while processing $w_1w_2 \in T(M)$ with $\delta(q_0, w_1) = q_1$, then $\delta(q_i, w_2) \in F$. So, $\delta(q_i, w_2) \in F$ for $i = 2, \ldots, k$. Thus we see that q_i, $i = 2, \ldots, k$ is reached on processing some $w \in T(M)$ iff q_1 is reached on processing w, i.e. q_1 of $[q_1]$ can play the role of $q_2, \ldots, q_k$. The above argument explains why we replace a state by the corresponding equivalence class.

Note: The construction of π_0, π_1, π_2, etc. is easy when the transition table is given. $\pi_0 = \{Q_1^0, Q_2^0\}$, where $Q_1^0 = F$ and $Q_2^0 = Q - F$. The subsets in π_1 are obtained by further partitioning the subsets of π_0. If $q_1, q_2 \in Q_1^0$, consider the states in each a-column, where $a \in \Sigma$ corresponding to q_1 and q_2. If they are in the same subset of π_0, q_1 and q_2 are 1-equivalent. If the states under some a-column are in different subsets of π_0, then q_1 and q_2 are not 1-equivalent. In general, $(k + 1)$-equivalent states are obtained by applying the above method for q_1 and q_2 in Q_i^k.

EXAMPLE 3.13

Construct a minimum state automaton equivalent to the finite automaton described by Fig. 3.12.

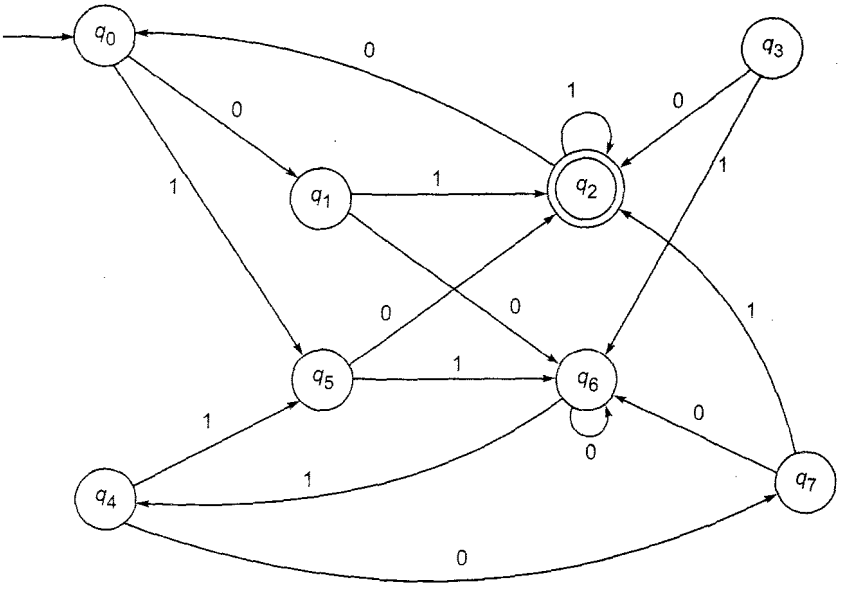


Fig. 3.12 Finite automaton of Example 3.13.

Solution

It will be easier if we construct the transition table as shown in Table 3.21.

TABLE 3.21 Transition Table for Example 3.13

State/Σ	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
(q_2)	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

By applying step 1, we get

$$Q_1^0 = F = \{q_2\}, \qquad Q_2^0 = Q - Q_1^0$$

So,

$$\pi_0 = \{\{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}\}$$

The $\{q_2\}$ in π_0 cannot be further partitioned. So, $Q'_1 = \{q_2\}$. Consider q_0 and $q_1 \in Q_2^0$. The entries under the 0-column corresponding to q_0 and q_1 are q_1 and q_6; they lie in Q_2^0. The entries under the 1-column are q_5 and q_2. $q_2 \in Q_1^0$ and $q_5 \in Q_2^0$. Therefore, q_0 and q_1 are not 1-equivalent. Similarly, q_0 is not 1-equivalent to q_3, q_5 and q_7.

Now, consider q_0 and q_4. The entries under the 0-column are q_1 and q_7. Both are in Q_2^0. The entries under the 1-column are q_5, q_5. So q_4 and q_0 are 1-equivalent. Similarly, q_0 is 1-equivalent to q_6. $\{q_0, q_4, q_6\}$ is a subset in π_1. So, $Q'_2 = \{q_0, q_4, q_6\}$.

Repeat the construction by considering q_1 and any one of the states q_3, q_5, q_7. Now, q_1 is not 1-equivalent to q_3 or q_5 but 1-equivalent to q_7. Hence, $Q'_3 = \{q_1, q_7\}$. The elements left over in Q_2^0 are q_3 and q_5. By considering the entries under the 0-column and the 1-column, we see that q_3 and q_5 are 1-equivalent. So $Q'_4 = \{q_3, q_5\}$. Therefore,

$$\pi_1 = \{\{q_2\}, \{q_0, q_4, q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$$

The $\{q_2\}$ is also in π_2 as it cannot be further partitioned. Now, the entries under the 0-column corresponding to q_0 and q_4 are q_1 and q_7, and these lie in the same equivalence class in π_1. The entries under the 1-column are q_5, q_5. So q_0 and q_4 are 2-equivalent. But q_0 and q_6 are not 2-equivalent. Hence, $\{q_0, q_4, q_6\}$ is partitioned into $\{q_0, q_4\}$ and $\{q_6\}$. q_1 and q_7 are 2-equivalent. q_3 and q_5 are also 2-equivalent. Thus, $\pi_2 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$. q_0 and q_4 are 3-equivalent. The q_1 and q_7 are 3-equivalent. Also, q_3 and q_5 are 3-equivalent. Therefore,

$$\pi_3 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$$

As $\pi_2 = \pi_3$, π_2 gives us the equivalence classes, the minimum state automaton is

$$M' = (Q', \{0, 1\}, \delta', q'_0, F')$$

where

$$Q' = \{[q_2], [q_0, q_4], [q_6], [q_1, q_7], [q_3, q_5]\}$$

$$q'_0 = [q_0, q_4], \qquad F' = [q_2]$$

and δ' is defined by Table 3.22.

TABLE 3.22 Transition Table of Minimum State Automaton for Example 3.13

State/Σ	0	1
$[q_0, q_4]$	$[q_1, q_7]$	$[q_3, q_5]$
$[q_1, q_7]$	$[q_6]$	$[q_2]$
$[q_2]$	$[q_0, q_4]$	$[q_2]$
$[q_3, q_5]$	$[q_2]$	$[q_6]$
$[q_6]$	$[q_6]$	$[q_0, q_4]$

Note: The transition diagram for the minimum state automaton is described by Fig. 3.13. The states q_0 and q_4 are identified and treated as one state. (So also are q_1, q_7 and q_3, q_5.) But the transitions in both the diagrams (i.e. Figs. 3.12 and 3.13) are the same. If there is an arrow from q_i to q_j with label a, then there is an arrow from $[q_i]$ to $[q_j]$ with the same label in the diagram for minimum state automaton. Symbolically, if $\delta(q_i, a) = q_j$, then $\delta'([q_i], a) = [q_j]$.

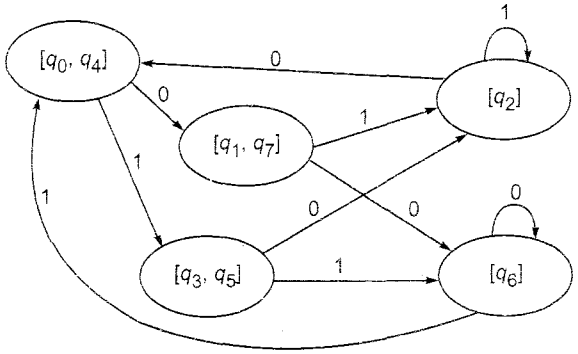


Fig. 3.13 Minimum state automaton of Example 3.13.

EXAMPLE 3.14

Construct the minimum state automaton equivalent to the transition diagram given by Fig. 3.14.

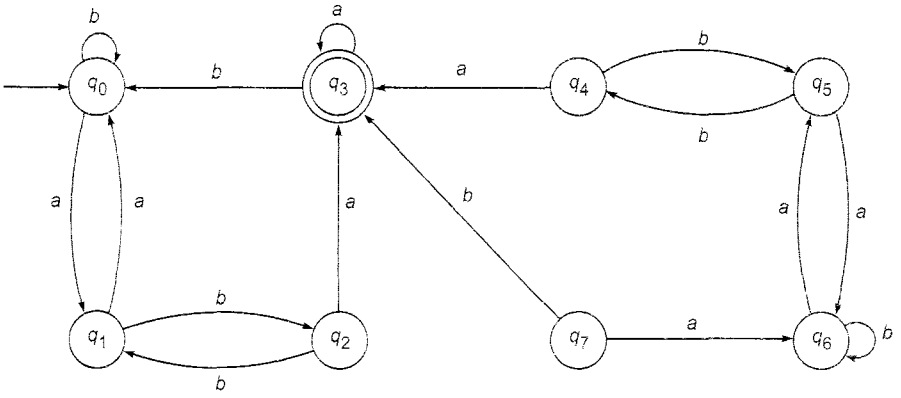


Fig. 3.14 Finite automaton of Example 3.14.

Solution

We construct the transition table as given by Table 3.23.

TABLE 3.23 Transition Table for Example 3.14

State/Σ	a	b
→ q_0	q_1	q_0
q_1	q_0	q_2
q_2	q_3	q_1
(q_3)	q_3	q_0
q_4	q_3	q_5
q_5	q_6	q_4
q_6	q_5	q_6
q_7	q_6	q_3

Since there is only one final state q_3, $Q_1^0 = \{q_3\}$, $Q_2^0 = Q - Q_1^0$. Hence, $\pi_0 = \{\{q_3\}, \{q_0, q_1, q_2, q_4, q_5, q_6, q_7\}\}$. As $\{q_3\}$ cannot be partitioned further, $Q'_1 = \{q_3\}$. Now q_0 is 1-equivalent to q_1, q_5, q_6, but not to q_2, q_4, q_7, and so $Q'_2 = \{q_0, q_1, q_5, q_6\}$. q_2 is 1-equivalent to q_4. Hence, $Q'_3 = \{q_2, q_4\}$. The only element remaining in Q_2^0 is q_7. Therefore, $Q'_4 = \{q_7\}$. Thus,

$$\pi_1 = \{\{q_3\}, \{q_0, q_1, q_5, q_6\}, \{q_2, q_4\}, \{q_7\}\}$$

$$Q_1^2 = \{q_3\}$$

q_0 is 2-equivalent to q_6 but not to q_1 or q_5. So,

$$Q_2^2 = \{q_0, q_6\}$$

As q_1 is 2-equivalent to q_5,

$$Q_3^2 = \{q_1, q_5\}$$

As q_2 is 2-equivalent to q_4,

$$Q_4^2 = \{q_2, q_4\}, \qquad Q_5^2 = \{q_7\}$$

Thus,

$$\pi_2 = \{\{q_3\}, \{q_0, q_6\}, \{q_1, q_5\}, \{q_2, q_4\}, \{q_7\}\}$$

$$Q_1^3 = \{q_3\}$$

As q_0 is 3-equivalent to q_6,

$$Q_2^3 = \{q_0, q_6\}$$

As q_1 is 3-equivalent to q_5,

$$Q_3^3 = \{q_1, q_5\}$$

As q_2 is 3-equivalent to q_4,

$$Q_4^3 = \{q_2, q_4\}, \quad Q_5^3 = \{q_7\}$$

Therefore,

$$\pi_3 = \{\{q_3\}, \{q_0, q_6\}, \{q_1, q_5\}, \{q_2, q_4\}, \{q_7\}\}$$

As $\pi_3 = \pi_2$, π_2 gives us the equivalence classes, the minimum state automaton is

$$M' = (Q', \{a, b\}, \delta', q'_0, F')$$

where

$$Q' = \{[q_3],\ [q_0,\ q_6],\ [q_1,\ q_5],\ [q_2,\ q_4],\ [q_7]\}$$

$$q_0' = [q_0,\ q_6], \qquad F' = [q_3]$$

and δ' is defined by Table 3.24.

TABLE 3.24 Transition Table of Minimum State Automaton for Example 3.14

State/Σ	*a*	*b*
$[q_0, q_6]$	$[q_1, q_5]$	$[q_0, q_6]$
$[q_1, q_5]$	$[q_0, q_6]$	$[q_2, q_4]$
$[q_2, q_4]$	$[q_3]$	$[q_1, q_5]$
$[q_3]$	$[q_3]$	$[q_0, q_6]$
$[q_7]$	$[q_0, q_6]$	$[q_3]$

Note: The transition diagram for M' is given by Fig. 3.15.

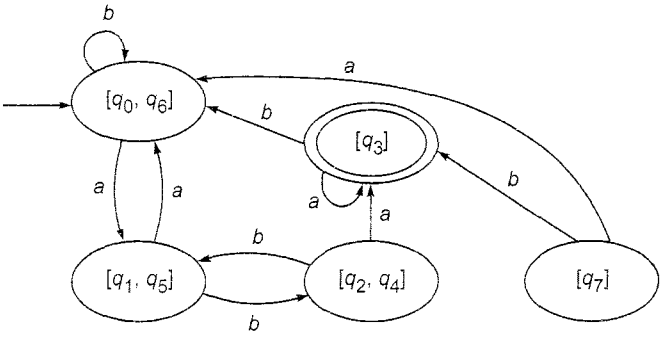


Fig. 3.15 Minimum state automaton of Example 3.14.

3.10 SUPPLEMENTARY EXAMPLES

EXAMPLE 3.15

Construct a DFA equivalent to the NDFA M whose transition diagram is given by Fig. 3.16.

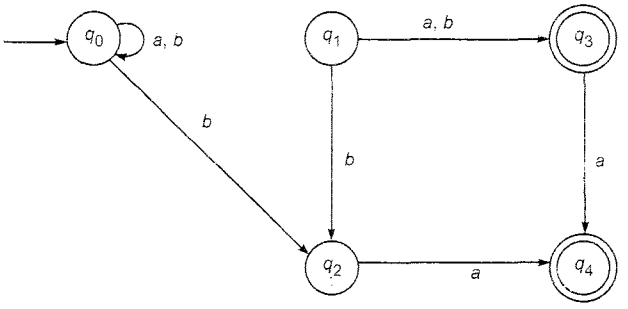


Fig. 3.16 NDFA of Example 3.15.

Solution

The transition table of M is given by Table 3.25.

TABLE 3.25 Transition Table for Example 3.15

State	a	b
$\rightarrow q_0$	q_0	q_0, q_2
q_1	q_3	q_2, q_3
q_2	q_4	—
(q_3)	q_4	—
(q_4)	—	—

For the equivalent DFA:

(i) The states are subsets of $Q = \{q_0, q_1, q_2, q_3, q_4\}$.

(ii) $[q_0]$ is the initial state.

(iii) The subsets of Q containing q_3 or q_4 are the final states.

(iv) δ is defined by Table 3.26. We start from $[q_0]$ and construct δ, only for those states reachable from $[q_0]$ (as in Example 3.8).

TABLE 3.26 Transition Table of DFA for Example 3.15

State	a	b
$[q_0]$	$[q_0]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_4]$	$[q_0, q_2]$
$[q_0, q_4]$	$[q_0]$	$[q_0, q_2]$

EXAMPLE 3.16

Construct a DFA equivalent to an NDFA whose transition table is defined by Table 3.27.

TABLE 3.27 Transition Table of NDFA for Example 3.16

State	a	b
q_0	q_1, q_3	q_2, q_3
q_1	q_1	q_3
q_2	q_3	q_2
(q_3)	—	—

Solution

Let M be the DFA defined by

$$M = (2^{\{q_0, q_1, q_2, q_3\}}, \{a, b\}, \delta, [q_0], F)$$

where F is the set of all subsets of $\{q_0, q_1, q_2, q_3\}$ containing q_3. δ is defined by Table 3.28.

TABLE 3.28 Transition Table of DFA for Example 3.16

State	*a*	*b*
$[q_0]$	$[q_1, q_3]$	$[q_2, q_3]$
$[q_1, q_3]$	$[q_1]$	$[q_3]$
$[q_1]$	$[q_1]$	$[q_3]$
$[q_3]$	$\emptyset$	$\emptyset$
$[q_2, q_3]$	$[q_3]$	$[q_2]$
$[q_2]$	$[q_3]$	$[q_2]$
$\emptyset$	$\emptyset$	$\emptyset$

EXAMPLE 3.17

Construct a DFA accepting all strings w over $\{0, 1\}$ such that the number of 1's in w is 3 mod 4.

Solution

Let M be the required NDFA. As the condition on strings of $T(M)$ does not at all involve 0, we can assume that M does not change state on input 0. If 1 appears in w $(4k + 3)$ times, M can come back to the initial state, after reading 4 1's and to a final state after reading 3 1's.

The required DFA is given by Fig. 3.17.

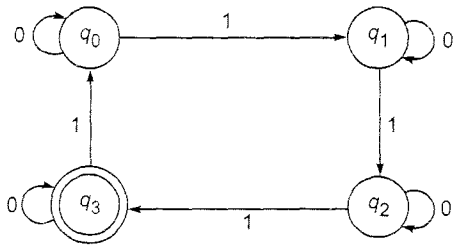


Fig. 3.17 DFA of Example 3.17.

EXAMPLE 3.18

Construct a DFA accepting all strings over $\{a, b\}$ ending in ab.

Solution

We require two transitions for accepting the string ab. If the symbol b is processed after aa or ba, then also we end in ab. So we can have states for

remembering *aa*, *ab*, *ba*, *bb*. The state corresponding to *ab* can be the final state in our DFA. Keeping these in mind we construct the required DFA. Its transition diagram is described by Fig. 3.18.

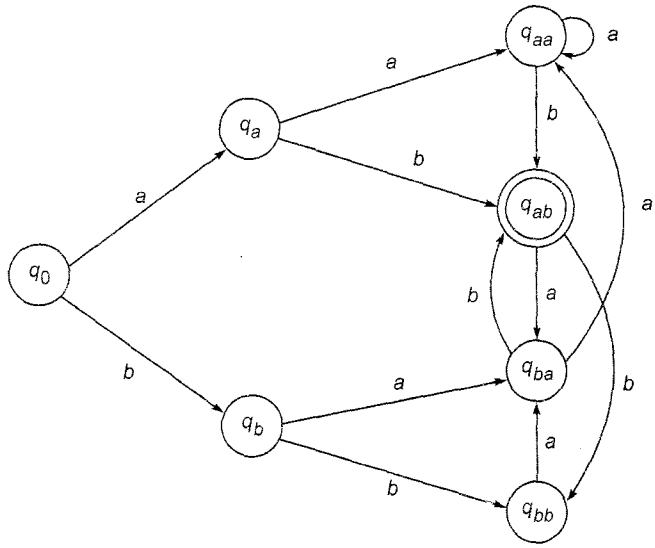


Fig. 3.18 DFA of Example 3.18.

EXAMPLE 3.19

Find $T(M)$ for the DFA M described by Fig. 3.19.

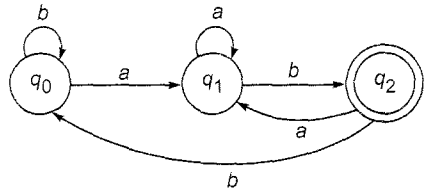


Fig. 3.19 DFA of Example 3.19.

Solution

$T(M) = \{w \in \{a, b\}^* \mid w$ ends in the substring $ab\}$

Note: If we apply the minimization algorithm to the DFA in Example 3.18, we get the DFA as in Example 3.19. (The student is advised to check.)

EXAMPLE 3.20

Construct a minimum state automaton equivalent to an automaton whose transition table is defined by Table 3.29.

TABLE 3.29 DFA of Example 3.20

State	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_3
q_2	q_3	q_4
q_3	q_1	q_5
q_4	q_4	q_2
(q_5)	q_6	q_6

Solution

$Q_1^0 = \{q_5\}$, $Q_2^0 = \{q_0, q_1, q_2, q_3, q_4\}$. So, $\pi_0 = \{\{q_5\}, \{q_0, q_1, q_2, q_3, q_4\}\}$
Q_1^0 cannot be partitioned further. So $\{q_5\} \in \pi_1$. Consider Q_2^0. q_0 is equivalent to q_1, q_2 and q_4. But q_0 is not equivalent to q_3 since $\delta(q_0, b) = q_2$ and $\delta(q_3, b) = \{q_5\}$.
Hence,

$$Q_1^1 = \{q_5\}, \quad Q_2^1 = \{q_0, q_1, q_2, q_4\}, \quad Q_3^1 = \{q_3\}$$

Therefore,

$$\pi_1 = \{\{q_5\}, \{q_0, q_1, q_2, q_4\}, \{q_3\}\}$$

q_0 is 2-equivalent to q_4 but not 2-equivalent to q_1 or q_2.
Hence,

$$\{q_0, q_4\} \in \pi_2$$

q_1 and q_2 are not 2-equivalent.
Therefore,

$$\pi_2 = \{\{q_5\}, \{q_3\}, \{q_0, q_4\}, \{q_1\}, \{q_2\}\}$$

As q_0 is not 3-equivalent to q_4, $\{q_0, q_4\}$ is further partitioned into $\{q_1\}$ and $\{q_4\}$.
So,

$$\pi_3 = \{\{q_0\}, \{q_1\}, \{q_2\}, \{q_3\}, \{q_4\}, \{q_5\}\}$$

Hence the minimum state automaton M' is the same as the given M.

EXAMPLE 3.21

Construct a minimum state automaton equivalent to a DFA whose transition table is defined by Table 3.30.

TABLE 3.30 DFA of Example 3.21

State	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_4	q_3
q_2	q_4	q_3
(q_3)	q_5	q_6
(q_4)	q_7	q_6
q_5	q_3	q_6
q_6	q_6	q_6
q_7	q_4	q_6

Therefore,

$$L(G) = \{0^n1^n \mid n \geq 0\}$$

EXAMPLE 4.3

If $G = (\{S\}, \{a\}, \{S \rightarrow SS\}, S)$, find the language generated by G.

Solution

$L(G) = \emptyset$, since the only production $S \rightarrow SS$ in G has no terminal on the right-hand side.

EXAMPLE 4.4

Let $G = (\{S, C\}, \{a, b\}, P, S)$, where P consists of $S \rightarrow aCa$, $C \rightarrow aCa \mid b$. Find $L(G)$.

Solution

$S \Rightarrow aCa \Rightarrow aba$. So $aba \in L(G)$

$S \Rightarrow aCa$ (by application of $S \rightarrow aCa$)

$\overset{*}{\Rightarrow} a^nCa^n$ (by application of $C \rightarrow aCa$ $(n - 1)$ times)

$\Rightarrow a^nba^n$ (by application of $C \rightarrow b$)

Hence, $a^nba^n \in L(G)$, where $n \geq 1$. Therefore,

$$\{a^nba^n \mid n \geq 1\} \subseteq L(G)$$

As the only S-production is $S \rightarrow aCa$, this is the first production we have to apply in the derivation of any terminal string. If we apply $C \rightarrow b$, we get aba. Otherwise we have to apply only $C \rightarrow aCa$, either once or several times. So we get a^nCa^n with a single variable C. To get a terminal string we have to replace C by b, by applying $C \rightarrow b$. So any derivation is of the form

$$S \overset{*}{\Rightarrow} a^nba^n \text{ with } n \geq 1$$

Therefore,

$$L(G) \subseteq \{a^nba^n \mid n \geq 1\}$$

Thus,

$$L(G) = \{a^nba^n \mid n \geq 1\}$$

EXERCISE Construct a grammar G so that $L(G) = \{a^nba^m \mid n, m \geq 1\}$.

Remark By applying the convention regarding the notation of variables, terminals and the start symbol, it will be clear from the context whether a symbol denotes a variable or terminal. We can specify a grammar by its productions alone.

EXAMPLE 4.5

If G is $S \to aS \mid bS \mid a \mid b$, find $L(G)$.

Solution

We show that $L(G) = \{a, b\}^+$. As we have only two terminals a, b, $L(G) \subseteq \{a, b\}^*$. All productions are S-productions, and so Λ can be in $L(G)$ only when $S \to \Lambda$ is a production in the grammar G. Thus,

$$L(G) \subseteq \{a, b\}^* - \{\Lambda\} = \{a, b\}^+$$

To show $\{a, b\}^+ \subseteq L(G)$, consider any string $a_1a_2 \ldots a_n$, where each a_i is either a or b. The first production in the derivation of $a_1a_2 \ldots a_n$ is $S \to aS$ or $S \to bS$ according as $a_1 = a$ or $a_1 = b$. The subsequent productions are obtained in a similar way. The last production is $S \to a$ or $S \to b$ according as $a_n = a$ or $a_n = b$. So $a_1a_2 \ldots a_n \in L(G)$. Thus, we have $L(G) = \{a, b\}^+$.

EXERCISE If G is $S \to aS \mid a$, then show that $L(G) = \{a\}^+$.

Some of the following examples illustrate the method of constructing a grammar G generating a given subset of strings over Σ. The difficult part is the construction of productions. We try to define the given set by recursion and then develop productions generating the strings in the given subset of Σ^*.

EXAMPLE 4.6

Let L be the set of all palindromes over $\{a, b\}$. Construct a grammar G generating L.

Solution

For constructing a grammar G generating the set of all palindromes, we use the recursive definition (given in Section 2.4) to observe the following:

(i) Λ is a palindrome.
(ii) a, b are palindromes.
(iii) If x is a palindrome axa, then bxb are palindromes.

So we define P as the set consisting of:

(i) $S \to \Lambda$
(ii) $S \to a$ and $S \to b$
(iii) $S \to aSa$ and $S \to bSb$

Let $G = (\{S\}, \{a, b\}, P, S)$. Then

$$S \Rightarrow \Lambda, \qquad S \Rightarrow a, \qquad S \Rightarrow b$$

Therefore,

$$\Lambda, a, b \in L(G)$$

If x is a palindrome of even length, then $x = a_1a_2 \ldots a_m a_m \ldots a_1$, where each a_i is either a or b. Then $S \overset{*}{\Rightarrow} a_1a_2 \ldots a_m a_m a_{m-1} \ldots a_1$ by applying $S \to aSa$ or $S \to bSb$. Thus, $x \in L(G)$.

If x is a palindrome of odd length, then $x = a_1a_2 \ldots a_nca_n \ldots a_1$, where a_i's and c are either a or b. So $S \overset{*}{\Rightarrow} a_1 \ldots a_nSa_n \ldots a_1 \Rightarrow x$ by applying $S \to aSa$, $S \to bSb$ and finally, $S \to a$ or $S \to b$. Thus, $x \in L(G)$. This proves $L = L(G)$.

EXAMPLE 4.7

Construct a grammar generating $L = \{wcw^T \mid w \in \{a, b\}^*\}$.

Solution

Let $G = (\{S\}, \{a, b, c\}, P, S)$, where P is defined as $S \to aSa \mid bSb \mid c$. It is easy to see the idea behind the construction. Any string in L is generated by recursion as follows: (i) $c \in L$; (ii) if $x \in L$, then $wxw^T \in L$. So, as in the earlier example, we have the productions $S \to aSa \mid bSb \mid c$.

EXAMPLE 4.8

Find a grammar generating $L = \{a^nb^nc^i \mid n \geq 1, i \geq 0\}$.

Solution

$$L = L_1 \cup L_2$$

$$L_1 = \{a^nb^n \mid n \geq 1\}$$

$$L_2 = \{a^nb^nc^i \mid n \geq 1, i \geq 0\}$$

We construct L_1 by recursion and L_2 by concatenating the elements of L_1 and c^i, $i \geq 1$. We define P as the set of the following productions:

$$S \to A, \qquad A \to ab, \qquad A \to aAb, \qquad S \to Sc$$

Let $G = (\{S, A\}, \{a, b, c\}, P, S)$. For $n \geq 1$, $i \to 0$, we have

$$S \overset{*}{\Rightarrow} Sc^i \Rightarrow Ac^i \overset{*}{\Rightarrow} a^{n-1}Ab^{n-1}c^i \Rightarrow a^{n-1}abb^{n-1}c^i = a^nb^nc^i$$

Thus,

$$\{a^nb^nc^i \mid n \geq 1, i \geq 0\} \subseteq L(G)$$

To prove the reverse inclusion, we note that the only S-productions are $S \to Sc$ and $S \to A$. If we start with $S \to A$, we have to apply

$$A \Rightarrow a^{n-1}Ab^{n-1} \overset{*}{\Rightarrow} a^nb^n, \text{ and so } a^nb^nc^0 \in L(G)$$

If we start with $S \to Sc$, we have to apply $S \to Sc$ repeatedly to get Sc^i. But to get a terminal string, we have to apply $S \to A$. As $A \overset{*}{\Rightarrow} a^nb^n$, the resulting terminal string is $a^nb^nc^i$. Thus, we have shown that

$$L(G) \subseteq \{a^nb^nc^i \mid n \geq 1, i \geq 0\}$$

Therefore,

$$L(G) = \{a^nb^nc^i \mid n \geq 1, i \geq 0\}$$

EXAMPLE 4.9

Find a grammar generating $\{a^j b^n c^n \mid n \geq 1, j \geq 0\}$.

Solution

Let $G = (\{S, A\}, \{a, b, c\}, P, S)$, where P consists of $S \rightarrow aS$, $S \rightarrow A$, $A \rightarrow bAc \mid bc$. As in the previous example, we can prove that G is the required grammar.

EXAMPLE 4.10

Let $G = (\{S, A_1\}, \{0, 1, 2\}, P, S)$, where P consists of $S \rightarrow 0SA_1 2$, $S \rightarrow 012$, $2A_1 \rightarrow A_1 2$, $1A_1 \rightarrow 11$. Show that

$$L(G) = \{0^n 1^n 2^n \mid n \geq 1\}$$

Solution

As $S \rightarrow 012$ is a production, we have $S \Rightarrow 012$, i.e. $012 \in L(G)$.
Also,

$$\begin{aligned}
S &\overset{*}{\Rightarrow} 0^{n-1} S(A_1 2)^{n-1} && \text{by applying } S \rightarrow 0SA_1 2 \quad (n-1) \text{ times} \\
&\Rightarrow 0^n 12(A_1 2)^{n-1} && \text{by applying } S \rightarrow 012 \\
&\overset{*}{\Rightarrow} 0^n 1A_1^{n-1} 2^n && \text{by applying } 2A_1 \rightarrow A_1 2 \quad \text{several times} \\
&\overset{*}{\Rightarrow} 0^n 1^n 2^n && \text{by applying } 1A_1 \rightarrow 11 \quad (n-1) \text{ times}
\end{aligned}$$

Therefore,

$$0^n 1^n 2^n \in L(G) \qquad \text{for all } n \geq 1$$

To prove that $L(G) \subseteq \{0^n 1^n 2^n \mid n \geq 1\}$, we proceed as follows: If the first production that we apply is $S \rightarrow 012$, we get 012. Otherwise we have to apply $S \rightarrow 0SA_1 2$ once or several times to get $0^{n-1} S(A_1 2)^{n-1}$. To eliminate S, we have to apply $S \rightarrow 012$. Thus we arrive at a sentential form $0^n 12(A_1 2)^{n-1}$. To eliminate the variable A_1, we have to apply $2A_1 \rightarrow A_1 2$ or $1A_1 \rightarrow 11$. Now, $2A_1 \rightarrow A_1 2$ interchanges 2 and A_1. Only $1A_1 \rightarrow 11$ eliminates A_1. The sentential form we have obtained is $0^n 12A_1 2A_1 2 \ldots A_1 2$. If we use $1A_1 \rightarrow 11$ before taking all 2's to the right, we will get 12 in the middle of the string. The A_1's appearing subsequently cannot be eliminated. So we have to bring all 2's to the right by applying $2A_1 \rightarrow A_1 2$ several times. Then we can apply $1A_1 \rightarrow 11$ repeatedly and get $0^n 1^n 2^n$ (as derived in the first part of the proof). Thus,

$$L(G) \subseteq \{0^n 1^n 2^n \mid n \geq 1\}$$

This shows that

$$L(G) = \{0^n 1^n 2^n \mid n \geq 1\}$$

In the next example we construct a grammar generating

$$\{a^n b^n c^n \mid n \geq 1\}$$

EXAMPLE 4.11

Construct a grammar G generating $\{a^n b^n c^n \mid n \geq 1\}$.

Solution

Let $L = \{a^n b^n c^n \mid n \geq 1\}$. We try to construct L by recursion. We already know how to construct $a^n b^n$ recursively.

As it is difficult to construct $a^n b^n c^n$ recursively, we do it in two stages: (i) we construct $a^n \alpha^n$ and (ii) we convert α^n into $b^n c^n$. For stage (i), we can have the following productions $S \to aS\alpha \mid a\alpha$. A natural choice for α (to execute stage (ii)) is bc. But converting $(bc)^n$ into $b^n c^n$ is not possible as $(bc)^n$ has no variables. So we can take $\alpha = BC$, where B and C are variables. To bring B's together we introduce $CB \to BC$. We introduce some more productions to convert B's into b's and C's into c's. So we define G as

$$G = (\{S, B, C\}, \{a, b, c\}, P, S)$$

where P consists of

$$S \to aSBC \mid aBC,\ CB \to BC,\ aB \to ab,\ bB \to bb,\ bC \to bc,\ cC \to cc$$

$$S \Rightarrow aBC \Rightarrow abC \Rightarrow abc$$

Thus,

$$abc \in L(G)$$

Also,

$S \overset{*}{\Rightarrow} a^{n-1}S(BC)^{n-1}$	by applying $S \to aSBc$	$(n-1)$ times
$\Rightarrow a^{n-1}aBC(BC)^{n-1}$	by applying $S \to aBC$	
$\overset{*}{\Rightarrow} a^n B^n C^n$	by applying $CB \to BC$ (since $CB \to BC$ interchanges B and C)	several times
$\Rightarrow a^{n-1}abB^{n-1}C^n$	by applying $aB \to ab$	once
$\overset{*}{\Rightarrow} a^n b^n C^n$	by applying $bB \to bb$	several times
$\Rightarrow a^n b^{n-1}bcC^{n-1}$	by applying $bC \to bc$	once
$\Rightarrow a^n b^n c^n$	by applying $cC \to cc$	several times

Therefore,

$$L(G) \subseteq \{a^n b^n c^n \mid n \geq 1\}$$

To show that $\{a^n b^n c^n \mid n \geq 1\} \subseteq L(G)$, it is enough to prove that the only way to arrive at a terminal string is to proceed as above in deriving $a^n b^n c^n$ $(n \geq 1)$.

To start with, we have to apply only S-production. If we apply $S \to aBC$, first we get abc. Otherwise we have to apply $S \to aSBC$ once or several times and get the sentential form $a^{n-1}S(BC)^{n-1}$. At this stage the only production we can apply is $S \to aBC$, and the resulting string is $a^n(BC)^n$.

In the derivation of $a^nb^nc^n$, we converted all B's into b's and only then converted C's into c's. We show that this is the only way of arriving at a terminal string.

$a^n(BC)^n$ is a string of terminals followed by a string of variables. The productions we can apply to $a^n(BC)^n$ are either $CB \rightarrow BC$ or one of $aB \rightarrow ab$, $bB \rightarrow bb$, $bC \rightarrow bc$, $cC \rightarrow cc$. By the application of any one of these productions. the we get a sentential form which is a string of terminals followed by a string of variables. Suppose a C is converted before converting all B's. Then we have $a^n(BC)^n \overset{*}{\Rightarrow} a^nb^ic\alpha$, where $i < n$ and α is a string of B's and C's containing at least one B. In $a^nb^ic\alpha$, the variables appear only in α. As c appears just before α, the only production we can apply is $cC \rightarrow cc$. If α starts with B, we cannot proceed. Otherwise we apply $cC \rightarrow cc$ repeatedly until we obtain the string of the form $a^nb^ic^jB\alpha'$. But the only productions involving B are $aB \rightarrow ab$ and $bB \rightarrow bb$. As B is preceded by c in $a^nb^ic^jB\alpha'$, we cannot convert B, and so we cannot get a terminal string. So $L(G) \subseteq \{a^nb^nc^n \mid n \geq 1\}$. Thus, we have proved that

$$L(G) = \{a^nb^nc^n \mid n \geq 1\}$$

EXAMPLE 4.12

Construct a grammar G generating $\{xx \mid x \in \{a, b\}^*\}$.

Solution

We construct G as follows:

$$G = (\{S, S_1, S_2, S_3, A, B\}, \{a, b\}, P, S)$$

where P consists of

$$\begin{aligned}
P_1 &: \quad S \rightarrow S_1S_2S_3 \\
P_2, P_3 &: S_1S_2 \rightarrow aS_1A, \qquad S_1S_2 \rightarrow bS_1B \\
P_4, P_5 &: AS_3 \rightarrow S_2aS_3, \qquad BS_3 \rightarrow S_2bS_3 \\
P_6, P_7, P_8, P_9 &: Aa \rightarrow aA, \qquad Ab \rightarrow bA, \qquad Ba \rightarrow aB, \qquad Bb \rightarrow bB \\
P_{10}, P_{11} &: aS_2 \rightarrow S_2a, \qquad bS_2 \rightarrow S_2b \\
P_{12}, P_{13} &: S_1S_2 \rightarrow \Lambda, \qquad S_3 \rightarrow \Lambda
\end{aligned}$$

Remarks The following remarks give us an idea about the construction of productions P_1–P_{13}.

1. P_1 is the only S-production.
2. Using $S_1S_2 \rightarrow aS_1A$, we can add terminal a to the left of S_1 and variable A to the right. A is used to make us remember that we have added the terminal a to the left of S_1. Using $AS_3 \rightarrow S_2aS_3$, we add a to the right of S_2.

3. Using $S_1S_2 \rightarrow bS_1B$, we add b to the left of S_1 and variable B to the right. Using $BS_3 \rightarrow S_2bS_3$, we add b to the right of S_2.
4. S_2 acts as a centre-marker.
5. We can add terminals only by using P_2–P_5.
6. P_6–P_9 simply interchange symbols. They push A or B to the right. This enables us to place A or B to the left of S_3. (Only then we can apply P_4 or P_5.)
7. $S_1S_2 \rightarrow \Lambda$, $S_3 \rightarrow \Lambda$ are used to completely eliminate S_1, S_2, S_3.
8. P_{10}, P_{11} are used to push S_2 to the left. This enables us to get S_2 to the right of S_1 (so that we can apply P_{12}).

Let $L = \{xx \mid x \in \{a, b\}^*\}$. We first prove that $L \subseteq L(G)$. Now, we have

$$S \Rightarrow S_1S_2S_3 \Rightarrow aS_1AS_3 \Rightarrow aS_1S_2aS_3 \tag{4.1}$$

or

$$S \Rightarrow S_1S_2S_3 \Rightarrow bS_1BS_3 \Rightarrow bS_1S_2bS_3 \tag{4.2}$$

Let us start with xx with $x \in \{ab\}^*$. We can apply (4.1) or (4.2), depending on the first symbol of x. If the first two symbols in x are ab (the other cases are similar), we have

$$S \overset{*}{\Rightarrow} a\underline{S_1S_2}aS_3 \Rightarrow abS_1\underline{Ba}S_3 \Rightarrow abS_1a\underline{BS_3} \Rightarrow abS_1\underline{aS_2}bS_3 \Rightarrow abS_1S_2abS_3$$

Repeating the construction for every symbol in x, we get $xS_1S_2xS_3$. On application of P_{12} and P_{13}, we get

$$S \overset{*}{\Rightarrow} xS_1S_2xS_3 \overset{*}{\Rightarrow} x\Lambda x\Lambda = xx$$

Thus, $L \subseteq L(G)$.

To prove that $L(G) \subseteq L$, we note that the first three steps in any derivation of $L(G)$ are given by (4.1) or (4.2). Thus in any derivation (except $S \rightarrow \Lambda$), we get $aS_1S_2aS_3$ or $bS_1S_2bS_3$ as a sentential form.

We can discuss the possible ways of reducing $aS_1S_2aS_3$ (the other case is similar) to a terminal string. The first production that we can apply to $aS_1S_2aS_3$ is one of $S_1S_2 \rightarrow \Lambda$, $S_3 \rightarrow \Lambda$, $S_1S_2 \rightarrow aS_1A$, $S_1S_2 \rightarrow bS_1B$.

Case 1 We apply $S_1S_2 \rightarrow \Lambda$ to $aS_1S_2aS_3$. In this case we get $a\Lambda aS_3$. As the productions involving S_3 on the left are P_4, P_5 or P_{13}, we have to apply only $S_3 \rightarrow \Lambda$ to aaS_3 and get $aa \in L$.

Case 2 We apply $S_3 \rightarrow \Lambda$ to $aS_1S_2aS_3$. In this case we get $aS_1S_2a\Lambda$. If we apply $S_1S_2 \rightarrow \Lambda$, we get $a\Lambda a\Lambda = aa \in L$; or we can apply $S_1S_2 \rightarrow aS_1A$ to aS_1S_2a to get aaS_1Aa. In the latter case, we can apply only $Aa \rightarrow aA$ to aaS_1Aa. The resulting string is aaS_1aA which cannot be reduced further.

From Cases 1 and 2 we see that either we have to apply both P_{12} and P_{13} or neither of them.

Case 3 In this case we apply $S_1S_2 \rightarrow aS_1A$ or $S_1S_2 \rightarrow bS_1B$. If we apply $S_1S_2 \rightarrow aS_1A$ to $aS_1S_2aS_3$, we get aaS_1AaS_3. By the nature of productions we have to follow only $aaS_1AaS_3 \Rightarrow aaS_1aAS_3 \Rightarrow a^2S_1aS_2aS_3 \Rightarrow a^2S_1S_2a^2S_3$. If

we apply $S_1S_2 \rightarrow bS_1B$, we get $abS_1S_2abS_3$. Thus the effect of applying $S_1S_2 \rightarrow aS_1A$ is to add a to the left of S_1S_2 and S_3.

If we apply $S_1S_2 \rightarrow \Lambda$, $S_3 \rightarrow \Lambda$ (By Cases 1 and 2 we have to apply both) we get $abab \in L$. Otherwise, by application of P_2 or P_3, we add the same terminal symbol to the left of S_1S_2 and S_3. The resulting string is of the form $xS_1S_2xS_3$. Ultimately, we have to apply P_{12} and P_{13} and get $x\Lambda x\Lambda = xx \in L$. So $L(G) \subseteq L$. Hence, $L(G) = L$.

EXAMPLE 4.13

Let $G = (\{S, A_1, A_2\}, \{a, b\}, P, S)$, where P consists of

$$S \rightarrow aA_1A_2a,\ A_1 \rightarrow baA_1A_2b,\ A_2 \rightarrow A_1ab,\ aA_1 \rightarrow baa,\ bA_2b \rightarrow abab$$

Test whether $w = baabbabaaabbaba$

is in $L(G)$.

Solution

We have to start with an S-production. At every stage we apply a suitable production which is likely to derive w. In this example, we underline the substring to be replaced by the use of a production.

$$\begin{aligned} S &\Rightarrow \underline{aA_1}\, A_2a \\ &\Rightarrow baa\,\underline{A_2}\, a \\ &\Rightarrow baa\,\underline{A_1}\, aba \\ &\Rightarrow baab\,\underline{aA_1}\, A_2baba \\ &\Rightarrow baabbaa\,\underline{A_2}\, baba \\ &\Rightarrow baabba\,\underline{aA_1}\, abbaba \\ &\Rightarrow baabbabaaabbaba = w \end{aligned}$$

Therefore,

$$w \in L(G)$$

EXAMPLE 4.14

If the grammar G is given by the productions $S \rightarrow aSa \mid bSb \mid aa \mid bb \mid \Lambda$, show that (i) $L(G)$ has no strings of odd length, (ii) any string in $L(G)$ is of length $2n$, $n \geq 0$, and (iii) the number of strings of length $2n$ is 2^n.

Solution

On application of any production (except $S \rightarrow \Lambda$), a variable is replaced by two terminals and at the most one variable. So, every step in any derivation increases the number of terminals by 2 except that involving $S \rightarrow \Lambda$. Thus, we have proved (i) and (ii).

To prove (iii), consider any string w of length $2n$. Then it is of the form $a_1a_2 \ldots a_na_n \ldots a_1$ involving n 'parameters' $a_1, a_2, \ldots, a_n$. Each a_i can be either a or b. So the number of such strings is 2^n. This proves (iii).

4.2 CHOMSKY CLASSIFICATION OF LANGUAGES

In the definition of a grammar (V_N, Σ, P, S), V_N and Σ are the sets of symbols and $S \in V_N$. So if we want to classify grammars, we have to do it only by considering the form of productions. Chomsky classified the grammars into four types in terms of productions (types 0–3).

A type 0 grammar is any phrase structure grammar without any restrictions. (All the grammars we have considered are type 0 grammars.)

To define the other types of grammars, we need a definition.

In a production of the form $\phi A\psi \rightarrow \phi\alpha\psi$, where A is a variable, ϕ is called the left context, ψ the right context, and $\phi\alpha\psi$ the replacement string.

EXAMPLE 4.15

(a) In $ab\underline{A}bcd \rightarrow ab\underline{AB}bcd$, ab is the left context, bcd is the right context, $\alpha = AB$.

(b) In $A\underline{C} \rightarrow A$, A is the left context, Λ is the right context. $\alpha = \Lambda$. The production simply erases C when the left context is A and the right context is Λ.

(c) For $C \rightarrow \Lambda$, the left and right contexts are Λ. And $\alpha = \Lambda$. The production simply erases C in any context.

A production without any restrictions is called a type 0 production.

A production of the form $\phi A\psi \rightarrow \phi\alpha\psi$ is called a type 1 production if $\alpha \neq \Lambda$. In type 1 productions, erasing of A is not permitted.

EXAMPLE 4.16

(a) $a\underline{A}bcD \rightarrow a\underline{bcD}bcD$ is a type 1 production where where a, bcD are the left context and right context, respectively. A is replaced by $bcD \neq \Lambda$.

(b) $A\underline{B} \rightarrow A\underline{bBc}$ is a type 1 production. The left context is A, the right context is Λ.

(c) $A \rightarrow abA$ is a type 1 production. Here both the left and right contexts are Λ.

Definition 4.7 A grammar is called type 1 or context-sensitive or context-dependent if all its productions are type 1 productions. The production $S \rightarrow \Lambda$ is also allowed in a type 1 grammar, but in this case S does not appear on the right-hand side of any production.

Definition 4.8 The language generated by a type 1 grammar is called a type 1 or context-sensitive language.

Note: In a context-sensitive grammar G, we allow $S \to \Lambda$ for including Λ in $L(G)$. Apart from $S \to \Lambda$, all the other productions do not decrease the length of the working string.

A type 1 production $\phi A \psi \to \phi \alpha \psi$ does not increase the length of the working string. In other words, $|\phi A \psi| \leq |\phi \alpha \psi|$ as $\alpha \neq \Lambda$. But if $\alpha \to \beta$ is a production such that $|\alpha| \leq |\beta|$, then it need not be a type 1 production. For example, $BC \to CB$ is not of type 1. We prove that such productions can be replaced by a set of type 1 productions (Theorem 4.2).

Theorem 4.1 Let G be a type 0 grammar. Then we can find an equivalent grammar G_1 in which each production is either of the form $\alpha \to \beta$, where α and β are strings of variables only, or of the form $A \to a$, where A is a variable and a is a terminal. G_1 is of type 1, type 2 or type 3 according as G is of type 1, type 2 or type 3.

Proof We construct G_1 as follows: For constructing productions of G_1, consider a production $\alpha \to \beta$ in G, where α or β has some terminals. In both α and β we replace every terminal by a new variable C_a and get α' and β'. Thus, corresponding to every $\alpha \to \beta$, where α or β contains some terminal, we construct $\alpha' \to \beta'$ and productions of the form $C_a \to a$ for every terminal a appearing in α or β. The construction is performed for every such $\alpha \to \beta$. The productions for G_1 are the new productions we have obtained through the above construction. For G_1 the variables are the variables of G together with the new variables (of the form C_a). The terminals and the start symbol of G_1 are those of G. G_1 satisfies the required conditions and is equivalent to G. So $L(G) = L(G_1)$. ∎

Definition 4.9 A grammar $G = (V_N, \Sigma, P, S)$ is monotonic (or length-increasing) if every production in P is of the form $\alpha \to \beta$ with $|\alpha| \leq |\beta|$ or $S \to \Lambda$. In the second case, S does not appear on the right-hand side of any production in P.

Theorem 4.2 Every monotonic grammar G is equivalent to a type 1 grammar.

Proof We apply Theorem 4.1 to get an equivalent grammar G_1. We construct G' equivalent to grammar G_1 as follows: Consider a production $A_1A_2 \ldots A_m \to B_1B_2 \ldots B_n$ with $n \geq m$ in G_1. If $m = 1$, then the above production is of type 1 (with left and right contexts being Λ). Suppose $m \geq 2$. Corresponding to $A_1A_2 \ldots A_m \to B_1B_2 \ldots B_n$, we construct the following type 1 productions introducing the new variables $C_1, C_2, \ldots, C_m$.

$$\underline{A_1} A_2 \ldots A_m \to \underline{C_1} A_2 \ldots A_m$$

$$C_1 \underline{A_2} \ldots A_m \to C_1 \underline{C_2} A_3 \ldots A_m$$

$$C_1 C_2 \underline{A_3} \ldots A_m \to C_1 C_2 \underline{C_3} A_4 \ldots A_m \ldots$$

$$C_1 C_2 \ldots C_{m-1} \underline{A_m} \to C_1 C_2 \ldots C_m B_{m+1} B_{m+2} \ldots B_n$$

$$\underline{C_1 C_2} \ldots C_m B_{m+1} \ldots B_n \rightarrow \underline{B_1 C_2} \ldots C_m B_{m+1} \ldots B_n$$

$$B_1 \underline{C_2 C_3} \ldots B_n \rightarrow B_1 \underline{B_2 C_3} \ldots B_n \ldots$$

$$B_1 B_2 \ldots \underline{C_m B_{m+1}} \ldots B_n \rightarrow B_1 B_2 \ldots \underline{B_m} \ldots B_n$$

The above construction can be explained as follows. The production

$$A_1 A_2 \ldots A_m \rightarrow B_1 B_2 \ldots B_n$$

is not of type 1 as we replace more than one symbol on L.H.S. In the chain of productions we have constructed, we replace A_1 by C_1, A_2 by $C_2 \ldots$, A_m by $C_m B_{m+1} \ldots B_n$. Afterwards, we start replacing C_1 by B_1, C_2 by B_2, etc. As we replace only one variable at a time, these productions are of type 1.

We repeat the construction for every production in G_1 which is not of type 1. For the new grammar G', the variables are the variables of G_1 together with the new variables. The productions of G' are the new type 1 productions obtained through the above construction. The terminals and the start symbol of G' are those of G_1.

G' is context-sensitive and from the construction it is easy to see that $L(G') = L(G_1) = L(G)$. ∎

Definition 4.10 A type 2 production is a production of the form $A \rightarrow \alpha$, where $A \in V_N$ and $\alpha \in (V_N \cup \Sigma)^*$. In other words, the L.H.S. has no left context or right context. For example, $S \rightarrow Aa$, $A \rightarrow a$, $B \rightarrow abc$, $A \rightarrow \Lambda$ are type 2 productions.

Definition 4.11 A grammar is called a type 2 grammar if it contains only type 2 productions. It is also called a context-free grammar (as A can be replaced by α in any context). A language generated by a context-free grammar is called a type 2 language or a context-free language.

Definition 4.12 A production of the form $A \rightarrow a$ or $A \rightarrow aB$, where $A, B \in V_N$ and $a \in \Sigma$, is called a type 3 production.

Definition 4.13 A grammar is called a type 3 or regular grammar if all its productions are type 3 productions. A production $S \rightarrow \Lambda$ is allowed in type 3 grammar, but in this case S does not appear on the right-hand side of any production.

EXAMPLE 4.17

Find the highest type number which can be applied to the following productions:

(a) $S \rightarrow Aa, \quad A \rightarrow c \mid Ba, \quad B \rightarrow abc$
(b) $S \rightarrow ASB \mid d, \quad A \rightarrow aA$
(c) $S \rightarrow aS \mid ab$

Solution

(a) $S \to Aa$, $A \to Ba$, $B \to abc$ are type 2 and $A \to c$ is type 3. So the highest type number is 2.

(b) $S \to ASB$ is type 2. $S \to d$, $A \to aA$ are type 3. Therefore, the highest type number is 2.

(c) $S \to aS$ is type 3 and $S \to ab$ is type 2. Hence the highest type number is 2.

4.3 LANGUAGES AND THEIR RELATION

In this section we discuss the relation between the classes of languages that we have defined under the Chomsky classification.

Let $\mathcal{L}_0$, $\mathcal{L}_{csl}$, $\mathcal{L}_{cfl}$ and $\mathcal{L}_{rl}$ denote the family of type 0 languages, context-sensitive languages, context-free languages and regular languages, respectively.

Property 1 From the definition, it follows that $\mathcal{L}_{rl} \subseteq \mathcal{L}_{cfl}$, $\mathcal{L}_{csl} \subseteq \mathcal{L}_0$, $\mathcal{L}_{cfl} \subseteq \mathcal{L}_0$.

Property 2 $\mathcal{L}_{cfl} \subseteq \mathcal{L}_{csl}$. The inclusion relation is not immediate as we allow $A \to \Lambda$ in context-free grammars even when $A \neq S$, but not in context-sensitive grammars (we allow only $S \to \Lambda$ in context-sensitive grammars). In Chapter 6 we prove that a context-free grammar G with productions of the form $A \to \Lambda$ is equivalent to a context-free grammar G_1 which has no productions of the form $A \to \Lambda$ (except $S \to \Lambda$). Also, when G_1 has $S \to \Lambda$, S does not appear on the right-hand side of any production. So G_1 is context-sensitive. This proves $\mathcal{L}_{cfl} \subseteq \mathcal{L}_{csl}$.

Property 3 $\mathcal{L}_{rl} \subseteq \mathcal{L}_{cfl} \subseteq \mathcal{L}_{csl} \subseteq \mathcal{L}_0$. This follows from properties 1 and 2.

Property 4 $\mathcal{L}_{rl} \subset_{\neq} \mathcal{L}_{cfl} \subset_{\neq} \mathcal{L}_{csl} \subset_{\neq} \mathcal{L}_0$.

In Chapter 5, we shall prove that $\mathcal{L}_{rl} \subset_{\neq} \mathcal{L}_{cfl}$. In Chapter 6, we shall prove that $\mathcal{L}_{cfl} \subset_{\neq} \mathcal{L}_{csl}$. In Section 9.7, we shall establish that $\mathcal{L}_{csl} \subset_{\neq} \mathcal{L}_0$.

Remarks 1. The grammars given in Examples 4.1–4.4 and 4.6–4.9 are context- free but not regular. The grammar given in Example 4.5 is regular. The grammars given in Examples 4.10 and 4.11 are not context-sensitive as we have productions of the form $2A_1 \to A_1 2$, $CB \to BC$ which are not type 1 rules. But they are equivalent to a context-sensitive grammar by Theorem 4.2.

2. Two grammars of different types may generate the same language. For example, consider the regular grammar G given in Example 4.5. It generates $\{a, b\}^+$. Let G' be given by $S \to SS \mid aS \mid bS \mid a \mid b$. Then $L(G') = L(G)$ as the productions $S \to aS \mid bS \mid a \mid b$ are in G as well, and $S \to SS$ does not generate any more string.

3. The type of a given grammar is easily decided by the nature of productions. But to decide on the type of a given subset of Σ^*, it is more difficult. By Remark 2, the same set of strings may be generated by a grammar of higher type. To prove that a given language is not regular or context-free, we need powerful theorems like Pumping Lemma.

4.4 RECURSIVE AND RECURSIVELY ENUMERABLE SETS

The results given in this section will be used to prove $\mathcal{L}_{csl} \subset_{\neq} \mathcal{L}_0$ in Section 9.7. For defining recursive sets, we need the definition of a procedure and an algorithm.

A *procedure* for solving a problem is a finite sequence of instructions which can be mechanically carried out given any input.

An *algorithm* is a procedure that terminates after a finite number of steps for any input.

Definition 4.14 A set X is recursive if we have an algorithm to determine whether a given element belongs to X or not.

Definition 4.15 A recursively enumerable set is a set X for which we have a procedure to determine whether a given element belongs to X or not.

It is clear that a recursive set is recursively enumerable.

Theorem 4.3 A context-sensitive language is recursive.

Proof Let $G = (V_N, \Sigma, P, S)$ and $w \in \Sigma^*$. We have to construct an algorithm to test whether $w \in L(G)$ or not. If $w = \Lambda$, then $w \in L(G)$ iff $S \to \Lambda$ is in P. As there are only a finite number of productions in P, we have to test whether $S \to \Lambda$ is in P or not.

Let $|w| = n \geq 1$. The algorithm is based on the construction of a sequence $\{W_i\}$ of subsets of $(V_N \cup \Sigma)^*$. W_i is simply the set of all sentential forms of length less than or equal to n, derivable in at most i steps. The construction is done recursively as follows:

(i) $W_0 = \{S\}$.

(ii) $W_{i+1} = W_i \cup \{\beta \in (V_N \cup \Sigma)^* |$ there exists α in W_i such that $\alpha \Rightarrow \beta$ and $|\beta| \leq n\}$.

W_i's satisfy the following:

(iii) $W_i \subseteq W_{i+1}$ for all $i \geq 0$.

(iv) There exists k such that $W_k = W_{k+1}$.

(v) If k is the smallest integer such that $W_k = W_{k+1}$, then $W_k = \{\alpha \in (V_N \cup \Sigma)^* | S \overset{*}{\Rightarrow} \alpha$ and $|\alpha| \leq n\}$.

The point (iii) follows from the point (ii). To prove the point (iv), we consider the number N of strings over $V_N \cup \Sigma$ of length less than or equal to n. If $|V_N \cup \Sigma| = m$, then $N = 1 + m + m^2 + \ldots + m^n$ since m^i is the number of strings of length i over $V_N \cup \Sigma$, i.e. $N = (m^{n+1} - 1)/(m - 1)$, and N is fixed as it depends only on n and m. As any string in W_i is of length at most n, $|W_i| \leq N$. Therefore, $W_k = W_{k+1}$ for some $k \leq N$. This proves the point (iv).

From point (ii) it follows that $W_k = W_{k+1}$ implies $W_{k+1} = W_{k+2}$.

$$\{\alpha \in (V_N \cup \Sigma)^* | S \overset{*}{\Rightarrow} \alpha, |\alpha| \leq n\} = W_1 \cup W_2 \cup \ldots \cup W_k \cup W_{k+1} \ldots$$
$$= W_1 \cup W_2 \cup \ldots \cup W_k$$
$$= W_k \text{ from point (iii)}$$

This proves the point (v).

From the point (v) it follows that $w \in L(G)$ (i.e. $S \overset{*}{\Rightarrow} w$) if and only if $w \in W_k$. Also, $W_1, W_2, \ldots, W_k$ can be constructed in a finite number of steps. We give the required algorithm as follows:

Algorithm to test whether $w \in L(G)$. 1. Construct $W_1, W_2, \ldots$ using the points (i) and (ii). We terminate the construction when $W_{k+1} = W_k$ for the first time.

2. If $w \in W_k$, then $w \in L(G)$. Otherwise, $w \notin L(G)$. (As $|W_k| \leq N$, testing whether w is in W_k requires at most N steps.) ∎

EXAMPLE 4.18

Consider the grammar G given by $S \to 0SA_12$, $S \to 012$, $2A_1 \to A_122$, $1A_1 \to 11$. Test whether (a) $00112 \in L(G)$ and (b) $001122 \in L(G)$.

Solution

(a) To test whether $w = 00112 \in L(G)$, we construct the sets W_0, W_1, W_2 etc. $|w| = 5$.

$$W_0 = \{S\}$$

$$W_1 = \{012, S, 0SA_12\}$$

$$W_2 = \{012, S, 0SA_12\}$$

As $W_2 = W_1$, we terminate. (Although $0SA_12 \Rightarrow 0012A_12$, we cannot include $0012A_12$ in W_1 as its length is > 5.) Then $00112 \notin W_1$. Hence, $00112 \notin L(G)$.

(b) To test whether $w = 001122 \in L(G)$. Here, $|w| = 6$. We construct W_0, W_1, W_2, etc.

$$W_0 = \{S\}$$

$$W_1 = \{012, S, 0SA_12\}$$

$$W_2 = \{012, S, 0SA_12, 0012A_12\}$$

$$W_3 = \{012, S, 0SA_12, 0012A_12, 001A_122\}$$

$$W_4 = \{012, S, 0SA_12, 0012A_12, 001A_122, 001122\}$$

$$W_5 = \{012, S, 0SA_12, 0012A_12, 001A_122, 001122\}$$

As $W_5 = W_4$, we terminate. Then $001122 \in W_4$. Thus, $001122 \in L(G)$.

The following theorem is of theoretical interest, and shows that there exists a recursive set over $\{0, 1\}$ which is not a context-sensitive language. The proof is by the diagonalization method which is used quite often in set theory.

Theorem 4.4 There exists a recursive set which is not a context-sensitive language over $\{0, 1\}$.

Proof Let $\Sigma = \{0, 1\}$. We write the elements of Σ^* as a sequence (i.e. the elements of Σ^* are enumerated as the first element, second element, etc.) For

example, one such way of writing is Λ, 0, 1, 00, 01, 10, 11, 000, ... In this case, 010 will be the 10th element.

As every grammar is defined in terms of a finite alphabet set and a finite set of productions, we can also write all context-sensitive grammars over Σ as a sequence, say $G_1, G_2, \ldots$.

We define $X = \{w_i \in \Sigma^* \mid w_i \notin L(G_i)\}$. We can show that X is recursive. If $w \in \Sigma^*$, then we can find i such that $w = w_i$. This can be done in a finite number of steps (depending on $|w|$). For example, if $w = 0100$, then $w = w_{20}$. As G_{20} is context-sensitive, we have an algorithm to test whether $w = w_{20} \in L(G_{20})$ by Theorem 4.3. So X is recursive.

We prove by contradiction that X is not a context-sensitive language. If it is so, then $X = L(G_n)$ for some n. Consider w_n (the nth element in Σ^*). By definition of X, $w_n \in X$ implies $w_n \notin L(G_n)$. This contradicts $X = L(G_n)$. $w_n \notin X$ implies $w_n \in L(G_n)$ and once again, this contradicts $X = L(G_n)$. Thus, $X \neq L(G_n)$ for any n, i.e. X is not a context-sensitive language. ∎

4.5 OPERATIONS ON LANGUAGES

We consider the effect of applying set operations on $\mathcal{L}_0$, $\mathcal{L}_{cs1}$, $\mathcal{L}_{cf1}$, $\mathcal{L}_{r1}$. Let A and B be any two sets of strings. The concatenation AB of A and B is defined by $AB = \{uv \mid u \in A, v \in B\}$. (Here, uv is the concatenation of the strings u and v.)

We define A^1 as A and A^{n+1} as A^nA for all $n \geq 1$.

The transpose set A^T of A is defined by

$$A^T = \{u^T \mid u \in A\}$$

Theorem 4.5 Each of the classes $\mathcal{L}_0$, $\mathcal{L}_{cs1}$, $\mathcal{L}_{cf1}$, $\mathcal{L}_{r1}$ is closed under union.

Proof Let L_1 and L_2 be two languages of the same type i. We can apply Theorem 4.1 to get grammars

$$G_1 = (V'_N, \Sigma_1, P_1, S_1) \qquad \text{and} \qquad G_2 = (V''_N, \Sigma_2, P_2, S_2)$$

of type i generating L_1 and L_2, respectively. So any production in G_1 or G_2 is either $\alpha \to \beta$, where α, β contain only variables or $A \to a$, where $A \in V_N$, $a \in \Sigma$.

We can further assume that $V'_N \cap V''_N = \emptyset$. (This is achieved by renaming the variables of V''_N if they occur in V'_N.)

Define a new grammar G_u as follows:

$$G_u = (V'_N \cup V''_N \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_u, S)$$

where S is a new symbol, i.e. $S \notin V'_N \cup V''_N$

$$P_u = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}$$

We prove $L(G_u) = L_1 \cup L_2$ as follows: If $w \in L_1 \cup L_2$, then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$ or $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} w$. Therefore,

$$S \underset{G_u}{\Rightarrow} S_1 \overset{*}{\underset{G_u}{\Rightarrow}} w \quad \text{or} \quad S \underset{G_u}{\Rightarrow} S_2 \overset{*}{\underset{G_u}{\Rightarrow}} w, \text{ i.e. } w \in L(G_u)$$

Thus, $L_1 \cup L_2 \subseteq L(G_u)$.

To prove that $L(G_u) \subseteq L_1 \cup L_2$, consider a derivation of w. The first step should be $S \Rightarrow S_1$ or $S \Rightarrow S_2$. If $S \Rightarrow S_1$ is the first step, in the subsequent steps S_1 is changed. As $V'_N \cap V''_N \neq \emptyset$, these steps should involve only the variables of V'_N and the productions we apply are in P_1. So $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Similarly, if the first step is $S \Rightarrow S_2$, then $S \underset{G_2}{\Rightarrow} S_2 \overset{*}{\underset{G_2}{\Rightarrow}} w$. Thus, $L(G_u) = L_1 \cup L_2$. Also, $L(G_u)$ is of type 0 or type 2 according as L_1 and L_2 are of type 0 or type 2. If Λ is not in $L_1 \cup L_2$, then $L(G_u)$ is of type 3 or type 1 according as L_1 and L_2 are of type 3 or type 1.

Suppose $\Lambda \in L_1$. In this case, define

$$G_u = (V'_N \cup V''_N \cup \{S, S'\}, \Sigma_1 \cup \Sigma_2, P_u, S')$$

where (i) S' is a new symbol, i.e. $S' \notin V'_N \cup V''_N \cup \{S\}$, and (ii) $P_u = P_1 \cup P_2 \cup \{S' \to S, S \to S_1, S \to S_2\}$. So, $L(G_u)$ is of type 1 or type 3 according as L_1 and L_2 are of type 1 or type 3. When $\Lambda \in L_2$, the proof is similar. ∎

Theorem 4.6 Each of the classes $\mathscr{L}_0$, $\mathscr{L}_{\text{cs1}}$, $\mathscr{L}_{\text{cfl}}$, $\mathscr{L}_{\text{rl}}$ is closed under concatenation.

Proof Let L_1 and L_2 be two languages of type i. Then, as in Theorem 4.5, we get $G_1 = (V'_N, \Sigma_1, P_1, S_1)$ and $G_2 = (V''_N, \Sigma_2, P_2, S_2)$ of the same type i. We have to prove that L_1L_2 is of type i.

Construct a new grammar G_{con} as follows:

$$G_{\text{con}} = (V'_N \cup V''_N \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_{\text{con}}, S)$$

where $S \notin V'_N \cup V''_N$.

$$P_{\text{con}} = P_1 \cup P_2 \cup \{S \to S_1S_2\}$$

We prove $L_1L_2 = L(G_{\text{con}})$. If $w = w_1w_2 \in L_1L_2$, then

$$S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w_1, \qquad S_2 \overset{*}{\underset{G_2}{\Rightarrow}} w_2$$

So,

$$S \underset{G_{\text{con}}}{\Rightarrow} S_1S_2 \overset{*}{\underset{G_{\text{con}}}{\Rightarrow}} w_1w_2$$

Therefore,

$$L_1L_2 \subseteq L(G_{\text{con}})$$

If $w \in L(G_{con})$, then the first step in the derivation of w is $S \Rightarrow S_1S_2$. As $V'_N \cap V''_N = \emptyset$ and the productions in G_1 or G_2 involve only the variables (except those of the form $A \to a$), we have $w = w_1w_2$, where $S \overset{*}{\underset{G_1}{\Rightarrow}} w_1$ and $S \overset{*}{\underset{G_2}{\Rightarrow}} w_2$. Thus $L_1L_2 = L(G_{con})$. Also, G_{con} is of type 0 or type 2 according as G_1 and G_2 are of type 0 or type 2. The above construction is sufficient when G_1 and G_2 are also of type 3 or type 1 provided $\Lambda \notin L_1 \cup L_2$.

Suppose G_1 and G_2 are of type 1 or type 3 and $\Lambda \in L_1$ or $\Lambda \in L_2$. Let $L'_1 = L_1 - \{\Lambda\}$, $L'_2 = L_2 - \{\Lambda\}$. Then

$$L_1L_2 = \begin{cases} L'_1L'_2 \cup L'_2 & \text{if } \Lambda \text{ is in } L_1 \text{ but not in } L_2 \\ L'_1L'_2 \cup L'_1 & \text{if } \Lambda \text{ is in } L_2 \text{ but not in } L_1 \\ L'_1L'_2 \cup L'_1 \cup L'_2 \cup \{\Lambda\} & \text{if } \Lambda \text{ is in } L_1 \text{ and also in } L_2 \end{cases}$$

As we have already shown that $\mathcal{L}_{csl}$ and $\mathcal{L}_{rl}$ are closed under union, L_1L_2 is of type 1 or type 3 according as L_1 and L_2 are of type 1 or type 3. ∎

Theorem 4.7 Each of the classes $\mathcal{L}_0$, $\mathcal{L}_{csl}$, $\mathcal{L}_{cfl}$, $\mathcal{L}_{rl}$ is closed under the transpose operation.

Proof Let L be a language of type i. Then $L = L(G)$, where G is of type i.

We construct a new grammar G^T as follows: $G^T = (V_N, \Sigma, P^T, S)$, where the productions of P^T are constructed by reversing the symbols on L.H.S. and R.H.S. of every production in P. Symbolically, $\alpha^T \to \beta^T$ is in P^T if $\alpha \to \beta$ is in P.

From the construction it is obvious that G^T is of type 0, 1 or 2 according as G is of type 0, 1 or 2 and $L(G^T) = L^T$. For regular grammar, the proof is given in Chapter 5.

It is more difficult to establish the closure property under intersection at present as we need the properties of families of languages under consideration. We state the results without proof. We prove some of them in Chapter 8.

Theorem 4.8 (i) Each of the families $\mathcal{L}_{01}$, $\mathcal{L}_{csl}$, $\mathcal{L}_{rl}$ is closed under intersection.

(ii) $\mathcal{L}_{cfl}$ is not closed under intersection. But the intersection of a context-free language and a regular language is context-free.

4.6 LANGUAGES AND AUTOMATA

In Chapters 7 and 9, we shall construct accepting devices for the four types of languages. Figure 4.1 describes the relation between the four types of languages and automata: TM, LBA, pda, and FA stand for Turing machine, linear bounded automaton, pushdown automaton and finite automaton, respectively.

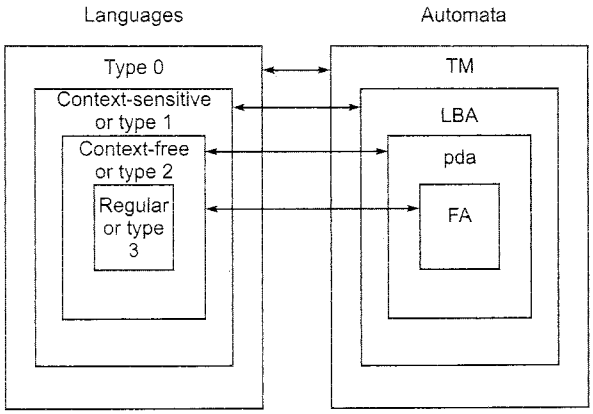


Fig. 4.1 Languages and the corresponding automata.

4.7 SUPPLEMENTARY EXAMPLES

EXAMPLE 4.19

Construct a context-free grammar generating

(a) $L_1 = \{a^n b^{2n} \mid n \geq 1\}$
(b) $L_2 = \{a^m b^n \mid m > n,\ m,\ n \geq 1\}$
(c) $L_3 = \{a^m b^n \mid m < n,\ m,\ n,\ \geq 1\}$
(d) $L_4 = \{a^m b^n \mid m,\ n \geq 0,\ m \neq n\}$

Solution

(a) Let $G_1 = (\{S\}, \{a, b\}, P, S)$ where P consists of $S \to aSbb$, $S \to abb$.

(b) Let $G_2 = (\{S, A\}, \{a, b\}, P, S)$ where P consists of $S \to aS \mid aA$, $A \to aAb$, $A \to ab$. It is easy to see that $L(G) \subseteq L_2$. We prove the difficult part. Let $a^m b^n \in L_2$. Then, $m > n \geq 1$. As $m > n$, we have $m - n \geq 1$. So the derivation of $a^m b^n = a^{m-n} a^n b^n$ is

$$S \overset{*}{\Rightarrow} a^{m-n-1} S \Rightarrow a^{m-n} aA \overset{*}{\Rightarrow} a^{m-n} a^{n-1} A b^{n-1} \Rightarrow a^m b^n$$

(c) Let $G_3 = (\{S, B\}, \{a, b\}, P, S)$ where P consists of $S \to Sb \mid Bb$, $B \to aBb$, $B \to ab$. This construction is similar to construction in (b). $S \to Sb \mid Bb$ are used to generate b^{n-m}. The remaining productions will generate $a^m b^m$. Hence $L(G_3) = L_3$.

(d) Note that $L_4 = L_2 \cup L_3 \cup L' \cup L''$ where $L' = \{b^n \mid n \geq 1\}$ and $L'' = \{a^n \mid n \geq 1\}$. It is easy to see how to construct grammars generating L_2, L_3 and L' and L''. Define G_4 by combining these constructions. Let

$G_4 = (\{S, S_1, S_2, S_3, S_4, A\}, \{a, b\}, P_4, S)$ where P_4 consists of

$$S \to S_1 \mid S_2 \mid S_3 \mid S_4, \quad S_1 \to aS_1 \mid Aa \mid aAb \mid ab,$$

$$S_2 \to S_2 b \mid Ab, \quad S_3 \to bS_3 \mid b, \quad \text{and} \quad S_4 \to aS_4 \mid a.$$

It is easy to see that $L(G_4) = L_4$.

EXAMPLE 4.20

Construct a grammar accepting

$L = \{w \in \{a, b\}^* \mid$ the number of a's in w is divisible by $3\}$.

Solution

Consider a string over $\{a, b\}$ having three a's. These three a's appear amidst the strings of b. A typical string is $b^m a b^n a b^s a b^t$. For generating b^m, we can have the productions $S \to bS$. For getting the first a, we can have $S \to aA$. For getting b^n afterwards, we have $A \to bA$. For getting the second a, we can have $A \to aB$. For getting b^s, we have $B \to bB$. For getting the third a and repetition of this pattern, we can have $B \to a \mid aS$.

Now we construct G as follows:

$G = (\{S, A, B\}, \{a, b\}, P, S)$ where P consists of

$$S \to bS,\ S \to aA,\ A \to bA,\ A \to aB,\ B \to bB,\ B \to a,\ B \to aS.$$

A string in L is of the form $y_1 y_2 \ldots y_n$ where each y_i is of the form $b^m a b^n a b^s a$ for some nonnegative integers m, n and s.

$$S \overset{*}{\Rightarrow} b^m S \Rightarrow b^m aA \overset{*}{\Rightarrow} b^m ab^n A \Rightarrow b^m ab^n aB \overset{*}{\Rightarrow} b^m ab^n ab^s S$$

Hence G is the required grammar.

EXAMPLE 4.21

Construct a grammar G such that

$L(G) = \{w \in \{a, b\} \mid w$ has an equal number of a's and b's$\}$.

Solution

Define $G = (\{S, A, B\}, \{a, b\}, P, S)$ where P consists of

$$S \to aB \mid bA,\quad A \to aS \mid bAA \mid a,\quad B \to bS \mid aBB \mid b$$

To prove that G accepts the given language, we prove the following by induction on $|w|$.

(i) $S \overset{*}{\Rightarrow} w$ if and only if w consists of an equal number of a's and b's.

(ii) $A \overset{*}{\Rightarrow} w$ if and only if w consists of one more a than the number of b's.

(iii) $B \overset{*}{\Rightarrow} w$ if and only if w consists of one more b than the number of a's.

The ponits (i), (ii), (iii) are true when $|w| = 1$. For $A \Rightarrow a$ and a is the only string of length one which can be derived from B. Also, no string of length one is derivable from S. Thus, there is basis for induction.

Assume points (i), (ii), (iii) to be true for all strings of length $k - 1$. Let $|w| = k$.

We prove the 'only if' part of (i). Let $S \overset{*}{\Rightarrow} w$. Then the first production has to be either $S \to aB$ or $S \to bA$. If the first production is $S \to aB$, then

$S \Rightarrow aB \overset{*}{\Rightarrow} w$. Hence $w = aw_1$, $B \overset{*}{\Rightarrow} w_1$ and $|w_1| = k - 1$. By induction hypothesis, the point (iii) is true for w_1. This means that w_1 has one more b than the number of a's. Hence $w = aw_1$ has an equal number of a's and b's. (The proof is similar if the first production is $S \rightarrow bA$.)

To prove the 'if part', assume that w has an equal number of a's and b's. If w starts with a, then $w = aw_1$, where $|w_1| = k - 1$. (If w starts with b the proof is similar.) Also w_1 has one more b than the number of a's. By induction hypothesis, the point (iii) is true for w_1. Then $B \overset{*}{\Rightarrow} w_1$. As $S \rightarrow aB$ is a production, we have $S \Rightarrow aB$. So $S \Rightarrow aB \overset{*}{\Rightarrow} aw_1 = w$, i.e. $S \overset{*}{\Rightarrow} w$, which proves the 'if part' of point (i).

Similarly we can prove the points (ii) and (iii) for a string w of length k. By the principle of induction the points (i), (ii), (iii) are true for all strings w.

In particular, from point (i), we can conclude that

$$L(G) = \{w \in \{a, b\} \mid w \text{ has an equal number of } a\text{'s and } b\text{'s}\}$$

EXAMPLE 4.22

Construct a grammar G accepting the set L of all strings over $\{a, b\}$ having more a's than b's.

Solution

For generating strings with a's but with no b's, we can have the production $S \rightarrow a$, $S \rightarrow aS$, $S \rightarrow Sa$. (If $xa \in L$, then $ax \in L$. Hence we have $S \rightarrow aS$ and $S \rightarrow Sa$.) If $x, y \in L$, then $xy \in L$ (also $yx \in L$). So xy or yx has at least two more a's than b's. So we can add a b. This can be done by having the productions $S \rightarrow bSS$, $S \rightarrow SbS$, $S \rightarrow SSb$. (i.e. b can be added at the beginning, or at the end of SS, or between S and S). With this motivation, we can construct G as follows:

$G = (\{S\}, \{a, b\}, P, S)$ where P consists of

$$S \rightarrow a \mid aS \mid Sa \mid bSS \mid SbS \mid SSb$$

We can easily prove that G accepts all strings over $\{a, b\}$ having more a's than b's.

EXAMPLE 4.23

Construct a grammar G accepting all strings over $\{a, b\}$ containing an unequal number of a's and b's.

Solution

As in Example 4.20, we can construct a grammar accepting all strings having more b's than a's. The required language is the union of the language of Example 4.20 and a similar one having more b's than a's. So we construct G as follows:

$G = (\{S, S_1, S_2\}, \{a, b\}, P, S)$ where P consists of

$S \to S_1 \mid S_2$

$S_1 \to a \mid aS_1 \mid S_1a \mid bS_1S_1 \mid S_1bS_1 \mid S_1S_1b$

$S_2 \to b \mid bS_2 \mid S_2b \mid aS_2S_2 \mid S_2aS_2 \mid SS_2a.$

G generates all strings over $\{a, b\}$ having an unequal number of a's and b's.

EXAMPLE 4.24

If L_1 and L_2 are the subsets of $\{a, b\}^*$, prove or disprove:

(a) If $L_1 \subseteq L_2$ and L_1 is not regular, then L_2 is not regular.

(b) If $L_1 \subseteq L_2$ and L_2 is not regular, then L_1 is not regular.

Solution

(a) Let $L_1 = \{a^n b^n \mid n \geq 1\}$. L_1 is not regular. Let $L_2 = \{a, b\}^*$. By Example 4.5, L_2 is regular. Hence (a) is not true.

(b) Let $L_2 = \{a^n b^n \mid n \geq 1\}$. It is not regular. But any finite subset is regular. Taking L_1 to be a finite subset of L_2, we disprove (b).

EXAMPLE 4.25

Show that the set of all non-palindromes over $\{a, b\}$ is a context-free language.

Solution

Let $w \in \{a, b\}^*$ be a non-palindrome. Then w may have the same symbol in the first and last places, same in the second place from the left and from the right, etc.; this pattern will not be there after a particular stage. The productions $S \to aSa \mid bSb \mid \Lambda$ may be used for fulfilling the palindrome-condition for the first and last few places. For violating the palindrome condition, the productions of the form $A \to aBb \mid bBa$ and $B \to aB \mid bB \mid \Lambda$ will be useful. So the required grammar is $G = (\{S, A, B\}, \{a, b\}, P, S)$ where P consists of

$$S \to aSa \mid bSb \mid A$$
$$A \to aBb \mid bBa$$
$$B \to aB \mid bB \mid \Lambda$$

SELF-TEST

Choose the correct answers to Questions 1–12:

1. For a grammar G with productions $S \to SS$, $S \to aSb$, $S \to bSa$, $S \to \Lambda$,

(a) $S \Rightarrow abba$ (b) $S \overset{*}{\Rightarrow} abba$

(c) $abba \notin L(G)$ (d) $S \overset{*}{\Rightarrow} aaa$

2. If $\alpha \overset{*}{\Rightarrow} \beta$ in a grammar G, then
(a) $\alpha \Rightarrow \beta$ (b) $\beta \Rightarrow \alpha$
(c) $\beta \overset{*}{\Rightarrow} \alpha$ (d) none of these

3. If $\alpha \rightarrow \beta$ is a production in a grammar G, then
(a) $\alpha\alpha \overset{*}{\Rightarrow} \beta\beta$ (b) $\alpha\alpha\beta \Rightarrow \beta\beta\alpha$
(c) $\alpha\alpha \Rightarrow \beta\alpha$ (d) $\alpha\alpha\alpha \overset{*}{\Rightarrow} \beta\beta\beta$

4. If a grammar G has three productions $S \rightarrow aSa \mid bsb \mid c$, then
(a) *abcba* and *bacab* $\in L(G)$ (b) *abcba* and *abcab* $\in L(G)$
(c) *accca* and *bcccb* $\in L(G)$ (d) *acccb* and *bccca* $\in L(G)$

5. The minimum number of productions for a grammar $G = (\{S\}, \{0, 1, 2, \ldots, 9\}, P, S)$ for generating $\{0, 1, 2, \ldots, 9\}$ is
(a) 9 (b) 10
(c) 1 (d) 2

6. If $G_1 = (N, T, P_1, S)$ and $G_2 = (N, T, P_2, S)$ and $P_1 \subseteq P_2$, then
(a) $L(G_1) \subseteq L(G_2)$ (b) $L(G_2) \subseteq L(G_1)$
(c) $L(G_1) = L(G_2)$ (d) none of these.

7. The regular grammar generating $\{a^n : n \geq 1\}$ is
(a) $(\{S\}, \{a\}, \{S \rightarrow aS\}, S)$
(b) $(\{S\}, \{a\}, \{S \rightarrow SS, S \rightarrow a\})$
(c) $(\{S\}, \{a\}, \{S \rightarrow aS\}, S)$
(d) $(\{S\}, \{a\}, \{S \rightarrow aS, S \rightarrow a, S)$

8. L = {theory, of, computer, science} can be generated by
(a) a regular grammar
(b) a context-free grammar but not a regular grammar
(c) a context-sensitive grammar but not a context-free grammar
(d) only by a type 0 grammar.

9. $\{a^n \mid n \geq 1\}$ is
(a) regular
(b) context-free but not regular
(c) context-sensitive but not context-free
(d) none of these.

10. $\{a^n b^n \mid n \geq 1\}$ is
(a) regular
(b) context-free but not regular
(c) context-sensitive but not context-free
(d) none of these.

11. $\{a^n b^n c^n \mid n \geq 1\}$ is
(a) regular
(b) context-free but not regular
(c) context-sensitive but not context-free
(d) none of these.

12. $\{a^n b^n c^m \mid n, m \geq 1\}$ is
(a) regular
(b) context-free but not regular
(c) context-sensitive but not context-free
(d) none of these.

State whether the following Statements 13–20 are true or false:

13. In a grammar $G = (V_N, \Sigma, P, S)$, V_N and Σ are finite but P can be infinite.

14. Two grammars of different types can generate the same language.

15. If $G = (V_N, \Sigma, P, S)$ and $P \neq \emptyset$, then $L(G) \neq \emptyset$.

16. If a grammar G has three productions, i.e. $S \to AA$, $A \to aa$, $A \to bb$, then $L(G)$ is finite.

17. If $L_1 = \{a^n b^m \mid m, n \geq 1\}$ and $L_2 = \{b^m c^p \mid m, p \geq 1\}$, then $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$.

18. If a grammar G has productions $S \to aS \mid bS \mid a$, then $L(G)$ = the set of all strings over $\{a, b\}$ ending in a.

19. The language $\{a^n bc^n \mid n \geq 1\}$ is regular.

20. If the productions of G are $S \to aS \mid Sb \mid a \mid b$, then $abab \in L(G)$.

EXERCISES

4.1 Find the language generated by the following grammars:
(a) $S \to 0S1 \mid 0A1$, $A \to 1A \mid 1$
(b) $S \to 0S1 \mid 0A \mid 0 \mid \mid 1B \mid 1$, $A \to 0A \mid 0$, $B \to 1B \mid 1$
(c) $S \to 0SBA \mid 01A$, $AB \to BA$, $1B \to 11$, $1A \to 10$, $0A \to 00$
(d) $S \to 0S1 \mid 0A1$, $A \to 1A0 \mid 10$
(e) $S \to 0A \mid 1S \mid 0 \mid 1$, $A \to 1A \mid 1S \mid 1$

4.2 Construct the grammar, accepting each of the following sets:
(a) The set of all strings over $\{0, 1\}$ consisting of an equal number of 0's and 1's.
(b) $\{0^n 1^m 0^m 1^n \mid m, n \geq 1\}$
(c) $\{0^n 1^{2n} \mid n \geq 1\}$
(d) $\{0^n 1^n \mid n \geq 1\} \cup \{1^m 0^m \mid m \geq 1\}$
(e) $\{0^n 1^m 0^n \mid m, n \geq 1\} \cup \{0^n 1^m 2^m \mid m, n \geq 1\}$.

4.3 Test whether 001100, 001010, 01010 are in the language generated by the grammar given in Exercise 4.1(b).

4.4 Let $G = (\{A, B, S\}, \{0, 1\}, P, S)$, where P consists of $S \to 0AB$, $A_0 \to S0B$, $A_1 \to SB1$, $B \to SA$, $B \to 01$. Show that $L(G) = \emptyset$.

4.5 Find the language generated by the grammar $S \to AB$, $A \to A1 \mid 0$, $B \to 2B \mid 3$. Can the above language be generated by a grammar of higher type?

4.6 State whether the following statements are true or false. Justify your answer with a proof or a counter-example.
(a) If G_1 and G_2 are equivalent, then they are of the same type.
(b) If L is a finite subset of Σ^*, then L is a context-free language.
(c) If L is a finite subset of Σ^*, then L is a regular language.

4.7 Show that $\{a^{n^2} \mid n \geq 1\}$ is generated by the grammar $S \to a$, $S \to A_3A_4$, $A_3 \to A_1A_3A_2$, $A_3 \to A_1A_2$, $A_1A_2 \to aA_2A_1$, $A_1a \to aA_1$, $A_2a \to aA_2$, $A_1A_4 \to A_4a$, $A_2A_4 \to A_5a$, $A_2A_5 \to A_5a$, $A_5 \to a$.

4.8 Construct (i) a context-sensitive but not context-free grammar, (ii) a context-free but not regular grammar, and (iii) a regular grammar to generate $\{a^n \mid n \geq 1\}$.

4.9 Construct a grammar which generates all even integers up to 998.

4.10 Construct context-free grammars to generate the following:
(a) $\{0^m1^n \mid m \neq n,\ m,\ n \geq 1\}$.
(b) $\{a^lb^mc^n \mid$ one of l, m, n equals 1 and the remaining two are equal$\}$.
(c) $\{0^m1^n \mid 1 \leq m \leq n\}$.
(d) $\{a^lb^mc^n \mid l + m = n\}$.
(e) The set of all strings over $\{0, 1\}$ containing twice as many 0's as 1's.

4.11. Construct regular grammars to generate the following:
(a) $\{a^{2n} \mid n \geq 1\}$.
(b) The set of all strings over $\{a, b\}$ ending in a.
(c) The set of all strings over $\{a, b\}$ beginning with a.
(d) $\{a^lb^mc^n \mid l, m, n \geq 1\}$.
(e) $\{(ab)^n \mid n \geq 1\}$.

4.12. Is $\underset{G}{\Rightarrow}$ an equivalence relation on $(V_N \cup \Sigma)^*$?

4.13. Show that $G_1 = (\{S\}, \{a, b\}, P_1, S)$, where $P_1 = \{S \to aSb \mid ab\}$ is equivalent to $G_2 = (\{S, A, B, C\}, \{a, b\}, P_2, S)$. Here P_2 consists of $S \to AC$, $C \to SB$, $S \to AB$, $A \to a$, $B \to b$.

4.14. If each production in a grammar G has some variable on its right-hand side, what can you say about $L(G)$?

4.15. Show that $\{abc, bca, cab\}$ can be generated by a regular grammar whose terminal set is $\{a, b, c\}$.

4.16. Construct a grammar to generate $\{(ab)^n \mid n \geq 1\} \cup \{(ba)^n \mid n \geq 1\}$.

4.17. Show that a grammar consisting of productions of the form $A \to xB \mid y$, where x, y are in Σ^* and A, $B \in V_N$, is equivalent to a regular grammar.

5 Regular Sets and Regular Grammars

In this chapter, we first define regular expressions as a means of representing certain subsets of strings over Σ and prove that regular sets are precisely those accepted by finite automata or transition systems. We use pumping lemma for regular sets to prove that certain sets are not regular. We then discuss closure properties of regular sets. Finally, we give the relation between regular sets and regular grammars.

5.1 REGULAR EXPRESSIONS

The regular expressions are useful for representing certain sets of strings in an algebraic fashion. Actually these describe the languages accepted by finite state automata.

We give a formal recursive definition of regular expressions over Σ as follows:

1. Any terminal symbol (i.e. an element of Σ), Λ and $\emptyset$ are regular expressions. When we view a in Σ as a regular expression, we denote it by $\mathbf{a}$.
2. The union of two regular expressions $\mathbf{R}_1$ and $\mathbf{R}_2$, written as $\mathbf{R}_1 + \mathbf{R}_2$, is also a regular expression.
3. The concatenation of two regular expressions $\mathbf{R}_1$ and $\mathbf{R}_2$, written as $\mathbf{R}_1\mathbf{R}_2$, is also a regular expression.
4. The iteration (or closure) of a regular expression $\mathbf{R}$, written as $\mathbf{R}^*$, is also a regular expression.
5. If $\mathbf{R}$ is a regular expression, then $(\mathbf{R})$ is also a regular expression.
6. The regular expressions over Σ are precisely those obtained recursively by the application of the rules 1–5 once or several times.

Notes: (1) We use **x** for a regular expression just to distinguish it from the symbol (or string) x.

(2) The parentheses used in Rule 5 influence the order of evaluation of a regular expression.

(3) In the absence of parentheses, we have the hierarchy of operations as follows: iteration (closure), concatenation, and union. That is, in evaluating a regular expression involving various operations, we perform iteration first, then concatenation, and finally union. This hierarchy is similar to that followed for arithmetic expressions (exponentiation, multiplication and addition).

Definition 5.1 Any set represented by a regular expression is called a *regular set.*

If, for example, $a, b \in \Sigma$, then (i) **a** denotes the set $\{a\}$, (ii) **a** + **b** denotes $\{a, b\}$, (iii) **ab** denotes $\{ab\}$, (iv) **a*** denotes the set $\{\Lambda, a, aa, aaa, \ldots\}$ and (v) **(a + b)*** denotes $\{a, b\}^*$.

The set represented by R is denoted by $L(\mathbf{R})$.

Now we shall explain the evaluation procedure for the three basic operations. Let $\mathbf{R}_1$ and $\mathbf{R}_2$ denote any two regular expressions. Then (i) a string in $L(\mathbf{R}_1 + \mathbf{R}_2)$ is a string from $\mathbf{R}_1$ or a string from $\mathbf{R}_2$; (ii) a string in $L(\mathbf{R}_1\mathbf{R}_2)$ is a string from $\mathbf{R}_1$ followed by a string from $\mathbf{R}_2$, and (iii) a string in $L(\mathbf{R}^*)$ is a string obtained by concatenating n elements for some $n \geq 0$. Consequently, (i) the set represented by $\mathbf{R}_1 + \mathbf{R}_2$ is the union of the sets represented by $\mathbf{R}_1$ and $\mathbf{R}_2$, (ii) the set represented by $\mathbf{R}_1\mathbf{R}_2$ is the concatenation of the sets represented by $\mathbf{R}_1$ and $\mathbf{R}_2$. (Recall that the concatenation AB of sets A and B of strings over Σ is given by $AB = \{w_1w_2 | w_1 \in A, w_2 \in B\}$, and (iii) the set represented by $\mathbf{R}^*$ is $\{w_1w_2 \ldots w_n | w_i$ is in the set represented by $\mathbf{R}$ and $n \geq 0.\}$ Hence,

$$L(\mathbf{R}_1 + \mathbf{R}_2) = L(\mathbf{R}_1) \cup L(\mathbf{R}_2), \qquad L(\mathbf{R}_1\mathbf{R}_2) = L(\mathbf{R}_1)L(\mathbf{R}_2)$$
$$L(\mathbf{R}^*) = (L(\mathbf{R}))^*$$

Also,

$$L(\mathbf{R}^*) = (L(\mathbf{R}))^* = \bigcup_{n=0}^{\infty} L(\mathbf{R})^n$$
$$L(\emptyset) = \emptyset, \qquad L(\mathbf{a}) = \{a\}.$$

Note: By the definition of regular expressions, the class of regular sets over Σ is closed under union, concatenation and closure (iteration) by the conditions 2, 3, 4 of the definition.

EXAMPLE 5.1

Describe the following sets by regular expressions: (a) $\{101\}$, (b) $\{abba\}$, (c) $\{01, 10\}$, (d) $\{\Lambda, ab\}$, (e) $\{abb, a, b, bba\}$, (f) $\{\Lambda, 0, 00, 000, \ldots\}$, and (g) $\{1, 11, 111, \ldots\}$.

Solution

(a) Now, $\{1\}$, $\{0\}$ are represented by **1** and **0**, respectively. 101 is obtained by concatenating 1, 0 and 1. So, $\{101\}$ is represented by **101**.

(b) **abba** represents $\{abba\}$.
(c) As $\{01, 10\}$ is the union of $\{01\}$ and $\{10\}$, we have $\{01, 10\}$ represented by **01 + 10**.
(d) The set $\{\Lambda, ab\}$ is represented by Λ + **ab**.
(e) The set $\{abb, a, b, bba\}$ is represented by **abb + a + b + bba**.
(f) As $\{\Lambda, 0, 00, 000, \ldots\}$ is simply $\{0\}^*$, it is represented by **0***.
(g) Any element in $\{1, 11, 111, \ldots\}$ can be obtained by concatenating 1 and any element of $\{1\}^*$. Hence **1(1)*** represents $\{1, 11, 111, \ldots\}$.

EXAMPLE 5.2

Describe the following sets by regular expressions:
(a) L_1 = the set of all strings of 0's and 1's ending in 00.
(b) L_2 = the set of all strings of 0's and 1's beginning with 0 and ending with 1.
(c) L_3 = $\{\Lambda, 11, 1111, 111111, \ldots\}$.

Solution

(a) Any string in L_1 is obtained by concatenating any string over $\{0, 1\}$ and the string 00. $\{0, 1\}$ is represented by **0 + 1**. Hence L_1 is represented by **(0 + 1)* 00**.
(b) As any element of L_2 is obtained by concatenating 0, any string over $\{0, 1\}$ and 1, L_2 can be represented by **0(0 + 1)* 1**.
(c) Any element of L_3 is either Λ or a string of even number of 1's, i.e. a string of the form $(11)^n$, $n \geq 0$. So L_3 can be represented by **(11)***.

5.1.1 IDENTITIES FOR REGULAR EXPRESSIONS

Two regular expressions **P** and **Q** are equivalent (we write **P = Q**) if **P** and **Q** represent the same set of strings.

We now give the identities for regular expressions; these are useful for simplifying regular expressions.

I_1 $\quad \emptyset + \mathbf{R} = \mathbf{R}$

I_2 $\quad \emptyset\mathbf{R} = \mathbf{R}\emptyset = \emptyset$

I_3 $\quad \Lambda\mathbf{R} = \mathbf{R}\Lambda = \mathbf{R}$

I_4 $\quad \Lambda^* = \Lambda$ and $\emptyset^* = \Lambda$

I_5 $\quad \mathbf{R} + \mathbf{R} = \mathbf{R}$

I_6 $\quad \mathbf{R}^*\mathbf{R}^* = \mathbf{R}^*$

I_7 $\quad \mathbf{R}\mathbf{R}^* = \mathbf{R}^*\mathbf{R}$

I_8 $\quad (\mathbf{R}^*)^* = \mathbf{R}^*$

I_9 $\quad \Lambda + \mathbf{R}\mathbf{R}^* = \mathbf{R}^* = \Lambda + \mathbf{R}^*\mathbf{R}$

I_{10} $\quad (\mathbf{PQ})^*\mathbf{P} = \mathbf{P}(\mathbf{QP})^*$

I_{11} $\quad (\mathbf{P} + \mathbf{Q})^* = (\mathbf{P}^*\mathbf{Q}^*)^* = (\mathbf{P}^* + \mathbf{Q}^*)^*$

I_{12} $\quad (\mathbf{P} + \mathbf{Q})\mathbf{R} = \mathbf{PR} + \mathbf{QR}$ and $\mathbf{R}(\mathbf{P} + \mathbf{Q}) = \mathbf{RP} + \mathbf{RQ}$

Note: By the 'set **P**' we mean the set represented by the regular expression **P**.

The following theorem is very much useful in simplifying regular expressions (i.e. replacing a given regular expression **P** by a simpler regular expression equivalent to **P**).

Theorem 5.1 (Arden's theorem) Let **P** and **Q** be two regular expressions over Σ. If **P** does not contain Λ, then the following equation in **R**, namely

$$\mathbf{R} = \mathbf{Q} + \mathbf{RP} \tag{5.1}$$

has a unique solution (i.e. one and only one solution) given by $\mathbf{R} = \mathbf{QP}^*$.

Proof $\mathbf{Q} + (\mathbf{QP}^*)\mathbf{P} = \mathbf{Q}(\Lambda + \mathbf{P}^*\mathbf{P}) = \mathbf{QP}^*$ by I_9

Hence (5.1) is satisfied when $\mathbf{R} = \mathbf{QP}^*$. This means $\mathbf{R} = \mathbf{QP}^*$ is a solution of (5.1).

To prove uniqueness, consider (5.1). Here, replacing **R** by $\mathbf{Q} + \mathbf{RP}$ on the R.H.S., we get the equation

$$\begin{aligned}
\mathbf{Q} + \mathbf{RP} &= \mathbf{Q} + (\mathbf{Q} + \mathbf{RP})\mathbf{P} \\
&= \mathbf{Q} + \mathbf{QP} + \mathbf{RPP} \\
&= \mathbf{Q} + \mathbf{QP} + \mathbf{RP}^2 \\
&= \mathbf{Q} + \mathbf{QP} + \mathbf{QP}^2 + \cdots + \mathbf{QP}^i + \mathbf{RP}^{i+1} \\
&= \mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^i) + \mathbf{RP}^{i+1}
\end{aligned}$$

From (5.1),

$$\mathbf{R} = \mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^i) + \mathbf{RP}^{i+1} \quad \text{for } i \geq 0 \tag{5.2}$$

We now show that any solution of (5.1) is equivalent to $\mathbf{QP}^*$. Suppose **R** satisfies (5.1), then it satisfies (5.2). Let w be a string of length i in the set **R**. Then w belongs to the set $Q(\Lambda + \mathbf{P} + \mathbf{P}^2 + \ldots + \mathbf{P}^i) + \mathbf{RP}^{i+1}$. As **P** does not contain Λ, $\mathbf{RP}^{i+1}$ has no string of length less than $i + 1$ and so w is not in the set $\mathbf{RP}^{i+1}$. This means that w belongs to the set $Q(\Lambda + \mathbf{P} + \mathbf{P}^2 + \ldots + \mathbf{P}^i)$, and hence to $\mathbf{QP}^*$.

Consider a string w in the set $\mathbf{QP}^*$. Then w is in the set $\mathbf{QP}^k$ for some $k \geq 0$, and hence in $Q(\Lambda + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^k)$. So w is on the R.H.S. of (5.2). Therefore, w is in **R** (L.H.S. of (5.2)). Thus **R** and $\mathbf{QP}^*$ represent the same set. This proves the uniqueness of the solution of (5.1). ∎

Note: Henceforth in this text, the regular expressions will be abbreviated r.e.

Example 5.3

(a) Give an r.e. for representing the set L of strings in which every 0 is immediately followed by at least two 1's.

(b) Prove that the regular expression $\mathbf{R} = \Lambda + \mathbf{1}^*(\mathbf{011})^*(\mathbf{1}^*\ (\mathbf{011})^*)^*$ also describes the same set of strings.

Solution

(a) If w is in L, then either (a) w does not contain any 0, or (b) it contains a 0 preceded by 1 and followed by 11. So w can be written as $w_1 w_2 \ldots w_n$, where each w_i is either 1 or 011. So L is represented by the r.e. **(1 + 011)***.

(b) $\mathbf{R} = \Lambda + \mathbf{P}_1\mathbf{P}_1^*$, where $\mathbf{P}_1 = \mathbf{1^*(011)^*}$

$= \mathbf{P}_1^*$ using I_9

$= \mathbf{(1^*(011)^*)^*}$

$= (\mathbf{P}_2^*\mathbf{P}_3^*)^*$ letting $\mathbf{P}_2 = \mathbf{1}$, $\mathbf{P}_3 = \mathbf{011}$

$= (\mathbf{P}_2 + \mathbf{P}_3)^*$ using I_{11}

$= \mathbf{(1 + 011)^*}$

EXAMPLE 5.4

Prove $\mathbf{(1 + 00^*1) + (1 + 00^*1)(0 + 10^*1)^* (0 + 10^*1) = 0^*1(0 + 10^*1)^*}$.

Solution

L.H.S. $= \mathbf{(1 + 00^*1)}\,(\Lambda + \mathbf{(0 + 10^*1)^*\,(0 + 10^*1)}\Lambda$ using I_{12}

$= \mathbf{(1 + 00^*1)\ (0 + 10^*1)^*}$ using I_9

$= (\Lambda + \mathbf{00^*)1\ (0 + 10^*1)^*}$ using I_{12} for $\mathbf{1 + 00^*1}$

$= \mathbf{0^*1(0 + 10^*1)^*}$ using I_9

$=$ R.H.S.

5.2 FINITE AUTOMATA AND REGULAR EXPRESSIONS

In this section we study regular expressions and their representation.

5.2.1 TRANSITION SYSTEM CONTAINING Λ-MOVES

The transition systems can be generalized by permitting Λ-transitions or Λ-moves which are associated with a null symbol Λ. These transitions can occur when no input is applied. But it is possible to convert a transition system with Λ-moves into an equivalent transition system without Λ-moves. We shall give a simple method of doing it with the help of an example.

Suppose we want to replace a Λ-move from vertex v_1 to vertex v_2. Then we proceed as follows:

Step 1 Find all the edges starting from v_2.

Step 2 Duplicate all these edges starting from v_1, without changing the edge labels.

Step 3 If v_1 is an initial state, make v_2 also as initial state.

Step 4 If v_2 is a final state, make v_1 also as the final state.

EXAMPLE 5.5

Consider a finite automaton, with Λ-moves, given in Fig. 5.1. Obtain an equivalent automaton without Λ-moves.

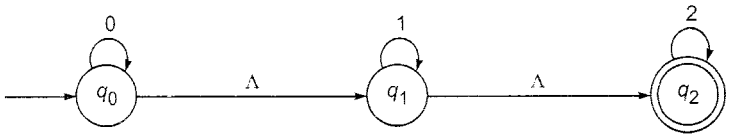


Fig. 5.1 Finite automaton of Example 5.5.

Solution

We first eliminate the Λ-move from q_0 to q_1 to get Fig. 5.2(a). q_1 is made an initial state. Then we eliminate the Λ-move from q_0 to q_2 in Fig. 5.2(a) to get Fig. 5.2(b). As q_2 is a final state, q_0 is also made a final state. Finally, the Λ-move from q_1 to q_2 is eliminated in Fig. 5.2(c).

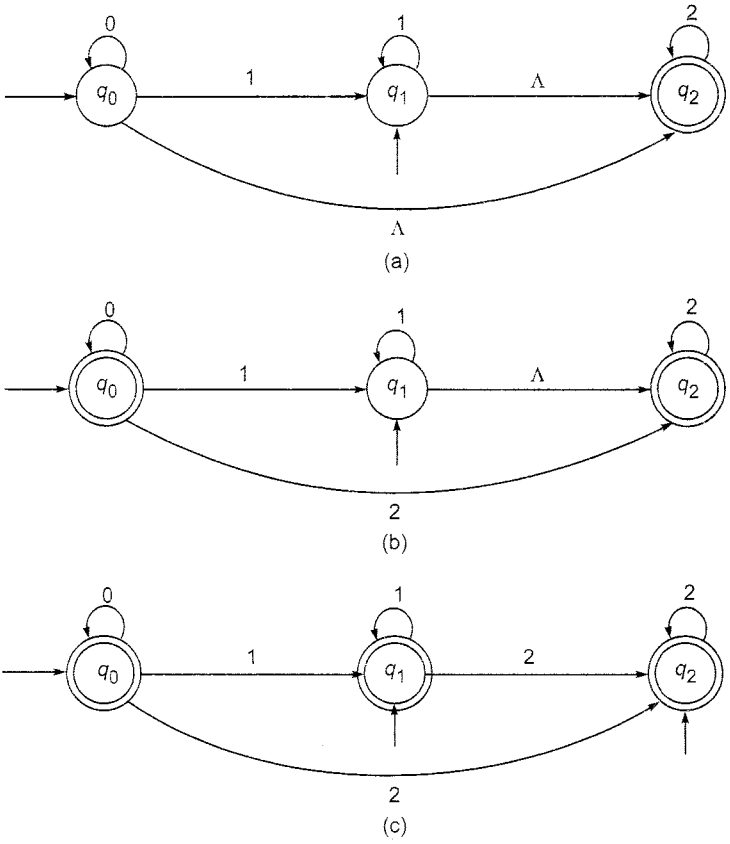


Fig. 5.2 Transition system for Example 5.5, without Λ-moves.

EXAMPLE 5.6

Consider a graph (i.e. transition system), containing a Λ-move, given in Fig. 5.3. Obtain an equivalent graph (i.e. transition system) without Λ-moves.

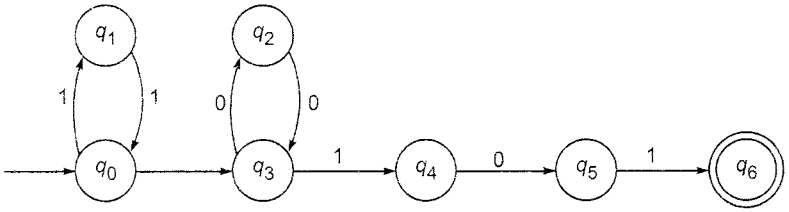


Fig. 5.3 Finite automaton of Example 5.6.

Solution

There is a Λ-move from q_0 to q_3. There are two edges, one from q_3 to q_2 with label 0 and another from q_3 to q_4 with label 1. We duplicate these edges from q_0. As q_0 is an initial state, q_3 is made an initial state. The resulting transition graph is given in Fig. 5.4.

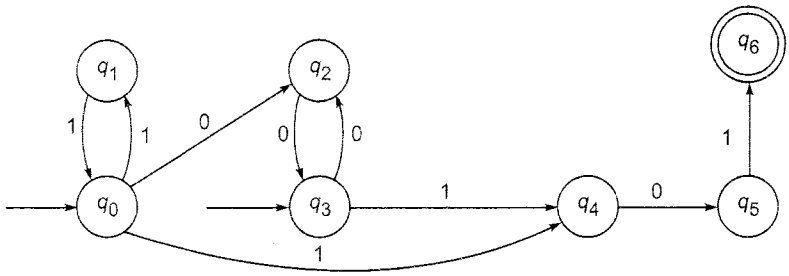


Fig. 5.4 Transition system for Example 5.6, without Λ-moves.

5.2.2 NDFAs WITH Λ-MOVES AND REGULAR EXPRESSIONS

In this section, we prove that every regular expression is recognized by a nondeterministic finite automaton (NDFA) with Λ-moves.

Theorem 5.2 (Kleene's theorem) If **R** is a regular expression over Σ representing $L \subseteq \Sigma^*$, then there exists an NDFA M with Λ-moves such that $L = T(M)$.

Proof The proof is by the principle of induction on the total number of characters in **R**. By 'character' we mean the elements of Σ, Λ, ∅, * and +. For example, if $\mathbf{R} = \Lambda + \mathbf{10^*11^*0}$, the characters are Λ, +, 1, 0, *, 1, 1, *, 0, and the number of characters is 9.

Let $L(\mathbf{R})$ denote the set represented by **R**.

Basis. Let the number of characters in **R** be 1. Then **R** = Λ, or **R** = ∅, or **R** = a_i, $a_i \in \Sigma$. The transition systems given in Fig. 5.5 will recognize these regular expressions.

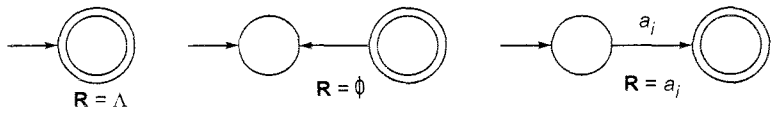


Fig. 5.5 Transition systems for recognizing elementary regular sets.

Induction step. Assume that the theorem is true for regular expressions having n characters. Let **R** be a regular expression having $n + 1$ characters. Then,

$$\mathbf{R} = \mathbf{P} + \mathbf{Q} \quad \text{or} \quad \mathbf{R} = \mathbf{PQ} \quad \text{or} \quad \mathbf{R} = \mathbf{P}^*$$

according as the last operator in **R** is +, product or closure. Also **P** and **Q** are regular expressions having n characters or less. By induction hypothesis, $L(\mathbf{P})$ and $L(\mathbf{Q})$ are recognized by M_1 and M_2 where M_1 and M_2 are NDFAs with Λ-moves, such that $L(\mathbf{P}) = T(M_1)$ and $L(\mathbf{Q}) = T(M_2)$. M_1 and M_2 are represented in Fig. 5.6.

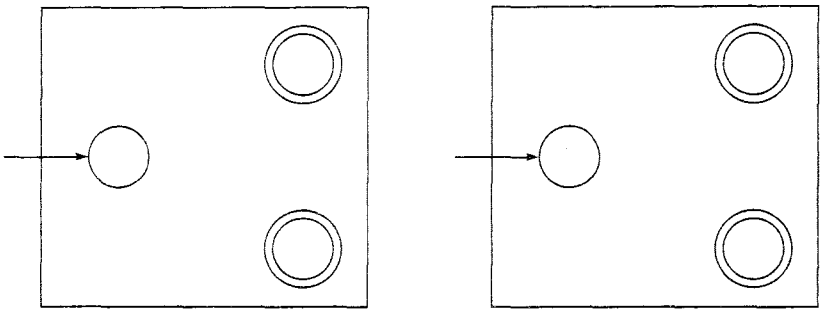


Fig. 5.6 Nondeterministic finite automata M_1 and M_2.

The initial state and the final states of M_1 and M_2 are represented in the usual way.

Case 1 **R** = **P** + **Q**. In this case we construct an NDFA M with Λ-moves that accepts $L(\mathbf{P} + \mathbf{Q})$ as follows: q_0 is the initial state of M, q_0 not in M_1 or M_2. q_f is the final state of M: once again q_f not in M_1 or M_2. M contains all the states of M_1 and M_2 and also their transitions. We add additional Λ-transitions from q_0 to the initial states of M_1 and M_2 and from the final states of M_1 and M_2 to q_f. The NDFA M is as in Fig. 5.7. It is easy to see that $T(M) = T(M_1) \cup T(M_2) = L(\mathbf{P} + \mathbf{Q})$.

Case 2 **R** = **PQ**. In this case we introduce q_0 as the initial state of M and q_f as the final state of M, both q_0, q_f not in M_1 or M_2. New Λ-transitions are added between q_0 and the initial state of M_1, between final states of M_1 and the initial state of M_2, and between final states of M_2 and the final state q_f of M. See Fig. 5.8.

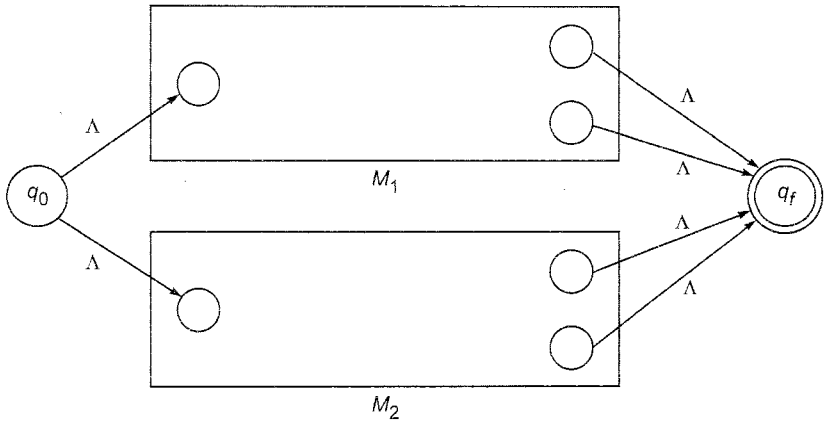


Fig. 5.7 NDFA accepting $L(\mathbf{P} + \mathbf{Q})$.

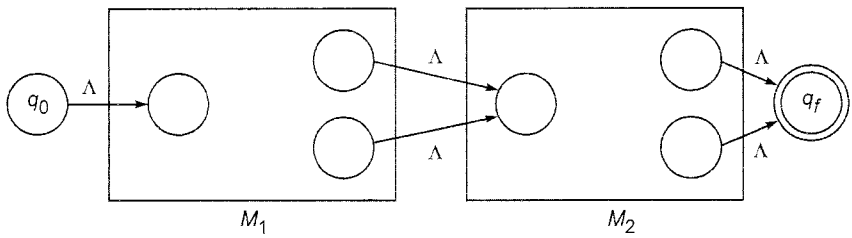


Fig. 5.8 NDFA accepting $L(\mathbf{PQ})$.

Case 3 $\mathbf{R} = (\mathbf{P})^*$. In this case, q_0, q and q_f are introduced. New Λ-transitions are introduced from q_0 to q, q to q_f, q to the initial state of M_1 and from the final states of M_1 to q. See Fig. 5.9.

Thus in all the cases, there exists an NDFA M with Λ-moves, accepting the regular expression $\mathbf{R}$ with $n + 1$ characters. By the principle of induction, this theorem is true for all regular expressions. ∎

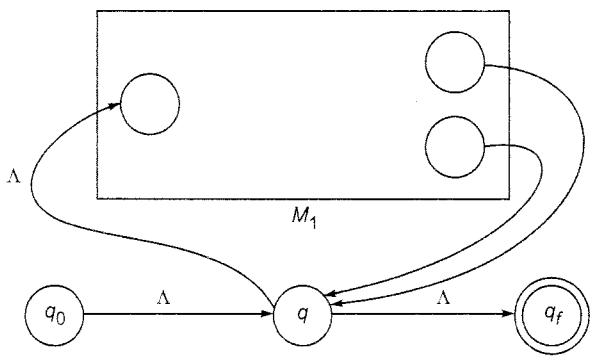


Fig. 5.9 NDFA accepting $L(\mathbf{P}^*)$.

Theorem 5.1 gives a method of constructing NDFAs accepting **P** + **Q**, **PQ** and **P*** using the NDFAs corresponding to **P** and **Q**. In the later sections we give a method of converting NDFA M with Λ-moves into an NDFA M_1 without Λ-moves and then into a DFA M_2 such that $T(M) = T(M_1) = T(M_2)$. Thus, if a regular expression **P** is given, we can construct a DFA accepting $L(\mathbf{P})$.

The following theorem is regarding the converse. Both the Theorems 5.2 and 5.3 prove the equivalence of regular expressions or regular sets and the sets accepted by deterministic finite automata.

Theorem 5.3 Any set L accepted by a finite automaton M is represented by a regular expression.

Proof Let

$$M = (\{q_1 \ldots q_m\}, \Sigma, \delta, q_1, F)$$

The construction that we give can be better understood in terms of the state diagram of M. If a string $w \in \Sigma^*$ is accepted by M, then there is a path from q_1 to some final state with path value w. So to each final state, say q_j, there corresponds a subset of Σ^* consisting of path values of paths from q_0 to q_j. As $T(M)$ is the union of such subsets of Σ^*, it is enough to represent them by regular expressions. So the main part of the proof lies in the construction of subsets of path values of paths from the state q_i to the state q_j.

Let P_{ij}^k denote the set of path values of paths from q_i to q_j whose intermediate vertices lie in $\{q_1, \ldots, q_k\}$. We construct P_{ij}^k for $k = 0, 1, \ldots, n$ recursively as follows:

$$P_{ij}^0 = \{a \in \Sigma \mid \delta(q_i, a) = q_j\} \tag{5.3}$$

$$P_{ii}^0 = \{a \in \Sigma \mid \delta(q_i, a) = q_i\} \cup \{\Lambda\} \tag{5.4}$$

$$P_{ij}^k = P_{ik}^{k-1}(P_{kk}^{k-1})^* P_{kj}^{k-1} \cup P_{ij}^{k-1} \tag{5.5}$$

In terms of the state diagram, the construction can be understood better. P_{ij}^0 simply denotes the set of path values (i.e. labels) of edges from q_i to q_j. In P_{ii}^0 we include Λ in addition to labels of self-loops from q_i. This explains (5.3) and (5.4).

Consider a path from q_i to q_j whose intermediate vertices lie in $\{q_1, \ldots, q_k\}$. If the path does not pass through q_k, then its path value lies in P_{ij}^{k-1}. Otherwise, the path passes through q_k possibly more than once. The path can be split into several paths with path values $w_1, w_2 \ldots w_l$ as in Fig. 5.10. $w = w_1 w_2 \ldots w_l$. w_1 is the path value of the path from q_i to q_k (without passing through q_k, i.e. q_k is not an intermediate vertex). $w_2, \ldots, w_{l-1}$ are the path values of paths from q_k to itself without passing through q_k. w_l is the path value of the path from q_k to q_j without passing through q_k. So w_1 is in P_{ik}^{k-1}, $w_2, \ldots, w_{l-1}$ are in (P_{kk}^{k-1}), and w_l is in P_{kj}^{k-1}. This explains (5.5).

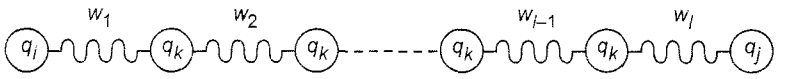


Fig. 5.10 A path from q_i to q_j.

We prove that the sets introduced by (5.3)–(5.5) are represented by regular expressions by induction on k (for all i and j). P_{ij}^0 is a finite subset of Σ, say $\{a_1, \ldots, a_r\}$. Then, P_{ij}^0 is represented by $\mathbf{P}_{ij}^0 = a_1 + a_2 + \cdots + a_r$. Similarly, we can construct $\mathbf{P}_{ii}^0$ representing P_{ii}^0. Thus, there is basis for induction.

Let us assume the result for $k - 1$, i.e. P_{ij}^{k-1} is represented by a regular expression $\mathbf{P}_{ij}^{k-1}$ for all i and j. From (5.5), we have $P_{ij}^k = P_{ik}^{k-1}(P_{kk}^{k-1})^* P_{kj}^{k-1} \cup P_{ij}^{k-1}$. So it is obvious that P_{ij}^k is represented by $\mathbf{P}_{ij}^k = \mathbf{P}_{ik}^{k-1}(\mathbf{P}_{kk}^{k-1})^* \mathbf{P}_{kj}^{k-1} \cup \mathbf{P}_{ij}^{k-1}$. Therefore, the result is true for all k. By the principle of induction, the sets constructed by (5.3)–(5.5) are represented by regular expressions.

As $Q = \{q_1, \ldots, q_m\}$, $\mathbf{P}_{1j}^m$ denotes the set of path values of all paths from q_1 to q_j. If $F = \{q_{f_1}, \ldots, q_{f_n}\}$, then $T(M) = \bigcup_{j=1}^{n} P_{1f_j}^m$. So $T(M)$ is represented by the regular expression $\mathbf{P}_{1f_1}^m + \cdots + \mathbf{P}_{1f_n}^m$. Thus, $L = T(M)$ is represented by a regular expression.

Note: P_{ij}^0 and P_{ii}^0 are the subsets of $\Sigma \cup \{\Lambda\}$, and so they are finite sets. So every P_{ij}^k is obtained by applying union, concatenation and closure to the set of all singletons in $\Sigma \cup \{\Lambda\}$. Using this we prove Kleene's theorem (Theorem 5.4) at the end of this section. Kleene's theorem characterizes the regular sets in terms of subsets of Σ and operations (union, concatenation, closure) on singletons in $\Sigma \cup \{\Lambda\}$.

5.2.3 Conversion of Nondeterministic Systems to Deterministic Systems

The construction we are going to give is similar to the construction of a DFA equivalent to an NDFA and involves three steps.

Step 1 Convert the given transition system into state transition table where each state corresponds to a row and each input symbol corresponds to a column.

Step 2 Construct the successor table which lists the subsets of states reachable from the set of initial states. Denote this collection of subsets by Q'.

Step 3 The transition graph given by the successor table is the required deterministic system. The final states contain some final state of NDFA. If possible, reduce the number of states.

Note: The construction is similar to that given in Section 3.7 for automata except for the initial step. In the earlier method for automata, we started with $[q_0]$. Here we start with the set of all initial states. The other steps are similar.

EXAMPLE 5.7

Obtain the deterministic graph (system) equivalent to the transition system given in Fig. 5.11.

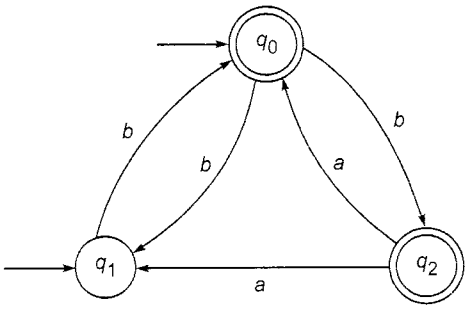


Fig. 5.11 Nondeterministic transition system of Example 5.7.

Solution

We construct the transition table corresponding to the given nondeterministic system. It is given in Table 5.1.

TABLE 5.1 Transition Table for Example 5.7

State/Σ	*a*	*b*
$\rightarrow(q_0)$		q_1, q_2
q_1		q_0
(q_2)	q_0, q_1	

We construct the successor table by starting with $[q_0, q_1]$. From Table 5.1 we see that $[q_0, q_1, q_2]$ is reachable from $[q_0, q_1]$ by a b-path. There are no a-paths from $[q_0, q_1]$. Similarly, $[q_0, q_1]$ is reachable from $[q_0, q_1, q_2]$ by an a-path and $[q_0, q_1, q_2]$ is reachable from itself. We proceed with the construction for all the elements in Q'.

We terminate the construction when all the elements of Q' appear in the successor table. Table 5.2 gives the successor table. From the successor table it is easy to construct the deterministic transition system described by Fig. 5.12

TABLE 5.2 Deterministic Transition Table for Example 5.7

Q	*a*	*b*
$[q_0, q_1]$	$\emptyset$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1]$	$[q_0, q_1, q_2]$
$\emptyset$	$\emptyset$	$\emptyset$

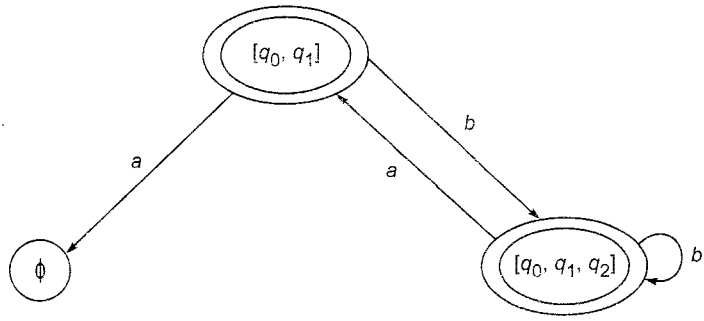


Fig. 5.12 Deterministic transition system for Example 5.7.

as q_0 and q_2 are the final states of the nondeterministic system $[q_0, q_1]$ and $[q_0, q_1, q_2]$ are the final states of the deterministic system.

5.2.4 Algebraic Method Using Arden's Theorem

The following method is an extension of the Arden's theorem (Theorem 5.1). This is used to find the r.e. recognized by a transition system.

The following assumptions are made regarding the transition system:

(i) The transition graph does not have Λ-moves.
(ii) It has only one initial state, say v_1.
(iii) Its vertices are $v_1 \ldots v_n$.
(iv) $\mathbf{V}_i$ the r.e. represents the set of strings accepted by the system even though v_i is a final state.
(v) α_{ij} denotes the r.e. representing the set of labels of edges from v_i to v_j. When there is no such edge, $\alpha_{ij} = \emptyset$. Consequently, we can get the following set of equations in $\mathbf{V}_1 \ldots \mathbf{V}_n$:

$$\mathbf{V}_1 = \mathbf{V}_1\alpha_{11} + \mathbf{V}_2\alpha_{21} + \cdots + \mathbf{V}_n\alpha_{n1} + \Lambda$$
$$\mathbf{V}_2 = \mathbf{V}_1\alpha_{12} + \mathbf{V}_2\alpha_{22} + \cdots + \mathbf{V}_n\alpha_{n2}$$
$$\vdots$$
$$\mathbf{V}_n = \mathbf{V}_1\alpha_{1n} + \mathbf{V}_2\alpha_{2n} + \cdots + \mathbf{V}_n\alpha_{nn}$$

By repeatedly applying substitutions and Theorem 5.1 (Arden's theorem), we can express $\mathbf{V}_i$ in terms of α_{ij}'s.

For getting the set of strings recognized by the transition system, we have to take the 'union' of all $\mathbf{V}_i$'s corresponding to final states.

EXAMPLE 5.8

Consider the transition system given in Fig. 5.13. Prove that the strings recognized are **(a + a(b + aa)*b)* a(b + aa)* a**.

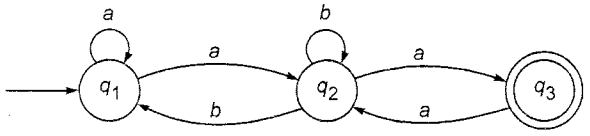


Fig. 5.13 Transition system of Example 5.8.

Solution

We can directly apply the above method since the graph does not contain any Λ-move and there is only one initial state.

The three equations for q_1, q_2 and q_3 can be written as

$$\mathbf{q_1 = q_1a + q_2b + \Lambda}, \qquad \mathbf{q_2 = q_1a + q_2b + q_3a}, \qquad \mathbf{q_3 = q_2a}$$

It is necessary to reduce the number of unknowns by repeated substitution. By substituting q_3 in the q_2-equation, we get by applying Theorem 5.1

$$\begin{aligned} \mathbf{q_2} &= \mathbf{q_1a + q_2b + q_2aa} \\ &= \mathbf{q_1a + q_2(b + aa)} \\ &= \mathbf{q_1a(b + aa)^*} \end{aligned}$$

Substituting $\mathbf{q_2}$ in $\mathbf{q_1}$, we get

$$\begin{aligned} \mathbf{q_1} &= \mathbf{q_1a + q_1a(b + aa)^*b + \Lambda} \\ &= \mathbf{q_1(a + a(b + aa)^*b) + \Lambda} \end{aligned}$$

Hence,

$$\begin{aligned} \mathbf{q_1} &= \mathbf{\Lambda(a + a(b + aa)^*b)^*} \\ \mathbf{q_2} &= \mathbf{(a + a(b + aa)^*b)^*\ a(b + aa)^*} \\ \mathbf{q_3} &= \mathbf{(a + a(b + aa)^*b)^*\ a(b + aa)^*a} \end{aligned}$$

Since q_3 is a final state, the set of strings recognized by the graph is given by

$$\mathbf{(a + a(b + aa)^*b)^*a(b + aa)^*a}$$

EXAMPLE 5.9

Prove that the finite automaton whose transition diagram is as shown in Fig. 5.14 accepts the set of all strings over the alphabet $\{a, b\}$ with an equal number of a's and b's, such that each prefix has at most one more a than the b's and at most one more b than the a's.

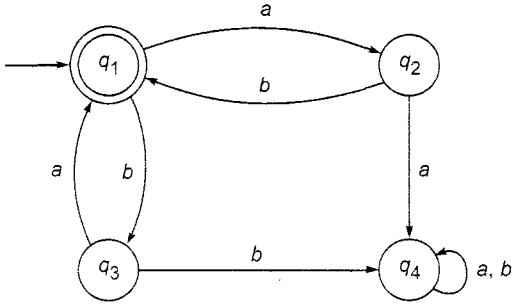


Fig. 5.14 Finite automaton of Example 5.9.

Solution

We can apply the above method directly since the graph does not contain the Λ-move and there is only one initial state. We get the following equations for $\mathbf{q}_1$, $\mathbf{q}_2$, $\mathbf{q}_3$, $\mathbf{q}_4$:

$$\mathbf{q}_1 = \mathbf{q}_2\mathbf{b} + \mathbf{q}_3\mathbf{a} + \Lambda$$

$$\mathbf{q}_2 = \mathbf{q}_1\mathbf{a}$$

$$\mathbf{q}_3 = \mathbf{q}_1\mathbf{b}$$

$$\mathbf{q}_4 = \mathbf{q}_2\mathbf{a} + \mathbf{q}_3\mathbf{b} + \mathbf{q}_4\mathbf{a} + \mathbf{q}_4\mathbf{b}$$

As $\mathbf{q}_1$ is the only final state and the $\mathbf{q}_1$-equation involves only $\mathbf{q}_2$ and $\mathbf{q}_3$, we use only $\mathbf{q}_2$- and $\mathbf{q}_3$-equations (the $\mathbf{q}_4$-equation is redundant for our purposes). Substituting for $\mathbf{q}_2$ and $\mathbf{q}_3$, we get

$$\mathbf{q}_1 = \mathbf{q}_1\mathbf{ab} + \mathbf{q}_1\mathbf{ba} + \Lambda = \mathbf{q}_1(\mathbf{ab} + \mathbf{ba}) + \Lambda$$

By applying Theorem 5.1, we get

$$\mathbf{q}_1 = \Lambda(\mathbf{ab} + \mathbf{ba})^* = (\mathbf{ab} + \mathbf{ba})^*$$

As $\mathbf{q}_1$ is the only final state, the strings accepted by the given finite automaton are the strings given by $(\mathbf{ab} + \mathbf{ba})^*$. As any such string is a string of *ab*'s, and *ba*'s, we get an equal number of *a*'s and *b*'s. If a prefix *x* of a sentence accepted by the finite automaton has an even number of symbols, then it should have an equal number of *a*'s and *b*'s since *x* is a substring formed by *ab*'s and *ba*'s. If the prefix *x* has an odd number of symbols, then we can write *x* as *ya* or *yb*. As *y* has an even number of symbols, *y* has an equal number of *a*'s and *b*'s. Thus, *x* has one more *a* than *b* or vice versa.

EXAMPLE 5.10

Describe in English the set accepted by the finite automaton whose transition diagram is as shown in Fig. 5.15.

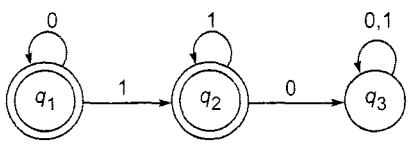


Fig. 5.15 Finite automaton of Example 5.10.

Solution

We can apply the above method directly as the transition diagram does not contain more than one initial state and there are no Λ-moves. We get the following equations for $\mathbf{q}_1$, $\mathbf{q}_2$, $\mathbf{q}_3$.

$$\mathbf{q}_1 = \mathbf{q}_1\mathbf{0} + \Lambda$$

$$\mathbf{q}_2 = \mathbf{q}_1\mathbf{1} + \mathbf{q}_2\mathbf{1}$$

$$\mathbf{q}_3 = \mathbf{q}_2\mathbf{0} + \mathbf{q}_3(\mathbf{0} + \mathbf{1})$$

By applying Theorem 5.1 to the $\mathbf{q}_1$-equation, we get

$$\mathbf{q}_1 = \Lambda\mathbf{0}^* = \mathbf{0}^*$$

So,

$$\mathbf{q}_2 = \mathbf{q}_1\mathbf{1} + \mathbf{q}_2\mathbf{1} = \mathbf{0}^*\mathbf{1} + \mathbf{q}_2\mathbf{1}$$

Therefore,

$$\mathbf{q}_2 = (\mathbf{0}^*\mathbf{1})\mathbf{1}^*$$

As the final states are $\mathbf{q}_1$ and $\mathbf{q}_2$, we need not solve for $\mathbf{q}_3$:

$$\mathbf{q}_1 + \mathbf{q}_2 = \mathbf{0}^* + \mathbf{0}^*(\mathbf{11}^*) = \mathbf{0}^*(\Lambda + \mathbf{11}^*) = \mathbf{0}^*(\mathbf{1}^*) \qquad \text{by } I_9$$

The strings represented by the transition graph are $\mathbf{0}^*\mathbf{1}^*$. We can interpret the strings in the English language in the following way: The strings accepted by the finite automaton are precisely the strings of any number of 0's (possibly Λ) followed by a string of any number of 1's (possibly Λ).

EXAMPLE 5.11

Construct a regular expression corresponding to the state diagram described by Fig. 5.16.

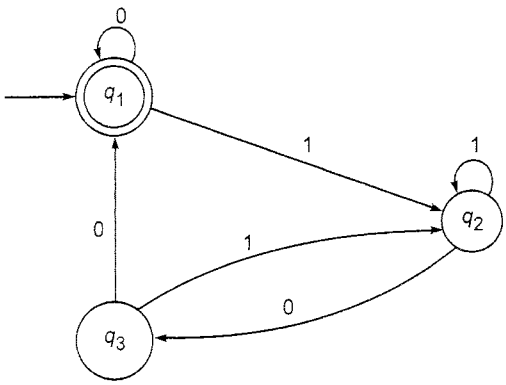


Fig. 5.16 Finite automaton of Example 5.11.

Solution

There is only one initial state. Also, there are no Λ-moves. The equations are

$$\mathbf{q}_1 = \mathbf{q}_1\mathbf{0} + \mathbf{q}_3\mathbf{0} + \Lambda$$

$$\mathbf{q}_2 = \mathbf{q}_1\mathbf{1} + \mathbf{q}_2\mathbf{1} + \mathbf{q}_3\mathbf{1}$$

$$\mathbf{q}_3 = \mathbf{q}_2\mathbf{0}$$

So,

$$\mathbf{q}_2 = \mathbf{q}_1\mathbf{1} + \mathbf{q}_2\mathbf{1} + (\mathbf{q}_2\mathbf{0})\mathbf{1} = \mathbf{q}_1\mathbf{1} + \mathbf{q}_2(\mathbf{1} + \mathbf{01})$$

By applying Theorem 5.1, we get

$$\mathbf{q}_2 = \mathbf{q}_1\mathbf{1}(\mathbf{1} + \mathbf{01})^*$$

Also,

$$\begin{aligned} q_1 &= q_1 0 + q_3 0 + \Lambda = q_1 0 + q_2 00 + \Lambda \\ &= q_1 0 + (q_1 1(1 + 01)^*)00 + \Lambda \\ &= q_1(0 + 1(1 + 01)^* 00) + \Lambda \end{aligned}$$

Once again applying Theorem 5.1, we get

$$q_1 = \Lambda(0 + 1(1 + 01)^* 00)^* = (0 + 1(1 + 01)^* 00)^*$$

As q_1 is the only final state, the regular expression corresponding to the given diagram is $(0 + 1(1 + 01)^* 00)^*$.

EXAMPLE 5.12

Find the regular expression corresponding to Fig. 5.17.

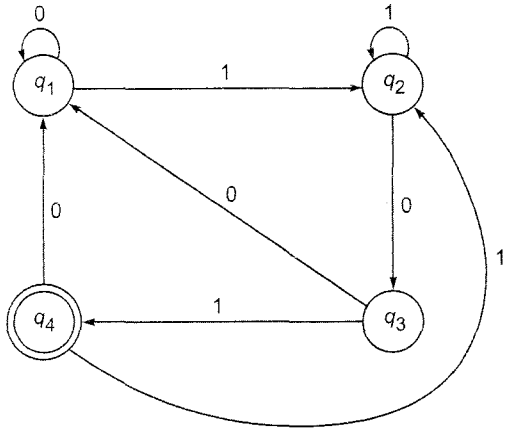


Fig. 5.17 Finite automaton of Example 5.12.

Solution

There is only one initial state, and there are no Λ-moves. So, we form the equations corresponding to q_1, q_2, q_3, q_4:

$$q_1 = q_1 0 + q_3 0 + q_4 0 + \Lambda$$

$$q_2 = q_1 1 + q_2 1 + q_4 1$$

$$q_3 = q_2 0$$

$$q_4 = q_3 1$$

Now,

$$q_4 = q_3 1 = (q_2 0)1 = q_2 01$$

Thus, we are able to write q_3, q_4 in terms of q_2. Using the q_2-equation, we get

$$q_2 = q_1 1 + q_2 1 + q_2 011 = q_1 1 + q_2(1 + 011)$$

By applying Theorem 5.1, we obtain

$$q_2 = (q_1\mathbf{1})(\mathbf{1} + \mathbf{011})^* = q_1(\mathbf{1}(\mathbf{1} + \mathbf{011})^*)$$

From the q_1-equation, we have

$$\begin{aligned} q_1 &= q_1\mathbf{0} + q_2\mathbf{00} + q_2\mathbf{010} + \Lambda \\ &= q_1\mathbf{0} + q_2(\mathbf{00} + \mathbf{010}) + \Lambda \\ &= q_1\mathbf{0} + q_1\mathbf{1}(\mathbf{1} + \mathbf{011})^* (\mathbf{00} + \mathbf{010}) + \Lambda \end{aligned}$$

Again, by applying Theorem 5.1, we obtain

$$q_1 = \Lambda(\mathbf{0} + \mathbf{1}(\mathbf{1} + \mathbf{011})^* (\mathbf{00} + \mathbf{010}))^*$$

$$\begin{aligned} q_4 &= q_2\mathbf{01} = q_1\mathbf{1}(\mathbf{1} + \mathbf{011})^* \mathbf{01} \\ &= (\mathbf{0} + \mathbf{1}(\mathbf{1} + \mathbf{011})^*(\mathbf{00} + \mathbf{010}))^*(\mathbf{1}(\mathbf{1} + \mathbf{011})^* \mathbf{01}) \end{aligned}$$

5.2.5 Construction of Finite Automata Equivalent to a Regular Expression

The method we are going to give for constructing a finite automaton equivalent to a given regular expression is called the *subset method* which involves two steps.

Step 1 Construct a transition graph (transition system) equivalent to the given regular expression using Λ-moves. This is done by using Theorem 5.2.

Step 2 Construct the transition table for the transition graph obtained in step 1. Using the method given in Section 5.2.3, construct the equivalent DFA. We reduce the number of states if possible.

EXAMPLE 5.13

Construct the finite automaton equivalent to the regular expression

$$(\mathbf{0} + \mathbf{1})^*(\mathbf{00} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*$$

Solution

Step 1 (Construction of transition graph) First of all we construct the transition graph with Λ-moves using the constructions of Theorem 5.2. Then we eliminate Λ-moves as discussed in Section 5.2.1.

We start with Fig. 5.18(a).

We eliminate the concatenations in the given r.e. by introducing new vertices q_1 and q_2 and get Fig. 5.18(b).

We eliminate the $*$ operations in Fig. 5.18(b) by introducing two new vertices q_5 and q_6 and the Λ-moves as shown in Fig. 5.18(c).

We eliminate concatenations and + in Fig. 5.18(c) and get Fig. 5.18(d).

We eliminate the Λ-moves in Fig. 5.18(d) and get Fig. 5.18(e) which gives the NDFA equivalent to the given i.e.

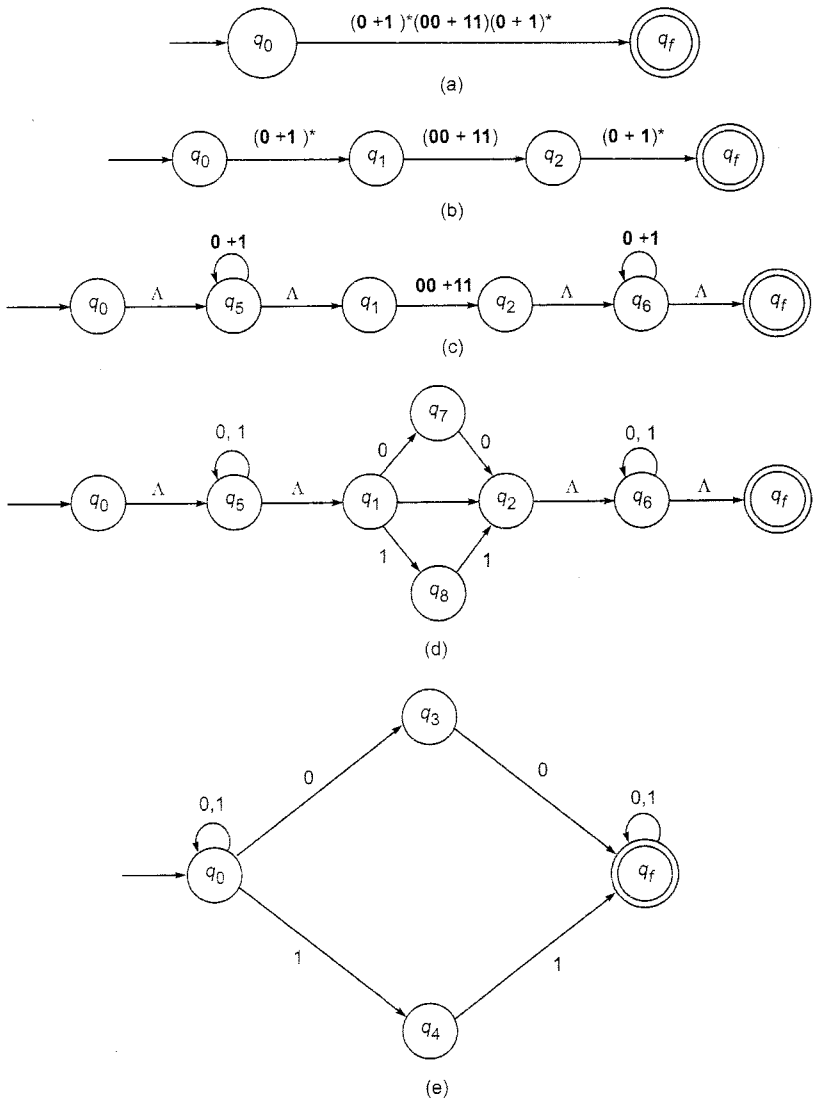


Fig. 5.18 Construction of finite automaton equivalent to **(0 + 1)*(00 + 11)(0 + 1)***.

Step 2 (Construction of DFA) We construct the transition table for the NDFA defined by Table 5.3.

TABLE 5.3 Transition Table for Example 5.13

State/Σ	0	1
$\rightarrow q_0$	q_0, q_3	q_0, q_4
q_3	q_f	
q_4		q_f
(q_f)	q_f	q_f

The successor table is constructed as given in Table 5.4.

TABLE 5.4 Transition Table for the DFA of Example 5.13.

Q	Q_0	Q_1
$\rightarrow [q_0]$	$[q_0, q_3]$	$[q_0, q_4]$
$[q_0, q_3]$	$[q_0, q_3, q_f]$	$[q_0, q_4]$
$[q_0, q_4]$	$[q_0, q_3]$	$[q_0, q_4, q_f]$
$[q_0, q_3, q_f]$	$[q_0, q_3, q_f]$	$[q_0, q_4, q_f]$
$[q_0, q_4, q_f]$	$[q_0, q_3, q_f]$	$[q_0, q_4, q_f]$

The state diagram for the successor table is the required DFA as described by Fig. 5.19. As q_f is the only final state of NDFA, $[q_0, q_3, q_f]$ and $[q_0, q_4, q_f]$ are the final states of DFA.

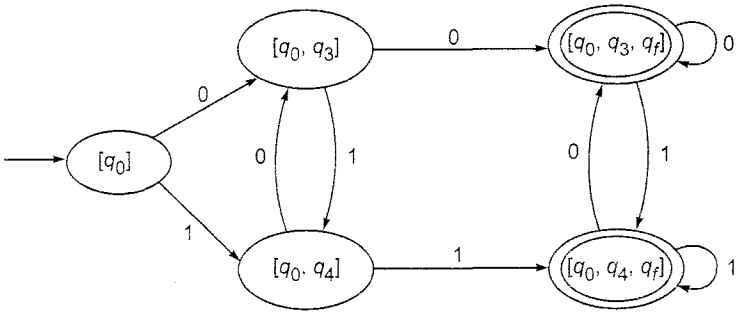


Fig. 5.19 Finite automaton of Example 5.13.

Finally, we try to reduce the number of states. (This is possible when two rows are identical in the successor table.) As the rows corresponding to $[q_0, q_3, q_f]$ and $[q_0, q_4, q_f]$ are identical, we identify them. The state diagram for the equivalent automaton, where the number of states is reduced, is described by Fig. 5.20.

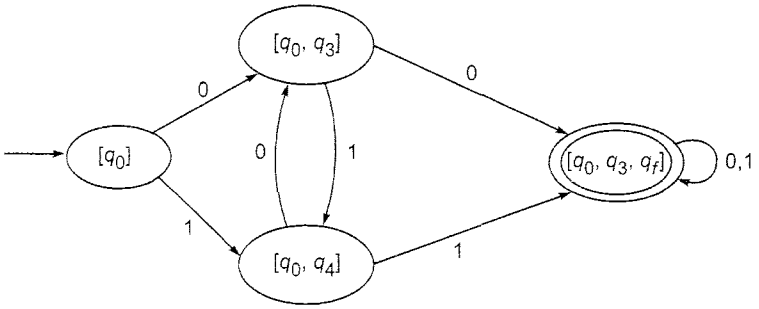


Fig. 5.20 Reduced finite automaton of Example 5.13.

Note: While constructing the transition graph equivalent to a given r.e., the operation (concatenation, *, +) that is eliminated first, depends on the regular expression.

EXAMPLE 5.14

Construct a DFA with reduced states equivalent to the r.e. **10 + (0 + 11))0*1**.

Solution

Step 1 (Construction of NDFA) The NDFA is constructed by eliminating the operation +, concatenation and *, and the Λ-moves in successive steps. The step-by-step construction is given in Figs. 5.21(a)–5.21(e).

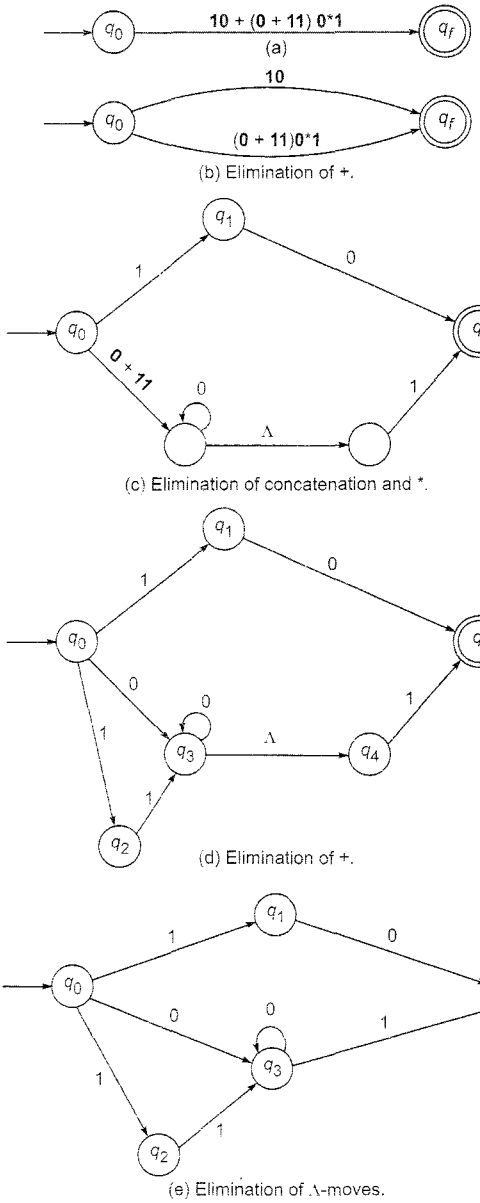


Fig. 5.21 Construction of finite automaton for Example 5.14.

Step 2 (Construction of DFA) For the NDFA given in Fig. 5.18(e), the corresponding transition table is defined by Table 5.5.

TABLE 5.5 Transition Table for Example 5.14

State/Σ	0	1
→ q_o	q_3	q_1, q_2
q_1	q_f	
q_2		q_3
q_3	q_3	q_f
(q_f)		

The successor table is constructed and given in Table 5.6.

In Table 5.6 the columns corresponding to $[q_f]$ and $\emptyset$ are identical. So we can identify $[q_f]$ and $\emptyset$.

TABLE 5.6 Transition Table of DFA for Example 5.14

Q	Q_0	Q_1
→ $[q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_3]$
($[q_f]$)	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$

The DFA with the reduced number of states corresponding to Table 5.6 is defined by Fig. 5.22.

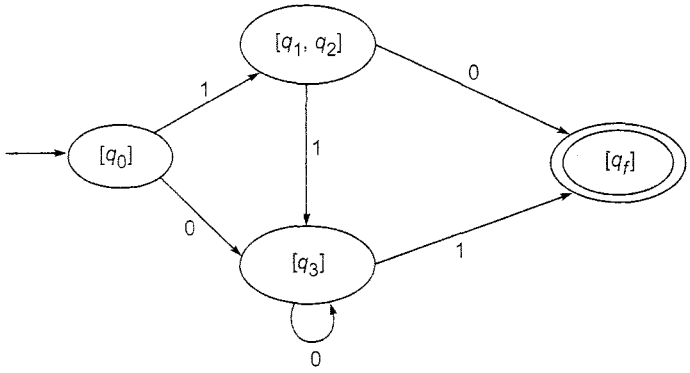


Fig. 5.22 Reduced DFA of Example 5.14.

5.2.6 Equivalence of Two Finite Automata

Two finite automata over Σ are equivalent if they accept the same set of strings over Σ. When the two finite automata are not equivalent, there is some string

w over Σ satisfying the following: One automaton reaches a final state on application of w, whereas the other automaton reaches a nonfinal state.

We give below a method, called the *comparison method*, to test the equivalence of two finite automata over Σ.

Comparison Method

Let M and M' be two finite automata over Σ. We construct a comparison table consisting of $n + 1$ columns, where n is the number of input symbols. The first column consists of pairs of vertices of the form (q, q'), where $q \in M$ and $q' \in M'$. If (q, q') appears in some row of the first column, then the corresponding entry in the a-column ($a \in \Sigma$) is (q_a, q'_a), where q_a and q'_a are reachable from q and q', respectively on application of a (i.e. by a-paths).

The comparison table is constructed by starting with the pair of initial vertices q_{in}, q'_{in} of M and M' in the first column. The first elements in the subsequent columns are (q_a, q'_a), where q_a and q'_a are reachable by a-paths from q_{in} and q'_{in}. We repeat the construction by considering the pairs in the second and subsequent columns which are not in the first column.

The row-wise construction is repeated. There are two cases:

Case 1 If we reach a pair (q, q') such that q is a final state of M, and q' is a nonfinal state of M' or vice versa, we terminate the construction and conclude that M and M' are not equivalent.

Case 2 Here the construction is terminated when no new element appears in the second and subsequent columns which are not in the first column (i.e. when all the elements in the second and subsequent columns appear in the first column). In this case we conclude that M and M' are equivalent.

EXAMPLE 5.15

Consider the following two DFAs M and M' over $\{0, 1\}$ given in Fig. 5.23. Determine whether M and M' are equivalent.

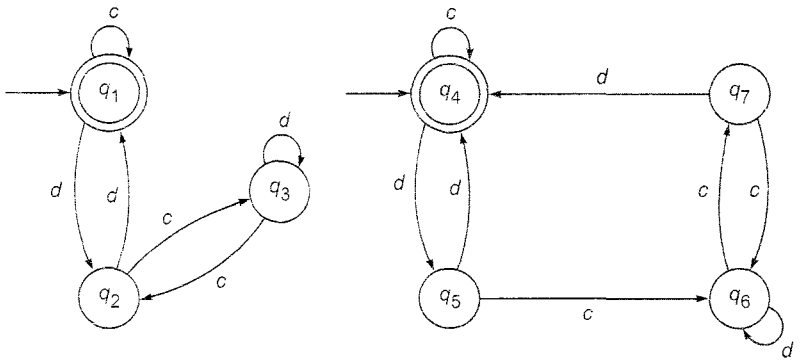


Fig. 5.23 (a) Automaton M and (b) automaton M'.

Solution

The initial states in M and M' are q_1 and q_4, respectively. Hence the first element of the first column in the comparison table must be (q_1, q_4). The first element in the second column is (q_1, q_4) since both q_1 and q_4 are c-reachable from the respective initial states. The complete table is given in Table 5.7.

TABLE 5.7 Comparison Table for Example 5.15

(q, q')	(q_c, q'_c)	(q_d, q'_d)
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_6)	(q_1, q_4)
(q_3, q_6)	(q_2, q_7)	(q_3, q_6)
(q_2, q_7)	(q_3, q_6)	(q_1, q_4)

As we do not get a pair (q, q'), where q is a final state and q' is a nonfinal state (or vice versa) at every row, we proceed until all the elements in the second and third columns are also in the first column. Therefore, M and M' are equivalent.

EXAMPLE 5.16

Show that the automata M_1 and M_2 defined by Fig. 5.24 are not equivalent.

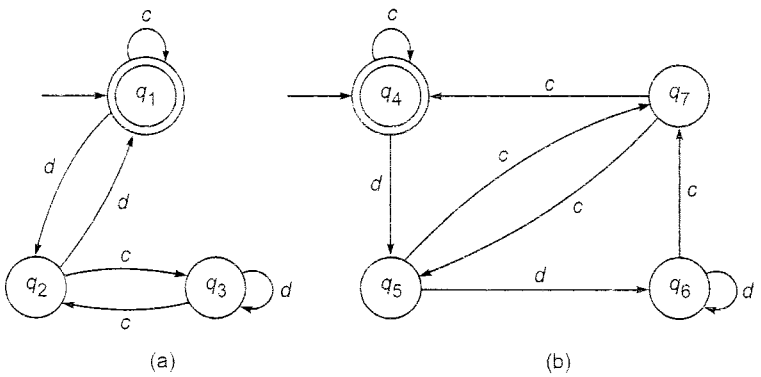


Fig. 5.24 (a) Automaton M_1 and (b) automaton M_2.

Solution

The initial states in M_1 and M_2 are q_1 and q_4, respectively. Hence the first column in the comparison table is (q_1, q_4). q_2 and q_5 are d-reachable from q_1 and q_4. We see from the comparison table given in Table 5.8 that q_1 and q_6 are d-reachable from q_2 and q_5, respectively. As q_1 is a final state in M_1, and q_6 is a nonfinal state in M_2, we see that M_1 and M_2 are not equivalent: we can also note that q_1 is dd-reachable from q_1, and hence dd is accepted by M_1. dd is not accepted by M_2 as only q_6 is dd-reachable from q_4, but q_6 is nonfinal.

TABLE 5.8 Comparison Table for Example 5.16

(q, q')	(q_c, q'_c)	(q_d, q'_d)
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_7)	(q_1, q_6)

5.2.7 EQUIVALENCE OF TWO REGULAR EXPRESSIONS

Suppose we are interested in testing the equivalence of two regular expressions, say **P** and **Q**. The regular expressions **P** and **Q** are equivalent iff they represent the same set. Also, **P** and **Q** are equivalent iff the corresponding finite automata are equivalent.

To prove the equivalence of **P** and **Q**, (i) we prove that the sets **P** and **Q** are the same. (For nonequivalence we find a string in one set but not in the other.) Or (ii) we use the identities to prove the equivalence of **P** and **Q**. Or (iii) we construct the corresponding FA M and M' and prove that M and M' are equivalent. (For nonequivalence we prove that M and M' are not equivalent.)

The method to be chosen depends on the problem.

EXAMPLE 5.17

Prove $(\mathbf{a} + \mathbf{b})^* = \mathbf{a}^*(\mathbf{ba}^*)^*$.

Solution

Let **P** and **Q** denote $(\mathbf{a} + \mathbf{b})^*$ and $\mathbf{a}^*(\mathbf{ba}^*)^*$, respectively. Using the construction in Section 5.2.5, **P** is given by the transition system depicted in Fig. 5.25.

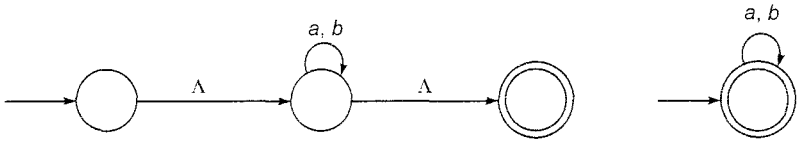


Fig. 5.25 Transition system for $(\mathbf{a} + \mathbf{b})^*$.

The transition system for **Q** is depicted in Fig. 5.26.

It should be noted that Figs. 5.25 and 5.26 are obtained after eliminating Λ-moves. As these two transition diagrams are the same, we conclude that $\mathbf{P} = \mathbf{Q}$.

We now summarize all the results and constructions given in this section.

(i) Every r.e. is recognized by a transition system (Theorem 5.2).

(ii) A transition system M can be converted into a finite automaton accepting the same set as M (Section 5.2.3).

(iii) Any set accepted by finite automaton is represented by an r.e. (Theorem 5.3).

(iv) A set accepted by a transition system is represented by an r.e. (from (ii) and (iii)).

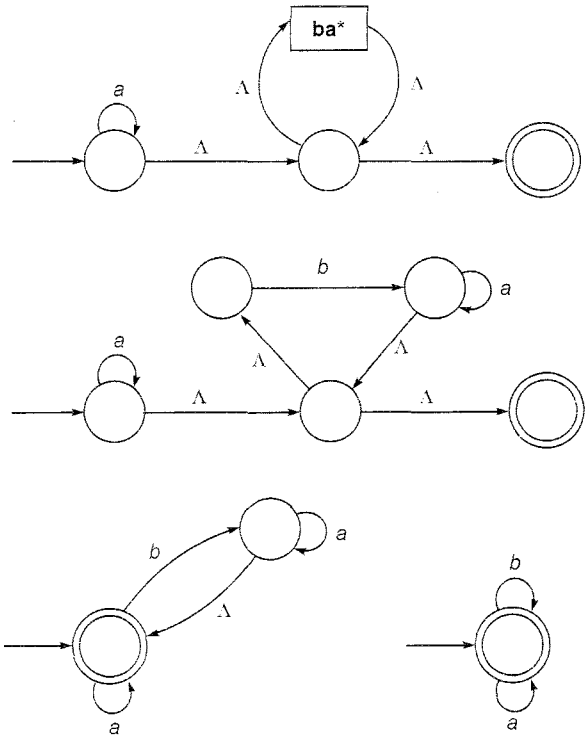


Fig. 5.26 Transition system for **a*(ba*)***.

(v) To get the r.e. representing a set accepted by a transition system, we can apply the algebraic method using the Arden's theorem (see Section 5.2.4).

(vi) If **P** is an r.e., then to construct a finite automaton accepting the set **P**, we can apply the construction given in Section 5.2.5.

(vii) A subset L of Σ^* is a regular set (or represented by an r.e.) iff it is accepted by an FA (from (i), (ii) and (iii)).

(viii) A subset L of Σ^* is a regular set iff it is recognized by a transition system (from (i) and (iv)).

(ix) The capabilities of finite automaton and transition systems are the same as far as acceptability of subsets of strings is concerned.

(x) To test the equivalence of two DFAs, we can apply the comparison method given in Section 5.2.6.

We conclude this section with the Kleene's theorem.

Theorem 5.4 (Kleene's theorem) The class of regular sets over Σ is the smallest class $\mathcal{R}$ containing $\{a\}$ for every $a \in \Sigma$ and closed under union, concatenation and closure.

Proof The set $\{a\}$ is represented by the regular expression **a**. So $\{a\}$ is regular for every $a \in \Sigma$. As the class of regular sets is closed under union, concatenation, and closure, $\mathcal{R}$ is contained in the class of regular sets.

Let L be a regular set. Then $L = T(M)$ for some DFA, $M = (\{q_1, \ldots, q_m\}, \Sigma, \delta, q_0, F)$. By Theorem 5.3,

$$L = \bigcup_{j=1}^{n} P_{1f_j}^{m}$$

where $F = \{q_{f_1} \ldots q_{f_n}\}$ and $P_{1f_j}^{m}$ is obtained by applying union, concatenation and closure to singletons in Σ. Thus, L is in $\mathcal{R}$. ∎

5.3 PUMPING LEMMA FOR REGULAR SETS

In this section we give a necessary condition for an input string to belong to a regular set. The result is called *pumping lemma* as it gives a method of pumping (generating) many input strings from a given string. As pumping lemma gives a necessary condition, it can be used to show that certain sets are not regular.

Theorem 5.5 (Pumping Lemma) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz$, $y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.

Proof Let

$$w = a_1a_2 \ldots a_m, \qquad m \geq n$$

$$\delta(q_0, a_1a_2 \ldots a_i) = q_i \quad \text{for } i = 1, 2, \ldots, m; \qquad Q_1 = \{q_0, q_1, \ldots, q_m\}$$

That is, Q_1 is the sequence of states in the path with path value $w = a_1a_2 \ldots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_j and $q_k (q_j = q_k)$. Then j and k satisfy the condition $0 \leq j < k \leq n$.

The string w can be decomposed into three substrings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$ and $a_{k+1} \ldots a_m$. Let x, y, z denote these strings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$, $a_{k+1} \ldots a_m$, respectively. As $k \leq n$, $|xy| \leq n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in Fig. 5.27.

The automaton M starts from the initial state q_0. On applying the string x, it reaches $q_j (= q_k)$. On applying the string y, it comes back to $q_j (= q_k)$. So after application of y^i for each $i \geq 0$, the automaton is in the same state q_j. On applying z, it reaches q_m, a final state. Hence, $xy^iz \in L$. As every state in Q_1 is obtained by applying an input symbol, $y \neq \Lambda$. ∎

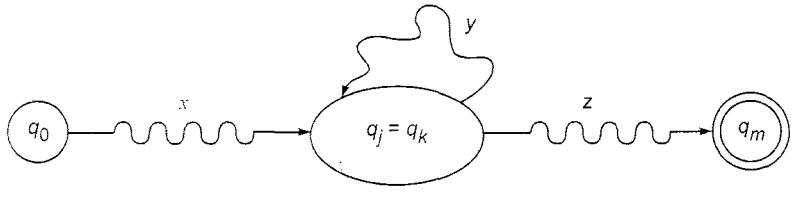


Fig. 5.27 String accepted by M.

Note: The decomposition is valid only for strings of length greater than or equal to the number of states. For such a string $w = xyz$, we can 'iterate' the substring y in xyz as many times as we like and get strings of the form xy^iz which are longer than xyz and are in L. By considering the path from q_0 to q_k and then the path from q_k to q_m (without going through the loop), we get a path ending in a final state with path value xz. (This corresponds to the case when $i = 0$.)

5.4 APPLICATION OF PUMPING LEMMA

This theorem can be used to prove that certain sets are not regular. We now give the steps needed for proving that a given set is not regular.

Step 1 Assume that L is regular. Let n be the number of states in the corresponding finite automaton.

Step 2 Choose a string w such that $|w| \geq n$. Use pumping lemma to write $w = xyz$, with $|xy| \leq n$ and $|y| > 0$.

Step 3 Find a suitable integer i such that $xy^iz \notin L$. This contradicts our assumption. Hence L is not regular.

Note: The crucial part of the procedure is to find i such that $xy^iz \notin L$. In some cases we prove $xy^iz \notin L$ by considering $|xy^iz|$. In some cases we may have to use the 'structure' of strings in L.

EXAMPLE 5.18

Show that the set $L = \{a^{i^2} \mid i \geq 1\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2 Let $w = a^{n^2}$. Then $|w| = n^2 > n$. By pumping lemma, we can write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$.

Step 3 Consider xy^2z. $|xy^2z| = |x| + 2|y| + |z| > |x| + |y| + |z|$ as $|y| > 0$. This means $n^2 = |xyz| = |x| + |y| + |z| < |xy^2z|$. As $|xy| \leq n$, we have $|y| \leq n$. Therefore,

$$|xy^2z| = |x| + 2|y| + |z| \leq n^2 + n$$

i.e.

$$n^2 < |xy^2z| \leq n^2 + n < n^2 + n + n + 1$$

Hence, $|xy^2z|$ strictly lies between n^2 and $(n + 1)^2$, but is not equal to any one of them. Thus $|xy^2z|$ is not a perfect square and so $xy^2z \notin L$. But by pumping lemma, $xy^2z \in L$. This is a contradiction.

EXAMPLE 5.19

Show that $L = \{a^p \mid p \text{ is a prime}\}$ is not regular.

Solution

Step 1 We suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2 Let p be a prime number greater than n. Let $w = a^p$. By pumping lemma, w can be written as $w = xyz$, with $|xy| \leq n$ and $|y| > 0$. x, y, z are simply strings of a's. So, $y = a^m$ for some $m \geq 1$ (and $\leq n$).

Step 3 Let $i = p + 1$. Then $|xy^iz| = |xyz| + |y^{i-1}| = p + (i - 1)m = p + pm$. By pumping lemma, $xy^iz \in L$. But $|xy^iz| = p + pm = p(1 + m)$, and $p(1 + m)$ is not a prime. So $xy^iz \notin L$. This is a contradiction. Thus L is not regular.

EXAMPLE 5.20

Show that $L = \{0^i1^i \mid i \geq 1\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2 Let $w = 0^n1^n$. Then $|w| = 2n > n$. By pumping lemma, we write $w = xyz$ with $|xy| \leq n$ and $|y| \neq 0$.

Step 3 We want to find i so that $xy^iz \notin L$ for getting a contradiction. The string y can be in any of the following forms:

Case 1 y has 0's, i.e. $y = 0^k$ for some $k \geq 1$.

Case 2 y has only 1's, i.e. $y = 1^l$ for some $l \geq 1$.

Case 3 y has both 0's and 1's, i.e. $y = 0^k1^j$ for some $k, j \geq 1$.

In Case 1, we can take $i = 0$. As $xyz = 0^n1^n$, $xz = 0^{n-k}1^n$. As $k \geq 1$, $n - k \neq n$. So, $xz \notin L$.

In Case 2, take $i = 0$. As before, xz is 0^n1^{n-l} and $n \neq n - l$. So, $xz \notin L$.

In Case 3, take $i = 2$. As $xyz = 0^{n-k}0^k1^j1^{n-j}$, $xy^2z = 0^{n-k}\,0^k1^j0^k1^j1^{n-j}$. As xy^2z is not of the form 0^i1^i, $xy^2z \notin L$.

Thus in all the cases we get a contradiction. Therefore, L is not regular.

EXAMPLE 5.21

Show that $L = \{ww \mid w \in \{a, b\}^*\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the automaton M accepting L.

Step 2 Let us consider $ww = a^n b a^n b$ in L. $|ww| = 2(n + 1) > n$. We can apply pumping lemma to write $ww = xyz$ with $|y| \neq 0$, $|xy| \leq n$.

Step 3 We want to find i so that $xy^i z \notin L$ for getting a contradiction. The string y can be in only one of the following forms:

Case 1 y has no b's, i.e. $y = a^k$ for some $k \geq 1$.

Case 2 y has only one b.

We may note that y cannot have two b's. If so, $|y| \geq n + 2$. But $|y| \leq |xy| \leq n$. In Case 1, we can take $i = 0$. Then $xy^0z = xz$ is of the form $a^m b a^n b$, where $m = n - k < n$ (or $a^n b a^m b$). We cannot write xz in the form uu with $u \in \{a, b\}^*$, and so $xz \notin L$. In Case 2 too, we can take $i = 0$. Then $xy^0z = xz$ has only one b (as one b is removed from xyz, b being in y). So $xz \notin L$ as any element in L should have an even number of a's and an even number of b's.

Thus in both the cases we get a contradiction. Therefore, L is not regular.

Note: If a set L of strings over Σ is given and if we have to test whether L is regular or not, we try to write a regular expression representing L using the definition of L. If this is not possible, we use pumping lemma to prove that L is not regular.

EXAMPLE 5.22

Is $L = \{a^{2n} \mid n \geq 1\}$ regular?

Solution

We can write a^{2n} as $a(a^2)^i a$, where $i \geq 0$. Now $\{(a^2)^i \mid i \geq 0\}$ is simply $\{a^2\}^*$. So L is represented by the regular expression $\mathbf{a(P)^*a}$, where $\mathbf{P}$ represents $\{a^2\}$. The corresponding finite automaton (using the construction given in Section 5.2.5) is shown in Fig. 5.28.

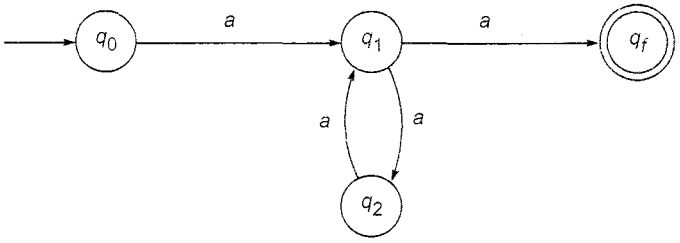


Fig. 5.28 Finite automaton of Example 5.22.

5.5 CLOSURE PROPERTIES OF REGULAR SETS

In this section we discuss the closure properties of regular sets under (i) set union, (ii) concatenation, (iii) closure (iteration), (iv) transpose, (v) set intersection, and (vi) complementation.

In Section 5.1, we have seen that the class of regular sets is closed under union, concatenation and closure.

Theorem 5.6 If L is regular then L^T is also regular.

Proof As L is regular by (vii), given at the end of Section 5.2.7, we can construct a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ such that $T(M) = L$.

We construct a transition system M' by starting with the state diagram of M, and reversing the direction of the directed edges. The set of initial states of M' is defined as the set F, and q_0 is defined as the (only) final state of M', i.e. $M' = (Q, \Sigma, \delta', F, \{q_0\})$.

If $w \in T(M)$, we have a path from q_0, to some final state in F with path value w. By 'reversing the edges', we get a path in M' from some final state in F to q_0. Its path value is w^T. So $w^T \in T(M')$. In a similar way, we can see that if $w_1 \in T(M')$, then $w_1^T \in T(M)$. Thus from the state diagram it is easy to see that $T(M') = T(M)^T$. We can prove rigorously that $w \in T(M)$ iff $w^T \in T(M')$ by induction on $|w|$. So $T(M)^T = T(M')$. By (viii) of Section 5.2.7, $T(M')$ is regular, i.e. $T(M)^T$ is regular. ∎

EXAMPLE 5.23

Consider the FA M given by Fig. 5.29. What is $T(M)$? Show that $T(M)^T$ is regular.

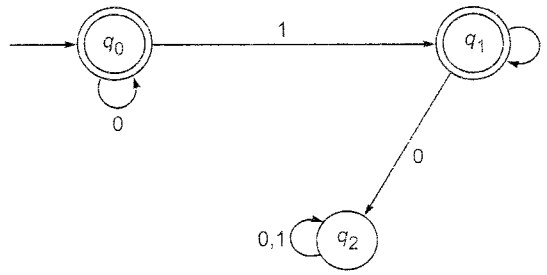


Fig. 5.29 Finite automaton of Example 5.23.

Solution

As the elements of $T(M)$ are given by path values of paths from q_0 to itself or from q_0 to q_1 (note that we have two final states q_0 and q_1), we can construct $T(M)$ by inspection.

As arrows do not come into q_0, the paths from q_0 to itself are self-loops repeated any number of times. The corresponding path values are 0^i, $i \geq 1$. As no arrow comes from q_2 to q_0 or q_1, the paths from q_0 to q_1 are of the form $q_0 \ldots \rightarrow q_0 \ldots q_1 \ldots \rightarrow q_1$. The corresponding path values are 0^i1^j, where $i \geq 0$ and $j \geq 1$. As the initial state q_0 is also a final state, $\Lambda \in T(M)$. Thus,

$$T(M) = \{0^i1^j \mid i, j \geq 0\}$$

Hence,

$$T(M)^T = \{1^j0^i \mid i, j \geq 0\}$$

The transition system M' is constructed as follows:

(i) The initial states of M' are q_0 and q_1.
(ii) The (only) final state of M' is q_0.
(iii) The direction of the directed edges is reversed. M' is given in Fig. 5.30.

From (i)–(iii) it follows that

$$T(M') = T(M)^T$$

Hence, $T(M)^T$ is regular.

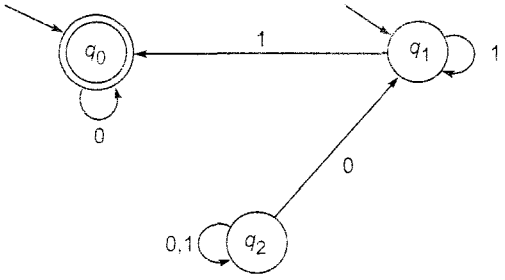


Fig. 5.30 Finite automaton of $T(M)^T$.

Note: In Example 5.23, we can see by inspection that $T(M') = \{1^i\ 0^i \mid i, j \geq 0\}$. The strings of $T(M')$ are obtained as path values of paths from q_0 to itself or from q_1 to q_0.

Theorem 5.7 If L is a regular set over Σ, then $\Sigma^* - L$ is also regular over Σ.

Proof As L is regular by (vii), given at the end of Section 5.2.7, we can construct a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting L, i.e. $L = T(M)$.

We construct another DFA $M' = (Q, \Sigma, \delta, q_0, F')$ by defining $F' = Q - F$, i.e. M and M' differ only in their final states. A final state of M' is a nonfinal state of M and vice versa. The state diagrams of M and M' are the same except for the final states.

$w \in T(M')$ if and only if $\delta(q_0, w) \in F' = Q - F$, i.e. iff $w \notin L$. This proves $T(M') = \Sigma^* - X$. ∎

Theorem 5.8 If X and Y are regular sets over Σ, then $X \cap Y$ is also regular over Σ.

Proof By DeMorgan's law for sets, $X \cap Y = \Sigma^* - ((\Sigma^* - X) \cup (\Sigma^* - Y))$. By Theorem 5.7, $\Sigma^* - X$ and $\Sigma^* - Y$ are regular. So, $(\Sigma^* - X) \cup (\Sigma^* - Y)$ is also regular. By applying Theorem 5.7, once again $\Sigma^* - ((\Sigma^* - X) \cup (\Sigma^* - Y))$ is regular, i.e. $X \cap Y$ is regular. ∎

5.6 REGULAR SETS AND REGULAR GRAMMARS

We have seen that regular sets are precisely those accepted by DFA. In this section we show that the class of regular sets over Σ is precisely the regular languages over the terminal set Σ.

5.6.1 Construction of a Regular Grammar Generating $T(M)$ for a Given DFA M

Let $M = (\{q_0, \ldots, q_n\}, \Sigma, \delta, q_0, F)$. If w is in $T(M)$, then it is obtained by concatenating the labels corresponding to several transitions, the first from q_0 and the last terminating at some final state. So for the grammar G to be constructed, productions should correspond to transitions. Also, there should be provision for terminating the derivation once a transition terminating at some final state is encountered. With these ideas in mind, we construct G as

$$G = (\{A_0, A_1, \ldots, A_n\}, \Sigma, P, A_0)$$

where P is defined by the following rules:

(i) $A_i \to aA_j$ is included in P if $\delta(q_i, a) = q_j \notin F$.
(ii) $A_i \to aA_j$ and $A_i \to a$ are included in P if $\delta(q_i, a) = q_j \in F$.

We can show that $L(G) = T(M)$ by using the construction of P. Such a construction gives

$$A_i \Rightarrow aA_j \quad \text{iff } \delta(q_i, a) = q_j$$
$$A_i \Rightarrow a \quad \text{iff } \delta(q_i, a) \in F$$

So,

$$A_0 \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1 \ldots a_{k-1}A_k \Rightarrow a_1a_2 \ldots a_k$$

iff $\delta(q_0, a_1) = q_1$, $\delta(q_1, a_2) = q_2, \ldots,$ $\delta(q_k, a_k) \in F$

This proves that $w = a_1 \ldots a_k \in L(G)$ iff $\delta(q_0, a_1 \ldots a_k) \in F$, i.e. iff $w \in T(M)$.

EXAMPLE 5.24

Construct a regular grammar G generating the regular set represented by $\mathbf{P = a^*b(a + b)^*}$.

Solution

We construct the DFA corresponding to **P** using the construction given in Section 5.2.5. The construction is shown in Fig. 5.31.

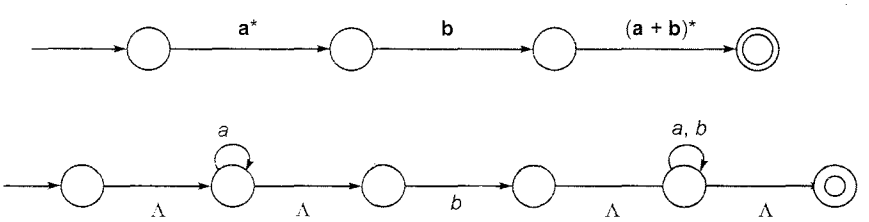


Fig. 5.31 DFA of Example 5.24, with Λ-moves.

After eliminating the Λ-moves, we get the DFA straightaway, as shown in Fig. 5.32.

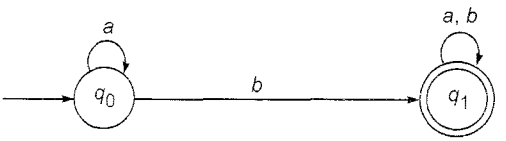


Fig. 5.32 DFA of Example 5.24, without Λ-moves.

Let $G = (\{A0, A_1\}, \{a, b\}, P, A_0)$, where P is given by

$$A_0 \to aA_0, \quad A_0 \to bA_1, \quad A_0 \to b$$

$$A_1 \to aA_1, \quad A_1 \to bA_1, \quad A_1 \to a, \quad A_1 \to b$$

G is the required regular grammar.

5.6.2 Construction of a Transition System M Accepting $L(G)$ for a Given Regular Grammar G

Let $G = (\{A_0, A_1, \ldots, A_n\}, \Sigma, P, A_0)$. We construct a transition system M whose (i) states correspond to variables, (ii) initial state corresponds to A_0, and (iii) transitions in M correspond to productions in P. As the last production applied in any derivation is of the form $A_i \to a$, the corresponding transition terminates at a new state, and this is the unique final state.

We define M as $(\{q_0, \ldots, q_n, q_f\}, \Sigma, \delta, q_0, \{q_f\})$ where δ is defined as follows:

(i) Each production $A_i \to aA_j$ induces a transition from q_i to q_j with label a.

(ii) Each production $A_k \to a$ induces a transition from q_k to q_f with label a.

From the construction it is easy to see that $A_0 \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1 \ldots a_{n-1}A_{n-1} \Rightarrow a_1 \ldots a_n$ is a derivation of $a_1a_2 \ldots a_n$ iff there is a path in M starting from q_0 and terminating in q_f with path value $a_1a_2 \ldots a_n$. Therefore, $L(G) = T(M)$.

EXAMPLE 5.25

Let $G = (\{A_0, A_1\}, \{a, b\}, P, A_0)$, where P consists of $A_0 \to aA_1$, $A_1 \to bA_1$, $A_1 \to a$, $A_1 \to bA_0$. Construct a transition system M accepting $L(G)$.

Solution

Let $M = (\{q_0, q_1, q_f\}, \{a, b\}, \delta, q_0, \{q_f\})$, where q_0 and q_1 correspond to A_0 and A_1, respectively and q_f is the new (final) state introduced. $A_0 \to aA_1$ induces a transition from q_0 to q_1 with label a. Similarly, $A_1 \to bA_1$ and $A_1 \to bA_0$ induce transitions from q_1 to q_1 with label b and from q_1 to q_0 with

label b, respectively. $A_1 \rightarrow a$ induces a transition from q_1 to q_f with label a. M is given in Fig. 5.33.

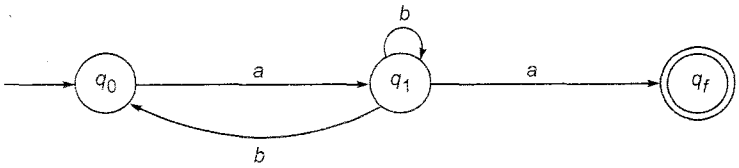


Fig. 5.33 Transition system for Example 5.25.

EXAMPLE 5.26

If a regular grammar G is given by $S \rightarrow aS \mid a$, find M accepting $L(G)$.

Solution

Let q_0 correspond to S and q_f be the new (final) state. M is given in Fig. 5.34. Symbolically,

$$M = (\{q_0, q_f\}, \{a\}, \delta, q_0, \{q_f\})$$

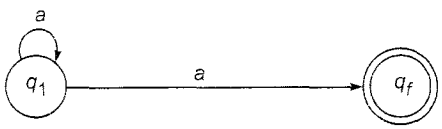


Fig. 5.34 Transition system for Example 5.26.

Note: If $S \rightarrow \Lambda$ is in P, the corresponding transition is from q_0 to q_f with label Λ.

By using the construction given in Section 5.2.3, we can construct a DFA M accepting $L(G)$ for a given regular grammar G.

5.7 SUPPLEMENTARY EXAMPLES

EXAMPLE 5.27

Find a regular expression corresponding to each of the following subsets of $\{a, b\}$.

(a) The set of all strings containing exactly $2a$'s.
(b) The set of all strings containing at least $2a$'s.
(c) The set of all strings containing at most $2a$'s.
(d) The set of all strings containing the substring aa.

Solution

(a) **b*ab*ab***
(b) **(a + b)*a(a + b)*a(a + b)***
(c) **b*ab*ab* + b*ab***
(d) **(a + b)*aa(a + b)***

EXAMPLE 5.28

Find a regular expression consisting of all strings over $\{a, b\}$ starting with any number of a's, followed by one or more b's, followed by one or more a's, followed by a single b, followed by any number of a's, followed by b and ending in any string of a's and b's.

Solution

The r.e. is **a*b b*a a*b(a + b)***.

EXAMPLE 5.29

Find the regular expression representing the set of all strings of the form

(a) $a^m b^n c^p$ where $m, n, p \geq 1$
(b) $a^m b^{2n} c^{3p}$ where $m, n, p \geq 1$
(c) $a^n b a^{2m} b^2$ where $m \geq 0, n \geq 1$

Solution

(a) **aa*bb*cc***
(b) **aa*(bb)(bb)*ccc(ccc)***
(c) **aa*b(aa)*bb**

EXAMPLE 5.30

Find the sets represented by the following regular expressions.

(a) **(a + b)*(aa + bb + ab + ba)***
(b) **(aa)* + (aaa)***
(c) **(1 + 01 + 001)*(Λ + 0 + 00)**
(d) **a + b(a + b)***

Solution

(a) The set of all strings having an odd number of symbols from $\{a, b\}$*
(b) $\{x \in \{a\}^* \mid |x| \text{ is divisible by 2 or 3}\}$
(c) The set of all strings over $\{0, 1\}$ having no substring of more than two adjacent 0's.
(d) $\{a, b, ba, bb, baa, bab, bba, bbb, \ldots\}$

EXAMPLE 5.31

Show that $\{w \in \{a, b\}^* \mid w$ contains an equal number of a's and b's$\}$ is not regular.

Solution

We prove this by contradiction. Assume that $L = T(M)$ for some DFA M with n states. Let $w = a^n b^n \in L$ and $|w| = 2^n$. Using the pumping lemma, we write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$. As $xyz = a^n b^n$, $xy = a^i$ where $i \leq n$ and hence $y = a^j$ for some j, $1 \leq j \leq n$. Consider xy^2z. Now xyz has an equal number of a's and b's. But xy^2z has $(n + j)$ a's and n b's. As $n + j \neq n$, $xy^2z \notin L$. This contradiction proves that L is not regular.

EXAMPLE 5.32

Show that $L = \{a^i b^j c^k \mid k > i + j\}$ is not regular.

Solution

We prove this by contradiction. Assume $L = T(M)$ for some DFA with n states. Choose $w = a^n b^n c^{3n}$ in L. Using the pumping lemma, we write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$. As $w = a^n b^n c^{3n}$, $xy = a^i$ for some $i \leq n$. This means that $y = a^j$ for some j, $1 \leq j \leq n$. Then $xy^{k+1}z = a^{n+jk} b^n c^{3n}$. Choosing k large enough so that $n + jk > 2n$, we can make $n + jk + n > 3n$. So, $xy^{k+1}z \notin L$. Hence L is not regular.

EXAMPLE 5.33

Prove the identities I_5, I_6, I_7, I_8, I_{11}, I_{12} given in Section 5.1.1.

Proof $L(\mathbf{R} + \mathbf{R}) = L(\mathbf{R}) \cup L(\mathbf{R}) = L(\mathbf{R})$. Hence I_5.

$L(\mathbf{R}^*\mathbf{R}^*) = L(\mathbf{R}^*)L(\mathbf{R}^*) = \{w_1 w_2 \mid w_1, w_2 \in L(\mathbf{R}^*)\}$. But $w_1 = x_1 x_2 \ldots x_n \in L(\mathbf{R})^*$ for $x_i \in L(\mathbf{R})$, $i = 1, 2, \ldots, n$. Similarly, $w_2 = y_1 y_2 \ldots y_m \in L(\mathbf{R}^*)$ for $y_j \in L(\mathbf{R})$, $j = 1, 2, \ldots, m$. So $w_1 w_2 \in L(\mathbf{R})^*$, proving I_6.

An element of $L(\mathbf{R}\mathbf{R}^*)$ is of the form $xy_1 \ldots y_n$ for some $x, y_1, \ldots, y_n \in L(\mathbf{R})$.

As $xy_1 \ldots y_n = (xy_1 \ldots y_{n-1})y_n \in L(\mathbf{R}^+\mathbf{R})$, I_7 follows.

It is easy to see that $L(\mathbf{R}^*) \subseteq L((\mathbf{R}^*)^*)$. Take $w \in L((\mathbf{R}^*)^*)$. Then $w = x_1 \ldots x_m$ where $x_i \in \mathbf{R}^*$, $i = 1, 2, \ldots, m$. Each x_i in turn can be written in the form $y_1 y_2 \ldots y_n$ for some $y_i \in L(\mathbf{R})$, $i = 1, 2, \ldots, n$. So, $w = x_1 \ldots x_m = z_1 \ldots z_k \in L(\mathbf{R}^*)$.

(***Note:*** $x_1 = z_1 \ldots z_t$, $x_2 = z_{t+1} \ldots z_{t+l}$ etc.)

Hence I_8.

To prove I_{11}, take $w \in L(\mathbf{P} + \mathbf{Q})^*$. Then $w = w_1 \ldots w_n$ where $w_i \in L(\mathbf{P}) \cup L(\mathbf{Q})$. By writing $w_i = \Lambda w_i = w_i \Lambda$, we note that $w_i \in \mathbf{P}^*\mathbf{Q}^*$. Hence $w \in L(\mathbf{P}^*\mathbf{Q}^*)^*$. To prove $L(\mathbf{P}^*\mathbf{Q}^*)^* \subseteq L(\mathbf{P} + \mathbf{Q})^*$, take $w \in L(\mathbf{P}^*\mathbf{Q}^*)^*$.

Then, $w = w_1 \ldots w_n$ where $w_i \in \mathbf{P^*Q^*}$. For simplicity take $n = 2$. (The result can be extended by induction for any n). Then $w_1 = x_1x_2 \ldots x_k$, where $x_i \in L(\mathbf{P})$ and $w_2 = y_1y_2 \ldots y_l$ where $y_j \in L(\mathbf{Q})$. So $w = x_1x_2 \ldots x_ky_1y_2 \ldots y_l$. Each x_i or y_j is in $L(\mathbf{P} + \mathbf{Q})$. Hence $w \in L(\mathbf{P} + \mathbf{Q})^*$, proving the first identity in I_{11}. The second identity can be proved in a similar way.

Finally, we prove I_{12}.

$$\begin{aligned} L((\mathbf{P} + \mathbf{Q})\mathbf{R}) &= L(\mathbf{P} + \mathbf{Q})L(\mathbf{R}) \\ &= (L(\mathbf{P}) \cup L(\mathbf{Q}))L(\mathbf{R}) \\ L(\mathbf{PR} + \mathbf{QR}) &= L(\mathbf{PR}) \cup L(\mathbf{QR}) \\ &= (L(\mathbf{P})L(\mathbf{R})) \cup (L(\mathbf{Q})L(\mathbf{R})) \end{aligned}$$

But $(A \cup B)C = AC \cup BC$ for $A, B, C \subseteq \Sigma^*$. For, a string w in $(A \cup B)C$ is the concatenation of a string w_1 in A or B and a string w_2 in C. If $w_1 \in A$, then $w_1w_2 \in AC$; if $w_1 \in B$, then $w_1w_2 \in BC$. Hence $w \in AC \cup BC$. The other inclusion can be proved similarly. I_{12} follows from $(A \cup B)C = AC \cup BC$.

EXAMPLE 5.34

Prove that $\mathbf{P} + \mathbf{PQ^*Q} = \mathbf{a^*bQ^*}$ where $\mathbf{P} = \mathbf{b} + \mathbf{aa^*b}$ and $\mathbf{Q}$ is any regular expression.

Proof

$$\begin{aligned} \text{L.H.S.} &= \mathbf{P}\Lambda + \mathbf{PQ^*Q} && \text{by } I_3 \\ &= \mathbf{P}(\Lambda + \mathbf{Q^*Q}) && \text{by } I_{12} \\ &= \mathbf{PQ^*} && \text{by } I_9 \\ &= (\mathbf{b} + \mathbf{aa^*b})\mathbf{Q^*} && \text{by definition of } \mathbf{P} \\ &= (\Lambda\mathbf{b} + \mathbf{aa^*b})\mathbf{Q^*} && \text{by } I_3 \\ &= (\Lambda + \mathbf{aa^*})\mathbf{bQ^*} && \text{by } I_{12} \\ &= \mathbf{a^*bQ^*} && \text{by } I_9 \\ &= \text{R.H.S.} \end{aligned}$$

EXAMPLE 5.35

Construct a regular grammar accepting $L = \{w \in \{a, b\}^* \mid w \text{ is a string over } \{a, b\} \text{ such that the number of } b\text{'s is 3 mod 4}\}$.

Solution

We construct a DFA M accepting L directly. The symbol a can occur in any place in w and b has to occur in $4k + 3$ places, where $k \geq 0$. So we can have states q_i, $i = 0, 1, 2, 3$, for remembering that the string processed so far has $4k$, $4k + 1$, $4k + 2$ and $4k + 3$ b's ($k \geq 0$). q_3 is the only final state. Also M does not change state on reading a's. The state diagram representing M is given in Fig. 5.35.

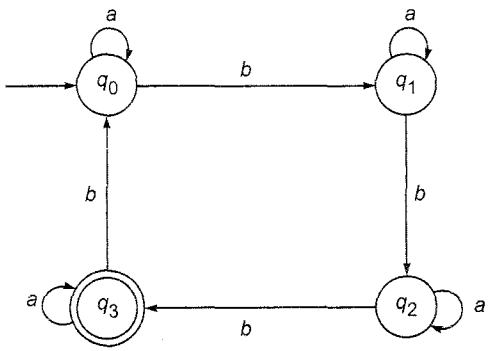


Fig. 5.35 DFA for Example 5.35.

By applying the construction given in Section 5.6.1, we can construct a regular grammar G accepting $L = T(M)$.

$G = (\{A_0, A_1, A_2, A_3\}, \{a, b\}, P, A_0)$ where P consists of $A_0 \to aA_0$, $A_0 \to bA_1$, $A_1 \to bA_1$, $A_1 \to bA_2$, $A_2 \to aA_2$, $A_2 \to bA_3$, $A_2 \to b$, $A_3 \to aA_3$, $A_3 \to aA_0$.

EXAMPLE 5.36

Let $G = (\{A_0, A_1, A_2, A_3\}, \{a, b\}, P, A_0)$, where P consists of $A_0 \to aA_0 \mid bA_1$, $A_1 \to aA_2 \mid aA_3$, $A_3 \to a \mid bA_1 \mid bA_3$, $A_3 \to b \mid bA_0$. Construct an NDFA accepting $L(G)$.

Solution

The NDFA accepting $L = L(G)$ is M where $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_4\})$. δ is described by the state diagram shown in Fig. 5.36.

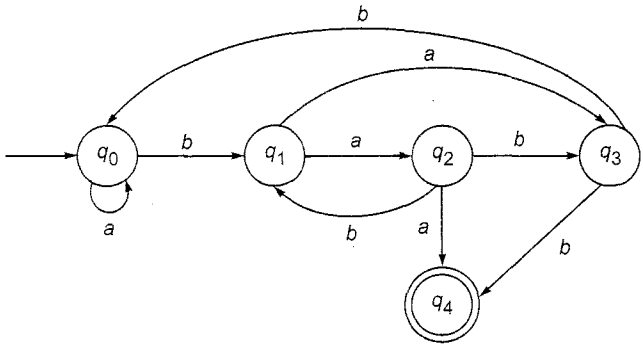


Fig. 5.36 NDFA for Example 5.36.

SELF-TEST

Choose the correct answer to Questions 1–10.

1. The set of all strings over $\{a, b\}$ of even length is represented by the regular expression
(a) **(ab + aa + bb + ba)*** (b) **(a + b)*(a* + b)***
(c) **(aa + bb)*** (d) **(ab + ba)***

2. The set of all strings over $\{a, b\}$ of length 4, starting with an a is represented by the regular expression
(a) **a(a + b)*** (b) **a(ab)***
(c) **(ab + ba)(aa + bb)** (d) **a(a + b)(a + b)(a + b)**

3. **(0*1*)*** is the same as
(a) **(0 + 1)*** (b) **(01)***
(c) **(10)*** (d) none of these.

4. If L is the set of all strings over $\{a, b\}$ containing at least one a, then it is not represented by the regular expression
(a) **b*a(a + b)*** (b) **(a + b)*a(b + a)***
(c) **(a + b)*ab*** (d) **(a + b)*a**

5. $\{a^{2n} \mid n \geq 1\}$ is represented by the regular expression
(a) **(aa)*** (b) **a***
(c) **aa*a** (d) **a*a***

6. The set of strings over $\{a, b\}$ having exactly $3b$'s is represented by the regular expression
(a) **a*bbb** (b) **a*ba*ba*b**
(c) **ba*ba*b** (d) **a*ba*ba*ba***

7. The set of all strings over $\{a, b\}$ having *abab* as a substring is represented by
(a) **a*ababb*** (b) **(a + b)*abab(a + b)***
(c) **a*b*ababa*b*** (d) **(a + b)*abab**

8. **(a + a*)*** is equivalent to
(a) **a(a*)*** (b) **a***
(c) **aa*** (d) none of these.

9. **a*(a + b)*** is equivalent to
(a) **a* + b*** (b) **(ab)***
(c) **a*b*** (d) none of these.

10. **ab* + b*** represents all strings w over $\{a, b\}$
(a) starting with an a and having no other a's or having no a's but only b's
(b) starting with an a followed by b's
(c) having no a's but only b's
(d) none of these.

State whether the following Statements 11–17 are true or false.

11. If Σ is finite then Σ^* is finite.

12. Every finite subset of Σ^* is a regular language.

13. Every regular language over Σ is finite.

14. a^4b^3 is in the regular set given by $\mathbf{a^*(a + b)b^*}$.

15. $\mathbf{aa^* + bb^*}$ is the same as $\mathbf{(a + b)^*}$.

16. The set of all strings starting with an a and ending in ab is defined by the regular expression $\mathbf{a(a + b)^*b}$.

17. The regular expression $\mathbf{(a + b)^*c^*}$ is the same as $\mathbf{a^*(b + c)^*}$.

EXERCISES

5.1 Represent the following sets by regular expressions:
(a) $\{0, 1, 2\}$.
(b) $\{1^{2n+1} \mid n > 0\}$.
(c) $\{w \in \{a, b\}^* \mid w$ has only one $a\}$.
(d) The set of all strings over $\{0, 1\}$ which has at most two zeros.
(e) $\{a^2, a^5, a^8, \ldots\}$.
(f) $\{a^n \mid n$ is divisible by 2 or 3 or $n = 5\}$.
(g) The set of all strings over $\{a, b\}$ beginning and ending with a.

5.2 Find all strings of length 5 or less in the regular set represented by the following regular expressions:
(a) $\mathbf{(ab + a)^*(aa + b)}$
(b) $\mathbf{(a^*b + b^*a)^*a}$
(c) $\mathbf{a^* + (ab + a)^*}$

5.3 Describe, in the English language, the sets represented by the following regular expressions:
(a) $\mathbf{a(a + b)^*ab}$
(b) $\mathbf{a^*b + b^*a}$
(c) $\mathbf{(aa + b)^*(bb + a)^*}$

5.4 Prove the following identity:

$$\mathbf{(a^*ab + ba)^*a^* = (a + ab + ba)^*}$$

5.5 Construct the transition systems equivalent to the regular expressions given in Exercise 5.2.

5.6 Construct the transition systems equivalent to the regular expressions given in Exercise 5.3.

5.7 Find the set of strings over $\Sigma = \{a, b\}$ recognized by the transition systems shown in Fig. 5.37(a–d).

5.8 Find the regular expression corresponding to the automaton given in Fig. 5.38.

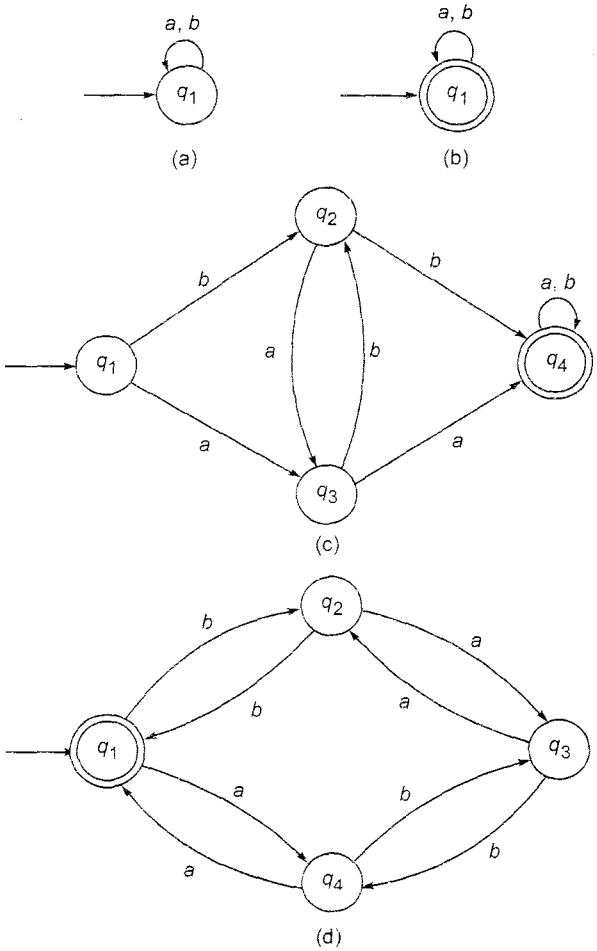


Fig. 5.37 Transition systems of Exercise 5.7.

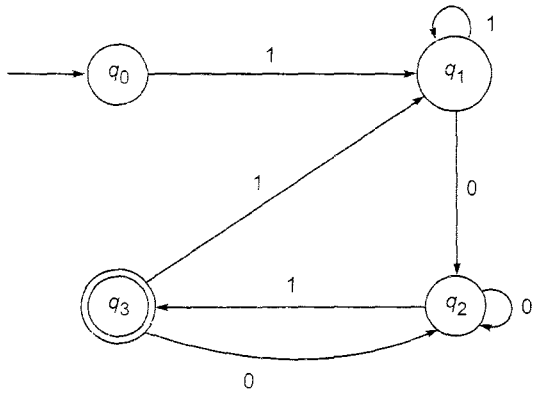


Fig. 5.38 Transition system of Exercise 5.8.

5.9 Construct a transition system corresponding to the regular expressions (i) **(ab + c*)*b** and (ii) **a + bb + bab*a**.

5.10 Find the regular expressions representing the following sets:
(a) The set of all strings over {0, 1} having at most one pair of 0's or at most one pair of 1's.
(b) The set of all strings over $\{a, b\}$ in which the number of occurrences of a is divisible by 3.
(c) The set of all strings over $\{a, b\}$ in which there are at least two occurrences of b between any two occurrences of a.
(d) The set of all strings over $\{a, b\}$ with three consecutive b's.
(e) The set of all strings over {0, 1} beginning with 00.
(f) The set of all strings over {0, 1} ending with 00 and beginning with 1.

5.11 Construct a deterministic finite automaton corresponding to the regular expression given in Exercise 5.2.

5.12 Construct a finite automaton accepting all strings over {0, 1} ending in 010 or 0010.

5.13 Construct a finite automaton M which can recognize DFA in a given string over the alphabet $\{A, B, \ldots, Z\}$. For example, M has to recognize DFA in the string ATXDFAMN.

5.14 Construct a finite automaton for the regular expression **(a + b)*abb**.

5.15 Show that there exists no finite automaton accepting all palindromes over $\{a, b\}$.

5.16 Show that $\{a^n b^n \mid n > 0\}$ is not a regular set without using the pumping lemma.

5.17 Using the pumping lemma, show that the following sets are not regular:
(a) $\{a^n b^{2n} \mid n > 0\}$;
(b) $\{a^n b^m \mid 0 < n < m\}$.

5.18 Show that $\{0^n 1^n \mid \text{g.c.d. } (m, n) = 1\}$ is not regular.

5.19 Show that a deterministic finite automaton with n states accepting a nonempty set accepts a string of length m, $m < n$.

5.20 Construct a finite automaton recognizing $L(G)$, where G is the grammar $S \rightarrow aS \mid bA \mid b$ and $A \rightarrow aA \mid bS \mid a$.

5.21 Find a regular grammar accepting the set recognized by the finite automaton given in Fig. 5.37(c).

5.22 Construct a regular grammar which can generate the set of all strings starting with a letter (A to Z) followed by a string of letters or digits (0 to 9).

5.23 Are the following true or false? Support your answer by giving proofs or counter-examples.

(a) If $L_1 \cup L_2$ is regular and L_1 is regular, then L_2 is regular.

(b) If L_1L_2 is regular and L_1 is regular, then L_2 is regular.

(c) If L^* is regular, then L is regular.

5.24 Construct a deterministic finite automaton equivalent to the grammar $S \rightarrow aS \mid bS \mid aA$, $A \rightarrow bB$, $B \rightarrow aC$, $C \rightarrow \Lambda$.

6 Context-Free Languages

In this chapter we study context-free grammars and languages. We define derivation trees and give methods of simplifying context-free grammars. The two normal forms—Chomsky normal form and Greibach normal form—are dealt with. We conclude this chapter after proving pumping lemma and giving some decision algorithms.

6.1 CONTEXT-FREE LANGUAGES AND DERIVATION TREES

Context-free languages are applied in parser design. They are also useful for describing block structures in programming languages. It is easy to visualize derivations in context-free languages as we can represent derivations using tree structures.

Let us recall the definition of a context-free grammar (CFG). G is context-free if every production is of the form $A \rightarrow \alpha$, where $A \in V_N$ and $\alpha \in (V_N \cup \Sigma)^*$.

EXAMPLE 6.1

Construct a context-free grammar G generating all integers (with sign).

Solution

Let

$$G = (V_N, \Sigma, P, S)$$

where

$$V_N = \{S, \text{(sign)}, \text{(digit)}, \text{(Integer)}\}$$

$$\Sigma = \{0, 1, 2, 3, \ldots, 9, +, -\}$$

P consists of $S \rightarrow \langle \text{sign} \rangle \langle \text{integer} \rangle$, $\langle \text{sign} \rangle \rightarrow + \mid -$,

$$\langle \text{integer} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{integer} \rangle \mid \langle \text{digit} \rangle$$

$$\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

$L(G)$ = the set of all integers. For example, the derivation of -17 can be obtained as follows:

$$\begin{aligned} S &\Rightarrow \langle \text{sign} \rangle \langle \text{integer} \rangle \Rightarrow - \langle \text{integer} \rangle \\ &\Rightarrow - \langle \text{digit} \rangle \langle \text{integer} \rangle \Rightarrow - 1 \langle \text{integer} \rangle \Rightarrow - 1 \langle \text{digit} \rangle \\ &\Rightarrow - 17 \end{aligned}$$

6.1.1 Derivation Trees

The derivations in a CFG can be represented using trees. Such trees representing derivations are called derivation trees. We give below a rigorous definition of a derivation tree.

Definition 6.1 A derivation tree (also called a parse tree) for a CFG $G = (V_N, \Sigma, P, S)$ is a tree satisfying the following conditions:

(i) Every vertex has a label which is a variable or terminal or Λ.
(ii) The root has label S.
(iii) The label of an internal vertex is a variable.
(iv) If the vertices $n_1, n_2, \ldots, n_k$ written with labels $X_1, X_2, \ldots, X_k$ are the sons of vertex n with label A, then $A \rightarrow X_1X_2 \ldots X_k$ is a production in P.
(v) A vertex n is a leaf if its label is $a \in \Sigma$ or Λ; n is the only son of its father if its label is Λ.

For example, let $G = (\{S, A\}, \{a, b\}, P, S)$, where P consists of $S \rightarrow aAS \mid a \mid SS$, $A \rightarrow SbA \mid ba$. Figure 6.1 is an example of a derivation tree.

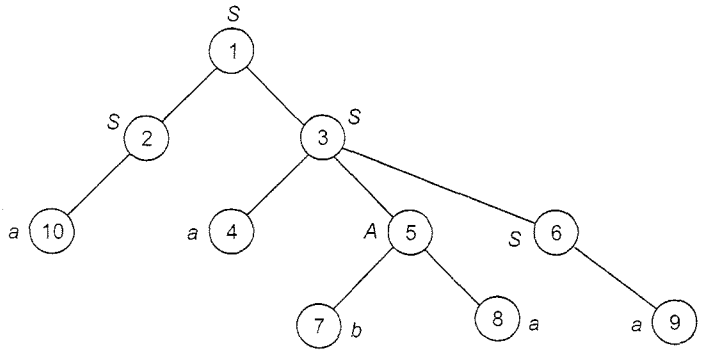


Fig. 6.1 An example of a derivation tree.

Note: Vertices 4-6 are the sons of 3 written from the left, and $S \rightarrow aAS$ is in P. Vertices 7 and 8 are the sons of 5 written from the left, and $A \rightarrow ba$ is a production in P. The vertex 5 is an internal vertex and its label is A, which is a variable.

Ordering of Leaves from the Left

We can order all the vertices of a tree in the following way: The successors of the root (i.e. sons of the root) are ordered from the left by the definition (refer to Section 1.2). So the vertices at level 1 are ordered from the left. If v_1 and v_2 are any two vertices at level 1 and v_1 is to the left of v_2, then we say that v_1 is to the left of any son of v_2. Also, any son of v_1 is to the left of v_2 and to the left of any son of v_2. Thus we get a left-to-right ordering of vertices at level 2. Repeating the process up to level k, where k is the height of the tree, we have an ordering of all vertices from the left.

Our main interest is in the ordering of leaves.

In Fig. 6.1, for example, the sons of the root are 2 and 3 ordered from the left. So, the son of 2, namely 10, is to the left of any son of 3. The sons of 3 ordered from the left are 4-5-6. The vertices at level 2 in the left-to-right ordering are 10-4-5-6. The vertex 4 is to the left of 6. The sons of 5 ordered from the left are 7-8. So 4 is to the left of 7. Similarly, 8 is to the left of 9. Thus the order of the leaves from the left is 10-4-7-8-9.

Note: If we draw the sons of any vertex keeping in mind the left-to-right ordering, we get the left-to-right ordering of leaves by 'reading' the leaves in the anticlockwise direction.

Definition 6.2 The yield of a derivation tree is the concatenation of the labels of the leaves without repetition in the left-to-right ordering.

The yield of the derivation tree of Fig. 6.1, for example, is *aabaa*.

Note: Consider the derivation tree in Fig. 6.1. As the sons of 1 are 2-3 in the left-to-right ordering, by condition (iv) of Definition 6.1, we have the production $S \rightarrow SS$. By applying the condition (iv) to other vertices, we get the productions $S \rightarrow a$, $S \rightarrow aAS$, $A \rightarrow ba$ and $S \rightarrow a$. Using these productions, we get the following derivation:

$$S \Rightarrow SS \Rightarrow as \Rightarrow aaAS \Rightarrow aabaS \Rightarrow aabaa$$

Thus the yield of the derivation tree is a sentential form in G.

Definition 6.3 A subtree of a derivation tree T is a tree (i) whose root is some vertex v of T, (ii) whose vertices are the descendants of v together with their labels, and (iii) whose edges are those connecting the descendants of v.

Figures 6.2 and 6.3, for example, give two subtrees of the derivation tree shown in Fig. 6.1.

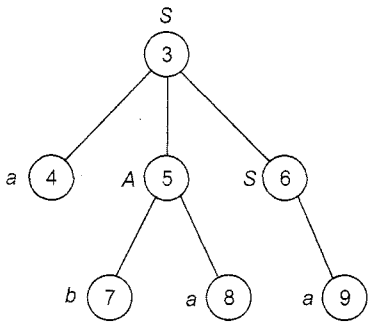


Fig. 6.2 A subtree of Fig. 6.1.

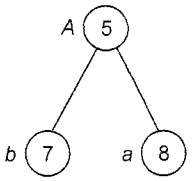


Fig. 6.3 Another subtree of Fig. 6.1.

Note: A subtree looks like a derivation tree except that the label of the root may not be S. It is called an A-tree if the label of its root is A.

Remark When there is no need for numbering the vertices, we represent the vertices by points. The following theorem asserts that sentential forms in CFG G are precisely the yields of derivation trees for G.

Theorem 6.1 Let $G = (V_N, \Sigma, P, S)$ be a CFG. Then $S \overset{*}{\Rightarrow} \alpha$ if and only if there is a derivation tree for G with yield α.

Proof We prove that $A \overset{*}{\Rightarrow} \alpha$ if and only if there is an A-tree with yield α. Once this is proved, the theorem follows by assuming that $A = S$.

Let α be the yield of an A-tree T. We prove that $A \overset{*}{\Rightarrow} \alpha$ by induction on the number of internal vertices in T.

When the tree has only one internal vertex, the remaining vertices are leaves and are the sons of the root. This is illustrated in Fig. 6.4.

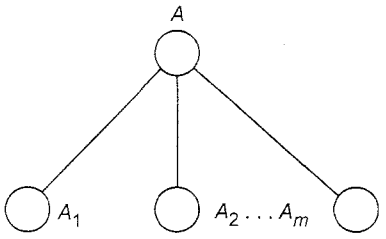


Fig. 6.4 A tree with only one internal vertex.

By condition (iv) of Definition 6.1, $A \to A_1A_2 \ldots A_m = \alpha$ is a production in G, i.e. $A \Rightarrow \alpha$. Thus there is basis for induction. Now assume the result for all trees with at most $k - 1$ internal vertices ($k > 1$).

Let T be an A-tree with k internal vertices ($k \geq 2$). Let $v_1, v_2, \ldots, v_m$ be the sons of the root in the left-to-right ordering. Let their labels be $X_1, X_2, \ldots, X_m$. By condition (iv) of Definition 6.1, $A \to X_1X_2 \ldots X_m$ is in P, and so

$$A \Rightarrow X_1X_2 \ldots X_m \tag{6.1}$$

As $k \geq 2$, at least one of the sons is an internal vertex. By the left-to-right ordering of leaves, α can be written as $\alpha_1\alpha_2 \ldots \alpha_m$, where α_i is obtained by the concatenation of the labels of the leaves which are descendants of vertex v_i. If v_i is an internal vertex, consider the subtree of T with v_i as its root. The number of internal vertices of the subtree is less than k (as there are k internal vertices in T and at least one of them, viz. its root, is not in the subtree). So by induction hypothesis applied to the subtree, $X_i \overset{*}{\Rightarrow} \alpha_i$. If v_i is not an internal vertex, i.e. a leaf, then $X_i = \alpha_i$.

Using (6.1), we get

$$A \Rightarrow X_1X_2 \ldots X_m \overset{*}{\Rightarrow} \alpha_1X_2X_3 \ldots X_m \ldots \overset{*}{\Rightarrow} \alpha_1\alpha_2 \ldots \alpha_m = \alpha,$$

i.e. $A \overset{*}{\Rightarrow} \alpha$. By the principle of induction, $A \overset{*}{\Rightarrow} \alpha$ whenever α is the yield of an A-tree.

To prove the 'only if' part, let us assume that $A \overset{*}{\Rightarrow} \alpha$. We have to construct an A-tree whose yield is α. We do this by induction on the number of steps in $A \overset{*}{\Rightarrow} \alpha$.

When $A \Rightarrow \alpha$, $A \rightarrow \alpha$ is a production in P. If $\alpha = X_1X_2 \ldots X_m$, the A-tree with yield α is constructed and given as in Fig. 6.5. So there is basis for induction. Assume the result for derivations in at most k steps. Let $A \overset{k}{\Rightarrow} \alpha$; we can split this as $A \Rightarrow X_1 \ldots X_m \overset{k-1}{\Rightarrow} \alpha$. Now, $A \Rightarrow X_1 \ldots X_m$ implies $A \rightarrow X_1X_2 \ldots X_m$ is a production in P. In the derivation $X_1X_2 \ldots X_m \overset{k-1}{\Rightarrow} \alpha$, either (i) X_i is not changed throughout the derivation, or (ii) X_i is changed in some subsequent step. Let α_i be the substring of α derived from X_i. Then $X_i \overset{*}{\Rightarrow} \alpha_i$ in (ii) and $X_i = \alpha_i$ in (i). As G is context-free, in every step of the derivation $X_1X_2 \ldots X_m \overset{*}{\Rightarrow} \alpha$, we replace a single variable by a string. As $\alpha_1, \alpha_2, \ldots, \alpha_m$, account for all the symbols in α, we have $\alpha = \alpha_1\alpha_2 \ldots \alpha_m$.

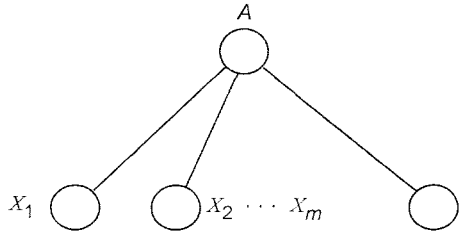


Fig. 6.5 Derivation tree for one-step derivation.

We construct the derivation tree with yield α as follows: As $A \rightarrow X_1 \ldots X_m$ is in P, we construct a tree with m leaves whose labels are $X_1, \ldots, X_m$ in the left-to-right ordering. This tree is given in Fig. 6.6. In (i) above, we leave the vertex v_i as it is. In (ii), $X_i \overset{*}{\Rightarrow} \alpha_i$ is less than k steps (as $X_1 \ldots X_m \overset{n-1}{\Rightarrow} a$). By induction hypothesis there exists an X_i-tree T_i with yield α_i. We attach the tree T_i at the vertex v_i (i.e. v_i is the root of T_i). The resulting tree is given in Fig. 6.7. In this figure, let i and j be the first and the last indexes such that X_i and X_j satisfy (ii). So, $\alpha_1 \ldots \alpha_{i-1}$ are the labels of leaves at level 1 in T. α_i is the yield of the X_i-tree T_i, etc.

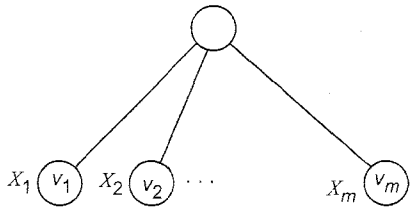


Fig. 6.6 Derivation tree with yield $X_1 X_2 \ldots X_m$.

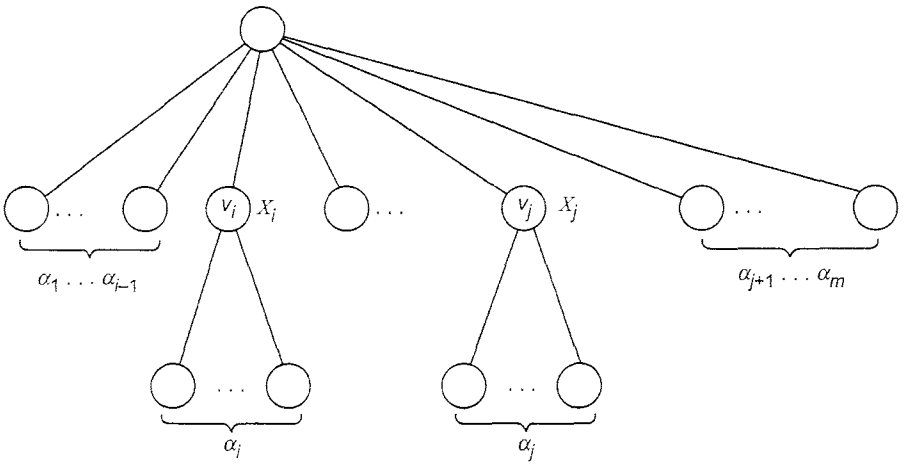


Fig. 6.7 Derivation tree with yield $\alpha_1\alpha_2 \ldots \alpha_n$.

Thus we get a derivation tree with yield α. By the principle of induction we can get the result for any derivation. This completes the proof of 'only if' part. ∎

Note: The derivation tree does not specify the order in which we apply the productions for getting α. So, the same derivation tree can induce several derivations.

The following remark is used quite often in proofs and constructions involving CFGs.

Remark If A derives a terminal string w and if the first step in the derivation is $A \Rightarrow A_1A_2 \ldots A_n$, then we can write w as $w_1w_2 \ldots w_n$ so that $A_i \overset{*}{\Rightarrow} w_i$. (Actually, in the derivation tree for w, the ith son of the root has the label A_i, and w_i is the yield of the subtree whose root is the ith son.)

EXAMPLE 6.2

Consider G whose productions are $S \to aAS \mid a$, $A \to SbA \mid SS \mid ba$. Show that $S \overset{*}{\Rightarrow} aabbaa$ and construct a derivation tree whose yield is $aabbaa$.

Solution

$$S \Rightarrow aAS \Rightarrow asbAS \Rightarrow aabAS \Rightarrow a^2bbaS \Rightarrow a^2b^2a^2 \qquad (6.2)$$

Hence, $S \overset{*}{\Rightarrow} a^2b^2a^2$. The derivation tree is given in Fig. 6.8.

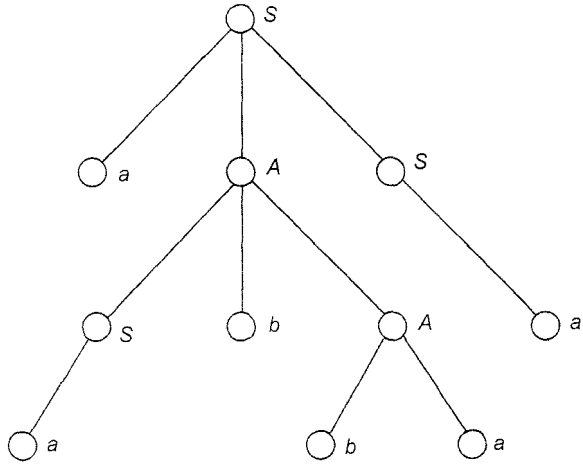


Fig. 6.8 The derivation tree with yield *aabbaa* for Example 6.2.

Note: Consider G as given in Example 6.2. We have seen that $S \overset{*}{\Rightarrow} a^2b^2a^2$, and (6.2) gives a derivation of $a^2b^2a^2$.

Another derivation of $a^2b^2a^2$ is

$$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa \qquad (6.3)$$

Yet another derivation of $a^2b^2a^2$ is

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aSbAa \Rightarrow aabAa \Rightarrow aabbaa \qquad (6.4)$$

In derivation (6.2), whenever we replace a variable X using a production, there are no variables to the left of X. In derivation (6.3), there are no variables to the right of X. But in (6.4), no such conditions are satisfied. These lead to the following definitions.

Definition 6.4 A derivation $A \overset{*}{\Rightarrow} w$ is called a *leftmost* derivation if we apply a production only to the leftmost variable at every step.

Definition 6.5 A derivation $A \overset{*}{\Rightarrow} w$ is a *rightmost* derivation if we apply production to the rightmost variable at every step.

Relation (6.2), for example, is a leftmost derivation. Relation (6.3) is a rightmost derivation. But (6.4) is neither leftmost nor rightmost. In the second step of (6.4), the rightmost variable S is not replaced. So (6.4) is not a rightmost derivation. In the fourth step, the leftmost variable S is not replaced. So (6.4) is not a leftmost derivation.

Theorem 6.2 If $A \overset{*}{\Rightarrow} w$ in G, then there is a leftmost derivation of w.

Proof We prove the result for every A in V_N by induction on the number of steps in $A \overset{*}{\Rightarrow} w$. $A \Rightarrow w$ is a leftmost derivation as the L.H.S. has only one variable. So there is basis for induction. Let us assume the result for derivations in atmost k steps. Let $A \overset{n+1}{\Rightarrow} w$. The derivation can be split as $A \Rightarrow X_1X_2 \ldots X_m \overset{k}{\Rightarrow} w$.

The string w can be split as $w_1w_2 \ldots w_m$ such that $X_i \Rightarrow w_i$ (see the Remark appended before Example 6.2). As $X_i \overset{*}{\Rightarrow} w_i$ involves atmost k steps by induction hypothesis, we can find a leftmost derivation of w_i. Using these leftmost derivations, we get a leftmost derivation of w given by

$$A \Rightarrow X_1X_2 \ldots X_m \overset{*}{\Rightarrow} w_1X_2 \ldots X_m \overset{*}{\Rightarrow} w_1w_2X_3 \ldots X_m \ldots \overset{*}{\Rightarrow} w_1w_2 \ldots w_m$$

Hence by induction the result is true for all derivations $A \overset{*}{\Rightarrow} w$. ∎

Corollary Every derivation tree of w induces a leftmost derivation of w.

Once we get some derivation of w, it is easy to get a leftmost derivation of w in the following way: From the derivation tree for w, at every level consider the productions for the variables at that level, taken in the left-to-right ordering. The leftmost derivation is obtained by applying the productions in this order.

EXAMPLE 6.3

Let G be the grammar $S \to 0B \mid 1A$, $A \to 0 \mid 0S \mid 1AA$, $B \to 1 \mid 1S \mid 0BB$. For the string 00110101, find (a) the leftmost derivation, (b) the rightmost derivation, and (c) the derivation tree.

Solution

(a) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011S$
$\Rightarrow 0^21^20B \Rightarrow 0^21^201S \Rightarrow 0^21^2010B \Rightarrow 0^21^20101$

(b) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1S \Rightarrow 00B10B$
$\Rightarrow 0^2B101S \Rightarrow 0^2B1010B \Rightarrow 0^2B10101 \Rightarrow 0^2110101.$

(c) The derivation tree is given in Fig. 6.9.

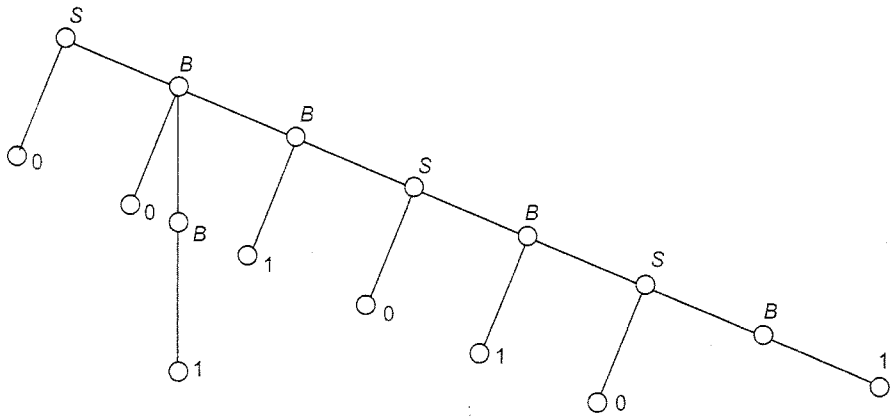


Fig. 6.9 The derivation tree with yield 00110101 for Example 6.3.

6.2 AMBIGUITY IN CONTEXT-FREE GRAMMARS

Sometimes we come across ambiguous sentences in the language we are using. Consider the following sentence in English: "In books selected information is given." The word 'selected' may refer to books or information. So the sentence may be parsed in two different ways. The same situation may arise in context-free languages. The same terminal string may be the yield of two derivation trees. So there may be two different leftmost derivations of w by Theorem 6.2. This leads to the definition of ambiguous sentences in a context-free language.

Definition 6.6 A terminal string $w \in L(G)$ is ambiguous if there exist two or more derivation trees for w (or there exist two or more leftmost derivations of w).

Consider, for example, $G = (\{S\}, \{a, b, +, *\}, P, S)$, where P consists of $S \to S + S \mid S * S \mid a \mid b$. We have two derivation trees for $a + a * b$ given in Fig. 6.10.

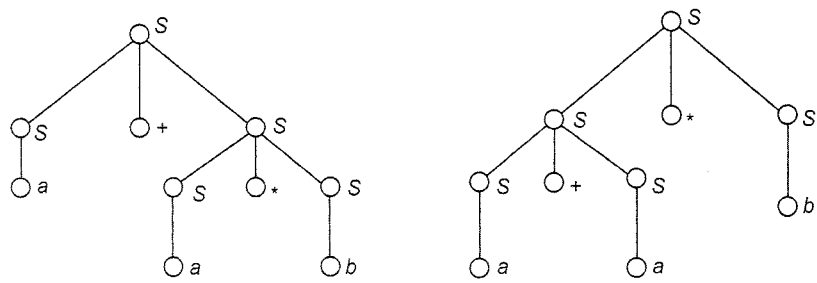


Fig. 6.10 Two derivation trees for $a + a * b$.

The leftmost derivations of $a + a * b$ induced by the two derivation trees are

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$$

$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$$

Therefore, $a + a * b$ is ambiguous.

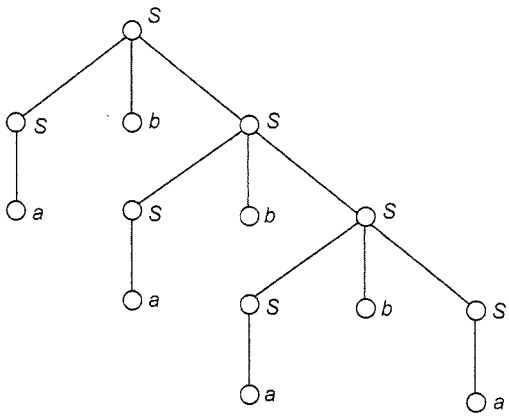
Definition 6.7 A context-free grammar G is ambiguous if there exists some $w \in L(G)$, which is ambiguous.

EXAMPLE 6.4

If G is the grammar $S \to SbS \mid a$, show that G is ambiguous.

Solution

To prove that G is ambiguous, we have to find a $w \in L(G)$, which is ambiguous. Consider $w = abababa \in L(G)$. Then we get two derivation trees for w (see Fig. 6.11). Thus, G is ambiguous.



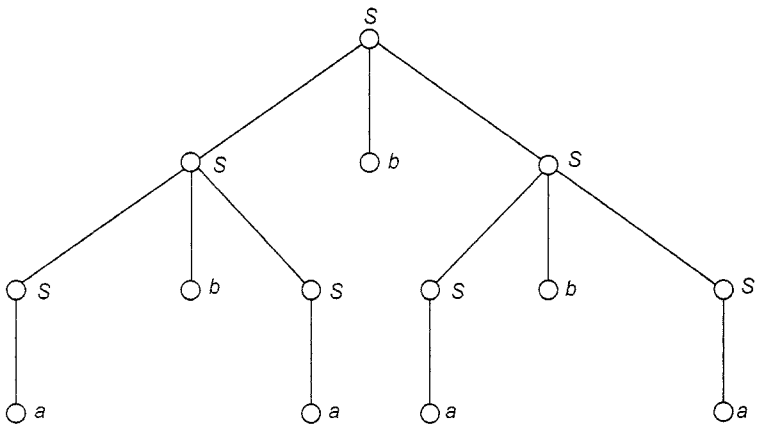


Fig. 6.11 Two derivation trees of *abababa* for Example 6.4.

6.3 SIMPLIFICATION OF CONTEXT-FREE GRAMMARS

In a CFG G, it may not be necessary to use all the symbols in $V_N \cup \Sigma$, or all the productions in P for deriving sentences. So when we study a context-free language $L(G)$, we try to eliminate those symbols and productions in G which are not useful for the derivation of sentences.

Consider, for example,

$$G = (\{S, A, B, C, E\}, \{a, b, c\}, P, S)$$

where

$$P = \{S \to AB, A \to a, B \to b, B \to C, E \to c \mid \Lambda\}$$

It is easy to see that $L(G) = \{ab\}$. Let $G' = (\{S, A, B\}, \{a, b\}, P', S)$, where P' consists of $S \to AB$, $A \to a$, $B \to b$. $L(G) = L(G')$. We have eliminated the symbols C, E and c and the productions $B \to C$, $E \to c \mid \Lambda$. We note the

following points regarding the symbols and productions which are eliminated:

(i) C does not derive any terminal string.
(ii) E and c do not appear in any sentential form.
(iii) $E \rightarrow \Lambda$ is a null production.
(iv) $B \rightarrow C$ simply replaces B by C.

In this section, we give the construction to eliminate (i) variables not deriving terminal strings, (ii) symbols not appearing in any sentential form, (iii) null productions, and (iv) productions of the form $A \rightarrow B$.

6.3.1 Construction of Reduced Grammars

Theorem 6.3 If G is a CFG such that $L(G) \neq \emptyset$, we can find an equivalent grammar G' such that each variable in G' derives some terminal string.

Proof *Let* $G = (V_N, \Sigma, P, S)$. We define $G' = (V'_N, \Sigma, G', S)$ as follows:

(a) *Construction of* V'_N:

We define $W_i \subseteq V_N$ by recursion:

$W_1 = \{A \in V_N | \text{there exists a production } A \rightarrow w \text{ where } w \in \Sigma^*\}$. (If $W_1 = \emptyset$, some variable will remain after the application of any production, and so $L(G) = \emptyset$.)

$$W_{i+1} = W_i \cup \{A \in V_N | \text{ there exists some production } A \rightarrow \alpha \text{ with } \alpha \in (\Sigma \cup W_i)^*\}$$

By the definition of W_i, $W_i \subseteq W_{i+1}$ for all i. As V_N has only a finite number of variables, $W_k = W_{k+1}$ for some $k \leq |V_N|$. Therefore, $W_k = W_{k+j}$ for $j \geq 1$. We define $V'_N = W_k$.

(b) *Construction of* P':

$$P' = \{A \rightarrow \alpha | A, \alpha \in (V'_N \cup \Sigma)^*\}$$

We can define $G' = (V'_N, \Sigma, P', S)$. S is in V_N. (We are going to prove that every variable in V_N derives some terminal string. So if $S \notin V_N$, $L(G) = \emptyset$. But $L(G) \neq \emptyset$.)

Before proving that G' is the required grammar, we apply the construction to an example.

EXAMPLE 6.5

Let $G = (V_N, \Sigma, P, S)$ be given by the productions $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$, $B \rightarrow C$, $E \rightarrow c$. Find G' such that every variable in G' derives some terminal string.

Solution

(a) *Construction of* V'_N:

$W_1 = \{A, B, E\}$ since $A \rightarrow a$, $B \rightarrow b$, $E \rightarrow c$ are productions with a terminal string on the R.H.S.

$$W_2 = W_1 \cup \{A_1 \in V_N | A_1 \to \alpha \text{ for some } \alpha \in (\Sigma \cup \{A, B, E\})^*\}$$
$$= W_1 \cup \{S\} = \{A, B, E, S\}$$
$$W_3 = W_2 \cup \{A_1 \in V_N | A_1 \to \alpha \text{ for some } \alpha \in (\Sigma \cup \{S, A, B, E\})^*\}$$
$$= W_2 \cup \emptyset = W_2$$

Therefore,

$$V'_N = \{S, A, B, F\}$$

(b) *Construction of P':*

$$P' = \{A_1 \to \alpha | A_1, \alpha \in (V'_N \cup \Sigma)^*\}$$
$$= \{S \to AB, A \to a, B \to b, E \to c\}$$

Therefore,

$$G' = (\{S, A, B, E\}, \{a, b, c\}, P', S)$$

Now we prove:

(i) If each $A \in V'_N$, then $A \overset{*}{\underset{G'}{\Rightarrow}} w$ for some $w \in \Sigma^*$; conversely, if $A \overset{*}{\underset{G}{\Rightarrow}} w$, then $A \in V'_N$.
(ii) $L(G') = L(G)$.

To prove (i) we note that $W_k = W_1 \cup W_2 \ldots \cup W_k$. We prove by induction on i that for $i = 1, 2, \ldots, k$, $A \in W_i$ implies $A \overset{*}{\underset{G'}{\Rightarrow}} w$ for some $w \in \Sigma^*$. If $A \in W_1$, then $A \underset{G}{\Rightarrow} w$. So the production $A \to w$ is in P'. Therefore, $A \overset{*}{\underset{G'}{\Rightarrow}} w$. Thus there is basis for induction. Let us assume the result for i. Let $A \in W_{i+1}$. Then either $A \in W_i$, in which case, $A \overset{*}{\underset{G'}{\Rightarrow}} w$ for some $w \in \Sigma^*$ by induction hypothesis. Or, there exists a production $A \to \alpha$ with $\alpha \in (\Sigma \cup w_i)^*$. By definition of P', $A \to \alpha$ is in P'. We can write $\alpha = X_1 X_2 \ldots X_m$ where $X_j \in \Sigma \cup W_i$. If $X_j \in W_i$ by induction hypothesis, $X_j \overset{*}{\underset{G'}{\Rightarrow}} w_j$ for some $w_j \in \Sigma^*$. So, $A \overset{*}{\underset{G'}{\Rightarrow}} w_1 w_2 \ldots w_m \in \Sigma^*$ (when X_j is a terminal, $w_j = X_j$). By induction the result is true for $i = 1, 2, \ldots, k$.

The converse part can be proved in a similar way by induction on the number of steps in the derivation $A \overset{*}{\underset{G}{\Rightarrow}} w$. We see immediately that $L(G') \subseteq L(G)$ as $V'_N \subseteq V_N$ and $P' \subseteq P$. To prove $L(G) \subseteq L(G')$, we need an auxiliary result

$$A \overset{*}{\underset{G'}{\Rightarrow}} w \qquad \text{if } A \overset{*}{\underset{G}{\Rightarrow}} w \text{ for some } w \in \Sigma^* \tag{6.5}$$

We prove (6.5) by induction on the number of steps in the derivation $A \overset{*}{\underset{G}{\Rightarrow}} w$. If $A \underset{G}{\Rightarrow} w$, then $A \to w$ is in P and $A \in W_1 \subseteq V'_N$. As $A \in V'_N$ and $w \in \Sigma^*$, $A \to w$ is in P'. So $A \underset{G'}{\Rightarrow} w$, and there is basis for induction. Assume (6.5) for derivations in at most k steps. Let $A \overset{k+1}{\underset{G}{\Rightarrow}} w$. By Remark appearing after

Theorem 6.1, we can split this as $A \underset{G}{\Rightarrow} X_1X_2 \ldots X_m \overset{*}{\underset{G}{\Rightarrow}} w_1w_2 \ldots w_m$ such that $X_j \overset{*}{\underset{G}{\Rightarrow}} w_j$. If $X_j \in \Sigma$, then $w_j = X_j$.

If $X_j \in V_N$ then by (i), $X_j \in V'_N$. As $X_j \overset{*}{\underset{G}{\Rightarrow}} w_j$ in at most k steps, $X_j \overset{*}{\underset{G'}{\Rightarrow}} w_j$. Also, $X_1, X_2, X_m \in (\Sigma \cup V'_N)^*$ implies that $A \to X_1X_2 \ldots X_m$ is in P'. Thus, $A \underset{G'}{\Rightarrow} X_1X_2 \ldots X_m \overset{*}{\underset{G'}{\Rightarrow}} w_1w_2 \ldots w_m$. Hence by induction, (6.5) is true for all derivations. In particular, $S \overset{*}{\underset{G}{\Rightarrow}} w$ implies $S \overset{*}{\underset{G'}{\Rightarrow}} w$. This proves that $L(G) \subseteq L(G')$, and (ii) is completely proved. ∎

Theorem 6.4 For every CFG $G = (V_N, \Sigma, P, S)$, we can construct an equivalent grammar $G' = (V_N, \Sigma', P', S)$ such that every symbol in $V_N \cup \Sigma'$ appears in some sentential form (i.e. for every X in $V'_N \cup \Sigma'$ there exists α such that $S \overset{*}{\underset{G'}{\Rightarrow}} \alpha$ and X is a symbol in the string α).

Proof We construct $G' = (V'_N, \Sigma', P', S)$ as follows:

(a) *Construction of W_i for $i \geq 1$:*

(i) $W_1 = \{S\}$.

(ii) $W_{i+1} = W_i \cup \{X \in V_N \cup \Sigma \mid$ there exists a production $A \to \alpha$ with $A \in W_i$ and α containing the symbol $X\}$.

We may note that $W_i \subseteq V_N \cup \Sigma$ and $W_i \subseteq W_{i+1}$. As we have only a finite number of elements in $V_N \cup \Sigma$, $W_k = W_{k+1}$ for some k. This means that $W_k = W_{k+j}$ for all $j \geq 0$.

(b) *Construction of V'_N, Σ' and P':*

We define

$$V'_N = V_N \cap W_k, \qquad \Sigma' = \Sigma \cup W_k$$

$$P' = \{A \to \alpha \mid A \in W_k\}.$$

Before proving that G' is the required grammar, we apply the construction to an example.

EXAMPLE 6.6

Consider $G = (\{S, A, B, E\}, \{a, b, c\}, P, S)$, where P consists of $S \to AB$, $A \to a$, $B \to b$, $E \to c$.

Solution

$$W_1 = \{S\}$$

$$W_2 = \{S\} \cup \{X \in V_N \cup \Sigma \mid \text{there exists a production } A \to \alpha \text{ with } A \in W_1 \text{ and } \alpha \text{ containing } X\}$$

$$= \{S\} \cup \{A, B\}$$

$W_3 = \{S, A, B\} \cup \{a, b\}$

$W_4 = W_3$

$V'_N = \{S, A, B\} \qquad \Sigma' = \{a, b\}$

$P' = \{S \to AB, A \to a, B \to b\}$

Thus the required grammar is $G' = (V'_N, \Sigma', P', S)$.

To complete the proof, we have to show that (i) every symbol in $V'_N \cup \Sigma'$ appears in some sentential form of G', and (ii) conversely, $L(G') = L(G)$.

To prove (i), consider $X \in V'_N \cup \Sigma' = W_k$. By construction $W_k = W_1 \cup W_2 \ldots \cup W_k$. We prove that $X \in W_i$, $i \leq k$, appears in some sentential form by induction on i. When $i = 1$, $X = S$ and $S \overset{*}{\underset{G'}{\Rightarrow}} S$. Thus, there is basis for induction. Assume the result for all variables in W_i. Let $X \in W_{i+1}$. Then either $X \in W_i$, in which case, X appears in some sentential form by induction hypothesis. Otherwise, there exists a production $A \to \alpha$, where $A \in W_i$ and α contains the symbol X_i. The A appears in some sentential form, say $\beta A \gamma$. Therefore,

$$S \overset{*}{\underset{G'}{\Rightarrow}} \beta A \gamma \underset{G'}{\Rightarrow} \beta \alpha \gamma$$

This means that $\beta\alpha\gamma$ is some sentential form and X is a symbol in $\beta\alpha\gamma$. Thus by induction the result is true for $X \in W_i$, $i \leq k$.

Conversely, if X appears in some sentential form, say $\beta X \gamma$, then $\Sigma \overset{l}{\underset{G}{\Rightarrow}} \beta X \gamma$. This implies $X \in W_l$. If $l \leq k$, then $W_l \subseteq W_k$. If $l > k$, then $W_l = W_k$. Hence X appears in $V'_N \cup \Sigma'$. This proves (i).

To prove (ii), we note $L(G') \subseteq L(G)$ as $V'_N \subseteq V_N$, $\Sigma' \subseteq \Sigma$ and $P' \subseteq P$. Let w be in $L(G)$ and $S = \alpha_1 \underset{G}{\Rightarrow} \alpha_2 \underset{G}{\Rightarrow} \alpha_3 = \ldots \Rightarrow \alpha_{n-1} \underset{G}{\Rightarrow} w$. We prove that every symbol in α_{i+1} is in W_{i+1} and $\alpha_i \underset{G'}{\Rightarrow} \alpha_{i+1}$ by induction on i. $\alpha_1 = S \underset{G}{\Rightarrow} \alpha_2$ implies $S \to \alpha_2$ is a production in P. By construction, every symbol in α_2 is in W_2 and $S \to \alpha_2$ is in P', i.e. $S \underset{G'}{\Rightarrow} \alpha_2$. Thus, there is basis for induction. Let us assume the result for i. Consider $\alpha_{i+1} \underset{G}{\Rightarrow} \alpha_{i+2}$. This one-step derivation can be written in the form

$$\beta_{i+1} A \gamma_{i+1} \Rightarrow \beta_{i+1} \alpha \gamma_{i+1}$$

where $A \to \alpha$ is the production we are applying. By induction hypothesis, $A \in W_{i+1}$. By construction of W_{i+2}, every symbol in α is in W_{i+2}. As all the symbols in β_{i+1} and γ_{i+1} are also in W_{i+1} by induction hypothesis, every symbol in $\beta_{i+1}\alpha\gamma_{i+1} = \alpha_{i+2}$ is in W_{i+2}. By the construction of P', $A \to \alpha$ is in P'. This means that $\alpha_{i+1} \underset{G'}{\Rightarrow} \alpha_{i+2}$. Thus the induction procedure is complete. So $S = \alpha_1 \underset{G'}{\Rightarrow} \alpha_2 \underset{G'}{\Rightarrow} \alpha_3 \overset{*}{\underset{G'}{\Rightarrow}} \ldots \alpha_{n-1} \underset{G'}{\Rightarrow} w$. Therefore, $w \in L(G')$. This proves (ii). ∎

Definition 6.8 Let $G = (V_N, \Sigma, P, S)$ be a CFG. G is said to be reduced or non-redundant if every symbol in $V_N \cup \Sigma$ appears in the course of the derivation of some terminal string, i.e. for every X in $V_N \cup \Sigma$, there exists a derivation $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w \in L(G)$. (We can say X is useful in the derivation of terminal strings.)

Theorem 6.5 For every CFG G there exists a reduced grammar G' which is equivalent to G.

Proof We construct the reduced grammar in two steps.

Step 1 We construct a grammar G_1 equivalent to the given grammar G so that every variable in G_1 derives some terminal string (Theorem 6.3).

Step 2 We construct a grammar $G' = (V'_N, \Sigma', P', S)$ equivalent to G_1 so that every symbol in G' appears in some sentential form of G' which is equivalent to G_1 and hence to G. G' is the required reduced grammar.

By step 2 every symbol X in G' appears in some sentential form, say $\alpha X \beta$. By step 1 every symbol in $\alpha X \beta$ derives some terminal string. Therefore, $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$ for some w in Σ^*, i.e. G' is reduced.

Note: To get a reduced grammar, we must first apply Theorem 6.3 and then Theorem 6.4. For, if we apply Theorem 6.4 first and then Theorem 6.3, we may not get a reduced grammar (refer to Exercise 6.8 at the end of the chapter).

EXAMPLE 6.7

Find a reduced grammar equivalent to the grammar G whose productions are

$$S \to AB \mid CA, \quad B \to BC \mid AB, \quad A \to a, \quad C \to aB \mid b$$

Solution

Step 1 $W_1 = \{A, C\}$ as $A \to a$ and $C \to b$ are productions with a terminal string on R.H.S.

$$W_2 = \{A, C\} \cup \{A_1 \mid A_1 \to \alpha \text{ with } \alpha \in (\Sigma \cup \{A, C\})^*\}$$
$$= \{A, C\} \cup \{S\} \text{ as we have } S \to CA$$
$$W_3 = \{A, C, S\} \cup \{A_1 \mid A_1 \to \alpha \text{ with } \alpha \in (\Sigma \cup \{S, A, C\})^*\}$$
$$= \{A, C, S\} \cup \emptyset$$

As $W_3 = W_2$,

$$V'_N = W_2 = \{S, A, C\}$$
$$P' = \{A_1 \to \alpha \mid A_1, \alpha \in (V'_N \cup \Sigma)^*\}$$
$$= \{S \to CA, A \to a, C \to b\}$$

Thus,

$$G_1 = (\{S, A, C\}, \{a, b\}, \{S \to CA, A \to a, C \to b\}, S)$$

Step 2 We have to apply Theorem 6.4 to G_1. Thus,

$$W_1 = \{S\}$$

As we have production $S \to CA$ and $S \in W_1$, $W_2 = \{S\} \cup \{A, C\}$
As $A \to a$ and $C \to b$ are productions with $A, C \in W_2$, $W_3 = \{S, A, C, a, b\}$

$$\text{As } W_3 = V'_N \cup \Sigma,\ P'' = \{S \to a \mid A_1 \in W_3\} = P'$$

Therefore,

$$G' = (\{S, A, C\}, \{a, b\}, \{S \to CA, A \to a, C \to b\}, S)$$

is the reduced grammar.

EXAMPLE 6.8

Construct a reduced grammar equivalent to the grammar

$$S \to aAa, \quad A \to Sb \mid bCC \mid DaA, \quad C \to abb \mid DD,$$
$$E \to aC, \quad D \to aDA$$

Solution

Step 1 $W_1 = \{C\}$ as $C \to abb$ is the only production with a terminal string on the R.H.S.

$$W_2 = \{C\} \cup \{E, A\}$$

as $E \to aC$ and $A \to bCC$ are productions with R.H.S. in $(\Sigma \cup \{C\})^*$

$$W_3 = \{C, E, A\} \cup \{S\}$$

as $S \to aAa$ and aAa is in $(\Sigma \cup W_2)^*$

$$W_4 = W_3 \cup \emptyset$$

Hence,

$$V'_N = W_3 = \{S, A, C, E\}$$
$$P' = \{A_1 \to \alpha \mid \alpha \in (V_N \cup \Sigma)^*\}$$
$$= \{S \to aAa, A \to Sb \mid bCC, C \to abb, E \to aC\}$$
$$G_1 = (V'_N, \{a, b\}, P', S)$$

Step 2 We have to apply Theorem 6.4 to G_1. We start with

$$W_1 = \{S\}$$

As we have $S \to aAa$,

$$W_2 = \{S\} \cup \{A, a\}$$

As $A \to Sb \mid bCC$,

$$W_3 = \{S, A, a\} \cup \{S, b, C\} = \{S, A, C, a, b\}$$

As we have $C \to abb$,

$$W_4 = W_3 \cup \{a, b\} = W_3$$

Hence,

$$P'' = \{A_1 \to \alpha \mid A_1 \in W_3\}$$
$$= \{S \to aAa,\ A \to Sb \mid bCC,\ C \to abb\}$$

Therefore,

$$G' = (\{S, A, C\}, \{a, b\}, P'', S)$$

is the reduced grammar.

6.3.2 Elimination of Null Productions

A context-free grammar may have productions of the form $A \to \Lambda$. The production $A \to \Lambda$ is just used to erase A. So a production of the form $A \to \Lambda$, where A is a variable, is called a *null production.* In this section we give a construction to eliminate null productions.

As an example, consider G whose productions are $S \to aS \mid aA \mid A$, $A \to \Lambda$. We have two null productions $S \to \Lambda$ and $A \to \Lambda$. We can delete $A \to \Lambda$ provided we erase A whenever it occurs in the course of a derivation of a terminal string. So we can replace $S \to aA$ by $S \to a$. If G_1 denotes the grammar whose produetions are $S \to aS \mid a \mid \Lambda$, then $L(G_1) = L(G) = \{a^n \mid n \geq 0\}$. Thus it is possible to eliminate the null production $A \to \Lambda$. If we eliminate $S \to \Lambda$, we cannot generate Λ in $L(G)$. But we can generate $L(G) - \{\Lambda\}$ even if we eliminate $S \to \Lambda$.

Before giving the construction we give a definition.

Definition 6.9 A variable A in a context-free grammar is nullable if $A \overset{*}{\Rightarrow} \Lambda$.

Theorem 6.6 If $G = (V_N, \Sigma, P, S)$ is a context-free grammar, then we can find a context-free grammar G_1 having no null prodctions such that $L(G_1) = L(G) - \{\Lambda\}$.

Proof We construct $G_1 = (V_N, \Sigma, P', S)$ as follows:

Step 1 *Construction of the set of nullable variables:*

We find the nullable variables recursively:

(i) $W_1 = \{A \in V_N \mid A \to \Lambda \text{ is in } P\}$

(ii) $W_{i+1} = W_i \cup \{A \in V_N \mid$ there exists a production $A \to \alpha$ with $\alpha \in W_i^*\}$.

By definition of W_i, $W_i \subseteq W_{i+1}$ for all i. As V_N is finite, $W_{k+1} = W_k$ for some $k \leq |V_N|$. So, $W_{k+j} = W_k$ for all j. Let $W = W_k$. W is the set of all nullable variables.

Step 2 (i) *Construction of P':*

Any production whose R.H.S. does not have any nullable variable is included in P'.

(ii) If $A \to X_1X_2 \ldots X_k$ is in P, the productions of the form $A \to \alpha_1\alpha_2 \ldots \alpha_k$ are included in P', where $\alpha_j = X_i$ if $X_i \notin W$. $\alpha_i = X_i$ or Λ if $X_i \in W$ and $\alpha_1\alpha_2 \ldots \alpha_k \neq \Lambda$. Actually, (ii) gives several productions in P'. The productions are obtained either by not erasing any nullable variable on the

R.H.S. of $A \to X_1X_2 \ldots X_k$ or by erasing some or all nullable variables provided some symbol appears on the R.H.S. after erasing.

Let $G_1 = (V_N, \Sigma, P', S)$. G_1 has no null productions.

Before proving that G_1 is the required grammar, we apply the construction to an example.

EXAMPLE 6.9

Consider the grammar G whose productions are $S \to aS \mid AB$, $A \to \Lambda$, $B \to \Lambda$, $D \to b$. Construct a grammar G_1 without null productions generating $L(G) - \{\Lambda\}$.

Solution

Step 1 *Construction of the set W of all nullable variables:*

$$W_1 = \{A_1 \in V_N \mid A_1 \to A \text{ is a production in } G\}$$
$$= \{A, B\}$$
$$W_2 = \{A, B\} \cup \{S\} \text{ as } S \to AB \text{ is a production with } AB \in W_1^*$$
$$= \{S, A, B\}$$
$$W_3 = W_2 \cup \emptyset = W_2$$

Thus,

$$W = W_2 = \{S, A, B\}$$

Step 2 *Construction of P′:*

(i) $D \to b$ is included in P'.
(ii) $S \to aS$ gives rise to $S \to aS$ and $S \to a$.
(iii) $S \to AB$ gives rise to $S \to AB$, $S \to A$ and $S \to B$.

(***Note:*** We cannot erase both the nullable variables A and B in $S \to AB$ as we will get $S \to \Lambda$ in that case.)

Hence the required grammar without null productions is

$$G_1 = (\{S, A, B, D\}, \{a, b\}, P, S)$$

where P' consists of

$$D \to b,\ S \to aS,\ S \to AB,\ S \to a,\ S \to A,\ S \to B$$

Step 3 $L(G_1) = L(G) - \{\Lambda\}$. To prove that $L(G) = L(G) - \{\Lambda\}$, we prove an auxiliary result given by the following relation:

For all $A \in V_N$ and $w \in \Sigma^*$,

$$A \underset{G_1}{\overset{*}{\Rightarrow}} w \text{ if and only if } A \underset{G}{\overset{*}{\Rightarrow}} w \text{ and } w \to \Lambda \tag{6.6}$$

We prove the 'if' part first. Let $A \underset{G}{\overset{*}{\Rightarrow}} w$ and $w \neq \Lambda$. We prove that $A \underset{G_1}{\overset{*}{\Rightarrow}} w$ by induction on the number of steps in the derivation $A \underset{G}{\overset{*}{\Rightarrow}} w$. If $A \underset{G}{\Rightarrow} w$ and

$w \neq \Lambda$, $A \to w$ is a production in P', and so $A \underset{G_1}{\Rightarrow} w$. Thus there is basis for induction. Assume the result for derivations in at most i steps. Let $A \overset{i+1}{\underset{G}{\Rightarrow}} w$ and $w \neq \Lambda$. We can split the derivation as $A \underset{G}{\Rightarrow} X_1X_2 \ldots X_k \overset{i}{\underset{G}{\Rightarrow}} w_1w_2 \ldots w_k$, where $w = w_1w_2 \ldots w_k$ and $A_j \overset{*}{\underset{G}{\Rightarrow}} w_j$. As $w \neq \Lambda$, not all w_j's are Λ. If $w_j \neq \Lambda$, then by induction hypothesis, $X_j \overset{i}{\underset{G_1}{\Rightarrow}} w_j$. If $w_j = \Lambda$, then $X_j \in W$. So using the production $A \to A_1A_2 \ldots A_k$ in P, we construct $A \to \alpha_1\alpha_2 \ldots \alpha_k$ in P', where $\alpha_j = X_j$ if $w_j \neq \Lambda$ and $\alpha_j = \Lambda$ if $w_j = \Lambda$ (i.e. $X_j \in W$). Therefore,

$$A \underset{G_1}{\Rightarrow} \alpha_1\alpha_2 \ldots \alpha_k \overset{*}{\underset{G_1}{\Rightarrow}} w_1\alpha_2 \ldots \alpha_k \overset{*}{\underset{G_1}{\Rightarrow}} \ldots \Rightarrow w_1w_2 \ldots w_k = w$$

By the principle of induction, the 'if' part of (6.6) is proved.

We prove the 'only if' part by induction on the number of steps in the derivation of $A \overset{*}{\underset{G_1}{\Rightarrow}} w$. If $A \underset{G_1}{\Rightarrow} w$, then $A \to w$ is in P_1. By construction of P', $A \to w$ is obtained from some production $A \to X_1X_2 \ldots X_n$ in P by erasing some (or none of the) nullable variables. Hence $A \underset{G}{\Rightarrow} X_1X_2 \ldots X_n \overset{*}{\underset{G}{\Rightarrow}} w$. So there is basis for induction. Assume the result for derivation in at most j steps. Let $A \overset{j+1}{\underset{G}{\Rightarrow}} w$. This can be split as $A \underset{G_1}{\Rightarrow} X_1X_2 \ldots X_k \overset{j}{\underset{G_1}{\Rightarrow}} w_1w_2 \ldots w_k$, where $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$. The first production $A \to X_1X_2 \ldots X_k$ in P' is obtained from some production $A \to \alpha$ in P by erasing some (or none of the) nullable variables in α. So $A \underset{G}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} X_1X_2 \ldots X_k$. If $X_i \in \Sigma$ then $X_i \overset{0}{\underset{G}{\Rightarrow}} X_i = w_i$. If $X_i \in V_N$ then by induction hypothesis, $X_i \overset{*}{\underset{G}{\Rightarrow}} w_i$. So, we get $A \overset{*}{\underset{G}{\Rightarrow}} X_1X_2 \ldots X_k \overset{*}{\underset{G}{\Rightarrow}} w_1w_2 \ldots w_k$. Hence by the principle of induction whenever $A \overset{*}{\underset{G_1}{\Rightarrow}} w$, we have $A \overset{*}{\underset{G}{\Rightarrow}} w$ and $w \neq \Lambda$. Thus (6.6) is completely proved.

By applying (6.6) to S, we have $w \in L(G_1)$ if and only if $w \in L(G)$ and $w \neq \Lambda$. This implies $L(G_1) = L(G) - \{\Lambda\}$. ∎

Corollary 1 There exists an algorithm to decide whether $\Lambda \in L(G)$ for a given context-free grammar G.

Proof $\Lambda \in L(G)$ if and only if $S \in W$, i.e. S is nullable. The construction given in Theorem 6.6 is recursive and terminates in a finite number of steps (actually in at most $|V_N|$ steps). So the required algorithm is as follows: (i) construct W; (ii) test whether $S \in W$.

Corollary 2 If $G = (V_N, \Sigma, P, S)$ is a context-free grammar we can find an equivalent context-free grammar $G_1 = (V'_N, \Sigma, P, S_1)$ without null productions except $S_1 \to \Lambda$ when Λ is in $L(G)$. If $S_1 \to \Lambda$ is in P_1, S_1 does not appear on the R.H.S. of any production in P_1.

Proof By Corollary 1, we can decide whether Λ is in $L(G)$.

Case 1 If Λ is not in $L(G)$, G_1 obtained by using Theorem 6.6 is the required equivalent grammar.

Case 2 If Λ is in $L(G)$, construct $G' = (V_N, \Sigma, P', S)$ using Theorem 6.6. $L(G') = L(G) - \{\Lambda\}$. Define $G_1 = (V_N \cup \{S_1\}, \Sigma, P_1, S_1)$, where $P_1 = P' \cup \{S_1 \to S, S_1 \to \Lambda\}$. S_1 does not appear on the R.H.S. of any production in P_1, and so G_1 is the required grammar with $L(G_1) = L(G)$. ∎

6.3.3 Elimination of Unit Productions

A context-free grammar may have productions of the form $A \to B$, $A, B \in V_N$.

Consider, for example, G as the grammar $S \to A$, $A \to B$, $B \to C$, $C \to a$. It is easy to see that $L(G) = \{a\}$. The productions $S \to A$, $A \to B$, $B \to C$ are useful just to replace S by C. To get a terminal string, we need $C \to a$. If G_1 is $S \to a$, then $L(G_1) = L(G)$.

The next construction eliminates productions of the form $A \to B$.

Definition 6.10 A unit production (or a chain rule) in a context-free grammar G is a production of the form $A \to B$, where A and B are variables in G.

Theorem 6.7 If G is a context-free grammar,we can find a context-free grammar G_1 which has no null productions or unit productions such that $L(G_1) = L(G)$.

Proof We can apply Corollary 2 of Theorem 6.6 to grammar G to get a grammar $G' = (V_N, \Sigma, P, S)$ without null productions such that $L(G') = L(G)$. Let A be any variable in V_N.

Step 1 *Construction of the set of variables derivable from A:*

Define $W_i(A)$ recursively as follows:

$$W_0(A) = \{A\}$$

$$W_{i+1}(A) = W_i(A) \cup \{B \in V_N \mid C \to B \text{ is in } P \text{ with } C \in W_i(A)\}$$

By definition of $W_i(A)$, $W_i(A) \subseteq W_{i+1}(A)$. As V_N is finite, $W_{k+1}(A) = W_k(A)$ for some $k \leq |V_N|$. So, $W_{k+j}(A) = W_i(A)$ for all $j \geq 0$. Let $W(A) = W_k(A)$. Then $W(A)$ is the set of all variables derivable from A.

Step 2 *Construction of A-productions* in G_1:

The A-productions in G_1 are either (i) the nonunit production in G' or (ii) $A \to \alpha$ whenever $B \to \alpha$ is in G with $B \in W(A)$ and $\alpha \notin V_N$.

(Actually, (ii) covers (i) as $A \in W(A)$). Now, we define $G_1 = (V_N, \Sigma, P_1, S)$, where P_1 is constructed using step 2 for every $A \in V_N$.

Before proving that G_1 is the required grammar, we apply the construction to an example.

EXAMPLE 6.10

Let G be $S \to AB$, $A \to a$, $B \to C \mid b$, $C \to D$, $D \to E$ and $E \to a$. Eliminate unit productions and get an equivalent grammar.

Solution

Step 1 $W_0(S) = \{S\}$, $\quad W_1(S) = W_0(S) \cup \emptyset$

Hence $W(S) = \{S\}$. Similarly,

$$W(A) = \{A\}, \quad W(E) = \{E\}$$

$$W_0(B) = \{B\}, \quad W_1(B) = \{B\} \cup \{C\} = \{B, C\}$$

$$W_2(B) = \{B, C\} \cup \{D\}, \quad W_3(B) = \{B, C, D\} \cup \{E\}, \quad W_4(B) = W_3(B)$$

Therefore,

$$W(B) = \{B, C, D, E\}$$

Similarly,

$$W_0(C) = \{C\}, \quad W_1(C) = \{C, D\}, \quad W_2\{C\} = \{C, D, E\} = W_3(C)$$

Therefore,

$$W(C) = \{C, D, E\}, \quad W_0(D) = \{D\}$$

Hence,

$$W_1(D) = \{D, E\} = W_2(D)$$

Thus,

$$W(D) = \{D, E\}$$

Step 2 The productions in G_1 are

$$S \to AB, \quad A \to a, \quad E \to a$$

$$B \to b \mid a, \quad C \to a, \quad D \to a$$

By construction, G_1 has no unit productions.

To complete the proof we have to show that $L(G') = L(G_1)$.

Step 3 $L(G') = L(G)$. If $A \to \alpha$ is in $P_1 - P$, then it is induced by $B \to \alpha$ in P with $B \in W(A)$, $\alpha \notin V_N$. $B \in W(A)$ implies $A \overset{*}{\underset{G'}{\Rightarrow}} B$. Hence, $A \overset{*}{\underset{G'}{\Rightarrow}} B \underset{G'}{\Rightarrow} \alpha$. So, if $A \underset{G_1}{\Rightarrow} \alpha$, then $A \overset{*}{\underset{G'}{\Rightarrow}} \alpha$. This proves $L(G_1) \subseteq L(G')$.

To prove the reverse inclusion, we start with a leftmost derivation

$$S \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \alpha_2 \ldots \underset{G}{\Rightarrow} \alpha_n = w$$

in G'.

Let i be the smallest index such that $\alpha_i \overset{*}{\underset{G'}{\Rightarrow}} \alpha_{i+1}$ is obtained by a unit production and j be the smallest index greater than i such that $\alpha_j \underset{G}{\Rightarrow} \alpha_{j+1}$ is obtained by a nonunit production. So, $S \underset{G_1}{\Rightarrow} \alpha_i$, and $\alpha_j \overset{*}{\underset{G'}{\Rightarrow}} \alpha_{j+1}$ can be written as

$$\alpha_i = w_i A_i \beta_i \Rightarrow w_i A_{i+1} \beta_i \Rightarrow \ldots \Rightarrow w_i A_j \beta_i \Rightarrow w_i \gamma \beta_i = \alpha_{j+1}$$

$A_j \in W(A_i)$ and $A_j \to \gamma$ is a nonunit production. Therefore, $A_j \to \gamma$ is a production in P_1. Hence, $\alpha_j \overset{*}{\underset{G_1}{\Rightarrow}} \alpha_{j+1}$. Thus, we have $S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha_{j+1}$.

Repeating the argument whenever some unit production occurs in the remaining part of the derivation, we can prove that $S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha_n = w$. This proves $L(G') \subseteq L(G)$. ∎

Corollary If G is a context-free grammar, we can construct an equivalent grammar G' which is reduced and has no null productions or unit productions.

Proof We construct G_1 in the following way:

Step 1 Eliminate null productions to get G_1 (Theorem 6.6 or Corollary 2 of this theorem).

Step 2 Eliminate unit productions in G_1 to get G_2 (Theorem 6.7).

Step 3 Construct a reduced grammar G' equivalent to G_1 (Theorem 6.5). G' is the required grammar equivalent to G.

Note: We have to apply the constructions only in the order given in the corollary of Theorem 6.7 to simplify grammars. If we change the order we may not get the grammar in the most simplified form (refer to Exercise 6.11).

6.4 NORMAL FORMS FOR CONTEXT-FREE GRAMMARS

In a context-free grammar, the R.H.S. of a production can be any string of variables and terminals. When the productions in G satisfy certain restrictions, then G is said to be in a 'normal form'. Among several 'normal forms' we study two of them in this section—the Chomsky normal form (CNF) and the Greibach normal form.

6.4.1 CHOMSKY NORMAL FORM

In the Chomsky normal form (CNF), we have restrictions on the length of R.H.S. and the nature of symbols in the R.H.S. of productions.

Definition 6.11 A context-free grammar G is in Chomsky normal form if every production is of the form $A \to a$, or $A \to BC$, and $S \to \Lambda$ is in G if

$\Lambda \in L(G)$. When Λ is in $L(G)$, we assume that S does not appear on the R.H.S. of any production.

For example, consider G whose productions are $S \to AB \mid \Lambda$, $A \to a$, $B \to b$. Then G is in Chomsky normal form.

Remark For a grammar in CNF, the derivation tree has the following property: Every node has atmost two descendants—either two internal vertices or a single leaf.

When a grammar is in CNF, some of the proofs and constructions are simpler.

Reduction to Chomsky Normal Form

Now we develop a method of constructing a grammar in CNF equivalent to a given context-free grammar. Let us first consider an example. Let G be $S \to ABC \mid aC$, $A \to a$, $B \to b$, $C \to c$. Except $S \to aC \mid ABC$, all the other productions are in the form required for CNF. The terminal a in $S \to aC$ can be replaced by a new variable D. By adding a new production $D \to a$, the effect of applying $S \to aC$ can be achieved by $S \to DC$ and $D \to a$. $S \to ABC$ is not in the required form, and hence this production can be replaced by $S \to AE$ and $E \to BC$. Thus, an equivalent grammar is $S \to AE \mid DC$, $E \to BC$, $A \to a$, $B \to b$, $C \to c$, $D \to a$.

The techniques applied in this example are used in the following theorem.

Theorem 6.8 (Reduction to Chomsky normal form). For every context-free grammar, there is an equivalent grammar G_2 in Chomsky normal form.

Proof (Construction of a grammar in CNF)

Step 1 *Elimination of null productions and unit productions:*

We apply Theorem 6.6 to eliminate null productions. We then apply Theorem 6.7 to the resulting grammar to eliminate chain productions. Let the grammar thus obtained be $G = (V_N, \Sigma, P, S)$.

Step 2 *Elimination of terminals on R.H.S.:*

We define $G_1 = (V'_N, \Sigma, P_1, S')$, where P_1 and V'_N are constructed as follows:

(i) All the productions in P of the form $A \to a$ or $A \to BC$ are included in P_1, All the variables in V_N are included in V'_N.

(ii) Consider $A \to X_1X_2 \ldots X_n$ with some terminal on R.H.S. If X_i is a terminal, say a_i, add a new variable C_{a_i} to V'_N and $C_{a_i} \to \alpha_i$ to P_1. In production $A \to X_1X_2 \ldots X_n$, every terminal on R.H.S. is replaced by the corresponding new variable and the variables on the R.H.S. are retained. The resulting production is added to P_1. Thus, we get $G_1 = (V'_N, \Sigma, P_1, S)$.

Step 3 *Restricting the number of variables on R.H.S.:*

For any production in P_1, the R.H.S. consists of either a single terminal (or

Λ in $S \to A$) or two or more variables. We define $G_2 = (V_N'', \Sigma, P_2, S)$ as follows:

(i) All productions in P_1 are added to P_2 if they are in the required form. All the variables in V_N' are added to V_N''.

(ii) Consider $A \to A_1A_2 \ldots A_m$, where $m \geq 3$. We introduce new productions $A \to A_1C_1$, $C_1 \to A_2C_2$, $\ldots$, $C_{m-2} \to A_{m-1}A_m$ and new variables $C_1, C_2, \ldots, C_{m-2}$. These are added to P'' and V_N'', respectively.

Thus, we get G_2 in Chomsky normal form.

Before proving that G_2 is the required equivalent grammar, we apply the construction to the context-free grammar given m Example 6.11.

EXAMPLE 6.11

Reduce the following grammar G to CNF. G is $S \to aAD$, $A \to aB \mid bAB$, $B \to b$, $D \to d$.

Solution

As there are no null productions or unit productions, we can proceed to step 2.

Step 2 Let $G_1 = (V_N', \{a, b, d\}, P_1, S)$, where P_1 and V_N' are constructed as follows:

(i) $B \to b$, $D \to d$ are included in P_1.

(ii) $S \to aAD$ gives rise to $S \to C_aAD$ and $C_a \to a$.
$A \to aB$ gives rise to $A \to C_aB$.
$A \to bAB$ gives rise to $A \to C_bAB$ and $C_b \to b$.
$V_N' = \{S, A, B, D, C_a, C_b\}$.

Step 3 P_1 consists of $S \to C_aAD$, $A \to C_aB \mid C_bAB$, $B \to b$, $D \to d$, $C_a \to a$, $C_b \to b$.

$A \to C_aB$, $B \to b$, $D \to d$, $C_a \to a$, $C_b \to b$ are added to P_2

$S \to C_aAD$ is replaced by $S \to C_aC_1$ and $C_1 \to AD$.

$A \to C_bAB$ is replaced by $A \to C_bC_2$ and $C_2 \to AB$.

Let

$$G_2 = (\{S, A, B, D, C_a, C_b, C_1, C_2\}, \{a, b, d\}, P_2, S)$$

where P_2 consists of $S \to C_aC_1$, $A \to C_aB \mid C_bC_2$, $C_1 \to AD$, $C_2 \to AB$, $B \to b$, $D \to d$, $C_a \to a$, $C_b \to b$. G_2 is in CNF and equivalent to G.

Step 4 $L(G) = L(G_2)$. To complete the proof we have to show that $L(G) = L(G_1) = L(G_2)$.

To show that $L(G) \subseteq L(G_1)$, we start with $w \in L(G)$. If $A \to X_1X_2 \ldots X_n$ is used in the derivation of w, the same effect can be achieved by using the corresponding production in P_1 and the productions involving the new variables. Hence, $A \overset{*}{\underset{G_1}{\Rightarrow}} X_1X_2 \ldots X_n$. Thus, $L(G) \subseteq L(G_1)$.

Let $w \in L(G_1)$. To show that $w \in L(G)$, it is enough to prove the following:

$$A \overset{*}{\underset{G}{\Rightarrow}} w \qquad \text{if } A \in V_N,\ A \overset{*}{\underset{G_1}{\Rightarrow}} w \tag{6.7}$$

We prove (6.7) by induction on the number of steps in $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

If $A \underset{G_1}{\Rightarrow} w$, then $A \to w$ is a production in P_1. By construction of P_1, w is a single terminal. So $A \to w$ is in P, i.e. $A \underset{G}{\Rightarrow} w$. Thus there is basis for induction.

Let us assume (6.7) for derivations in at most k steps. Let $A \overset{k+1}{\underset{G_1}{\Rightarrow}} w$. We can split this derivation as $A \underset{G_1}{\Rightarrow} A_1A_2 \ldots A_m \overset{k}{\underset{G_1}{\Rightarrow}} w_1 \cdot w_m = w$ such that $A_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$. Each A_i is either in V_N or a new variable, say C_{a_i}. When $A_i \in V_N$, $A_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ is a derivation in at most k steps, and so by induction hypothesis, $A_i \overset{*}{\underset{G}{\Rightarrow}} w_i$. When $A_i = C_{a_i}$, the production $C_{a_i} \to a_i$ is applied to get $A_i \overset{*}{\Rightarrow} w_i$. The production $A \to A_1A_2 \ldots A_m$ is induced by a production $A \to X_1X_2 \ldots X_m$ in P where $X_i = A_i$ if $A_i \in V_N$ and $X_i = w_i$ if $A_i = C_{a_i}$. So $A \underset{G}{\Rightarrow} X_1X_2 \ldots X_m \overset{*}{\underset{G}{\Rightarrow}} w_1w_2 \ldots w_m$, i.e. $A \overset{*}{\underset{G}{\Rightarrow}} w$. Thus, (6.7) is true for all derivations. Therefore, $L(G) = L(G_1)$.

The effect of applying $A \to A_1A_2 \ldots A_m$ in a derivation for $w \in L(G_1)$ can be achieved by applying the productions $A \to A_1C_1$, $C_1 \to A_2C_2$, ..., $C_{m-2} \to A_{m-1}A_m$ in P_2. Hence it is easy to see that $L(G_1) \subseteq L(G_2)$.

To prove $L(G_2) \subseteq L(G_1)$, we can prove an auxiliary result

$$A \overset{*}{\underset{G_1}{\Rightarrow}} w \qquad \text{if } A \in V'_N,\ A \overset{*}{\underset{G_2}{\Rightarrow}} w \tag{6.8}$$

Condition (6.8) can be proved by induction on the number of steps in $A \overset{*}{\underset{G_2}{\Rightarrow}} w$. Applying (6.7) to S, we get $L(G_2) \subseteq L(G)$. Thus,

$$L(G) = L(G_1) = L(G_2) \quad \blacksquare$$

EXAMPLE 6.12

Find a grammar in Chomsky normal form equivalent to $S \to aAbB$, $A \to aA \mid a$, $B \to bB \mid b$.

Solution

As there are no unit productions or null productions, we need not carry out step 1. We proceed to step 2.

Step 2 Let $G_1 = (V'_N, \{a, b\}, P_1, S)$, where P_1 and V'_N are constructed as follows:

(i) $A \to a$, $B \to b$ are added to P_1.

(ii) $S \to aAbB$, $A \to aA$, $B \to bB$ yield $S \to C_aAC_bB$, $A \to C_aA$, $B \to C_bB$, $C_a \to a$, $C_b \to b$.

$$V'_N = \{S, A, B, C_a, C_b\}.$$

Step 3 P_1 consists of $S \to C_aAC_bB$, $A \to C_aA$, $B \to C_bB$, $C_a \to a$, $C_b \to b$, $A \to a$, $B \to b$.

$S \to C_aAC_bB$ is replaced by $S \to C_aC_1$, $C_1 \to AC_2$, $C_2 \to C_bB$

The remaining productions in P_1 are added to P_2. Let

$$G_2 = (\{S, A, B, C_a, C_b, C_1, C_2\}, \{a, b\}, P_2, S),$$

where P_2 consists of $S \to C_aC_1$, $C_1 \to AC_2$, $C_2 \to C_bB$, $A \to C_aA$, $B \to C_bB$, $C_a \to a$, $C_b \to b$, $A \to a$, and $B \to b$.

G_2 is in CNF and equivalent to the given grammar.

EXAMPLE 6.13

Find a grammar in CNF equivalent to the grammar

$$S \to \sim S \mid [S \supset S] \mid p \mid q \qquad (S \text{ being the only variable})$$

Solution

As the given grammar has no unit or null productions, we omit step 1 and proceed to step 2.

Step 2 Let $G_1 = (V'_N, \Sigma, P_1, S)$, where P_1 and V'_N are constructed as follows:

(i) $S \to p \mid q$ are added to P_1.

(ii) $S \to \sim S$ induces $S \to AS$ and $A \to \sim$.

(iii) $S \to [S \supset S]$ induces $S \to BSCSD$, $B \to [$, $C \to \supset$, $D \to]$

$$V'_N = \{S, A, B, C, D\}$$

Step 3 P_1 consists of $S \to p \mid q$, $S \to AS$, $A \to \sim$, $B \to [$, $C \to \supset$, $D \to]$, $S \to BSCSD$.

$S \to BSCSD$ is replaced by $S \to BC_1$, $C_1 \to SC_2$, $C_2 \to CC_3$, $C_3 \to SD$. Let

$$G_2 = (\{S, A, B, C, D, C_1, C_2, C_3\}, \Sigma, P_2, S)$$

where P_2 consists of $S \to p \mid q \mid AS \mid BC_1$, $A \to \sim$, $B \to [$, $C \to \supset$, $D \to]$, $C_1 \to SC_2$, $C_2 \to CC_3$, $C_3 \to SD$. G_2 is in CNF and equivalent to the given grammar.

6.4.2 Greibach Normal Form

Greibach normal form (GNF) is another normal form quite useful in some proofs and constructions. A context-free grammar generating the set accepted by a pushdown automaton is in Greibach normal form as will be seen in Theorem 7.4.

Definition 6.12 A context-free grammar is in Greibach normal form if every production is of the form $A \to a\alpha$, where $\alpha \in V_N^*$ and $a \in \Sigma$(α may be Λ), and $S \to \Lambda$ is in G if $\Lambda \in L(G)$. When $\Lambda \in L(G)$, we assume that S does not appear on the R.H.S. of any production. For example, G given by $S \to aAB \mid \Lambda$, $A \to bC$, $B \to b$, $C \to c$ is in GNF.

Note: A grammar in GNF is a natural generalisation of a regular grammar. In a regular grammar the productions are of the form $A \to a\alpha$, where $a \in \Sigma$ and $\alpha \in V_N \cup \{\Lambda\}$, i.e. $A \to a\alpha$, with αV_N^* and $|\alpha| \leq 1$. So for a grammar in GNF or a regular grammar, we get a (single) terminal and a string of variables (possibly Λ) on application of a production (with the exception of $S \to \Lambda$).

The construction we give in this section depends mainly on the following two technical lemmas:

Lemma 6.1 Let $G = (V_N, \Sigma, P, S)$ be a CFG. Let $A \to B\gamma$ be an A-production in P. Let the B-productions be $B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_s$. Define

$$P_1 = (P - \{A \to B\gamma\}) \cup \{A \to \beta_i\gamma \mid 1 \leq i \leq s\}.$$

Then, $G_1 = (V_N, \Sigma, P_1, S)$ is a context-free grammar equivalent to G.

Proof If we apply $A \to B\gamma$ in some derivation for $w \in L(G)$, we have to apply $B \to \beta_i$ for some i at a later step. So $A \overset{*}{\underset{G}{\Rightarrow}} B_i\gamma$. The effect of applying $A \to B\gamma$ and eliminating B in grammar G is the same as applying $A \to \beta_i\gamma$ for some i in grammar G_1. Hence $w \in L(G_1)$, i.e. $L(G) \subseteq L(G_1)$. Similarly, instead of applying $A \to \beta_i\gamma$, we can apply $A \to B\gamma$ and $B \to \beta_i$ to get $A \overset{*}{\underset{G}{\Rightarrow}} \beta_i\gamma$. This proves $L(G_1) \subseteq L(G)$. ∎

Note: Lemma 6.1 is useful for deleting a variable B appearing as the first symbol on the R.H.S. of some A-production, provided no B-production has B as the first symbol on R.H.S.

The construction given in Lemma 6.1 is simple. To eliminate B in $A \to B\gamma$, we simply replace B by the right-hand side of every B-production.

For example, using Lemma 6.1, we can replace $A \to Bab$ by $A \to aAab$, $A \to bBab$, $A \to aaab$, $A \to ABab$ when the B-productions are $B \to aA \mid bB \mid aa \mid AB$.

The lemma is useful to eliminate A from the R.H.S. of $A \to A\alpha$.

Lemma 6.2 Let $G = (V_N, \Sigma, P, S)$ be a context-free grammar. Let the set of A-productions be $A \to A\alpha_1 \mid \ldots A\alpha_r \mid \beta_1 \mid \ldots \mid \beta_s$ (β_i's do not start with A).

Let Z be a new variable. Let $G_1 = (V_N \cup \{Z\}, \Sigma, P_1, S)$, where P_1 is defined as follows:

(i) The set of A-productions in P_1 are $A \to \beta_1 | \beta_2 | \ldots | \beta_s$

$$A \to \beta_1 Z | \beta_2 Z | \ldots | \beta_s Z$$

(ii) The set of Z-productions in P_1 are $Z \to \alpha_1 | \alpha_2 | \ldots | \alpha_r$

$$Z \to \alpha_1 Z | \alpha_2 Z \ldots | \alpha_r Z$$

(iii) The productions for the other variables are as in P. Then G_1 is a CFG and equivalent to G.

Proof To prove $L(G) \subseteq L(G)$, consider a leftmost derivation of w in G. The only productions in $P - P_1$ are $A \to A\alpha_1 | A\alpha_2 \ldots | A\alpha_r$. If $A \to A\alpha_{i_1}$, $A \to A\alpha_{i_2}, \ldots, A \to A\alpha_{i_k}$ are used, then $A \to \beta_j$ should be used at a later stage (to eliminate A). So we have $A \underset{G_1}{\overset{*}{\Rightarrow}} \beta_j A\alpha_{i_1} \ldots \alpha_{i_k}$ while deriving w in G. However,

$$A \underset{G_1}{\Rightarrow} \beta_j Z \underset{G_1}{\Rightarrow} \beta_j \alpha_{i_1} Z \ldots \underset{G_1}{\Rightarrow} \beta_j \alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_k}$$

i.e.

$$A \underset{G_1}{\overset{*}{\Rightarrow}} \beta_j A \alpha_{i_1} \ldots \alpha_{i_k}$$

Thus, A can be eliminated by using productions in G_1. Therefore, $w \in L(G_1)$.

To prove $L(G_1) \subseteq L(G)$, consider a leftmost derivation of w in G_1. The only productions in $P_1 - P$ are $A \to \beta_1 Z | \beta_2 Z | \ldots | \beta_s Z$, $Z \to \alpha_1 | \ldots | \alpha_r$, $Z \to \alpha_1 Z | \alpha_2 Z | \ldots | \alpha_r Z$. If the new variable Z appears in the course of the derivation of w, it is because of the application of $A \to \beta_j Z$ in some earlier step. Also, Z can be eliminated only by a production of the form $Z \to \alpha_j$ or $Z \to \alpha_j Z$ for some i and j in a later step. So we get $A \underset{G_1}{\overset{*}{\Rightarrow}} \beta_j \alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_k}$ in the course of the derivation of w. But, we know that $A \underset{G}{\overset{*}{\Rightarrow}} \beta_j \alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_k}$. Therefore, $w \in L(G)$. ∎

EXAMPLE 6.14

Apply Lemma 6.2 to the following A-productions in a context-free grammar G.

$$A \to aBD | bDB | c, \qquad A \to AB | AD$$

Solution

In this example, $\alpha_1 = B$, $\alpha_2 = D$, $\beta_1 = aBD$, $\beta_2 = bDB$, $\beta_3 = c$. So the new productions are:

(i) $A \to aBD | bDB | c$, $\quad A \to aBDZ | bDBZ | cZ$

(ii) $Z \to B$, $Z \to D$, $\quad Z \to BZ | DZ$

Theorem 6.9 (Reduction to Greibach normal form). Every context-free language L can be generated by a context-free grammar G in Greibach normal form.

Proof We prove the theorem when $\Lambda \notin L$ and then extend the construction to L having Λ.

Case 1 *Construction of G (when* $\Lambda \in L$*):*

Step 1 We eliminate null productions and then construct a grammar G in Chomsky normal form generating L. We rename the variables as $A_1, A_2, \ldots, A_n$ with $S = A_1$. We write G as $(\{A_1, A_2, \ldots, A_n\}, \Sigma, P, A_1)$.

Step 2 To get the productions in the form $A_i \to a\gamma$ or $A_i \to A_j\gamma$, where $j > i$, convert the A_i-productions ($i = 1, 2, \ldots, n-1$) to the form $A_i \to A_i\gamma$ such that $j > i$. Prove that such modification is possible by induction on i.

Consider A_1-productions. If we have some A_1-productions of the form $A_1 \to A_1\gamma$, then we can apply Lemma 6.2 to get rid of such productions. We get a new variable, say Z_1, and A_1-productions of the form $A_1 \to a$ or $A_1 \to A_j\gamma'$, where $j > 1$. Thus there is basis for induction.

Assume that we have modified A_1-productions, A_2-productions ... A_i-productions. Consider A_{i+1}-productions. Productions of the form $A_{i+1} \to a\gamma$ required no modification. Consider the first symbol (this will be a variable) on the R.H.S. of the remaining A_{i+1}-productions. Let t be the smallest index among the indices of such symbols (variables). If $t > i + 1$, there is nothing to prove. Otherwise, apply the induction hypothesis to A_t-productions for $t \leq i$. So any A_t-production is of the form $A_t \to A_j\gamma$, where $j > t$ or $A_t \to a\gamma'$. Now we can apply Lemma 6.1 to A_{i+1}-production whose R.H.S. starts with A_t. The resulting A_{i+1}-productions are of the form $A_{i+1} \to A_j\gamma$, where $j > t$ (or $A_{i+1} \to a\gamma'$).

We repeat the above construction by finding t for the new set of A_{i+1}-productions. Ultimately, the A_{i+1}-productions are converted to the form $A_{i+1} \to A_j\gamma$; where $j \geq i + 1$ or $A_{i+1} \to a\gamma'$. Productions of the form $A_{i+1} \to A_{i+1}\gamma$ can be modified by using Lemma 6.2. Thus we have converted A_{i+1}-productions to the required form. By the principle of induction, the construction can be carried out for $i = 1, 2, \ldots, n$. Thus for $i = 1, 2, \ldots, n-1$, any A_i-production is of form $A_i \to A_j\gamma$, where $j > i$ or $A_i \to a\gamma'$. Any A_n-production is of the form $A_n \to A_n\gamma$ or $A_n \to a\gamma'$.

Step 3 Convert A_n-productions to the form $A_n \to a\gamma$. Here, the productions of the form $A_n \to A_n\gamma$ are eliminated using Lemma 6.2. The resulting A_n-productions are of the form $A_n \to a\gamma$.

Step 4 Modify the A_i-productions to the form $A_i \to a\gamma$ for $i = 1, 2, \ldots, n-1$. At the end of step 3, the A_n-productions are of the form $A_n \to a\gamma$. The A_{n-1}-productions are of the form $A_{n-1} \to a\gamma'$ or $A_{n-1} \to A_n\gamma$. By applying Lemma 6.1, we eliminate productions of the form $A_{n-1} \to A_n\gamma$. The resulting

A_{n-1}-productions are in the required form. We repeat the construction by considering A_{n-2}, A_{n-3}, . . ., A_1.

Step 5 Modify Z_i-productions. Every time we apply Lemma 6.2, we get a new variable. (We take it as Z_i when we apply the Lemma for A_i-productions.) The Z_i-productions are of the form $Z_i \to \alpha Z_i$ or $Z_i \to \alpha$ (where α is obtained from $A_i \to A_i\alpha$), and hence of the form $Z_i \to a\gamma$ or $Z_i \to A_k\gamma$ for some k. At the end of step 4, the R.H.S. of any A_k-production starts with a terminal. So we can apply Lemma 6.1 to eliminate $Z_i \to A_k\gamma$. Thus at the end of step 5, we get an equivalent grammar G_1 in GNF.

It is easy to see that G_1 is in GNF. We start with G in CNG. In G any A-production is of the form $A \to a$ or $A \to AB$ or $A \to CD$. When we apply Lemma 6.1 or Lemma 6.2 in step 2, we get new productions of the form $A \to a\alpha$ or $A \to \beta$, where $\alpha \in V_N^*$ and $\beta \in V_N^+$ and $a \in \Sigma$. In steps 3–5, the productions are modified to the form $A \to a\alpha$ or $Z \to a'\alpha'$, where a, $a' \in \Sigma$ and α, $\alpha' \in V_N^*$.

Case 2 *Construction of G when* $\Lambda \in L$:

By the previous construction we get $G' = (V_N', \Sigma, P_1, S)$ in GNF such that $L(G') = L - \{\Lambda\}$. Define a new grammar G_1 as

$$G_1 = (V_N' \cup \{S'\}, \Sigma, P_1 \cup \{S' \to S, S' \to \Lambda\}, S')$$

$S' \to S$ can be eliminated by using Theorem 6.7. As S-productions are in the required form, S'-productions are also in the required form. So $L(G) = L(G_1)$ and G_1 is in GNF. ∎

Remark Although we convert the given grammar to CNF in the first step, it is not necessary to convert all the productions to the form required for CNF. In steps 2–5, we do not disturb the productions of the form $A \to a\alpha$, $a \in \Sigma$ and $\alpha \in V_N^*$. So such productions can be allowed in G (in step 1). If we apply Lemma 6.1 or 6.2 as in steps 2–5 to productions of the form $A \to \alpha$, where $\alpha \in V_N^*$ and $|\alpha| \geq 2$, the resulting productions at the end of step 5 are in the required form (for GNF). Hence we can allow productions of the form $A \to \alpha$, where $\alpha \in V_N^*$, and $|\alpha| \geq 2$.

Thus we can apply steps 2–5 to a grammar whose productions are either $A \to a\alpha$ where $a \in V_N^*$, or $A \to \alpha \in V_N^*$ where $|\alpha| \geq 2$. To reduce the productions to the form $A \to \alpha \in V_N^*$ where $|\alpha| \geq 2$, we can apply step 2 of Theorem 6.8.

EXAMPLE 6.15

Construct a grammar in Greibach normal form equivalent to the grammar $S \to AA \mid a$, $A \to SS \mid b$.

Solution

The given grammar is in CNF. S and A are renamed as A_1 and A_2, respectively. So the productions are $A_1 \to A_1A_2 \mid a$ and $A_2 \to A_1A_1 \mid b$. As the

given grammar has no null productions and is in CNF we need not carry out step 1. So we proceed to step 2.

Step 2 (i) A_1-productions are in the required form. They are $A_1 \to A_2A_2 \mid a$.

(ii) $A_2 \to b$ is in the required form. Apply Lemma 6.1 to $A_2 \to A_1A_1$. The resulting productions are $A_2 \to A_2A_2A_1$, $A_2 \to aA_1$. Thus the A_2-productions are

$$A_2 \to A_2A_2A_1, \quad A_2 \to aA_1, \quad A_2 \to b$$

Step 3 We have to apply Lemma 6.2 to A_2-productions as we have $A_2 \to A_2A_2A_1$. Let Z_2 be the new variable. The resulting productions are

$$A_2 \to aA_1, \quad A_2 \to b$$
$$A_2 \to aA_1Z_2, \quad A_2 \to bZ_2$$
$$Z_2 \to A_2A_1, \quad Z_2 \to A_2A_1Z_2.$$

Step 4 (i) The A_2-productions are $A_2 \to aA_1 \mid b \mid aA_1Z_2 \mid bZ_2$.

(ii) Among the A_1-productions we retain $A_1 \to a$ and eliminate $A_1 \to A_2A_2$ using Lemma 6.1. The resulting productions are $A_1 \to aA_1A_2 \mid bA_2$. $A_1 \to aA_1Z_2A_2 \mid bZ_2A_2$. The set of all (modified) A_1-productions is

$$A_1 \to a \mid aA_1A_2 \mid bA_2 \mid aA_1Z_2A_2 \mid bZ_2A_2$$

Step 5 The Z_2-productions to be modified are $Z_2 \to A_2A_1$, $Z_2 \to A_2A_1Z_2$. We apply Lemma 6.1 and get

$$Z_2 \to aA_1A_1 \mid bA_1 \mid aA_1Z_2A_1 \mid bZ_2A_1$$
$$Z_2 \to aA_1A_1Z_2 \mid bA_1Z_2 \mid aA_1Z_2A_1Z_2 \mid bZ_2A_1Z_2$$

Hence the equivalent grammar is

$$G' = (\{A_1, A_2, Z_2\}, \{a, b\}, P_1, A_1)$$

where P_1 consists of

$$A_1 \to a \mid aA_1A_2 \mid bA_2 \mid aA_1Z_2A_1 \mid bZ_2A_2$$
$$A_2 \to aA_1 \mid b \mid aA_1Z_2 \mid bZ_2$$
$$Z_2 \to aA_1A_1 \mid bA_1 \mid aA_1Z_2A_1 \mid bZ_2A_1$$
$$Z_2 \to aA_1A_1Z_2 \mid bA_1Z_2 \mid aA_1Z_2A_1Z_2 \mid bZ_2A_1Z_2$$

EXAMPLE 6.16

Convert the grammar $S \to AB$, $A \to BS \mid b$, $B \to SA \mid a$ into GNF.

Solution

As the given grammar is in CNF, we can omit step 1 and proceed to step 2 after renaming S, A, B as A_1, A_2, A_3, respectively. The productions are $A_1 \to A_2A_3$, $A_2 \to A_3A_1 \mid b$, $A_3 \to A_1A_2 \mid a$.

Step 2 (i) The A_1-production $A_1 \rightarrow A_2A_3$ is in the required form.
(ii) The A_2-productions $A_2 \rightarrow A_3A_1 \mid b$ are in the required form.
(iii) $A_3 \rightarrow a$ is in the required form.

Apply Lemma 6.1 to $A_3 \rightarrow A_1A_2$. The resulting productions are $A_3 \rightarrow A_2A_3A_2$. Applying the lemma once again to $A_3 \rightarrow A_2A_3A_2$, we get

$$A_3 \rightarrow A_3A_1A_3A_2 \mid bA_3A_2.$$

Step 3 The A_3-productions are $A_3 \rightarrow a \mid bA_3A_2$ and $A_3 \rightarrow A_3A_1A_3A_2$. As we have $A_3 \rightarrow A_3A_1A_3A_2$, we have to apply Lemma 6.2 to A_3-productions. Let Z_3 be the new variable. The resulting productions are

$$A_3 \rightarrow a \mid bA_3A_2, \qquad A_3 \rightarrow aZ_3 \mid bA_3A_2Z_3$$

$$Z_3 \rightarrow A_1A_3A_2, \qquad Z_3 \rightarrow A_1A_3A_2Z_3$$

Step 4 (i) The A_3-productions are

$$A_3 \rightarrow a \mid bA_3A_2 \mid aZ_3 \mid bA_3A_2Z_3 \tag{6.9}$$

(ii) Among the A_2-productions, we retain $A_2 \rightarrow b$ and eliminate $A_2 \rightarrow A_3A_1$ using Lemma 6.1. The resulting productions are

$$A_2 \rightarrow aA_1 \mid bA_3A_2A_1 \mid aZ_3A \mid bA_3A_2Z_3A_1$$

The modified A_2-productions are

$$A_2 \rightarrow b \mid aA_1 \mid bA_3A_2A_1 \mid aZ_3A_1 \mid bA_3A_2Z_3A_1 \tag{6.10}$$

(iii) We apply Lemma 6.1 to $A_1 \rightarrow A_2A_3$ to get

$$A_1 \rightarrow bA_3 \mid aA_1A_3 \mid bA_3A_2A_1A_3 \mid aZ_3A_1A_3 \mid bA_3A_2Z_3A_1A_3 \tag{6.11}$$

Step 5 The Z_3-productions to be modified are

$$Z_3 \rightarrow A_1A_3A_2 \mid A_1A_3A_2Z_3$$

We apply Lemma 6.1 and get

$$\begin{aligned}
Z_3 &\rightarrow bA_3A_3A_2 \mid bA_3A_2Z_3 \\
Z_3 &\rightarrow aA_1A_3A_3A_2 \mid aA_1A_3A_3A_2Z_3 \\
Z_3 &\rightarrow bA_3A_2A_1A_3A_3A_2 \mid bA_3A_2A_1A_3A_3A_2Z_3 \\
Z_3 &\rightarrow aZ_3A_1A_3A_3A_2 \mid aZ_3A_1A_3A_3A_2Z_3 \\
Z_3 &\rightarrow bA_3A_2Z_3A_1A_3A_3A_2 \mid bA_3A_2Z_3A_1A_3A_3A_2Z_3
\end{aligned} \tag{6.12}$$

The required grammar in GNF is given by (6.9)–(6.12).

The following example uses the Remark appearing after Theorem 6.9. In this example we retain productions of the form $A \rightarrow a\alpha$ and replace the terminals only when they appear as the second or subsequent symbol on R.H.S. (Example 6.17 gives productions to generate arithmetic expressions involving a and operations like +, $*$ and parentheses.)

EXAMPLE 6.17

Find a grammar in GNF equivalent to the grammar

$$E \to E + T \mid T,\ T \to T * F \mid F,\ F \to (E) \mid a$$

Solution

Step 1 We first eliminate unit productions. Hence

$$W_0(E) = \{E\}, \qquad W_1(E) = \{E\} \cup \{T\} = \{E, T\}$$

$$W_2(E) = \{E, T\} \cup \{F\} = \{E, T, F\}$$

So,

$$W(E) = \{E, T, F\}$$

So,

$$W_0(T) = \{T\}, \qquad W_1(T) = \{T\} \cup \{F\} = \{T, F\}$$

Thus,

$$W(T) = \{T, F\}$$

$$W_0(F) = \{F\}, \qquad W_1(F) = \{F\} = W(F)$$

The equivalent grammar without unit productions is, therefore, $G_1 = (V_N, \Sigma, P_1, S)$, where P_1 consists of

(i) $E \to E + T \mid T * F \mid (E) \mid a$
(ii) $T \to T * F \mid (E) \mid a$, and
(iii) $F \to (E) \mid a$.

We apply step 2 of reduction to CNF. We introduce new variables A, B, C corresponding to +, *,). The modified productions are

(i) $E \to EAT \mid TBF \mid (EC \mid a$
(ii) $T \to TBF \mid (EC \mid a$
(iii) $F \to (EC \mid a$
(iv) $A \to +,\ B \to *,\ C \to)$

The variables A, B, C, F, T and E are renamed as A_1, A_2, A_3, A_4, A_5, A_6. Then the productions become

$$A_1 \to +, \qquad A_2 \to *, \qquad A_3 \to), \qquad A_4 \to (A_6A_3 \mid a \tag{6.13}$$

$$A_5 \to A_5A_2A_4 \mid (A_6A_3 \mid a$$

$$A_6 \to A_6A_1A_5 \mid A_5A_2A_4 \mid (A_6A_3 \mid a$$

Step 2 We have to modify only the A_5- and A_6-productions. $A_5 \to A_5A_2A_4$ can be modified by using Lemma 6.2. The resulting productions are

$$A_5 \to (A_6A_3 \mid a, \qquad A_5 \to (A_6A_3Z_5 \mid aZ_5 \tag{6.14}$$

$$Z_5 \to A_2A_4 \mid A_2A_4Z_5$$

$A_6 \to A_5A_2A_4$ can be modified by using Lemma 6.1. The resulting productions are

$$A_6 \to (A_6A_3A_2A_4 \mid aA_2A_4 \mid (A_6A_3Z_5A_2A_4 \mid aZ_5A_2A_4$$

$A_6 \to (A_6A_3 \mid a$ are in the proper form.

Step 3 $A_6 \rightarrow A_6A_1A_5$ can be modified by using Lemma 6.2. The resulting productions give all the A_6-productions:

$$A_6 \rightarrow (A_6A_3A_2A_4 \mid aA_2A_4 \mid (A_6A_3Z_5A_2A_4$$

$$A_6 \rightarrow aZ_5A_2A_4 \mid (A_6A_3 \mid a \tag{6.15}$$

$$A_6 \rightarrow (A_6A_3A_2A_4Z_6 \mid aA_2A_4Z_6 \mid (A_6A_3Z_5A_2A_4Z_6$$

$$A_6 \rightarrow aZ_5A_2A_4Z_6 \mid (A_6A_3Z_6 \mid aZ_6 \tag{6.16}$$

$$A_6 \rightarrow A_1A_5 \mid A_1A_5Z_6$$

Step 4 The step is not necessary as A_i–productions for $i = 5, 4, 3, 2, 1$ are in the required form.

Step 5 The Z_5-productions are $Z_5 \rightarrow A_2A_4 \mid A_2A_4Z_5$. These can be modified as

$$Z_5 \rightarrow *A_4 \mid *A_4Z_5 \tag{6.17}$$

The Z_6-productions are $Z_6 \rightarrow A_1A_5 \mid A_1A_5Z_6$. These can be modified as

$$Z_6 \rightarrow +A_5 \mid +A_5Z_6 \tag{6.18}$$

The required grammar in GNF is given by (6.13)–(6.18).

6.5 PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

The pumping lemma for context-free languages gives a method of generating an infinite number of strings from a given sufficiently long string in a context-free language L. It is used to prove that certain languages are not context-free. The construction we make use of in proving pumping lemma yields some decesion algorithms regarding context-free languages.

Lemma 6.3 Let G be a context-free grammar in CNF and T be a derivation tree in G. If the length of the longest path in T is less than or equal to k, then the yield of T is of length less than or equal to 2^{k-1}.

Proof We prove the result by induction on k, the length of the longest path for all A-trees (Recall an A-tree is a derivation tree whose root has label A).

When the longest path in an A-tree is of length 1, the root has only one son whose label is a terminal (when the root has two sons, the labels are variables). So the yield is of length 1. Thus, there is basis for induction.

Assume the result for $k - 1$ ($k > 1$). Let T be an A-tree with a longest path of length less than or equal to k. As $k > 1$, the root of T has exactly two sons with labels A_1 and A_2. The two subtrees with the two sons as roots have the longest paths of length less than or equal to $k - 1$ (see Fig. 6.12).

If w_1 and w_2 are their yields, then by induction hypothesis, $|w_1| \leq 2^{k-2}$, $|w_2| \leq 2^{k-2}$. So the yield of $T = w_1w_2$, $|w_1w_2| \leq 2^{k-2} + 2^{k-2} = 2^{k-1}$. By the principle of induction, the result is true for all A-trees, and hence for all derivation trees.

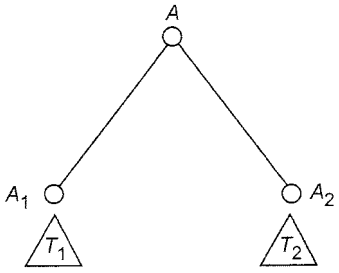


Fig. 6.12 Tree T with subtrees T_1 and T_2.

Theorem 6.10 (Pumping lemma for context-free languages). Let L be a context-free language. Then we can find a natural number n such that:

(i) Every $z \in L$ with $|z| \geq n$ can be written as $uvwxy$ for some strings u, v, w, x, y.
(ii) $|vx| \geq 1$.
(iii) $|vwx| \leq n$.
(iv) $uv^kwx^ky \in L$ for all $k \geq 0$.

Proof By Corollary 1 of Theorem 6.6, we can decide whether or not $\Lambda \in L$. When $\Lambda \in L$, we consider $L - \{\Lambda\}$ and construct a grammar $G = (V_N, \Sigma, P, S)$ in CNF generating $L - \{\Lambda\}$ (when $\Lambda \notin L$, we construct G in CNF generating L).

Let $|V_N| = m$ and $n = 2^m$. To prove that n is the required number, we start
with $z \in L$, $|z| \geq 2^m$, and construct a derivation tree T (parse tree) of z. If the length of a longest path in T is at most m, by Lemma 6.3, $|z| \leq 2^{m-1}$ (since z is the yield of T). But $|z| \geq 2^m > 2^{m-1}$. So T has a path, say Γ, of length greater than or equal to $m + 1$. Γ has at least $m + 2$ vertices and only the last vertex is a leaf. Thus in Γ all the labels except the last one are variables. As $|V_N| = m$, some label is repeated.

We choose a repeated label as follows: We start with the leaf of Γ and travel along Γ upwards. We stop when some label, say B, is repeated. (Among several repeated labels, B is the first.) Let v_1 and v_2 be the vertices with label B, v_1 being nearer the root. In Γ, the portion of the path from v_1 to the leaf has only one label, namely B, which is repeated, and so its length is at most $m + 1$.

Let T_1 and T_2 be the subtrees with v_1, v_2 as roots and z_1, w as yields, respectively. As Γ is a longest path in T, the portion of Γ from v_1 to the leaf is a longest path in T_1 and of length at most $m + 1$. By Lemma 6.3, $|z_1| \leq 2^m$ (since z_1 is the yield of T_1).

For better understanding, we illustrate the construction for the grammar whose productions are $S \to AB$, $A \to aB \mid a$, $B \to bA \mid b$, as in Fig. 6.13. In the figure,

$$\Gamma = S \to A \to B \to A \to B \to b$$
$$z = ababb, \quad z_1 = bab, \quad w = b$$
$$v = ba, \quad x = \Lambda, \quad u = a, \quad y = b$$

As z and z_1 are the yields of T and a proper subtree T_1 of T, we can write $z = uz_1y$. As z_1 and w are the yields of T_1 and a proper subtree T_2 of T_1, we can write $z_1 = vwx$. Also, $|vwx| > |w|$. So, $|vx| \geq 1$. Thus, we have $z = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$. This proves the points (i)–(iii) of the theorem.

As T is an S-tree and T_1, T_2 are B-trees, we get $S \overset{*}{\Rightarrow} uBy$, $B \overset{*}{\Rightarrow} vBx$ and $B \overset{*}{\Rightarrow} w$. As $S \overset{*}{\Rightarrow} uBy \Rightarrow uwy$, $uv^0wx^0y \in L$. For $k \geq 1$, $S \overset{*}{\Rightarrow} uBy \overset{*}{\Rightarrow} uv^kBx^ky \overset{*}{\Rightarrow} uv^kwx^ky \in L$. This proves the point (iv) of the theorem. ∎

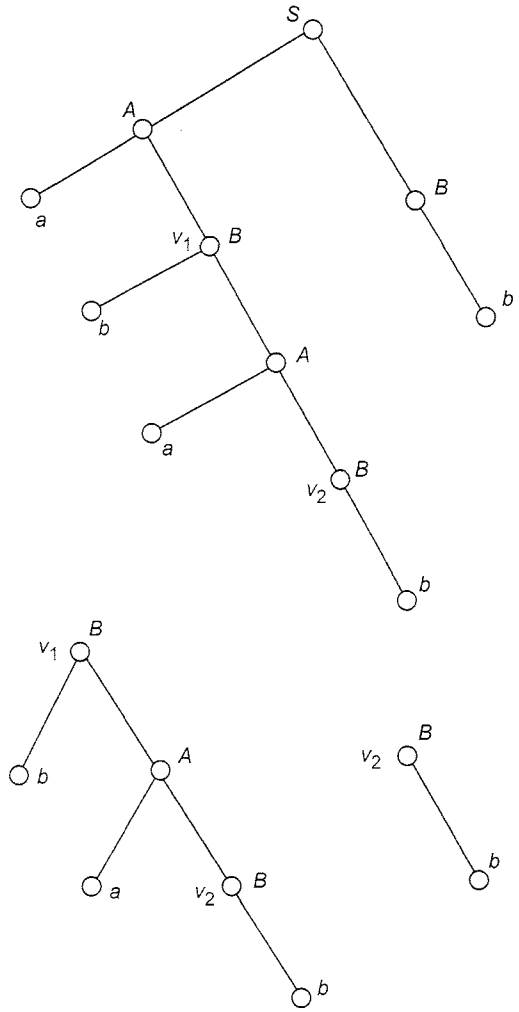


Fig. 6.13 Tree T and its subtrees T_1 and T_2.

Corollary Let L be a context-free language and n be the natural number obtained by using the pumping lemma. Then (i) $L \neq \emptyset$ if and only if there exists $w \in L$ with $|w| < n$, and (ii) L is infinite if and only if there exists $z \in L$ such that $n \leq |z| < 2n$.

Proof (i) We have to prove the 'only if' part. If $z \in L$ with $|z| \geq n$, we apply the pumping lemma to write $z = uvwxy$, where $1 \leq |vx| \leq n$. Also, $uwy \in L$ and $|uwy| < |z|$. Applying the pumping lemma repeatedly, we can get $z' \in L$ such that $|z'| < n$. Thus (i) is proved.

(ii) If $z \in L$ such that $n \leq |z| < 2n$, by pumping lemma we can write $z = uvwxy$. Also, $uv^k wx^k y \in L$ for all $k \geq 0$. Thus we get an infinite number of elements in L. Conversely, if L is infinite, we can find $z \in L$ with $|z| \geq n$. If $|z| < 2n$, there is nothing to prove. Otherwise, we can apply the pumping lemma to write $z = uvwxy$ and get $uwy \in L$. Every time we apply the pumping lemma we get a smaller string and the decrease in length is at most n (being equal to $|vx|$). So, we ultimately get a string z' in L such that $n \leq |z'| < 2n$. This proves (ii). ∎

Note: As the proof of the corollary depends only on the length of vx, we can apply the corollary to regular sets as well (refer to pumping lemma for regular sets).

The corollary given above provides us algorithms to test whether a given context-free language is empty or infinite. But these algorithms are not efficient. We shall give some other algorithms in Section 6.6.

We use the pumping lemma to show that a language L is not a context-free language. We assume that L is context-free. By applying the pumping lemma we get a contradiction.

The procedure can be carried out by using the following steps:

Step 1 Assume L is context-free. Let n be the natural number obtained by using the pumping lemma.

Step 2 Choose $z \in L$ so that $|z| \geq n$. Write $z = uvwxy$ using the pumping lemma.

Step 3 Find a suitable k so that $uv^k wx^k y \notin L$. This is a contradiction, and so L is not context-free.

EXAMPLE 6.18

Show that $L = \{a^n b^n c^n \mid n \geq 1\}$ is not context-free but context-sensitive.

Solution

We have already constructed a context-sensitive grammar G generating L (see Example 4.11). We note that in every string of L, any symbol appears the same number of times as any other symbol. Also a cannot appear after b, and c cannot appear before b, and so on.

Step 1 Assume L is context-free. Let n be the natural number obtained by using the pumping lemma.

Step 2 Let $z = a^n b^n c^n$. Then $|z| = 3n > n$. Write $z = uvwxy$, where $|vx| \geq 1$, i.e. at least one of v or x is not Λ.

Step 3 $uvwxy = a^n b^n c^n$. As $1 \leq |vx| \leq n$, v or x cannot contain all the three symbols a, b, c. So, (i) v or x is of the form $a^i b^j$ (or $b^i c^j$) for some i, j such that $i + j \leq n$. Or (ii) v or x is a string formed by the repetition of only one symbol among a, b, c.

When v or x is of the form $a^i b^j$, $v^2 = a^i b^j a^i b^j$ (or $x^2 = a^i b^j a^i b^j$). As v^2 is a substring of uv^2wx^2y, we cannot have uv^2wx^2y of the form $a^m b^m c^m$. So, $uv^2wx^2y \notin L$.

When both v and x are formed by the repetition of a single symbol (e.g. $u = a^i$ and $v = b^j$ for some i and j, $i \leq n, j \leq n$), the string uwy will contain the remaining symbol, say a_1. Also, a_1^n will be a substring of uwy as a_1 does not occur in v or x. The number of occurrences of one of the other two symbols in uwy is less than n (recall $uvwxy = a^n b^n c^n$), and n is the number of occurrences of a_1. So $uv^0wx^0y = uwy \notin L$.

Thus for any choice of v or x, we get a contradiction. Therefore, L is not context-free.

EXAMPLE 6.19

Show that $L = \{a^p \mid p \text{ is a prime}\}$ is not a context-free language.

Solution

We use the following property of L: If $w \in L$, then $|w|$ is a prime.

Step 1 Suppose $L = L(G)$ is context-free. Let n be the natural number obtained by using the pumping lemma.

Step 2 Let p be a prime number greater than n. Then $z = a^p \in L$. We write $z = uvwxy$.

Step 3 By pumping lemma, $uv^0wx^0y = uwy \in L$. So $|uwy|$ is a prime number, say q. Let $|vx| = r$. Then, $|uv^q wx^q y| = q + qr$. As $q + qr$ is not a prime, $uv^q wx^q y \notin L$. This is a contradiction. Therefore, L is not context-free.

6.6 DECISION ALGORITHMS FOR CONTEXT-FREE LANGUAGES

In this section we give some decision algorithms for context-free languages and regular sets.

(i) *Algorithm for deciding whether a context-free language L is empty.* We can apply the construction given in Theorem 6.3 for getting $V'_N = W_k$. L is nonempty if and only if $S \in W_k$.

(ii) *Algorithm for deciding whether a context-free language L is finite.* Construct a non-redundant context-free grammar G in CNF generating $L - \{\Lambda\}$. We draw a directed graph whose vertices are variables in G. If $A \to BC$ is a production, there are directed edges from A to B and A to C. L is finite if and only if the directed graph has no cycles.

(iii) *Algorithm for deciding whether a regular language L is empty.* Construct a deterministic finite automaton M accepting L. We construct the set of all states reachable from the initial state q_0. We find the states which are reachable from q_0 by applying a single input symbol. These states are arranged as a row under columns corresponding to every input symbol. The construction is repeated for every state appearing in an earlier row. The construction terminates in a finite number of steps. If a final state appears in this tabular column, then L is nonempty. (Actually, we can terminate the construction as soon as some final state is obtained in the tabular column.) Otherwise, L is empty.

(iv) *Algorithm for deciding whether a regular language L is infinite.* Construct a deterministic finite automaton M accepting L. L is infinite if and only if M has a cycle.

6.7 SUPPLEMENTARY EXAMPLES

EXAMPLE 6.20

Consider a context-free grammar G with the following productions,

$$S \rightarrow ASA \mid B$$
$$B \rightarrow aCb \mid bCa$$
$$C' \rightarrow ACA \mid A$$
$$A \rightarrow a \mid b$$

and answer the following questions:

(a) What are the variables and terminals of G?
(b) Give three strings of length 7 in $L(G)$.
(c) Are the following strings in $L(G)$?
  (i) *aaa* (ii) *bbb* (iii) *aba* (iv) *abb*
(d) True or false: $C \Rightarrow bab$
(e) True or false: $C \overset{*}{\Rightarrow} bab$
(f) True or false: $C \overset{*}{\Rightarrow} abab$
(g) True or false: $C \overset{*}{\Rightarrow} AAA$
(h) Is Λ in $L(G)$?

Solution

(a) $V_N = \{S, A, B, C\}$ and $\Sigma = \{a, b\}$
(b) $S \overset{*}{\Rightarrow} A^2SA^2 \Rightarrow A^2BA^2 \Rightarrow A^2aCbA^2 \Rightarrow A^2aAbA^2 \overset{*}{\Rightarrow} ababbab$

So $ababbab \in L(G)$.

$S \overset{*}{\Rightarrow} A^2aAbA^2$ (as in the derivation of the first string)

$\overset{*}{\Rightarrow} aaaabaa$

$S \overset{*}{\Rightarrow} A^2aAbA^2 \overset{*}{\Rightarrow} bbabbbb$

So $ababbab$, $aaaabaa$, $bbabbbb$ are in $L(G)$.

(c) $S \to ASA$. If $B \overset{*}{\Rightarrow} w$, then w starts with a and ends in b or vice versa and $|w| \geq 3$. If aaa is in $L(G)$, then the first two steps in the derivation of aaa should be $S \Rightarrow ASA \Rightarrow ABA$ or $S \overset{*}{\Rightarrow} aBa$. The length of the terminal string thus derived is of length 5 or more. Hence $aaa \notin L(G)$. A similar argument shows that $bbb \notin L(G)$.

$S \Rightarrow B \Rightarrow acb \Rightarrow aAb \Rightarrow abb$. So $abb \in L(G)$

(d) False, since the single-step derivations starting with C can only be $C \Rightarrow ACA$ or $C \Rightarrow A$.

(e) $C \Rightarrow ACA \Rightarrow AAA \overset{*}{\Rightarrow} bab$. True

(f) Let $w = abab$. If $C \overset{*}{\Rightarrow} w$, then $C \Rightarrow ACA \overset{*}{\Rightarrow} w$ or $C \Rightarrow A \Rightarrow w$. In the first case $|w| = 3, 5, 7, \ldots$. As $|w| = 4$, and $A \Rightarrow w$ if and only if $w = a$ or b, the second case does not arise. Hence (f) is false.

(g) $C \Rightarrow ACA \Rightarrow AAA$. Hence $C \overset{*}{\Rightarrow} AAA$ is true.

(h) $\Lambda \notin L(G)$.

EXAMPLE 6.21

If G consists of the productions $S \to aSa \mid bSb \mid aSb \mid bSa \mid \Lambda$, show that $L(G)$ is a regular set.

Solution

First of all, we show that $L(G)$ consists of the set L of all strings over $\{a, b\}$, of even length. It is easy to see that $L(G) \subseteq L$. Consider a string w of even length. Then, $w = a_1a_2 \ldots a_{2n-1}a_{2n}$ where each a_i is either a or b. Hence

$S \Rightarrow a_1Sa_{2n} \Rightarrow a_1a_2Sa_{2n-1}a_{2n} \Rightarrow a_1a_2 \ldots a_nSa_{n+1} \ldots a_{2n} \Rightarrow a_1a_2 \ldots a_{2n}$. Hence $L \subseteq L(G)$.

Next we prove that $L = L(G_1)$ for some regular grammar G_1. Define $G_1 = (\{S, S_1, S_2, S_3, S_4\}, \{a, b\}, P, S)$ where P consists of $S \to aS_1$, $S_1 \to aS$, $S \to aS_2$, $S_2 \to bS$, $S \to bS_3$, $S_3 \to bS$, $S \to bS_4$, $S_4 \to aS$, $S \to \Lambda$.

Then $S \overset{*}{\Rightarrow} a_1a_2S$ where $a_1 = a$ or b and $a_2 = a$ or b. It is easy to see that $L(G_1) = L$. As G_1 is regular, $L(G) = L(G_1)$ is a regular set.

EXAMPLE 6.22

Reduce the following grammar to CNF:

$$S \to ASA \mid bA, \qquad A \to B \mid S, \qquad B \to c$$

Solution

Step 1 *Elimination of unit productions:*
The unit productions are $A \to B$, $A \to S$.

$$W_0(S) = \{S\},\ W_1(S) = \{S\} \cup \emptyset = \{S\}$$

$$W_0(A) = \{A\},\ W_1(A) = \{A\} \cup \{S, B\} = \{S, A, B\}$$

$$W_2(A) = \{S, A, B\} \cup \emptyset = \{S, A, B\}$$

$$W_0(B) = \{B\},\ W_1(B) = \{B\} \cup \emptyset = \{B\}$$

The productions for the equivalent grammar without unit productions are

$$S \to ASA \mid bA,\ B \to c$$

$$A \to ASA \mid bA,\ A \to c$$

So, $G_1 = (\{S, A, B\}, \{b, c\}, P, S)$ where P consists of $S \to ASA \mid bA$, $B \to c$, $A \to ASA \mid bA \mid c$.

Step 2 *Elimination of terminals in R.H.S.:*

$S \to ASA$, $B \to c$, $A \to ASA \mid c$ are in proper form. We have to modify $S \to bA$ and $A \to bA$.

Replace $S \to bA$ by $S \to C_bA$, $C_b \to b$ and $A \to bA$ by $A \to C_bA$, $C_b \to b$.

So, $G_2 = (\{S, A, B, C_b\}, \{b, c\}, P_2, S)$ where P_2 consists of

$$S \to ASA \mid C_bA$$

$$A \to ASA \mid c \mid C_bA$$

$$B \to c,\ C_b \to b$$

Step 3 *Restricting the number of variables on R.H.S.:*

$S \to ASA$ is replaced by $S \to AD$, $D \to SA$

$A \to ASA$ is replaced by $A \to AE$, $E \to SA$

So the equivalent grammar in CNF is

$$G_3 = (\{S, A, B, C_b, D, E\}, \{b, c\}, P_3, S)$$

where P_3 consists of

$$S \to C_bA \mid AD$$

$$A \to c \mid C_bA \mid AE$$

$$B \to c,\ C_b \to b,\ D \to SA,\ E \to SA$$

EXAMPLE 6.23

Let $G = (V_N, \Sigma, P, S)$ be a context-free grammar without null productions or unit productions and k be the maximum number of symbols on the R.H.S. of

any production of G. Show that there exists an equivalent grammar G_1 in CNF, which has at most $(k-1)|P| + |\Sigma|$ productions.

Solution

In step 2 (Theorem 6.8), a production of the form $A \to X_1X_2 \ldots X_n$ is replaced by $A \to Y_1Y_2, \ldots, Y_n$ where $Y_i = X_i$ if $X_i \in V_N$ and Y_i is a new variable if $X_i \in \Sigma$. We also add productions of the form $Y_i \to X_i$ whenever $X_i \in \Sigma$. As there are $|\Sigma|$ terminals, we have a maximum of $|\Sigma|$ productions of the form $Y_i \to X_i$ to be added to the new grammar. In step 3 (Theorem 6.8), $A \to A_1A_2 \ldots A_n$ is replaced by $n-1$ productions, $A \to A_1D_1$, $D_1 \to A_2D_2 \ldots D_{n-2} \to A_{n-1}A_n$. Note that $n \leq k$. So the total number of new productions obtained in step 3, is at most $(k-1)|P|$. Thus the total number of productions in CNF is at most $(k-1)|P| + |\Sigma|$.

Example 6.24

Reduce the following CFG to GNF:

$$S \to ABb|a, \qquad A \to aaA, \qquad B \to bAb$$

Solution

The valid productions for a grammar in GNF are $A \to a\alpha$, where $a \in \Sigma$, $\alpha \in V_N^*$.

So, $S \to ABb$ can be replaced by $S \to ABC$, $C \to b$.

$A \to aaA$ can be replaced by $A \to aDA$, $D \to a$.

$B \to bAb$ can be replaced by $B \to bAC$, $C \to b$.

So the revised productions are:

$$S \to ABC \mid a, \qquad A \to aDA, \qquad B \to bAC, \qquad C \to b, \qquad D \to a.$$

Name S, A, B, C, D as A_1, A_2, A_3, A_4, A_5.

Now we proceed to step 2.

Step 2 $G_1 = (\{A_1, A_2, A_3, A_4, A_5\}, \{a, b\}, P_1, A_1)$ where P_1 consists of

$$A_1 \to A_2A_3A_4 \mid a, \quad A_2 \to aA_5A_2, \quad A_3 \to bA_2A_4, \quad A_4 \to b, \quad A_5 \to a$$

The only production to be modified using step 4 (refer to Theorem 6.9) is $A_1 \to A_2A_3A_4$.

Replace $A_1 \to A_2A_3A_4$ by $A_1 \to aA_5A_2A_3A_4$.

The required grammar in GNF is

$G_2 = (\{A_1, A_2, A_3, A_4, A_5\}, \{a, b\}, P_2, A_1)$ where P_2 consists of

$$A_1 \to aA_5A_3A_4 \mid a$$
$$A_2 \to aA_5A_2$$
$$A_3 \to bA_2A_4, \qquad A_4 \to b, \qquad A_5 \to a$$

EXAMPLE 6.25

If a context-free grammar is defined by the productions

$$S \to a \mid Sa \mid bSS \mid SSb \mid SbS$$

show that every string in $L(G)$ has more a's than b's.

Proof We prove the result by induction on $|w|$, where $w \in L(G)$.

When $|w| = 1$, then $w = a$. So there is basis for induction.

Assume that $S \overset{*}{\Rightarrow} w$, $|w| < n$ implies that w has more a's than b's. Let $|w| = n > 1$. Then the first step in the derivation $S \overset{*}{\Rightarrow} w$ is $S \Rightarrow bSS$ or $S \Rightarrow SSb$ or $S \Rightarrow SbS$. In the first case, $S \Rightarrow bSS \overset{*}{\Rightarrow} bw_1w_2 = w$ for some $w_1, w_2 \in \Sigma^*$ and $S \overset{*}{\Rightarrow} w_1$, $S \overset{*}{\Rightarrow} w_2$. By induction hypothesis each of w_1 and w_2 has more a's than b's. So w_1w_2 has at least two more a's than b's. Hence bw_1w_2 has more a's than b's. The other two cases are similar. By the principle of induction, the result is true for all $w \in L(G)$.

EXAMPLE 6.26

Show that a CFG G with productions $S \to SS \mid (S) \mid \Lambda$ is ambiguous.

Solution

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow \Lambda(S) \Rightarrow \Lambda(\Lambda) = (\Lambda)$$

Also,

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (\Lambda)S \Rightarrow (\Lambda)\Lambda = (\Lambda)$$

Hence G is ambiguous.

EXAMPLE 6.27

Is it possible for a regular grammar to be ambiguous?

Solution

Let $G = (V_N, \Sigma, P, S)$ be regular. Then every production is of the form $A \to aB$ or $A \to b$. Let $w \in L(G)$. Let $S \overset{*}{\Rightarrow} w$ be a leftmost derivation. We prove that any leftmost derivation $A \overset{*}{\Rightarrow} w$, for every $A \in V_N$ is unique by induction on $|w|$. If $|w| = 1$, then $w = a \in T$. The only production is $A \to a$. Hence there is basis for induction. Assume that any leftmost derivation of the form $A \overset{*}{\Rightarrow} w$ is unique when $|w| = n - 1$. Let $|w| = n$ and $A \overset{*}{\Rightarrow} w$ be a leftmost derivation.

Take $w = aw_1$, $a \in T$. Then the first step of $A \overset{*}{\Rightarrow} w$ has to be $A \Rightarrow aB$ for some $B \in V_N$. Hence the leftmost derivation $A \overset{*}{\Rightarrow} w$ can be split into $A \Rightarrow aB \overset{*}{\Rightarrow} aw_1$. So, we get a leftmost derivation $B \overset{*}{\Rightarrow} w_1$. By induction hypothesis, $B \overset{*}{\Rightarrow} w_1$ is unique. So, we get a unique leftmost derivation of w. Hence a regular grammar cannot be ambiguous.

SELF-TEST

1. Consider the grammer G which has the productions

$$A \rightarrow a \mid Aa \mid bAA \mid AAb \mid AbA$$

and answer the following questions:

(a) What is the start symbol of G?
(b) Is *aaabb* in $L(G)$?
(c) Is *aaaabb* in $L(G)$?
(d) Show that *abb* is not in $L(G)$.
(e) Write the labels of the nodes of the following derivation tree T which are not labelled. It is given that T is the derivation tree whose yield is in $\{a, b\}^*$.

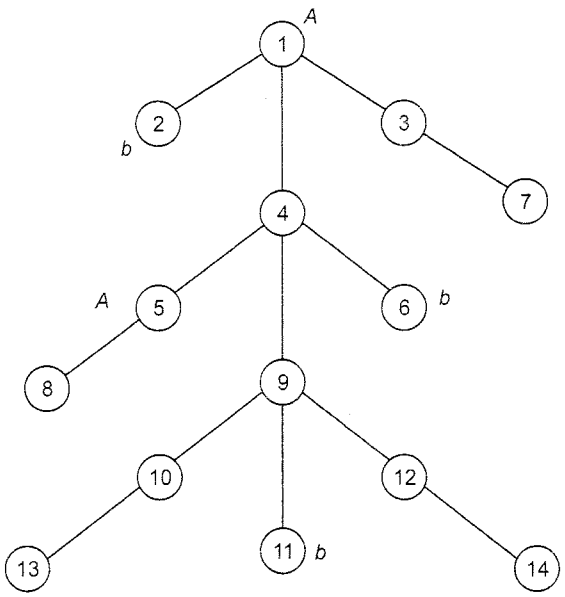


Fig. 6.14 Derivation tree for Question 1(e).

2. Consider the grammar G which has the following productions

$$S \rightarrow aB|bA,\ A \rightarrow aS \mid bAA \mid a,\ B \rightarrow bS \mid aBB \mid b.$$

and state whether the following statements are true or false.

(a) $L(G)$ is finite.
(b) $abbbaa \in L(G)$
(c) $aab \notin L(G)$
(d) $L(G)$ has some strings of odd length.
(e) $L(G)$ has some strings of even length.

3. State whether the following statements are true or false.
   (a) A regular language is context-free.
   (b) There exist context-free languages that are not regular.
   (c) The class of context-free languages is closed under union.
   (d) The class of context-free languages is closed under intersection.
   (e) The class of context-free languages is closed under complementation.
   (f) Every finite subset of $\{a, b\}^*$ is a context-free language.
   (g) $\{a^n b^n c^n \mid n \geq 1\}$ is a context-free language.
   (h) Any derivation tree for a regular grammar is a binary tree.

EXERCISES

6.1 Find a derivation tree of $a * b + a * b$ given that $a * b + a * b$ is in $L(G)$, where G is given by $S \rightarrow S + S \mid S * S$, $S \rightarrow a \mid b$.

6.2 A context-free grammar G has the following productions:

$$S \rightarrow 0S0 \mid 1S1 \mid A, \qquad A \rightarrow 2B3, \qquad B \rightarrow 2B3 \mid 3$$

Describe the language generated by the parameters.

6.3 A derivation tree of a sentential form of a grammar G is given in Fig. 6.15.

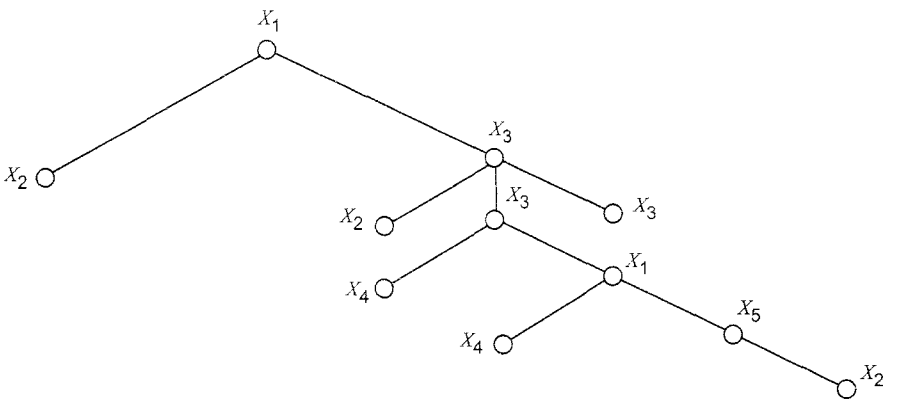


Fig. 6.15 Derivation tree for Exercise 6.3.

   (a) What symbols are necessarily in V_N?
   (b) What symbols are likely to be in Σ?
   (c) Determine if the following strings are sentential forms: (i) X_4X_2, (ii) $X_2X_2X_3X_2X_3X_3$, and (iii) $X_2X_4X_4X_2$.

6.4 Find (i) a leftmost derivation, (ii) a rightmost derivation, and (iii) a derivation which is neither leftmost nor rightmost of *abababa*, given that *abababa* is in $L(G)$, where G is the grammar given in Example 6.4.

6.5 Consider the following productions:

$$S \rightarrow aB \mid bA$$
$$A \rightarrow aS \mid bAA \mid a$$
$$B \rightarrow bS \mid aBB \mid b$$

For the string *aaabbabbba*, find
(a) the leftmost derivation,
(b) the rightmost derivation, and
(c) the parse tree.

6.6 Show that the grammar $S \rightarrow a \mid abSb \mid aAb$, $A \rightarrow bS \mid aAAb$ is ambiguous.

6.7 Show that the grammar $S \rightarrow aB \mid ab$, $A \rightarrow aAB \mid a$, $B \rightarrow ABb \mid b$ is ambiguous.

6.8 Show that if we apply Theorem 6.4 first and then Theorem 6.3 to a grammar G, we may not get a reduced grammar.

6.9 Find a reduced grammar equivalent to the grammar $S \rightarrow aAa$, $A \rightarrow bBB$, $B \rightarrow ab$, $C \rightarrow aB$.

6.10 Given the grammar $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow C \mid b$, $C \rightarrow D$, $D \rightarrow E$, $E \rightarrow a$, find an equivalent grammar which is reduced and has no unit productions.

6.11 Show that for getting an equivalent grammar in the most simplified form, we have to eliminate unit productions first and then the redundant symbols.

6.12 Reduce the following grammars to Chomsky normal form:
(a) $S \rightarrow 1A \mid 0B$, $A \rightarrow 1AA \mid 0S \mid 0$, $B \rightarrow 0BB \mid 1S \mid 1$
(b) $G = (\{S\}, \{a, b, c\}, \{S \rightarrow a \mid b \mid cSS\}, S)$
(c) $S \rightarrow abSb \mid a \mid aAb$, $A \rightarrow bS \mid aAAb$.

6.13 Reduce the grammars given in Exercises 6.1, 6.2, 6.6, 6.7, 6.9, 6.10 to Chomsky normal form.

6.14 Reduce the following grammars to Greibach normal form:
(a) $S \rightarrow SS$, $S \rightarrow 0S1 \mid 01$
(b) $S \rightarrow AB$, $A \rightarrow BSB$, $A \rightarrow BB$, $B \rightarrow aAb$, $B \rightarrow a$, $A \rightarrow b$
(c) $S \rightarrow A0$, $A \rightarrow 0B$, $B \rightarrow A0$, $B \rightarrow 1$

6.15 Reduce the grammars given in Exercises 6.1, 6.2, 6.6, 6.7, 6.9, 6.10 to Greibach normal form.

6.16 Construct the grammars in Chomsky normal form generating the following:
(a) $\{wcw^r \mid w \in 0\ \{a, b\}^*\}$,
(b) the set of all strings over $\{a, b\}$ consisting of equal number of a's and b's,

(c) $\{a^m b^n \mid m \neq n,\ m,\ n \geq 1\}$, and
(d) $\{a^n b^m c^n \mid m,\ n \geq 1\}$.

6.17 Construct grammars in Greibach normal form generating the sets given in Exercise 6.16.

6.18 If $w \in L(G)$ and $|w| = k$, where G is in (i) Chomsky normal form, (ii) Greibach normal form, what can you say about the number of steps in the derivation of w?

6.19 Show that the language $\{a^{n^2} \mid n \geq 1\}$ is not context-free.

6.20 Show that the following are not context-free languages:
(a) The set of all strings over $\{a, b, c\}$ in which the number of occurrences of a, b, c is the same.
(b) $\{a^m b^m c^n \mid m \leq n \leq 2m\}$.
(c) $\{a^m b^n \mid n = m^2\}$.

6.21 A context-free grammar G is called a right-linear grammar if each production is of the form $A \rightarrow wB$ or $A \rightarrow w$, where A, B are variables and $w \in \Sigma^*$. (G is said to be left-linear if the productions are of the form $A \rightarrow Bw$ or $A \rightarrow w$. G is linear if the productions are of the form $A \rightarrow vBw$ or $A \rightarrow w$.) Prove the following:
(a) A right-linear or left-linear grammar is equivalent to a regular grammar.
(b) A linear grammar is not necessarily equivalent to a regular grammar.

6.22 A context-free grammar G is said to be self-embedding if there exists some useful variable A such that $A \overset{*}{\Rightarrow} uAv$, where $u, v \in \Sigma^*$, $u, v \neq \Lambda$. Show that a context-free language is regular iff it is generated by a nonselfembedding grammar.

6.23 Show that every context-free language without Λ is generated by a context-free grammar in which all productions are of the form $A \rightarrow a$, $A \rightarrow a\alpha b$.

7 Pushdown Automata

In this chapter we introduce pushdown automaton (pda). We discuss two types of acceptance of sets by pushdown automata. Finally, we prove that the sets accepted by pushdown automata are precisely the class of context-free languages.

7.1 BASIC DEFINITIONS

We have seen that the regular languages are precisely those accepted by finite automata. If M is a finite automaton accepting L, it is constructed in such a way that states act as a form of primitive memory. The states 'remember' the variables encountered in the course of derivation of a string. (In M, the states correspond to variables.) Let us consider $L = \{a^n b^n \mid n \geq 1\}$. This is a context-free language but not regular. ($S \rightarrow aSb \mid ab$ generates L. Using the pumping lemma we can show that L is not regular; cf. Example 5.20.)

A finite automaton cannot accept L, i.e. strings of the form $a^n b^n$, as it has to remember the number of a's in a string and so it will require an infinite number of states. This difficulty can be avoided by adding an auxiliary memory in the form of a 'stack' (In a stack we add the elements in a linear way. While removing the elements we follow the last-in-first-out (LIFO) basis, i.e. the most recently added element is removed first.) The a's in the given string are added to the stack. When the symbol b is encountered in the input string, an a is removed from the stack. Thus the matching of number of a's and the number of b's is accomplished. This type of arrangement where a finite automaton has a stack leads to the generation of a pushdown automaton.

Before giving the rigorous definition, let us consider the components of a pushdown automaton and the way it operates. It has a read-only input tape,

an input alphabet, a finite state control, a set of final states, and an initial state as in the case of an FA. In addition to these, it has a stack called the pushdown store (abbreviated PDS). It is a read-write pushdown store as we add elements to PDS or remove elements from PDS. A finite automaton is in some state and on reading, an input symbol moves to a new state. The pushdown automaton is also in some state and on reading an input symbol and the topmost symbol in PDS, it moves to a new state and writes (adds) a string of symbols in PDS. Figure 7.1 illustrates the pushdown automaton.

We now give a formal definition of a pushdown automaton.

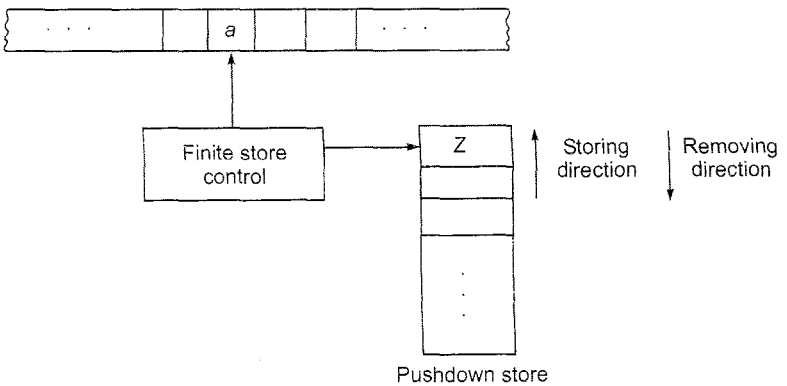


Fig. 7.1 Model of a pushdown automaton.

Definition 7.1 A pushdown automaton consists of

(i) a finite nonempty set of states denoted by Q,
(ii) a finite nonempty set of input symbols denoted by Σ,
(iii) a finite nonempty set of pushdown symbols denoted by Γ,
(iv) a special state called the initial state denoted by q_0,
(v) a special pushdown symbol called the *initial symbol* on the pushdown store denoted by Z_0,
(vi) a set of final states, a subset of Q denoted by F, and
(vii) a transition function δ from $Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.

Symbolically, a pda is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

Note: When $\delta(q, a, Z) = \emptyset$ for $(q, a, Z) \in Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$, we do not mention it.

EXAMPLE 7.1

Let

$$A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

$$Q = \{q_0, q_1, q_f\}, \quad \Sigma = \{a, b\}, \quad \Gamma = \{a, Z_0\}, \quad F = \{q_f\}$$

and δ is given by

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}, \ \delta(q_1, b, a) = \{(q_1, \Lambda)\}$$
$$\delta(q_0, a, a) = \{(q_0, aa)\}, \ \delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$$
$$\delta(q_0, b, a) = \{(q_1, \Lambda)\}$$

Remarks 1. $\delta(q, a, Z)$ is a finite subset of $Q \times \Gamma^*$. The elements of $\delta(q, a, Z)$ are of the form (q', α), where $q' \in Q$, $\alpha \in \Gamma^*$. $\delta(q, a, Z)$ may be the empty set.

2. At any time the pda is in some state q and the PDS has some symbols from Γ. The pda reads an input symbol a and the topmost symbol Z in PDS. Using the transition function δ, the pda makes a transition to a state q' and writes a string α after removing Z. The elements in PDS which were below Z initially are not disturbed. Here (q', α) is one of the elements of the finite set $\delta(q, a, Z)$. When $\alpha = \Lambda$, the topmost symbol, Z, is erased.

3. The behaviour of a pda is nondeterministic as the transition is given by any element of $\delta(q, a, Z)$.

4. As δ is defined on $Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$, the pda may make transition without reading any input symbol (when $\delta(q, \Lambda, Z)$ is defined as a nonempty set for $q \in Q$ and $Z \in \Gamma$). Such transitions are called Λ-moves.

5. The pda cannot take a transition when PDS is empty (We can apply δ only when the pda reads an input symbol and the topmost pushdown symbol in PDS). In this case the pda halts.

6. When we write $\alpha = Z_1Z_2 \ldots Z_m$ in PDS, Z_1 is the topmost element, Z_2 is below Z_1, etc. and Z_m is below Z_{m-1}.

In the case of finite automaton, it is enough to specify the current state at any time and the remaining input string to be processed. But as we have the additional structure, namely the PDS in pda, we have to specify the current state, the remaining input string to be processed, and the symbols in the PDS. This leads us to the next definition.

Definition 7.2 Let $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a pda. An instantaneous description (ID) is (q, x, α), where $q \in Q$, $x \in \Sigma^*$ and $\alpha \in \Gamma^*$.

For example, $(q, a_1a_2 \ldots a_n, Z_1Z_2 \ldots Z_m)$ is an ID. This describes the pda when the current state is q, the input string to be processed is $a_1a_2 \ldots a_n$. The pda will process $a_1a_2 \ldots a_n$ in that order. The PDS has $Z_1, Z_2, \ldots, Z_m$ with Z_1 at the top. Z_2 is the second element from the top, etc. and Z_m is the lowest element in PDS.

Definition 7.3 An initial ID is (q_0, x, Z_0). This means that initially the pda is in the initial state q_0, the input string to be processed is x, and the PDS has only one symbol, namely Z_0.

Note: In an ID (q, x, α), x may be Λ. In this case the pda makes a Λ-move.

For a finite automaton, the working can be described in terms of change of states. In the case of pda, because of its additional structure, namely PDS, the working can be described in terms of change of IDs. So we have the following definition:

Definition 7.4 Let A be a pda. A move relation, denoted by $\vdash$, between IDs is defined as

$$(q, a_1a_2 \ldots a_n, Z_1Z_2 \ldots Z_m) \vdash (q', a_2a_3 \ldots a_n, \beta Z_2 \ldots Z_m)$$

if $\delta(q, a_1, Z_1)$ contains (q', β).

Note: The move relation

$$(q, a_1a_2 \ldots a_n, Z_1Z_2 \ldots Z_m) \vdash (q', a_2a_3 \ldots a_n, \beta Z_2 \ldots Z_m)$$

can be described as follows: The pda in state q with $Z_1Z_2 \ldots Z_m$ in PDS (Z_1 is at the top) reads the input symbol a_1. When $(q', \beta) \in \delta(q, a_1, Z_1)$, the pda moves to a state q' and writes β on the top of $Z_2 \ldots Z_m$. After this transition, the input string to be processed is $a_2a_3 \ldots a_n$.

If $\beta = Y_1Y_2 \ldots Y_k$, then Fig. 7.2 illustrates the move relation.

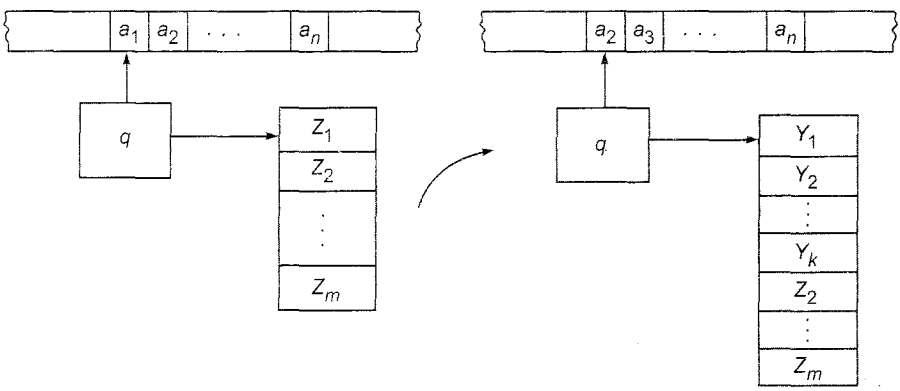


Fig. 7.2 An illustration of the move relation.

Remark As $\vdash$ defines a relation in the set of all IDs of a pda, we can define the reflexive-transitive closure $\overset{*}{\vdash}$ which represents a definite sequence of n moves, where n is any non-negative integer.

If $(q, x, \alpha) \overset{*}{\vdash} (q', y, \beta)$ represents n moves, we write $(q, x, \alpha) \overset{n}{\vdash} (q', y, \beta)$. In particular, $(q, x, \alpha) \overset{0}{\vdash} (q, x, \alpha)$. Also, $(q, x, \alpha) \overset{*}{\vdash} (q', y, \beta)$ can be split as

$$(q, x, a) \vdash (q_1, x_1, \alpha_1) \vdash (q_2, x_2, \alpha_2) \vdash \ldots \vdash (q', y, P)$$

for some $x_1, x_2, \ldots, \in \Sigma^*$, $\alpha_1, \alpha_2, \ldots, \in \Gamma^*$.

Note: When we deal with more than one pda, we also specify the pda while describing the move relation. For example, a move relation in A is denoted by $\underset{A}{\vdash}$.

The next two results are properties of the relation $\vdash^*$ and are frequently used in constructions and proofs.

Result 1 If

$$(q_1, x, \alpha) \vdash^* (q_2, \Lambda, \beta) \tag{7.1}$$

then for every $y \in \Sigma^*$,

$$(q_1, xy, \alpha) \vdash^* (q_2, y, \beta) \tag{7.2}$$

Conversely, if $(q_1, xy, \alpha) \vdash^* (q_2, y, \beta)$ for some $y \in \Sigma^*$, then $(q_1, x, \alpha) \vdash^* (q_2, \Lambda, \beta)$.

Proof The result can be easily understood once we refer to Fig. 7.2, providing an illustration of the move relation.

If the pda is in state q_1 with α in PDS, and the moves given by (7.1) are effected by processing the string x, the pda moves to state q_2 with β in PDS. The same transition is effected by starting with the input string xy and processing only x. In this case, y remains to be processed and hence we get (7.2).

We can prove the converse part, i.e. (7.2) implies (7.1) in a similar way.

Result 2 If

$$(q, x, \alpha) \vdash^* (q', \Lambda, \gamma) \tag{7.3}$$

then for every $\beta \in \Gamma^*$,

$$(q, x, \alpha\beta) \vdash^* (q', \Lambda, \gamma\beta) \tag{7.4}$$

Proof The sequence of moves given by (7.3) can be split as

$$(q, x, \alpha) \vdash (q_1, x_1, \alpha_1) \vdash (q_2, x_2, \alpha_2) \vdash \cdots \vdash (q', \Lambda, \gamma)$$

Consider $(q_i, x_i, \alpha_i) \vdash (q_{i+1}, x_{i+1}, \alpha_{i+1})$. Let $\alpha_i = Z_1Z_2 \ldots Z_m$. As a result of this move, Z_1 is erased and some string is placed above $Z_2 \ldots Z_m$. So, $Z_2 \ldots Z_m$ is not affected. If we have β below $Z_2 \ldots Z_m$, then also $Z_2 \ldots Z_m\beta$ is not affected. So we obtain $(q_i, x_i, \alpha_i\beta) \vdash (q_{i+1}, x_{i+1}, \alpha_{i+1}\beta)$. Therefore, we get a sequence of moves

$$(q, x, \alpha\beta) \vdash (q_1, x_1, \alpha_1\beta) \vdash \ldots \vdash (q', \Lambda, \gamma\beta)$$

i.e.

$$(q, x, \alpha\beta) \vdash^* (q', \Lambda, \gamma\beta) \quad \blacksquare$$

Note: In general, (7.4) need not imply (7.3). Consider, for instance,

$$A = (\{q_0\}, \{a, b\}, \{Z_0\}, \delta, q_0, Z_0, \emptyset)$$

where

$$\delta(q_0, a, Z_0) = \{(q_0, \Lambda)\}, \ \delta(q_0, b, Z_0) = \{(q_0, Z_0Z_0)\}$$

$$\begin{aligned}
&(q_0, aab, Z_0Z_0Z_0Z_0) \\
&\vdash (q_0, ab, Z_0Z_0Z_0) \\
&\vdash (q_0, b, Z_0Z_0) \\
&\vdash (q_0, \Lambda, Z_0Z_0Z_0)
\end{aligned}$$

i.e.

$$(q_0, aab, Z_0Z_0Z_0Z_0) \vdash^* (q_0, \Lambda, Z_0Z_0Z_0)$$

However, $(q_0, aab, Z_0) \vdash (q_0, ab, \Lambda)$; hence the pda cannot make any more transitions as the PDS is empty. This shows that (7.4) does not imply (7.3) if we assume $\alpha = Z_0Z_0Z_0$, $\beta = Z_0$, $\gamma = Z_0Z_0$.

EXAMPLE 7.2

$$A = (\{q_0, q_1, q_f\}, \{a, b, c\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_f\})$$

is a pda, where δ is defined as

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}, \quad \delta(q_0, b, Z_0) = \{(q_0, bZ_0)\} \tag{7.5}$$

$$\delta(q_0, a, a) = \{(q_0, aa)\}, \quad \delta(q_0, b, a) = \{(q_0, ba)\} \tag{7.6}$$

$$\delta(q_0, a, b) = \{(q_0, ab)\}, \quad \delta(q_0, b, b) = \{(q_0, bb)\} \tag{7.7}$$

$$\delta(q_0, c, a) = \{(q_1, a)\}, \quad \delta(q_0, c, b) = \{(q_1, b)\}, \ \delta(q_0, c, Z_0) = \{(q_1, Z_0)\} \tag{7.8}$$

$$\delta(q_1, a, a) = \delta(q_1, b, b) = \{(q_1, \Lambda)\} \tag{7.9}$$

$$\delta(q_1, \Lambda, Z_0) = \{(q_f, Z_0)\} \tag{7.10}$$

We can explain δ as follows:

If A is in initial ID, then using Rule (7.5), A pushes the first symbol of the input string on PDS if it is a or b. By Rules (7.6) and (7.7), the symbols of the input string are pushed on PDS until it sees the centre-marker c. By Rule (7.8), on seeing c, the pda moves to state q_1 without making any changes in PDS. By Rule (7.9), the pda erases the topmost symbol if it coincides with the current input symbol (i.e. if they do not match, the pda halts). By Rule (7.10), the pda reaches the final state q_f only when the input string is exhausted, and then the PDS has only Z_0.

We can explain the concepts of ID, moves, etc. for this pda A. Suppose the input string is *acab*. We will see how the pda processes this string. An initial configuration is $(q_0, bacab, Z_0)$. We get the following moves:

$(q_0, bacab, Z_0) \vdash (q_0, acab, bZ_0)$ by Rule (7.5)

$\vdash (q_0, cab, abZ_0)$ by Rule (7.7)

$\vdash (q_1, ab, abZ_0)$ by Rule (7.8)

$\vdash (q_0\ b, bZ_0)$ by Rule (7.9)

$\vdash (q_1, \Lambda, Z_0)$ by Rule (7.10)

$\vdash (q_f, \Lambda, Z_0)$ by Rule (7.10)

i.e.

$$(q_0, bacab, Z_0) \vdash^* (q_f, Z_0)$$

Proceeding in a similar way, we can show that

$$(q_0, wcw^T, Z_0) \vdash^* (q_f, \Lambda, Z_0) \quad \text{for all } w \in \{a, b\}^*$$

Suppose an initial configuration is $(q_0, abcbb, Z_0)$. Then, we have

$$\begin{aligned}(q_0, abcbb, Z_0) &\vdash (q_0, bcbb, aZ_0) && \text{by Rule (7.5)}\\ &\vdash (q_0, cbb, baZ_0) && \text{by Rule (7.6)}\\ &\vdash (q_1, bb, baZ_0) && \text{by Rule (7.8)}\\ &\vdash (q_1, b, aZ_0) && \text{by Rule (7.9)}\end{aligned}$$

Once the pda is in ID (q_1, b, aZ_0), it has to halt as $\delta(q_1, b, a) = \emptyset$. Hence, we have

$$(q_0, abcbb, Z_0) \vdash^* (q_1, b, aZ_0)$$

As $\delta(q_0, c, Z_0) = \emptyset$, the pda cannot make any transition if it starts with an ID of the form (q_0, cw, Z_0).

Note: In Example 7.2, each $\delta(q, a, Z)$ is either empty or consists of a single element. So for making transitions, the pda has only one choice and the behaviour is deterministic.

In general, a deterministic pda can be defined as follows:

Definition 7.5 A pda $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is deterministic if (i) $\delta(q, a, Z)$ is either empty or a singleton, and (ii) $\delta(q, \Lambda, Z) \neq \emptyset$ implies $\delta(q, a, Z) = \emptyset$ for each $a \in \Sigma$.

Consider the pda given in Example 7.2. $\delta(q, a, Z)$ given by Rules (7.5)–(7.10) are singletons. Also, $\delta(q_1, a, Z_0) = \emptyset$ and $\delta(q_1, a, Z_0) = \emptyset$ for all $a \in \Sigma$. So the pda given in Example 7.2 is deterministic.

7.2 ACCEPTANCE BY pda

A pda has final states like a nondeterministic finite automaton and has also the additional structure, namely PDS. So we can define acceptance of input strings by pda in terms of final states or in terms of PDS.

Definition 7.6 Let $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a pda. The set accepted by pda by final state is defined by

$$T(A) = \{w \in \Sigma^* | (q_0, w, Z_0) \vdash^* (q_f, \Lambda, \alpha) \text{ for some } q_f \in F \text{ and } \alpha \in \Gamma^*\}$$

EXAMPLE 7.3

Construct a pda A accepting $L = \{wcw^T | w \in \{a, b\}^*\}$ by final state.

Solution

Consider the pda given in Example 7.2. Let $wcw^T \in L$. Write $w = a_1a_2 \ldots a_n$, where each a_i is either a or b. Then, we have

$$
\begin{aligned}
&(q_0,\ a_1a_2 \ldots a_ncw^T,\ Z_0) \\
&\vdash^* (q_0,\ cw^T,\ a_na_{n-1} \ldots a_1Z_0) && \text{by Rules (7.5)–(7.7)} \\
&\vdash^* (q_1,\ a_na_{n-1} \ldots a_1,\ a_na_{n-1} \ldots a_1Z_0) && \text{by Rule (7.8)} \\
&\vdash^* (q_1,\ \Lambda,\ Z_0) && \text{by Rule (7.9)} \\
&\vdash (q_f,\ \Lambda,\ Z_0) && \text{by Rule (7.10)}
\end{aligned}
$$

Therefore, $wcw^T \in T(A)$, i.e. $L \subseteq T(A)$.

To prove the reverse inclusion, it is enough to show that $L^c \subseteq T(A)^c$. Let $x \in L^c$.

Case 1 x does not have the symbol c. In this case the pda never makes a transition to q_1. So the pda cannot make a transition to q_f as we cannot apply Rule (7.10). Thus, $x \in T(A)^c$.

Case 2

$$x = w_1cw_2, \qquad w_2 \neq w_1^T$$

$$
\begin{aligned}
&(q_0,\ w_1cw_2,\ Z_0) \\
&\vdash^* (q_0,\ cw_2,\ w_1^TZ_0) \\
&\vdash (q_1,\ w_2,\ w_1^TZ_0)
\end{aligned}
$$

As $w_2 \neq w_1^T$, the pda cannot reach an ID of the form (q_1, Λ, Z_0). So we cannot apply (7.10). Therefore, $x \in T(A)^c$.

Thus we have proved $L^c \subseteq T(A)^c$.

The next definition describes the second type of acceptance.

Definition 7.7 Let $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a pda. The set $N(A)$ accepted by null store (or empty store) is defined by

$$N(A) = \{w \in \Sigma^* | (q_0, w, Z_0) \vdash^* (q, \Lambda, \Lambda) \text{ for some } q \in Q\}$$

In other words, w is in $N(A)$ if A is in initial ID (q_0, w, Z_0) and empties the PDS after processing all the symbols of w. So in defining $N(A)$, we consider the change brought about on PDS by application of w, and not the transition of states.

EXAMPLE 7.4

Consider the pda A given by Example 7.2 with an additional rule:

$$\delta(q_f, \Lambda, Z_0) = \{(q_f, \Lambda)\} \tag{7.11}$$

Then,

$$N(A) = \{wcw^T \mid w \in \{a, b\}^*\}$$

Solution

From the construction of A, we see that the Rules (7.5)–(7.10) cannot erase Z_0. We make a provision for erasing Z_0 from PDS by Rule (7.11). By Example 7.2, $wcw^T \in T(A)$ if and only if the pda reaches the ID (q_f, Λ, Z_0). By (7.11), PDS can be emptied by Λ-moves if and only if the pda reaches the ID (q_f, Λ, Z_0). Hence,

$$N(A) = \{wcw^T \mid w \in \{a, b\}^*\}$$

In the next theorem we prove that the set accepted by a pda A by null store is accepted by some pda B by final state.

Theorem 7.1 If $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a pda accepting L by empty store, we can find a pda

$$B = (Q', \Sigma, \Gamma', \delta_B, q'_0, Z_0, F')$$

which accepts L by final state, i.e. $L = N(A) = T(B)$.

Proof B is constructed in such a way that (i) by the initial move of B, it reaches an initial ID of A, (ii) by the final move of B, it reaches its final state, and (iii) all intermediate moves of B are as in A.

Let us define B as follows:

$$B = (Q', \Sigma, \Gamma', \delta_B, q'_0, Z'_0, F)$$

where

q'_0 is a new state (not in Q),

$F' = \{q_f\}$, with q_f as a new state (not in Q),

$Q' = Q \cup \{q'_0, q_f\}$,

Z'_0 is a new start symbol for PDS of B,

$\Gamma' = \Gamma \cup \{Z_0\}$, and

δ_B is given by the rules R_1, R_2, R_3

with

R_1: $\delta_B(q'_0, \Lambda, Z'_0) = \{(q_0, Z_0Z'_0)\}$,

R_2: $\delta_B(q, a, Z) = \delta(q, a, Z)$ for all (q, a, Z) in $Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$

R_3: $\delta_B(q, \Lambda, Z'_0) = \{(q_f, \Lambda)\}$ for all $q \in Q$.

By R_1, the pda B moves from an initial ID of B to an initial ID of A. R_1 gives a Λ-move. As a result of R_1, B moves to the initial state of A with the start symbol Z_0 on the top of PDS.

R_2 is used to simulate A. Once B reaches an initial ID of A, R_2 can be used to simulate moves of A. We can repeatedly apply R_2 until Z'_0 is pushed to the top of PDS. As Z'_0 is a new pushdown symbol, we have to use R_3.

R_3 gives a Λ-move. Using R_3, B moves to the new (final) state q_f erasing Z'_0 in PDS.

Thus the behaviour of B and A are similar except for the Λ-moves given by R_1 and R_3. Also, $w \in T(B)$ if and only if B reaches q_f, i.e. if and only

if the PDS has no symbols from Γ (since B can reach q_f only by the application of R_3). This suggests that $T(B) = N(A)$.

Now we prove rigorously that $N(A) = T(B)$. Suppose $w \in N(A)$. Then by the definition of $N(A)$, $(q_0, w, Z_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \Lambda)$ for some $q \in Q$. Using R_2, we see that

$$(q_0, w, Z_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \Lambda)$$

By Result 2,

$$(q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, Z'_0) \tag{7.12}$$

By R_1,

$$(q'_0, \Lambda, Z_0) \underset{B}{\vdash} (q_0, \Lambda, Z_0Z'_0)$$

By Result 1, we have

$$(q'_0, w, Z'_0) \underset{B}{\vdash} (q_0, w, Z'_0Z_0) \tag{7.13}$$

By R_3,

$$(q, \Lambda, Z'_0) \underset{B}{\vdash} (q_f, \Lambda, \Lambda) \tag{7.14}$$

Combining (7.12)–(7.14), we have

$$(q'_0, w, Z'_0) \overset{*}{\underset{B}{\vdash}} (q_f, \Lambda, \Lambda)$$

This proves that $w \in T(B)$, i.e. $N(A) \subseteq T(B)$.

To prove $T(B) \subseteq N(A)$, we start with $w \subseteq T(B)$. Then

$$(q'_0, w, Z'_0) \overset{*}{\underset{B}{\vdash}} (q_f, \Lambda, \alpha) \tag{7.15}$$

But B can reach q_f only by the application of R_3. To apply R_3, Z'_0 should be the topmost element on PDS. Z'_0 is placed initially, and so when it is on the top there are no other elements in PDS. So $\alpha = \Lambda$, and (7.15) actually reduces to

$$(q'_0, w, Z'_0) \overset{*}{\underset{B}{\vdash}} (q_f, \Lambda, \Lambda) \tag{7.16}$$

In (7.16), the initial and final steps are effected only by Λ-moves. The intermediate steps are induced by the corresponding moves of A. So (7.16) can be split as $(q'_0, \Lambda w, Z'_0) \underset{B}{\vdash} (q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, Z'_0)$ for some $q \in Q$. Thus, $(q'_0, \Lambda w, Z'_0) \underset{B}{\vdash} (q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, Z'_0) \underset{B}{\vdash} (q_f, \Lambda, \Lambda)$. As we get $(q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, Z'_0)$ by applying R_2 several times and R_2 does not affect Z'_0 at the bottom, we have $(q_0, w, Z_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \Lambda)$. By the construction of R_2, we have $(q_0, w, Z_0) \overset{*}{\underset{A}{\vdash}} (q, \Lambda, \Lambda)$, which means $w \in N(A)$. Thus, $T(B) \subseteq N(A)$, and hence $T(B) = N(A) = L$. ∎

Note: From the construction of B, it is easy to see that B is deterministic if and only if A is deterministic.

EXAMPLE 7.5

Consider the pda A given in Example 7.1 (Take $F = \emptyset$). Determine $N(A)$. Also construct a pda B such that $T(B) = N(A)$.

Solution

$$A = (\{q_0, q_1\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, \emptyset),$$

where δ is given by

$$\begin{aligned}
R_1\text{: } \delta(q_0, a, Z_0) &= \{(q_0, aZ_0)\} \\
R_2\text{: } \delta(q_0, a, a) &= \{(q_0, aa)\} \\
R_3\text{: } \delta(q_0, b, a) &= \{(q_1, \Lambda)\} \\
R_4\text{: } \delta(q_1, b, a) &= \{(q_1, \Lambda)\} \\
R_5\text{: } \delta(q_1, \Lambda, Z_0) &= \{(q_1, \Lambda)\}
\end{aligned}$$

R_1 is used to store a in PDS if it is the first symbol of an input string. R_2 can be used repeatedly to store a^n in PDS. When b is encountered for the first time in the input string, a is erased (in PDS) using R_3. Also, the pda makes a transition to state q_1. After processing the entire input string, if Z_0 remains in PDS, it can be erased using the null move given by R_5. So, if $w = a^n b^n$, then we have

$$\begin{aligned}
(q_0, a^n b^n, Z_0) &\vdash^* (q_0, b^n, a^n Z_0) && \text{by applying } R_1 \text{ and } R_2 \\
&\vdash^* (q_1, \Lambda, Z_0) && \text{by applying } R_3 \text{ and } R_4 \\
&\vdash (q_1, \Lambda, \Lambda) && \text{by applying } R_5
\end{aligned}$$

Therefore, $a^n b^n \in N(A)$.

If $w \in N(A)$, then $(q_0, w, Z_0) \vdash^* (q_1, \Lambda, \Lambda)$. (Note that the PDS can be empty only when A is in state q_1.) Also, w should start with a. Otherwise, we cannot make any move. We store the symbol a in PDS if the current input symbol is a and the topmost symbol in PDS is a or Z_0. On seeing the input symbol b, the pda erases the symbol a in PDS. The pda enters the ID (q_1, Λ, Λ) only by the application of R_5. The pda can reach the ID (q_1, Λ, Z_0) only by erasing the a's in pda. This is possible only when the number of b's is equal to number of a's, and so $w = a^n b^n$. Thus, we have proved that $N(A) = \{a^n b^n \mid n \geq 1\}$.

Now let

$$B = (Q', \{a, b\}, \Gamma', \delta_B, q'_0, Z'_0, F')$$

where

$$Q' = \{q_0, q'_0, q_1, q_f\}, \qquad F' = \{q_1\}, \qquad \Gamma' = \{a, b, Z'_0\}$$

and δ_B is defined by

$$\delta_B(q'_0, \Lambda, Z'_0) = \{(q_0, Z_0Z_0)\}$$

$$\delta_B(q_1, a, Z_0) = \{(q_0, aZ_0)\}$$

$$\delta_B(q_0, a, a) = \{(q_0, aa)\}$$

$$\delta_B(q_0, b, a) = \{(q_1, \Lambda)\}$$

$$\delta_B(q_1, b, a) = \{(q_1, \Lambda)\}$$

$$\delta_B(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$$

$$\delta_B(q_0, \Lambda, Z'_0) = \{(q_f, \Lambda)\}$$

$$\delta_B(q_1, \Lambda, Z'_0) = \{(q_f, \Lambda)\}$$

Thus,

$$T(B) = N(A) = \{a^n b^n | n \geq 1\}$$

The following theorem asserts that the set accepted by a pda A by final state is accepted by some pda B by null store.

Theorem 7.2 If $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ accepts L by final state, we can find a pda B accepting L by empty store; i.e. $L = T(A) = N(B)$.

Proof B is constructed from A in such a way that (i) by the initial move of B an initial ID of A is reached, (ii) once B reaches an initial ID of A, it behaves like A until a final state of A is reached, and (iii) when B reaches a final state of A, it guesses whether the input string is exhausted. Then B simulates A or it erases all the symbols in PDS.

The actual construction of B is as follows:

$$B = (Q \cup \{q'_0, d\}, \Sigma, \Gamma \cup \{Z'_0\}, \delta_B, q'_0, Z'_0, \emptyset)$$

where q'_0 is a new state (not in Q), d is a new (dead) state, and Z'_0 is the new start symbol for PDS of B.

δ_B is defined by rules R_1, R_2, R_3 and R_4 as

R_1: $\delta_B(q'_0, \Lambda, Z'_0) = \{(q_0, Z_0Z_0)\}$

R_2: $\delta_B(q, a, Z) = \delta(q, a, Z)$ for all $a \in \Sigma$, $q \in Q$, $Z \in \Gamma$

R_3: $\delta_B(q, \Lambda, Z) = \delta(q, \Lambda, Z) \cup \{(d, \Lambda)\}$ for all $Z \in \Gamma \cup \{Z'_0\}$ and $q \in F$

R_4: $\delta_B(q, \Lambda, Z) = \{(d, \Lambda)\}$ for all $Z \in \Gamma \cup \{Z'_0\}$

Using R_1, B enters an initial ID of A and the start symbol Z_0 is placed on top of PDS.

Using R_2, B can simulate A until it reaches a final state of A. On reaching a final state of A, B makes a guess whether the input string is exhausted or

not. When the input string is not exhausted, B once again simulates A. Otherwise, B enters the dead state d. Rule R_4 gives a Λ-move. Using these Λ-moves, B erases all the symbols on PDS.

Now $w \in T(A)$ if and only if A reaches a final state. On reaching a final state of A, the pda can reach the state d and erase all the symbols in the PDS by Λ-moves. So, it is intuitively clear that $w \in T(A)$ if and only if $w \in N(B)$. We now prove rigorously that $T(A) = N(B)$.

Suppose $w \in T(A)$. Then for some $q \in F$, $\alpha \in \Gamma^*$,

$$(q_0, w, Z_0) \overset{*}{\underset{A}{\vdash}} (q, \Lambda, \alpha)$$

Using R_2, we get

$$(q_0, w, Z_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \alpha)$$

Applying Result 2, we obtain

$$(q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \alpha Z'_0) \tag{7.17}$$

As

$$(q'_0, \Lambda, Z'_0) \underset{B}{\vdash} (q_0, A, Z_0Z'_0)$$

using Result 1, we get

$$(q'_0, w, Z'_0) \underset{B}{\vdash} (q_0, w, Z_0Z'_0) \tag{7.18}$$

From (7.18) and (7.17), we can deduce

$$(q'_0, w, Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \alpha Z'_0) \tag{7.19}$$

By applying R_3 once and R_4 repeatedly, we get

$$(q, \Lambda, \alpha Z'_0) \overset{*}{\underset{B}{\vdash}} (d, \Lambda, \Lambda) \tag{7.20}$$

Relations (7.19) and (7.20) imply that $(q'_0, w, Z'_0) \overset{*}{\underset{B}{\vdash}} (d, \Lambda, \Lambda)$. Thus we have proved $T(A) \subseteq N(B)$.

To prove that $N(B) \subseteq T(A)$, we start with $w \in N(B)$. This means that for some state q of B,

$$(q'_0, w, Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \Lambda) \tag{7.21}$$

As the initial move of B can be made only by using R_1, the first move of (7.21) is $(q'_0, \Lambda w, Z'_0) \underset{B}{\vdash} (q_0, w, Z_0Z'_0)$.

Z'_0 in the PDS can be erased only when B enters d; B can enter d only when it reaches a final state q of A in an earlier step. So (7.21) can be split as

$$(q_0, \Lambda w, Z_0) \underset{B}{\vdash} (q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \alpha Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \Lambda)$$

for some $q \in F$ and $\alpha \in \Gamma^*$. But $(q_0, w, Z_0Z'_0) \overset{*}{\underset{B}{\vdash}} (q, \Lambda, \alpha Z'_0)$ can be obtained only by the application of R_2. So the moves involved are those induced by the moves of A. As Z'_0 is not a pushdown symbol in A, Z'_0 lying at the bottom is not affected by these moves. Hence

$$(q'_0, \Lambda w, Z'_0) \overset{*}{\underset{A}{\vdash}} (q, \Lambda, \alpha), \quad q \in F$$

So $w \in T(A)$ and $N(B) \subseteq T(A)$. Thus,

$$L = N(B) = T(A)$$

EXAMPLE 7.6

Construct a pda A accepting the set of all strings over $\{a, b\}$ with equal number of a's and b's.

Solution

Let

$$A = (\{q\}, [a, b], [Z_0, a, b], \delta, q, Z_0, \emptyset)$$

where δ is defined by the following rules:

$$\delta(q, a, Z_0) = \{(q, aZ_0)\} \quad \delta(q, b, Z_0) = \{(q, bZ_0)\}$$

$$\delta(q, a, a) = \{(q, aa)\} \quad \delta(q, b, b) = \{(q, bb)\}$$

$$\delta(q, a, b) = \{(q, \Lambda)\} \quad \delta(q, b, a) = \{(q, \Lambda)\}$$

$$\delta(q, \Lambda, Z_0) = \{(q, \Lambda)\}$$

The construction of δ is similar to that of the pda given in Example 7.2. But here we want to match the number of occurrences of a and b; so, the construction is simpler. We start by storing a symbol of the input string and continue storing until the other symbol occurs. If the topmost symbol in PDS is a and the current input symbol is b, a in PDS is erased. If w has equal number of a's and b's, then $(q, w, Z_0) \overset{*}{\vdash} (q, \Lambda, Z_0) \vdash (q, \Lambda, \Lambda)$. So $w \in N(A)$. We can show that $N(A)$ is the given set of strings over $\{a, b\}$ using the construction of δ.

7.3 PUSHDOWN AUTOMATA AND CONTEXT-FREE LANGUAGES

In this section we prove that the sets accepted by pda (by null store or final state) are precisely the context-free languages.

Theorem 7.3 If L is a context-free language, then we can construct a pda A accepting L by empty store, i.e. $L = N(A)$.

Proof We construct A by making use of productions in G.

Step 1 (Construction of A) Let $L = L(G)$, where $G = (V_N, \Sigma, P, S)$ is a context-free grammar. We construct a pda A as

$$A = ((q), \Sigma, V_N \cup \Sigma, \delta, q, S, \emptyset)$$

where δ is defined by the following rules:

$$R_1\text{: } \delta(q, \Lambda, A) = \{(q, \alpha) \mid A \to \alpha \text{ is in } P\}$$

$$R_2\text{: } \delta(q, a, a) = \{(q, \Lambda)\} \quad \text{for every } a \text{ in } \Sigma$$

We can explain the construction in the following way: The pushdown symbols in A are variables and terminals. If the pda reads a variable A on the top of PDS, it makes a Λ-move by placing the R.H.S. of any A-production (after erasing A). If the pda reads a terminal a on PDS and if it matches with the current input symbol, then the pda erases a. In other cases the pda halts.

If $w \in L(G)$ is obtained by a leftmost derivation

$$S \Rightarrow u_1A_1\alpha_1 \Rightarrow u_1u_2A_2\alpha_2\alpha_1 \Rightarrow \ldots \Rightarrow w,$$

then A can empty the PDS on application of input string w. The first move of A is by a Λ-move corresponding to $S \to u_1A_1\alpha_1$. The pda erases S and stores $u_1A_1\alpha_1$. Then using R_2, the pda erases the symbols in u_1 by processing a prefix of w. Now, the topmost symbol in PDS is A_1. Once again by applying the Λ-move corresponding to $A_1 \to u_2A_2\alpha_2$, the pda erases A_2 and stores $u_2A_2\alpha_2$ above α_1. Proceeding in this way, the pda empties the PDS by processing the entire string w.

Before proving that $L(G) = N(A)$ (step 2), we apply the construction to an example.

EXAMPLE 7.7

Construct a pda A equivalent to the following context-free grammar: $S \to 0BB$, $B \to 0S \mid 1S \mid 0$. Test whether 010^4 is in $N(A)$.

Solution

Define pda A as follows:

$$A = (\{q\}, \{0, 1\}, \{S, B, 0, 1\}, \delta, q, S, \emptyset)$$

δ is defined by the following rules:

$$R_1\text{: } \delta(q, \Lambda, S) = \{(q, 0BB)\}$$

$$R_2\text{: } \delta(q, \Lambda, B) = \{(q, 0S), (q, 1S), (q, 0)\}$$

$$R_3\text{: } \delta(q, 0, 0) = \{(q, \Lambda)\}$$

$$R_4\text{: } \delta(q, 1, 1) = \{(q, \Lambda)\}$$

$(q, 010^4, S)$

$\vdash (q, 010^4, 0BB)$ by Rule R_1

$\vdash (q, 10^4, BB)$ by Rule R_3

$\vdash (q, 10^4, 1SB)$ by Rule R_2 since $(q, 1S) \in \alpha(q, \Lambda, B)$

$\vdash (q, 0^4, SB)$ by Rule R_4

$\vdash (q, 0^4, 0BBB)$ by Rule R_1

$\vdash (q, 0^3, BBB)$ by Rule R_3

$\overset{*}{\vdash} (q, 0^3, 000)$ by Rule R_2 since $(q, 0) \in \alpha(q, \Lambda, B)$

$\overset{*}{\vdash} (q, \Lambda, \Lambda)$ by Rule R_3

Thus,

$$010^4 \subseteq N(A)$$

Note: After entering $(q, 10^4, BB)$, the pda may halt for a different sequence of moves, for example, $(q, 10^4, BB) \vdash (q, 10^4, 0B) \vdash (q, 10^4, 00)$. As $\delta(q, 1, 0)$ is the empty set, the pda halts.

Let us continue with the proof of the theorem.

Step 2 (Proof of the construction, i.e. $L(G) = N(A)$). First we prove $L(G) \subseteq N(A)$. Let $w \in L(G)$. Then it can be derived by a leftmost derivation. Any sentential form in a leftmost derivation is of the form $\mu A\alpha$, where $u \in \Sigma^*$, $A \in V_N$ and $\alpha \in (V_N \cup \Sigma)^*$. We prove the following auxiliary result: If $S \overset{*}{\Rightarrow} uA\alpha$ by a leftmost derivation, then

$$(q, uv, S) \overset{*}{\vdash} (q, v, A\alpha) \quad \text{for every } v \in \Sigma^* \tag{7.22}$$

We prove (7.22) by induction on the number of steps in the derivation of $uA\alpha$. If $S \overset{0}{\Rightarrow} uA$, then $u = \Lambda$, $\alpha = \Lambda$, and $S = A$. As $(q, v, S) \overset{*}{\vdash} (q, v, S)$, there is basis for induction.

Suppose $S \overset{n+1}{\Rightarrow} uA\alpha$ by a leftmost derivation. This derivation can be split as $S \overset{n}{\Rightarrow} u_1A_1\alpha_1 \Rightarrow uA\alpha$. If the A_1-production we apply in the last step is $A_1 \rightarrow u_2A\alpha_2$, then $u = u_1u_2$, $\alpha = \alpha_2\alpha_1$.

As $S \overset{n}{\Rightarrow} u_1A_1\alpha_1$, by induction hypothesis,

$$(q, u_1u_2v, S) \overset{*}{\vdash} (q, u_2v, A_1\alpha_1) \tag{7.23}$$

As $A_1 \rightarrow u_2A\alpha_2$ is a production in P, by Rule R_1 we get $(q, \Lambda, A_1) \vdash (q, \Lambda, u_2A\alpha_2)$. Applying Results 1 and 2 in Section 7.1, we get

$$(q, u_2v, A_1\alpha_1) \vdash (q, u_2v, u_2A\alpha_2\alpha_1)$$

$$\vdash (q, v, A\alpha_2\alpha_1) \quad \text{by Rule } R_2$$

Hence,

$$(q, u_2v, A_1\alpha_1) \overset{*}{\vdash} (q, v, A\alpha_2\alpha_1) \tag{7.24}$$

But $u_1u_2 = u$ and $\alpha_2\alpha_1 = \alpha$. So from (7.23) and (7.24), we have

$$(q, uv, S) \vdash^{*} (q, v, A\alpha)$$

Thus (7.22) is true for $S \overset{n+1}{\Rightarrow} uA\alpha$. By the principle of induction, (7.22) is true for any derivation. Now we prove that $L(G) \subseteq N(A)$. Let $w \in L(G)$. Then, w can be obtained from a leftmost derivation,

$$S \overset{*}{\Rightarrow} uAv \Rightarrow uu'v = w$$

From (7.22),

$$(q, uu'v, S) \vdash^{*} (q, u'v, Av)$$

As $A \rightarrow u'$ is in P,

$$(q, u'v, Av) \vdash^{*} (q, u'v, u'v)$$

By Rule R_2,

$$(q, u'v, u'v) \vdash^{*} (q, \Lambda, \Lambda)$$

Therefore,

$$w = uu'v \in N(A) \qquad \text{proving } L(G) \subseteq N(A)$$

Next we prove

$$N(A) \subseteq L(G)$$

Before proving the inclusion, let us prove the following auxiliary result:

$$S \overset{*}{\Rightarrow} u\alpha \qquad \text{if } (q, uv, S) \vdash^{*} (q, v, \alpha) \tag{7.25}$$

We prove (7.25) by the number of moves in $(q, uv, S) \vdash^{*} (q, v, \alpha)$.

If $(q, uv, S) \vdash^{0} (q, v, \alpha)$, then $u = \Lambda$, $S = \alpha$; obviously, $S \overset{0}{\Rightarrow} \Lambda\alpha$. Thus there is basis for induction.

Let us assume (7.25) when the number of moves is n. Assume

$$(q, uv, S) \vdash^{n+1} (q, v, \alpha) \tag{7.26}$$

The last move in (7.26) is obtained either from $(q, \Lambda, A) \vdash (q, \Lambda, \alpha')$ or from $(q, a, a) \vdash (q, \Lambda, \Lambda)$. In the first case, (7.26) can be split as

$$(q, uv, S) \vdash^{n} (q, v, A\alpha_2) \vdash (q, v, \alpha_1\alpha_2) = (q, v, \alpha)$$

By induction hypothesis, $S \overset{*}{\Rightarrow} uA\alpha_2$, and the last move is induced by $A \rightarrow \alpha_1$. Thus, $S \overset{*}{\Rightarrow} uA\alpha_2$ implies $\alpha_1\alpha_2 = \alpha$. So,

$$S \overset{*}{\Rightarrow} uA\alpha_2 \Rightarrow u\alpha_1\alpha_2 = u\alpha$$

In the second case, (7.26) can be split as

$$(q, uv, S) \vdash^{n} (q, av, a\alpha) \vdash (q, v, \alpha)$$

Also, $u = u'a$ for some $u' \in \Sigma$. So, $(q, u'av, S) \vdash^{n} (q, av, a\alpha)$ implies (by induction hypothesis) $S \overset{*}{\Rightarrow} u'a\alpha = u\alpha$. Thus in both the cases we have shown that $S \overset{*}{\Rightarrow} u\alpha$. By the principle of induction, (7.25) is true.

Now, we can prove that if $w \in N(A)$ then $w \in L(G)$. As $w \in N(A)$, we have $(q, w, S) \overset{*}{\vdash} (q, \Lambda, \Lambda)$. By taking $u = w$, $v = \Lambda$, $\alpha = \Lambda$ and applying (7.25), we get $S \overset{*}{\Rightarrow} w\Lambda = w$, i.e. $w \in L(G)$. Thus,

$$L(G) = N(A)$$

Theorem 7.4 If $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a pda, then there exists a context-free grammar G such that $L(G) = N(A)$.

Proof We first give the construction of G and then prove that $N(A) = L(G)$.

Step 1 (Construction of G). We define $G = (V_N, \Sigma, P, S)$, where

$$V_N = \{S\} \cup \{[q, Z, q'] \mid q, q' \in Q, Z \in \Gamma\}$$

i.e. any element of V_N is either the new symbol S acting as the start symbol for G or an ordered triple whose first and third elements are states and the second element is a pushdown symbol.

The productions in P are induced by moves of pda as follows:

R_1: S-productions are given by $S \to [q_0, Z_0, q]$ for every q in Q.

R_2: Each move erasing a pushdown symbol given by $(q', \Lambda) \in \delta(q, a, Z)$ induces the production $[q, Z, q'] \to a$.

R_3: Each move not erasing a pushdown symbol given by $(q_1, Z_1Z_2 \ldots Z_m) \in \delta(q, a, Z)$ induces many productions of the form

$$[q, Z, q'] \to a[q_1, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q']$$

where each of the states $q', q_2, \ldots, q_m$ can be any state in Q. Each move yields many productions because of R_3. We apply this construction to an example before proving that $L(G) = N(A)$.

EXAMPLE 7.8

Construct a context-free grammar G which accepts $N(A)$, where

$$A = (\{q_0, q_1\}, \{a, b\}, \{Z_0, Z\}, \delta, q_0, Z_0, \emptyset)$$

and δ is given by

$$\begin{aligned}
\delta(q_0, b, Z_0) &= \{(q_0, ZZ_0)\} \\
\delta(q_0, \Lambda, Z_0) &= \{(q_0, \Lambda)\} \\
\delta(q_0, b, Z) &= \{(q_0, ZZ)\} \\
\delta(q_0, a, Z) &= \{(q_1, Z)\} \\
\delta(q_1, b, Z) &= \{(q_1, \Lambda)\} \\
\delta(q_1, a, Z_0) &= \{(q_0, Z_0)\}
\end{aligned}$$

Solution

Let

$$G = (V_N, \{a, b\}, P, S)$$

where V_N consists of S, $[q_0, Z_0, q_0]$, $[q_0, Z_0, q_1]$, $[q_0, Z, q_0]$, $[q_0, Z, q_1]$, $[q_1, Z_0, q_0]$, $[q_1, Z_0, q_1]$, $[q_1, Z, q_0]$, $[q_1, Z, q_1]$.

The productions are

$$P_1: S \to [q_0, Z_0, q_0]$$

$$P_2: S \to [q_0, Z_0, q_1]$$

$\delta(q_0, b, Z_0) = \{(q_0, ZZ_0)\}$ yields

$$P_3: [q_0, Z_0, q_0] \to b[q_0, Z, q_0][q_0, Z_0, q_0]$$

$$P_4: [q_0, Z_0, q_0] \to b[q_0, Z, q_1][q_1, Z_0, q_0]$$

$$P_5: [q_0, Z_0, q_1] \to b[q_0, Z, q_0][q_0, Z_0, q_1]$$

$$P_6: [q_0, Z_0, q_1] \to b[q_0, Z, q_1][q_1, Z_0, q_1]$$

$\delta(q_0, \Lambda, Z_0) = \{(q_0, \Lambda)\}$ gives

$$P_7: [q_0, Z_0, q_0] \to \Lambda$$

$\delta(q_0, b, Z) = \{(q_0, ZZ)\}$ gives

$$P_8: [q_0, Z, q_0] \to b[q_0, Z, q_0][q_0, Z, q_0]$$

$$P_9: [q_0, Z, q_0] \to b[q_0, Z, q_1][q_1, Z, q_0]$$

$$P_{10}: [q_0, Z, q_1] \to b[q_0, Z, q_0][q_0, Z, q_1]$$

$$P_{11}: [q_0, Z, q_1] \to b[q_0, Z, q_1][q_1, Z, q_1]$$

$\delta(q_0, a, Z) = \{(q_1, Z)\}$ yields

$$P_{12}: [q_0, Z, q_0] \to a[q_1, Z, q_0]$$

$$P_{13}: [q_0, Z, q_1] \to a[q_1, Z, q_1]$$

$\delta(q_1, b, Z) = \{(q_1, \Lambda)\}$ gives

$$P_{14}: [q_1, Z, q_1] \to b$$

$\delta(q_1, a, Z_0) = \{(q_0, Z_0)\}$ gives

$$P_{15}: [q_1, Z_0, q_0] \to a[q_0, Z_0, q_0]$$

$$P_{16}: [q_1, Z_0, q_1] \to a[q_0, Z_0, q_1]$$

P_1–P_{16} give the productions in P.

Using the techniques given in Chapter 6, we can reduce the number of variables and productions.

Step 2 Proof of the construction, i.e. $N(A) = L(G)$.

Before proving that $N(A) = L(G)$, we note that a variable $[q, Z, q']$ indicates that for the pda the current state is q and the topmost symbol in PDS is Z. In the course of a derivation, a state q' is chosen in such a way that the PDS is emptied ultimately. This corresponds to applying R_2. (Note that the production given by R_2 replaces a variable by a terminal.)

To prove $N(A) = L(G)$, we need an auxiliary result, i.e.

$$[q, Z, q'] \overset{*}{\underset{G}{\Rightarrow}} w \tag{7.27}$$

if and only if

$$(q, w, Z) \overset{*}{\vdash} (q', \Lambda, \Lambda) \tag{7.28}$$

We prove the 'if' part by induction on the number of steps in (7.28). If $(q, w, Z) \vdash (q', \Lambda, \Lambda)$, then w is either a in Σ or a in Λ. So, we have

$$(q', \Lambda) \in \delta(q, w, Z)$$

By R_2 we get a production $[q, Z, q'] \to w$. So, $[q, Z, q'] \underset{G}{\Rightarrow} w$. Thus there is basis for induction.

Let us assume the result, namely that (7.28) implies (7.27) when the former has less than k moves. Consider $(q, w, Z) \overset{k}{\vdash} (q', \Lambda, \Lambda)$. This can be split as

$$(q, aw', Z) \vdash (q_1, w', Z_1Z_2 \ldots Z_m) \overset{k-1}{\vdash} (q', \Lambda, \Lambda) \tag{7.29}$$

where $w = aw'$ and $a \in \Sigma$ or $a = \Lambda$, depending on the first move.

Consider the second part of (7.29). This means that the PDS has $Z_1Z_2 \ldots Z_m$, initially, and on application of w, the PDS is emptied. Each move of pda can either erase the topmost symbol on the PDS or replace the topmost symbol by some non-empty string. So several moves may be required for getting on the top of PDS. Let w_1 be the prefix of w such that the PDS has $Z_2Z_3 \ldots Z_m$ after the application of w_1. We can note that the string $Z_2Z_3 \ldots Z_m$ is not disturbed while applying w_1. Lel w_i be the substring of w such that the PDS has $Z_{i+1} \ldots Z_m$ on application of w_i. $Z_{i+1} \ldots Z_m$ is not disturbed while applying $w_1, w_2, \ldots, w_i$. The changes in PDS are illustrated in Fig. 7.3.

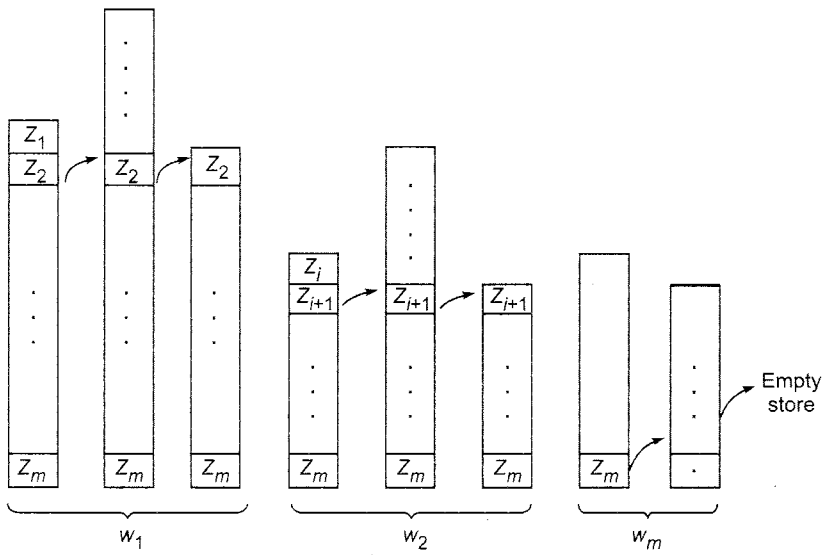


Fig. 7.3 Illustration of changes in pushdown store.

In terms of IDs, we have

$$(q_i, w_i, Z_i) \vdash^* (q_{i+1}, \Lambda, \Lambda) \text{ for } i = 1, 2, \ldots, m, q_{m+1} = q' \quad (7.30)$$

As each move in (7.30) requires less than k steps, by induction hypothesis we have

$$[q_i, Z_i, q_{i+1}] \overset{*}{\underset{G}{\Rightarrow}} w_i \quad \text{for } i = 1, 2, \ldots, m \quad (7.31)$$

The first part of (7.29) is given by $(q_1, Z_1Z_2 \ldots Z_m) \in \delta(q, a, Z)$. By R_3 we get a production

$$[q, Z, q'] \overset{*}{\Rightarrow} a[q_1, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q'] \quad (7.32)$$

From (7.31) and (7.32), we get

$$[q, Z, q'] \overset{*}{\Rightarrow} aw_1w_2 \ldots w_m = w$$

By the principle of induction, (7.28) implies (7.27).

We prove the 'only if' part by induction on the number of steps in the derivation of (7.27). Suppose $[q, Z, q'] \Rightarrow w$. Then $[q, Z, q'] \rightarrow w$ is a production in P. This production is obtained by R_2. So $w = \Lambda$ or $w \in \Sigma$ and $(q', \Lambda) \in \delta(q, w, Z)$. This gives the move $(q, w, Z) \vdash (q, \Lambda, \Lambda)$. Thus there is basis for induction.

Assume the result for derivations where the number of steps is less than k. Consider $[q, Z, q'] \overset{k}{\Rightarrow} w$. This can be split as

$$[q, Z, q'] \Rightarrow a[q_1, Z_1, q_2][q_2, Z_2, q_3] \ldots [q_m, Z_m, q'] \overset{k-1}{\Rightarrow} w \quad (7.33)$$

As G is context-free, we can write $w = aw_1w_2 \ldots w_m$, where

$$[q_i, Z_i, q_{i+1}] \overset{*}{\underset{G'}{\Rightarrow}} w_i \quad \text{and} \quad q_{m+1} = q'$$

By induction hypothesis, we have

$$(q_i, w_i, Z_i) \vdash^* (q_{i+1}, \Lambda, \Lambda)_i \quad \text{for } i = 1, 2, \ldots, m \quad (7.34)$$

By applying Results 1 and 2, we get

$$(q_i, w_i, Z_iZ_{i+1} \ldots Z_m) \vdash^* (q_{i+1}, \Lambda, Z_{i+1} \ldots Z_m)$$

$$(q_i, w_iw_{i+1} \ldots w_m, Z_i \ldots Z_m) \vdash (q_{i+1}, w_{i+1} \ldots w_m, Z_{i+1} \ldots Z_m) \quad (7.35)$$

By combining the moves given by (7.35), we get

$$(q_1, w_1w_2 \ldots w_m, Z_1 \ldots Z_m) \vdash^* (q', \Lambda, \Lambda) \quad (7.36)$$

The first step in (7.33) is induced by $(q_1, Z_1Z_2 \ldots Z_m) \in \delta(q, a, Z)$. The corresponding move is

$$(q, a, Z) \vdash (q_1, \Lambda, Z_1Z_2 \ldots Z_m)$$

By applying the Result 1, we get

$$(q, aw_1 \ldots w_m, Z) \vdash (q_1, w_1 \ldots w_m, Z_1 \ldots Z_m) \quad (7.37)$$

From (7.37) and (7.36), we get $(q, w, Z) \vdash^* (q', \Lambda, \Lambda)$. By the principle of induction, (7.27) implies (7.28).

Thus we have proved the auxiliary result. In particular,

$$[q_0, Z_0, q_1] \stackrel{*}{\Rightarrow} w \quad \text{iff } (q_0, w, Z_0) \vdash^* (q', \Lambda, \Lambda) \tag{7.38}$$

Now $w \in L(G)$

iff $S \stackrel{*}{\Rightarrow} w$
iff $S \Rightarrow [q_0, Z_0, q'] \stackrel{*}{\Rightarrow} w$ (for some q' by R_1)
iff $(q_0, w, Z_0) \vdash^* (q', \Lambda, \Lambda)$ by the auxiliary result
iff $w \in N(A)$

Thus, $N(A) = L(G)$. ∎

Corollary If A is a pda, then there exists a context-free grammar G such that $T(A) = L(G)$.

Proof By Theorem 7.2 we can find a pda A' such that $T(A) = N(A')$. By Theorem 7.4 we can construct G such that $N(A') = L(G)$. Thus $T(A) = L(G)$. ∎

EXAMPLE 7.9

Construct a pda accepting $\{a^n b^m a^n \mid m, n \geq 1\}$ by null store. Construct the corresponding context-free grammar accepting the same set.

Solution

The pda A accepting $\{a^n b^m a^n \mid m, n \geq 1\}$ is defined as follows:

$$A = (\{q_0, q_1\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, \emptyset)$$

where δ is defined by

$$R_1\text{: } \delta(q_0, a, \text{Zo}) = \{(q_0, aZ_0)\}$$
$$R_2\text{: } \delta(q_0, a, a) = \{(q_0, aa)\}$$
$$R_3\text{: } \delta(q_0, b, a) = \{(q_1, a)\}$$
$$R_4\text{: } \delta(q_1, b, a) = \{(q_1, a)\}$$
$$R_5\text{: } \delta(q_1, a, a) = \{(q_1, \Lambda)\}$$
$$R_6\text{: } \delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$$

This is a modification of δ given in Example 7.2.

We start storing a's until a b occurs (Rules R_1 and R_2). When the current input symbol is b, the state changes, but no change in PDS occurs (Rule R_3). Once all the b's in the input string are exhausted (using Rule R_4), the remaining a's are erased (Rule R_5). Using R_6, Z_0 is erased. So,

$$(q_0, a^n b^m a^n, Z_0) \vdash^* (q_1, \Lambda, Z_0) \vdash (q_1, \Lambda, \Lambda)$$

This means that $a^n b^m a^n \in N(A)$. We can show that

$$N(A) = \{a^n b^m a^n \mid m, n \geq 1\}$$

by using Rules R_1–R_6.

Define $G = (V_N, \{a, b\}, P, S)$, where V_N consists of

$$[q_0, Z_0, q_0], [q_1, Z_0, q_0], [q_0, a, q_0], [q_1, a, q_0]$$
$$[q_0, Z_0, q_1], [q_1, Z_0, q_1], [q_0, a, q_1], [q_1, a, q_1]$$

The productions in P are constructed as follows:

The S-productions are

$$P_1: S \to [q_0, Z_0, q_0], \qquad P_2: S \to [q_0, Z_0, q_1]$$

$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$ induces

$$P_3: [q_0, Z_0, q_0] \to a[q_0, a, q_0][q_0, Z_0, q_0]$$
$$P_4: [q_0, Z_0, q_0] \to a[q_0, a, q_1][q_1, Z_0, q_0]$$
$$P_5: [q_0, Z_0, q_1] \to a[q_0, a, q_0][q_0, Z_0, q_1]$$
$$P_6: [q_0, Z_0, q_1] \to a[q_0, a, q_1][q_1, Z_0, q_1]$$

$\delta(q_0, a, a) = \{(q_0, aa)\}$ yields

$$P_7: [q_0, a, q_0] \to a[q_0, a, q_0][q_0, a, q_0]$$
$$P_8: [q_0, a, q_0] \to a[q_0, a, q_1][q_1, a, q_0]$$
$$P_9: [q_0, a, q_1] \to a[q_0, a, q_0][q_0, a, q_1]$$
$$P_{10}: [q_0, a, q_1] \to a[q_0, a, q_1][q_1, a, q_1]$$

$\delta(q_0, b, a) = (q_1, a)\}$ gives

$$P_{11}: [q_0, a, q_0] \to b[q_1, a, q_0]$$
$$P_{12}: [q_0, a, q_1] \to b[q_1, a, q_1]$$

$\delta(q_1, b, a) = \{(q_1, a)\}$ yields

$$P_{13}: [q_1, a, q_0] \to b[q_1, a, q_0]$$
$$P_{14}: [q_1, a, q_1] \to b[q_1, a, q_1]$$

$\delta(q_1, a, a) = \{(q_1, \Lambda)\}$ gives

$$P_{15}: [q_1, a, q_1] \to a$$

$\delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$ yields

$$P_{16}: [q_1, Z_0, q_1] \to \Lambda$$

Note: When the number of states is a large number, it is neither necessary nor advisable to write all the productions. We construct productions involving those variables appearing in some sentential form. Using the constructions in Chapter 6, we can simplify the grammar further.

Theorem 7.5 The intersection of a context-free language L and a regular language R is a context-free language.

Proof Let L be accepted by a pda $A = (Q_A, \Sigma, T, \delta_A, q_0, Z_0, F_A)$ by final state and R by DFA $M = (Q_M, \Sigma, \delta_M, P_0, F_M)$.

We define a pda M' accepting $L \cap R$ by final state in such a way that M' simulates moves of A on input a in Σ and changes the state of M using δ_M. On input Λ, M' simulates A without changing the state of M. Let

$$M' = (Q_M \times Q_A, \Sigma, \Gamma, \delta, [p_0, q_0], Z_0, F_M \times F_A)$$

where δ is defined as follows:

$\delta([p, q], a, X)$ contains $([P', q'], \gamma)$ when $\delta_M(p, a) = p'$ and $\delta_A(q, a, X)$ contains (q', γ). $\delta([p, q], \Lambda, X)$ contains $([P, q'], \gamma)$ when $\delta_A(q, \Lambda, X)$ contains (q', γ).

To prove $T(M') = L \cap R$ we need an auxiliary result, i.e.

$$([p_0, q_0], w, Z_0) \overset{i}{\underset{M'}{\vdash}} ([p, q], \Lambda, \gamma) \tag{7.39}$$

if and only if

$$(q_0, w, Z_0) \overset{i}{\underset{A}{\vdash}} (q, \Lambda, \gamma) \quad \text{and} \quad \delta_M(p_0, w) = p \tag{7.40}$$

We prove the 'only if' part by induction on i (the number of steps). If $i = 0$, the proof is trivial (In this case, $p = p_0$, $q = q_0$, $w = \Lambda$ and $\gamma = Z_0$). Thus there is basis for induction. Let us assume that (7.39) implies (7.40) when the former has $i - 1$ steps.

Let $([p_0, q_0], w'a, Z_0) \overset{i}{\underset{M'}{\vdash}} ([p, q], \Lambda, \gamma)$. This can be split into $([p_0, q_0], w'a, Z_0) \overset{i-1}{\underset{M'}{\vdash}} ([p', q'], a, \beta) \underset{M'}{\vdash} ([p, q], \Lambda, \gamma)$, where $w = w'a$ and a is in Σ or $a = \Lambda$ depending on the last move. By induction hypothesis, we have $(q_0, w', Z_0) \overset{i-1}{\underset{A}{\vdash}} (q', \Lambda, \beta)$ and $\delta_M(p', w') = p'$. By definition of δ, $([p', q'], a, \beta) \underset{M'}{\vdash} ([p, q], \Lambda, \gamma)$ implies $(q', a, \beta) \underset{M'}{\vdash} (q, \Lambda, \gamma)$ and $\delta_M(p', a) = p$. (***Note:*** $p' = p$ when $a = \Lambda$.) So, $\delta_M(p_0, w'a) = \delta_M(p', a) = p$. By combining the moves of A, we get $(q_0, w'a, Z_0) \overset{i-1}{\underset{A}{\vdash}} (q', a, \beta) \underset{A}{\vdash} (q, A, \gamma)$, i.e. $(q_0, w, Z_0) \overset{i}{\underset{A}{\vdash}} (q, \Lambda, \gamma)$. So the result is true for i steps.

By the principle of induction the 'only if' part is proved.

We prove the 'if' part also by induction on i. It is trivial to see that there is basis for induction.

Let us assume (7.40) with $i - 1$ steps. Assume that $(q_0, w, Z_0) \overset{i}{\underset{A}{\vdash}} (q, A, \gamma)$ and $\delta_M(p_0, w') = p$. Writing w as $w'a$ and taking $\delta_M(p_0, w')$ as p', we get $(q_0, w'a, Z_0) \overset{i-1}{\underset{A}{\vdash}} (q', \alpha, \beta) \underset{A}{\vdash} (q, \Lambda, \gamma)$. So, $(q_0, w', Z_0) \overset{i-1}{\underset{A}{\vdash}} (q', \Lambda, \beta)$.

By induction hypothesis, we get $([p_0, q_0], w', Z_0) \overset{i-1}{\underset{M}{\vdash}} ([p', q'], \Lambda, \beta)$. Also, $\delta_M(p', a) = p$ and $(q', a, \beta) \underset{A}{\vdash} (q, \Lambda, \gamma)$ implies $([p', q'], \alpha, \beta) \underset{M}{\vdash} ([p, q], \Lambda, \gamma)$.

Combining the moves, we get $([p_0, q_0], w, Z_0) \overset{i}{\underset{M}{\vdash}} ([p, q], \Lambda, \gamma)$.

Thus the result is true for i steps. By the principle of induction, the 'if' part is proved.

Note: In Chapter 8 we will prove that the intersection of two context-free languages need not be context-free (Property 3, Section 8.3).

7.4 PARSING AND PUSHDOWN AUTOMATA

In a natural language, parsing is the process of splitting a sentence into words. There are two types of parsing, namely the top-down parsing and the bottom-up parsing. Suppose we want to parse the sentence "Ram ate a mango." If NP, VP, N, V, ART denote noun predicate, verb predicate, noun, verb and article, then the top-down parsing can be done as follows:

$S \rightarrow$ NPVP
$\rightarrow$ Name VP
$\rightarrow$ Ram V NP
$\rightarrow$ Ram ate ART N
$\rightarrow$ Ram ate a N
$\rightarrow$ Ram ate a mango

The bottom-up parsing for the same sentence is

Ram ate a mango $\rightarrow$ Name ate a mango
$\rightarrow$ Name verb a mango
$\rightarrow$ Name V ART N
$\rightarrow$ NP VN P
$\rightarrow$ NP VP
$\rightarrow S$

In the case of formal languages, we derive a terminal string in $L(G)$ by applying the productions of G. If we know that $w \in \Sigma^*$ in $L(G)$, then $S \overset{*}{\Rightarrow} w$. The process of the reconstruction of the derivation of w is called parsing. Parsing is possible in the case of some context-free languages.

Parsing becomes important in the case of programming languages. If a statement in a programming language is given, only the derivation of the statement can give the meaning of the statement. (This is termed *semantics*.)

As mentioned earlier, there are two types of parsing: top-down parsing and bottom-up parsing.

In top-down parsing, we attempt to construct the derivation (or the corresponding parse tree) of the input string, starting from the root (with label S) and ending in the given input string. This is equivalent to finding a leftmost derivation. On the other hand, in bottom-up parsing we build the derivation from the given input string to the top (root with label S).

7.4.1 TOP-DOWN PARSING

In this section we present certain techniques for top-down parsing which can be applied to a certain subclass of context-free languages. We illustrate them by means of some examples. We discuss LL(1) parsing, LL(k) parsing, left factoring and the technique to remove left recursion.

EXAMPLE 7.10

Let $G = (\{S, A, B\}, \{a, b\}, P, S)$ where P consists of $S \rightarrow aAB$, $S \rightarrow bBA$, $A \rightarrow bS$, $A \rightarrow a$, $B \rightarrow as$, $B \rightarrow b$. $w = abbbab$ is in $L(G)$. Let us try to get a leftmost derivation of w. When we start with S we have two choices: $S \rightarrow aAB$ and $S \rightarrow bBA$. By looking at the first symbol of w, we see that $S \rightarrow bBA$ will not yield w. So we choose $S \rightarrow aAB$ as the production to be applied in step 1 and we get $S \Rightarrow aAB$. Now consider the leftmost variable A in the sentential form aAB. We have to apply an A-production among the productions $A \rightarrow bS$ and $A \rightarrow a$. $A \rightarrow a$ will not yield w subsequently since the second symbol in w is b. So, we choose $A \rightarrow bS$ and get $S \Rightarrow aAB \Rightarrow abSB$. Also, the substring ab of w is a substring of the sentential form $abSB$. By looking ahead for one symbol, namely the symbol b, we decide to apply $S \rightarrow bBA$ in the third step. This leads to $S \Rightarrow aAB \Rightarrow abSB \Rightarrow abbBAB$. The leftmost variable in the sentential form $abbBAB$ is B. By looking ahead for one symbol which is b, we apply the B-production $B \rightarrow b$ in the fourth step. On similar considerations, we apply $A \rightarrow a$ and $B \rightarrow b$ in the last two steps to get the leftmost derivation.

$$S \Rightarrow aAB \Rightarrow abSB \Rightarrow abbBAB \Rightarrow abbbAB \Rightarrow abbbaB \Rightarrow abbbab$$

Thus in the case of the given grammar, we are able to construct a leftmost derivation of w by looking ahead for one symbol in the input string. In order to do top-down parsing for a general string in $L(G)$, we prepare a table called the *parsing table*. The table provides the production to be applied for a given variable with a particular look ahead for one symbol.

For convenience, we denote the productions $S \rightarrow aAB$, $S \rightarrow bBA$, $A \rightarrow bS$, $A \rightarrow a$, $B \rightarrow aS$ and $B \rightarrow b$ by $P_1, P_2, \ldots, P_6$. Let E denote an error. It indicates that the given input string is not in $L(G)$. The table for the given grammar is given in Table 7.1.

TABLE 7.1 Parsing Table for Example 7.10

	Λ	a	b
S	E	P_1	P_2
A	E	P_4	P_3
B	E	P_5	P_6

For example, if A is the leftmost variable in a sentential form and the first symbol in unprocessed substring of the given input string is b, then we have to apply P_3.

A grammar possessing this property (by looking ahead for one symbol in the input string we can decide the production to be applied in the next step) is called an LL(1) grammar.

EXAMPLE 7.11

Let G be a context-free grammar having the productions $S \to F + S$, $S \to F * S$, $S \to F$ and $F \to a$. Consider $w = a + a * a$. This is a string in $L(G)$. Let us try to get the top-down parsing for w.

Looking ahead for one symbol will not help us. For the string $a + a * a$, we can apply $F \to a$ on seeing a. But if a is followed by + or *, we cannot apply a. So in this case it is necessary to look ahead for two symbols.

When we start with S we have three productions $S \to F + S$, $S \to F * S$ and $S \to F$. The first two symbols in $a + a * a$ are $a +$. This forces us to apply only $S \to F + S$ and not other S-productions. So, $S \to F + S$. We can apply $F \to a$ now to get $S \Rightarrow F + S \Rightarrow a + S$. Now the remaining part of w is $a * a$. The first two symbols $a *$ suggest that we apply $S \to F * S$ in the third step. So, $S \overset{*}{\Rightarrow} a + S \Rightarrow a + F * S$. As the third symbol in w is a, we apply $F \to a$, yielding $S \overset{*}{\Rightarrow} a + F * S \Rightarrow a + a * S$. The remaining part of the input string w is a. So, we have to apply $S \to F$ and $F \to a$. Thus the leftmost derivation of $a + a * a$ is $S \Rightarrow F + S \Rightarrow a + S \Rightarrow a + F * S \Rightarrow a + a * S \Rightarrow a + a * F \Rightarrow a + a * a$.

As in Example 7.10, we can prepare a table (Table 7.2) which enables us to get a leftmost derivation for any input string. P_1, P_2, P_3 and P_4 denote the productions $S \to F + S$, $S \to F * S$, $S \to F$ and $F \to a$. E denotes an error.

TABLE 7.2 Parsing Table for Example 7.11

	Λ	*a*	+	*	*aa*	*a*+	*a**
S	*E*	P_3	*E*	*E*	*E*	P_1	P_2
F	*E*	P_4	*E*	*E*	*E*	P_4	P_4
	+*a*	++	+*	**a*	*+	**	
S	*E*	*E*	*E*	*E*	*E*	*E*	
F	*E*	*E*	*E*	*E*	*E*	*E*	

For example, if the leftmost variable in a sentential form is F and the next two symbols to be processed are $a *$, then we apply P_4, i.e. $F \to a$. When we encounter $* a$ as the next two symbols, an error is indicated in the table and so the input string is not in $L(G)$.

A grammar G having the property (by looking ahead for k symbols we derive a given input string in $L(G)$), is called an LL(k) grammar. The grammar given in Example 7.11 is an LL(2) grammar.

In Examples 7.10 and 7.11 for getting a leftmost derivation, one production among several choices was obtained by look ahead for k symbols. This kind of nondeterminism cannot be resolved in some grammars even by looking ahead.

This is the case when a grammar has two A-productions of the form $A \to \alpha\beta$ and $A \to \alpha\gamma$. By a technique called 'left factoring', we resolve this nondeterminism. Another troublesome phenomenon in a context-free grammar which creates a problem is called left recursion. A variable A is called left recursive if there is an A-production of the form $A \to A\alpha$. Such a production can cause a top-down parser into an infinite loop. Left factoring and technique for avoiding left recursion are provided in Theorems 7.6 and 7.7.

Theorem 7.6 Let G be a context-free grammar having two A-productions of the form $A \to \alpha\beta$ and $A \to \alpha\gamma$. If $A \to \alpha\beta$ and $A \to \alpha\gamma$ are replaced by $A \to \alpha A'$, $A' \to \beta$ and $A' \to \gamma$, where A' is a new variable then the resulting grammar is equivalent to G.

Proof The equivalence can be proved by showing that the effect of applying $A \to a\beta$ and $A \to \alpha\gamma$ in a derivation can be realised by applying $A \to \alpha A'$, $A' \to \beta$ and $A' \to \gamma$ and vice versa.

Note: The technique of avoiding nondeterminism using Theorem 7.6 is called left factoring.

Theorem 7.7 Let G be a context-free grammar. Let the set of all A-productions be $\{A \to A\alpha_1, \ldots, A \to A\alpha_n, A \to \beta_1, \ldots, A \to \beta_m\}$. Then the grammar G' obtained by introducing a new variable A' and replacing all A-productions in G by $A \to \beta_1 A', \ldots, A \to \beta_m A', A' \to \alpha_1 A', \ldots, A' \to \alpha_n A'$ and $A' \to \Lambda$ is equivalent to G.

Proof Similar to proof of Lemma 6.3.

Theorems 7.6 and 7.7 are useful to construct a top-down parser only for certain context-free grammars and not for all context-free grammars. We summarize our discussion as follows:

Construction of Top-Down Parser

Step 1 Eliminate left recursion in G by repeatedly applying Theorem 7.7 to all left recursive variables.

Step 2 Apply Theorem 7.6 to get left factoring wherever necessary.

Step 3 If the resulting grammar is LL(k) for some natural number k, apply top-down parsing using the techniques explained in Examples 7.10 and 7.11.

EXAMPLE 7.12

Consider the language consisting of all arithmetic expressions involving +, *, (and) over the variables $x1$ and $x2$. This language is generated by a grammar

$G = (\{T, F, E\}\ \Sigma, P, E)$, where $\Sigma = \{x, 1, 2, +, *, (,)\}$ and P consists of

$$E \to E + T \qquad F \to (E)$$
$$E \to T \qquad F \to x1$$
$$T \to T * F \qquad F \to x2$$
$$T \to F$$

Let us construct a top-down parser for $L(G)$.

Step 1 We eliminate left recursion by applying Theorem 7.7 to the left recursive variables E and T. We replace $E \to E + T$ and $E \to T$ by $E \to TE'$, $E' \to + TE'$ and $E' \to \Lambda$ (E' is a new variable). Similarly, $T \to T * F$ and $T \to F$ are replaced by $T \to FT'$, $T' \to * FT'$ and $T' \to \Lambda$. The resulting equivalent grammar is

$G_1 = (\{T, F, E, T', E'\}, \Sigma, P', E)$, where P' consists of

$$E \to TE' \qquad T \to \Lambda$$
$$E' \to +TE' \qquad F \to (E)$$
$$E' \to \Lambda \qquad F \to x1$$
$$T \to FT' \qquad F \to x2$$
$$T' \to * FT'$$

Step 2 We apply Theorem 7.6 for left factoring to $F \to x1$ and $F \to x2$ to get new productions $F \to xN \to N \to 1$ and $N \to 2$.

The resulting equivalent grammar is

$G_2 = (\{T, F, E, T', E'\}, \Sigma, P'', E)$ where P'' consists of

$$P_1: E \to TE' \qquad P_6: T' \to \Lambda$$
$$P_2: E' \to +TE' \qquad P_7: F \to (E)$$
$$P_3: E' \to \Lambda \qquad P_8: F \to xN$$
$$P_4: T \to FT' \qquad P_9: N \to 1$$
$$P_5: T'' \to * FT' \qquad P_{10}: N \to 2$$

Step 3 The grammar G_2 obtained in step 2 is an LL(1) grammar. The parsing table is given in Table 7.3.

TABLE 7.3 Parsing Table for Example 7.12

	Λ	x	1	2	+	*	(	)
E	E	P_1	E	E	E	E	P_1	E
T	E	P_4	E	E	E	E	P_4	E
F	E	P_8	E	E	E	E	P_7	E
T'	P_6	E	E	E	E	P_5	E	P_6
E'	P_3	E	E	E	P_2	E	E	P_3
N	E	E	P_9	P_{10}	E	E	E	E

7.4.2 Top-down Parsing Using Deterministic pda's

We have seen that pda's are the accepting devices for context-free languages. Theorem 7.3 gives us a method of constructing a pda accepting a given context-free language by empty store. In certain cases the construction can be modified in such a way that a leftmost derivation of a given input string can be obtained while testing to know whether the given string is accepted by the pda. This is the case when the given grammar is LL(1). We illustrate this by constructing a (deterministic) pda accepting the language given in Example 7.10 and a leftmost derivation of a given input string using the pda.

EXAMPLE 7.13

For the grammar given in Example 7.10, construct a deterministic pda accepting $L(G)$ and a leftmost derivaiton of *abbab*.

Solution

We construct a pda accepting $L(G)\$$ ($ is a symbol indicating the end of the input string). This is done by using Theorem 7.3. The transitions are

$$\delta(q, \Lambda, A) = \{(q, \alpha) \mid A \to \alpha \text{ is in } P\}$$

$$\delta(q, t, t) = \{(q, \Lambda)\} \quad \text{for every } t \text{ in } \Sigma$$

This pda is not deterministic as we have two S-productions, two A-productions, etc. In Example 7.10 we resolved the nondeterminism by looking ahead for one more symbol in the input string to be processed. In the construction of pda this can be achieved by changing the state from q to q_a on reading a. When the pda is in state q_a and the current symbol is S we choose the transition resulting in (q, aAB). Now the deterministic pda accepting $L(G)\$$ by null store is

$$A = (\{p, q, q_a, q_b\}, \{a, b, \$\}, \{S, A, B, a, b, Z_0\}, \delta, p, Z_0, \emptyset)$$

where δ is defined by the following rules:

$$R_1 : \delta(p, \Lambda, Z_0) = (q, s)$$

$$R_2 : \delta(q, a, \Lambda) = (q_a, \Lambda)$$

$$R_3 : \delta(q_a, \Lambda, a) = (q, e)$$

$$R_4 : \delta(q, b, \Lambda) = (q_b, \Lambda)$$

$$R_5 : \delta(q_a, \Lambda, b) = (q, e)$$

$$R_6 : \delta(q_a, \Lambda, S) = (q_a, aAB)$$

$$R_7 : \delta(q_b, \Lambda, S) = (q_b, bBA)$$

$$R_8 : \delta(q_a, \Lambda, A) = (q_a, a)$$

$$R_9 : \delta(q_b, \Lambda, A) = (q_b, bS)$$

R_{10} : $\delta(q_a, \Lambda, B) = (q_a, aS)$

R_{11} : $\delta(q_b, \Lambda, B) = (q_b, b)$

R_{12} : $\delta(q, \$, Z_0) = (q, \Lambda)$

Here R_1 changes the initial ID (p, w, Z) into (q, w, SZ). R_2 and R_4 are for remembering the next symbol. R_6–R_{11} are simulating the productions. R_3 and R_5 are for matching the current input symbol and the topmost symbol on PDS and for erasing it (in PDS). Finally, R_{12} is a move for erasing Z and making the PDS empty when the last symbol \$ of the input string is read.

To get a leftmost derivation for an input string w, apply the unique transition given by R_1 to R_{12}. When we apply R_6 to R_{11}, we are using a corresponding production. By recording these productions we can test whether $w \in L(G)$ and get a leftmost derivation. The parsing for the input string *abbbab* is given m Table 7.4.

The last column of Table 7.4 gives us a leftmost derivation of *abbbab*. It is $S \Rightarrow aAB \Rightarrow abSB \Rightarrow abbBAB \Rightarrow abbbAB \Rightarrow abbbaB \Rightarrow abbbab$.

TABLE 7.4 Top-down Parsing for w of Example 7.13

Step	*State*	*Unread input*	*Pushdown stack*	*Transition used*	*Production applied*
1	p	*abbbab*\$	Z_0	—	—
2	q	*abbbab*\$	SZ_0	R_1	
3	q_a	*bbbab*\$	SZ_0	R_2	
4	q_a	*bbbab*\$	$aABZ_0$	R_6	$S \rightarrow aAB$
5	q	*bbbab*\$	ABZ_0	R_3	
6	q_b	*bbab*\$	ABZ_0	R_4	
7	q_b	*bbab*\$	$bSBZ_0$	R_9	$A \rightarrow bS$
8	q	*bbab*\$	SBZ_0	R_5	
9	q_b	*bab*\$	SBZ_0	R_4	
10	q_b	*bab*\$	$bBABZ_0$	R_7	$S \rightarrow bBA$
11	q	*bab*\$	$BABZ_0$	R_5	
12	q_b	*ab*\$	$BABZ_0$	R_4	
13	q_b	*ab*\$	$bABZ_0$	R_{11}	$B \rightarrow b$
14	q	*ab*\$	ABZ_0	R_5	
15	q_a	*b*\$	ABZ_0	R_2	
16	q_a	*b*\$	aBZ_0	R_8	$A \rightarrow a$
17	q	*b*\$	BZ_0	R_3	
18	q_b	\$	BZ_0	R_4	
19	q_b	\$	bZ_0	R_{11}	$B \rightarrow b$
20	q	\$	Z_0	R_5	
21	q	Λ	Λ	R_{12}	

7.4.3 BOTTOM-UP PARSING

In bottom-up parsing we build the derivation tree from the given input string to the top (the root with label S). For certain classes of grammars, called weak precedence grammars, we can construct a deterministic pda which acts as a bottom-up parser. We illustrate the method by constructing the parser for the grammar given in Example 7.12.

In bottom-up parsing we have to reverse the productions to get S finally. This suggests the following moves for a pda acting as bottom-up parser.

(i) $\delta(p, \Lambda, \alpha^T) = \{(P, A) | \text{there exists a production } A \to \alpha\}$
(ii) $\delta(p, \sigma, \Lambda) = \{(p, \sigma)\}$ for all σ in Σ.

Using (i) we replace α^T on the basis by A when $A \to \alpha$ is a production. The input symbol σ is moved onto the stack using (ii). For acceptability, we require some moves when the PDS has S or Z_0 on the top.

As in top-down parsing we construct the pda accepting $L(G)\$$. Here we will have two types of operations, namely shifting and reducing. By shifting we mean pushing the input symbol onto the stack (moves given by (ii)). By reducing we mean replacing α^T by A when $A \to \alpha$ is a production in G (moves given by (i)).

At every step we have (i) to decide whether to shift or to reduce (ii) to choose the prefix of the string on PDS for reducing, once we have decided to reduce. For (i) we use a relation P called a precedence relation. If $(a, b) \in P$ where a is the topmost symbol on PDS and b is the input symbol then we reduce. Otherwise we shift b onto the stack. Regarding (ii), we choose the longest prefix of the string on the PDS of the form α^T to be reduced to A (when $A \to \alpha$ is a production).

We illustrate the method using the grammar given in Example 7.12.

EXAMPLE 7.14

Construct a bottom-up parser for the language $L(G)\$$. where G is the grammar given in Example 7.12.

Here the productions are $E \to E + T$, $E \to T$, $T \to T * F$, $T \to F$, $F \to (E)$, $F \to x1$ and $F \to x2$. Using these productions, we can construct the precedence relation P. It is given in Table 7.5. If (a, b) is in P, then we have a tick mark '$\checkmark$' in the (a, b) cell of the table. Using Table 7.5 we can decide the moves. For example, if the stack symbol is F and the next input symbol is $*$, then we apply reduction. If the stack symbol is E, then any input symbol is pushed onto the stack.

TABLE 7.5 The Precedence Relation for Example 7.14.

Stack symbol/ Input symbol	x	(	)	1	2	+	*	$
Z_0								
x								
(								
)			✓			✓	✓	✓
1			✓			✓	✓	✓
2			✓			✓	✓	✓
+								
*								
E								
T			✓			✓		✓
F			✓			✓	✓	✓

Using the precedence relation and the moves given at the beginning of this section we can construct a deterministic pda. As in the construction of top-down parser, when we look ahead for one symbol we 'remember' it by changing state.

The deterministic pda which acts as a bottom-up parser is

$$A = (Q, \Sigma', \Gamma, \delta, p, Z_0, \emptyset)$$

where

$$\Sigma' = \Sigma \cup \{\$\}, \quad Q = \{p\} \cup \{p_\sigma : \sigma \in \Sigma'\}$$

$$\Gamma = \{E, T, F\} \cup \Sigma \cup \{Z_0, \$\}$$

and δ is given by the following rules:

R_1 : $\delta(p, \sigma, \Lambda) = (p, \Lambda)$

R_2 : $\delta(p_\sigma, \Lambda, a) = (p, \sigma a)$ for all $(a, \sigma) \notin P$

R_3 : $\delta(p_\sigma, \Lambda, T + E) = (p_0, E)$ when $(T, \sigma) \in P$

R_4 : $\delta(p_\sigma, \Lambda, Ta) = (p_\sigma, Ea)$ when $(T, \sigma) \in P$ and $a \in \Gamma - \{+\}$

R_5 : $\delta(p_\sigma, \Lambda, F * T) = (p_\sigma, T)$ when $(F, \sigma) \in P$

R_6 : $\delta(p_\sigma, \Lambda, Fa) = (p_\sigma, Ta)$ when $(F, \sigma) \in P$ and $a \in \Gamma - \{*\}$

R_7 : $\delta(p_\sigma, \Lambda, EC) = (p_\sigma, F)$ when $(\ ), \sigma) \in P$

R_8 : $\delta(p_\sigma, \Lambda, 1x) = (p_\sigma, F)$ when $(1, \sigma) \in P$

R_9 : $\delta(p_\sigma, \Lambda, 2x) = (p_\sigma, F)$ when $(2, \sigma) \in P$

R_{10} : $\delta(p_S, \Lambda, E) = (p_S, \Lambda)$

R_{11} : $\delta(p_S, \Lambda, Z_0) = (p_S, \Lambda)$

As A is deterministic, we have dropped parentheses on the R.H.S. of $R_1 - R_{11}$. Using the pda A, we can get a bottom-up parsing for any input string w. The bottom-up parsing for $x1 + (x2)$ is given in Table 7.6.

Table 7.6 Bottom-up Parsing for $x1 + (x2)$

Step	State	Unread input	Pushdown stack	Rule used	Production applied
1	p	$x1 + (x2)$	Z_0	—	
2	p_x	$1 + (x2)$	Z_0	R_1	
3	p	$1 + (x2)$	xZ_0	R_2	
4	p_1	$+ (x2)$	xZ_0	R_1	
5	p	$+ (x2)$	$1xZ_0$	R_2	
6	p_+	$+ (x2)$	$1xZ_0$	R_1	
7	p_+	$+ (x2)$	FZ_0	R_8	$F \to x1$
8	p_+	$+ (x2)$	TZ_0	R_6	$T \to F$
9	p_+	$+ (x2)$	EZ_0	R_4	$E \to T$
10	p	$(x2)$	$+EZ_0$	R_2	
11	$p_($	$x2)$	$+EZ_0$	R_1	
12	p	$x2)$	$(+EZ_0$	R_2	
13	p_x	$2)$	$(+EZ_0$	R_1	
14	p	$2)$	$x(+EZ_0$	R_2	
15	p_2	$)$	$x(+EZ_0$	R_1	
16	p	$)$	$2x(+EZ_0$	R_2	
17	$p_)$	$	$2x(+EZ_0$	R_1	
18	$p_)$	$	$F(+EZ_0$	R_9	$F \to x2$
19	$p_)$	$	$T(+EZ_0$	R_6	$T \to F$
20	$p_)$	$	$E(+EZ_0$	R_4	$E \to T$
21	p	$	$)E(+EZ_0$	R_2	
22	p_S	Λ	$)E(+EZ_0$	R_1	
23	p_S	Λ	$F+EZ_0$	R_7	$F \to (E)$
24	p_S	Λ	$T+EZ_0$	R_6	$T \to F$
25	p_S	Λ	EZ_0	R_3	$E \to E + T$
26	p_S	Λ	Z_0	R_{10}	
27	p_S	Λ	Λ	R_{11}	

By backtracking the productions that we applied, we get a rightmost derivation $E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + (E) \Rightarrow E + (T) \Rightarrow E + (F) \Rightarrow E + (x2) \Rightarrow T + (x2) \Rightarrow F + (x2) \Rightarrow x1 + (x2)$.

In Chapter 8 we will discuss how LR(k) grammars are amenable for parsing.

7.5 SUPPLEMENTARY EXAMPLES

EXAMPLE 7.15

Construct a pda accepting all palindromes over $\{a, b\}$.

Solution

Let $L = \{w \in \{a, b\}^* \mid w = w^T\}$. Before constructing the required pda, note that L consists of palindromes of odd or even length. If w in L is of odd

length, there is a middle symbol which need not be compared with any other symbol. If w in L is of even length, the rightmost symbol of the first half is the same as the leftmost symbol of the second half. The key idea of the construction is to store the symbols in the first half (or the symbols lying to the left of the middle of the input string) and matching them with the symbols in the second half (or the symbols to right of the middle of the input string). If there is no matching, the machine halts.

We define the pda M as follows:

$$M = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

where δ is defined by

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0), (q_1, Z_0)\} \tag{7.41}$$

$$\delta(q_0, b, Z_0) = \{(q_0, bZ_0), (q_1, Z_0)\} \tag{7.42}$$

$$\delta(q_0, a, a) = \{(q_0, aa), (q_1, a)\} \tag{7.43}$$

$$\delta(q_0, b, a) = \{(q_0, ba), (q_1, a)\} \tag{7.44}$$

$$\delta(q_0, a, b) = \{(q_0, ab), (q_1, b)\} \tag{7.45}$$

$$\delta(q_0, b, b) = \{(q_0, bb), (q_1, b)\} \tag{7.46}$$

$$\delta(q_0, \Lambda, Z_0) = \{(q_1, Z_0)\} \tag{7.47}$$

$$\delta(q_0, \Lambda, a) = \{(q_1, a)\} \tag{7.48}$$

$$\delta(q_0, \Lambda, b) = \{(q_1, b)\} \tag{7.49}$$

$$\delta(q_1, a, a) = \{(q_1, \Lambda)\} \tag{7.50}$$

$$\delta(q_1, b, b) = \{(q_1, \Lambda)\} \tag{7.51}$$

$$\delta(q_1, \Lambda, Z_0) = \{(q_2, Z_0)\} \tag{7.52}$$

Obviously, M is a nondeterministic pda. (7.41)–(7.46) give us two choices. The first choice can be used for storing the input symbol without changing the state. (7.47)–(7.49) are used for indicating that the first half of the input string is over; there is change of state in this case. The second choice of (7.41)–(7.46) is used when w is a palindrome of odd length and the middle symbol of w is reached; in this case, we ignore the middle symbol and make no change in PDS. (7.50) and (7.51) are used to match symbols of the input string (second half) and cancel the symbol in PDS when they match. (7.52) is used to move to the final state q_2 after reaching the bottom of PDS.

We can prove that L is accepted by M by final state. The reader can take an odd palindrome and an even palindrome and check that the final state q_2 is reached.

EXAMPLE 7.16

Construct a deterministic pda accepting $L = \{w \in \{a, b\}^* \mid$ the number of a's in w equals the number of b's in $w\}$ by final state.

Solution

We define a pda M as follows:

$$M = (\{q_0, q_1\}, \{a, b\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_1\})$$

where δ is defined by

$$\delta(q_0, a, Z_0) = \{(q_1, Z_0)\} \tag{7.53}$$
$$\delta(q_0, b, Z_0) = \{(q_0, bZ_0)\} \tag{7.54}$$
$$\delta(q_0, a, b) = \{(q_0, \Lambda)\} \tag{7.55}$$
$$\delta(q_0, b, b) = \{(q_0, bb)\} \tag{7.56}$$
$$\delta(q_1, a, Z_0) = \{(q_1, aZ_0)\} \tag{7.57}$$
$$\delta(q_1, b, Z_0) = \{(q_0, aZ_0)\} \tag{7.58}$$
$$\delta(q_1, a, a) = \{(q_1, aa)\} \tag{7.59}$$
$$\delta(q_1, b, a) = \{(q_1, \Lambda)\} \tag{7.60}$$

The construction can be explained as follows:

If the pda M is in the final state q_1, it means it has seen more a's than b's. On seeing the first a, M changes state (from q_0 to q_1) ((7.53)). Afterwards it stores the a's in PDS without changing state ((7.57) and (7.59)). It stores the initial b in PDS ((7.54)) and also the subsequent b's ((7.56)). The pda cancels a in the input string, with the first (topmost) b in PDS ((7.55)). If all b's are matched with stored a's, and M sees the bottom of PDS, M moves from q_1 to q_0 ((7.58)). The b's in the input string are cancelled on seeing a in the PDS (7.60)).

M is deterministic since δ is not defined for input Λ. The reader is advised to check that q_1 is reached on seeing an input string w in L.

EXAMPLE 7.17

Construct a pda M accepting $L = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$ by final state.

Solution

We define pda M as follows:

$$M = (\{q_0, q_1, \ldots, q_6\}, \{a, b, c\}, \{Z_0, X\}, \delta, q_0, Z_0, \{q_1, q_3\})$$

where δ is given by

$$\delta(q_0, \Lambda, Z_0) = \{(q_1, Z_0), (q_2, Z_0), (q_3, Z_0)\} \tag{7.61}$$
$$\delta(q_1, c, Z_0) = \{(q_1, Z_0)\} \tag{7.62}$$
$$\delta(q_2, a, Z_0) = \{(q_2, XZ_0)\} \tag{7.63}$$
$$\delta(q_2, a, X) = \{(q_2, XX)\} \tag{7.64}$$
$$\delta(q_2, b, X) = (q_4, b, X) = \{(q_4, \Lambda)\} \tag{7.65}$$
$$\delta(q_4, \Lambda, Z_0) = \{(q_1, Z_0)\} \tag{7.66}$$

$$\delta(q_3, a, Z_0) = \{(q_3, Z_0)\} \tag{7.67}$$

$$\delta(q_3, b, Z_0) = \{(q_5, XZ_0)\} \tag{7.68}$$

$$\delta(q_5, b, X) = \{(q_5, XX)\} \tag{7.69}$$

$$\delta(q_5, c, X) = \delta(q_6, c, X) = \{(q_6, \Lambda)\} \tag{7.70}$$

$$\delta(q_6, \Lambda, Z_0) = \{(q_3, \Lambda)\} \tag{7.71}$$

The states q_1, q_2, q_3 stand for $a^0b^0c^k$, $a^ib^ic^k$, $a^ib^ic^j$ respectively where $i > 0$, $j \geq 0$, $k \geq 0$. (7.61) indicates the initial guess. The three choices correspond to the three cases.

$$(q_0, c^k, Z_0) = (q_0, \Lambda c^k, Z_0) \vdash (q_1, c^k, Z_0) \vdash (q_1, \Lambda, Z_0)$$

by (7.61) and (7.62). As q_1 is a final state, $c^k \in T(M)$.

The pda in state q_2 will not change state and stores a's in the input string as X's in PDS ((7.63) and (7.64)). On seeing the first b after many a's, M changes its state to q_4 and cancels X in PDS for subsequent b's ((7.64)). If it reaches the bottom of PDS, M goes back to q_1, which is an accepting state ((7.76)). So M accepts a^ib^i. It continuous to be in state q_1 on seeing c's subsequently ((7.62)). So, M accepts $a^ib^ic^k$.

For dealing with $a^ib^ic^j$, M makes the initial guess using (7.61) and reaches state q_3. It simply reads a's without changing state or PDS ((7.67)). M subsequently replaces b with X and changes to state q_5. Afterwards M goes on changing b's to X's ((7.69)). On seeing a c, M changes state. Subsequent c's are matched with X's (which correspond to b's read earlier) and X's in PDS are cancelled. On reaching the bottom of PDS, M reaches q_3, a final state ((7.71)).

Thus, $a^0b^0c^k$, $a^ib^ic^k$, $a^ib^ic^j \in T(M)$ for $i > 0$, $j \geq 0$, $k \geq 0$. Hence, $T(M) = L$.

EXAMPLE 7.18

Convert the grammar $S \rightarrow aSb \mid A$, $A \rightarrow bSa \mid S \mid \Lambda$ to a pda that accepts the same language by empty stack.

Solution

We construct a pda A as

$$A = (\{q\}, \{a, b\}, \{S, A, a, b\}, \delta, q, S, \emptyset)$$

where δ is defined by the following rules

$$\delta(q, \Lambda, S) = \{(q, aSb), (q, A)\}$$

$$\delta(q, \Lambda, A) = \{(q, bSA), (q, S), (q, \Lambda)\}$$

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

$$\delta(q, b, b) = \{(q, \Lambda)\}$$

and A is the required pda.

EXAMPLE 7.19

If A is a pda, then show that there is a one-state pda A, such that

$$N(A) = N(A_1).$$

Solution

By Theorem 7.4, we get a context-free grammar G such that $L(G) = N(A)$. Denote G by (V_N, Σ, P, S). By Theorem 7.3, we can get a one-state pda A_1 given by

$$A_1 = (\{q\}, \Sigma, V_N \cup \Sigma, \delta, q, S, \emptyset)$$

such that $N(A_1) = L(G)$.

SELF-TEST

Choose the correct answer to Questions 1–6.

1. If $\delta(q, a_1, Z_1)$ contains (q', β), then
(a) $(q, a_1a_2, Z_1Z_2) \vdash (q', a_2, \beta Z_2)$
(b) $(q, a_2a_2, Z_1Z_2) \vdash (q', a_1a_2, \beta Z_2)$
(c) $(q, a_1a_2, Z_2) \vdash (q', a_1, Z_1)$
(d) $(q, a_1a_2, Z_1Z_2) \vdash (q', a_2, Z_1Z_2)$

2. In a deterministic pda, $|\delta(q, a, Z)|$ is
(a) equal to 1
(b) less than or equal to 1
(c) greater than 1
(d) greater than or equal to 1

3. In a deterministic pda:
(a) $\delta(q, a, Z) = \emptyset \Rightarrow \delta(q, \Lambda, Z) \neq \emptyset$
(b) $\delta(q, a, Z) \neq \emptyset \Rightarrow \delta(q, \Lambda, Z) = \emptyset$
(c) $\delta(q, \Lambda, Z) \neq \emptyset \Rightarrow \delta(q, a, Z) \neq \emptyset$
(d) $\delta(q, \Lambda, Z) \neq \emptyset \Rightarrow \delta(q, a, Z) = \emptyset$

4. $\{a^nb^n | n \geq 1\}$ is accepted by a pda
(a) by null store and also by final state.
(b) by null store but not by final state.
(c) by final state but not by null store.
(d) by none of these.

5. $\{a^nb^{2n} | n \geq 1\}$ is accepted by
(a) a finite automaton
(b) a nondeterministic finite automaton
(c) a pda
(d) none of these.

6. The intersection of a context-free language and a regular language is
   (a) context-free
   (b) regular but not context-free
   (c) neither context-free nor regular.
   (d) both regular and context-free.

Fill up the blanks:

7. In bottom-up parsing, we build the deviation from _____ to _____ .
8. In LR(1) grammar, we can decide the production to be applied in the next step by _____ .
9. $w \in T(A)$, where A is a pda if $(q_0, w, Z_0) \vdash^* $ _____ .
10. $w \in N(A)$, where A is a pda if $(q_0, w, Z_0) \vdash^* $ _____ .

EXERCISES

7.1 If an initial ID of the pda A in Example 7.2 is $(q_0, aacaa, Z_0)$, what is the ID after the processing of *aacaa*? If the input string is (i) *abcba*, (ii) *abcb*, (iii) *acba*, (iv) *abac*, (v) *abab*, will A process the entire string? If so, what will be the final ID?

7.2 What is the ID that the pda A given in Example 7.5 reaches after processing (i) a^3b^2, (ii) a^2b^3, (iii) a^5, (iv) b^5, (v) b^3a^2, (vi) *ababab* if A starts with the initial ID?

7.3 Construct a pda accepting by empty store each of the following languages.
   (a) $\{a^n b^m a^n \mid m, n \geq 1\}$
   (b) $\{a^n b^{2n} \mid n \geq 1\}$
   (c) $\{a^n b^m c^n \mid m, n \geq 1\}$
   (d) $\{a^m b^n \mid m > n \geq 1\}$

7.4 Construct a pda accepting by final state each of the languages given in Exercise 7.3.

7.5 Construct a context-free grammar generating each of the following languages, and hence a pda accepting each of them by empty store.
   (a) $\{a^n b^n \mid n \geq 1\} \cup \{a^m b^{2m} \mid m \geq 1\}$
   (b) $\{a^n b^m a^n \mid m, n \geq 1\} \cup \{a^n c^n \mid n \geq 1\}$
   (c) $\{a^n b^m c^m d^n \mid m, n \geq 1\}$

7.6 Let $L = \{a^m b^n \mid n < m\}$. Construct (i) a context-free grammar accepting L, (ii) a pda accepting L by empty store, and (iii) a pda accepting L by final state.

7.7 Do Exercise 7.6 by taking L to be the set of all strings over $\{a, b\}$ consisting of twice as many a's as b's.

7.8 Construct a pda accepting the set of all even-length palindromes over $\{a, b\}$ by empty store.

7.9 Show that the set of all strings over $\{a, b\}$ consisting of equal number of a's and b's is accepted by a deterministic pda.

7.10 Apply the construction given in Theorem 7.4 to the pda M given in Example 7.1 to get a context-free grammar G accepting $N(M)$.

7.11 Apply the construction given in Theorem 7.4 to the pda obtained by solving Exercise 7.4.

7.12 Show that $\{a^n b^n \mid n \geq 1\} \cup \{a^m b^{2m} \mid m \geq 1\}$ cannot be accepted by a deterministic pda.

7.13 Show that a regular set accepted by a deterministic finite automaton with n states is accepted to final state by a deterministic pda with n states and one pushdown symbol. Deduce that every regular set is a deterministic context-free language.
(A context-free language is deterministic if it is accepted by a deterministic pda.)

7.14 Show that every regular set accepted by a finite automaton with n states is accepted by a deterministic pda with one state and n pushdown symbols.

7.15 If L is accepted by a deterministic pda A, then show that L is accepted by deterministic pda A which never adds more than one symbol at a time (i.e. if $\delta(q, a, z) = (q', \gamma)$, then $|\gamma| \leq 2$).

7.16 If L is accepted by a deterministic pda A, then show that L is accepted by a deterministic pda A which always (i) removes the topmost symbol, or (ii) does not change the topmost symbol, or (iii) pushes a single symbol above the topmost symbol.

8 LR(k) Grammars

In this chapter we study LR(k) grammars (a subclass of context-free grammars) which play an important role in the study of programming languages and the design of compilers. For example, a typical programming language such as ALGOL has LR(1) parser.

8.1 LR(k) GRAMMARS

In Chapters 4 and 6 we were mainly interested in generating strings using productions and in performing the membership test. In the design of programming languages and compilers, it is essential to develop the parsing techniques, i.e. techniques for obtaining the 'reverse derivation' of a given string in a context-free language. In other words, we require techniques to find a derivation tree for a given sentence w in a context-free language.

To find a derivation tree for a given sentence w, we can start with w and replace a substring, say w_1 of w, by a variable A if $A \rightarrow w_1$ is a production. We repeat the process until we get S. But this is more easily said than done, for at every stage there may be several choices and we have to choose one among them. If we make a wrong choice, we will not get S, and in this case we have to backtrack and try some other substring. However, for a certain subclass of context-free grammars, it is possible to carry out the process, i.e. getting the derivation in the reverse order for a given string w in a deterministic way. LR(k) grammars form one such subclass. Here, LR(k) stands for left-to-right scan of the input string producing a rightmost derivation using the k symbol look-ahead on the input string.

Before discussing the LR(k) grammars, we should note that although parsing gives only the syntactical structure of a string, it is the first step in understanding the 'meaning' of the sentence.

Consider some sentential form $\alpha\beta w$ of a context-free grammar G, where $\alpha, \beta \in (V_N \cup \Sigma)^*$ and $w \in \Sigma^*$. Suppose we are interested in finding the production applied in the last step of the derivation for $\alpha\beta w$. If $A \to \beta$ is a production, it is likely that $A \to \beta$ is the production applied in the last step, but we cannot definitely say that this is the case. If it is possible to assert that $A \to \beta$ is the production applied in the last step by looking ahead for k symbols (i.e. k symbols to the right of β in $\alpha\beta w$), then G is called an LR(k) grammar. The production $A \to \beta$ is called a handle production and β is called a handle.

We write $\alpha \underset{R}{\overset{*}{\Rightarrow}} \beta$ if β is derived from α by a right-most derivation. Before giving the rigorous definition of an LR(k) grammar, let us consider a grammar for which parsing is possible by looking ahead for one symbol.

EXAMPLE 8.1

Let G be $S \to AB$, $A \to aAb$, $A \to \Lambda$, $B \to Bb$, $B \to b$. It is easy to see that $L(G) = \{a^m b^n \mid n > m \geq 1\}$. Some sentential forms of G obtained by right-most derivations are AB, ABb^k, $a^nAb^mb^k$, a^nb^{m+k}, where $k \geq 1$. AB appears as the R.H.S. of $S \to AB$. So AB may be a handle for AB or ABb^k. If we apply the handle to AB, we get $S \underset{R}{\Rightarrow} AB$. If we apply the handle to ABb^k, we get $Sb^k \Rightarrow ABb^k$. But Sb^k is not a sentential form. So to decide whether AB can be a handle, we have to scan the symbol to the right of AB. If it is Λ, then AB serves as a handle. If the next symbol is b, AB cannot be a handle. So only by looking ahead for one symbol we are able to decide whether AB is a handle. Let us consider a^2b^3. As we scan from left to right, we see that the handle production $A \to \Lambda$ may be applied. Λ can serve as a handle only when it is taken between the rightmost a and the leftmost b. In this case we get $a^2Ab^3 \underset{R}{\Rightarrow} a^2b^3$, and we are able to decide that $A \to \Lambda$ is a handle production only by looking ahead of one symbol (to the right of Λ). If Λ is taken between two a's, we get $aAab^3 \underset{R}{\Rightarrow} a^2b^3$. But $aAab^3$ is not a sentential form. Similarly, we can see that the correct handle production can be determined by looking ahead of one symbol for various sentential forms.

A rigorous definition of an LR(k) grammar is now given.

Definition 8.1 Let $G = (V_N, \Sigma, P, S)$ be a context-free grammar in which $S \overset{n}{\Rightarrow} S$ only when $n = 0$. G is an LR(k) grammar ($k \geq 0$) if

(i) $S \underset{R}{\overset{*}{\Rightarrow}} \alpha Aw \underset{R}{\Rightarrow} \alpha\beta w$, where $\alpha, \beta \in V_N^*$, $w \in \Sigma^*$,

(ii) $S \underset{R}{\overset{*}{\Rightarrow}} \alpha'A'w' \underset{R}{\Rightarrow} \alpha'\beta'w'$, where $\alpha', \beta' \in V^*$, $w' \in \Sigma^*$, and

(iii) the first $|\alpha\beta| + k$ symbols of $\alpha\beta w$ and $\alpha'\beta'w'$ coincide. Then $\alpha = \alpha'$, $A = A'$, $\beta = \beta'$.

Remarks 1. If $\alpha\beta w$ or $\alpha'\beta'w'$ have less than $|\alpha\beta| + k$ symbols, we add some 'blank symbols', say S, on the right and compare.

2. It is easy to see how we can get the derivation tree for a given terminal string. For getting the derivation tree, we want to get the derivation "in the reverse order". Suppose a sentential form $\alpha\beta w$ is encountered. We can get a right-most derivation of βw in the following way: If $A \to \beta$ is a production, then we have to decide whether $A \to \beta$ is used in the last step of a right-most derivative of $\alpha\beta w$. On seeing k symbols beyond β in $\alpha\beta w$, we are able to decide that $A \to \beta$ is the required production in the first step. For, if $\alpha'\beta'w'$ is another sentential form satisfying condition (iii), then we can apply $A' \to \beta'$ in the last step of a right-most derivation of $\alpha'\beta'w'$. But by definition it follows that $A = A'$, $\beta = \beta'$ and $\alpha = \alpha'$. So $A \to \beta$ is the only possible production we can apply and we are able to decide this after 'seeing' the k symbols beyond β. We repeat the process until we get S.

3. If G is an LR(k) grammar, it is an LR(k') grammar for all $k' > k$.

EXAMPLE 8.2

Let G be the grammar $S \to aA$, $A \to Abb \mid b$. Show that G is an LR(0) grammar.

Solution

It is easy to see that any element in $L(G)$ is of the form ab^{2n+1}. The sentential forms of G, are aA, aAb^{2n}, ab^{2n+1}. Let us find out the last production applied in the derivation of ab^{2n+1}. As aA, Abb, b are the possible right-hand sides of productions, only $A \to b$ can be the last production; we are able to decide this without looking at any symbol to the right of b. Similarly, the last productions for aAb^{2n} and aA are $A \to Abb$ and $S \to aA$, respectively. (We are able to say that $A \to Abb$ is the last production for any sentential form aAb^{2n} for all $n \geq 1$.) Thus, G is an LR(0) grammar.

EXAMPLE 8.3

Consider the grammar G given in Example 8.1. Show that G is an LR(1) grammar, but not an LR(0) grammar. Also, find the derivation tree for a^2b^4.

Solution

In Example 8.1 we have shown that for sentential forms of G we can determine the last step of a right-most derivation by looking ahead of one symbol. So G is LR(1). We have also seen that $S \to AB$ is a handle production for the sentential form AB, but not for ABb^k. In other words, the handle production cannot be determined without looking ahead. So G is not LR(0).

To get the derivation tree for a^2b^4, we scan a^2b^4 from left to right. After scanning a, we look ahead. If the next symbol is a, we continue to scan. If the next symbol is b, we decide that $A \to \Lambda$ is the required handle production. Thus the last step of the right-most derivation of a^2b^4 is

$$a^2Ab^4 \underset{R}{\Rightarrow} a^2\Lambda b^4$$

To get the last step of a^2Ab^4, we scan a^2Ab^4 from left to right. aAb is a possible handle. We are able to decide that this is the right handle without looking ahead and so we get

$$aAbb^2 \underset{R}{\Rightarrow} a^2Ab^4$$

Once again using the handle aAb, we obtain

$$Ab^2 \underset{R}{\Rightarrow} aAbb^2$$

To get the last step of the rightmost derivation of Ab^2, we scan Ab^2. A possible handle production is $B \rightarrow b$. We also note that this handle production can be applied to the first b we encounter, but not to the last b. So, we get $ABb \underset{R}{\Rightarrow} Ab^2$.

For ABb, a possible a-handle is Bb. Hence, we get $AB \underset{R}{\Rightarrow} ABb$. Finally, we obtain $S \underset{R}{\Rightarrow} AB$. Thus we have the following derivations:

$a^2Ab^4 \underset{R}{\Rightarrow} a^2Ab^4$ by looking ahead of one symbol

$aAbb^2 \underset{R}{\Rightarrow} a^2Ab^4$ by not looking ahead of any symbol

$Ab^2 \underset{R}{\Rightarrow} aAbb^2$ by not looking ahead of any symbol

$ABb \underset{R}{\Rightarrow} Ab^2$ by not looking ahead of any symbol

$AB \underset{R}{\Rightarrow} ABb$ by not looking ahead of any symbol

$S \underset{R}{\Rightarrow} AB$ by looking ahead of one symbol

The derivation tree for a^2b^4 is as shown in Fig. 8.1.

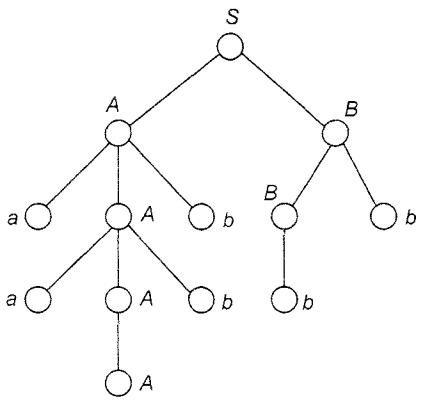


Fig. 8.1 Derivation tree for a^2b^4.

8.2 PROPERTIES OF LR(k) GRAMMARS

In this section we give some important properties of LR(k) grammars which are useful for parsing and other applications.

Recall the definition of an ambiguous grammar. A grammar G is ambiguous if there exists $w \in L(G)$ which has two derivation trees. The next theorem gives the relation between LR(k) grammars and unambiguous grammars.

Property 1 Every LR(k) grammar G is unambiguous.

Proof We have to show that for any $x \in \Sigma^*$, there exists a unique right-most derivation. Suppose we have two rightmost derivations for x, namely

$$S \underset{R}{\overset{*}{\Rightarrow}} \alpha Aw \underset{R}{\Rightarrow} \alpha\beta w = x \tag{8.1}$$

$$S \underset{R}{\overset{*}{\Rightarrow}} \alpha' A' w' \underset{R}{\Rightarrow} a'b'w' = x \tag{8.2}$$

As $\alpha\beta w = \alpha'\beta' w'$, from the definition it follows that $\alpha = \alpha'$, $A = A'$ and $\beta = \beta'$. As $a\beta w = \alpha'\beta' w'$, we get $w = w'$, and so $\alpha Aw = \alpha' A' w'$. Hence the last step in the derivations (8.1) and (8.2) is the same. Repeating the arguments for the other sentential forms derived in the course of (8.1) and (8.2), we can show that (8.1) is the same as (8.2). Therefore, G is unambiguous. ∎

We have seen that the deterministic and the nondeterministic finite automata behave in the same way in so far as acceptability of languages is concerned. The same is the case with Turing machines. But the behaviour of deterministic and nondeterministic pushdown automata is different. In Chapter 7 we have proved "that any pushdown automaton accepts a context-free language and for any context-free language L, we can construct a pushdown automaton accepting L. The following property gives the relation between LR(k) grammars and pushdown automata.

Property 2 If G is an LR(k) grammar, there exists a deterministic pushdown automaton A accepting $L(G)$.

Property 3 If A is a deterministic pushdown automaton A, there exists an LR(1) grammar G such that $L(G) = N(A)$.

Property 4 If G is an LR(k) grammar, where $k > 1$, then there exists an equivalent grammar G_1 which is LR(1). In so far as languages are concerned, it is enough to study the languages generated by LR(0) grammars and LR(1) grammars.

Definition 8.2 A context-free language is said to be deterministic if it is accepted by a deterministic pushdown automaton.

Property 5 The class of deterministic languages is a proper subclass of the class of context-free languages.

The class of deterministic languages can be denoted by $\mathcal{L}_{\text{dcfl}}$.

Property 6 $\mathcal{L}_{\text{dcfl}}$ is closed under complementation but not under union and intersection.

The following definition is useful in characterizing the languages accepted by an LR(0) grammar.

Definition 8.3 A context-free language has prefix property if no proper prefix of strings of L belongs to L.

Property 7 A context-free language is generated by an LR(0) grammar if and only if it is accepted by a deterministic pushdown automaton and has prefix property.

Property 8 There is an algorithm to decide whether a given context-free grammar is LR(k) for a given natural number k.

8.3 CLOSURE PROPERTIES OF LANGUAGES

We discussed closure properties under union, concatenation, and so on in Chapter 4. In this section we will discuss closure properties under intersection, complementation, etc. Recall that $\mathcal{L}_0$, $\mathcal{L}_{cs1}$, $\mathcal{L}_{cfl}$, $\mathcal{L}_{rl}$ are the families of type 0 languages, context-sensitive languages, context-free languages and regular languages, respectively.

Property 1 Each of the classes $\mathcal{L}_0$, $\mathcal{L}_{cs1}$, $\mathcal{L}_{cfl}$, $\mathcal{L}_{rl}$ is closed under union, concatenation, closure and transpose operations (Theorems 4.5–4.7).

Property 2 $\mathcal{L}_{rl}$ is closed under intersection and complementation (Theorems 5.7 and 5.8).

Property 3 $\mathcal{L}_{cfl}$ is not closed under intersection and complementation.

We establish property 3 by a counter-example. We have already seen that $L_1 = \{a^n b^n c^i \mid n \geq 1, i \geq 0\}$ and $L_2 = \{a^i b^n c^n \mid n \geq 1, j \geq 0\}$ are context-free languages (Examples 4.8 and 4.9). $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$. In Example 6.18, we have shown that $\{a^n b^n c^n \mid n \geq 1\}$ is not context-free. Thus, $\mathcal{L}_{cfl}$ is not closed under intersection.

Using DeMorgan's law, we can write $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. We have proved in Chapter 4 that $\mathcal{L}_{cfl}$ is closed under union. If $\mathcal{L}_{cfl}$ were closed under complementation, then $L_1 \cap L_2$ turns out to be context-free which is not true. Hence, $\mathcal{L}_{cfl}$ is not closed under complementation.

8.4 SUPPLEMENTARY EXAMPLES

EXAMPLE 8.4

Show that the grammar $S \rightarrow aAc$, $A \rightarrow Abb \mid b$ is an LR(0) grammar.

Solution

It is easy to see that $L(G) = \{ab^{2n+1}c \mid n \geq 0\}$. The sentential forms of G are aAc, $aAb^{2n}c$, $ab^{2n+1}c$. We consider the last production applied in the derivation of $ab^{2n+1}c$. As aAc, Abb and b are the possible right-hand sides of productions,

only $A \rightarrow b$ can be the last production in the rightmost derivation of $ab^{2n+1}c$. (We do not have aAc and Abb as substrings of $ab^{2n+1}c$). Similarly the last productions for aAc and $aAb^{2n}c$ are $S \rightarrow aAc$ and $A \rightarrow Abb$ respectively. Hence G is an LR(0) grammar.

EXAMPLE 8.5

Show that $S \rightarrow aAb$, $A \rightarrow cAc \mid c$ is not LR(k) for any natural number k.

Solution

It is easy to see that

$$L(G) = \{ac^{2n+1}b \mid n \geq 0\}$$

Consider $acccb \in L(G)$. The last production is $A \rightarrow c$. But we can apply this handle only by knowing the entire string. This can be applied to the middle c but this is known only after looking at two symbols beyond the c which replaces A. Continuing this argument, we can decide the handle of $ac^{2n+1}b$ by only looking at $n + 1$ symbols beyond the c which replaces A. So it is not LR(k) for any k.

EXAMPLE 8.6

Give an example of a language which can be generated by an LR(k) grammar for some k and also by a grammar that is not LR(k) for any k.

Solution

Consider $\{ac^{2n+1}b \mid n \geq 0\}$. This is generated by the grammar $S \rightarrow aAb$, $A \rightarrow cAc \mid c$ which is not LR(k) for any k.

This language can also be generated by the grammar $S \rightarrow aAb$, $A \rightarrow Acc \mid c$. This is LR(0). (This grammar is similar to the grammar in Example 8.4.)

SELF-TEST

Choose the correct answer to Questions 1–5:

1. An LR(k) grammar has to be
   (a) a type 0 grammar
   (b) a type 1 grammar
   (c) a type 2 grammar
   (d) none of these.
2. An LR(k) grammar is
   (a) always unambiguous
   (b) always ambiguous
   (c) need not be unambiguous
   (d) none of these.

3. A handle is
   (a) a string of variables and terminals
   (b) a string of variables
   (c) a string of terminals
   (d) a production.

4. The automaton corresponding to an LR(k) grammar is
   (a) a deterministic finite automaton
   (b) a nondeterministic finite automaton
   (c) a deterministic PDA
   (d) a nondeterministic PDA.

5. $\mathscr{L}_{\text{dcfl}}$ is closed under
   (a) union
   (b) complementation
   (c) intersection
   (d) none of these.

EXERCISES

8.1 Show that the grammar $S \to aAb,\ A \to aAb \mid a$ is an LR(1) or is it an LR(0)?

8.2 Show that the grammar $S \to 0A2,\ A \to 1A1,\ A \to 1$ is not an LR(0).

8.3 Is $S \to AB,\ S \to aA,\ A \to aA,\ A \to a,\ B \to a$ an LR(k) for some k?

8.4 Show that $\{a^m b^n c^n \mid m, n \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}$ cannot be generated by an LR(k) grammar for any k.

8.5 Are the following statements true? (a) If G is unambiguous, it is LR(k) for some k. (b) If G is unambiguous, it is LR(k) for every k. Justify your answer.

8.6 Is $S \to C \mid D,\ C \to aC \mid b,\ D \to aD \mid$ an LR(0)?

8.7 For a production $A \to \beta$ of a context-free grammar G and w in $\Sigma^*\$^k$ ($\$$ is a symbol not in $V_N \cup \Sigma$), define $R_k(w)$ to be the set of all strings of the form $\alpha\beta w$ such that $A \to \beta$ is a handle for $\alpha\beta ww'$ for some w' in $\Sigma^*\ \$^*$ and $S\$ \underset{R}{\overset{*}{\Rightarrow}} \alpha A w w' \underset{R}{\Rightarrow} \alpha\beta w w'$. (In other words, a string $\alpha\beta w$ is in $R_k(w)$ if we get a penultimate step of a rightmost derivation of $\alpha\beta ww'$ for some w'.) Show that $R_k(w)$ is a regular set.

[*Hint:* Define $G' = (V'_N,\ V_N \cup \Sigma,\ P',\ S')$, where

$$V_N = \{[A,\ w] \mid A \in V_N,\ w \in \Sigma^*\$^k \text{ and } |w| = k\}$$

$$S' = [S,\ \$^k]$$

Also show that each production in P induces a production in P' in the following manner:

(a) If $A \to x$ is in P, where $x \in \Sigma^*$, then $[A, w] \to xw$ is included in P'.

(b) If $A \to X_1X_2 \ldots X_m$ is in P, $X_j \in V_N$, $X_{j+1} \ldots X_m w \overset{*}{\underset{G}{\Rightarrow}} w'w''$ for some w', w'' in $\Sigma^*\$^*$ with $|w'| = k$, then $[A, w] \to B_1 \ldots B_{j-1} [B_j, w']$ is included in P'.

As the productions in P' have either a terminal string or a terminal string followed by a variable on R.H.S., G' can be reduced to an equivalent regular grammar. Use the principle of induction to show that

$$[S, \$^k] \overset{*}{\underset{G'}{\Rightarrow}} [A, w] \text{ if and only if for some } w'S\$^k \overset{*}{\underset{R}{\Rightarrow}} \alpha A w w'.$$

This will establish $L(G') = R_k(w)$.

8.8 Prove that a context-free grammar G is an LR(k) if and only if the following holds: A string γ in $R_k(w)$ corresponding to a production $A_1 \to \beta_1$ is a substring of some element δ in $R_k(w')$ corresponding to a production $A_2 \to \beta_2$ implies $\gamma = \delta$, $A_1 = A_2$, $\beta_1 = \beta_2$.

[*Hint:* The proof follows from the definition of LR(k) grammars.]

8.9 Prove Property 2 of Section 8.2.

Solution We give an outline of the construction of the required dpda A. A accepts a string w if $S\$^k$ remains in the stack after the processing of w. For this purpose, A has to simulate the reverse derivation of w. This is achieved by finding suitable handles. $R_k(w)$'s are defined precisely for this purpose (refer to Exercise 8.7). As $R_k(w)$'s are regular, there exist deterministic finite automata $M_k(w)$ corresponding to $R_k(w)$.

For our deterministic pda A to contain the information regarding the finite automata, the pushdown store of A is required to have an additional track. In the first track, symbols from $V_N \cup \Sigma$ are written or erased. In the additional track, the information regarding $M_k(w)$'s is stored in the form of maps. The map N_α gives the states of finite automata $M_k(w)$ after the processing of a string α for all productions in G and strings w in $\Sigma^*\$^*$ of length k. The existence of a suitable handle is indicated by a final state of $M_k(w)$ (in the second track).

We can describe the way A acts as follows: A is capable of reading $k + 1$ symbols on PDS, where l is the length of the longest R.H.S. of productions of G. (This can be achieved by modifying the finite control.) A reads the top m symbols on track 1 for some $m \leq k + l$. The second track is suitably manipulated. For example, if $X_1 \ldots X_m$ is in track 1, then track 2 stores the maps $N_{X1}N_{X1X2} \ldots N_{X1 \ldots Xm}$. If a suitable handle is found, then a sentential form is obtained by replacing the R.H.S. of the

handle of its L.H.S. in the given string on track 1. As G is LR(k), this can be done in almost one way.

The above process is repeated until it is no longer possible. If m symbols are not sufficient to carry out the process, more symbols are read and placed on the stack (track 1).

If stack 1 has $S\$^k$ at a particular stage, A accepts the corresponding string. A is the required dpda accepting $L(G)$.

9 Turing Machines and Linear Bounded Automata

In the early 1930s, mathematicians were trying to define effective computation. Alan Turing in 1936, Alanzo Church in 1933, S.C. Kleene in 1935, Schonfinkel in 1965 gave various models using the concept of Turing machines, λ-calculus, combinatory logic, post-systems and μ-recursive functions. It is interesting to note that these were formulated much before the electro-mechanical/electronic computers were devised. Although these formalisms, describing effective computations, are dissimilar, they turn to be equivalent.

Among these formalisms, the Turing's formulation is accepted as a model of algorithm or computation. The Church–Turing thesis states that any algorithmic procedure that can be carried out by human beings/computer can be carried out by a Turing machine. It has been universally accepted by computer scientists that the Turing machine provides an ideal theoretical model of a computer.

Turing machines are useful in several ways. As an automaton, the Turing machine is the most general model. It accepts type-0 languages. It can also be used for computing functions. It turns out to be a mathematical model of partial recursive functions. Turing machines are also used for determining the undecidability of certain languages and measuring the space and time complexity of problems. These are the topics of discussion in this chapter and some of the subsequent chapters.

For formalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional paper (instead of a two-dimensional paper as is usually done) which can be viewed as a tape divided into cells.

One scans the cells one at a time and usually performs one of the three simple operations, namely (i) writing a new symbol in the cell being currently

scanned, (ii) moving to the cell left of the present cell, and (iii) moving to the cell right of the present cell. With these observations in mind, Turing proposed his 'computing machine.'

9.1 TURING MACHINE MODEL

The Turing machine can be thought of as finite control connected to a R/W (read/write) head. It has one tape which is divided into a number of cells. The block diagram of the basic model for the Turing machine is given in Fig. 9.1.

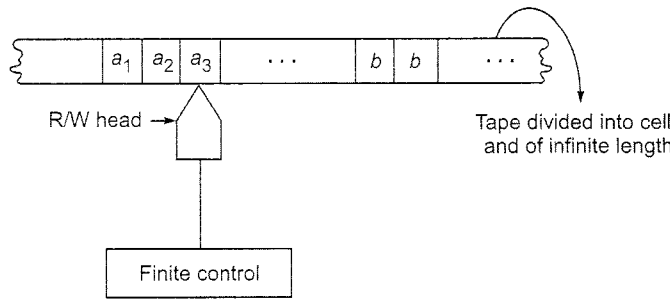


Fig. 9.1 Turing machine model.

Each cell can store only one symbol. The input to and the output from the finite state automaton are effected by the R/W head which can examine one cell at a time. In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine

(i) a new symbol to be written on the tape in the cell under the R/W head,
(ii) a motion of the R/W head along the tape: either the head moves one cell left (L), or one cell right (R),
(iii) the next state of the automaton, and
(iv) whether to halt or not.

The above model can be rigorously defined as follows:

Definition 9.1 A Turing machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where

1. Q is a finite nonempty set of states,
2. Γ is a finite nonempty set of tape symbols,
3. $b \in \Gamma$ is the blank,
4. Σ is a nonempty set of input symbols and is a subset of Γ and $b \notin \Sigma$,
5. δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head; $D = L$ or R according as the movement is to the left or right.
6. $q_0 \in Q$ is the initial state, and
7. $F \subseteq Q$ is the set of final states.

Notes: (1) The acceptability of a string is decided by the reachability from the initial state to some final state. So the final states are also called the accepting states.

(2) δ may not be defined for some elements of $Q \times \Gamma$.

9.2 REPRESENTATION OF TURING MACHINES

We can describe a Turing machine employing (i) instantaneous descriptions using move-relations, (ii) transition table, and (iii) transition diagram (transition graph).

9.2.1 REPRESENTATION BY INSTANTANEOUS DESCRIPTIONS

'Snapshots' of a Turing machine in action can be used to describe a Turing machine. These give 'instantaneous descriptions' of a Turing machine. We have defined instantaneous descriptions of a pda in terms of the current state, the input string to be processed, and the topmost symbol of the pushdown store. But the input string to be processed is not sufficient to be defined as the ID of a Turing machine, for the R/W head can move to the left as well. So an ID of a Turing machine is defined in terms of the entire input string and the current state.

Definition 9.2 An ID of a Turing machine M is a string $\alpha\beta\gamma$, where β is the present state of M, the entire input string is split as $\alpha\gamma$, the first symbol of γ is the current symbol a under the R/W head and γ has all the subsequent symbols of the input string, and the string α is the substring of the input string formed by all the symbols to the left of a.

EXAMPLE 9.1

A snapshot of Turing machine is shown in Fig. 9.2. Obtain the instantaneous description.

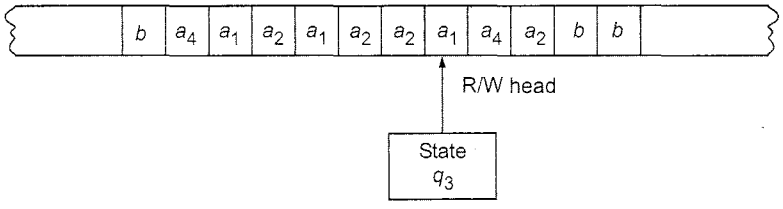


Fig. 9.2 A snapshot of Turing machine.

Solution

The present symbol under the R/W head is a_1. The present state is q_3. So a_1 is written to the right of q_3. The nonblank symbols to the left of a_1 form the string $a_4a_1a_2a_1a_2a_2$, which is written to the left of q_3. The sequence of nonblank symbols to the right of a_1 is a_4a_2. Thus the ID is as given in Fig. 9.3.

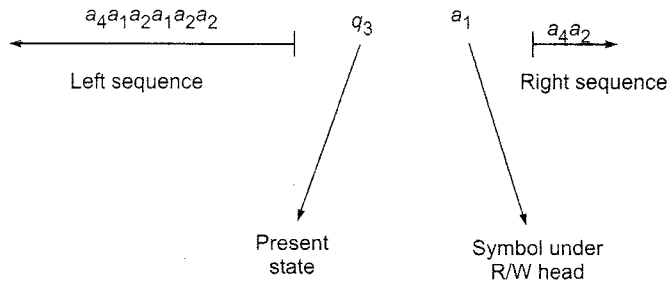


Fig. 9.3 Representation of ID.

Notes: (1) For constructing the ID, we simply insert the current state in the input string to the left of the symbol under the R/W head.

(2) We observe that the blank symbol may occur as part of the left or right substring.

Moves in a TM

As in the case of pushdown automata, $\delta(q, x)$ induces a change in ID of the Turing machine. We call this change in ID a move.

Suppose $\delta(q, x_i) = (p, y, L)$. The input string to be processed is $x_1x_2 \ldots x_n$, and the present symbol under the R/W head is x_i. So the ID before processing x_i is

$$x_1x_2 \ldots x_{i-1}qx_i \ldots x_n$$

After processing x_i, the resulting ID is

$$x_1 \ldots x_{i-2}\, px_{i-1}yx_{i+1} \ldots x_n$$

This change of ID is represented by

$$x_1x_2 \ldots x_{i-1}\, q\, x_i \ldots x_n \vdash x_i \ldots x_{i-2}\, px_{i-1}\, yx_{i+1} \ldots x_n$$

If $i = 1$, the resulting ID is $p\, y\, x_2x_3 \ldots x_n$.

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

$$x_1x_2 \ldots x_{i-1}q\, x_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}\, y\, px_{i+1} \ldots x_n$$

If $i = n$, the resulting ID is $x_1x_2 \ldots x_{n-1}\, y\, p\, b$.

We can denote an ID by I_j for some j. $I_j \vdash I_k$ defines a relation among IDs. So the symbol $\vdash^*$ denotes the reflexive–transitive closure of the relation $\vdash$. In particular, $I_j \vdash^* I_j$. Also, if $I_1 \vdash^* I_n$, then we can split this as $I_1 \vdash I_2 \vdash I_3 \vdash \ldots \vdash I_n$ for some IDs, $I_2, \ldots, I_{n-1}$.

Note: The description of moves by IDs is very much useful to represent the processing of input strings.

9.2.2 Representation by Transition Table

We give the definition of δ in the form of a table called the transition table. If $\delta(q, a) = (\gamma, \alpha, \beta)$, we write $\alpha\beta\gamma$ under the α-column and in the q-row. So if

we get $\alpha\beta\gamma$ in the table, it means that α is written in the current cell, β gives the movement of the head (L or R) and γ denotes the new state into which the Turing machine enters.

Consider, for example, a Turing machine with five states $q_1, \ldots, q_5$, where q_1 is the initial state and q_5 is the (only) final state. The tape symbols are 0, 1 and b. The transition table given in Table 9.1 describes δ.

TABLE 9.1 Transition Table of a Turing Machine

Present state	*Tape symbol*		
	b	0	1
$\rightarrow q_1$	$1Lq_2$	$0Rq_1$	
q_2	bRq_3	$0Lq_2$	$1Lq_2$
q_3		bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
(q_5)	$0Lq_2$		

As in Chapter 3, the initial state is marked with $\rightarrow$ and the final state with ○.

EXAMPLE 9.2

Consider the TM description given in Table 9.1. Draw the computation sequence of the input string 00.

Solution

We describe the computation sequence in terms of the contents of the tape and the current state. If the string in the tape is $a_1a_2 \ldots a_j a_{j+1} \ldots a_m$ and the TM in state q is to read a_{j+1}, then we write $a_1a_2 \ldots a_j q\, a_{j+1} \ldots a_m$.

For the input string $00b$, we get the following sequence:

$$q_100b \vdash 0q_10b \vdash 00q_1b \vdash 0q_201 \vdash q_2001$$
$$\vdash q_2b001 \vdash bq_3001 \vdash bbq_401 \vdash bb0q_41 \vdash bb0_1q_4b$$
$$\vdash bb010q_5 \vdash bb01q_200 \vdash bb0q_2100 \vdash bbq_20100$$
$$\vdash bq_2b0100 \vdash bbq_30100 \vdash bbbq_4100 \vdash bbb_1q_400$$
$$\vdash bbb10q_40 \vdash bbb100q_4b \vdash bbb1000q_5b$$
$$\vdash bbb100q_200 \vdash bbb10q_2000 \vdash bbb1q_20000$$
$$\vdash bbbq_210000 \vdash bbq_2b10000 \vdash bbbq_310000 \vdash bbbbq_50000$$

9.2.3 Representation by Transition Diagram

We can use the transition systems introduced in Chapter 3 to represent Turing machines. The states are represented by vertices. Directed edges are used to

represent transition of states. The labels are triples of the form (α, β, γ), where $\alpha, \beta \in \Gamma$ and $\gamma \in \{L, R\}$. When there is a directed edge from q_i to q_j with label (α, β, γ), it means that

$$\delta(q_i, \alpha) = (q_j, \beta, \gamma)$$

During the processing of an input string, suppose the Turing machine enters q_i and the R/W head scans the (present) symbol α. As a result, the symbol β is written in the cell under the R/W head. The R/W head moves to the left or to the right, depending on γ, and the new state is q_j.

Every edge in the transition system can be represented by a 5-tuple $(q_i, \alpha, \beta, \gamma, q_j)$. So each Turing machine can be described by the sequence of 5-tuples representing all the directed edges. The initial state is indicated by $\rightarrow$ and any final state is marked with $\circ$.

EXAMPLE 9.3

M is a Turing machine represented by the transition system in Fig. 9.4. Obtain the computation sequence of M for processing the input string 0011.

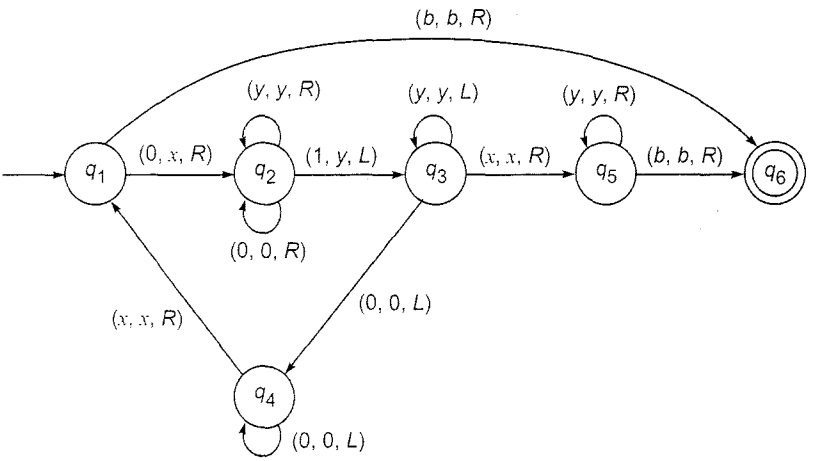


Fig. 9.4 Transition system for M.

Solution

The initial tape input is $b0011b$. Let us assume that M is in state q_1 and the R/W head scans 0 (the first 0). We can represent this as in Fig. 9.5. The figure can be represented by

$$\begin{array}{c} \downarrow \\ b0011b \\ q_1 \end{array}$$

From Fig. 9.4 we see that there is a directed edge from q_1 to q_2 with the label $(0, x, R)$. So the current symbol 0 is replaced by x and the head moves right. The new state is q_2. Thus, we get

$$\begin{array}{c} \downarrow \\ bx011b \\ q_2 \end{array}$$

The change brought about by processing the symbol 0 can be represented as

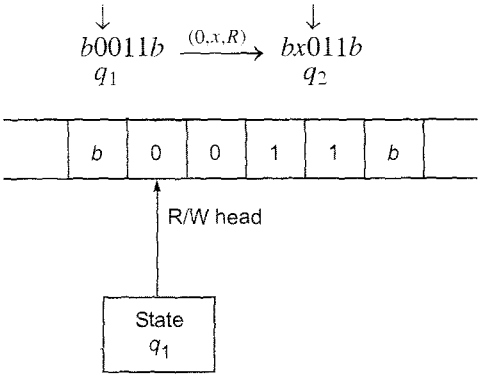


Fig. 9.5 TM processing 0011.

The entire computation sequence reads as follows:

$$\underset{q_1}{\downarrow}\, b0011b \xrightarrow{(0,x,R)} \underset{q_2}{\downarrow}\, bx011b \xrightarrow{(0,0,R)} \underset{q_2}{\downarrow}\, bx011b$$

$$\xrightarrow{(1,y,L)} \underset{q_3}{\downarrow}\, bx0y1b \xrightarrow{(0,0,L)} \underset{q_4}{\downarrow}\, bx0y1b \xrightarrow{(x,x,R)} \underset{q_1}{\downarrow}\, bx0y1b$$

$$\xrightarrow{(0,x,R)} \underset{q_2}{\downarrow}\, bxxy1b \xrightarrow{(y,y,R)} \underset{q_2}{\downarrow}\, bxxy1b \xrightarrow{(1,y,L)} \underset{q_3}{\downarrow}\, bxxyyb$$

$$\xrightarrow{(y,y,L)} \underset{q_3}{\downarrow}\, bxxyyb \xrightarrow{(x,x,R)} \underset{q_5}{\downarrow}\, bxxyyb \xrightarrow{(y,y,R)} \underset{q_5}{\downarrow}\, bxxyyb$$

$$\xrightarrow{(y,y,R)} \underset{q_5}{\downarrow}\, bxxyyb \xrightarrow{(b,b,R)} \underset{q_6}{\downarrow}\, bxxyybb$$

9.3 LANGUAGE ACCEPTABILITY BY TURING MACHINES

Let us consider the Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$. A string w in Σ^* is said to be accepted by M if $q_0 w \overset{*}{\vdash} \alpha_1 p \alpha_2$ for some $p \in F$ and $\alpha_1, \alpha_2 \in \Gamma^*$.

M does not accept w if the machine M either halts in a nonaccepting state or does not halt.

It may be noted that though there are other equivalent definitions of acceptance by the Turing machine, we will be not discussing them in this text.

EXAMPLE 9.4

Consider the Turing machine M described by the transition table given in Table 9.2. Describe the processing of (a) 011, (b) 0011, (c) 001 using IDs. Which of the above strings are accepted by M?

TABLE 9.2 Transition Table for Example 9.4

Present state	Tape symbol				
	0	1	x	y	b
$\rightarrow q_1$	xRq_2				bRq_5
q_2	$0Rq_2$	yLq_3		yRq_2	
q_3	$0Lq_4$		xRq_5	yLq_3	
q_4	$0Lq_4$		xRq_1		
q_5				$yxRq_5$	bRq_6
(q_6)					

Solution

(a) $q_1011 \vdash xq_211 \vdash q_3xy1 \vdash xq_5y1 \vdash xyq_51$

As $\delta(q_5, 1)$ is not defined, M halts; so the input string 011 is not accepted.

(b) $q_10011 \vdash xq_2011 \vdash x0q_211 \vdash xq_30y1 \vdash q_4x0y1 \vdash xq_10y1$.

$\vdash xxq_2y1 \vdash xxyq_21 \vdash xxq_3yy \vdash xq_3xyy \vdash xxq_5yy$

$\vdash xxyq_5y \vdash xxyyq_5b \vdash xxyybq_6$

M halts. As q_6 is an accepting state, the input string 0011 is accepted by M.

(c) $q_1001 \vdash xq_201 \vdash x0q_21 \vdash xq_30y \vdash q_4x0y$

$\vdash xq_10y \vdash xxq_2y \vdash xxyq_2$

M halts. As q_2 is not an accepting state, 001 is not accepted by M.

9.4 DESIGN OF TURING MACHINES

We now give the basic guidelines for designing a Turing machine.

(i) The fundamental objective in scanning a symbol by the R/W head is to 'know' what to do in the future. The machine must remember the past symbols scanned. The Turing machine can remember this by going to the next unique state.

(ii) The number of states must be minimized. This can be achieved by changing the states only when there is a change in the written symbol or when there is a change in the movement of the R/W head. We shall explain the design by a simple example.

EXAMPLE 9.5

Design a Turing machine to recognize all strings consisting of an even number of 1's.

Solution

The construction is made by defining moves in the following manner:

(a) q_1 is the initial state. M enters the state q_2 on scanning 1 and writes b.

(b) If M is in state q_2 and scans 1, it enters q_1 and writes b.

(c) q_1 is the only accepting state.

So M accepts a string if it exhausts all the input symbols and finally is in state q_1. Symbolically,

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

where δ is defined by Table 9.3.

TABLE 9.3 Transition Table for Example 9.5

Present state	1
$\rightarrow(q_1)$	bq_2R
q_2	bq_1R

Let us obtain the computation sequence of 11. Thus, $q_1 11 \vdash bq_2 1 \vdash bbq_1$. As q_1 is an accepting state, 11 is accepted. $q_1 111 \vdash bq_2 11 \vdash bbq_1 1 \vdash bbbq_2$. M halts and as q_2 is not an accepting state, 111 is not accepted by M.

EXAMPLE 9.6

Design a Turing machine over $\{1, b\}$ which can compute a concatenation function over $\Sigma = \{1\}$. If a pair of words (w_1, w_2) is the input, the output has to be $w_1 w_2$.

Solution

Let us assume that the two words w_1 and w_2 are written initially on the input tape separated by the symbol b. For example, if $w_1 = 11$, $w_2 = 111$, then the input and output tapes are as shown in Fig. 9.6.

Fig. 9.6 Input and output tapes.

We observe that the main task is to remove the symbol b. This can be done in the following manner:

(a) The separating symbol b is found and replaced by 1.

(b) The rightmost 1 is found and replaced by a blank b.
(c) The R/W head returns to the starting position.

A computation is illustrated in Table 9.4.

TABLE 9.4 Computation for $11b111$

$q_0 11b111 \vdash 1q_0 1b111 \vdash 11q_0 b111 \vdash 111q_1 111$
$\vdash 1111q_1 11 \vdash 11111q_1 1 \vdash 111111q_1 b \vdash 11111q_2 1b$
$\vdash 1111q_3 1bb \vdash 111q_3 11bb \vdash 11q_3 111bb \vdash 1q_3 1111bb$
$\vdash q_3 11111bb \vdash q_3 b11111bb \vdash bq_f 11111bb$

From the above computation sequence for the input string $11b111$, we can construct the transition table given in Table 9.5.

For the input string $1b1$, the computation sequence is given as

$$q_0 1b1 \vdash 1q_0 b1 \vdash 11q_1 1 \vdash 111q_1 b \vdash 11q_2 b \vdash 1q_3 1bb$$
$$\vdash q_3 11bb \vdash q_3 b11bb \vdash bq_f 11bb.$$

TABLE 9.5 Transition Table for Example 9.6

Present state	*Tape symbol*	
	1	*b*
$\rightarrow q_0$	$1Rq_0$	$1Rq_1$
q_1	$1Rq_1$	bLq_2
q_2	bLq_3	—
q_3	$1Lq_3$	bRq_f
(q_f)	—	—

EXAMPLE 9.7

Design a TM that accepts

$$\{0^n 1^n | n \geq 1\}.$$

Solution

We require the following moves:

(a) If the leftmost symbol in the given input string w is 0, replace it by x and move right till we encounter a leftmost 1 in w. Change it to y and move backwards.
(b) Repeat (a) with the leftmost 0. If we move back and forth and no 0 or 1 remains, move to a final state.
(c) For strings not in the form $0^n 1^n$, the resulting state has to be nonfinal.

Keeping these ideas in our mind, we construct a TM M as follows:

where

$$M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$$
$$Q = \{q_0, q_1, q_2, q_3, q_f\}$$
$$F = \{q_f\}$$
$$\Sigma = \{0, 1\}$$
$$\Gamma = \{0, 1, x, y, b\}$$

The transition diagram is given in Fig. 9.7. M accepts $\{0^n1^n \mid n \geq 1\}$. The moves for 0011 and 010 are given below just to familiarize the moves of M to the reader.

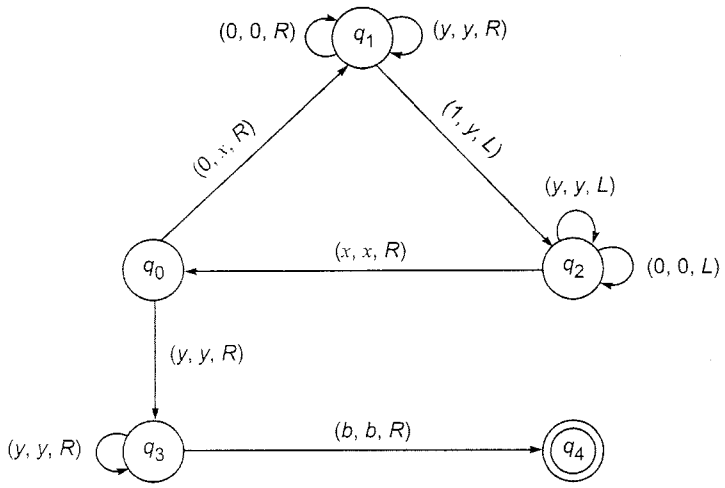


Fig. 9.7 Transition diagram for Example 9.7.

$$\begin{aligned} q_00011 &\vdash xq_1011 \vdash x0q_111 \vdash xq_20y1 \\ &\vdash q_2x0y1 \vdash xq_00y1 \vdash xxq_1y1 \vdash xxyq_11 \\ &\vdash xxq_2yy \vdash xq_2xyy \vdash xxq_0yy \vdash xxyq_3y \\ &\vdash xxyyq_3 = xxyyq_3b \vdash xxyybq_4b \end{aligned}$$

Hence 0011 is accepted by M.

$$q_0010 \vdash xq_110 \vdash q_2xy0 \vdash xq_0y0 \vdash xyq_30$$

As $\delta(q_3, 0)$ is not defined, M halts. So 010 is not accepted by M.

EXAMPLE 9.8

Design a Turing machine M to recognize the language

$$\{1^n2^n3^n \mid n \geq 1\}.$$

Solution

Before designing the required Turing machine M, let us evolve a procedure for processing the input string 112233. After processing, we require the ID to be of the form $bbbbbbq_7$. The processing is done by using five steps:

Step 1 q_1 is the initial state. The R/W head scans the leftmost 1, replaces 1 by b, and moves to the right. M enters q_2.

Step 2 On scanning the leftmost 2, the R/W head replaces 2 by b and moves to the right. M enters q_3.

Step 3 On scanning the leftmost 3, the R/W head replaces 3 by b, and moves to the right. M enters q_4.

Step 4 After scanning the rightmost 3, the R/W heads moves to the left until it finds the leftmost 1. As a result, the leftmost 1, 2 and 3 are replaced by b.

Step 5 Steps 1–4 are repeated until all 1's, 2's and 3's are replaced by blanks.

The change of IDs due to processing of 112233 is given as

$$q_1112233 \vdash bq_212233 \vdash blq_2233 \vdash b1bq_3233 \vdash b1b2q_333$$

$$\vdash b1b2bq_43 \vdash b1b_2q_5b3 \vdash b1bq_52b3 \vdash b1q_5b2b3 \vdash bq_51b2b3$$

$$\vdash q_6b1b2b3 \vdash bq_11b2b3 \vdash bbq_2b2b3 \vdash bbbq_22b3$$

$$\vdash bbbbq_3b3 \vdash bbbbbq_33 \vdash bbbbbbq_4b \vdash bbbbbq_7bb$$

Thus,

$$q_1112233 \vdash^* q_7bbbbbb$$

As q_7 is an accepting state, the input string 112233 is accepted.

Now we can construct the transition table for M. It is given in Table 9.6.

TABLE 9.6 Transition Table for Example 9.7

Present state	Input tape symbol			
	1	2	3	b
$\rightarrow q_1$	bRq_2			bRq_1
q_2	$1Rq_2$	bRq_3		bRq_2
q_3		$2Rq_3$	bRq_4	bRq_3
q_4			$3Lq_5$	bLq_7
q_5	$1Lq_6$	$2Lq_5$		bLq_5
q_6	$1Lq_6$			bRq_1
(q_7)				

It can be seen from the table that strings other than those of the form $0^n1^n2^n$ are not accepted. It is advisable to compute the computation sequence for strings like 1223, 1123, 1233 and then see that these strings are rejected by M.

9.5 DESCRIPTION OF TURING MACHINES

In the examples discussed so far, the transition function δ was described as a partial function (function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is not defined for all (q, x)) by spelling out the current state, the input symbol, the resulting state, the tape symbol replacing the input symbol and the movement of R/W head to the left or right. We can call this a formal description of a TM. Just as we have the machine language and higher level languages for a computer, we can have a higher level of description, called the *implementation description*. In this case we describe the movement of the head, the symbol stored etc. in English. For example, a single instruction like 'move to right till the end of the input string' requires several moves. A single instruction in the implementation description is equivalent to several moves of a standard TM (Hereafter a standard TM refers to the TM defined in Definition 9.1). At a higher level we can give instructions in English language even without specifying the state or transition function. This is called a *high-level description*.

In the remaining sections of this chapter and later chapters, we give implementation description or high-level description.

9.6 TECHNIQUES FOR TM CONSTRUCTION

In this section we give some high-level conceptual tools to make the construction of TMs easier. The Turing machine defined in Section 9.1 is called the standard Turing machine.

9.6.1 TURING MACHINE WITH STATIONARY HEAD

In the definition of a TM we defined $\delta(q, a)$ as (q', y, D) where $D = L$ or R. So the head moves to the left or right after reading an input symbol. Suppose, we want to include the option that the head can continue to be in the same cell for some input symbol. Then we define $\delta(q, a)$ as (q', y, S). This means that the TM, on reading the input symbol a, changes the state to q' and writes y in the current cell in place of a and continues to remain in the same cell. In terms of IDs,

$$wqax \vdash wq'yx$$

Of course, this move can be simulated by the standard TM with two moves, namely

$$wqax \vdash wyq''x \vdash wq'yx$$

That is, $\delta(q, a) = (q', y, S)$ is replaced by $\delta(q, a) = (q'', y, R)$ and $\delta(q'', X) = (q', y, L)$ for any tape symbol X.

Thus in this model $\delta(q, a) = (q', y, D)$ where $D = L, R$ or S.

9.6.2 Storage in the State

We are using a state, whether it is of a FA or pda or TM, to 'remember' things. We can use a state to store a symbol as well. So the state becomes a pair (q, a) where q is the state (in the usual sense) and a is the tape symbol stored in (q, a). So the new set of states becomes $Q \times \Gamma$.

EXAMPLE 9.9

Construct a TM that accepts the language **0 1* + 1 0*.**

Solution

We have to construct a TM that remembers the first symbol and checks that it does not appear afterwards in the input string. So we require two states, q_0, q_1. The tape symbols are 0, 1 and b. So the TM, having the 'storage facility in state', is

$$M = (\{q_0, q_1\} \times \{0, 1, b\}, \{0, 1\}, \{0, 1, b\}, \delta, [q_0, b], \{[q_1, b]\})$$

We describe δ by its implementation description.

1. In the initial state, M is in q_0 and has b in its data portion. On seeing the first symbol of the input sting w, M moves right, enters the state q_1 and the first symbol, say a, it has seen.
2. M is now in $[q_1, a]$. (i) If its next symbol is b, M enters $[q_1, b]$, an accepting state. (ii) If the next symbol is a, M halts without reaching the final state (i.e. δ is not defined). (iii) If the next symbol is $\bar{a}$ ($\bar{a} = 0$ if $a = 1$ and $\bar{a} = 1$ if $a = 0$), M moves right without changing state.
3. Step 2 is repeated until M reaches $[q_1, b]$ or halts (δ is not defined for an input symbol in w).

9.6.3 Multiple Track Turing Machine

In the case of TM defined earlier, a single tape was used. In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. Hence the only difference between the standard TM and the TM with multiple tracks is the set of tape symbols. In the case of the standard Turing machine, tape symbols are elements of Γ; in the case of TM with multiple track, it is Γ^k. The moves are defined in a similar way.

9.6.4 Subroutines

We know that subroutines are used in computer languages, when some task has to be done repeatedly. We can implement this facility for TMs as well.

First, a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt. For using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of TM.

We use this concept to design a TM for performing multiplication of two positive integers.

EXAMPLE 9.10

Design a TM which can multiply two positive integers.

Solution

The input (m, n), m, n being given, the positive integers are represented by 0^m10^n. M starts with 0^m10^n in its tape. At the end of the computation, 0^{mn}(mn in unary representation) surrounded by b's is obtained as the ouput.

The major steps in the construction are as follows:

1. 0^m10^n1 is placed on the tape (the output will be written after the rightmost 1).
2. The leftmost 0 is erased.
3. A block of n 0's is copied onto the right end.
4. Steps 2 and 3 are repeated m times and 10^m10^{mn} is obtained on the tape.
5. The prefix 10^m1 of 10^m10^{mn} is erased, leaving the product mn as the output.

For every 0 in 0^m, 0^n is added onto the right end. This requires repetition of step 3. We define a subroutine called COPY for step 3.

For the subroutine COPY, the initial state is q_1 and the final state is q_5. δ is given by the transition table (see Table 9.7).

TABLE 9.7 Transition Table for Subroutine COPY

State	*Tape symbol*			
	0	1	2	*b*
q_1	$q_2 2R$	$q_4 1L$	—	—
q_2	$q_2 0R$	$q_2 1R$	—	$q_3 0L$
q_3	$q_3 0L$	$q_3 1L$	$q_1 2R$	—
q_4	—	$q_5 1R$	$q_4 0L$	—
q_5	—	—	—	—

The Turing machine M has the initial state q_0. The initial ID for M is $q_00^m10^n1$. On seeing 0, the following moves take place (q_6 is a state of M). $q_00^m10^n1 \vdash bq_60^{m-1}10^n1 \vdash^* b0^{m-1}q_610^n1 \vdash b0^{m-1}1q_10^n1$. q_1 is the initial state

of COPY. The TM M_1 performs the subroutine COPY. The following moves take place for M_1: $q_1 0^n 1 \vdash 2q_2 0^{n-1} 1 \vdash^* 20^{n-1} 1 q_3 b \vdash 20^{n-1} q_3 10 \vdash^* 2q_1 0^{n-1} 10$. After exhausting 0's, q_1 encounters 1. M_1 moves to state q_4. All 2's are converted back to 0's and M_1 halts in q_5. The TM M picks up the computation by starting from q_5. The q_0 and q_6 are the states of M. Additional states are created to check whether each 0 in 0^m gives rise to 0^n at the end of the rightmost 1 in the input string. Once this is over, M erases $10^n 1$ and finds 0^{nm} in the input tape.

M can be defined by

$$M = (\{q_0, q_1, \ldots, q_{12}\}, \{0, 1\}, \{0, 1, 2, b\}, \delta, q_0, b, \{q_{12}\})$$

where δ is defined by Table 9.8.

TABLE 9.8 Transition Table for Example 9.10

	0	1	2	b
q_0	$q_6 bR$	—	—	—
q_6	$q_6 0R$	$q_1 1R$	—	—
q_5	$q_7 0L$	—	—	—
q_7	—	$q_8 1L$	—	—
q_8	$q_9 0L$	—	—	$q_{10} bR$
q_9	$q_9 0L$	—	—	$q_0 bR$
q_{10}	—	$q_{11} bR$	—	—
q_{11}	$q_{11} bR$	$q_{12} bR$	—	—

Thus M performs multiplication of two numbers in unary representation.

9.7 VARIANTS OF TURING MACHINES

The Turing machine we have introduced has a single tape. $\delta(q, a)$ is either a single triple (p, y, D), where $D = R$ or L, or is not defined. We introduce two new models of TM:

(i) a TM with more than one tape

(ii) a TM where $\delta(q, a) = \{(p_1, y_1, D_1), (p_2, y_2, D_2), \ldots, (p_r, y_r, D_r)\}$. The first model is called a multitape TM and the second a nondeterministic TM.

9.7.1 MULTITAPE TURING MACHINES

A multitape TM has a finite set Q of states, an initial state q_0, a subset F of Q called the set of final states, a set P of tape symbols, a new symbol b, not in P called the blank symbol. (We assume that $\Sigma \subseteq \Gamma$ and $b \notin \Sigma$.)

There are k tapes, each divided into cells. The first tape holds the input string w. Initially, all the other tapes hold the blank symbol.

Initially the head of the first tape (input tape) is at the left end of the input w. All the other heads can be placed at any cell initially.

δ is a partial function from $Q \times \Gamma^k$ into $Q \times \Gamma^k \times \{L, R, S\}^k$. We use implementation description to define δ. Figure 9.8 represents a multitape TM. A move depends on the current state and k tape symbols under k tape heads.

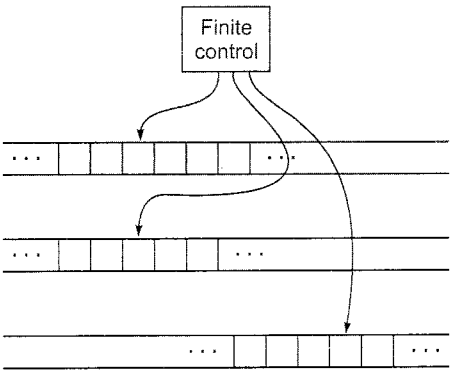


Fig. 9.8 Multitape Turing machine.

In a typical move:

(i) M enters a new state.
(ii) On each tape, a new symbol is written in the cell under the head.
(iii) Each tape head moves to the left or right or remains stationary. The heads move independently: some move to the left, some to the right and the remaining heads do not move.

The initial ID has the initial state q_0, the input string w in the first tape (input tape), empty strings of b's in the remaining $k - 1$ tapes. An accepting ID has a final state, some strings in each of the k tapes.

Theorem 9.1 Every language accepted by a multitape TM is acceptable by some single-tape TM (that is, the standard TM).

Proof Suppose a language L is accepted by a k-tape TM M. We simulate M with a single-tape TM with $2k$ tracks. The second, fourth, . . ., $(2k)$th tracks hold the contents of the k-tapes. The first, third, . . ., $(2k - 1)$th tracks hold a head marker (a symbol say X) to indicate the position of the respective tape head. We give an 'implementation description' of the simulation of M with a single-tape TM M_1. We give it for the case $k = 2$. The construction can be extended to the general case.

Figure 9.9 can be used to visualize the simulation. The symbols A_2 and B_5 are the current symbols to be scanned and so the headmarker X is above the two symbols.

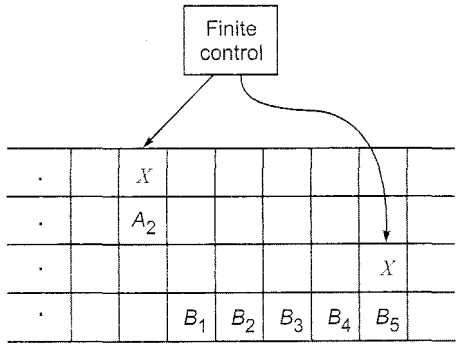


Fig. 9.9 Simulation of multitape TM.

Initially the contents of tapes 1 and 2 of M are stored in the second and fourth tracks of M_1. The headmarkers of the first and third tracks are at the cells containing the first symbol.

To simulate a move of M, the $2k$-track TM M_1 has to visit the two headmarkers and store the scanned symbols in its control. Keeping track of the headmarkers visited and those to be visited is achieved by keeping a count and storing it in the finite control of M_1. Note that the finite control of M_1 has also the information about the states of M and its moves. After visiting both head markers, M_1 knows the tape symbols being scanned by the two heads of M.

Now M_1 revisits each of the headmarkers:

(i) It changes the tape symbol in the corresponding track of M_1 based on the information regarding the move of M corresponding to the state (of M) and the tape symbol in the corresponding tape M.
(ii) It moves the headmarkers to the left or right.
(iii) M_1 changes the state of M in its control.

This is the simulation of a single move of M. At the end of this, M_1 is ready to implement its next move based on the revised positions of its headmarkers and the changed state available in its control.

M_1 accepts a string w if the new state of M, as recorded in its control at the end of the processing of w, is a final state of M.

Definition 9.3 Let M be a TM and w an input string. The running time of M on input w, is the number of steps that M takes before halting. If M does not halt on an input string w, then the running time of M on w is infinite.

Note: Some TMs may not halt on all inputs of length n. But we are interested in computing the running time, only when the TM halts.

Definition 9.4 The time complexity of TM M is the function $T(n)$, n being the input size, where $T(n)$ is defined as the maximum of the running time of M over all inputs w of size n.

Theorem 9.2 If M_1 is the single-tape TM simulating multitape TM M, then the time taken by M_1 to simulate n moves of M is $O(n^2)$.

Proof Let M be a k-tape TM. After n moves of M, the head markers of M_1 will be separated by $2n$ cells or less. (At the worst, one tape movement can be to the left by n cells and another can be to the right by n cells. In this case the tape headmarkers are separated by $2n$ cells. In the other cases, the 'gap' between them is less). To simulate a move of M, the TM M_1 must visit all the k headmarkers. If M starts with the leftmost headmarker, M_1 will go through all the headmarkers by moving right by at most $2n$ cells. To simulate the change in each tape, M_1 has to move left by at most $2n$ cells; to simulate changes in k tapes, it requires at most two moves in the reverse direction for each tape.

Thus the total number of moves by M_1 for simulating one move of M is atmost $4n + 2k$. ($2n$ moves to right for locating all headmarkers, $2n + 2k$ moves to the left for simulating the change in the content of k tapes.) So the number of moves of M_1 for simulating n moves of M is $n(4n + 2k)$. As the constant k is independent of n, the time taken by M_1 is $O(n^2)$.

9.7.2 Nondeterministic Turing Machines

In the case of standard Turing machines (hereafter we refer to this machine as deterministic TM), $\delta(q_1, a)$ was defined (for some elements of $Q \times \Gamma$) as an element of $Q \times \Gamma \times \{L, R\}$. Now we extend the definition of δ. In a nondeterministic TM, $\delta(q_1, a)$ is defined as a subset of $Q \times \Gamma \times \{L, R\}$.

Definition 9.5 A nondeterministic Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ where

1. Q is a finite nonempty set of states
2. Γ is a finite nonempty set of tape symbols
3. $b \in \Gamma$ is called the blank symbol
4. Σ is a nonempty subset of Γ, called the set of input symbols. We assume that $b \notin \Sigma$.
5. q_0 is the initial state
6. $F \subseteq Q$ is the set of final states
7. δ is a partial function from $Q \times \Gamma$ into the power set of $Q \times \Gamma \times \{L, R\}$.

Note: If $q \in Q$ and $x \in \Gamma$ and $\delta(q, x) = \{(q_1, y_1, D_1), (q_2, y_2, D_2), \ldots, (q_n, y_n, D_n)\}$ then the NTM can chose any one of the actions defined by (q_i, y_i, D_i) for $i = 1, 2, \ldots, n$.

We can also express this in terms of $\vdash$ relation. If $\delta(q, x) = \{(q_i, y_i, D_i) | i = 1, 2, \ldots, n\}$ then the ID $zqxw$ can change to any one of the n IDs specified by the n-element set $\delta(q, x)$.

Suppose $\delta(q, x) = \{(q_1, y_1, L), (q_2, y_2, R), (q_3, y_3, L)\}$. Then

$$z_1z_2 \cdots z_kqxz_{k+1} \cdots z_n \vdash z_1z_2 \cdots z_{k-1}q_1z_ky_1z_{k+1} \cdots z_n$$

or

$$z_1z_2 \cdots z_kqxz_{k+1} \cdots z_n \vdash z_1z_2 \cdots z_ky_2q_2z_{k+1} \cdots z_n$$

or

$$z_1z_2 \cdots z_kqxz_{k+1} \cdots z_n \vdash z_1z_2 \cdots z_{k-1}q_3z_ky_3z_{k+1} \cdots z_n.$$

So on reading the input symbol, the NTM M whose current ID is $z_1z_2 \ldots z_kqxz_{k+1} \ldots z_n$ can change to any one of the three IDs given earlier.

Remark When $\delta(q, x) = \{(q_i, y_i, D_i) \mid i = 1, 2, \ldots, n\}$ then NTM chooses any one of the n triples totally (that is, it cannot take a state from one triple, another tape symbol from a second triple and a third D(L or R) from a third triple, etc.

Definition 9.6 $w \in \Sigma^*$ is accepted by a nondeterministic TM M if $q_0w \vdash^* xq_fy$ for some final state q_f.

The set of all strings accepted by M is denoted by $T(M)$.

Note: As in the case of NDFA, an ID of the form xqy (for some $q \notin F$) may be reached as the result of applying the input string w. But w is accepted by M as long as there is some sequence of moves leading to an ID with an accepting state. It does not matter that there are other sequences of moves leading to an ID with a nonfinal state or TM halts without processing the entire input string.

Theorem 9.3 If M is a nondeterministic TM, there is a deterministic TM M_1 such that $T(M) = T(M_1)$.

Proof We construct M_1 as a multitape TM. Each symbol in the input string leads to a change in ID. M_1 should be able to reach all IDs and stop when an ID containing a final state is reached. So the first tape is used to store IDs of M as a sequence and also the state of M. These IDs are separated by the symbol * (included as a tape symbol). The current ID is known by marking an x along with the ID-separator * (The symbol * marked with x is a new tape symbol.) All IDs to the left of the current one have been explored already and so can be ignored subsequently. Note that the current ID is decided by the current input symbol of w.

Figure 9.10 illustrates the deterministic TM M_1.

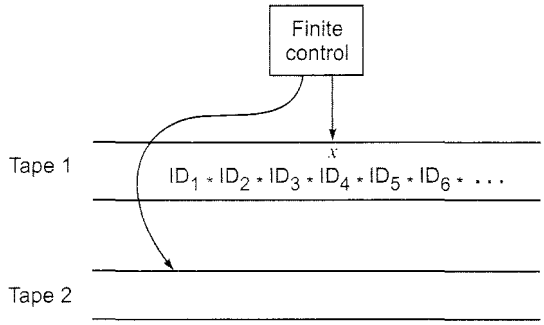


Fig. 9.10 The deterministic TM simulating M.

To process the current ID, M_1 performs the following steps.

1. M_1 examines the state and the scanned symbol of the current ID. Using the knowledge of moves of M stored in the finite control of M_1, M_1 checks whether the state in the current ID is an accepting state of M. In this case M_1 accepts and stops simulating M.

2. If the state q say in the current ID $xqay$ is not an accepting state of M_1 and $\delta(q, a)$ has k triples, M_1 copies the ID $xqay$ in the second tape and makes k copies of this ID at the end of the sequence of IDs in tape 2.
3. M_1 modifies these k IDs in tape 2 according to the k choices given by $\delta(q, a)$.
4. M_1 returns to the marked current ID, erases the mark x and marks the next ID-separator * with x (to the * which is to the left of the next ID to be processed). Then M_1 goes back to step 1.

M_1 stops when an accepting state of M is reached in step 1.

Now M_1 accepts an input string w only when it is able to find that M has entered an accepting state, after a finite number of moves. This is clear from the simulated sequence of moves of M_1 (ending in step 1)

We have to prove that M_1 will eventually reach an accepting ID (that is, an ID having an accepting state of M) if M enters an accepting ID after n moves. Note each move of M is simulated by several moves of M_1.

Let m be the maximum number of choices that M has for various (q, a)'s. (It is possible to find m since we have only finite number of pairs in $Q \times \Gamma$.) So for each initial ID of M, there are at most m IDs that M can reach after one move, at most m^2 IDs that M can reach after two moves, and so on. So corresponding to n moves of M, there are at most $1 + m + m^2 + \cdots + m^n$ moves of M_1. Hence the number of IDs to be explored by M_1 is at most nm^n.

We assume that M_1 explores these IDs. These IDs have a tree structure having the initial ID as its root. We can apply breadth-first search of the nodes of the tree (that is, the nodes at level 1 are searched, then the nodes at level 2, and so on.) If M reaches an accepting ID after n moves, then M_1 has to search atmost nm^n IDs before reaching an accepting ID. So, if M accepts w, then M_1 also accepts w (eventually). Hence $T(M) = T(M_1)$.

9.8 THE MODEL OF LINEAR BOUNDED AUTOMATON

This model is important because (a) the set of context-sensitive languages is accepted by the model, and (b) the infinite storage is restricted in size but not in accessibility to the storage in comparison with the Turing machine model. It is called the *linear bounded automaton* (LBA) because a linear function is used to restrict (to bound) the length of the tape.

In this section we define the model of linear bounded automaton and develop the relation between the linear bounded automata and context-sensitive languages. It should be noted that the study of context-sensitive languages is important from practical point of view because many compiler languages lie between context-sensitive and context-free languages.

A linear bounded automaton is a nondeterministic Turing machine which has a single tape whose length is not infinite but bounded by a linear function

of the length of the input string. The models can be described formally by the following set format:

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

All the symbols have the same meaning as in the basic model of Turing machines with the difference that the input alphabet Σ contains two special symbols ¢ and \$. ¢ is called the left-end marker which is entered in the left-most cell of the input tape and prevents the R/W head from getting off the left end of the tape. \$ is called the right-end marker which is entered in the right-most cell of the input tape and prevents the R/W head from getting off the right end of the tape. Both the endmarkers should not appear on any other cell within the input tape, and the R/W head should not print any other symbol over both the endmarkers.

Let us consider the input string w with $|w| = n - 2$. The input string w can be recognized by an LBA if it can also be recognized by a Turing machine using no more than kn cells of input tape, where k is a constant specified in the description of LBA. The value of k does not depend on the input string but is purely a property of the machine. Whenever we process any string in LBA, we shall assume that the input string is enclosed within the endmarkers ¢ and \$. The above model of LBA can be represented by the block diagram of Fig. 9.11. There are two tapes: one is called the input tape, and the other, working tape. On the input tape the head never prints and never moves to the left. On the working tape the head can modify the contents in any way, without any restriction.

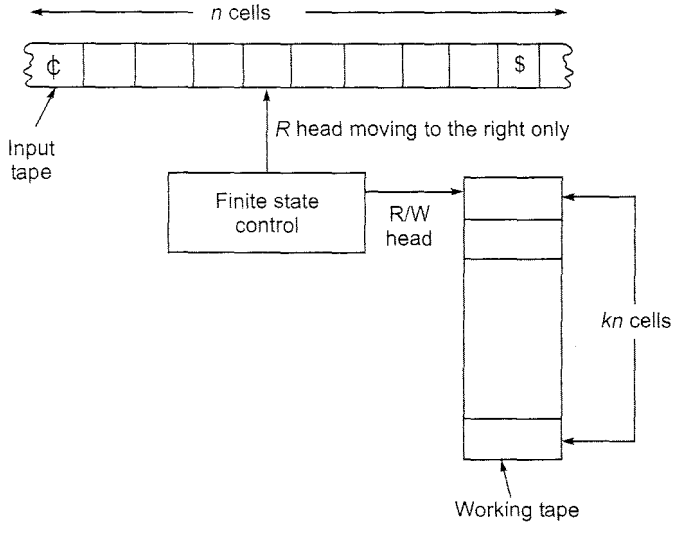


Fig. 9.11 Model of linear bounded automaton.

In the case of LBA, an ID is denoted by (q, w, k), where $q \in Q$, $w \in \Gamma$ and k is some integer between 1 and n. The transition of IDs is similar except

that k changes to $k - 1$ if the R/W head moves to the left and to $k + 1$ if the head moves to the right.

The language accepted by LBA is defined as the set

$$\{w \in (\Sigma - \{¢, \$\})^* | (q_0, ¢w\$, 1) \vdash^* (q, \alpha, i)$$

for some $q \in F$ and for some integer i between 1 and $n\}$.

Note: As a null string can be represented either by the absence of input string or by a completely blank tape, an LBA may accept the null string.

9.8.1 Relation Between LBA and Context-sensitive Languages

The set of strings accepted by nondeterministic LBA is the set of strings generated by the context-sensitive grammars, excluding the null strings. Now we give an important result:

If L is a context-sensitive language, then L is accepted by a linear bounded automaton. The converse is also true.

The construction and the proof are similar to those for Turing machines with some modifications.

9.9 TURING MACHINES AND TYPE 0 GRAMMARS

In this section we construct a type 0 grammar generating the set accepted by a given Turing machine M. The productions are constructed in two steps. In step 1 we construct productions which transform the string $[q_1¢\,w\$]$ into the string $[q_2b]$, where q_1 is the initial state, q_2 is an accepting state, ¢ is the left-endmarker, and \$ is the right-endmarker. The grammar obtained by applying step 1 is called the *transformational grammar*. In step 2 we obtain inverse production rules by reversing the productions of the transformational grammar to get the required type 0 grammar G. The construction is in such a way that w is accepted by M if and only if w is in $L(G)$.

9.9.1 Construction of a Grammar Corresponding to TM

For understanding the construction, we have to note that a transition of ID corresponds to a production. We enclose IDs within brackets. So acceptance of w by M corresponds to the transformation of initial ID $[q_1\,¢\,w\,\$]$ into $[q_2b]$. Also, the 'length' of ID may change if the R/W head reaches the left-end or the right-end, i.e. when the left-hand side or the right-hand side bracket is reached. So we get productions corresponding to transition of IDs with (i) no change in length, and (ii) change in length. We assume that the transition table is given.

We now describe the construction which involves two steps:

Step 1 (i) *No change in length of* IDs: (a) *Right move.* a_kRq_l corresponding to q_i-row and a_j-column leads to the production

$$q_ia_j \rightarrow a_kq_l$$

(b) *Left move.* a_kLq_l corresponding to q_i-row and a_j-column yields several productions

$$a_mq_ia_j \rightarrow q_la_ma_k \qquad \text{for all } a_m \in \Gamma$$

(ii) *Change in length of* IDs: (a) *Left-end.* a_kLq_l corresponding to q_i-row and a_j-column gives

$$[q_ia_j \rightarrow [q_lba_k$$

When b occurs next to the left-bracket, it can be deleted. This is achieved by including the production $[b \rightarrow [$.

(b) *Right-end.* When b occurs to the left of], it can be deleted. This is achieved by the production

$$a_jb] \rightarrow a_j] \qquad \text{for all } a_j \in \Gamma$$

When the R/W head moves to the right of], the length increases. Corresponding to this we have a production

$$q_i] \rightarrow q_ib] \qquad \text{for all } q_i \in Q$$

(iii) *Introduction of endmarkers.* For introducing endmarkers for the input string, the following productions are included:

$$a_i \rightarrow [q_1 \text{¢} a_i \qquad \text{for } a_i \in \Gamma,\ a_i \neq b$$

$$a_i \rightarrow a_i\$] \qquad \text{for all } a_i \in \Gamma,\ a_i \neq b$$

For removing the brackets from $[q_2b]$, we include the production

$$[q_2b] \rightarrow S$$

Recall that q_1 and q_2 are the initial and final states, respectively.

Step 2 To get the required grammar, reverse the arrows of the productions obtained in step 1. The productions we get can be called *inverse productions.* The new grammar is called the *generative grammar.* We illustrate the construction with an example.

EXAMPLE 9.11

Consider the TM described by the transition table given in Table 9.9. Obtain the inverse production rules.

Solution

In this example, q_1 is both initial and final.

Step 1 (i) *Productions corresponding to right moves*

$$q_1\text{¢} \rightarrow \text{¢}q_1, \qquad q_11 \rightarrow bq_2, \qquad q_21 \rightarrow bq_1 \tag{9.1}$$

(ii) (a) *Productions corresponding to left-end*

$$[b \rightarrow [\tag{9.2}$$

(b) *Productions corresponding to rig't-end*

$$bb] \rightarrow b], \quad 1b] \rightarrow 1], \quad q_1] \rightarrow q_1 b], \quad q_2] \rightarrow q_2 b] \tag{9.3}$$

(iii) $$1 \rightarrow [q_1 ¢1, \quad 1 \rightarrow 1\$], \quad [q_1 b] \rightarrow S \tag{9.4}$$

TABLE 9.9 Transition Table for Example 9.11

Present state	¢	*b*	1
$\rightarrow(q_1)$	$¢Rq_1$		bRq_2
q_2			bRq_1

Step 2 The inverse productions are obtained by reversing the arrows of the productions (9.1)–(9.4).

$$¢q_1 \rightarrow q_1 ¢, \quad bq_2 \rightarrow q_1 1, \quad bq_1 \rightarrow q_2 1$$

$$[\rightarrow [b, \; b] \rightarrow bb], \quad 1] \rightarrow 1b]$$

$$q_1 b \rightarrow q_1], \quad q_2 b \rightarrow q_2], \quad [q_1 ¢1 \rightarrow 1$$

$$1\$] \rightarrow 1, \quad S \rightarrow [q_1 b]$$

Thus we have shown that there exists a type 0 grammar corresponding to a Turing machine. The converse is also true (we are not proving this), i.e. given a type 0 grammar G, there exists a Turing machine accepting $L(G)$. Actually, the class of recursively enumerable sets, the type 0 languages, and the class of sets accepted by TM are one and the same. We have shown that there exists a recursively enumerable set which is not a context-sensitive language (see Theorem 4.4). As a recursive set is recursively enumerable, Theorem 4.4 gives a type 0 language which is not type 1. Hence, $\mathcal{L}_{\text{csl}} \subset \mathcal{L}_0$ (cf Property 4, Section 4.3) is established.

9.10 LINEAR BOUNDED AUTOMATA AND LANGUAGES

A linear bounded automaton M accepts a string w if, after starting at the initial state with R/W head reading the left-endmarker, M halts over the right-endmarker in a final state. Otherwise, w is rejected.

The production rules for the generative grammar are constructed as in the case of Turing machines. The following additional productions are needed in the case of LBA.

$$a_i q_f \$ \rightarrow q_f \$ \qquad \text{for all } a_i \in \Gamma$$

$$¢q_f \$ \rightarrow ¢q_f, \qquad ¢q_f \rightarrow q_f$$

EXAMPLE 9.12

Find the grammar generating the set accepted by a linear bounded automaton M whose transition table is given in Table 9.10.

TABLE 9.10 Transition Table for Example 9.12

Present state	*Tape input symbol*			
	¢	$	0	1
$\rightarrow q_1$	$\not\!\!\text{¢}Rq_1$		$1Lq_2$	$0Rq_2$
q_2	$\not\!\!\text{¢}Rq_4$		$1Rq_3$	$1Lq_1$
q_3		$\$Lq_1$	$1Rq_3$	$1Rq_3$
(q_4)		Halt	$0Lq_4$	$0Rq_4$

Solution

Step 1 (A) (i) *Productions corresponding to right moves.* The seven right moves in Table 9.10 give the following productions:

$$\begin{aligned} &q_1\not\!\!\text{¢} \rightarrow \not\!\!\text{¢}q_1, \quad && q_30 \rightarrow 1q_3 \\ &q_11 \rightarrow 0q_2, && q_31 \rightarrow 1q_3 \\ &q_2\not\!\!\text{¢} \rightarrow \not\!\!\text{¢}q_4, && q_41 \rightarrow 0q_4 \\ &q_20 \rightarrow 1q_3 \end{aligned} \tag{9.5}$$

(ii) *Productions corresponding to left moves.* There are four left moves in Table 9.10. Each left move yields four productions (corresponding to the four tape symbols). These are:

(a) $1Lq_2$ corresponding to q_1-row and 0-column gives

$$\not\!\!\text{¢}q_10 \rightarrow q_2\not\!\!\text{¢}1,\ \$q_10 \rightarrow q_2\$1,\ 0q_10 \rightarrow q_201,\ 1q_10 \rightarrow q_211 \tag{9.6}$$

(b) $1Lq_1$ corresponding to q_1-row and 1-column yields

$$\not\!\!\text{¢}q_21 \rightarrow q_1\not\!\!\text{¢}1,\ \$q_21 \rightarrow q_1\$1,\ 0q_21 \rightarrow q_101,\ 1q_21 \rightarrow q_111 \tag{9.7}$$

(c) $\$Lq_1$ corresponding to q_3-row and \$-column gives

$$\not\!\!\text{¢}q_3\$ \rightarrow q_1\not\!\!\text{¢}\,\$,\ \$q_3\$ \rightarrow q_1\$\$,\ 0q_3\$ \rightarrow q_10\$,\ 1q_3\$ \rightarrow q_11\$ \tag{9.8}$$

(d) $0Lq_4$ corresponding to q_4-row and 0-column yields

$$\not\!\!\text{¢}q_40 \rightarrow q_4\not\!\!\text{¢}0,\ \$q_40 \rightarrow q_4\$0,\ 0q_40 \rightarrow q_400,\ 1q_40 \rightarrow q_410 \tag{9.9}$$

(B) There are no productions corresponding to change in length.

(C) The productions for introducing the endmarkers are

$$\begin{aligned} &\not\!\!\text{¢} \rightarrow [q_1\not\!\!\text{¢}\not\!\!\text{¢}, && \not\!\!\text{¢} \rightarrow \not\!\!\text{¢}\$] \\ &\$ \rightarrow [q_1\not\!\!\text{¢}\$, && \$ \rightarrow \$\$] \\ &0 \rightarrow [q_1\not\!\!\text{¢}0, && 0 \rightarrow 0\$] \\ &1 \rightarrow [q_1\not\!\!\text{¢}1, && 1 \rightarrow 1\$] \end{aligned} \tag{9.10}$$

$$[q_4] \rightarrow S \tag{9.11}$$

(D) The LBA productions are

$$\begin{aligned} &\text{¢}q_4\$ \to q_4\$, \qquad && \text{¢}q_4\$ \to \text{¢}q_4 \\ &\$q_4\$ \to q_4\$, && \text{¢}q_4 \to q_4 \\ &0q_4\$ \to q_4\$, \\ &1q_4\$ \to q_4\$ \end{aligned} \tag{9.12}$$

Step 2 The productions of the generative grammar are obtained by reversing the arrows of productions given by (9.5)–(9.12).

9.11 SUPPLEMENTARY EXAMPLES

EXAMPLE 9.13

Design a TM that copies strings of 1's.

Solution

We design a TM so that we have ww after copying $w \in \{1\}^*$. Define M by

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

where δ is defined by Table 9.11.

TABLE 9.11 Transition Table for Example 9.13

Present state	*Tape symbol*		
	1	b	a
q_0	q_0aR	q_1bL	—
q_1	$q_1 1L$	q_3bR	$q_2 1R$
q_2	$q_2 1R$	$q_1 1L$	—
q_3	—	—	—

The procedure is simple.

M replaces every 1 by the symbol a. Then M replaces the rightmost a by 1. It goes to the right end of the string and writes a 1 there. Thus M has added a 1 for the rightmost 1 in the input string w. This process can be repeated.

M reaches q_1 after replacing all 1's by a's and reading the blank at the end of the input string. After replacing a by 1, M reaches q_2. M reaches q_3 at the end of the process and halts. If $w = 1^n$, than we have 1^{2n} at the end of the computation. A sample computation is given below.

$$\begin{aligned} q_0 11 &\vdash aq_0 1 \vdash aaq_0 b \vdash aq_1 a \\ &\vdash a1q_2 b \vdash aq_1 11 \vdash q_1 a11 \\ &\vdash 1q_2 11 \vdash 11q_2 1 \vdash 111q_2 b \\ &\vdash 11q_2 11 \vdash 1q_1 111 \\ &\vdash q_1 1111 \vdash q_1 b1111 \vdash q_3 1111 \end{aligned}$$

EXAMPLE 9.14

Construct a TM to accept the set L of all strings over $\{0,1\}$ ending with 010.

Solution

L is certainly a regular set and hence a deterministic automaton is sufficient to recognize L. Figure 9.12 gives a DFA accepting L.

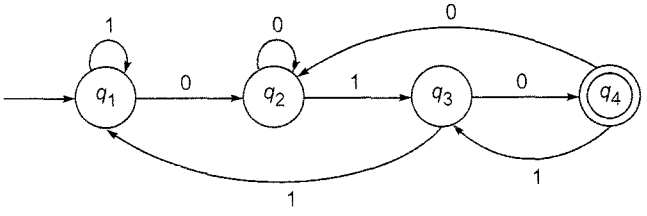


Fig. 9.12 DFA for Example 9.14.

Converting this DFA to a TM is simple. In a DFA M, the move is always to the right. So the TM's move will always be to the right. Also M reads the input symbol and changes state. So the TM M_1 does the same; it reads an input symbol, does not change the symbol and changes state. At the end of the computation, the TM sees the first blank b and changes to its final state. The initial ID of M_1 is q_0w. By defining $\delta(q_0, b) = (q_1, b, R)$, M_1 reaches the initial state of M. M_1 can be described by Fig. 9.13.

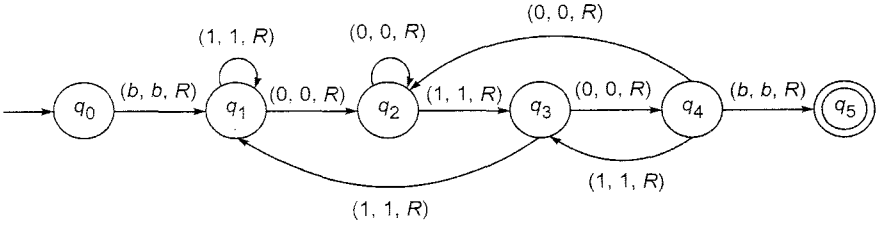


Fig. 9.13 TM for Example 9.14.

Note: q_5 is the unique final state of M_1. By comparing Figs. 9.12 and 9.13 it is easy to see that strings of L are accepted by M_1.

EXAMPLE 9.15

Design a TM that reads a string in $\{0, 1\}^*$ and erases the rightmost symbol.

Solution

The required TM M is given by

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, b\}, \delta, q_0, b, \{q_4\})$$

where δ is defined by

$$\delta(q_0, 0) = (q_1, 0, R) \qquad \delta(q_0, 1) = (q_1, 1, R) \qquad (R_1)$$

$$\delta(q_1, 0) = (q_1, 0, R) \qquad \delta(q_1, 1) = (q_1, 1, R) \qquad (R_2)$$

$$\delta(q_1, b) = (q_2, b, L) \qquad (R_3)$$

$$\delta(q_2, 0) = (q_3, b, L) \qquad \delta(q_2, 1) = (q_3, b, L) \qquad (R_4)$$

$$\delta(q_3, 0) = (q_3, 0, L) \qquad \delta(q_3, 1) = (q_3, 1, L) \qquad (R_5)$$

$$\delta(q_3, b) = (q_4, b, R) \qquad (R_6)$$

Let w be the input string. By (R_1) and (R_2), M reads the entire input string w. At the end, M is in state q_1. On seeing the blank to the right of w, M reaches the state q_2 and moves left. The rightmost string in w is erased (by (R_4)) and the state becomes q_3. Afterwards M moves to the left until it reaches the left-end of w. On seeing the blank b to the right of w, M changes its state to q_4, which is the final state of M. From the construction it is clear that the rightmost symbol of w is erased.

EXAMPLE 9.16

Construct a TM that accepts $L = \{0^{2^n} \mid n \geq 0\}$.

Solution

Let w be an input string in $\{0\}^*$. The TM accepting L functions as follows:

1. It writes b (blank symbol) on the leftmost 0 of the input string w. This is done to mark the left-end of w.
2. M reads the symbols of w from left to right and replaces the alternate 0's with x's.
3. If the tape contains a single 0 in step 2, M accepts w.
4. If the tape contains more than one 0 and the number of 0's is odd in step 2, M rejects w.
5. M returns the head to the left-end of the tape (marked by blank b in step 1).
6. M goes to step 2.

Each iteration of step 2 reduces w to half its size. Also whether the number of 0's seen is even or odd is known after step 2. If that number is odd and greater than 1, w cannot be 0^{2^n} (step 4). In this case M rejects w. If the number of 0's seen is 1 (step 3), M accepts w (In this case 0^{2^n} is reduced to 0 in successive stages of step 2).

We define M by

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_f, q_t\}, \{0\}, \{0, x, b\}, \delta, q_0, b, \{q_f\})$$

where δ is defined by Table 9.12.

TABLE 9.12 Transition Table for Example 9.16

Present state	Tape symbol		
	0	b	x
q_0	bRq_1	bRq_t	xRq_t
q_1	xRq_2	bRq_f	xRq_1
q_2	$0Rq_3$	bRq_4	xRq_2
q_3	xRq_2	bRq_6	xRq_3
q_4	$0Lq_4$	bRq_1	xLq_4
q_f	—	—	—
q_t	—	—	—

From the construction, it is apparent that the states are used to know whether the number of 0's read is odd or even.

We can see how M processes 0000.

$$\begin{aligned} q_0 0000 &\vdash bq_1 000 \vdash bxq_2 00 \vdash bxq_3 0 \vdash bx0xq_2 b \\ &\vdash bx0q_4 xb \vdash bxq_4 0xb \vdash bq_4 x0xb \vdash q_4 bx0xb \\ &\vdash bq_1 x0xb \vdash bxq_1 0xb \vdash bxxq_2 xb \vdash bxxxq_2 b \\ &\vdash bxxq_4 xb \vdash bxq_4 xxb \vdash bq_4 xxxb \vdash q_4 bxxxb \\ &\vdash bq_1 xxxb \vdash bxq_1 xxb \vdash bxxq_1 xb \vdash bxxxq_1 b \\ &\vdash bxxxbq_f. \end{aligned}$$

Hence M accepts w.

Also note that M always halts. If M reaches q_f, the input string w is accepted by M. If M reaches q_t, w is not accepted by M; in this case M halts in the trap state.

EXAMPLE 9.17

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

where δ is given by

$$\delta(q_0, 0) = (q_1, 1, R) \qquad (R_1)$$

$$\delta(q_1, 1) = (q_0, 0, R) \qquad (R_2)$$

$$\delta(q_1, b) = (q_2, b, R) \qquad (R_3)$$

Find $T(M)$.

Solution

Let $w \in T(M)$. As $\delta(q_0, 1)$ is not defined, w cannot start with 1. From (R_1) and (R_2), we can conclude that M starts from q_0 and comes back to q_0 after reaching 01.

So, $q_0(01)^n \overset{*}{\vdash} (10)^n q_0$. Also, $q_0 0b \vdash 1q_1 b \vdash 1bq_2$.

So, $(01)^n0 \in T(M)$. Also, $(01)^n0$ is the only string that makes M move from q_0 to q_2. Hence, $T(M) = \{(01)^n0 \mid n \geq 0\}$.

SELF-TEST

Choose the correct answer to Questions 1–10:

1. For the standard TM:
(a) $\Sigma = \Gamma$
(b) $\Gamma \subseteq \Sigma$
(c) $\Sigma \subseteq \Gamma$
(d) Σ is a proper subset of Γ.

2. In a standard TM, $\delta(q, a)$, $q \in Q$, $a \in \Gamma$ is
(a) defined for all $(q, a) \in Q \times \Gamma$
(b) defined for some, not necessarily for all $(q, a) \in Q \times \Gamma$
(c) defined for no element (q, a) of $Q \times \Gamma$
(d) a set of triples with more than one element.

3. If $\delta(q, x_i) = (p, y, L)$, then
(a) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-2}px_{i-1}yx_{i+1} \ldots x_n$
(b) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$
(c) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1 \ldots x_{i-3}px_{i-2}x_{i-1}yx_{i+1} \ldots x_n$
(d) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1 \ldots x_{i+1}pyx_{i+2} \ldots x_n$

4. If $\delta(q, x_i) = (p, y, R)$, then
(a) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$
(b) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_ipx_{i+1} \ldots x_n$
(c) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}px_ix_{i+1} \ldots x_n$
(d) $x_1x_2 \ldots x_{i-1}qx_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}ypx_{i+1} \ldots x_n$

5. If $\delta(q, x_1) = (p, y, L)$, then
(a) $qx_1x_2 \ldots x_n \vdash pyx_2 \ldots x_n$
(b) $qx_1x_2 \ldots x_n \vdash ypx_2 \ldots x_n$
(c) $qx_1x_2 \ldots x_n \vdash pbx_1 \ldots x_n$
(d) $qx_1x_2 \ldots x_n \vdash pbx_2 \ldots x_n$

6. If $\delta(q, x_n) = (p, y, R)$, then
(a) $x_1 \ldots x_{n-1}qx_n \vdash pyx_2x_3 \ldots x_n$
(b) $x_1 \ldots x_{n-1}qx_n \vdash^* pyx_2x_3 \ldots x_n$
(c) $x_1 \ldots x_{n-1}qx_n \vdash x_1x_2 \ldots x_{n-1}ypb$
(d) $x_1 \ldots x_{n-1}qx_n \vdash^* x_1x_2 \ldots x_{n-1}ypb$

7. For the TM given in Example 9.6:
(a) $q_01b11 \vdash^* bq_f11bb1$
(b) $q_01b11 \vdash bq_f11bb1$
(c) $q_01b11 \vdash 1q_0b111$
(d) $q_01b11 \vdash q_2b11bb1$

8. For the TM given in Example 9.4:
   (a) 011 is accepted by M
   (b) 001 is accepted by M
   (c) 00 is accepted by M
   (d) 0011 is accepted by M.
9. For the TM given in Example 9.5:
   (a) 1 is accepted by M
   (b) 11 is accepted by M
   (c) 111 is accepted by M
   (d) 11111 is accepted by M
10. In a standard TM $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ the blank symbol b is
   (a) in $\Sigma - \Gamma$
   (b) in $\Gamma - \Sigma$
   (c) $\Gamma \cap \Sigma$
   (d) none of these

EXERCISES

9.1 Draw the transition diagram of the Turing machine given in Table 9.1.

9.2 Represent the transition function of the Turing machine given in Example 9.2 as a set of quintuples.

9.3 Construct the computation sequence for the input $1b11$ for the Turing machine given in Example 9.5.

9.4 Construct the computation sequence for strings 1213, 2133, 312 for the Turing machine given in Example 9.8.

9.5 Explain how a Turing machine can be considered as a computer of integer functions (i.e. as one that can compute integer functions; we shall discuss more about this in Chapter 11).

9.6 Design a Turing machine that converts a binary string into its equivalent unary string.

9.7 Construct a Turing machine that enumerates $\{0^n1^n \mid n \geq 1\}$.

9.8 Construct a Turing machine that can accept the set of all even palindromes over $\{0, 1\}$.

9.9 Construct a Turing machine that can accept the strings over $\{0, 1\}$ containing even number of 1's.

9.10 Design a Turing machine to recognize the language $\{a^nb^nc^m \mid n, m \geq 1\}$.

9.11 Design a Turing machine that can compute proper subtraction, i.e. $m \dot{-} n$, where m and n are positive integers. $m \dot{-} n$ is defined as $m - n$ if $m > n$ and 0 if $m \leq n$.

10 Decidability and Recursively Enumerable Languages

In this chapter the formal definition of an algorithm is given. The problem of decidability of various class of languages is discussed. The theorem on halting problem of Turing machine is proved.

10.1 THE DEFINITION OF AN ALGORITHM

In Section 4.4, we gave the definition of an algorithm as a procedure (finite sequence of instructions which can be mechanically carried out) that terminates after a finite number of steps for any input. The earliest algorithm one can think of is the Euclidean algorithm, for computing the greatest common divisor of two natural numbers. In 1900, the mathematician David Hilbert, in his famous address at the International congress of mathematicians in Paris, averred that every definite mathematical problem must be susceptible for an exact settlement either in the form of an exact answer or by the proof of the impossibility of its solution. He identified 23 mathematical problems as a challenge for future mathematicians; only ten of the problems have been solved so far.

Hilbert's tenth problem was to devise 'a process according to which it can be determined by a finite number of operations', whether a polynomial over Z has an integral root. (He did not use the word 'algorithm' but he meant the same.) This was not answered until 1970.

The formal definition of algorithm emerged after the works of Alan Turing and Alanzo Church in 1936. The Church–Turing thesis states that any algorithmic procedure that can be carried out by a human or a computer, can also be carried out by a Turing machine. Thus the Turing machine arose as an ideal theoretical model for an algorithm. The Turing machine provided a machinery to mathematicians for attacking the Hilberts' tenth problem. The problem can be restated as follows: does there exist a TM that can accept a

polynomial over n variables if it has an integral root and reject the polynomial if it does not have one.

In 1970, Yuri Matijasevic, after studying the work of Martin Davis, Hilary Putnam and Julia Robinson showed that no such algorithm (Turing machine) exists for testing whether a polynomial over n variables has integral roots. Now it is universally accepted by computer scientists that Turing machine is a mathematical model of an algorithm.

10.2 DECIDABILITY

We are familiar with the recursive definition of a function or a set. We also have the definitions of recursively enumerable sets and recursive sets (refer to Section 4.4). The notion of a recursively enumerable set (or language) and a recursive set (or language) existed even before the dawn of computers.

Now these terms are also defined using Turing machines. When a Turing machine reaches a final state, it 'halts.' We can also say that a Turing machine M halts when M reaches a state q and a current symbol a to be scanned so that $\delta(q, a)$ is undefined. There are TMs that never halt on some inputs in any one of these ways. So we make a distinction between the languages accepted by a TM that halts on all input strings and a TM that never halts on some input strings.

Definition 10.1 A language $L \subseteq \Sigma^*$ is recursively enumerable if there exists a TM M, such that $L = T(M)$.

Definition 10.2 A language $L \subseteq \Sigma^*$ is recursive if there exists some TM M that satisfies the following two conditions.

(i) If $w \in L$ then M accepts w (that is, reaches an accepting state on processing w) and halts.

(ii) If $w \notin L$ then M eventually halts, without reaching an accepting state.

Note: Definition 10.2 formalizes the notion of an 'algorithm'. An algorithm, in the usual sense, is a well-defined sequence of steps that always terminates and produces an answer. The Conditions (i) and (ii) of Definition 10.2 assure us that the TM always halts, accepting w under Condition (i) and not accepting under Condition (ii). So a TM, defining a recursive language (Definition 10.2) always halts eventually just as an algorithm eventually terminates.

A problem with only two answers Yes/No can be considered as a language L. An instance of the problem with the answer 'Yes' can be considered as an element of the corresponding language L; an instance with answer 'No' is considered as an element not in L.

Definition 10.3 A problem with two answers (Yes/No) is decidable if the corresponding language is recursive. In this case, the language L is also called *decidable*.

Definition 10.4 A problem/language is undecidable if it is not decidable.

Note: A decidable problem is called a solvable problem and an undecidable problem an unsolvable problem by some authors.

10.3 DECIDABLE LANGUAGES

In this section we consider the decidability of regular and context-free languages.

First of all, we consider the problem of testing whether a deterministic finite automaton accepts a given input string w.

Definition 10.5

$$A_{DFA} = \{(B, w) \mid B \text{ accepts the input string } w\}$$

Theorem 10.1 A_{DFA} is decidable.

Proof To prove the theorem, we have to construct a TM that always halts and also accepts A_{DFA}. We describe the TM M using high level description (refer to Section 9.5). Note that a DFM B always ends in some state of B after n transitions for an input string of length n.

We define a TM M as follows:

1. Let B be a DFA and w an input string. (B, w) is an input for the Turing machine M.
2. Simulate B and input w in the TM M.
3. If the simulation ends in an accepting state of B, then M accepts w. If it ends in a nonaccepting state of B, then M rejects w.

We can discuss a few implementation details regarding steps 1, 2 and 3 above. The input (B, w) for M is represented by representing the five components $Q, \Sigma, \delta, q_0, f$ by strings of Σ^* and input string $w \in \Sigma^*$. M checks whether (B, w) is a valid input. If not, it rejectes (B, w) and halts. If (B, w) is a valid input, M writes the initial state q_0 and the leftmost input symbol of w. It updates the state using δ and then reads the next symbol in w. This explains step 2.

If the simulation ends in an accepting state w, then M accepts (B, w). Otherwise, M rejects (B, w). This is the description of step 3.

It is evident that M accepts (B, w) if and only if w is accepted by the DFA B. ∎

Definition 10.6

$$A_{CFG} = \{(G, w) \mid \text{the context-free grammar } G \text{ accepts the input string } w\}$$

Theorem 10.2 A_{CFG} is decidable.

Proof We convert a CFG into Chomsky normal form. Then any derivation of w of length k requires $2k - 1$ steps if the grammar is in CNF (refer to Example 6.18). So for checking whether the input string w of length k is

in $L(G)$, it is enough to check derivations in $2k - 1$ steps. We know that there are only finitely many derivations in $2k - 1$ steps. Now we design a TM M that halts as follows.

1. Let G be a CFG in Chomsky normal form and w an input string. (G, w) is an input for M.
2. If $k = 0$, list all the single-step derivations. If $k \neq 0$, list all the derivations with $2k - 1$ steps.
3. If any of the derivations in step 2 generates the given string w, M accepts (G, w). Otherwise M rejects.

The implementation of steps 1–3 is similar to the steps in Theorem 10.1. (G, w) is represented by representing the four components V_N, Σ, P, S of G and input string w. The next step of the derivation is got by the production to be applied.

M accepts (G, w) if and only if w is accepted by the CFG G.

In Theorem 4.3, we proved that a context-sensitive language is recursive. The main idea of the proof of Theorem 4.3 was to construct a sequence $\{W_0, W_1, \ldots, W_k\}$ of subsets of $(V_N \cup \Sigma)^*$, that terminates after a finite number of iterations. The given string $w \in \Sigma^*$ is in $L(G)$ if and only if $w \in W_k$. With this idea in mind we can prove the decidability of the context-sensitive language. ∎

Definition 10.7 $A_{CSG} = \{(G, w) \mid$ the context-sensitive grammar G accepts the input string $w\}$.

Theroem 10.3 A_{CSG} is decidable.

Proof The proof is a modification of the proof of Theorem 10.2. In Theorem 10.2, we considered derivations with $2k - 1$ steps for testing whether an input string of length k was in $L(G)$. In the case of context-sensitive grammar we construct $W_i = \{\alpha \in (V_N \cup \Sigma)^* \mid S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ in i or fewer steps and $|\alpha| \leq n\}$. There exists a natural number k such that $W_k = W_{k+1} = W_{k+2} = \ldots$ (refer to proof of Theorem 4.3).

So $w \in L(G)$ if and only if $w \in W_k$. The construction of W_k is the key idea used in the construction of a TM accepting A_{CSG}. Now we can design a Turing machine M as follows:

1. Let G be a context-sensitive grammar and w an input string of length n. Then (G, w) is an input for TM.
2. Construct $W_0 = \{S\}$. $W_{i+1} = W_i \cup \{\beta \in (V_N \cup \Sigma)^* \mid$ there exists $\alpha_i \in W_i$ such that $\alpha \Rightarrow \beta$ and $|\beta| \leq n\}$. Continue until $W_k = W_{k+1}$ for some k. (This is possible by Theorem 4.3.)
3. If $w \in W_k$, $w \in L(G)$ and M accepts (G, w); otherwise M rejects (G, w). ∎

Note: If $\mathcal{L}_d$ denotes the class of all decidable languages over Σ, then

$$\mathcal{L}_{rl} \subseteq \mathcal{L}_{cfl} \subseteq \mathcal{L}_{csl} \subseteq \mathcal{L}_d$$

10.4 UNDECIDABLE LANGUAGES

In this section we prove the existence of languages that are not recursively enumerable and address the undecidability of recursively enumerable languages.

Theorem 10.4 There exists a language over Σ that is not recursively enumerable.

Proof A language L is recursively enumerable if there exists a TM M such that $L = T(M)$. As Σ is finite, Σ^* is countable (that is, there exists a one-to-one correspondence between Σ^* and N).

As a Turing machine M is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$ and each member of the 7-tuple is a finite set, M can be encoded as a string. So the set I of all TMs is countable.

Let $\mathcal{L}$ be the set of all languages over Σ. Then a member of $\mathcal{L}$ is a subset of Σ^* (Note that Σ^* is infinite even though Σ is finite). We show that $\mathcal{L}$ is uncountable (that is, an infinite set not in one-to correspondence with N).

We prove this by contradiction. If $\mathcal{L}$ were countable then $\mathcal{L}$ can be written as a sequence $\{L_1, L_2, L_3, \ldots\}$. We write Σ^* as a sequence $\{w_1, w_2, w_3, \ldots\}$. So L_i can be represented as an infinite binary sequence $x_{i1}x_{i2}x_{i3}\ldots$ where

$$x_{ij} = \begin{cases} 1 & \text{if } w_j \in L_i \\ 0 & \text{otherwise} \end{cases}$$

Using this representation we write L_i as an infinite binary sequence.

$$\begin{aligned} L_1 &: x_{11}x_{12}x_{13} \ldots x_{1j} \ldots \\ L_2 &: x_{21}x_{22}x_{23} \ldots x_{2j} \ldots \\ &\vdots \qquad\qquad\quad \vdots \\ L_i &: x_{i1}x_{i2}x_{i3} \ldots x_{ij} \ldots \end{aligned}$$

Fig. 10.1 Representation of $\mathcal{L}$.

We define a subset L of Σ^* by the binary sequence $y_1y_2y_3 \ldots$ where $y_i = 1 - x_{ii}$. If $x_{ii} = 0$, $y_i = 1$ and if $x_{ii} = 1$, $y_i = 0$. Thus according to our assumption the subset L of Σ^* represented by the infinite binary sequence $y_1y_2y_3 \ldots$ should be L_k for some natural number k. But $L \neq L_k$, since $w_k \in L$ if and only if $w_k \notin L_k$. This contradicts our assumption that $\mathcal{L}$ is countable. Therefore $\mathcal{L}$ is uncountable. As I is countable, $\mathcal{L}$ should have some members not corresponding to any TM in I. This proves the existence of a language over Σ tnat is not recursively enumerable. ∎

Definition 10.8 $A_{TM} = \{(M, w) \mid \text{The TM } M \text{ accepts } w\}$.

Theorem 10.5 A_{TM} is undecidable.

Proof We can prove that A_{TM} is recursively enumerable. Construct a TM U as follows:

(M, w) is an input to U. Simulate M on w. If M enters an accepting state, U accepts (M, w). Hence A_{TM} is recursively enumerable. We prove that A_{TM} is undecidable by contradiction. We assume that A_{TM} is decidable by a TM H that eventually halts on all inputs. Then

$$H(M, w) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

We construct a new TM D with H as subroutine. D calls H to determine what M does when it receives the input $\langle M \rangle$, the encoded description of M as a string. Based on the received information on $(M, \langle M \rangle)$, D rejects M if M accepts $\langle M \rangle$ and accepts M if M rejects $\langle M \rangle$. D is described as follows:

1. $\langle M \rangle$ is an input to D, where $\langle M \rangle$ is the encoded string representing M.
2. D calls H to run on $(M, \langle M \rangle)$
3. D rejects $\langle M \rangle$ if H accepts $(M, \langle M \rangle)$ and accepts $\langle M \rangle$ if H rejects $(M, \langle M \rangle)$.

Now step 3 can be described as follows:

$$D(\langle M \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$$

Let us look at the action of D on the input $\langle D \rangle$. According to the construction of D,

$$D(\langle D \rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D \rangle \end{cases}$$

This means D accepts $\langle D \rangle$ if D does not accept $\langle D \rangle$, which is a contradiction. Hence ATM is undecidable. ∎

The Turing machine U used in the proof of Theorem 10.5 is called the *universal Turing machine*. U is called universal since it is simulating any other Turing machine.

10.5 HALTING PROBLEM OF TURING MACHINE

In this section we introduce the reduction technique. This technique is used to prove the undecidability of halting problem of Turing machine.

We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

For example, if A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Note: If A is reducible to B and B is decidable then A is decidable. If A is reducible to B and A is undecidable, then B is undecidable.

Theorem 10.6 $HALT_{TM} = \{(M, w) \mid$ The Turing machine M halts on input $w\}$ is undecidable.

Proof We assume that $HALT_{TM}$ is decidable, and get a contradiction. Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a TM M_2 as follows:

1. For M_2, (M, w) is an input.
2. The TM M_1 acts on (M, w).
3. If M_1 rejects (M, w) then M_2 rejects (M, w).
4. If M_1 accepts (M, w), simulate the TM M on the input string w until M halts.
5. If M has accepted w, M_2 accepts (M, w); otherwise M_2 rejects (M, w).

When M_1 accepts (M, w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q', a)$ is undefined till some symbol a in w is reached. In the first case (the first alternative of step 5) M_2 accepts (M, w). In the second case (the second alternative of step 5) M_2 rejects (M, w).

It follows from the definition of M_2 that M_2 halts eventually.

Also, $$T(M_2) = \{(M, w) \mid \text{The Turing machine accepts } w\}$$
$$= A_{TM}$$

This is a contradiction since A_{TM} is undecidable. ∎

10.6 THE POST CORRESPONDENCE PROBLEM

The Post Correspondence Problem (PCP) was first introduced by Emil Post in 1946. Later, the problem was found to have many applications in the theory of formal languages. The problem over an alphabet Σ belongs to a class of yes/no problems and is stated as follows: Consider the two lists $x = (x_1 \ldots x_n)$, $y = (y_1 \ldots y_n)$ of nonempty strings over an alphabet $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exist $i_1, \ldots, i_m$, where $1 \leq i_j \leq n$, such that

$$x_{i_1} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$$

Note: The indices i_j's need not be distinct and m may be greater than n. Also, if there exists a solution to PCP, there exist infinitely many solutions.

EXAMPLE 10.1

Does the PCP with two lists $x = (b, bab^3, ba)$ and $y = (b^3, ba, a)$ have a solution?

Solution

We have to determine whether or not there exists a sequence of substrings of x such that the string formed by this sequence and the string formed by the sequence of corresponding substrings of y are identical. The required sequence is given by $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$, i.e. (2, 1, 1,3), and $m = 4$. The corresponding strings are

bab^3	b	b	ba	=	ba	b^3	b^3	a
x_2	x_1	x_1	x_3		y_2	y_1	y_1	y_3

Thus the PCP has a solution.

EXAMPLE 10.2

Prove that PCP with two lists $x = (01, 1, 1)$, $y = (01^2, 10, 1^1)$ has no solution.

Solution

For each substring $x_i \in x$ and $y_i \in y$, we have $|x_i| < |y_i|$ for all i. Hence the string generated by a sequence of substrings of x is shorter than the string generated by the sequence of corresponding substrings of y. Therefore, the PCP has no solution.

Note: If the first substring used in PCP is always x_1 and y_1, then the PCP is known as the *Modified Post Correspondence Problem.*

EXAMPLE 10.3

Explain how a Post Correspondence Problem can be treated as a game of dominoes.

Solution

The PCP may be thought of as a game of dominoes in the following way: Let each domino contain some x_i in the upper-half, and the corresponding substring of y in the lower-half. A typical domino is shown as

x_i	upper-half
y_i	lower-half

The PCP is equivalent to placing the dominoes one after another as a sequence (of course repetitions are allowed). To win the game, the same string should appear in the upper-half and in the lower-half. So winning the game is equivalent to a solution of the PCP.

We state the following theorem by Emil Post without proof.

Theorem 10.7 The PCP over Σ for $|\Sigma| \geq 2$ is unsolvable.

It is possible to reduce the PCP to many classes of two outputs (yes/no) problems in formal language theory. The following results can be proved by the reduction technique applied to PCP.

1. If L_1 and L_2 are any two context-free languages (type 2) over an alphabet Σ and $|\Sigma| \geq 2$, there is no algorithm to determine whether or not
   (a) $L_1 \cap L_2 = \emptyset$,
   (b) $L_1 \cap L_2$ is a context-free language,
   (c) $L_1 \subseteq L_2$, and
   (d) $L_1 = L_2$.
2. If G is a context-sensitive grammar (type 1), there is no algorithm to determine whether or not
   (a) $L(G) = \emptyset$,
   (b) $L(G)$ is infinite, and
   (c) $x_0 \in L(G)$ for a fixed string x_0.
3. If G is a type 0 grammar, there is no algorithm to determine whether or not any string $x \in \Sigma^*$ is in $L(G)$.

10.7 SUPPLEMENTARY EXAMPLES

EXAMPLE 10.4

If L is a recursive language over Σ, show that $\bar{L}$ ($\bar{L}$ is defined as $\Sigma^* - L$) is also recursive.

Solution

As L is recursive, there is a Turing machine M that halts and $T(M) = L$. We have to construct a TM M_1, such that $T(M_1) = \bar{L}$ and M_1 eventually halts.

M_1 is obtained by modifying M as follows:

1. Accepting states of M are made nonaccepting states of M_1.
2. Let M_1 have a new state q_f. After reaching q_f, M_1 does not move in further transitions.
3. If q is a nonaccepting state of M and $\delta(q, x)$ is not defined, add a transition from q to q_f for M_1.

As M halts, M_1 also halts. (If M reaches an accepting state on w, then M_1 does not accept w and halts and conversely.)

Also M_1 accepts w if and only if M does not accept w. So I is recursive.

EXAMPLE 10.5

If L and $\overline{L}$ are both recursively enumerable, show that L and $\overline{L}$ are recursive.

Solution

Let M_1 and M_2 be two TMs such that $L = T(M_1)$ and $\overline{L} = T(M_2)$. We construct a new two-tape TM M that simulates M_1 on one tape and M_2 on the other.

If the input string w of M is in L, then M_1 accepts w and we declare that M accepts w. If $w \in \overline{L}$, then M_2 accepts w and we declare that M halts without accepting. Thus in both cases, M eventually halts. By the construction of M it is clear that $T(M) = T(M_1) = L$. Hence L is recursive. We can show that $\overline{L}$ is recursive, either by applying Example 10.4 or by interchanging the roles of M_1 and M_2 in defining acceptance by M.

EXAMPLE 10.6

Show that $\overline{A}_{TM}$ is not recursively enumerable.

Solution

We have already seen that A_{TM} is recursively enumerable (by Theorem 10.5). If $\overline{A}_{TM}$ were also recursively enumerable, then A_{TM} is recursive (by Example 10.5). This is a contradiction since A_{TM} is not recursive by Theorem 10.5. Hence $\overline{A}_{TM}$ is not recursively enumerable.

EXAMPLE 10.7

Show that the union of two recursively enumerable languages is recursively enumerable and the union of two recursive languages is recursive.

Solution

Let L_1 and L_2 be two recursive languages and M_1, M_2 be the corresponding TMs that halt. We design a TM M as a two-tape TM as follows:

1. w is an input string to M.
2. M copies w on its second tape.
3. M simulates M_1 on the first tape. If w is accepted by M_1, then M accepts w.
4. M simulates M_2 on the second tape. If w is accepted by M_2, then M accepts w.

M always halts for any input w.

Thus $L_1 \cup L_2 = T(M)$ and hence $L_1 \cup L_2$ is recursive.

If L_1 and L_2 are recursively enumerable, then the same conclusion gives a proof for $L_1 \cup L_2$ to be recursively enumerable. As M_1 and M_2 need not halt, M need not halt.

SELF-TEST

1. What is the difference between a recursive language and a recursively enumerable language?
2. The DFA M is given by

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

where δ is defined by the transition Table 10.1.

TABLE 10.1 Transition Table for Self-Test 2

State	0	1
$\rightarrow(q_0)$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Answer the following:
(a) Is $(M, 001101)$ in A_{DFA}?
(b) Is $(M, 01010101)$ in A_{DFA}?
(c) Does $M \in A_{DFA}$?
(d) Find w such that $(M, w) \notin A_{DFA}$.

3. What do you mean by saying that the halting problem of TM is undecidable?
4. Describe A_{DFA}, A_{CFG}, A_{CSG}, A_{TM}, and $HALT_{TM}$.
5. Give one language from each of $\mathcal{L}_{rl}$, $\mathcal{L}_{cfl}$, $\mathcal{L}_{csl}$.
6. Give a language
   (a) which is in $\mathcal{L}_{csl}$ but not in $\mathcal{L}_{rl}$
   (b) which is in $\mathcal{L}_{cfl}$ but not in $\mathcal{L}_{csl}$
   (c) which is in $\mathcal{L}_{cfl}$ but not in $\mathcal{L}_{rl}$.

EXERCISES

10.1 Describe the Euclid's algorithm for finding the greatest common divisor of two natural numbers.

10.2 Show that $A_{NDFA} = \{(B, w) \mid B \text{ is an } N_{DFA} \text{ and } B \text{ accepts } w\}$ is decidable.

10.3 Show that $E_{DFA} = \{M \mid M \text{ is a } D_{FA} \text{ and } T(M) = \emptyset\}$ is decidable.

10.4 Show that $EQ_{DFA} = \{(A, B) \mid A \text{ and } B \text{ are DFAs and } T(A) = T(B)\}$ is decidable

10.5 Show that E_{CFG} is decidable (E_{CFG} is defined in a way similar to that of E_{DFA}).

10.6 Give an example of a language that is not recursive but recursively enumerable.

10.7 Do there exist languages that are not recursively enumerable?

10.8 Let L be a language over Σ. Show that only one of the following are possible for L and $\overline{L}$.
(a) Both L and $\overline{L}$ are recursive.
(b) Neither L nor $\overline{L}$ is recursive.
(c) L is recursively enumerable but $\overline{L}$ is not.
(d) $\overline{L}$ is recursively enumerable but L is not.

10.9 What is the difference between A_{TM} and $HALT_{TM}$?

10.10 Show that the set of all real numbers between 0 and 1 is uncountable. (A set S is uncountable if S is infinite and there is no one-to-one correspondence between S and the set of all natural numbers.)

10.11 Why should one study undecidability?

10.12 Prove that the recursiveness problem of type 0 grammar is unsolvable.

10.13 Prove that there exists a Turing machine M for which the halting problem is unsolvable.

10.14 Show that there exists a Turing machine M over $\{0, 1\}$ and a state q_m such that there is no algorithm to determine whether or not M will enter the state q_m when it begins with a given ID.

10.15 Prove that the problem of determining whether or not a TM over $\{0,1\}$ will ever print the symbol 1, with a given tape configuration, is unsolvable.

10.16 (a) Show that $\{x \mid x \text{ is a set and } x \notin x\}$ is not a set. (Note that this seems to be well-defined. This is one version of Russell's paradox.)
(b) A village barber shaves those who do not shave themselves but no others. Can he achieve his goal? For example, who is to shave the barber? (This is a popular version of Russell's paradox.)

Hints: (a) Let $S = \{x \mid x \text{ be a set and } x \notin x\}$. If S were a set, then $S \in S$ or $S \notin S$. If $S \notin S$ by the 'definition' of S, then $S \in S$. On the other hand, if $S \in S$ by the 'definition' of S, then $S \notin S$. Thus we can neither assert that $S \notin S$ nor $S \in S$. (This is Russell's paradox.) Therefore, S is not a set.

(b) Let $S = \{x \mid x \text{ be a person and } x \text{ does not shave himself}\}$. Let b denote the barber. Examine whether $b \in S$. (The argument is similar to that given for (a).) It will be instructive to read the proof of HP of Turing machines and this example, in order to grasp the similarity.

10.17 Comment on the following: "We have developed an algorithm so complicated that no Turing machine can be constructed to execute the algorithm no matter how much (tape) space and time is allowed."

10.18 Prove that PCP is solvable if $|\Sigma| = 1$.

10.19 Let $x = (x_1 \ldots x_n)$ and $y = (y_1 \ldots y_n)$ be two lists of nonempty strings over Σ and $|\Sigma| \geq 2$. (i) Is PCP solvable for $n = 1$? (ii) Is PCP solvable for $n = 2$?

10.20 Prove that the PCP with $\{(01, 011), (1, 10), (1, 11)\}$ has no solution. (Here, $x_1 = 01$, $x_2 = 1$, $x_3 = 1$, $y_1 = 011$, $y_2 = 10$, $y_3 = 11$.)

10.21 Show that the PCP with $S = \{(0, 10), (1^20, 0^3), (0^21, 10)\}$ has no solution. [*Hint:* No pair has common nonempty initial substring.]

10.22 Does the PCP with $x = (b^3, ab^2)$ and $y = (b^3, bab^3)$ have a solution?

10.23 Find at least three solutions to PCP defined by the dominoes:

1
111

10
0

10111
10

10.24 (a) Can you simulate a Turing machine on a general-purpose computer? Explain.

(b) Can you simulate a general-purpose computer on a Turing machine? Explain.

11 Computability

In this chapter we shall discuss the class of primitive recursive functions—a subclass of partial recursive functions. The Turing machine is viewed as a mathematical model of a partial recursive function.

11.1 INTRODUCTION AND BASIC CONCEPTS

In Chapters 5, 7 and 9, we considered automata as the accepting devices. In this chapter we will study automata as the computing machines. The problem of finding out whether a given problem is 'solvable' by automata reduces to the evaluation of functions on the set of natural numbers or a given alphabet by mechanical means.

We start with the definition of partial and total functions.

A partial function f from X to Y is a rule which assigns to every element of X at most one element of Y.

A total function from X to Y is a rule which assigns to every element of X a unique element of Y. For example, if R denotes the set of all real numbers, the rule f from R to itself given by $f(r) = +\sqrt{r}$ is a partial function since $f(r)$ is not defined as a real number when r is negative. But $g(r) = 2r$ is a total function from R to itself. (Note that all the functions considered in the earlier chapters were total functions.)

In this chapter we consider total functions from X^k to X, where $X = \{0, 1, 2, 3, \ldots\}$ or $X = \{a, b\}^*$. Throughout this chapter we denote $(0, 1, 2, \ldots)$ by N and (a, b) by Σ. (Recall that X^k is the set of all k-tuples of elements of X.) For example, $f(m, n) = m - n$ defines a partial function from N to itself as $f(m, n)$ is not defined when $m - n < 0$; $g(m, n) = m + n$ defines a total function from N to itself.

Remark A partial or total function f from X^k to X is also called a function of k variables and denoted by $f(x_1, x_2, \ldots, x_k)$. For example, $f(x_1, x_2) = 2x_1 + x_2$ is a function of two variables: $f(1, 2) = 4$, 1 and 2 are called arguments and 4 is called a value. $g(w_1, w_2) = w_1w_2$ is a function of two variables ($w_1w_2 \in \Sigma^*$); $g(ab, aa) = abaa$, ab, aa are called arguments and $abaa$ is a value.

11.2 PRIMITIVE RECURSIVE FUNCTIONS

In this section we construct primitive recursive functions over N and Σ. We define some initial functions and declare them as primitive recursive functions. By applying certain operations on the primitive recursive functions obtained so far, we get the class of primitive recursive functions.

11.2.1 INITIAL FUNCTIONS

The initial functions over N are given in Table 11.1. In particular,

$$S(4) = 5, \quad Z(7) = 0$$

$$U_2^3(2, 4, 7) = 4, \quad U_1^3(2, 4, 7) = 2, \quad U_3^3(2, 4, 7) = 7$$

TABLE 11.1 Initial Functions Over N

Zero function Z defined by $Z(x) = 0$.
Successor function S defined by $S(x) = x + 1$.
Projection function U_i^n defined by $U_i^n(x_1, \ldots, x_n) = x_i$.

Note: As $U_1^1(x) = x$ for every x in N, U_1^1 is simply the identity function. So U_i^n is also termed a generalized identity function.

The initial functions over Σ are given in Table 11.2. In particular,

$$\text{nil } (abab) = \Lambda$$

$$\text{cons } a(abab) = aabab$$

$$\text{cons } b(abab) = babab$$

Note: We note that cons $a(x)$ and cons $b(x)$ simply denote the concatenation of the 'constant' string a and x and the concatenation of the constant string b and x.

TABLE 11.2 Initial Functions Over $\{a, b\}$

nil $(x) = \Lambda$
cons $a(x) = ax$
cons $b(x) = bx$

In the following definition, we introduce an operation on functions over X.

Definition 11.1 If $f_1, f_2, \ldots, f_k$ are partial functions of n variables and g is a partial function of k variables, then the composition of g with $f_1, f_2, \ldots, f_k$ is a partial function of n variables defined by

$$g(f_1(x_1, x_1, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_k(x_1, x_2, \ldots, x_n))$$

If, for example, f_1, f_2 and f_3 are partial functions of two variables and g is a partial function of three variables, then the composition of g with f_1, f_2, f_3 is given by $g(f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2))$.

EXAMPLE 11.1

Let $f_1(x, y) = x + y$, $f_2(x, y) = 2x$, $f_3(x, y) = xy$ and $g(x, y, z) = x + y + z$ be functions over N. Then

$$\begin{aligned} g(f_1(x, y), f_2(x, y), f_3(x, y)) &= g(x + y, 2x, xy) \\ &= x + y + 2x + xy \end{aligned}$$

Thus the composition of g with f_1, f_2, f_3 is given by a function h:

$$h(x, y) = x + y + 2x + xy$$

Note: Definition 11.1 generalizes the composition of two functions. The concept is useful where a number of outputs become the inputs for a subsequent step of a program.

The composition of g with $f_1, \ldots, f_n$ is total when $g, f_1, f_2, \ldots, f_n$ are total. The function given in Example 11.1 is total as f_1, f_2, f_3 and g are total.

EXAMPLE 11.2

Let $f_1(x, y) = x - y$, $f_2(x, y) = y - x$ and $g(x, y) = x + y$ be functions over N. The function f_1 is defined only when $x \geq y$ and f_2 is defined only when $y \geq x$. So f_1 and f_2 are defined only when $x = y$. Hence when $x = y$,

$$g(f_1(x, y), f_2(x, y)) = g(x - x, x - x) = g(0, 0) = 0$$

Thus the composition of g with f_1 and f_2 is defined only for (x, x), where $x \in N$.

EXAMPLE 11.3

Let $f_1(x_1, x_2) = x_1x_2$, $f_2(x_1, x_2) = \Lambda$, $f_3(x_1, x_2) = x_1$, and $g(x_1, x_2, x_3) = x_2x_3$ be functions over Σ. Then

$$g(f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2)) = g(x_1x_2, \Lambda, x_1) = \Lambda x_1 = x_1$$

So the composition of g with f_1, f_2, f_3 is given by a function h, where $h(x_1, x_2) = x_1$.

The next definition gives a mechanical process of computing a function.

Definition 11.2 A function $f(x)$ over N is defined by recursion if there exists a constant k (a natural number) and a function $h(x, y)$ such that

$$f(0) = k, \qquad f(n + 1) = h(n, f(n)) \tag{11.1}$$

By induction on n, we can define $f(n)$ for all n. As $f(0) = k$, there is basis for induction. Once $f(n)$ is known, $f(n + 1)$ can be evaluated by using (11.1).

EXAMPLE 11.4

Define $n!$ by recursion.

Solution

$f(0) = 1$ and $f(n + 1) = h(n, f(n))$, where $h(x, y) = S(x) * y$.

The above definition can be generalized for $f(x_1, x_2, \ldots, x_n, x_{n+1})$. We fix n variables in $f(x_1, x_2, \ldots, x_{n+1})$, say, $x_1, x_2, \ldots, x_n$. We apply Definition 11.2 to $f(x_1, x_2, \ldots, x_n, y)$. In place of k we get a function $g(x_1, x_2, \ldots, x_n)$ and in place of $h(x, y)$, we obtain $h(x_1, x_2, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$.

Definition 11.3 A function f of $n + 1$ variables is defined by recursion if there exists a function g of n variables, and a function h of $n + 2$ variables, and f is defined as follows:

$$f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n) \tag{11.2}$$

$$f(x_1 \ldots x_n, y + 1) = h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)) \tag{11.3}$$

We may note that f can be evaluated for all arguments $(x_1, x_2, \ldots, x_n, y)$ by induction on y for fixed $x_1, x_2, \ldots, x_n$. The process is repeated for every $x_1, x_2, \ldots, x_n$.

Now we can define the primitive recursive functions over N.

11.2.2 Primitive Recursive Functions Over *N*

Definition 11.4 A total function f over N is called primitive recursive (i) if it is any one of the three initial functions, or (ii) if it can be obtained by applying composition and recursion a finite number of times to the set of initial functions.

EXAMPLE 11.5

Show that the function $f_1(x, y) = x + y$ is primitive recursive.

Solution

f_1 is a function of two variables. If we want f_1 to be defined by recursion, we need a function g of a single variable and a function h of three variables.

$$f_1(x, 0) = x + 0 = x$$

By comparing $f_1(x, 0)$ with L.H.S. of (11.2), we see that g can be defined by

$$g(x) = x = U_1^1(x)$$

Also, $f_1(x, y + 1) = x + (y + 1) = (x + y) + 1 = f_1(x, y) + 1$
By comparing $f_1(x, y + 1)$ with L.H.S. of (11.3), we have

$$h(x, y, f_1(x, y)) = f_1(x, y) + 1 = S(f_1(x, y)) = S(U_3^3(x, y, f_1(x, y)))$$

Define $h(x, y, z) = S(U_3^3(x, y, z))$. As $g = U_1^1$, it is an initial function. The function h is obtained from the initial functions U_3^3 and S by composition, and by recursion using g and h. Thus f_1 is obtained by applying composition and recursion a finite number of times to initial functions U_1^1, U_3^3 and S. So f_1 is primitive recursive.

Note: A total function is primitive recursive if it can be obtained by applying composition and recursion a finite number of times to primitive recursive functions $f_1, f_2, \ldots, f_m$. This is clear as each f_i is obtained by applying composition and recursion a finite number of times to initial functions.

EXAMPLE 11.6

The function $f_2(x, y) = x * y$ is primitive recursive.

Solution

As multiplication of two natural numbers is simply repeated addition, f_2 has to be primitive recursive. We prove this as follows:

$$f_2(x, 0) = 0, \qquad f_2(x, y + 1) = x * (y + 1) = f_2(x, y) + x$$

i.e. $f_2(x, y + 1) = f_1(f_2(x, y), x)$. Comparing these with (11.2) and (11.3), we can write

$f_2(x, 0) = Z(x)$ and $f_2(x, y + 1) = f_1(U_3^3(x, y, f_2(x, y)), U_1^3(x, y, f_2(x, y))))$

By taking $g = Z$ and h defined by

$$h(x, y, z) = f_1(U_3^3(x, y, z), U_1^3(x, y, z))$$

we see that f_2 is defined by recursion. As g and h are primitive recursive, f_2 is primitive recursive (by the above note).

EXAMPLE 11.7

Show that $f(x, y) = x^y$ is a primitive recursive function.

Solution

We define

$$f(x, 0) = 1$$

$$f(x, y + 1) = x * f(x, y)$$

$$= U_1^3(x, y, f(x, y)) * U_3^3(x, y, f(x, y))$$

Therefore, $f(x, y)$ is primitive recursive.

EXAMPLE 11.8

Show that the following functions are primitive recursive:

(a) The predecessor function $p(x)$ defined by

$$p(x) = x - 1 \quad \text{if } x \neq 0, \quad p(x) = 0 \quad \text{if } x = 0.$$

(b) The proper subtraction function $\dot{-}$ defined by

$$x \dot{-} y = x - y \quad \text{if } x \geq y \quad \text{and} \quad x \dot{-} y = 0 \quad \text{if } x < y.$$

(c) The absolute value function $|\ |$ given by

$$|x| = x \quad \text{if } x \geq 0, \quad |x| = -x \quad \text{if } x < 0.$$

(d) min (x, y), i.e. minimum of x and y.

Solution

(a) $p(0) = 0$ and $p(y + 1) = U_1^2(y, p(y))$

(b) $x \dot{-} 0 = x$ and $x \dot{-} (y + 1) = p(x \dot{-} y)$

(c) $|x - y| = (x \dot{-} y) + (y \dot{-} x)$

(d) $\min(x, y) = x \dot{-} (x \dot{-} y)$

The first function is defined by recursion using an initial function. So it is primitive recursive.

The second function is defined by recursion using the primitive recursive function p and so it is primitive recursive. Similarly, the last two functions are primitive recursive.

11.2.3 PRIMITIVE RECURSIVE FUNCTIONS OVER $\{a, b\}$

For constructing the primitive recursive function over $\{a, b\}$, the process is similar to that of function over N except for some minor modifications. It should be noted that Λ plays the role of 0 in (11.2) and ax or bx plays the role of $y + 1$ in (11.3). Recall that Σ denotes $\{a, b\}$.

Definition 11.5 A function $f(x)$ over Σ is defined by recursion if there exists a 'constant' string $w \in \Sigma^*$ and functions $h_1(x, y)$ and $h_2(x, y)$ such that

$$f(\Lambda) = w \tag{11.4}$$

$$f(ax) = h_1(x, f(x)) \tag{11.5}$$

$$f(bx) = h_2(x, f(x))$$

(h_1 and h_2 may be functions in one variable.)

Definition 11.6 A function $f(x_1, x_2, \ldots, x_n)$ over Σ is defined by recursion if there exist functions $g(x_1, \ldots, x_{n-1})$, $h_1(x_1, \ldots, x_{n+1})$, $h_2(x_1, \ldots, x_{n+1})$, such that

$$f(\Lambda, x_2, \ldots, x_n) = g(x_2, \ldots, x_n) \tag{11.6}$$

$$f(ax_1, x_2, \ldots, x_n) = h_1(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n)) \tag{11.7}$$

$$f(bx_1, x_2, \ldots, x_n) = h_2(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n))$$

(h_1 and h_2 may be functions of m variables, where $m < n + 1$.)

Now we can define the class of primitive recursive functions over Σ.

Definition 11.7 A total function f is primitive recursive (i) if it is any one of the three initial functions (given in Table 11.2), or (ii) if it can be obtained by applying composition and recursion a finite number of times to the initial functions.

In Example 11.9 we give some primitive recursive functions over Σ.

Note: As in the case of functions over N, a total function over Σ is primitive recursive if it is obtained by applying composition and recursion a finite number of times to primitive recursive function $f_1, f_2, \ldots, f_m$.

EXAMPLE 11.9

Show that the following functions are primitive recursive:

(a) Constant functions a and b (i.e. $a(x) = a$, $b(x) = b$)
(b) Identity function
(c) Concatenation
(d) Transpose
(e) Head function (i.e. head $(a_1a_2 \ldots, a_n) = a_1$)
(f) Tail function (i.e. tail $(a_1a_2 \ldots a_n) = a_2 \ldots, a_n$)
(g) The conditional function "if $x_1 \neq \Lambda$, then x_2 else x_3."

Solution

(a) As $a(x) = \text{cons } a \text{ (nil } (x))$, the function $a(x)$ is the composition of the initial function cons a with the initial function nil and is hence primitive recursive.

(b) Let us denote the identity function by id. Then,

$$\text{id}(\Lambda) = \Lambda$$

$$\text{id}(ax) = \text{cons } a(x)$$

$$\text{id}(bx) = \text{cons } b(x)$$

So id is defined by recursion using cons a and cons b. Therefore, the identity function is primitive recursive.

(c) The concatenation function can be defined by

$$\text{concat}(x_1, x_2) = x_1x_2$$

$$\text{concat}(\Lambda, x_2) = \text{id}(x_2)$$

$$\text{concat}(ax_1, x_2) = \text{cons } a \text{ (concat}(x_1, x_2))$$

$$\text{concat}(bx_1, x_2) = \text{cons } b \text{ (concat}(x_1, x_2))$$

So concat is defined by recursion using id, cons a and cons b. Therefore, concat is primitive recursive.

(d) The transpose function can be defined by $\text{trans}(x) = x^T$. Then

$$\text{trans}(\Lambda) = \Lambda$$

$$\text{trans}(ax) = \text{concat}(\text{trans}(x), a(x))$$

$$\text{trans}(bx) = \text{concat}(\text{trans}(x), b(x))$$

Therefore, trans(x) is primitive recursive.

(e) The head function head(x) satisfies

$$\text{head}(\Lambda) = \Lambda$$

$$\text{head}(ax) = a(x)$$

$$\text{head}(bx) = b(x)$$

Therefore, head(x) is primitive recursive.

(f) The tail function tail(x) satisfies

$$\text{tail}(\Lambda) = \Lambda$$

$$\text{tail}(ax) = \text{id}(x)$$

$$\text{tail}(bx) = \text{id}(x)$$

Therefore, tail(x) is primitive recursive.

(g) The conditional function can be defined by

$$\text{cond}(x_1, x_2, x_3) = \text{“if } x_1 \neq \Lambda \text{ then } x_2 \text{ else } x_3\text{”}$$

Then,

$$\text{cond}(\Lambda, x_2, x_3) = \text{id}(x_3)$$

$$\text{cond}(ax_1, x_2, x_3) = \text{id}(x_2)$$

$$\text{cond}(bx_1, x_2, x_3) = \text{id}(x_2)$$

Therefore, id(x_1, x_2, x_3) is primitive recursive.

11.3 RECURSIVE FUNCTIONS

By introducing one more operation on functions, we define the class of recursive functions, which includes the class of primitive recursive functions.

Definition 11.8 Let $g(x_1, x_2, \ldots, x_n, y)$ be a total function over N. g is a regular function if there exists some natural number y_0 such that $g(x_1, x_2, \ldots, x_n, y_0) = 0$ for all values $x_1, x_2, \ldots, x_n$ in N.

For instance, $g(x, y) = \min(x, y)$ is a regular function since $g(x, 0) = 0$ for all x in N. But $f(x, y) = |x - y|$ is not regular since $f(x, y) = 0$ only when $x = y$, and so we cannot find a fixed y such that $f(x, y) = 0$ for all x in N.

Definition 11.9 A function $f(x_1, x_2, \ldots, x_n)$ over N is defined from a total function $g(x_1, x_2, \ldots, x_n, y)$ by minimization if

(a) $f(x_1, x_2, \ldots, x_n)$ is the least value of all y's such that $g(x_1, x_2, \ldots, x_n, y) = 0$ if it exists. The least value is denoted by $\mu_y(g(x_1, x_2, \ldots, x_n, y) = 0)$.
(b) $f(x_1, x_2, \ldots, x_n)$ is undefined if there is no y such that $g(x_1, x_2 \ldots x_n, y) = 0$.

Note: In general, f is partial. But, if g is regular then f is total.

Definition 11.10 A function is recursive if it can be obtained from the initial functions by a finite number of applications of composition, recursion and minimization over regular functions.

Definition 11.11 A function is partial recursive if it can be obtained from the initial functions by a finite number of applications of composition, recursion and minimization.

EXAMPLE 11.10

$f(x) = x/2$ is a partial recursive function over N.

Solution

Let $g(x, y) = |2y - x|$, where $2y - x = 0$ for some y only when x is even. Let $f_1(x) = \mu_y(|2y - x| = 0)$. Then $f_1(x)$ is defined only for even values of x and is equal to $x/2$. When x is odd, $f_1(x)$ is not defined. f_1 is partial recursive. As $f(x) = x/2 = f_1(x)$, f is a partial recursive function.

The following example gives a recursive function which is not primitive recursive.

EXAMPLE 11.11

The Ackermann's function is defined by

$$A(0, y) = y + 1 \tag{11.8}$$

$$A(x + 1, 0) = A(x, 1) \tag{11.9}$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y)) \tag{11.10}$$

$A(x, y)$ can be computed for every (x, y), and hence $A(x, y)$ is total. The Ackermann's function is not primitive recursive but recursive.

EXAMPLE 11.12

Compute $A(1, 1)$, $A(2, 1)$, $A(1, 2)$, $A(2, 2)$.

Solution

$$
\begin{aligned}
A(1, 1) &= A(0 + 1, 0 + 1) \\
&= A(0, A(1, 0)) && \text{by (11.10)} \\
&= A(0, A(0, 1)) && \text{by (11.9)} \\
&= A(0, 2) && \text{by (11.8)} \\
&= 3 && \text{by (11.8)} \\
A(1, 2) &= A(0 + 1, 1 + 1) \\
&= A(0, A(1, 1)) && \text{by (11.10)} \\
&= A(0, 3) \\
&= 4 && \text{by (11.8)} \\
A(2, 1) &= A(1 + 1, 0 + 1) \\
&= A(1, A(2, 0)) && \text{by (11.10)} \\
&= A(1, A(1, 1)) && \text{by (11.9)} \\
&= A(1, 3) \\
&= A(0 + 1, 2 + 1) \\
&= A(0, A(1, 2)) && \text{by (11.10)} \\
&= A(0, 4) \\
&= 5 \\
A(2, 2) &= A(1 + 1, 1 + 1) \\
&= A(1, A(2, 1)) && \text{by (11.10)} \\
&= A(1, 5) \\
A(1, 5) &= A(0 + 1, 4 + 1) \\
&= A(0, A(1, 4)) && \text{by (11.10)} \\
&= 1 + A(1, 4) && \text{by (11.8)} \\
&= 1 + A(0 + 1, 3 + 1) \\
&= 1 + A(0, A(1, 3)) \\
&= 1 + 1 + A(1, 3) \\
&= 1 + 1 + 1 + A(1, 2) = 1 + 1 + 1 + 4 \\
&= 7
\end{aligned}
$$

As $A(2, 2) = A(1, 5)$, we have $A(2, 2) = 7$

So far we have dealt with recursive and partial recursive functions over N. We can define partial recursive functions over Σ using the primitive recursive predicates and the minimization process. As the process is similar, we will discuss it here.

The concept of recursion occurs in some programming languages when a procedure has a call to the same procedure for a different parameter. Such a procedure is called a recursive procedure. Certain programming languages like C, C++ allow recursive procedures.

11.4 PARTIAL RECURSIVE FUNCTIONS AND TURING MACHINES

In this section we prove that partial recursive functions introduced in the earlier sections are Turing-computable.

11.4.1 COMPUTABILITY

In mid 1930s, mathematicians and logicians were trying to rigorously define computability and algorithms. In 1934 Kurt Gödel pointed out that primitive recursive functions can be computed by a finite procedure (i.e. an algorithm). He also hypothesized that any function computable by a finite procedure can be specified by a recursive function. Around 1936, Turing and Church independently designed a 'computing machine' (later termed *Turing machine*) which can carry out a finite procedure.

For formalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional paper (instead of a two-dimensional paper as is usually done) which can be viewed as a tape divided into cells. He scans the cells one at a time and usually performs one of the three simple operations, namely (i) writing a new symbol in the cell he is scanning, (ii) moving to the cell left of the present cell, and (iii) moving to the cell right of the present cell. These observations led Turing to propose a computing machine. The Turing machine model we have introduced in Chapter 9 is based on these three simple operations but with slight variations. In order to introduce computability, we consider the Turing machine model due to Post. In the present model the transition function is represented by a set of quadruples (i.e. 4-tuples), whereas the transition function of the model we have introduced in Chapter 9 can be represented by a set of quintuples (5-tuples). For example, $\delta(q_i, a) = (q_j, \alpha, \beta)$ is represented by the quintuple $q_i a \alpha \beta q_j$. Using the model specifying the transition function in terms of quadruples, we define Turing-computable functions and prove that partially recursive functions are Turing-computable.

11.4.2 A Turing Model for Computation

As in the model introduced in Chapter 9, Q, q_0 and Γ denote the set of states, the initial state, and the set of tape symbols, respectively. The blank symbol b is in Γ. The only difference is in the transition function. In the present model the transition function represents only one of the following three basic operations:

(i) Writing a new symbol in the cell scanned
(ii) Moving to the left cell
(iii) Moving to the right cell

Each operation is followed by a change of state. Suppose the Turing machine M is in state q and scans a_i. If a_i is written and M enters q', then this basic operation is represented by the quadruple qa_ia_jq'. Similarly, the other two operations are represented by the quadruples qa_iLq' and qa_iRq'. Thus the transition function can be specified by a set P of quadruples. As in Chapter 9, we can define instantaneous descriptions, i.e. IDs.

Each quadruple induces a change of IDs. For example, qa_ia_jq' induces

$$\alpha qa_i\beta \vdash \alpha q'a_j\beta$$

The quadruple qa_iLq' induces

$$a_1a_2 \ldots a_{i-1}qa_i \ldots a_n \vdash a_1a_2 \ldots a_{i-2}q'a_{i-1}a_i \ldots a_n$$

and qa_iRq' induces

$$a_1 \ldots a_{i-1}qa_i \ldots a_n \vdash a_1 \ldots a_iq'a_{i+1} \ldots a_n$$

When we require M to perform some computation, we 'feed' the input by initial tape expression denoted by X. So q_0X is the initial ID for the given input. For computing with the given input X, the Turing machine processes X using appropriate quadruples in P. As a result, we have $q_0X = \text{ID}_1 \vdash \text{ID}_2 \vdash \ldots$. When an ID, say ID_n, is reached, which cannot be changed using any quadruple in P, M halts. In this case, ID_n is called a terminal ID. Actually, $aq_1\alpha\beta$ is a terminal ID if there is no quadruple starting with q_1a. The terminal ID is called the result of X and denoted by Res(X). The computed value corresponding to input X can be obtained by deleting the state appearing in it as also some more symbols from Res(X).

11.4.3 Turing-computable Functions

Before developing the concept of Turing-computable functions, let us recall Example 9.6. The TM developed in Example 9.6 concatenates two strings α and β. Initially, α and β appear on the input tape separated by a blank b. Finally, the concatenated string $\alpha\beta$ appears on the input tape. The same method can be adopted with slight modifications for computing $f(x_1, \ldots, x_m)$. Suppose we want to construct a TM which can compute $f(x_1, \ldots, x_m)$ over

N for given arguments $a_1, \ldots, a_m$. Initially, the input $a_1, a_2, \ldots, a_m$ appears on the input tape separated by markers $x_1, \ldots, x_m$. The computed value $f(a_1, \ldots, a_m)$, say, c appears on the input tape , once the computation is over. To locate c we need another marker, say y. The value c appears to the right of x_m and to the left of y. To make the construction simpler, we use the tally notation to represent the elements of N. In the tally notation, 0 is represented by a string of b's. A positive integer n is represented by a string consisting of n 1's. So the initial tape expression takes the form $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m by$. As a result of computation, the initial ID $q_0 1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m\ by$ is changed to a terminal ID of the form $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^c q' y$ for some $q' \in Q$. In fact, the position of q' in a terminal ID is immaterial and it can appear anywhere in Res(X). The computed value is found between x_m and y. Sometimes we may have to omit the leading b's.

We say that a function $f(x_1, \ldots, x_m)$ is Turing-computable for arguments $a_1, \ldots, a_m$ if there exists a Turing machine for which

$$q_0 1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m by \overset{*}{\vdash} \mathrm{ID}_n$$

where ID_n is a terminal ID containing $f(a_1, \ldots, a_m)$ to the left of y.

Our ultimate aim is to prove that partial recursive functions are Turing-computable. For this purpose, first of all we prove that the three initial primitive recursive functions are Turing-computable.

11.4.4 Construction of the Turing Machine That Can Compute the Zero Function Z

The zero function Z is defined as $Z(a_1) = 0$ for all $a_1 \geq 0$. So the initial tape expression can be taken as $X = 1^{a_1}x_1 by$. As we require the computed value $Z(a_1)$, namely 0, to appear to the left of y, we require the machine to halt without changing the input. (Note that 0 is represented by b in the tally notation.)

Thus we define a TM by taking $Q = \{q_0, q_1\}$, $\Gamma = \{b, 1, x_1, y\}$, $X = 1^{a_1}x_1 by$. P consists of $q_0 bRq_0$, $q_0 1Rq_0$, $q_0 x_1 x_1 q_1$. $q_0 bRq_0$ and $q_0 1Rq_0$ are used to move to the right until x_1 is encountered. $q_0 x_1 x_1 q_1$ enables the TM to enter the state q_1. M enters q_1 without altering the tape symbol. In terms of change of IDs, we have

$$q_0 1^{a_1}x_1 by \overset{*}{\vdash} 1^{a_1} q_0 x_1 by \vdash 1^{a_1} q_1 x_1 by$$

As there is no quadruple starting with q_1, M halts and Res(X) = $1^{a_1}q_1 x_1 by$. By deleting q_1 in Res(X), we get $1^{a_1}x_1 by$ (which is the same as X) yielding 0 (given by b).

Note: We can also represent the quadruples in a tabular form which is similar to the transition table obtained in Chapter 9. In this case we have to specify (i) the new symbol written, or (ii) the movement to the left (denoted by L), or (iii) the movement to the right (denoted by R). So we get Table 11.3.

TABLE 11.3 Representation of Quadruples

State	*b*	*1*	x_1	*y*
q_0	(R, q_0)	(R, q_0)	(x_1, q_1)	
q_1				

11.4.5 Construction of the Turing Machine for Computing—The Successor Function

The successor function S is defined by $S(a_1) = a_1 + 1$ for all $a_1 \geq 0$. So the initial tape expression can be taken as $X = 1^{a_1}x_1by$ (as in the case of the zero function). At the end of the computation, we require 1^{a_1+1} to appear to the left of y. Hence we define a TM by taking

$$Q = \{q_0, \ldots, q_9\}, \qquad \Gamma = \{b, 1, x_1, y\}, \qquad X = 1^{a_1}x_1 by$$

where P consists of

(i) q_0bRq_0, q_01bq_1, $q_0x_1Rq_6$
(ii) q_1bRq_1, q_11Rq_1, $q_1x_1Rq_1$, q_1y1q_2
(iii) q_21Rq_2, q_2byq_3,
(iv) q_3bLq_3, q_31Lq_3, q_3yLq_3, $q_3x_1Lq_4$
(v) q_41Lq_4, q_4b1q_5,
(vi) q_51Rq_0,
(vii) q_6bRq_6, q_61Rq_6, $q_6x_1Rq_6$, q_6yLq_7
(viii) q_71Lq_7, q_7b1q_8,
(ix) q_8bLq_8, q_81Lq_8, q_8yLq_8, $q_8x_1x_1q_9$.

The corresponding operations can be explained as follows:

(i) If M starts from the initial ID, the head replaces the first 1 it encounters by b. Afterwards the head moves to the right until it encounters y (as a result of q_01bq_1, q_1bRq_1, q_11Rq_1, $q_1x_1Rq_1$).
(ii) y is replaced by 1 and M enters q_2. Once the end of the input tape is reached, y is added to the next cell. M enters q_3 (q_1y1q_2, q_21Rq_2, q_2byq_3).
(iii) Then the head moves to the left and the state is not changed until x_1 is encountered (q_3yLq_3, q_3yLq_3, q_3bLq_3).
(iv) On encountering x_1, the head moves to the left and M enters q_4. Once again the head moves to the left till the left end of the input string is reached ($q_3x_1Lq_4$, q_11Lq_4).
(v) The leftmost blank (written in point (i)) is replaced by 1 and M enters q_5 (q_4b1q_5).

Thus at the end of operations (i)–(v), the input part remains unaffected but the first 1 is added to the left of y.

(vi) Then the head scans the second 1 of the input string and moves right, and M enters q_0 ($q_5 1 R q_0$).

Operations (i)–(vi) are repeated until all the 1's of the input part (i.e. in 1^{a_1}) are exhausted and 11 ... 1 (a_1 times) appear to the left of y. Now the present state is q_0, and the current symbol is x_1.

(vii) M in state q_0 scans x_1, moves right, and enters q_6. It continues to move to the right until it encounters y ($q_0 x_1 R q_6$, $q_6 b R q_6$, $q_6 1 R q_6$, $q_6 x_1 R q_6$).

(viii) On encountering y, the head moves to the left and M enters q_7, after which the head moves to the left until it encounters b appearing to the left of 1^{a_1} of the output part. This b is changed to 1, and M enters q_8 ($q_6 y L q_7$, $q_1 1 L q_7$, $q_7 b_1 q_8$).

(ix) Once M is in q_8, the head continues to move to the left and on scanning x_1, M enters q_9. As there is no quadruple starting with q_9, M halts ($q_8 b L q_8$, $q_8 1 L q_8$, $q_8 x_1 x_1 q_9$).

The machine halts, and the terminal ID is $1^{a_1} q_9 x_1 1^{a_1+1} y$. For example, let us compute $S(1)$. In this case the initial ID is $q_0 1 x_1 by$. As a result of the computation, we have the following moves:

$$
\begin{aligned}
& q_0 1 x_1 by \vdash q_1 b x_1 by \vdash b q_1 x_1 by \\
\vdash\ & b x_1 q_1 by \vdash b x_1 b q_1 y \vdash b x_1 b q_2 1 \\
\vdash\ & b x_1 b 1 q_2 b \vdash b x_1 b 1 q_3 y \vdash b x_1 b q_3 1 y \\
\vdash^*\ & b q_3 x_1 b 1 y \vdash q_4 b x_1 b 1 y \vdash q_5 1 x_1 b 1 y \\
\vdash^*\ & 1 q_6 x_1 b 1 y \vdash^* 1 x_1 b 1 q_6 y \vdash 1 x_1 b q_7 1 y \\
\vdash\ & 1 x_1 q_7 b 1 y \vdash 1 x_1 q_8 1 1 y \vdash 1 q_8 x_1 1 1 y \\
\vdash\ & 1 q_9 x_1 1 1 y
\end{aligned}
$$

Thus, M halts and $S(1) = 2$ (given by 11 to the left of y).

11.4.6 Construction of the Turing Machine for Computing the Projection U_i^m

Recall $U_i^m(a_1, \ldots, a_m) = a_i$. The initial tape expression can be taken as

$$X = 1^{a_1} x_1 1^{a_2} x_2 \ldots 1^{a_m} x_m by$$

We define a Turing machine by taking $Q = \{q_0, \ldots, q_8\}$

$\Gamma = \{b, 1, x_1, \ldots, x_m, y\}$. P consists of

$q_0 z R q_0$ for all $z \in \Gamma - \{x_i\}$

$q_0 x_i L q_1$, $q_1 b b q_8$, $q_1 1 b q_2$

$q_2 z R q_2$ for all $z \in \Gamma - \{y\}$

$q_2 y 1 q_3$, $q_3 1 R q_3$, $q_3 b y q_4$

q_4zLq_4 for all $z \in \Gamma - \{x_i\}$

$q_4x_iLq_5$, q_51Lq_5, q_5b1q_6, q_61Lq_7, q_71bq_2

q_7zRq_8 for all $z \in \Gamma - \{1\}$

The operations of M are as follows:

(i) M starts from the initial ID and the head moves to the right until it encounters x_i (q_0zRq_0).

(ii) On seeing x_i, the head moves to the left ($q_0x_iLq_1$).

(iii) The head replaces 1 (the rightmost 1 in 1^{a_i}) by b (q_11bq_2).

(iv) The head moves to the right until it encounters y and replaces y by 1 (q_2zRq_2, $z \in \Gamma - \{y\}$ and q_2y1q_3).

(v) On reaching the right end, the head scans b and replaces this b by y (q_3byq_4).

(vi) The head moves to the left until it scans the symbol b. This b is replaced by 1 (q_4zLq_4, $z \in \Gamma - \{x_i\}$, $q_4x_iLq_5$, q_5b1q_6).

(vii) The head moves to the left and one of the 1's in 1^{a_i} is replaced by b. M reaches q_2 (q_61Lq_7, q_71bq_2).
As a result of (i)–(vii), one of the 1's in 1^{a_i} is replaced by b and 1 is added to the left of y. Steps (iv)–(vii) are repeated for all 1's in 1^{a_i}.

(viii) On scanning x_{i-1}, the head moves to the right and M enters q_8 ($q_7x_{i-1}Rq_8$).

As there are no quadruples starting with q_8, the Turing machine M halts. When $i \neq 1$ and $a_i \neq 0$, the terminal ID is $1^{a_1}x_1 \ldots x_{i-1}q_81^{a_i}x_i \ldots x_nb1^{a_i}y$. For example, let us compute $U_2^3(1, 2, 1)$:

$$
\begin{aligned}
& q_01x_11 1x_21x_3by && \vdash^{*} 1x_1 11q_0x_21x_3by \\
\vdash\ & 1x_11q_11x_21x_3by && \vdash\ 1x_11q_2bx_21x_3by \\
\vdash^{*}\ & 1x_11bx_21x_3bq_2y && \vdash\ 1x_11bx_21x_3bq_31 \\
\vdash\ & 1x_11bx_21x_3b1q_3b && \vdash\ 1x_11bx_21x_3b1q_4y \\
\vdash^{*}\ & 1x_11bq_4x_21x_3b1y && \vdash\ 1x_11q_5bx_21x_3b1y \\
\vdash\ & 1x_11q_61x_21x_3b1y && \vdash\ 1x_1q_711x_21x_3b1y \\
\vdash\ & 1x_1q_2b1x_21x_3b1y &&
\end{aligned}
$$

From the above derivation, we see that

$$1x_11q_2bx_21x_3by \vdash^{*} 1x_1q_2b1x_21x_3b1y$$

Repeating the above steps, we get

$$1x_1q_2b1x_21x_3b1y \vdash^{*} 1x_1q_81x_21x_3b11y$$

It should be noted that this construction is similar to that for the successor function. While computing U_i^m, the head skips the portion of the input corresponding to a_j, $j \neq i$. For every 1 in 1^{a_i}, 1 is added to the left of y.

Thus we have shown that the three initial primitive recursive functions are Turing-computable. Next we construct Turing machines that can perform composition, recursion, and minimization.

11.4.7 Construction of the Turing Machine That Can Perform Composition

Let $f_1(x_1, x_2, \ldots, x_m), \ldots, f_k(x_1, \ldots, x_m)$ be Turing-computable functions. Let $g(y_1, \ldots, y_k)$ be Turing-computable. Let $h(x_1, \ldots, x_m) = g(f_1(x_1, \ldots, x_m) \ldots, f_k(x_1, \ldots, x_m))$. We construct a Turing machine that can compute $h(a_1, \ldots, a_m)$ for given arguments $a_1, \ldots, a_m$. This involves the following steps:

Step 1 Construct Turing machines $M_1, \ldots, M_k$ which can compute $f_1, \ldots, f_k$, respectively. For the TMs $M_1, \ldots, M_k$, let $\Gamma = \{1, b, x_1, x_2, \ldots, x_m, y\}$ and $X = 1^{a_1}x_1 \ldots 1^{a_m} x_m by$. But the number of states for these TMs will vary. Let $n_1 + 1, \ldots, n_k + 1$ be the number of states for $M_1, \ldots, M_k$, respectively. As usual, the initial state is q_0 and the states for M_i are $q_0, \ldots, q_{n_i}$. As in the earlier constructions, the set P_i of quadruples for M_i is constructed in such a way that there is no quadruple starting with q_{n_i}.

Step 2 Let $f_i(a_1, \ldots, a_m) = b_i$ for $i = 1, 2, \ldots, k$. At the end of step 1, we have M_i's and the computed values b_i's. As g is Turing-computable, we can construct a TM M_{k+1} which can compute $g(b_1, \ldots, b_k)$. For M_{k+1},

$$\Gamma = \{1, b, x'_1, \ldots, x'_m, y\}, \qquad X' = 1^{b_1}x'_1 \ldots 1^{a_m}x'_m by$$

(We use different markers for M_{k+1} so that the TM computing h to be constructed need not scan the inputs $a_1, \ldots, a_m$.) Let $n_{k+1} + 1$ be the number of states of M_{k+1}. As in the earlier constructions, M_{k+1} has no quadruples starting with q_{k+1}.

Step 3 At the end of step 2, we have TMs $M_1, \ldots, M_k, M_{k+1}$ which give $b_1, \ldots, b_m$ and $g(b_1, \ldots, b_k) = c$ (say), respectively. So we are able to compute $h(a_1, \ldots, a_m)$ using $k + 1$ Turing machines. Our objective is to construct a single TM M_{k+2} which can compute $h(a_1, \ldots, a_m)$. We outline the construction of M without giving the complete details of the encoding mechanism. For M, let

$$\Gamma = \{1, b, x_1, \ldots, x_m, x'_1, \ldots, x'_m, y\}$$

$$X = 1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m by$$

(i) In the beginning, M simulates M_1. As a result, the value $b_1 = f_1(a_1, \ldots, a_m)$ is obtained as output. Thus we get the tape expression $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^{b_1}y$ which is the same as that obtained by M_1 while halting. M does not halt but changes y to x'_1 and adds by to the right of x'_1. The head moves to the left to reach the beginning of X.

(ii) The tape expression obtained at the end of (i) is

$$1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^{b_1}x'_1 by$$

The construction given in (i) is repeated, i.e. M simulates $M_2, \ldots, M_k$, changes y to x'_i, and adds by to the right of x'_i. After simulating M_k, the tape expression is

$$X' = 1^{a_1}x_1 \ldots 1^{a_m}x_m 1^{b_1}x'_1 \ldots 1^{b_{k-1}}x_{k-1} 1^{b_k}x'_k by$$

Then the head moves to the left until it is positioned at the cell having 1 just to the right of x_m.

(iii) M simulates M_{k+1}. M_{k+1} with initial tape expression X' halts with the tape expression $1^{b_1}x'_1 \ldots 1^{b_k}x'_m 1^c y$. As a result, the corresponding tape expression for M is obtained as

$$1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^{b_1}x'_1 \ldots 1^{b_k}x'_k 1^c y$$

(iv) The required value is obtained to the left of y, but $1^{b_1}x'_1 \ldots 1^{b_k}x'_k$ also appears to the left of c. M erases all these symbols and moves $1^c y$ just to the right of x_m. The head moves to the cell having x_m and M halts. The final tape expression is $1^{a_1}x_1 1^{a_2}x_2 \ldots 1^{a_m}x_m 1^c y$.

11.4.8 Construction of the Turing Machine That Can Perform Recursion

Let $g(x_1, \ldots, x_m)$, $h(y_1, y_2, \ldots, y_{m+2})$ be Turing-computable. Let $f(x_1, \ldots, x_{m+1})$ be defined by recursion as follows:

$$f(x_1, \ldots, x_m, 0) = g(x_1 \ldots x_m)$$

$$f(x_1, \ldots, x_m, y + 1) = h(x_1, \ldots, x_m, y, f(x_1, \ldots, x_m, y))$$

For the Turing machine M, computing $f(a_1, \ldots, a_m, c)$, (say k), X is taken as

$$1^{a_1}x_1 \ldots 1^{a_m}x_m 1^c x_{m+1} by$$

As the construction is similar to the construction for computing composition, we outline below the steps of the construction.

Step 1 Let M simulate the Turing machine M' which computes $g(a_1, \ldots, a_m)$. The computed value, namely $g(a_1, \ldots, a_m)$, is placed to the left of y. If $c = 0$, then the computed value $g(a_1, \ldots, a_m)$ is $f(a_1, \ldots, a_m, 0)$. The head is placed to the right of x_m and M halts.

Step 2 If c is not equal to zero, 1^c to the left of x_{m+1} is replaced by b^c. The marker y is changed to x_{m+2} and by is added to the right of x_{m+2}. The head moves to the left of 1^{a_1}.

Step 3 h is computable. M is allowed to compute h for the arguments $a_1, \ldots, a_m, 0, g(a_1, \ldots, a_m)$ which appear to the left of $x_1, \ldots, x_m, x_{m+1}, x_{m+2}$,

respectively. The computed value is $f(a_1, \ldots, a_m, 1)$. And $f(a_1, \ldots, a_m, 2)$ $\ldots f(a_1, \ldots, a_m, c)$ are computed successively by replacing the rightmost b and computing h for the respective arguments.

The computation stops with a terminal ID, namely

$$b1^{a_1}x_1 1^{a_2} \ldots q_f 1^c x_{n+1} 1^k y, \quad k = f(a_1, \ldots, a_m, c)$$

11.4.9 Construction of the Turing Machine That Can Perform Minimization

When $f(x_1, \ldots, x_m)$ is defined from $g(x_1, \ldots, x_m, y)$ by minimization, $f(x_1, \ldots, x_m)$ is the least of all k's such that $g(x_1, \ldots, x_m, k) = 0$. So the problem reduces to computing $g(a_1, \ldots, a_m, k)$ for given arguments $a_1, \ldots, a_m$ and for values of k starting from 0. $f(a_1, \ldots, a_m)$ is the first k for which $g(a_1, \ldots, a_m, k) = 0$. Hence as soon as the computed value of $g(a_1, \ldots, a_m, y)$ is zero, the required Turing machine M has to halt. Of course, when no such y exists, M never halts, and $f(a_1, \ldots, a_m)$ is not defined.

Thus the construction of M is in such a way that it simulates the TM that computes $g(a_1, \ldots, a_m, k)$ for successive values of k. Once the computed value $g(a_1, \ldots, a_m, k) = 0$ for the first time, M erases by and changes x_{m+1} to y. The head moves to the left of x_m and M halts.

As partial recursive functions are obtained from the initial functions by a finite number of applications of composition, recursion and minimization (Definition 11.11) by the various constructions we have made in this section, the partial recursive functions become Turing-computable.

Using Godel numbering which converts operations of Turing machines into numeric quantities, it can be proved that Turing-computable functions are partial recursive. (For proof, refer Mendelson (1964).)

11.5 SUPPLEMENTARY EXAMPLES

EXAMPLE 11.13

Show that the function $f(x_1, x_2, \ldots, x_n) = 4$ is primitive recursive.

Solution

$$\begin{aligned} 4 &= S^4(0) \\ &= S^4(Z(x_1)) \\ &= S^4(Z(U_1^n(x_1, x_2, \ldots, x_n))) \end{aligned}$$

i.e.

$$f(x_1, x_2, \ldots, x_n) = S^4(Z(U_1^n(x_1, x_2, \ldots, x_n))).$$

As f is the composition of initial functions, f is primitive recursive.

EXAMPLE 11.14

If $f(x_1, x_2)$ is primitive recursive, show that $g(x_1, x_2, x_3, x_4) = f(x_1, x_4)$ is primitive recursive.

Solution

$$\begin{aligned} & g(x_1, x_2, x_3, x_4) \\ &= f(x_1, x_4) \\ &= f(U_1^4(x_1, x_2, x_3, x_4), U_4^4(x_1, x_2, x_3, x_4)) \end{aligned}$$

U_1^4 and U_4^4 are initial functions and hence primitive recursive. f is primitive recursive. As the function g is obtained by applying composition to primitive recursive functions, g is primitive recursive (by the Note appearing at the end of Example 11.5).

EXAMPLE 11.15

If $f(x, y)$ is primitive recursive, show that $g(x, y) = f(4, y)$ is primitive recursive.

Solution

Let $h(x, y) = 4$. h is primitive recursive by Example 11.13.

$$\begin{aligned} & g(x, y) \\ &= f(4, y) \\ &= f(h(x, y), U_2^2(x, y)) \end{aligned}$$

As f and g are primitive recursive and U_2^2 is an initial function, g is primitive recursive.

EXAMPLE 11.16

Show that $f(x, y) = x^2y^4 + 7xy^3 + 4y^5$ is primitive recursive.

Solution

As $f_1(x, y) = x + y$ is primitive recursive (Example 9.5), it is enough to prove that each summand of $f(x, y)$ is primitive recursive.

But,

$$x^2y^4 = U_1^2(x, y) * U_1^2(x, y) * U_2^2(x, y) * U_2^2(x, y) * U_2^2(x, y) * U_2^2(x, y)$$

As multiplication is primitive recursive, $g(x, y) = x^2y^4$ is primitive recursive.

As $h(x, y) = xy^3$ is primitive recursive, $7xy^3 = xy^3 + \cdots + xy^3$ is primitive recursive. Similarly, $4y^5$ is primitive recursive.

SELF-TEST

Choose the correct answer to Questions 1–10.

1. $S(Z(6))$ is equal to
(a) $U_1^3(1, 2, 3)$
(b) $U_2^3(1, 2, 3)$
(c) $U_3^3(1, 2, 3)$
(d) none of these.

2. Cons $a(y)$ is equal to
(a) $\wedge$
(b) ya
(c) ay
(d) a

3. $\min(x, y)$ is equal to
(a) $x \dot{-} (x \dot{-} y)$
(b) $y \dot{-} (y \dot{-} x)$
(c) $x - y$
(d) $y - x$

4. $A(1, 2)$ is equal to
(a) 3
(b) 4
(c) 5
(d) 6

5. $f(x) = x/3$ over N is
(a) total
(b) partial
(c) not partial
(d) total but not partial.

6. $\psi_{\{4\}}(3)$ is equal to
(a) 0
(b) 3
(c) 4
(d) none of these.

7. $\text{sgn}(x)$ takes the value 1 if
(a) $x < 0$
(b) $x \leq 0$
(c) $x > 0$
(d) $x \geq 0$

8. $\psi_A + \psi_B = \psi_{A \cup B}$ if
(a) $A \cup B = A$
(b) $A \cup B = B$
(c) $A \cap B = A$
(d) $A \cap B = \emptyset$

9. $U_2^4(S(4), S(5), S(6), Z(7))$ is
(a) 6
(b) 5
(c) 4
(d) 0

10. If $g(x, y) = \min(x, y)$ and $h(x, y) = |x - y|$, then:
(a) Both functions are regular functions.
(b) The first function is regular and the second is not regular.
(c) Neither of the functions is regular.
(d) The second function is not regular.

State whether the Statements 11–15 are true or false.

11. $f(x, y) = x + y$ is primitive recursive.

12. $3 \div 4 = 0$.

13. The transpose function is not primitive recursive.

14. The Ackermann's function is recursive but not primitive recursive.

15. $A(2, 2) = 7$.

EXERCISES

11.1 Test which of the following functions are total. If a function is not total, specify the arguments for which the function is defined.
(a) $f(x) = x/3$ over N
(b) $f(x) = 1/(x - 1)$ over N
(c) $f(x) = x^2 - 4$ over N
(d) $f(x) = x + 1$ over N
(e) $f(x) = x^2$ over N

11.2 Show that the following functions are primitive recursive:

(a) $\chi_{\{0\}}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$

(b) $f(x) = x^2$

(c) $f(x, y) =$ maximum of x and y

(d) $f(x) = \begin{cases} x/2 & \text{when } x \text{ is even} \\ (x - 1)/2 & \text{when } x \text{ is odd} \end{cases}$

(e) The sign function defined by
$\text{sgn}(0) = 0$, $\text{sgn}(x) = 1$ if $x > 0$.

(f) $L(x, y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x \leq y \end{cases}$

(g) $E(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$

11.3 Compute $A(3, 2)$, $A(2, 3)$, $A(3, 3)$.

11.4 Show that the following functions are primitive recursive:

(a) $q(x, y)$ = the quotient obtained when x is divided by y

(b) $r(x, y)$ = the remainder obtained when x is divided by y

(c) $f(x) = \begin{cases} 2x & \text{if } x \text{ is a perfect square} \\ 2x+1 & \text{otherwise} \end{cases}$

11.5 Show that $f(x)$ = integral part of $\sqrt{x}$ is partial recursive.

11.6 Show that the Fibonacci numbers are generated by a primitive recursive function.

11.7 Let $f(0) = 1$, $f(1) = 2$, $f(2) = 3$ and $f(x + 3) = f(x) + f(x + 1)^2 + f(x + 2)^3$. Show that $f(x)$ is primitive recursive.

11.8 The characteristic function χ_A of a given set A is defined as

$$\chi_A(a) = \begin{cases} 0 & \text{if } a \notin A \\ 1 & \text{if } a \in A \end{cases}$$

If A, B are subsets of N and χ_A, χ_B are recursive, show that χ_{A^C}, $\chi_{A \cup B}$, $\chi_{A \cap B}$ are also recursive.

11.9 Show that the characteristic function of the set of all even numbers is recursive. Prove that the characteristic function of the set of all odd integers is recursive.

11.10 Show that the function $f(x, y) = x - y$ is partial recursive.

11.11 Show that a constant function over N, i.e. $f(n) = k$ for all n in N where k is a fixed number, is primitive recursive.

11.12 Show that the characteristic function of a finite subset of N is primitive recursive.

11.13 Show that the addition function $f_1(x, y)$ is Turing-computable. (Represent x and y in tally notation and use concatenation.)

11.14 Show that the Turing machine M in the Post notation (i.e. the transition function specified by quadruples) can be simulated by a Turing machine M (as defined in Chapter 9).

[*Hint:* The transition given by a quadruple can be simulated by two quintuples of M' by adding new states to M'.]

11.15 Compute $Z(4)$ using the Turing machine constructed for computing the zero function.

11.16 Compute $S(3)$ using the Turing machine which computes S.

11.17 Compute $U_1^3(2, 1, 1)$, $U_2^3(1, 2, 1)$, $U_3^3(1, 2, 1)$ using the Turing machines which can compute the projection functions.

11.18 Construct a Turing machine which can compute $f(x) = x + 2$.

11.19 Construct a Turing machine which can compute $f(x_1, x_2) = x_1 + 2$ for the arguments 1, 2 (i.e. $x_1 = 1$, $x_2 = 2$).

11.20 Construct a Turing machine which can compute $f(x_1, x_2) = x_1 + x_2$ for the arguments 2, 3 (i.e. $x_1 = 2$, $x_2 = 3$).

12 Complexity

When a problem/language is decidable, it simply means that the problem is computationally solvable in principle. It may not be solvable in practice in the sense that it may require enormous amount of computation time and memory. In this chapter we discuss the computational complexity of a problem. The proofs of decidability/undecidability are quite rigorous, since they depend solely on the definition of a Turing machine and rigorous mathematical techniques. But the proof and the discussion in complexity theory rests on the assumption that $\mathbf{P} \neq \mathbf{NP}$. The computer scientists and mathematicians strongly believe that $\mathbf{P} \neq \mathbf{NP}$, but this is still open.

This problem is one of the challenging problems of the 21st century. This problem carries a prize money of \$1M. **P** stands for the class of problems that can be solved by a deterministic algorithm (i.e. by a Turing machine that halts) in polynomial time; **NP** stands for the class of problems that can be solved by a nondeterministic algorithm (that is, by a nondeterministic TM) in polynomial time; **P** stands for polynomial and **NP** for nondeterminisitc polynomial. Another important class is the class of *NP*-complete problems which is a subclass of **NP**.

In this chapter these concepts are formalized and Cook's theorem on the *NP*-completeness of SAT problem is proved.

12.1 GROWTH RATE OF FUNCTIONS

When we have two algorithms for the same problem, we may require a comparison between the running time of these two algorithms. With this in mind, we study the growth rate of functions defined on the set of natural numbers.

In this section, N denotes the set of natural numbers.

Definition 12.1 Let $f, g : N \to R^+$ (R^+ being the set of all positive real numbers). We say that $f(n) = O(g(n))$ if there exist positive integers C and N_0 such that

$$f(n) \leq Cg(n) \qquad \text{for all } n \geq N_0.$$

In this case we say f is of the order of g (or f is 'big oh' of g)

Note: $f(n) = O(g(n))$ is not an equation. It expresses a relation between two functions f and g.

EXAMPLE 12.1

Let $f(n) = 4n^3 + 5n^2 + 7n + 3$. Prove that $f(n) = O(n^3)$.

Solution

In order to prove that $f(n) = O(n^3)$, take $C = 5$ and $N_0 = 10$. Then

$$f(n) = 4n^3 + 5n^2 + 7n + 3 \leq 5n^3 \qquad \text{for } n \geq 10$$

When $n = 10$, $5n^2 + 7n + 3 = 573 < 10^3$. For $n > 10$, $5n^2 + 7n + 3 < n^3$. Then, $f(n) = O(n^3)$.

Theorem 12.1 If $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree k over Z and $a_k > 0$, then $p(n) = O(n^k)$.

Proof $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$. As a_k is an integer and positive, $a_k \geq 1$.

As $a_{k-1}, a_{k-2}, \ldots, a_1, a_0$ and k are fixed integers, choose N_0 such that for all $n \geq N_0$ each of the numbers

$$\frac{|a_{k-1}|}{n}, \frac{|a_{k-2}|}{n^2}, \ldots, \frac{|a_1|}{n^{k-1}}, \frac{|a_0|}{n^k} \text{ is less than } \frac{1}{k} \qquad (*)$$

Hence,

$$\left|\frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k}\right| < 1$$

As $a_k \geq 1$, $\dfrac{p(n)}{n^k} = a_k + \dfrac{a_{k-1}}{n} + \cdots + \dfrac{a_1}{n^{k-1}} + \dfrac{a_0}{n^k} > 0 \qquad$ for all $n \geq N_0$

Also,

$$\frac{p(n)}{n^k} = a_k + \left(\frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}\right)$$

$$\leq a_k + 1 \qquad \text{by } (*)$$

So,

$$p(n) \leq Cn^k, \qquad \text{where} \qquad C = a_k + 1$$

Hence,

$$p(n) = O(n^k). \qquad \blacksquare$$

Corollary The order of a polynomial is determined by its degree.

Definition 12.2 An exponential function is a function $q : N \to N$ defined by

$$q(n) = a^n \quad \text{for some fixed } a > 1.$$

When n increases, each of n, n^2, 2^n increases. But a comparison of these functions for specific values of n will indicate the vast difference between the growth rate of these functions.

TABLE 12.1 Growth Rate of Polynomial and Exponential Functions

n	$f(n) = n^2$	$g(n) = n^2 + 3n + 9$	$q(n) = 2^n$
1	1	13	2
5	25	49	32
10	100	139	1024
50	2500	2659	$(1.13)10^{15}$
100	10000	10309	$(1.27)10^{30}$
1000	1000000	1003009	$(1.07)10^{301}$

From Table 12.1, it is easy to see that the function $q(n)$ grows at a very fast rate when compared to $f(n)$ or $g(n)$. In particular the exponential function grows at a very fast rate when compared to any polynomial of large degree. We prove a precise statement comparing the growth rate of polynomials and exponential function.

Definition 12.3 We say $g \neq O(f)$, if for any constant C and N_0, there exists $n \geq N_0$ such that $g(n) > Cf(n)$.

Definition 12.4 If f and g are two functions and $f = O(g)$, but $g \neq O(f)$, we say that the growth rate of g is greater than that of f. (In this case $g(n)/f(n)$ becomes unbounded as n increases to ∞.)

Theorem 12.2 The growth rate of any exponential function is greater than that of any polynomial.

Proof Let $p(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0$ and $q(n) = a^n$ for some $a > 1$.

As the growth rate of any polynomial is determined by its term with the highest power, it is enough to prove that $n^k = O(a^n)$ and $a^n \neq O(n^k)$. By L'Hospital's rule, $\dfrac{\log n}{n}$ tends to 0 as $n \to \infty$. (Here $\log n = \log_e n$.) If

$$z(n) = \left[e^{k\left(\frac{\log n}{n}\right)} \right]$$

then,

$$(z(n))^n = \left[e^{k\left(\frac{\log n}{n}\right)} \right]^n = e^{k \log n} = e^{\log n^k} = n^k$$

As n gets large, $k\left(\dfrac{\log n}{n}\right)$ tends to 0 and hence $z(n)$ tends to 0.

So we can choose N_0 such that $z(n) \leq a$ for all $n \geq N_0$. Hence $n^k = z(n)^n \leq a^n$, proving $n^k = O(a^n)$.

To prove $a^n \neq O(n^k)$, it is enough to show that a^n/n^k is unbounded for large n. But we have proved that $n^k \leq a^n$ for large n and any positive integer k and hence for $k + 1$. So $n^{k+1} \leq a^n$ or $\dfrac{a^n}{n^{k+1}} \geq 1$.

Multiplying by n, $n\left(\dfrac{a^n}{n^{k+1}}\right) \geq n$, which means $\dfrac{a^n}{n^k}$ is unbounded for large values of n. ∎

Note: The function $n^{\log n}$ lies between any polynomial function and a^n for any constant a. As $\log n \geq k$ for a given constant k and large values of n, $n^{\log n} \geq n^k$ for large values of n. Hence $n^{\log n}$ dominates any polynomial. But $n^{\log n} = (e^{\log n})^{\log n} = e^{(\log n)^2}$. Let us calculate $\lim\limits_{x\to\infty} \dfrac{(\log x)^2}{cx}$. By L'Hospital's rule, $\lim\limits_{x\to\infty} \dfrac{(\log x)^2}{cx} = \lim\limits_{x\to\infty} (2\log x)\dfrac{1/x}{c} = \lim\limits_{x\to\infty} \dfrac{2\log x}{cx} = \lim\limits_{x\to\infty} \dfrac{2}{cx} = 0$.

So $(\log n)^2$ grows more slowly than cn. Hence $n^{\log n} = e^{(\log n)^2}$ grows more slowly than 2^{cn}. The same holds good when logarithm is taken over base 2 since $\log_e n$ and $\log_2 n$ differ by a constant factor.

Hence there exist functions lying between polynomials and exponential functions.

12.2 THE CLASSES *P* AND *NP*

In this section we introduce the classes **P** and **NP** of languages.

Definition 12.5 A Turing machine M is said to be of time complexity $T(n)$ if the following holds: Given an input w of length n, M halts after making at most $T(n)$ moves.

Note: In this case, M eventually halts. Recall that the standard TM is called a deterministic TM.

Definition 12.6 A language L is in class **P** if there exists some polynomial $T(n)$ such that $L = T(M)$ for some deterministic TM M of time complexity $T(n)$.

EXAMPLE 12.2

Construct the time complexity $T(n)$ for the Turing machine M given in Example 9.7.

Solution

In Example 9.7, the step (i) consists of going through the input string (0^n1^n) forward and backward and replacing the leftmost 0 by x and the leftmost 1 by y. So we require at most $2n$ moves to match a 0 with a 1. Step (ii) is repetition of step (i) n times. Hence the number of moves for accepting a^nb^n is at most $(2n)(n)$. For strings not of the form a^nb^n, TM halts with less than $2n^2$ steps. Hence $T(M) = O(n^2)$.

We can also define the complexity of algorithms. In the case of algorithms. $T(n)$ denotes the running time for solving a problem with an input of size n, using this algorithm.

In Example 12.2, we use the notation $\leftarrow$ which is used in expressing algorithm. For example, $a \leftarrow b$ means replacing a by b.

$\lceil a \rceil$ denotes the smallest integer greater than or equal to a. This is called the *ceiling function*.

EXAMPLE 12.3

Find the running time for the Euclidean algorithm for evaluating gcd(a, b) where a and b are positive integers expressed in binary representation.

Solution

The Euclidean algorithm has the following steps:

1. The input is (a, b)
2. Repeat until $b = 0$
3. Assign $a \leftarrow a \bmod b$
4. Exchange a and b
5. Output a.

Step 3 replaces a by $a \bmod b$. If $a/2 \geq b$, then $a \bmod b < b \leq a/2$. If $a/2 < b$, then $a < 2b$. Write $a = b + r$ for some $r < b$. Then $a \bmod b = r < b < a/2$. Hence $a \bmod b \leq a/2$. So a is reduced by at least half in size on the application of step 3. Hence one iteration of step 3 and step 4 reduces a and b by at least half in size. So the maximum number of times the steps 3 and 4 are executed is $\min\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$. If n denotes the maximum of the number of digits of a and b, that is $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$ then the number of iterations of steps 3 and 4 is $O(n)$. We have to perform step 2 at most $\min\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\}$ times or n times. Hence $T(n) = nO(n) = O(n^2)$.

Note: The Euclidean algorithm is a polynomial algorithm.

Definition 12.7 A language L is in class **NP** if there is a nondeterministic TM M and a polynomial time complexity $T(n)$ such that $L = T(M)$ and M executes at most $T(n)$ moves for every input w of length n.

We have seen that a deterministic TM M_1 simulating a nondeterministic TM M exists (refer to Theorem 9.3). If $T(n)$ is the complexity of M, then the complexity of the equivalent deterministic TM M_1 is $2^{O(T(n))}$. This can be justified as follows. The processing of an input string w of length n by M is equivalent to a 'tree' of computations by M_1. Let k be the maximum of the number of choices forced by the nondeterministic transition function. (It is $\max|\delta(q, x))|$, the maximum taken over all states q and all tape symbol X.) Every branch of the computation tree has a length $T(n)$ or less. Hence the total number of leaves is atmost $kT(n)$. Hence the complexity of M_1 is at most $2^{O(T(n))}$.

It is not known whether the complexity of M_1 is less than $2^{O(T(n))}$. Once again an answer to this question will prove or disprove **P** ≠ **NP**. But there do exist algorithms where $T(n)$ lies between a polynomial and an exponential function (refer to Section 12.1).

12.3 POLYNOMIAL TIME REDUCTION AND *NP*-COMPLETENESS

If P_1 and P_2 are two problems and $P_2 \in \mathbf{P}$, then we can decide whether $P_1 \in \mathbf{P}$ by relating the two problems P_1 and P_2. If there is an algorithm for obtaining an instance of P_2 given any instance of P_1, then we can decide about the problem P_1. Intuitively if this algorithm is a polynomial one, then the problem P_1 can be decided in polynomial time.

Definition 12.8 Let P_1 and P_2 be two problems. A reduction from P_1 to P_2 is an algorithm which converts an instance of P_1 to an instance of P_2. If the time taken by the algorithm is a polynomial $p(n)$, n being the length of the input of P_1, then the reduction is called a polynomial reduction P_1 to P_2.

Theorem 12.3 If there is a polynomial time reduction from P_1 to P_2 and if P_2 is in **P** then P_1 is in **P**.

Proof Let m denote the size of the input of P_1. As there is a polynomial-time reduction of P_1 to P_2, the corresponding instance of P_2 can be got in polynomial-time. Let it be $O(m^j)$, So the size of the resulting input of P_2 is atmost cm^j for some constant c. As P_2 is in **P**, the time taken for deciding the membership in P_2 is $O(n^k)$, n being the size of the input of P_2. So the total time taken for deciding the membership of m-size input of P_1 is the sum of the time taken for conversion into an instance of P_2 and the time for decision of the corresponding input in P_2. This is $O[m^j + (cm^j)^k]$, which is the same as $O(m^{jk})$. So P_1 is in **P**. ∎

Definition 12.9 Let L be a language or problem in **NP**. Then L is *NP*-complete if

1. L is in **NP**

2. For every language L' in **NP** there exists a polynomial-time reduction of L' to L.

Note: The class of *NP*-complete languages is a subclass of **NP**.

The next theorem can be used to enlarge the class of *NP*-complete problems provided we have some known *NP*-complete problems.

Theorem 12.4 If P_1 is *NP*-complete, and there is a polynomial-time reduction of P_1 to P_2, then P_2 is *NP*-complete.

Proof If L is any language in **NP**, we show that there is a polynomial-time reduction of L to P_2. As P_1 is *NP*-complete, there is a polynomial-time reduction of L to P_1. So the time taken for converting an n-size input string w in L to a string x in P_1 is at most $p_1(n)$ for some polynomial p_1. As there is a polynomial-time reduction of P_1 to P_2, there exists a polynomial p_2 such that the input x to P_1 is transferred into input y to P_2 in at most $p_2(n)$ time. So the time taken for transforming w to y is at most $p_1(n) + p_2(p_1(n))$. As $p_1(n) + p_2(p_1(n))$ is a polynomial, we get a polynomial-time reduction of L to P_2. Hence P_2 is *NP*-complete. ∎

Theorem 12.5 If some *NP*-complete problem is in **P**, then **P** = **NP**.

Proof Let P be an *NP*-complete problem and $P \in \mathbf{P}$. Let L be any *NP*-complete problem. By definition, there is a polynomial-time reduction of L to P. As P is in **P**, L is also in **P** by Theorem 12.3. Hence **NP** = **P**.

12.4 IMPORTANCE OF *NP*-COMPLETE PROBLEMS

In Section 12.3, we proved theorems regarding the properties of *NP*-complete problems. At the beginning of this chapter we noted that the computer scientists and mathematicians strongly believe that $\mathbf{P} \neq \mathbf{NP}$. At the same time, no problem in **NP** is proved to be in **P**. The entire complexity theory rests on the strong belief that $\mathbf{P} \neq \mathbf{NP}$.

Theorem 12.4 enables us to extend the class of *NP*-complete problems, while Theorem 12.5 asserts that the existence of one *NP*-complete problem admitting a polynomial-time algorithm will prove **P** = **NP**. More than 2500 *NP*-complete problems in various fields have been found so far.

We will prove the existence of an *NP*-complete problem in Section 12.5. We will give a list of *NP*-complete problems in Section 12.6. Thousands of *NP*-complete problems in various branches such as Operations Research, Logic, Graph Theory, Combinatorics, etc. have been constructed so far. A polynomial-time algorithm for any one of there problems will yield a proof of **P** = **NP**. But such multitude of *NP*-complete problems only strengthens the belief of the computer scientists that $\mathbf{P} \neq \mathbf{NP}$. We will discuss more about this in Section 12.7.

12.5 SAT IS *NP*-COMPLETE

In this section, we prove that the satisfiability problem for boolean expressions (whether a boolean expression is satisfiable) is *NP*-complete. This is the first problem to be proved *NP*-complete. Cook proved this theorem in 1971.

12.5.1 BOOLEAN EXPRESSIONS

In Section 1.1.2, we defined a well-formed formula involving propositional variables. A boolean expression is a well-formed formula involving boolean variables x, y, z replacing propositions P, Q, R and connectives $\vee$, $\wedge$ and $\neg$. The truth value of a boolean expression in x, y, z is determined from the truth values of x, y, z and the truth tables for $\vee$, $\wedge$ and $\neg$. For example, $\neg x \wedge \neg (y \vee z)$ is a boolean expression. The expression $\neg x \wedge \neg (y \vee z)$ is true when x is false, y is false and z is false.

Definition 12.10 (a) A truth assignment t for a boolean expression E is the assignment of truth values T or F to each of the variables in E. For example, $t = (F, F, F)$ is a truth assignment for (x, y, z) where x, y, z are the variables in a boolean expression $E(x, y, z) = \neg x \wedge \neg (y \vee z)$.

The value $E(t)$ of the boolean expression E given a truth assignment t is the truth value of the expression of E, if the truth values give by t are assigned to the respective variables.

If $t = (F, F, F)$ then the truth values of $\neg x$ and $\neg (y \vee z)$ are T and T. Hence the value of $E = \neg x \wedge \neg (y \vee z)$ is T. So $E(t) = T$.

Definition 12.11 A truth assignment t satisfies a boolean expression E if the truth value of $E(t)$ is T. In other words, the truth assignment t makes the expression E true.

Definition 12.12 A boolean expression E is satisfiable if there exists at least one truth assignment t that satisfies E (that is $E(t) = T$). For example, $E = \neg x \wedge \neg (y \vee z)$ is satisfiable since $E(t) = T$ when $t = (F, F, F)$.

12.5.2 CODING A BOOLEAN EXPRESSION

The symbols in a boolean expression are the variables x, y, z, etc. the connectives $\vee$, $\wedge$, $\neg$, and parantheses (and). Thus a boolean expression in three variables will have eight distinct symbols. The variables are written as x_1, x_2, x_3, etc. Also we use x_n only after using x_1, x_2, ..., x_{n-1} for variables.

We encode a boolean expression as follows:

1. The variables x_1, x_2, x_3, ... are written as $x1$, $x10$, $x11$, ... etc. (The binary representation of the subscript is written after x.)
2. The connectives $\vee$, $\wedge$, $\neg$, (, and) are retained in the encoded expression.

For example, $\neg x \wedge \neg (y \vee z)$ is encoded as $\neg x1 \wedge \neg (x10 \vee x11)$, (where x, y, z are represented by x_1, x_2, x_3).

Note: Any boolean expression is encoded as a string over $\Sigma = \{x, 0, 1, \vee, \wedge, \neg, c, (,)\}$

Consider a boolean expression having m occurrences of variables, connectives and parantheses. The variable x_m can be represented using $1 + \log_2 m$ symbols (x together with the digits in the binary representation of m). The other occurrences require less symbols. So any occurrence of a variable, connective or a parenthesis requires at most $1 + \log_2 m$ symbols over Σ. So the length of the encoded expression is at most $O(m \log m)$.

As our interest is only in deciding whether a problem can be solved in polynomial-time, we need not distinguish between the length of the coded expression and the number of occurrences of variables etc. in a boolean expression.

12.5.3 Cook's Theorem

In this section we define the SAT problem and prove the Cook's theorem that SAT is *NP*-complete.

Definition 12.13 The satisfiability problem (SAT) is the problem:

Given a boolean expression, is it satisfiable?

Note: The SAT problem can also be formulated as a language. We can define SAT as the set of all coded boolean expressions that are satisfiable. So the problem is to decide whether a given coded boolean expression is in SAT.

Theorem 12.6 (Cook's theorem) SAT is *NP*-complete.

Proof PART I: SAT $\in$ **NP**.

If the encoded expression E is of length n, then the number of variables is $\lceil n/2 \rceil$. Hence, for guessing a truth assignment t we can use multitape TM for E. The time taken by a multitape NTM M is $O(n)$. Then M evaluates the value of E for a truth assignment t. This is done in $O(n^2)$ time. An equivalent single-tape TM takes $O(n^4)$ time. Once an accepting truth assignment is found, M accepts E and M and halts. Thus we have found a polynomial time NTM for SAT. Hence SAT $\in$ **NP**.

PART II: POLYNOMIAL-TIME REDUCTION OF ANY L IN **NP** TO SAT.

1. Construction of NTM for L

Let L be any language in **NP**. Then there exists a single-tape NTM M and a polynomial $p(n)$ such that the time taken by M for an input of length n is at most $p(n)$ along any branch. We can further assume that this M never writes a blank on any move and never moves its head to the left of its initial tape position (refer to Example 12.6).

If M accepts an input w and $|w| = n$, then there exists a sequence of moves of M such that

1. α_0 is the initial ID of M with input w.
2. $\alpha_0 \vdash \alpha_1 \vdash \cdots \vdash \alpha_k$, $k \leq p(n)$.
3. α_k is an ID with an accepting state.
4. Each α_i is a string of nonblanks, its leftmost symbol being the leftmost symbol of w (the only exception occurs when the processing of w is complete, in which case the ID is qb).

2. Representation of Sequence of Moves of M

As the maximum number of steps on w is $p(n)$ we need not bother about the contents beyond $p(n)$ cells. We can write α_i as a sequence of $p(n) + 1$ symbols (one symbol for the state and the remaining symbols for the tape symbols). So $\alpha_i = X_{i0}X_{i1} \ldots X_{i,\,p(n)}$.

By assuming $Q \cap \Gamma = \emptyset$, we can locate the state in α_i and hence the position of the tape head. The length of some ID may be less than $p(n)$. In this case we pad the ID on the right with blank symbols, so that all IDs are of the same length $p(n) + 1$. Also the acceptance may happen earlier. If α_m is an accepting ID in the course of processing w, then we write $\alpha_0 \vdash \ldots \vdash \alpha_m \vdash \alpha_m \ldots \vdash \alpha_m = \alpha_{i,p(n)}$.

Thus all IDs have $p(n) + 1$ symbols and any computation has $p(n)$ moves.

TABLE 12.2 Array of IDs

ID	0	1	...	$j-1$	j	$j+1$	...	$p(n)$
α_0	X_{00}	X_{01}		...				$X_{0,p(n)}$
α_1	X_{10}	X_{11}						$X_{1,p(n)}$
α_i	X_{i0}	X_{i1}	...	$X_{i,j-1}$	X_{ij}	$X_{i,j+1}$	...	$X_{i,p(n)}$
α_{i+1}	$X_{i+1,0}$	$X_{i+1,1}$		$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$	...	$X_{i+1,p(n)}$
$\alpha_{p(n)}$				...				$X_{p(n),p(n)}$

So we can represent any computation as an $(p(n) + 1) \times (p(n) + 1)$ array as in Table 12.2.

3. Representation of IDs in Terms of Boolean Variables

We define a boolean variable y_{ijA} corresponding to (i, j)th entry in the ith ID. The variable y_{ijA} represents the proposition that $x_{ij} = A$, where A is a state or tape symbol and $0 \leq i, j \leq p(n)$.

We simulate the sequence of IDs leading to the acceptance of an input string w by a boolean expression. This is done in such a way that M accepts w if an only if the simulated boolean expression $E_{M,w}$ is satisfiable.

4. Polynomial Reduction of M to SAT

In order to check that the reduction of M to SAT is correct, we have to ensure the correctness of

(a) the initial ID,
(b) the accepting ID, and
(c) the intermediate moves between successive IDs.

(a) Simulation of initial ID

X_{00} must start with the initial state q_0 of M followed by the symbols of $w = a_1 a_2 \ldots a_n$ of length n and ending with b's (blank symbol). The corresponding boolean expression S is defined as

$$S = y_{00q_0} \wedge y_{01a_1} \wedge y_{01a_2} \wedge \ldots \wedge y_{0na_n} \wedge y_{0,n+1,b} \wedge \ldots \wedge y_{0,p(n),b}$$

Thus given an encoding of M and w, we can write S in a tape of a multiple TM M_1. This takes $O(p(n))$ time.

(b) Simulation of accepting ID

$\alpha_{p(n)}$ is the accepting ID. If $p_1, p_2, \ldots, p_k$ are the accepting states of M, then $\alpha_{p(n)}$ contains one of p_i's, $1 \le i \le k$ in any place j. If $\alpha_{p(n)}$ contains an accepting state p_i in jth position, then $x_{p(n),j}$ is the accepting state p_i. The corresponding boolean expression covering all the cases ($0 \le j \le p(n)$, $1 \le i \le k$) is given by

$$F = F_0 \vee F_1 \vee \ldots \vee F_{p(n)}$$

where

$$F_j = y_{p(n),j,p_1} \vee y_{p(n),j,p_2} \vee \ldots \vee y_{p(n),j,p_k}$$

Each F_j has k variables and hence has constant number of symbols depending on M but not on n. The number of F_j's in F is $p(n)$. Thus given an encoding of M and w, F can be written in $O(p(n))$ time on the multiple TM M_1.

(c) Simulation of intermediate moves

We have to simulate valid moves $\alpha_i \vdash \alpha_{i+1}$, $i = 0, 1, 2, \ldots p(n)$. Corresponding to each move, we have to define a boolean variable N_i. Hence the entire sequence of IDs leading to acceptance of w is

$$N = N_0 \wedge N_1 \wedge \ldots \wedge N_{p(n)-1}$$

First of all note that the symbol $X_{i+1,j}$ can be determined from $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$ by the move (if there is one changing α_i to a different α_{i+1}). For every position (i, j), we have two cases:

Case 1 The state of α_i is at position j.
Case 2 The state of α_i is not in any of the $(j-1)$th, jth and $(j+1)$th positions.

Case 1 is taken care of by a variable A_{ij} and Case 2 by a variable B_{ij}.

The variable N_i will be designed in such a way that it gurantees that ID α_{i+1} is one of the IDs that follows the ID α_i.

$X_{i+1,j}$ can be determined from

(i) the three symbols $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$ above it

(ii) the move chosen by the nondeterministic TM M when one of the three symbols (in (i)) is a state.

If the state of α_i is not X_{ij}, $X_{i,j-1}$ or $X_{i,j+1}$, then $X_{i+1,j} = X_{ij}$. This is taken care of by the variable B_{ij}.

If X_{ij} is the state of α_i, then $X_{i,j+1}$ is being scanned by the state X_{ij}. The move corresponding to the state-tape symbol pair $(X_{ij}, X_{i,j+1})$ will determine the sequence $X_{i+1,j-1}\, X_{i+1,j}\, X_{i+1,j+1}$. This is taken care of by the variable A_{ij}.

We write $N_i = \wedge_j (A_{ij} \vee B_{ij})$, where $\wedge$ is taken over all j's, $0 \le j \le p(n)$.

(i) Formulation of B_{ij} When the state of α_i is none of $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$, then the transition corresponding to $\alpha_i \vdash \alpha_{i+1}$ will not affect $X_{i,j+1}$. In this case $X_{i+1,j} = X_{ij}$

Denote the tape symbols by $Z_1, Z_2, \ldots, Z_r$. Then $X_{i,j-1}$, $X_{i,j}$ and $X_{i,j+1}$ are the only tape symbols. So we write B_{ij} as

$$
\begin{aligned}
B_{ij} = {} & (y_{i,j-1,Z_1} \vee y_{i,j-1,Z_2} \vee \cdots \vee y_{i,j-1,Z_r}) \wedge \\
& (y_{i,j,Z_1} \vee y_{i,j,Z_2} \cdots \vee y_{i,j,Z_r}) \wedge \\
& (y_{i,j+1,Z_1} \vee y_{i,j+1,Z_2} \cdots \vee y_{i,j+1,Z_r}) \wedge \\
& (y_{i,j,Z_1} \wedge y_{i+1,j,Z_1}) \vee (y_{i,j,Z_2} \wedge y_{i+1,j,Z_2} \vee \cdots \vee (y_{i,j,Z_r} \wedge y_{i+1,j,Z_r})
\end{aligned}
$$

This first line of B_{ij} says that $X_{i,j-1}$ is one of the tape symbols $Z_1, Z_2, \ldots, Z_r$. The second and third lines are regarding $X_{i,j}$ and $X_{i,j+1}$. The fourth line says that X_{ij} and $X_{i,j+1}$ are the same and the common value is any one of $Z_1, Z_2, \ldots, Z_r$.

Recall that the head of M never moves to the left of 0-cell and does not have to move to the right of the $p(n)$-cell. So B_{i0} will not have the first line and $B_{i,p(n)}$ will not have the third line.

(ii) Formulation of A_{ij} This step corresponds to the correctness of the 2×3 array (see Table 12.3).

TABLE 12.3 Valid Computation

$X_{i,j-1}$	X_{ij}	$X_{i,j+1}$
$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$

The expression B_{ij} takes care of the case when the state of α_i is not at the position $X_{i,j-1}$, $X_{i,j}$ or $X_{i,j+1}$. The A_{ij} corresponds to the case when the state of α_i is at the position X_{ij}. In this case we have to assign boolean variables to six positions given in Table 12.3 so that the transition corresponding to $\alpha_i \vdash \alpha_{i+1}$ is described by the variables in the box correctly.

We say that that an assignment of symbols to the six variables in the box is valid if

1. X_{ij} is a state but $X_{i,j-1}$ and $X_{i,j+1}$ are tape symbols.
2. Exactly one of $X_{i+1,j-1}$, $X_{i+1,j}$, $X_{i+1,j+1}$ is a state.
3. There is a move which explains how $(X_{i,j-1}, X_{i,j}, X_{i,j+1})$ changes to $(X_{i+1,j-1}, X_{i+1,j}, X_{i+1,j+1})$ in $\alpha_i \vdash \alpha_{i+1}$.

There are only a finite number of valid assignments and A_{ij} is obtained by applying OR (that is $\vee$) to these valid assignments. A valid assignment corresponds to one of the following four cases:

Case A $(p, C, L) \in \delta(q, A)$
Case B $(p, C, R) \in \delta(q, A)$
Case C $\alpha_i = \alpha_{i+1}$ (when α_i and α_{i+1} contain an accepting state)
Case D $j = 0$ and $j = p(n)$

Case A Let D be some tape symbol of M. Then $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ and $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = pDC$. This can be expressed by the boolean variable.

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,D} \wedge y_{i+1,j+1,C}$$

Case B As in case A, let D be any tape symbol. In this case $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ and $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = DCp$. The corresponding boolean expression is

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,D} \wedge y_{i+1,j,C} \wedge y_{i+1,j+1,p}$$

Case C In this case $X_{i,j-1}X_{ij}X_{i,j+1} = X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$.

In this case the same tape symbol say D appears in $X_{i,j-1}$ and $X_{i+1,j-1}$; some other tape symbol say D' in $X_{i,j+1}$ and $X_{i+1,j+1}$. $X_{i,j}$ and $X_{i+1,j}$ contain the same state. One typical boolean expression is

$$y_{i,j-1,Z_k} \wedge y_{i,j,q} \wedge y_{i,j+1,Z_l} \wedge y_{i+1,j-1,Z_k} \wedge y_{i+1,j,q} \wedge y_{i+1,j+1,Z_l}$$

Case D When $j = 0$, we have only $X_{i0}X_{i1}$ and $X_{i+1,0}X_{i+1,1}$. This is a special case of Case B. $j = p(n)$ corresponds to a special case of Case A.

So, A_{ij} is defined as the OR of all valid terms obtained in Case A to Case D.

(iii) Definition of N_i and N We define N_i and N by

$$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \ldots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

$$N = N_0 \wedge N_1 \wedge N_2 \wedge \ldots \wedge N_{p(n)-1}$$

(iv) Time taken for writing N The time taken to write B_{ij} is a constant depending on the number $|\Gamma|$ of tape symbols. (Actually the number of variables in B_{ij} is $5|\Gamma|$). The time taken to write A_{ij} depends only on the number of moves of M. As N_i is obtained by applying OR to $A_{ij} \wedge B_{ij}$, $0 \le i \le p(n) - 1$, $0 \le j \le p(n) - 1$, the time taken to write on N_i is $O(p(n))$. As N is obtained by applying $\wedge$ to $N_0, N_1, \ldots, N_{p(n)-1}$, the time taken to write N is $p(n)O(p(n)) = O(p^2(n))$.

5. Completion of Proof

Let $E_{M,w} = S \wedge N \wedge F$.

We have seen that the time taken to write S and F are $O(p(n))$ and the time taken for N is $O(p^2(n))$. Hence the time taken to write $E_{M,w}$ is $O(p^2(n))$.

Also M accepts w if and only if $E_{M,w}$ is satisfiable.

Hence the deterministic multitape TM M_1 can convert w to a boolean expression $E_{M,w}$ in $O(p^2(n))$ time. An equivalent single tape TM takes $O(p^4(n))$ time. This proves the Part II of the Cook's theorem, thus completing the proof of this theorem. ∎

12.6 OTHER *NP*-COMPLETE PROBLEMS

In the last section, we proved the *NP*-completeness of SAT. Actually it is difficult to prove the *NP*-completeness of any problem. But after getting one *NP*-complete problem such as SAT, we can prove the *NP*-completeness of problem P' by obtaining a polynomial reduction of SAT to P'. The polynomial reduction of SAT to P' is relatively easy. In this section we give a list of *NP*-complete problems without proving their *NP*-completeness. Many of the *NP*-complete problems are of practical interest.

1. CSAT—Given a boolean expression in CNF (conjunctive normal form—Definition 1.10), is it satisfiable?

   We can prove that CSAT is *NP*-complete by proving that CSAT is in **NP** and getting a polynomial reduction from SAT to CSAT.
2. Hamiltonian circuit problem—Does G have a Hamiltonian circuit (i.e. a circuit passing through each edge of G exactly once)?
3. Travelling salesman problem (TSP)—Given n cities, the distance between them and a number D, does there exist a tour programme for a salesman to visit all the cities exactly once so that the distance travelled is at most D?
4. Vertex cover problem—Given a graph G and a natural number k, does there exist a vertex cover for G with k vertices? (A subsets C of vertices of G is a vertex cover for G if each edge of G has an odd vertex in C.)

5. Knapsack problem—Given a set $A = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and an integer K, does there exist a subset B of A such that

$$\sum_{b_j \in B} b_j = K?$$

This list of *NP*-complete problems can be expanded by having a polynomial reduction of known *NP*-complete problems to the problems which are in **NP** and which are suspected to be *NP*-complete.

12.7 USE OF *NP*-COMPLETENESS

One practical use in discovering that problem is *NP*-complete is that it prevents us from wasting our time and energy over finding polynomial or easy algorithms for that problem.

Also we may not need the full generality of an *NP*-complete problem. Particular cases may be useful and they may admit polynomial algorithms. Also there may exist polynomial algorithms for getting an approximate optimal solution to a given *NP*-complete problem.

For example, the travelling salesman problem satisfying the triangular inequality for distances between cities (i.e. $d_{ij} \leq d_{ik} + d_{kj}$ for all i, j, k) has approximate polynomial algorithm such that the ratio of the error to the optimal values of total distance travelled is less than or equal to 1/2.

12.8 QUANTUM COMPUTATION

In the earlier sections we discussed the complexity of algorithm and the dead end was the open problem **P** = **NP**. Also the class of *NP*-complete problems provided us with a class of problems. If we get a polynomial algorithm for solving one *NP*-complete problem we can get a polynomial algorithm for any other *NP*-complete problem.

In 1982, Richard Feynmann, a Nobel laurate in physics suggested that scientists should start thinking of building computers based on the principles of quantum mechanics. The subject of physics studies elementary objects and simple systems and the study becomes more intersting when things are larger and more complicated. Quantum computation and information based on the principles of Quantum Mechanics will provide tools to fill up the gulf between the small and the relatively complex systems in physics. In this section we provide a brief survey of quantum computation and information and its impact on complexity theory.

Quantum mechanics arose in the early 1920s, when classical physics could not explain everything even after adding ad hoc hypotheses. The rules of quantum mechanics were simple but looked counterintuitive, and even Albert Einstein reconciled himself with quantum mechanics only with a pinch of salt.

Quantum Mechanics is real black magic calculus.

—A. Einstein

12.8.1 QUANTUM COMPUTERS

We know that a bit (a 0 or a 1) is the fundamental concept of classical computation and information. Also a classical computer is built from an electronic circuit containing wires and logical gates. Let us study quantum bits and quantum circuits which are analogous to bits and (classical) circuits.

A quantum bit, or simply qubit can be described mathematically as

$$|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$$

The qubit can be explained as follows. A classical bit has two states, a 0 and a 1. Two possible states for a qubit are the states $|0\rangle$ and $|1\rangle$. (The notation $|.\rangle$ is due to Dirac.) Unlike a classical bit, a qubit can be in infinite number of states other than $|0\rangle$ and $|1\rangle$. It can be in a state $|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The 0 and 1 are called the computational basis states and $|\psi\rangle$ is called a superposition. We can call $|\psi\rangle = \alpha|0\rangle + \beta|0\rangle$ a quantum state.

In the classical case, we can observe it as a 0 or a 1. But it is not possible to determine the quantum state on observation. When we measure/observe a qubit, we get either the state $|0\rangle$ with probability $|\alpha|^2$ or the state $|1\rangle$ with probability $|\beta|^2$.

This is difficult to visualize, using our 'classical thinking' but this is the source of power of the quantum computation.

Multiple qubits can be defined in a similar way. For example, a two-qubit system has four computational basis states, $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ and quantum states $|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$ with $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

Now we define the qubit gates. The classical NOT gate interchanges 0 and 1. In the case of the qubit the NOT gate, $\alpha|0\rangle + \beta|1\rangle$, is changed to $\alpha|1\rangle + \beta|0\rangle$.

The action of the qubit NOT gate is linear on two-dimensional complex vector spaces. So the qubit NOT gate can be described by

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

The matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ is a unitary matrix. (A matrix A is unitary if A adj$A = I$.)

We have seen earlier that {NOR} is functionally complete (refer to Exercises of Chapter 1). The qubit gate corresponding to NOR is the controlled-NOT or CNOT gate. It can be described by

$$|A, B\rangle \rightarrow |A, B \oplus A\rangle$$

where $\oplus$ denotes addition modulo 2. The action on computational basis is $|00\rangle \rightarrow |00\rangle$, $|01\rangle \rightarrow |01\rangle$, $|10\rangle \rightarrow |11\rangle$, $|11\rangle \rightarrow |10\rangle$. It can be described by the following 4×4 unitary matrix:

$$\cup_{CN} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Now, we are in a position to define a quantum computer:

A quantum computer is a system built from quantum circuits, containing wires and elementary quantum gates, to carry out manipulation of quantum information.

12.8.2 Church-Turing Thesis

Since 1970s many techniques for controlling the single quantum systems have been developed but with only modest success. But an experimental prototype for performing quantum cryptography, even at the initial level may be useful for some real-world applications.

Recall the Church–Turing thesis which asserts that any algorithm that can be performed on any computing machine can be performed on a Turing machine as well.

Miniaturization of chips has increased the power of the computer. The growth of computer power is now described by Moore's law, which states that the computer power will double for constant cost once in every two years. Now it is felt that a limit to this doubling power will be reached in two or three decades, since the quantum effects will begin to interfere in the functioning of electronic devices as they are made smaller and smaller. So efforts are on to provide a theory of quantum computation which will compensate for the possible failure of the Moore's law.

As an algorithm requiring polynomial time was considered as an efficient algorithm, a strengthened version of the Church–Turing thesis was enunciated.

Any algorithmic process can be simulated efficiently by a Turing machine.

But a challenge to the strong Church–Turing thesis arose from analog computation. Certain types of analog computers solved some problems efficiently whereas these problems had no efficient solution on a Turing machine. But when the presence of noise was taken into account, the power of the analog computers disappeared.

In mid-1970s, Robert Solovay and Volker Strassen gave a randomized algorithm for testing the primality of a number. (A deterministic polynomial algorithm was given by Manindra Agrawal, Neeraj Kayal and Nitein Saxena of IIT Kanpur in 2003.) This led to the modification of the Church thesis.

Strong Church-Turing Thesis

Any algorithmic process can be simulated efficiently using a nondeterministic Turing machine.

In 1985, David Deutsch tried to build computing devices using quantum mechanics.

> *Computers are physical objects, and computations are physical processes. What computers can or cannot compute is determined by the law of physics alone, and not by pure mathematics*
>
> —David Deutsch

But it is not known whether Deutsch's notion of universal quantum computer will efficiently simulate any physical process. In 1994, Peter Shor proved that finding the prime factors of a composite number and the discrete logarithm problem (i.e. finding the positive value of s such that $b = a^s$ for the given positive integers a and b) could be solved efficiently by a quantum computer. This may be a pointer to proving that quantum computers are more efficient than Turing machines (and classical computers).

12.8.3 Power of Quantum Computation

In classical complexity theory, the classes **P** and **NP** play a major role, but there are other classes of interest. Some of them are given below:

L—The class of all decision problems which may be decided by a TM running in logarithmic space.

PSPACE—The class of decision problems which may be decided on a Turing machine using a polynomial number of working bits, with no limitation on the amount of time that may be used by the machine.

EXP—The class of all decision problems which may be decided by a TM in exponential time, that is, $O(2^{n^k})$, k being a constant.

The hierarchy of these classes is given by

$$\mathbf{L} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$$

The inclusions are strongly believed to be strict but none of them has been proved so far in classical complexity theory.

We also have two more classes.

BPP—The class of problems that can be solved using the randomized algorithm in polynomial time, if a bounded probability of error (say 1/10) is allowed in the solution of the problem.

BQP—The class of all computational problems which can be solved efficiently (in polynomial time) on a quantum computer where a bounded probability of error is allowed. It is easy to see that **BPP** $\subseteq$ **BQP**. The class **BQP** lies somewhere between **P** and **PSPACE**, but where exactly it lies with respect to **P**, **NP** and **PSPACE** is not known.

It is easy to give non-constructive proofs that many problems are in **EXP**, but it seems very hard to prove that a particular class of problems is in **EXP** (the possibility of a polynomial algorithm of these problems cannot be ruled out).

As far as quantum computation is concerned, two important classes are considered. One is **BQP**, which is analogous to **BPP**. The other is **NPI** (**NP** intermediate) defined by

NPI — The class of problems which are neither in **P** nor *NP*-complete

Once again, no problem is shown to be in **NPI**. In that case **P** ≠ **NP** is established.

Two problems are likely to be in **NPI**, one being the factoring problem (i.e. given a composite number n to find its prime factors) and the other being the graph isomorphism problems (i.e. to find whether the given undirected graphs with the same set of vertices are isomorphic).

A quantum algorithm for factoring has been discovered. Peter Shor announced a quantum order-finding algorithm and proved that factoring could be reduced to order-finding. This has motivated a search for a fast quantum algorithm for other problems suspected to be in **NPI**.

Grover developed an algorithm called the quantum search algorithm. A loose formulation of this means that a quantum computer can search a particular item in a list of N items in $O(\sqrt{N})$ time and no further improvement is possible. If it were $O(\log N)$, then a quantum computer can solve an *NP*-complete problem in an efficient way. Based on this, some researchers feel that the class **BQP** cannot contain the class of *NP*-complete problems.

If it is possible to find some structure in the class of *NP*-complete problems then a more efficient algorithm may become possible. This may result in finding efficient algorithms for *NP*-complete problems. If it is possible to prove that quantum computers are strictly more powerful than classical computers, then it will follow that **P** is properly contained in **PSPACE**. Once again, there is no proof so far for $\mathbf{P} \subsetneq \mathbf{PSPACE}$.

12.8.4 Conclusion

Deutsch proposed the first blueprint of a quantum computer. As a single qubit can store two states 0 and 1 in quantum superposition, adding more qubits to the memory register will increase the storage capacity exponentially. When this happens, exponential complexity will reduce to polynomial complexity. Peter Shor's algorithm led to the hope that quantum computer may work efficiently on problems of exponential complexity.

But problems arise at the implementation stage. When more interacting qubits are involved in a circuit, the surrounding environment is affected by those interactions. It is difficult to prevent them. Also quantum computation will spread outside the computational unit and will irreversibly dissipate useful

information to the environment. This process is called *decoherence*. The problem is to make qubits interact with themselves but not with the environment. Some physicists are pessimistic and conclude that the efforts cannot go beyond a few simple experiments involving only a few qubits.

But some researchers are optimistic and believe that efforts to control decoherence will bear fruit in a few years rather than decades.

It remains a fact that optimism, however overstretched, makes things happen. The proof of Fermat's last theorem and the four colour problem are examples of these. Thomas Watson, the Chairman of IBM, predicted in 1943, "I think there is a world market for maybe five computers". But the growth of computers has very much surpassed his prediction.

Charles Babbage (1791–1871) conceived of most of the essential elements of a modern computer in his analytical engine. But there was not sufficient technology available to implement his ideas. In 1930s, Alan Turing and John von Neumann thought of a theoretical model. These developments in 'Software' were matched by 'Hardware' support, resulting in the first computer in the early 1950s. Then, the microprocessors in 1970s led to the design of smaller computers with more capacity and memory.

But computer scientists realized that hardware development will improve the power of a computer only by a multiplicative constant factor. The study of **P** and **NP** led to developing approximate polynomial algorithms to *NP*-complete problems. Once again the importance of software arose. Now the quantum computers may provide the impetus to the development of computers from the hardware side.

The problem of developing quantum computers seems to be very hard but the history of sciences indicates that quantum computers may rule the universe in a few decades.

12.9 SUPPLEMENTARY EXAMPLES

EXAMPLE 12.4

Suppose that there is an *NP*-complete problem P that has a deterministic solution taking $O(n^{\log n})$ time (here log n denotes $\log_2 n$). What can you say about the running time of any other *NP*-complete problem Q?

Solution

As $Q \in \mathbf{NP}$, there exists a polynomial $p(n)$ such that the time for reduction of Q to P is atmost $p(n)$. So the running time for Q is $O(p(n) + p(n)^{\log p(n)})$. As $p(n)^{\log p(n)}$ dominates $p(n)$, we can omit $p(n)$ in $p(n) + p(n)^{\log p(n)}$. If the degree of $p(n)$ is k, then $p(n) = O(n^k)$. So we can replace $p(n)$ by n^k. So $p(n)^{\log p(n)} = O((n^k)^{k \log n}) = O(n^{k^2 \log n})$. Hence the running time of Q is $O^{(c \log n)}$ for some constant c.

EXAMPLE 12.5

Show that **P** is closed under (a) union, (b) concatenation, and (c) complementation.

Solution

Let L_1 and L_2 be two languages in **P**. Let w be an input of length n.

(a) To test whether $w \in L_1 \cup L_2$, we test whether $w \in L_1$. This takes polynomial time $p(n)$. If $w \notin L_1$, test another $w \in L_2$. This takes polynomial time $q(n)$. The total time taken for testing whether $w \in L_1 \cup L_2$ is $p(n) + q(n)$, which is also a polynomial in n. Hence $L_1 \cup L_2 \in \mathbf{P}$.

(b) Let $w = x_1x_2 \ldots x_n$. For each k, $1 \leq k \leq n-1$, test whether $x_1x_2 \ldots x_k \in L_1$ and $x_{k+1}x_{k+2} \ldots x_n \in L_2$. If this happens, $w \in L_1L_2$. If the test fails for all k, $w \notin L_1L_2$. The time taken for this test for a particular k is $p(n) + q(n)$, where $p(n)$ and $q(n)$ are polynomials in n. Hence the total time for testing for all k's is at most n times the polynomial $p(n) + q(n)$. As $n(pn) + q(n)$ is a polynomial, $L_1L_2 \in \mathbf{P}$.

(c) Let M be the polynomial time TM for L_1. We construct a new TM M_1 as follows:

1. Each accepting state of M is a nonaccepting state of M_1 from which there are no further moves. So if M accepts w, M_1 on reading w will halt without accepting.
2. Let q_f be a new state, which is the accepting state of M_1. If $\delta(q, a)$ is not defined in M, define $\delta_{M_1}(q, a) = (q_f, a, R)$. So, $w \notin L$ if and only if M accepts w and halts. Also M_1 is a polynomial-time TM. Hence $L_1^c \in \mathbf{P}$.

EXAMPLE 12.6

Show that every language accepted by a standard TM M is also accepted by a TM M_1 with the following conditions:

1. M_1's head never moves to the left of its initial position.
2. M_1 will never write a blank.

Solution

It is easy to implement Condition 2 on the new machine. For the new TM, create a new blank b'. If the blank is written by M, the new Turing machine writes b'. The move of this new TM on seeing b' is the same as the move of M for b. The new TM satisfies the Condition 2. Denote the modified TM by M itself. Define the modified M by

$$M = (Q, \Sigma, \Gamma, \delta, q_2, b, F)$$

Define a new TM M_1 as

$$M_1 = (Q_1, \Sigma \times \{b\}, \Gamma_1, \delta_1, q_0, [b, b], F_1)$$

where

$$Q_1 = \{q_0, q_1\} \cup (Q \times \{U, L\})$$

$$\Gamma_1 = (\Gamma \times \Gamma) \cup \{[x, *] \mid x \in \Gamma\}$$

q_0 and q_1 are used to initiate the initial move of M. The two-way infinite tape of M is divided into two tracks as in Table 12.4. Here $*$ is the marker for the leftmost cell of the lower track. The state $[q, U]$ denotes that M_1 simulates M on the upper track. $[q, L]$ dentoes that M_1 simulates M on the lower track. If M moves to the left of the cell with $*$, M_1 moves to the right of the lower track.

TABLE 12.4 Folded Two-way Tape

X_0	X_1	X_2	.	. .	
$*$	X_{-1}	X_{-2}	X_{-3}	.	. .

We can define F_1 of M_1 by

$$F_1 = F \times \{U, L\}$$

We can describe δ as follows:

1. $\delta_1(q_0, [a, b]) = (q_1, [a, *], R)$

   $$\delta(q_1, [X, b]) = ([q_2, U], [X, b], L)$$

   By Rule 1, M_1 marks the leftmost cell in the lower track with $*$ and initiates the initial move of M.

2. If $\delta(q, X) = (p, Y, D)$ and $Z \in \Gamma$, then:

   (i) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ and

   (ii) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \bar{D})$

   where $\bar{D} = L$ if $D = R$ and $\bar{D} = R$ if $D = L$.

   By Rule 2, M_1 simulates the moves of M on the appropriate track. In (i) the action is on the upper track and Z on the lower track is not changed. In (ii) the action is on the lower track and hence the movement is in the opposite direction $\bar{D}$; the symbol in the upper track is not changed.

3. If $\delta(q, X) = (p, Y, R)$ then

   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$

   When M_1 see $*$ in the lower track, M moves right and simulates M on the upper track.

4. If $\delta(q, X) = (p, Y, L)$, then

   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$

   When M_1 sees $*$ in the lower track and M's movement is to the left of the cell of the two-way tape corresponding to the $*$ cell in the lower track, the M's movement is to X_{-1} and the M_1's movement is also to X_{-1} but towards the right. As the tape of M is folded on the

cell with $*$. the movement of M to the left of the $*$ cell is equivalent to the movement of M_1 to the right.

M reaches q in F if and only if M_1 reaches $[q, L]$ or $[q, R]$. Hence $T(M) = T(M_1)$.

EXAMPLE 12.7

We can define the 2SAT problem as the satisfiability problem for boolean expressions written as $\wedge$ of clauses having two or fewer variables. Show that 2SAT is in **P**.

Solution

Let the boolean expression E be an instance of the 2SAT problem having n variables.

Step 1 Let E have clauses consisting of a single variable (x_i or $\overline{x}_i$). If (x_i) appears as a clause in E, then x_i has to be assigned the truth value T in order to make E satisfiable. Assign the truth value T to x_i. Once x_i has the truth value T, then $(x_i \vee x_j)$ has the truth value T irrespective of the truth value of x_j (Note that x_j can also be $\overline{x}_j$). So $(x_i \vee x_j)$ or $(x_i \vee \overline{x}_j)$ can be deleted from E. If E contains $(\overline{x}_i \vee x_j)$ as a clause, then x_j should be assigned the truth value T in order to make E satisfiable. Hence we replace $(\overline{x}_i \vee x_j)$ by x_j in E so that x_j should be assigned the truth value T is order to make E satisfiable. Hence we replace $(\overline{x}_i \vee x_j)$ by x_j in E so that x_j can be assigned the truth value T later. If we repeat this process of eliminating clauses with a single variable (or its negation), we end up in two cases.

Case 1 We end up with $(x_i) \wedge (\overline{x}_i)$. In this case E is not satisfiable for any assignment of truth values. We stop.

Case 2 In this case all clauses of E have two variables. (A typical clause is $x_i \vee x_j$ or $x_i \vee \overline{x}_j$.)

Step 2 We have the apply step 2 only in Case 2 of step 1. We have already assigned truth values for variables not appearing in the reduced expression E. Choose one of the remaining variables appearing in E. If we have chosen x_i, assign the truth value T to x_i. Delete $x_i \vee x_j$ or $x_i \vee \overline{x}_j$ from E. If $\overline{x}_i \vee x_j$ appears in E, delete $\overline{x}_i$ to get (x_j). Repeat step 1 for clauses consisting of a single variable. If Case 1 occurs, assign the truth value F for x_i and proceed with E that we had before applying step 1.

Proceeding with these iterations, we end up either in unsatisfiability of E or satisfiability of E.

Step 2 consists of repetition of step 1 at most n times and step 1 requires $O(n)$ basic steps.

Let n be the number of clauses in E. Step 1 consists of deleting $(x_i \vee x_j)$ from E or deleting $\overline{x}_i$ from $(\overline{x}_i \vee x_j)$. This is done at most n times for each clause. In step 2, step 1 is applied at most two times, one for x_i and the second for $\overline{x}_i$. As the number of variables appearing in E is less than or equal to n, we delete $(x_i \vee x_j)$ or delete $\overline{x}_i$ from $(\overline{x}_i \vee x_j)$ at most $O(n)$ times while applying steps 1 and 2 repeatedly. Hence 2SAT is in **P**.

SELF-TEST

Choose the correct answer to Questions 1–7:

1. If $f(n) = 2n^3 + 3$ and $g(n) = 10000n^2 + 1000$, then:
   (a) the growth rate of g is greater than that of f.
   (b) the growth rate of f is greater than that of g.
   (c) the growth rate of f is equal to that of g.
   (d) none of these.
2. If $f(n) = n^3 + 4n + 7$ and $g(n) = 1000n^2 + 10000$, then $f(n) + g(n)$ is
   (a) $O(n^2)$
   (b) $O(n)$
   (c) $O(n^3)$
   (d) $O(n^5)$
3. If $f(n) = O(n^k)$ and $g(n) = O(n^l)$, then $f(n)g(n)$ is
   (a) $\max\{k, l\}$
   (b) $k + l$
   (c) kl
   (d) none of these.
4. The gcd of (1024, 28) is
   (a) 2
   (b) 4
   (c) 7
   (d) 14
5. $\lceil 10.7 \rceil + \lceil 9.9 \rceil$ is equal to
   (a) 19
   (b) 20
   (c) 18
   (d) none of these.
6. $\log_2 1024$ is equal to
   (a) 8
   (b) 9
   (c) 10
   (d) none of these.

7. The truth value of $f(x, y, z) = (x \vee \neg y) \wedge (\neg x \vee y) \wedge z$ is T if x, y, z have the truth values
   (a) T, T, T
   (b) F, F, F
   (c) T, F, F
   (d) F, T, F

State whether the following Statements 8–15 are true or false.

8. If the truth values of x, y, z are T, F, F respectively, then the truth value of $f(x, y, z) = x \wedge \neg(y \vee z)$ is T.
9. The complexity of a k-tape TM and an equivalent standard TM are the same.
10. If the time complexity of a standard TM is polynomial, then the time complexity of an equivalent k-tape TM is exponential.
11. If the time complexity of a standard TM is polynomial, then the time complexity of an equivalent NTM is exponential.
12. $f(x, y, z) = (x \vee y \vee z) \wedge (\neg x \wedge \neg y \wedge \neg z)$ is satisfiable.
13. $f(x, y, z) = (x \vee y) \wedge (\neg x \wedge \neg y)$ is satisfiable.
14. If f and g are satisfiable expressions, then $f \vee g$ is satisfiable.
15. If f and g are satisfiable expressions, then $f \wedge g$ is satisfiable.

EXERCISES

12.1 If $f(n) = O(n^k)$ and $g(n) = O(n^l)$, then show that $f(n) + g(n) = O(n^t)$ where $t = \max\{k, l\}$ and $f(n)g(n) = O(n^{k+l})$.

12.2 Evaluate the growth rates of (i) $f(n) = 2n^2$, (ii) $g(n) = 10n^2 + 7n \log n + \log n$, (iii) $h(n) = n^2 \log n + 2n \log n + 7n + 3$ and compare them.

12.3 Use the O-notation to estimate (i) the sum of squares of first n natural numbers, (ii) the sum of cubes of first n natural numbers, (iii) the sum of the first n terms of a geometric progression whose first term is a and the common ratio is r, and (iv) the sum of the first n terms of the arithmetic progression whose first term is a and the common difference is d.

12.4 Show that $f(n) = 3n^2 \log_2 n + 4n \log_3 n + 5 \log_2 \log_2 n + \log n + 100$ dominates n^2 but is dominated by n^3.

12.5 Find the gcd (294, 15) using the Euclid's algorithm.

12.6 Show that there are five truth assignments for (P, Q, R) satisfying $P \vee (\neg P \wedge \neg Q \wedge R)$.

12.7 Find whether $(P \wedge Q \wedge R) \wedge \neg Q$ is satisfiable.

12.8 Is $f(x, y, z, w) = (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y} \vee \overline{z})$ satisfiable?

12.9 The set of all languages whose complements are in **NP** is called **CO-NP**. Prove that **NP** = **CO-NP** if and only if there is some *NP*-complete problem whose complement is in **NP**.

12.10 Prove that a boolean expression E is a tautology if and only if $\neg E$ is unsatisfiable (refer to Chapter 1 for the definition of tautology).

Answers to Self-Tests

Chapter 1

1. (d) **2.** (a) **3.** (c) **4.** (c)
5. (b) **6.** *F* **7.** *F, T, F; F, T, T* **8.** *T*

Chapter 2

1. (b) **2.** (c) **3.** (a) **4.** (b)
5. (c) **6.** (b) **7.** (c) **8.** (d)
9. (d) **10.** (d)

Chapter 3

1. (d) **2.** (a) **3.** (d) **4.** (a)
5. (d) **6.** *T* **7.** *F* **8.** *T*
9. *T* **10.** *F* **11.** *T* **12.** *T*
13. *F* **14.** *T* **15.** *F*

Chapter 4

1. (b) **2.** (d) **3.** (c) **4.** (a)
5. (b) **6.** (a) **7.** (d) **8.** (a)
9. (a) **10.** (b) **11.** (c) **12.** (b)
13. *F* **14.** *T* **15.** *F* **16.** *T*
17. *F* **18.** *T* **19.** *F* **20.** *F*

Chapter 5

1. (a) **2.** (d) **3.** (a) **4.** (d)
5. (a) **6.** (d) **7.** (b) **8.** (b)
9. (d) **10.** (a) **11.** *F* **12.** *F*
13. *F* **14.** *T* **15.** *F* **16.** *T*
17. *F*

Chapter 6

1. (a) A (b) Yes (c) Yes
(e) Lables for nodes 1–14 are A, b, A, A, A, b, a, a, A, A, b, A, a, a.

2. (a) F (b) T (c) T (d) F
(e) T

3. (a) T (b) T (c) T (d) F
(e) F (f) T (g) F (h) T

Chapter 7

1. (a) **2.** (b) **3.** (d) **4.** (a)
5. (c) **6.** (a) **7.** input string to S
8. looking ahead by one symbol
9. $(q_f, \wedge, \alpha)$ for some $q_f \in F$ and $\alpha \in \Gamma^*$
10. $(q, \wedge, \wedge)$ for some state q.

Chapter 8

1. (c) **2.** (a) **3.** (a) **4.** (c)
5. (b)

Chapter 9

1. (d) **2.** (b) **3.** (a) **4.** (d)
5. (a) **6.** (c) **7.** (b) **8.** (d)
9. (b) **10.** (b)

Chapter 11

1. (a) **2.** (c) **3.** (a) **4.** (b)
5. (b) **6.** (a) **7.** (c) **8.** (d)
9. (a) **10.** (b) **11.** T **12.** T
13. F **14.** T **15.** T

Chapter 12

1. (b) **2.** (c) **3.** (b) **4.** (b)
5. (a) **6.** (c) **7.** (a) **8.** T
9. F **10.** F **11.** T **12.** F
13. T **14.** T **15.** F

Solutions (or Hints) to Chapter-end Exercises

Chapter 1

1.1 All the sentences except (g) are propositions.

1.2 Let L, E, and G denote $a < b$, $a = b$ and $a > b$ respectively. Then the sentence can be written as

$$(E \wedge \neg G \wedge \neg L) \bar{\vee} (G \wedge \neg E \wedge \neg L) \bar{\vee} (L \wedge \neg E \wedge \neg G).$$

1.3 (i) David gets a first class or he does not get a first class. Using the truth table given in Table A1.1, $\bar{\vee}$ is associative since the columns corresponding to $(P \bar{\vee} Q) \bar{\vee} R$ and $P \bar{\vee} (Q \bar{\vee} R)$ coincide.

TABLE A1.1 Truth Table for Exercise 1.3

P	Q	R	$P \bar{\vee} Q$	$Q \bar{\vee} R$	$(P \bar{\vee} Q) \bar{\vee} R$	$P \bar{\vee} (Q \bar{\vee} R)$
T	T	T	F	F	T	T
T	T	F	F	T	F	F
T	F	T	T	T	F	F
T	F	F	T	F	T	T
F	T	T	T	F	F	F
F	T	F	T	T	T	T
F	F	T	F	T	T	T
F	F	F	F	T	F	F

The commutative and distributive properties of Exclusive OR can be proved similarly.

1.4 Using $\vee$ and $\neg$, all other connectives can be described. $P \wedge Q$ and $P \Rightarrow Q$ can be expressed as $\neg (\neg P \vee \neg Q)$ and $\neg P \vee Q$ respectively.

1.5 $\neg P = P \uparrow P$

$P \wedge Q = (P \uparrow Q) \uparrow (P \uparrow Q)$

$P \vee Q = (P \uparrow P) \uparrow (Q \uparrow Q)$

Verify these three equations using the truth tables.

1.6 $\neg P = P \downarrow P$

$P \vee Q = (P \downarrow Q) \downarrow (P \downarrow Q)$

$P \wedge Q = (P \downarrow P) \downarrow (Q \downarrow Q)$

1.7 (a) The truth table is given in Table A1.2.

TABLE A1.2 Truth Table for Exercise 1.7(a)

P	Q	R	$P \vee Q$	$P \vee R$	$R \vee Q$	$P \vee R \Rightarrow R \vee Q$	$P \vee Q \Rightarrow ((P \vee R) \Rightarrow (R \vee Q))$
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	T	T	T	T
T	F	F	T	T	F	F	F
F	T	T	T	T	T	T	T
F	T	F	T	F	T	T	T
F	F	T	F	T	T	T	T
F	F	F	F	F	F	T	T

1.8 (a) $\neg P \Rightarrow (\neg P \wedge Q) \equiv \neg(\neg P) \vee (\neg P \wedge Q)$ by I_{12}

$\equiv P \vee (\neg P \wedge Q)$ by I_7

$\equiv (P \vee \neg P) \wedge (P \vee Q)$ by I_4

$\equiv \mathbf{T} \wedge (P \vee Q)$ by I_8

$\equiv P \vee Q$ by I_9

$\neg P \Rightarrow (\neg P \Rightarrow (\neg P \wedge Q)) \equiv \neg(\neg P) \vee (P \vee Q)$ by I_{12}

$\equiv P \vee (P \vee Q)$ by I_7

$\equiv P \vee Q$ by I_3 and I_1

1.9 We prove I_5 and I_6 using the truth table.

TABLE A1.3 Truth Table for Exercise 1.9

P	Q	$P \wedge Q$	$\neg P$	$\neg Q$	$P \vee (P \wedge Q)$	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$
T	T	T	F	F	T	F	F
T	F	F	F	T	T	T	T
F	T	F	T	F	F	T	T
F	F	F	T	T	F	T	T

$P \vee (P \wedge Q) \equiv P$ since the columns corresponding to P and $P \vee (P \wedge Q)$ are identical; $\neg(P \wedge Q) = \neg P \vee \neg Q$ is true since the columns corresponding to $\neg(P \wedge Q)$ and $\neg P \vee \neg Q$ are identical.

1.11 We construct the truth table for $(P \Rightarrow \neg P) \Rightarrow \neg P$.

TABLE A1.4 Truth Table for Exercise 1.11

P	$\neg P$	$P \Rightarrow \neg P$	$(P \Rightarrow \neg P) \Rightarrow \neg P$
T	F	F	T
F	T	T	T

As the column corresponding to $(P \Rightarrow \neg P) \Rightarrow \neg P$ has T for all combinations, $(P \Rightarrow \neg P) \Rightarrow \neg P$ is a tautology.

1.12 $P \wedge (P \Rightarrow \neg Q) \equiv P \wedge (\neg P \vee \neg Q) \equiv (P \wedge \neg P) \vee (P \wedge \neg Q) \equiv \mathbf{F} \vee (P \wedge \neg Q) \equiv P \wedge \neg Q$

So $(P \wedge (P \Rightarrow \neg Q)) \vee (Q \Rightarrow \neg Q) \equiv (P \wedge \neg Q) \vee (\neg Q \vee \neg Q) \equiv (P \wedge \neg Q) \vee \neg Q \equiv \neg Q$. Hence

$((P \wedge (P \Rightarrow \neg Q)) \vee (Q \Rightarrow \neg Q)) \Rightarrow \neg Q \equiv (\neg Q \Rightarrow \neg Q) \equiv \neg (\neg Q) \vee \neg Q \equiv Q \vee \neg Q = \mathbf{T}$

1.13 $\alpha \equiv (Q \wedge \neg R \wedge \neg S) \vee (R \wedge S)$
$\equiv (Q \wedge \neg R \wedge \neg S) \vee (R \wedge S \wedge Q) \vee (R \wedge \mathrm{S} \wedge \neg Q)$
$\equiv (Q \wedge \neg R \wedge \neg S) \vee (Q \wedge R \wedge S) \vee (\neg Q \wedge R \wedge S)$

1.14 Let the literals be P, Q, R. Then
$\alpha \equiv 110 \vee 100 \vee 010 \vee 000$
$\equiv ((P \wedge Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R)) \vee (010 \vee 000)$
$\equiv (P \wedge \neg R) \vee (\neg P \wedge Q \wedge \neg R) \vee (\neg P \wedge \neg Q \wedge \neg R))$
$\equiv (P \wedge \neg R) \vee (\neg P \wedge \neg R)$
$\equiv \neg R$

1.15 (a) The given premises are: (i) $P \Rightarrow Q$, (ii) $R \Rightarrow \neg Q$. To derive $P \Rightarrow \neg R$, we assume (iii) P as an additional premise and deduce $\neg R$.

1. P	Premise (iii)
2. $P \Rightarrow Q$	Premise (i)
3. Q	RI_4
4. $\neg (\neg Q)$	I_7
5. $R \Rightarrow \neg Q$	Premise (ii)
6. $\neg R$	RI_5
7. $P \Rightarrow \neg R$	Lines 1 and 6

Hence the argument is valid.

(b) Valid

(c) Let the given premises be (i) P, (ii) Q, (iii) $\neg Q \Rightarrow R$, (iv) $Q \Rightarrow \neg R$. Then

1. Q	Premise (ii)
2. $\neg R$	Premise (iv)

Hence the given argument is valid.

(d) Let the given premises be (i) S, (ii) P, (iii) $P \Rightarrow Q \wedge R$, (iv) $Q \vee \mathrm{S} \Rightarrow T$. Then

1. P	Premise (ii)
2. $Q \wedge R$	Premise (iii)
3. Q	RI_3
4. $Q \vee S$	RI_1
5. T	Premise (iv)

Hence the argument is valid. Note that in (c) and (d) the conclusions are obtained without using some of the given premises.

1.16 We name the propositions in the following way:

R denotes 'Ram is clever'.
P denotes 'Prem is well-behaved'.
J denotes 'Joe is good'.
S denotes 'Sam is bad'.
L denotes 'Lal is educated'.

The given premises are (i) $R \Rightarrow P$, (ii) $J \Rightarrow S \wedge \neg P$, (iii) $L \Rightarrow J \vee R$. We have to derive $L \wedge \neg P \Rightarrow S$. Assume (iv) $L \wedge \neg P$ as an additional premise.

1.	$L \wedge \neg P$	Premise (iv)
2.	L	RI_3
3.	$\neg P$	RI_3
4.	$J \vee R$	Premise (iii)
5.	$R \Rightarrow P$	Premise (i)
6.	$\neg R$	Line 3, Premise (i) and RI_5
7.	J	Lines 5 and 4 and RI_6
8.	$S \wedge \neg P$	Premise (ii)
9.	S	RI_3

Hence, $L \wedge \neg P \Rightarrow S$.

1.17 The candidate should be a graduate and know Visual Basic, JAVA and C++.

1.18 {5, 6, 7, . . .}.

1.19 Let the universe of discourse be the set of all complex numbers. Let $P(x)$ denote 'x is a root of $t^2 + at + b = 0$'. Let a and b be nonzero real numbers and $b \neq 1$. Let $P(x)$ denote 'x is a root of $t^2 + at + b = 0$'. Let $Q(x)$ denote 'x is a root of $bt^2 + at + 1 = 0$'. If x is a root of $t^2 + at + b = 0$ then $1/x$ is a root of $bt^2 + at + 1 = 0$. But x is a root of $t^2 + at + b = 0$ as well as that of $bt^2 + at + 1 = 0$ only when $x = \pm 1$. This is not possible since $b \neq 1$. So, $\exists x\,(P(x) \Rightarrow Q(x)) \Leftrightarrow (\exists x\,P(x) \Rightarrow \exists x\,Q(x))$ is not valid.

1.20 Similar to Example 1.22.

1.21 Let the universe of discourse be the set of all persons. Let $P(x)$ denote 'x is a philosopher'. Let $Q(x)$ denote 'x is not money-minded'. Let $R(x)$ denote 'x is not clever'. Then the given sentence is

$$(\forall x\,(P(x) \Rightarrow Q(x))) \wedge (\exists x(\neg Q(x) \wedge R(x))) \Rightarrow (\exists x(\neg P(x) \wedge R(x)))$$

1.	$\neg Q(c) \wedge R(c)$	RI_{14}
2.	$\neg Q(c)$	RI_3
3.	$\neg P(c)$	RI_5
4.	$R(c)$	Line I and RI_3
5.	$\neg Q(c) \wedge R(c)$	Lines 2 and 4

Hence the given sentence is true.

1.22 Similar to Exercise 1.21.

Chapter 2

2.1 (a) The set of all strings in $\{a, b, c\}^*$.
(b) $A \cap B = \{b\}$. Hence $(A \cap B)^* = \{b^n \mid n \geq 0\}$.
(c) The set of all strings in $\{a, b, c\}^*$ which are in $\{a, b\}^*$ or in $\{b, c\}^*$.
(d) $A^* \cap B^* = \{b^n \mid n \geq 0\}$.
(e) $A - B = \{a\}$. Hence $(A - B)^* = \{a^n \mid n \geq 0\}$.
(f) $(B - A)^* = \{c^n \mid n \geq 0\}$.

2.2 (a) Yes
(b) Yes
(c) Yes. The identity element is Λ.
(d) o is not commutative since x o $y \neq y$ o x when $x = ab$ and $y = ba$; in this case x o $y = abba$ and y o $x = baab$.

2.3 (a) o is commutative and associative. (b) $\emptyset$ is the identity element with respect to o. (c) $A \cup B = A \cup C$ does not imply that $B = C$. For example, take $A = \{a, b\}$, $B = \{b, c\}$ and $C = \{c\}$. Then $A \cup B = A \cup C = \{a, b, c\}$. Obviously, $B \neq C$.

2.4 (a) True. 1 is the identity element.
(b) False. 0 does not have an inverse.
(c) True. 0 is the identity element.
(d) True. $\emptyset$ is the identity element. The inverse of A is A^c.

2.5 (c) Obviously, mRm. If mRn, then $m - n = 3a$. So, $n - m = 3(-a)$. Hence nRm. If mRn and nRp, then $m - n = 3a$ and $n - p = 3b$. $m - p = 3(a + b)$, i.e. mRp.

2.6 (a) R *is* not reflexive.
(b) R is neither reflexive nor transitive.
(c) R is not symmetric since $2R4$, whereas $4R'2$.
(d) R is not reflexive since $1R'1$ $(1 + 1 \neq 10)$.

2.7 An equivalence class is the set of all strings of the same length. There is an equivalence class corresponding to each non-negative number. For a non-negative number n, the corresponding equivalence class is the set of all strings of length n.

2.8 R is not an equivalence relation since it is not symmetric, for example, $abRaba$, whereas $abaR'ab$.

2.9 $R = \{(1, 2), (2, 3), (1, 4), (4, 2), (3, 4)\}$
$R^2 = \{(1, 3), (2, 4), (1, 2), (4, 3), (3, 2)\}$
$R^3 = \{(1, 4), (2, 2), (1, 3), (4, 4), (3, 3)\}$
$R^4 = \{(1, 2), (2, 3), (1, 4), (4, 2), (3, 4)\} = R$
Hence
$R^+ = R \cup R^2 \cup R^3$
$R^* = R^+ \cup \{(1, 1)\}$

2.11 $R^+ = R^* = R$, Since $R^2 = R$ (an equivalence relation is transitive).

2.12 Suppose $f(x) = f(y)$. Then $ax = ay$. So, $x = y$. Therefore, f is one-one. f is not onto as any string with b as the first symbol cannot be written as $f(x)$ for any $x \in \{a, b\}^*$.

2.14 (a) Tree given in Fig. 2.9.

2.15 (a) Yes.
(b) 4, 5, 6 and 8
(c) 1, 2, 3 and 7
(d) 3 (The longest path is $1 \rightarrow 3 \rightarrow 7 \rightarrow 8$)
(e) 4–5–6–8
(f) 2
(g) 6 and 7

2.16 Form a graph G whose vertices are persons. There is an edge connecting A and B if A knows B. Apply Theorem 2.3 to graph G.

2.17 Proof is by induction on $|X|$. When $|X| = 1$, X is a singleton. Then $2^X = \{\emptyset, X\}$. There is basis for induction. Assume $|2^X| = 2^{|X|}$ when X has $n - 1$ elements. Let $Y = \{a_1, a_2, \ldots, a_n\}$

$Y = X \cup \{a_n\}$, where $X = \{a_1, a_2, \ldots, a_{n-1}\}$. Then X has $n - 1$ elements. As X has $n - 1$ elements, $|2^X| = 2^{|X|}$ by induction hypothesis.

Take any subset Y_1 of Y. Either Y_1 is a subset of X or $Y_1 - \{a_n\}$ is a subset of X. So each subset of Y gives rise to two subsets of X. Thus, $|2^Y| = 2|2^X|$. But $|2^X| = 2^{|X|}$. Hence $|2^Y| = 2^{|Y|}$. By induction the result is true for all sets X.

2.18 (a) When $n = 1$, $1^2 = \dfrac{1(1+1)(1+2)}{6} = 1$. Thus there is basis for induction. Assume the result for $n - 1$. Then

$$\sum_{k=1}^{n} k^2 = \sum_{k=1}^{n-1} k^2 + n^2$$

$$= \frac{(n-1)(n-1+1)(2n-1)}{6} + n^2 \quad \text{[by induction hypothesis]}$$

$$= \frac{n(n+1)(2n+2)}{6} \quad \text{on simplification.}$$

Thus the result is true for n.
(c) When $n = 2$, $10^{2n} - 1 = 9999$ which is divisible by 11. Thus there is basis for induction. Assume $10^{2(n-1)} - 1$ is divisible by 11. Then, $10^{2n} - 1 = 10^2 10^{2(n-1)} - 1 = 10^2[10^{2(n-1)} - 1] + 10^2 - 1$. As $10^{2(n-1)} - 1$ and $10^2 - 1$ are divisible by 11, $10^{2n} - 1$ is divisible by 11. Thus the result is true for n.

2.19 (b) $2^2 > 2$ [Basis for induction]. Assume $2^{n-1} > n - 1$ for $n > 2$. Then, $2^n = 2 \cdot 2^{n-1} > 2(n - 1)$, i.e. $2^n > n + n - 2 > n$ (since $n > 2$). The result is true for n. By the principle of induction, the result is true for all $n > 1$.

2.20 (a) When $n = 1$, $F(2n + 1) = F(3) = F(1) + F(2) = F(0) + F(2)$. So,

$$F(2n + 1) = \sum_{k=0}^{1} F(2k).$$

Thus there is basis for induction. Assume

$$F(2n - 1) = \sum_{k=0}^{n-1} F(2k) \qquad \text{[by induction hypothesis]}$$

$$F(2n + 1) = F(2n - 1) + F(2n) \qquad \text{[by definition]}$$

By induction hypothesis,

$$F(2n + 1) = \sum_{k=0}^{n-1} F(2k) + F(2n) = \sum_{k=0}^{n} F(2k)$$

So the result is true for n.

2.21 In a simple graph, any edge connects two distinct nodes. The number of ways of choosing two nodes out of n given nodes is ${}^nC_2 = \frac{n(n-1)}{2}$. So the maximum number of edges in a simple graph is $\frac{n(n-1)}{2}$.

2.22 We prove by induction on $|w|$. When $w = \Lambda$, we have $ab\Lambda = \Lambda ab$. Clearly, $|\Lambda| = 0$, which is even. Thus there is basis for induction. Assume the result for all w with $|w| < n$. Let w be of length n and $abw = wab$. As $abw = wab$, $w = abw_1$ for some w in $\{a, b\}^*$. So $ababw_1 = abw_1ab$ and hence $abw_1 = w_1ab$. By induction hypothesis, $|w_1|$ is even. As $|w| = |w_1| + 2$, $|w|$ is also even. Hence by the principle of induction, the result is true for all w.

2.23 Let $P(n)$ be the 'open the nth envelope'. As the person opens the first envelope, $P(1)$ is true. Assume $P(n - 1)$ is true. Then the person follows the instruction contained therein. So the nth envelope is opened, i.e. $P(n)$ is true. By induction, $P(n)$ is true for all n.

Chapter 3

3.1 101101 and 000000 are accepted by M. 11111 is not accepted by M.

3.2 $\{q_0, q_1, q_4\}$. Now, $\delta(q_0, 010) = \{q_0, q_3\}$ and so 010 is not accepted by M.

3.3 Both the strings are not accepted by M.

3.4 As $\delta(q_1, a) = \delta(q_1, a)$, R is reflexive. Obviously it is symmetric. If q_1Rq_2, then $\delta(q_1, a) = \delta(q_2, a)$.
If q_2Rq_3 then $\delta(q_2, a) = \delta(q_3, a)$. Thus $\delta(q_1, a) = \delta(q_3, a)$, implying that q_1Rq_3. So R is an equivalence relation.

3.5 The state table of NDFA accepting $\{ab, ba\}$ is defined by Table A3.1.

TABLE A3.1 State Table for Exercise 3.5

State/Σ	a	b
q_0	q_1	q_2
q_1		q_3
q_2	q_3	
(q_3)		

The state table of the corresponding DFA is defined by Table A3.2.

TABLE A3.2 State Table of DFA for Exercise 3.5

State/Σ	a	b
$[q_0]$	$[q_1]$	$[q_2]$
$[q_1]$	$\emptyset$	$[q_3]$
$[q_2]$	$[q_3]$	$\emptyset$
$([q_3])$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$

3.6 The NDFA accepting the given set of strings is described by Fig. A3.1. The corresponding state table is defined by Table A3.3.

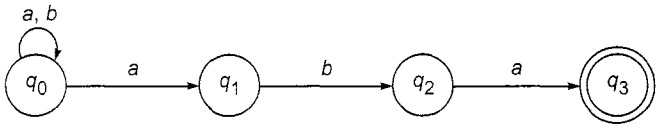


Fig. A3.1 NDFA for Exercise 3.6.

TABLE A3.3 State Table for Exercise 3.6

State/Σ	a	b
q_0	q_0, q_1	q_0
q_1		q_2
q_2	q_3	
q_3		

The DFA accepting the given set is defined by Table A3.4.

TABLE A3.4 State Table of DFA for Exercise 3.6

State/Σ	a	b
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1, q_3]$	$[q_0]$
$[q_0, q_1, q_3]$	$[q_0, q_1]$	$[q_0, q_2]$

3.7 The state table for the required DFA is defined by Table A3.5.

TABLE A3.5 State Table for Exercise 3.7

State	0	1	2
$[q_0]$	$[q_1\ q_4]$	$[q_4]$	$[q_2, q_3]$
$[q_4]$	$\emptyset$	$\emptyset$	$\emptyset$
$[q_1, q_4]$	$\emptyset$	$[q_4]$	$\emptyset$
$[q_2, q_3]$	$\emptyset$	$[q_4]$	$[q_2, q_3]$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

3.9 The state table for the required DFA is defined by Table A3.6.

TABLE A3.6 State Table for Exercise 3.9

State	0	1
$[q_1]$	$[q_2, q_3]$	$[q_1]$
$[q_2, q_3]$	$[q_1, q_2]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_1, q_2, q_3]$	$[q_1]$
$[q_1, q_2, q_3]$	$[q_1, q_2, q_3]$	$[q_1, q_2]$

3.10 The required transition system is given in Fig. A3.2. Let Σ denote $\{a, b, c, \ldots, z\}$. * denotes any symbol in $\Sigma - \{c, r\}$. ** denotes any symbol in $\Sigma - \{c, a, r\}$. *** denotes any symbol in Σ.

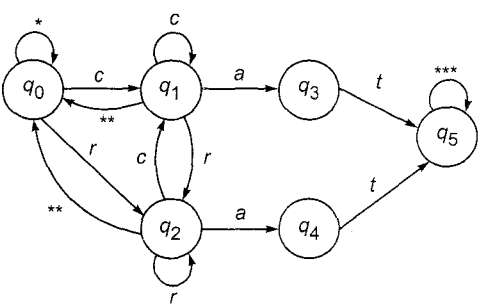


Fig. A3.2 Transition system for Exercise 3.10

3.11 The corresponding Mealy machine is defined by Table A3.7.

TABLE A3.7 Mealy Machine of Exercise 3.11

Present state	*Next state*			
	$a = 0$		$a = 1$	
	state	*output*	*state*	*output*
q_0	q_1	0	q_2	1
q_1	q_3	1	q_2	1
q_2	q_2	1	q_1	0
q_3	q_0	1	q_3	1

3.12 q_1 is associated with 1 and q_2 is associated with 0 and 1. Similarly, q_3 is associated with 0 and 1, whereas q_4 is associated with 1. The state table with new states q_1, q_{20}, q_{21}, q_{30}, q_{31} and q_4 is defined by Table A3.8.

TABLE A3.8 State Table for Exercise 3.12

Present state	*Next state*			
	$a = 0$		$a = 1$	
	state	*output*	*state*	*output*
q_1	q_1	0	q_{20}	0
q_{20}	q_4	1	q_4	1
q_{21}	q_4	1	q_4	1
q_{30}	q_{21}	1	q_{31}	1
q_{31}	q_{21}	1	q_{31}	1
q_4	q_{30}	0	q_1	1

The revised state table is defined by Table A3.9.

TABLE A3.9 Revised State Table for Exercise 3.12

Present state	*Next state*		
	$a = 0$	$a = 1$	*output*
→ q_0	q_1	q_{20}	0
q_1	q_1	q_{20}	1
q_{20}	q_4	q_4	0
q_{21}	q_4	q_4	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_{30}	q_1	1

3.13 The Mealy machine is described by Fig. A3.3.

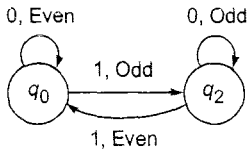


Fig. A3.3 Mealy machine of Exercise 3.13.

3.14 π_i's are given below:

$\pi_0 = \{\{q_6\}, \{q_0, q_1, q_2, q_3, q_4, q_5\}\}$

$\pi_1 = \{\{q_6\}, \{q_0, q_1, q_2, q_3, q_5\}, \{q_4\}\}$

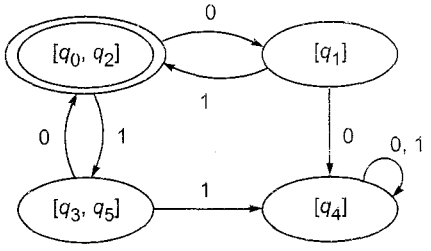
$\pi_2 = \{\{q_6\}, \{q_4\}, \{q_0, q_1, q_3\}, \{q_2, q_5\}\}$

$\pi_3 = \{\{q_6\}, \{q_4\}, \{q_0\}, \{q_1\}, \{q_3\}, \{q_2, q_5\}\}$

$\pi_4 = \{\{q_6\}, \{q_4\}, \{q_0\}, \{q_1\}, \{q_3\}, \{q_2\}, \{q_5\}\}$

Here $\pi = Q$. The minimum state automaton is simply the given automaton.

3.16



Chapter 4

4.1 (a) $S \overset{*}{\Rightarrow} 0^n S 1^n \overset{*}{\Rightarrow} 0^n 0^m A 1^m 1^n$, $n \geq 0$, $m \geq 1$.

$A \overset{*}{\Rightarrow} 1^k A \Rightarrow 1^{k+1}$, $k \geq 0$.

$0^m 1^n \in L(G)$ when $n > m \geq 1$. So $L(G) = \{0^m 1^n : n > m \geq 1\}$

(b) $L(G) = \{0^m 1^n \mid m \neq n$ and at least one of m and $n \geq 1\}$. Clearly, $0^m \in L(G)$ and $1^n \in L(G)$, where $m, n \geq 1$.

For $m > n$, $S \overset{*}{\Rightarrow} 0^n S 1^n \Rightarrow 0^n 0 A 1^n \overset{*}{\Rightarrow} 0^n 00^{m-n-1} 1^n = 0^m 1^n$. Thus $0^m 1^n \in L(G)$.

(c) $L(G) = \{0^n 1^n 0^n \mid n \geq 1\}$. The proof is similar to that of Example 4.10.

(d) $L(G) = \{0^n 1^m 0^m 1^n \mid m, n \geq 1\}$.

For $m, n \geq 1$,

$S \overset{*}{\Rightarrow} 0^{n-1} S 1^{n-1} \Rightarrow 0^{n-1} 0 A 1 1^{n-1} \Rightarrow 0^n 1^{m-1} A 0^{m-1} 1^{n-1} \Rightarrow 0^n 1^m 0^m 1^n$

So,

$\{0^n 1^m 0^m 1^n \mid m, n \geq 1\} \subseteq L(G)$.

It is easy to prove the other inclusion.

(e) $L(G) = \{x \in \{0, 1\}^+ \mid x$ does not contain two consecutive 0's$\}$

4.2 (a) $G = (\{S, A, B\}, \{0, 1\}, P, S)$, where P consists of $S \rightarrow 0B \mid 1A$, $A \rightarrow 0 \mid 0S \mid 1AA$, $B \rightarrow 1 \mid 1S \mid 0BB$.

Prove by induction on $|w|$, $w \in \Sigma^*$, that

(i) $S \overset{*}{\Rightarrow} w$ if and only if w consists of an equal number of 0's and 1's

(ii) $A \overset{*}{\Rightarrow} w$ if and only if w has one more 0 than it has 1's.

(iii) $B \overset{*}{\Rightarrow} w$ if and only if w has one more 1 than it has 0's.

$A \Rightarrow 0$, $B \Rightarrow 1$ and S does not derive any terminal string of length one.

Thus there is basis for induction. Assume (i), (ii) and (iii) are true for strings of length $k - 1$. Let w be a string in Σ^* with $|w| = k$. Suppose $S \Rightarrow w$. The first step in this derivation is either $S \Rightarrow 0B$ or $S \Rightarrow 1A$. In the first case $w = 0w_1$, $B \overset{*}{\Rightarrow} w_1$ and $|w_1| = k - 1$. By induction hypothesis, w has one more 1 than it has 0's. Hence w has an equal number of 0's and 1's. To prove the converse part, assume w has an equal number of 0's and 1's and $|w| = k$. If w starts with 0, then $w = 0w_1$ where $|w_1| = k - 1$. w_1 has one more 1 than it has 0's. By induction hypothesis, $B \overset{*}{\Rightarrow} w_1$. Hence $S \Rightarrow 0B \Rightarrow 0w_1 = w$. Thus (i) is proved for all strings w. The proofs for (ii) and (iii) are similar.

(b) The required grammar G has the following productions:

$$S \to 0S1,\ S \to 0A1,\ A \to 1A0,\ A \to 10.$$

Obviously, $L(G) \subseteq \{0^n1^m0^m1^n \mid m, n \geq 1\}$. For getting any terminal string, the first production is $S \to 0S1$ or $S \to 0A1$, the last production is $A \to 10$. By applying $S \to 0S1$ $(n - 1)$ times, $S \to 0A1$ (once), $A \to 1A0$ $(m - 1)$ times, and $A \to 10$ (once) we get $0^n1^m0^m1^n$. So, $\{0^n1^m0^m1^n \mid m, n \geq 1\} \subseteq L(G)$.

(c) The required productions are $S \to 0S11 \mid 011$.

(d) The required productions are $S \to 0A1 \mid 1B0$, $A \to 0A1 \mid \Lambda$, $B \to 1B0 \mid \Lambda$.

(e) Modify the constructions given in Example 4.7 to get the required grammar.

4.3 For the derivation of 001100, 001010 or 01010, the first production cannot be $S \to 0S1$. The other possible productions are $S \to 0A$ and $S \to 1B$. In these cases the resulting terminal strings are 0^n or 1^n. So none of the given strings are in the language generated by the grammar given in Exercise 4.1(b).

4.4 It is easy to see that any derivation should start with $S \Rightarrow 0AB \Rightarrow 0ASA \Rightarrow 0A0ABA$ or $S \Rightarrow 0AB \Rightarrow 0A01 \Rightarrow 0S0B1$. If we apply $S \to 0AB$, we get A in the sentential. form. If we try to eliminate A using $A0 \to S0B$ or $A1 \to SB1$, we get S in the sentential form. So one of the two variables, namely A or S, can never be eliminated.

4.5 The language generated by the given grammar is $\{01^m2^n3 \mid m, n \geq 1\}$. This language can also be generated by a regular grammar (refer to Chapter 5).

4.6 (a) False. Refer to Remark 2 on page 123.

(b) True. If $L = \{w_1, w_2, \ldots, w_n\}$, then $G = (\{S\}, \Sigma, P, S)$, where P consists of $S \to w_1 \mid w_2 \mid \ldots \mid w_n$.

(c) True. By Theorem 4.5 it is enough to show that $\{w\}$ is regular where $w \in \Sigma^*$. Let $w = a_1a_2 \ldots a_n$. Then the grammar whose

productions are $S \to a_1A_1$, $A_1 \to a_2A_2$, ... $A_{m-1} \to a_m$ generate $\{w\}$.

4.7 We prove (a) $S \Rightarrow A_1^nA_2^nA_4$, (b) $A_1^nA_2^nA_4 \overset{*}{\Rightarrow} a^{n^2}A_2^nA_4$, (c) $A_2^nA_1^nA_4 \Rightarrow a^{2n+1}$.

We first prove (a). $S \Rightarrow A_3A_4 \Rightarrow A_1A_3A_2A_4 \overset{*}{\Rightarrow} A_1^{n-1}A_3A_2^{n-1}A_4 \Rightarrow A_1^{n-1}A_1A_2A_2^{n-1}A_4$ (We are applying $A_3 \to A_1A_3A_2(n-2)$ times and $A_3 \to A_1A_2$ once.) Hence (a). To prove (b) start with $A_1^nA_2^nA_4$. Then $A_1^{n-1}A_1A_2A_2^{n-1}A_4 \Rightarrow A_1^{n-1}aA_2A_1A_2^{n-1}A_4 \Rightarrow A_1^{n-2}aA_1A_2A_1A_2^{n-1}A_4 \Rightarrow A_1^{n-2}a^2A_2A_1A_1A_2^{n-1}A_4 \Rightarrow A_1^{n-2}a^2A_2A_1aA_2A_1A_2^{n-2}A_4 \overset{*}{\Rightarrow} A_1^{n-2}a^3A_2A_1A_2A_1A_2^{n-2}A_4 \Rightarrow A_1^{n-2}a^3A_2aA_2A_1A_1A_2^{n-2}A_4 \overset{*}{\Rightarrow} a^4A_1^{n-2}A_2^2A_1^2A_2^{n-2}A_4$.

Proceeding in a similar way, we get $A_1^nA_2^nA_4 \overset{*}{\Rightarrow} a^{n^2}A_2^nA_1^nA_4$. Hence (b).

Finally, $A_2^nA_1^nA_4 \Rightarrow A_2^nA_1^{n-1}A_4a \overset{*}{\Rightarrow} A_2^nA_4a^n \Rightarrow A_2^{n-1}A_5a^{n+1} \overset{*}{\Rightarrow} A_5a^{2n} \Rightarrow a^{2n+1}$. (We apply $A_1A_4 \to A_4a$, $A_2A_4 \to A_5a$, $A_2A_5 \to A_5a$ and finally $A_5 \to a$).

Using (a), (b) and (c), we get $S \Rightarrow a^{(n+1)^2}$.

4.8 The productions for (i) are $S \to aS \mid B$, $aS \to aa$, $B \to a$. For (ii) the productions are $S \to AS \mid a$, $A \to a$. For (iii) the productions are $S \to aS \mid a$.

4.9 The required grammar $G = (\{S, S_1, A, B\}, \Sigma, P, S)$, where $\Sigma = \{0, 1, 2, \ldots, 9\}$ and P consists of

$S \to 0 \mid 2 \mid 4 \mid 6 \mid 8$, $S \to AS_1$, $A \to 1 \mid 2 \mid \ldots \mid 9$

$S_1 \to 0 \mid 2 \mid 4 \mid 6 \mid 8$, $S \to ABS_1$, $B \to 1 \mid 2 \mid \ldots \mid 9$

$S \to 0 \mid 2 \mid 4 \mid 6 \mid 8$ generate even integers with one digit.

$S \to AS_1$ and A-productions and S_1-productions generate all even numbers with two digits. The remaining productions can be used to generate all even integers with three digits.

4.10 (a) $G = (\{S, A, B\}, \{0, 1\}, P, S)$, where P consists of $S \to 0S1 \mid 0A \mid 1B \mid 0 \mid 1$, $A \to 0A \mid 0$, $B \to 1B \mid 1$. Using $S \to 0S1$, $S \to 0$, $S \to 1$, we can get 0^m1^n, where m and differ by 1. To get more 0's (than 1's) in the string we have to apply $A \to 0A \mid 0$, $S \to 0A$ repeatedly. To get more 1's, apply $S \to 1B$, $B \to 1B \mid 1$ repeatedly.

(b) The required productions are $S \to aS_1$, $S_1 \to bS_1c$, $S_1 \to bc$, $S \to aS_2c$, $S_2 \to aS_2c$, $S_2 \to b$, $S \to S_3c$, $S_3 \to aS_3b$, $S_3 \to ab$. The first three productions derive ab^nc^n. $S \to aS_2c$ and the S_2-productions generate a^nbc^n. The remaining productions generate a^nb^nc.

(c) The required productions are $S \to 0S1$, $S \to 01$, $S \to 0A1$, $A \to 1A$, $A \to 1$.

(d) The required productions are $S \to aSc$, $S \to ac$, $S \to bc$, $S \to bS_1c$, $S_1 \to bS_1c$, $S_1 \to bc$. A typical string in the given language can be written in the form $a^1b^mc^mc^1$ where 1, $m \geq 0$. $S \to aSc$, $S \to bc$

generate a^1c^1 for $1 \geq 1$. $S \to bS_1c$, $S_1 \to bS_1c$, $S_1 \to bc$ generate b^mc^m for $m \geq 1$. For getting $a^1b^mc^mc^1$, we have to apply $S \to aSc$ l times; $S \to bc$, $S \to bS_1c$, $S_1 \to bS_1c$ and $S_1 \to bc$ are to be applied. For $m = 1$, $S \to bc$ has to be applied. For $m > 1$, we have to apply $S \to bS_1c$, $S_1 \to bS_1c$ and $S_1 \to bc$ repeatedly. The terminal c is added whenever the terminal a or b is added in the course of the derivation. This takes care of the condition $1 + m = n$.

(e) Let G be a context-free grammar whose productions are $S \to S0S1S0S$, $S \to S0S0S1S$, $S \to S1S0S0S$, $S \to A$. It is easy to see that elements in $L(G)$ are in L. Let $w \in L$. We prove that $w \in L(G)$ by induction on $|w|$. Note that every string in L is of length $3n$, $n \geq 1$. When $|w| = 3$, w has to be one of 010, 001 or 100. These strings can be derived by applying $S \to S0S1S0S$, $S \to S0S0S1S$ and $S \to S1S0S0S$ and then $S \to \Lambda$. Thus there is basis for induction. Assume the result for all strings of length $3n - 3$. Let $w \in L$ and let $|w| = 3n$, w should contain one of 010, 001 or 100 as a substring. Call the substring w_1. Write $w = w_2w_1w_3$. Then $|w_2w_3| = 3n - 3$ and by induction hypothesis $S \overset{*}{\Rightarrow} w_2w_3$. Note that all the productions (except $S \to \Lambda$) yield a sentential form starting and ending with S and having the symbol S between every pair of terminals. Without loss of generality, we can assume that the last step in the derivation $S \overset{*}{\Rightarrow} w_2w_3$ is of the form $w_2Sw_3 \Rightarrow w_2w_3$. So, $S \Rightarrow w_2Sw_3$. But $w_1 \in L$ and so $S \overset{*}{\Rightarrow} w_1$. Thus, $S \overset{*}{\Rightarrow} w_2w_1w_3$. In other words, $w \in L(G)$. By the principle of induction, $L = L(G)$.

4.11 The required productions are:
(a) $S \to aS_1$, $S_1 \to aS$, $S \to aS_2$, $S_2 \to a$
(b) $S \to aS$, $S \to bS$, $S \to a$
(c) $S \to aS_1$, $S_1 \to aS_1$, $S_1 \to bS_1$, $S_1 \to a$, $S_1 \to b$
(d) $S \to aS_1$, $S_1 \to aS_1$, $S_1 \to bS_2$, $S_2 \to bS_2$, $S_3 \to cS_3$, $S_3 \to c$
(e) $S \to aS_1$, $S_1 \to bS$, $S \to aS_2$, $S_2 \to b$.

4.12 $\overset{*}{\Rightarrow}$ is not symmetric and so the relation is not an equivalence relation (Refer to Note (ii), page 109)

4.13 It is clear that $L(G_1) = \{a^nb^n \mid n \geq 1\}$. In G_2, $S \Rightarrow AC \Rightarrow ASB \overset{*}{\Rightarrow} A^{n-1}SB^{n-1}$. Also, $S \Rightarrow AB \overset{*}{\Rightarrow} ab$. Hence $S \overset{*}{\Rightarrow} a^nb^n$ for all $n \geq 1$. This means $L(G_2) = \{a^nb^n \mid n \geq 1\} = L(G_1)$.

4.14 $L(G) = \emptyset$, for we get a variable on the application of each production and so no terminal string results.

4.15 The required productions are $S \to aS_1$, $S_1 \to bS_2$, $S_2 \to c$, $S \to bS_3$, $S_3 \to cS_4$, $S_4 \to a$, $S \to cS_5$, $S_5 \to aS_6$, $S_6 \to b$.

4.16 The required productions are $S \to S_1$, $S_1 \to abS_1$, $S_1 \to ab$, $S \to S_2$, $S_2 \to baS_2$, $S_2 \to ba$.

4.17 Let the given grammar be G_1. The production $A \to xB$ where $x = a_1a_2 \ldots a_n$ is replaced by $A \to a_1A_1$, $A_1 \to a_2A_2$, ..., $A_{n-1} \to a_nB$. The production $A \to y$, where $y = b_1b_2 \ldots b_m$ is replaced by $A \to b_1B_1$, $B_1 \to b_2B_2$, ..., $B_{m-1} \to b_m$. The grammar G_2 whose productions are the new productions is regular and equivalent to G_1.

Chapter 5

5.1 (a) **0 + 1 + 2**

(b) **1(11)***

(c) w in the given set has only one a which can occur anywhere in w. So $w = xay$, where x and y consist of some b's (or none). Hence the given set is represented by **b* ab***.

(d) Here we have three cases: w contains no a, one a or two a's. Arguing as in (c), the required regular expression is

b* + b* ab* + b* ab* ab*

(e) **aa(aaa)***

(f) **(aa)* + (aaa)* + aaaaa**

(g) **a(a + b)* a**

5.2 (a) The strings of length at most 4 in **(ab + a)*** are Λ, a, ab, aa, aab, aba, $abab$, a^3, $aaba$, $aaab$ and a^4. The strings in **aa + b** are aa and b. Concatenating (i) strings of length at most 3 from the first set and aa and (ii) strings of length 4 and b, we get the required strings. They are aa, aaa, $abaa$, $aaaa$, $aabaa$, $abaaa$, a^5, $ababb$, $aabab$, $aaabb$, and $aaaab$.

(c) The strings in **(ab + a) + (ab + a)**^{2} are a, ab, aa, $abab$, aab and aba. The strings of length 5 or less in **(ab + a)**^{3} are a^3, $abaa$, $aaab$, $ababa$. The strings of length 5 or less in **(ab + a)**^{4} are a^4, a^3ab, aba^3. In **(ab + a)**^{5}, a^5 is the only string of length 5 or less. The strings in **a*** are in **(ab + a)*** as well. Hence the required strings are Λ, a, ab, a^2, $abab$, aab, aba, a^3, $abaa$, $aaab$, $ababa$, a^4, $aaaab$, $abaaa$, and a^5.

5.3 (a) The set of all strings starting with a and ending in ab.

(b) The strings are either strings of a's followed by one b or strings of b's followed by one a.

(c) The set of all strings of the form vw where a's occur in pairs in v and b's occur in pairs in w.

5.5 The transition system equivalent to **(ab + a)*(aa + b)** (5.2(a)) is given in Fig. A5.1.

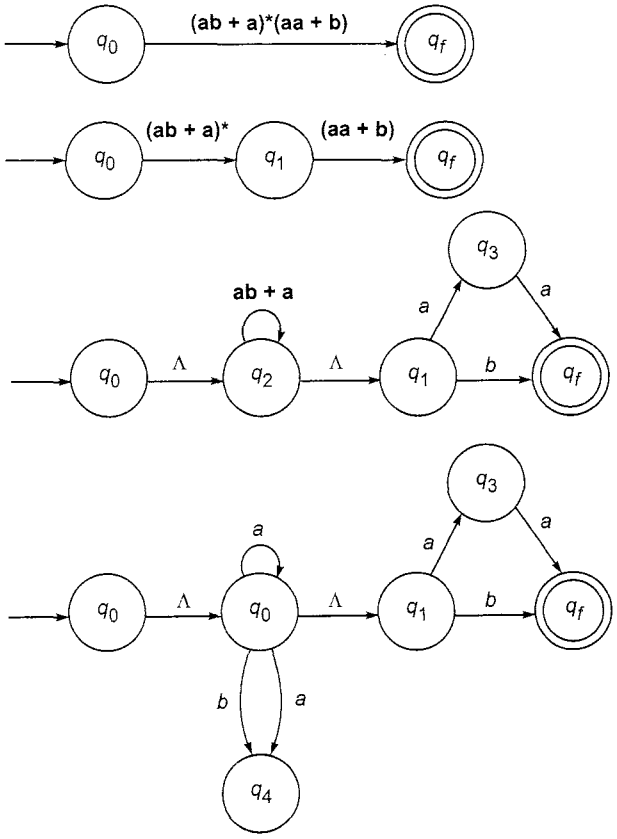


Fig. A5.1 Transition system for Exercise 5.5.

5.6 The transition system equivalent to **a(a + b)*ab** (5.3)(a)) is given in Fig. A5.2.

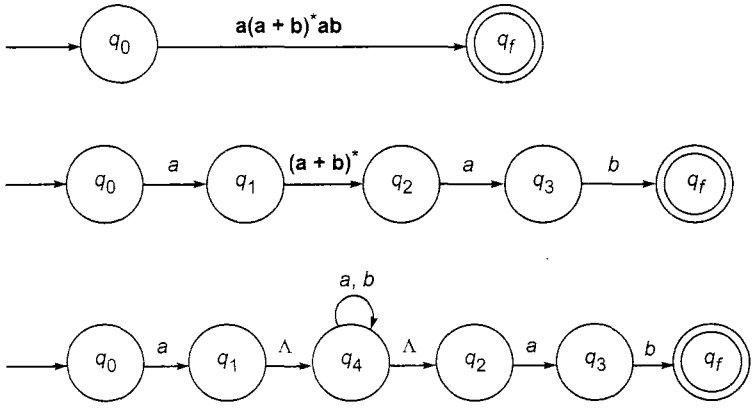


Fig. A5.2 Transition ,system for Exercise 5.6.

5.7 (a) $\emptyset$; (b) $(a + b)^*$; (c) the set of all strings over $\{a, b\}$ containing two successive a's or two successive b's; (d) the set of all strings containing even number of a's and even number of b's.

5.8 We get the following equations:

$$q_0 = \Lambda$$

$$q_1 = q_0\mathbf{1} + q_1\mathbf{1} + q_3\mathbf{1}$$

$$q_2 = q_1\mathbf{0} + q_2\mathbf{0} + q_3\mathbf{0}$$

$$q_3 = q_2\mathbf{1}$$

Therefore,

$$q_2 = q_1\mathbf{0} + q_2\mathbf{0} + q_2\mathbf{10} = q_1\mathbf{0} + q_2(\mathbf{0} + \mathbf{10})$$

Applying Theorem 5.1, we get

$$q_2 = q_1\mathbf{0}(\mathbf{0} + \mathbf{10})^*$$

Now,

$$q_1 = \mathbf{1} + q_1\mathbf{1} + q_2\mathbf{11} = \mathbf{1} + q_1\mathbf{1} + q_1\mathbf{0}(\mathbf{0} + \mathbf{10})^*\mathbf{11}$$

$$= \mathbf{1} + q_1(\mathbf{1} + \mathbf{0}(\mathbf{0} + \mathbf{10})^*\ \mathbf{11})$$

By Theorem 5.1, we get

$$q_1 = \mathbf{1}(\mathbf{1} + \mathbf{0}(\mathbf{0} + \mathbf{10})^*\mathbf{11})^*$$

$$q_3 = q_2\mathbf{1} = \mathbf{1}(\mathbf{1} + \mathbf{0}(\mathbf{0} + \mathbf{10})\mathbf{11})^*\ \mathbf{0}(\mathbf{0} + \mathbf{10})^*\mathbf{1}$$

As q_3 is the only final state, the regular expression corresponding to the given diagram is $\mathbf{1}(\mathbf{1} + \mathbf{0}(\mathbf{0} + \mathbf{10})^*\ \mathbf{11})^*\ \mathbf{0}\ (\mathbf{0} + \mathbf{10})^*\ \mathbf{1}$.

5.9 The transition system corresponding to $(\mathbf{ab} + \mathbf{c}^*)^*\mathbf{b}$ is given in Fig. A5.3.

5.10 (a) The required regular expression
$\mathbf{(1 + 01)}^* + \mathbf{(1 + 01)}^*\ \mathbf{00(1 + 01)}^* + \mathbf{(0 + 10)}^* + \mathbf{(0 + 10)}^*\ \mathbf{11(0 + 10)}^*\ \mathbf{(1 + 01)}^*$ represents the set of all strings containing no pair of 0's. $\mathbf{(1 + 01)}^*\ \mathbf{00}\ \mathbf{(1 + 01)}^*$ represents the set of all strings containing exactly one pair of 0's. The remaining two expressions correspond to a pair of 1's.

(c) Let w be in the given set L. If w has n a's then it is in a set represented by $(\mathbf{b})^*$. If w has only one a then it is in a set represented by $\mathbf{b}^*\mathbf{ab}^*$. If w has more than one a, write $w = w_1aw_2$, where w does not contain any a. Then w_1 is in a set represented by $(\mathbf{b} + \mathbf{abb})^*$. So the given set is represented by the regular expression $\mathbf{b}^* + (\mathbf{b} + \mathbf{abb})^*\ \mathbf{ab}^*$. (Note that the regular set corresponding to $\mathbf{b}^*$ is a subset of the set corresponding to $(\mathbf{b} + \mathbf{abb})^*$

(d) $\mathbf{(0 + 1)}^*\ \mathbf{000}\ \mathbf{(0 + 1)}^*$

(e) $\mathbf{00(0 + 1)}^*$

(f) $\mathbf{1(0 + 1)}^*\ \mathbf{00}$.

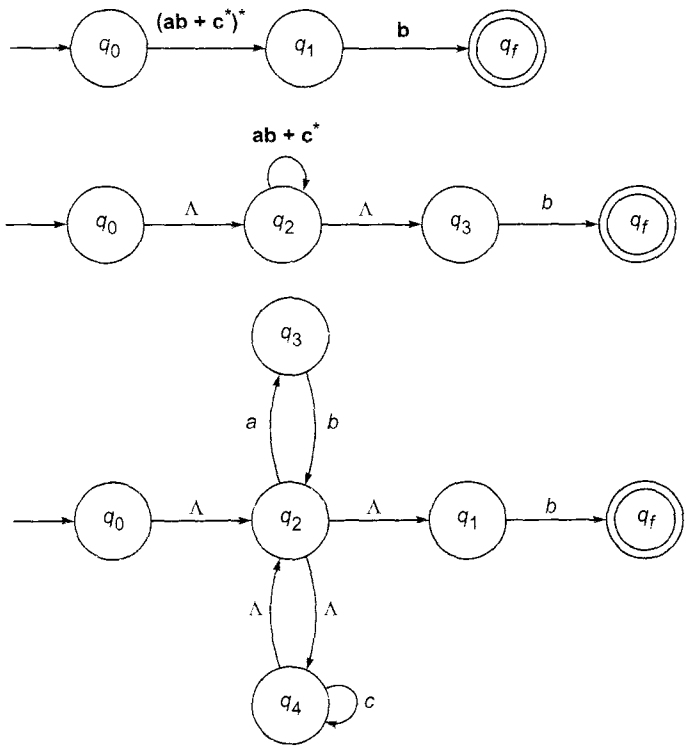


Fig. A5.3 Transition system for Exercise 5.9.

5.12 The corresponding regular expression is **(0 + 1)* (010 + 0010)**. We can construct the transition system with Λ-moves. Eliminating Λ-moves, we get the NDFA accepting the given set of strings. The NDFA is described by Fig. A5.4.

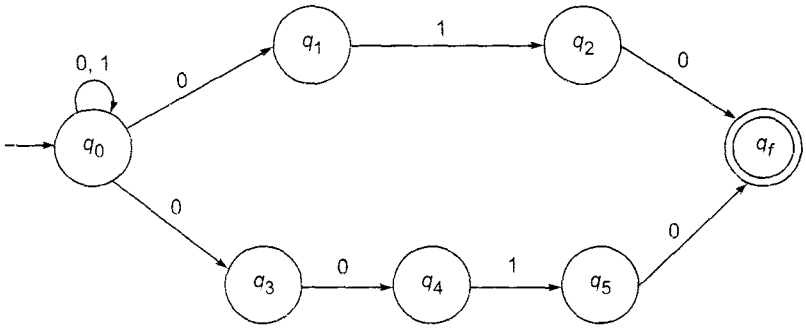


Fig. A5.4 NDFA for Exercise 5.12.

The equivalent DFA is defined by Table A5.1.

TABLE A5.1 State Table of DFA for Exercise 5.12

State/Σ	0	1
$[q_0]$	$[q_0, q_1, q_3]$	$[q_0]$
$[q_0, q_1, q_3]$	$[q_0, q_1, q_3, q_4]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1, q_3, q_f]$	$[q_0]$
$[q_0, q_1, q_3, q_4]$	$[q_0, q_1, q_3, q_4]$	$[q_0, q_2, q_5]$
$[q_0, q_1, q_3, q_f]$	$[q_0, q_1, q_3, q_4]$	$[q_0, q_2]$
$[q_0, q_2, q_5]$	$[q_0, q_1, q_3, q_f]$	$[q_0]$

5.13 Similar to Exercise 3.10.

5.14 The state table for the NDFA accepting **(a + b)* abb** is defined by Table A5.2.

TABLE A5.2 State Table for Exercise 5.14

State/Σ	a	b
q_0	q_0, q_1	q_0
q_1		q_2
q_2		q_f
q_f		

The corresponding DFA is defined by Table A5.3.

TABLE A5.3 State Table of DFA for Exercise 5.14

State/Σ	a	b
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_f]$
$[q_0, q_f]$	$[q_0, q_1]$	$[q_0, q_f]$

5.15 Let L be the set of all palindromes over $\{a, b\}$. Suppose it is accepted by a finite automaton $M = (Q, \Sigma, \delta, q, F)$. $\{\delta(q_0, a^n) \mid n \geq 1\}$ is a subset of Q and hence finite. So $\delta(q_0, a^n) = \delta(q_0, a^m)$ for some m and n, $m < n$. As $a^n b^{2n} a^n \in L$, $\delta(q_0, a^n b^{2n} a^n) \in F$. But $\delta(q_0, a^m) = \delta(q_0, a^n)$. Hence $\delta(q_0, a^m b^{2n} a^n) = \delta(q_0, a^n b^{2n} a^n)$, which means $a^m b^{2n} a^n \in L$. This is a contradiction since $a^m b^{2n} a^n$ is not a palindrome (remember $m < n$). Hence L is not accepted by a finite automaton.

5.16 The proof is by contradiction. Let $L = \{a^n b^n \mid n > 0\}$. $\{\delta(q_0, a^n) \mid n \geq 0\}$ is a subset of Q and hence finite. So $\delta(q_0, a^n) = \delta(q_0, a^m)$ for some m and n, $m \neq n$. So $\delta(q_0, a^m b^n) = \delta(\delta(q_0, a^m), b^n) = \delta(\delta(q_0, a^n), b^n) = \delta(q_0, a^n b^n)$. As $a^n b^n \in L$, $\delta(q_0, a^n b^n)$ is a final state and so is $\delta(q_0, a^m b^n)$. This means $a^m b^n \in L$ with $m \neq n$, which is a contradiction. Hence L is not regular

5.17 (a) Let $L = \{a^n b^{2n} \mid n > 0\}$. We prove L is not regular by contradiction. If L is regular, we can apply the pumping lemma. Let n be the number of states. Let $w = a^n b^{2n}$. By pumping lemma, $w = xyz$ with $|xy| \leq n$, $|y| > 0$ and $xz \in L$. As $|xy| \leq n$, $xy = a^m$ and $y = a^l$ where $0 < l \leq n$. So $xz = a^{n-l}b^{2n} \in L$, a contradiction since $n - l \neq n$. Thus L is not regular.

(b) Let $L = \{a^n b^m \mid 0 < n < m\}$. We show that L is not regular. Let n be the number of states and $w = a^n b^m$, where $m > n$. As in (a), $y = a^l$, where $0 < l \leq n$. By pumping lemma $xy^k z \in L$ for $k \geq 0$. So $a^{n-l}a^{lk}b^m \in L$ for all $k \geq 0$. For sufficiently large k, $n - l + lk > m$. This is a contradiction. Hence L is not regular.

5.19 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting a nonempty language. Then there exists $w = a_1 a_2 \ldots a_p$ accepted by M. If $p < n$, the result is true. Suppose $p > n$. Let $\delta(q_0, a_1 a_2 \ldots a_i) = q_i$ for $i = 1, 2, \ldots, p$. As $p > n$, the sequence of states $\{q_1, q_2, \ldots, q_p\}$ must have a pair of repeated states. Take the first pair (q_j, q_k) (Note $q_j = q_k$). Then $\delta(q_0, a_1 a_2 \ldots, a_j) = q_j$, $\delta(q_j, a_{j+1} \ldots a_k) = q_j$ and $\delta(q_j, a_{k+1} \ldots, a_p) \in F$. So $\delta(q_0, a_1 a_2 \ldots a_j a_{k+1} \ldots a_p) = \delta(q_j, a_1 a_2 \ldots, a_p) \in F$. Thus we have found a string in Σ^* whose length is less than p (and differs from $|w|$ by $k - j$). Repeating the process, we get a string of length m, where $m < n$.

5.20 Let $M = (\{q_0, q_1, q_f\}, \{a, b\}, \delta, q_0, \{q_f\})$, where q_0 and q_1 correspond to S and A respectively.

Then the NDFA accepting $L(G)$ is defined by Table A5.4.

Table A5.4 Table for Exercise 5.20

State/Σ	a	b
q_0	q_0	q_1, q_f
q_1	q_1, q_f	q_0

5.21 The transitions are:

$\delta(q_1, a) = q_4$, $\quad \delta(q_2, b) = q_1$

$\delta(q_1, b) = q_2$, $\quad \delta(q_4, a) = q_1$

$\delta(q_2, a) = q_3$

$\delta(q_3, a) = q_2$

$\delta(q_3, b) = q_4$

$\delta(q_4, b) = q_3$

Let A_1, A_2, A_3, A_4 correspond to q_1, q_2, q_3, q_4. The induced productions are $A_1 \to aA_4$, $A_1 \to bA_2$, $A_2 \to aA_3$, $A_3 \to aA_2$, $A_3 \to$

bA_4, $A_4 \rightarrow bA_3$ (corresponding to the first six transitions) and $A_2 \rightarrow bA_1$, $A_2 \rightarrow b$, $A_4 \rightarrow aA_1$, $A_4 \rightarrow a$ (corresponding to the last two transitions). So $G = (\{A_1, A_2, A_3, A_4\}, \{a, b\}, P, A_1)$, where P consists of the induced productions given above.

5.22 The required productions are:

$S \rightarrow AS_1|BS_1| \ldots |ZS_1$,

$S_1 \rightarrow AS_1|BS_1| \ldots |ZS_1$,

$S_1 \rightarrow 0S1|1S_1| \ldots |9S_1$

$S_1 \rightarrow A|B| \ldots |Z$ and $S_1 \rightarrow 0|1| \ldots | 9$

5.24 The given grammar is equivalent to $G = (\{S, A, B\}, \{a, b\}, P, S)$, where P consists of $S \rightarrow aS|bS|aA$, $A \rightarrow bB$, $B \rightarrow a$($B \rightarrow aC$ and $C \rightarrow \Lambda$ is replaced by $B \rightarrow a$). Let q_0, q_1 and q_2 correspond to S, A and B. q_f is the only final state. The transition system M accepting $L(G)$ is given as

$$M = (\{q_0, q_1, q_2, q_f\}, \{a, b\}, \delta, q_0, \{q_f\})$$

where q_0, q_1, and q_2 correspond to S, A and B and q_f is the (only) final state. $S \rightarrow aS$, $S \rightarrow bS$, $S \rightarrow aA$ and $A \rightarrow bB$ induce transitions from q_0 to q_0 with, labels b and a, from $q_0 \rightarrow q_1$ with label a and from q_1 to q_2 with label b. $B \rightarrow a$ induces a transition from q_2 to q_f with label a. M is defined by Fig. A5.5.

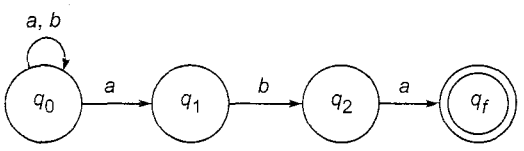


Fig A5.5 Transition system for Exercise 5.24.

The equivalent DFA is given in Table A5.5.

TABLE A5.5 State Table of DFA for Exercise 5.24

State/Σ	a	b
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1, q_f]$	$[q_0]$
$[q_0, q_1, q_f]$	$[q_0, q_1]$	$[q_0, q_2]$

Chapter 6

6.1. The derivation tree is given in Fig. A6.1.

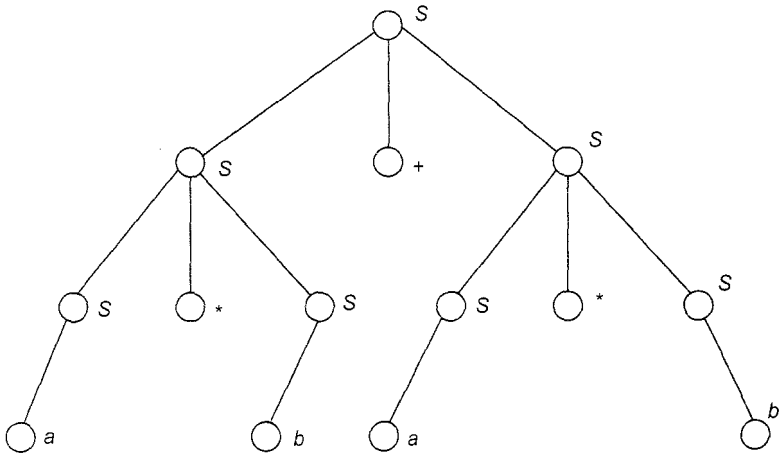


Fig. A6.1 Derivation tree for Exercise 6.1.

6.2. $S \to 0S0$, $S \to 1S1$ and $S \to A$ give the derivation $S \overset{*}{\Rightarrow} wAw$, where $w \in \{0, 1\}^+$. $A \to 2B3$ and $B \to 2B3$ give $A \overset{*}{\Rightarrow} 2^mB3^m$. Finally, $B \to 3$ gives $B \Rightarrow 3$. Hence $S \overset{*}{\Rightarrow} wAw \overset{*}{\Rightarrow} w2^mB3^mw \Rightarrow w2^m3^{m+1}w$. Thus, $L(G) \subseteq \{w2^m3^{m+1}w \mid w \in \{0, 1\}^* \text{ and } m \geq 1\}$. The reverse inclusion can be proved similarly.

6.3 (a) X_1, X_3, X_5; (b) X_2, X_4; (c) As $X_1 = S$, X_4X_2 and $X_2X_4X_4X_2$ are sentential forms.

6.4 (i) $S \Rightarrow SbS \Rightarrow abS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababSbS \Rightarrow abababS \Rightarrow abababa$.

(ii) $S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow SbSbSbS \Rightarrow SbSbSba \Rightarrow SbSbaba \Rightarrow Sbababa \Rightarrow abababa$

(iii) $S \Rightarrow SbS \Rightarrow abS \Rightarrow abSbS \Rightarrow abSbSbS \Rightarrow ababSbS \Rightarrow abababS \Rightarrow abababa$

6.5 (a) $S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaaBBB \Rightarrow aaabBB \Rightarrow aaabbB \Rightarrow aaabbaBB \Rightarrow aaabbabB \Rightarrow aaabbabbS \Rightarrow aaabbabbbA \Rightarrow aaabbabbba$

(b) $S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbbA \Rightarrow aaBbba \Rightarrow aaaBBbba \Rightarrow aaabBbba \Rightarrow aaabbSbba \Rightarrow aaabbaBbba \Rightarrow aaabbabbba$.

6.6 $abab$ has two different derivations $S \Rightarrow abSb \Rightarrow abab$ (using $S \to abSb$ and $S \to a$) $S \Rightarrow aAb \Rightarrow abSb \Rightarrow abab$ (using $S \to aAb$, $A \to bS$ and $S \to a$).

6.7 ab has two different derivations.

$S \Rightarrow ab$ (using $S \to ab$)
$S \Rightarrow aB \Rightarrow ab$ (using $S \to aB$ and $B \to b$).

6.8 Consider $G = (\{S, A, B\}, \{a, b\}, P, S)$, where P consists of $S \to AB \mid ab$ and $B \to b$.

Step 1 When we apply Theorem 6.4, we get
$W_1 = \{S\}$, $W_2 = \{S\} \cup \{A, B, a, b\} = W_3$
Hence $G_1 = G$.

Step 2 When we apply Theorem 6.3, we obtain
$W_1 = \{S, B\}$, $W_2 = \{S, B\} \cup \emptyset = W_3$
So, $G_2 = (\{S, B\}, \{a, b\}, \{S \to ab, B \to b\}, S)$.
Obviously, G_2 is not a reduced grammar since $B \to b$ does not appear in the course of derivation of any terminal string.

6.9 **Step 1** Applying Theorem 6.3, we have
$W_1 = \{B\}$, $W_2 = \{B\} \cup \{C, A\}$, $W_3 = \{A, B, C\} \cup \{S\} = V_N$
Hence $G_1 = G$.
Step 2 Applying Theorem 6.4, we obtain
$W_1 = \{S\}$, $W_2 = \{S\} \cup \{A, a\}$, $W_3 = \{S, A, a\} \cup \{B, b\}$
$W_4 = \{S, A, B, a, b\} \cup \emptyset$
Hence, $G_2 = (\{S, A, B\}, \{a, b\}, P, S)$, where P consists of $S \to aAa$, $A \to bBB$ and $B \to ab$.

6.10 The given grammar has no null productions. So we have to eliminate unit productions. This has already been done in Example 6.10. The resulting equivalent grammar is $G = (\{S, A, B, C, D, E\}, \{a, b\}, P, S)$, where P consists of $S \to AB$, $A \to a$, $B \to b \mid a$, $C \to a$, $D \to a$ and $E \to a$. Apply step 1 of Theorem 6.5. As every variable derives some terminal string, the resulting grammar is G itself.

Now apply step 2 of Theorem 6.5. Then
$W_1 = \{S\}$, $W_2 = \{S\} \cup \{A, B\} = \{S, A, B\}$, $W_3 = \{S, A, B\} \cup \{a, b\} = \{S, A, B, a, b\}$ and $W_4 = W_3$.
Hence the reduced grammar is $G' = (\{S, A, B\}, \{a, b\}, P', S)$, where $P' = \{S \to AB, A \to a, B \to b, B \to a\}$.

6.11 We prove that by eliminating redundant symbols (using Theorem 6.3 and Theorem 6.4) and then Unit productions, we may not get an equivalent grammar in the most simplified form. Consider the grammar G whose productions are $S \to AB$, $A \to a$, $B \to C$, $B \to b$, $C \to D$, $D \to E$ and $E \to a$.

Step 1 Using Theorem 6.3, we get
$W_1 = \{A, B, E\}$, $W_2 = \{A, B, E\} \cup \{S, D\}$,
$W_3 = \{S, A, B, D, E\} \cup \{C\} = V_N$.
Hence $G_1 = G$.

Step 2 Using Theorem 6.4, we obtain
$W_1 = \{S\}$, $W_2 = \{S\} \cup \{A, B\}$, $W_3 = \{S, A, B\} \cup \{a, c, b\}$,
$W_4 = \{S, A, B, c, a, b\} \cup \{D\}$,
$W_5 = \{S, A, B, C, D, a, b\} \cup \{E\} = V_N \cup \Sigma$.
Hence $G_2 = G_1 = G$.

Step 3 We eliminate unit productions. We then have
$W(S) = \{S\}$ $W(A) = \{A\}$, $W(E) = \{E\}$ $W_0(B) = \{B\}$,
$W_1(B) = \{B\} \cup \{C\}$, $W_2\{B) = \{B, C\} \cup \{D\}$,
$W_3(B) = \{B, C, D, E\} = W(B)$, $W(C) = \{C, D, E\}$, $W(D) = \{D, E\}$.
The productions in the new grammar G_3 are $S \to AB$, $A \to a$, $B \to b$, $B \to a$ and $E \to a$. G_3 contains the redundant symbol E. So G_3 is not the equivalent grammar in the most simplified form.

6.12 (a) As there are no null productions or unit productions, we can proceed to step 2.

Step 2 Let $G_1 = (V'_N, \{0, 1\}, P_1, S)$, where P_1 and V'_N are constructed as follows:

(i) $A \to 0$, $B \to 1$ are included in P_1.
(ii) $S \to 1A$, $B \to 1S$ give rise to $S \to C_1A$, $B \to C_1S$ and $C_1 \to 1$.
(iii) $S \to 0B$, $A \to 0S$ give rise to $S \to C_0B$, $A \to C_0S$ and $C_0 \to 0$.
(iv) $A \to 1AA$, $B \to 0BB$ give rise to $A \to C_1AA$ and $B \to C_0BB$.
$V'_N = \{S, A, B, C_0, C_1\}$.

Step 3 $G_2 = (V''_N, \{0, 1\}, P_2, S)$, where P_2 and V''_N, are constructed as follows:

(i) $A \to 0$, $B \to 1$, $S \to C_1A$, $B \to C_1S$, $C_1 \to 1$, $S \to C_0B$, $A \to C_0S$, $C_0 \to 0$ are included in P_2.
(ii) $A \to C_1AA$ and $B \to C_0BB$ are replaced by $A \to C_1D_1$, $D_1 \to AA$, $B \to C_0D_2$, $D_2 \to BB$.

Thus, $G_2 = (\{S, A, B, C_0, C_1, D_1, D_2\}, \{0, 1\}, P_2, S)$ is in CNF and equivalent to the given grammar where P_2 consists of $S \to C_1A|C_0B$, $A \to 0|C_0S|C_1D_1$, $B \to 1|C_1S|C_0D_2$, $C_1 \to 1$, $C_0 \to 0$, $D_1 \to AA$ and $D_2 \to BB$.

(b) **Step 2** $G_1 = (V'_N, \{a, b, c\}, P_1, S)$, where P_1 and V'_N are defined as follows:

(i) $S \to a$, $S \to b$ are included in P_1
(ii) $S \to cSS$ is replaced by $S \to CSS$, $C \to c$, $V'_N = \{S, C\}$

Step 3 $G_2 = (V''_N, \{a, b, c\}, P_2, S)$, where P_2 is defined as follows:

(i) $S \to a$, $S \to b$, $C \to c$ are included in P_2.
(ii) $S \to CSS$ is replaced by $S \to CD$ and $D \to SS$.

Thus, the equivalent grammar in CNF is $G_2 = (\{S, C, D\}, \{a, b, c\}, P_2, S)$, where P_2 consists of $S \to a|b|CD$, $C \to c$, $D \to SS$.

6.13 Consider $G = (\{S\}, \{a, b, +, *\}, P, S)$, where P consists of $S \to S + S$, $S \to S * S$, $S \to a$, $S \to b$.

Step 2 $G_1 = (V'_N, \{a, b, +, *\}, P_1, S)$, where P_1 is constructed as follows:

(i) $S \to a$, $S \to b$ are included in P_1,
(ii) $S \to S + S$ and $S \to S * S$ are replaced by $S \to SAS$, $S \to SBS$, $A \to +$, $B \to *$
$V'_N = \{S, A, B\}$

Step 3 $G_2 = (V_N'', \{a, b, +, *\}, P_2, S)$, where P_2 is constructed as follows:

(i) $S \to a$, $S \to b$, $A \to +$ and $B \to *$ are included in P_2.
(ii) $S \to SAS$ and $S \to SBS$ give rise to $S \to SA_1$, $A_1 \to AS$, $S \to SB_1$, $B_1 \to BS$.

The required grammar in CNF is

$$G_2 = (\{S, A, B, A_1, B_1\}, \{a, b, +, *\}, P_2, S)$$

where P_2 consists of

$$S \to a|b|SA_1|SB_1,\ A \to +,\ B \to *,\ A_1 \to AS \text{ and } B_1 \to BS$$

6.14 (a) Rename S as A_1, By Remark following Theorem 6.9, it is enough to replace terminals by new variables to get an equivalent grammar G_1. Now, G_1 is defined as

$$G_1 = (\{A_1, A_2, A_3\}, \{0, 1\}, P_1, A_1)$$

where P_1 consists of

$$A_1 \to A_1A_1|A_2A_1A_3|A_2A_3,\ A_2 \to 0 \text{ and } A_3 \to 1$$

This completes step 1.

Step 2 All productions of G_1 except $A_1 \to A_1A_1$ are in proper form. Applying Lemma 6.2 to $A_1 \to A_1A_1$, we get a new variable Z_1 and new productions $A_1 \to A_2A_1A_3Z_1 \mid A_2A_3Z_1$, $Z_1 \to A_1$, $Z_1 \to A_1Z_1$. The new grammar is

$$G_2 = (\{A_1, A_2, A_3, Z_1\}, \{a, b\}, P_2, A_1)$$

where P_2 consists of

$$A_1 \to A_2A_1A_3 \mid A_2A_3 \mid A_2A_1A_3Z_1 \mid A_2A_3Z_1$$
$$Z_1 \to A_1Z_1,\ Z_1 \to A_1,\ A_2 \to 0 \text{ and } A_3 \to 1$$

Step 3 As A_3-productions and A_2-productions are in proper form we have to modify only the A_1-productions using Lemma 6.1. So the modified A_1-productions are

$$A_1 \to 0A_1A_3 \mid 0A_3 \mid 0A_1A_3Z_1 \mid 0A_3Z_1$$

Step 4 The productions $Z_1 \to A_1$ and $Z_1 \to A_1Z_1$ are modified using Lemma 6.1. They are:

$$Z_1 \to 0A_1A_3 \mid 0A_3 \mid 0A_1A_3Z_1 \mid 0A_3Z_1$$
$$Z_1 \to 0A_1A_3Z_1 \mid 0A_3Z_1 \mid 0A_1A_3Z_1Z_1 \mid 0A_3Z_1Z_1$$

Thus the required equivalent grammar in GNF is

$G_3 = (\{A_1, A_2, A_3, Z_1\}, \{0, 1\}, P_3, A_1)$, where P_1 consists of

$$A_1 \to 0A_1A_3 \mid 0A_3 \mid 0A_1A_3Z_1 \mid 0A_3Z_1$$
$$A_2 \to 0,\ A_3 \to 1$$
$$Z_1 \to 0A_1A_3 \mid 0A_3 \mid 0A_1A_3Z_1 \mid 0A_3Z_1$$
$$Z_1 \to 0A_1A_3Z_1 \mid 0A_3Z_1 \mid 0A_1A_3Z_1Z_1 \mid 0A_3Z_1Z_1$$

(b) **Step 1** Replace $B \to aSb$ by $B \to aSC$ and $C \to b$. Rename S, A, B and C by A_1, A_2, A_3 and A_4. The resulting grammar is

$$G_1 = (\{A_1, A_2, A_3, A_4\}, \{a, b\}, P_1, A_1)$$

where P_1 consists of $A_1 \to A_2A_3$, $A_2 \to A_3A_3$

$A_2 \to A_3A_1A_3$, $A_2 \to b$, $A_3 \to aA_2A_4$, $A_3 \to a$ and $A_4 \to b$.

Steps 2, 3 and 4 Step 2 construction is not necessary for G_1. The only A_4-production, $A_4 \to b$, is in proper form. So we go to step 4. The modified A_2-productions are:

$$A_2 \to aA_2A_4A_3 \mid aA_3 \mid aA_2A_4A_1A_3 \mid aA_1A_3 \mid b.$$

The modified A_1-productions are:

$$A_1 \to aA_2A_4A_3A_3 \mid aA_3A_3 \mid aA_2A_4A_1A_3A_3 \mid aA_1A_3A_3 \mid bA_3.$$

Step 5 is not necessary since there is no new variable in the form in Z_i. So an equivalent grammar in GNF is

$$G_2 = (\{A_1, A_2, A_3, A_4\}, \{a, b\}, P_2, A_1)$$

where P_2 consists of

$$A_1 \to aA_2A_4A_3A_3 \mid aA_3A_3 \mid aA_2A_4A_1A_3A_3 \mid aA_1A_3A_3 \mid bA_3$$
$$A_2 \to aA_2A_4A_3 \mid aA_3 \mid aA_2A_4A_1A_3 \mid aA_1A_3 \mid b$$
$$A_3 \to aA_2A_4 \mid a, A_4 \to b.$$

6.15 The grammar given in Exercise 6.7 has the following productions: $S \to aB$, $S \to ab$, $A \to aAB$, $A \to a$, $B \to ABb$ and $B \to b$. Of these, $S \to aB$, $A \to aAB$, $A \to a$ and $B \to b$ are in the required form. So we replace the terminals which appear in the second and subsequent places of the R.H.S. of $S \to ab$ and $B \to ABb$ by a new variable C and add a production $C \to b$. Renaming S, B, A and C as A_1, A_2, A_3 and A_4, the modified productions turn out to be $A_1 \to aA_2$, $A_1 \to aA_4$, $A_3 \to aA_3A_2$, $A_3 \to a$, $A_2 \to A_3A_2A_4$, $A_2 \to b$ and $A_4 \to b$.
This completes the first three steps:

Step 4 $A_2 \to A_3A_2A_4$ is replaced by $A_2 \to aA_3A_2A_2A_4 \mid aA_2A_4$. The other productions are in the proper form. The resulting grammar in GNF has the productions

$$A_1 \to aA_2 \mid aA_4, \quad A_2 \to aA_3A_2A_2A_4 \mid aA_2A_4 \mid b, \quad A_3 \to aA_3A_2 \mid a, A_4 \to b.$$

The grammar in Exercise 6.10 has unit productions. Eliminating unit productions, we get an equivalent grammar G, where

$$G = (\{S, A, B, C, D, E\}, \{a, b\}, P, S)$$

where P consists of $S \to AB$, $A \to a$, $B \to a \mid b$, $C \to a$, $D \to a$ and $E \to a$. Rename S, A, B, C, D and E as A_1, A_2, A_3, A_4, A_5 and A_6.

Thus P consists of $A_1 \to A_2A_3$, $A_2 \to a$, $A_3 \to a \mid b$, $A_4 \to a$, $A_5 \to a$ and $A_6 \to a$.

We have to modify only $A_1 \to A_2A_3$ using Lemma 6.1.

Thus an equivalent grammar in GNF has the following productions:

$A_1 \to aA_3$, $A_2 \to a$, $A_3 \to a \mid b$, $A_4 \to a$, $A_5 \to a$ and $A_6 \to a$.

6.16. (a) The given language is generated by a grammar whose productions are $S \to aSa \mid bSb \mid c$.

Step 2 (i) $S \to c$ is in P_1

(ii) $S \to aSa$ and $S \to bSb$ give rise to

$$S \to ASA,\ S \to BSB,\ A \to a,\ B \to b \text{ in } P_1$$

Thus $G_1 = (\{S, A, B\}, \{a, b, c\}, P_1, S)$, where P_1 consists of $S \to ASA \mid BSB \mid c$, $A \to a$ and $B \to b$.

Step 3 The equivalent grammar G_2 in CNF is defined by

$G_2 = (\{S, A, B, A_1, B_1\}, \{a, b, c\}, P_2, S)$, where P_2 consists of

$$S \to AA_1,\ A_1 \to SA,\ S \to BB_1,\ B_1 \to SB,\ A \to a,\ B \to b,\ S \to c.$$

(b) The grammar generating the given set is having the productions $S \to bA \mid aB$, $A \to bAA \mid aS \mid a$, $B \to aBB \mid bS \mid b$.

Step 2 The productions obtained in this step are:

$S \to B_1A$, $B_1 \to b$, $S \to A_1B$, $A_1 \to a$, $A \to B_1AA$, $A \to A_1S$, $A \to a$, $B \to A_1BB$, $B \to B_1S$, $B \to b$.

Step 3 The equivalent grammar in CNF is given by

$G_2 = (V''_N, \{a, b\}, P_2, S)$, where P_2 consists of

(i) $S \to B_1A$, $B_1 \to b$, $S \to A_1B$, $A_1 \to a$, $A \to A_1S$, $A \to a$, $B \to B_1S$, $B \to b$

(ii) $A \to B_1C_1$, $C_1 \to AA$, $B \to A_1C_2$, $C_2 \to BB$ (corresponding to $A \to B_1AA$ and $B \to A_1BB$).

6.17 (a) The grammar generating the given language has the productions $S \to aSa$, $S \to bSb$, $S \to c$. The first two productions will be in GNF if the last symbols on R.H.S. are variables. Hence $S \to aSa$, $S \to bSb$ can be replaced by $S \to aSA$, $A \to a$, $S \to bSB$, $B \to b$. Hence $G' = (\{S, A, B\}, \{a, b\}, P', S)$, where P' consists of $S \to aSA \mid bSB$, $S \to c$, $A \to a$, $B \to b$ is in GNF and is generating the given language.

(b) The given language is generated by

$G = (\{S, A, B\}, \{a, b\}, P, S)$, where P consists of

$S \to bA \mid aB$, $A \to bAA \mid aS \mid a$, $B \to aBB \mid bS \mid b$. This itself is in GNF.

(c) The given language is generated by a grammar whose productions are $S \to aAb$, $S \to aA$, $A \to aA$, $A \to a$, $S \to a$, $S \to bB$, $B \to b$ and $S \to b$. Of these productions we have to modify only one production namely, $S \to aSb$. This is done by replacing this

production by $S \to aSB_1$, $B_1 \to b$. (***Note:*** In this problem we can also replace $S \to aSb$ by $S \to aSB$ alone. $B \to b$ is already in the grammar and there are no other B-productions.)

(d) The given language is generated by a grammar whose productions are $S \to aSb$, $S \to cS$, $S \to c$. The equivalent grammar in GNF is

$$G = (\{S, B\}, \{a, b, c\}, P, S),$$

where P consists of $S \to aSB$, $B \to b$, $S \to cS$, $S \to c$.

6.18 (i) Let $w \in L(G)$ and $|w| = k$. In the Chomsky normal form, each production yields one terminal or two variables but nothing else. For getting the terminals in w, we have to apply production of the form $A \to a$ (k times). The corresponding string of variables, which is of length k, can be obtained by $k - 1$ steps. (Each production $A \to BC$ increases the number of variables by one.) So the total number of steps is $2k - 1$. (The reader is advised to prove this result by induction on $|w|$.)

(ii) When G is in GNF, the number of steps in the derivation of w is $k (k = |w|)$. The number of terminals increases by 1 for each application of a production to a sentential form. Hence the number of steps in the derivation of w is k.

6.19 **Step 1** Let n be the natural number obtained by applying pumping lemma.

Step 2 Let $z = a^{n^2}$. Write $z = uvwxy$ where $1 \leq |vx| \leq n$. (This is possible since $|vwx| \leq n$ by (ii) of pumping lemma.) Let $|vx| = m$, $m \leq n$. By pumping lemma, uv^2wx^2y is in L. As $|uv^2wx^2y| > n^2$, $|uv^2wx^2y| = k^2$, where $k \geq n + 1$. But $|uv^2wx^2y| = n^2 + m < n^2 + 2n + 1$. So $|uv^2wx^2y|$ strictly lies between n^2 and $(n + 1)^2$ which means $uv^2wx^2y \notin L$, a contradiction. Hence $\{a^{n^2}: n \geq 1\}$ is not context-free.

6.20 (a) Take $z = a^nb^nc^n$ in $L(G)$. Write $z = uvwxy$, where $1 \leq |vx| \leq n$. So vx cannot contain all the three symbols a, b and c. So uv^2wx^2y contain additional occurrences of two symbols (found in vx) and the number of occurrences of the third symbol remains the same. This means the number of occurrences of the three symbols in uv^2wx^2y are not the same and so $uv^2wx^2y \notin L$. This is a contradiction. Hence the language is not context-free.

(b) As usual, n is the integer obtained from pumping lemma. Let $z = a^nb^nc^{2n}$. Then $z = uvwxy$, where $1 \leq |vx| \leq n$. So vx cannot contain all the three symbols a, b and c. If vx contains only a's and b's then we can choose i such that uv^iwx^iy has more than $2n$ occurrences of a (or b) and exactly $2n$ occurrences of c. This means $uv^iwx^iy \notin L$, a contradiction. In other cases too, we can get a contradiction by proper choice of i. Thus the given language is not context-free.

6.21 (a) Suppose $G = (V_N, \Sigma, P, S)$ is right-linear. A production of the form $A \to a_1a_2 \ldots a_mB$, $m \geq 2$ can be replaced by $A \to a_1A_1$, $A_1 \to a_2A_2 \ldots$, $A_{m-1} \to a_mB$. $A \to b_1b_2 \ldots b_m$, $m \geq 2$, can be replaced by $A \to b_1B_1$, $B_1 \to b_2B_2, \ldots, B_{m-2} \to b_{m-1}B_{m-1}$, $B_{m-1} \to b_m$. The required equivalent regular grammar G' is defined by the new productions constructed above.

If $G = (V_N, \Sigma, P, S)$ is left-linear, then an equivalent right-linear grammar can be defined as $G_1 = (V'_N, \Sigma, P_1, S)$, where P_1 consists of

(i) $S \to w$ when $S \to w$ is in P and $w \in \Sigma^*$,
(ii) $S \to wA$ when $A \to w$ is in P and $w \in \Sigma^*$,
(iii) $A \to wB$ when $B \to Aw$ is in P and $w \in \Sigma^*$,
(iv) $A \to w$ when $S \to Aw$ is in P and $w \in \Sigma^*$.

Let $w \in L(G)$. If $S \Rightarrow w$ then $S \to w$ is in P. Therefore, $S \to w$ is in P_1 (by (i)).

Assume $S \Rightarrow A_1w_1 \Rightarrow A_2w_2w_1 \Rightarrow \ldots A_{m-1}w_{m-1} \ldots w_1 \Rightarrow w_mw_{m-1} \ldots w_1 = w$ is a derivation in G. Then the productions applied in the derivation are $S \to A_1w_1$, $A_1 \to A_2w_2, \ldots, A_{m-1} \to w_m$. The induced productions in G_1 are

$$A_1 \to w_1,\ A_2 \to w_2A_1,\ A_3 \to w_3A_2,\ \ldots,\ S \to w_mA_{m-1}$$

(by (ii), (iii) and (iv) in the construction of P_1).
Taking the productions in the reverse order we get a derivation of G_1 as follows:

$$S \Rightarrow w_mA_{m-1} \Rightarrow \ldots \Rightarrow w_mw_{m-1} \ldots w_3A_2 \Rightarrow w_m \ldots w_2A_1 \Rightarrow w_m \ldots w_1$$

Thus $L(G) \subseteq L(G')$. The other inclusion can be proved in a similar way. So G is equivalent to a right-linear grammar G_1 which is equivalent to a regular grammar.

(b) Let $G = (\{S, A\}, \{a, b, c\}, P, S)$. where P consists of $S \to Sc \mid Ac$, $A \to aAb \mid ab$. G is linear (by the presence of $A \to aAb$).

$$L(G) = \{a^nb^nc^m \mid m, n \geq 1\}$$

Using pumping lemma we prove that $L(G)$ is not regular. Let n be the number of states in a finite automaton accepting $L(G)$.

Let $w = a^nb^nc^n$. By pumping lemma $w = xyz$, where $|xy| \leq n$ and $|y| > 0$. If $y = a^k$ then $xz = a^{n-k}b^nc^m$. This is not in $L(G)$. By pumping lemma. $xz \in L(G)$ a contradiction.

6.22 $L = L(G)$, where G is a regular grammar. For every variable A in G. $A \overset{*}{\Rightarrow} \alpha$ implies $\alpha = uB$, where $u \in \Sigma^*$ and $B \in V$, Thus G is nonself-embedding. To prove the sufficiency part, assume that G is a nonself-embedding, context-free grammar. If G' is reduced, in Greibach normal form and equivalent to G, then G' is also nonself-embedding. (This can be proved.) Let $|\Sigma| = n$ and m be the maximum of the lengths of right-hand sides of productions in G'. Let α be any

sentential form. By considering leftmost derivations, we can show that the number of variables in α is $\leq mn$. (Use the fact that G' is in GNF). Define:

$G_1 = (V'_N, \Sigma, P_1, S)$ where
$V'_N = \{[\alpha] \mid |\alpha| \leq mn \text{ and } a \in V_N^+\}$
$S_1 = [S_1]$
$P_1 = \{[A\beta] \to b[a\ \beta] \mid A \to b\alpha \text{ is in } P,\ \beta \in V'_N \text{ and } |\alpha\beta| \leq mn\}$
G_1 is regular. It can be verified that $L(G_1) = L(G')$.

Chapter 7

7.1 $(q_0, aacaa, Z_0) \vdash (q_0, acaa, aZ_0) \vdash (q_0, caa, aaZ_0) \vdash (q_1, a, aaZ_0)$ $\vdash (q_1, a, aZ_0) \vdash (q_1, \Lambda, Z_0) \vdash (q_f, \Lambda, Z_0)$.

(i) Yes, the final ID is (q_f, Λ, Z_0).
(ii) Yes, the final ID is (q_1, Λ, aZ_0).
(iii) No, the pda halts at (q_1, ba, aZ_0).
(iv) Yes, the final ID is $(q_1, \Lambda, abaZ_0)$
(v) Yes, the final ID is $(q_0, \Lambda, babaZ_0)$.

7.2 (i) (q_1, Λ, aZ_0).
(ii) Halts at (q_1, b, Λ).
(iii) (q_0, A, a^5Z_0).
(iv) Does not move.
(v) Does not move.
(vi) Halts at (q_1, ab, Z_0).

7.3 (a) Example 7.9.
(b) The required pda A is defined as follows:

$A = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, \emptyset)$. δ is defined by

$$\delta(q_0, a, Z_0) = \{(q_1, aZ_0)\}, \qquad \delta(q_1, a, a) = \{(q_1, aa)\}$$
$$\delta(q_1, b, a) = \{(q_2, a)\}, \qquad \delta(q_2, b, a) = \{(q_1, \Lambda)\}$$
$$\delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}.$$

(c) $A = (\{q_0, q_1\}, \{a, b, c\}, \{Z_0, Z_1\}, \delta, q_0, Z_0, \emptyset)$

δ is defined by
$\delta(q_0, a, Z_0) = \{(q_0, Z_1Z_0)\}, \qquad \delta(q_0, a, Z_1) = \{(q_0, Z_1Z_1)\}$
$\delta(q_0, b, Z_1) = \{(q_1, \Lambda)\}, \qquad \delta(q_1, b, Z_1) = \{(q_1, \Lambda)\}$
$\delta(q_1, c, Z_0) = \{(q_1, Z_0)\}, \qquad \delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$
Note that, on reading a, we add Z_1; on reading b we remove Z_1 and the state is changed. If the input is completely read and the stack symbol is Z_0, then it is removed by a Λ-move.

7.4 (a) Example 7.9 gives a pda accepting $\{a^nb^ma^n \mid m, n \geq 1\}$ by null store. Using Theorem 7.1, a pda B accepting the given language by final state is constructed.

$$B = (\{q_0, q_1, q'_0, q_f\}, \{a, b\}, \{a, Z_0, Z'_0\}, \delta, q'_0, Z'_0, \{q_f\})$$

δ is defined by

$\delta(q'_0, \Lambda, Z'_0) = \{(q_0, Z_0Z'_0)\}$

$\delta(q_0, \Lambda, Z'_0) = \{(q_f, \Lambda)\} = \delta(q_0, \Lambda, Z'_0)$

$\delta(q_1, \Lambda, Z'_0) = \{(q_f, \Lambda)\} = \delta(q_1, \Lambda, Z'_0)$

$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}, \quad \delta(q_0, a, a) = \{(q_0, aa)\}$

$\delta(q_0, b, a) = \{(q_1, a)\}, \quad \delta(q_1, b, a) = \{(q_1, a)\}$

$\delta(q_1, a, a) = \{(q_f, \Lambda)\}, \quad \delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$

7.5 (a) $G = (\{S, S_1, S_2\}, \{a, b\}, P, S)$ generates the given language, where P consists of $S \to S_1$, $S \to S_2$, $S_1 \to aS_1b$, $S_1 \to ab$, $S_2 \to aS_2bb$, $S_2 \to abb$. The pda accepting $L(G)$ by null store is

$$A = (\{q\}, \{a, b\}, \{S, S_1, S_2, a, b\}, \delta, q, S, \emptyset)$$

where δ is defined by the following rules:

$$\delta(q, \Lambda, S) = \{(q, S_1), (q, S_2)\}$$
$$\delta(q, \Lambda, S_1) = \{(q, aS_1, b), (q, ab)\}$$
$$\delta(q, \Lambda, S_2) = \{(q, aS_2bb), (q, abb)\}$$
$$\delta(q, a, a) = \delta(q, b, b) = \{(q, \Lambda)\}$$

7.6 (i) $G = (\{S\}, \{a, b\}, P, S)$, where P consists of $S \to aSb$, $S \to aS$, $S \to a$, generates
$\{a^mb^n \mid n < m\}$. For $S \Rightarrow a^mSb^m$, $m \geq 0$.
$S \Rightarrow a^n$, $n \geq 1$, and hence $S \Rightarrow a^ma^nb^m$, $m \geq 0$, $n \geq 1$.
So $L(G) \subseteq \{a^mb^n \mid n < m\}$. The other inclusion can be proved similarly.

(ii) The pda A accepting $L(G)$ by null store is given by

$$A = (\{q\}, \{a, b\}, \{S, a, b\}, \delta, q, S, \emptyset)$$

where δ *is* defined by the following rules:

$$\delta(q, \Lambda, S) = \{(q, aSb), (q, aS), (q, a)\}$$
$$\delta(q, a, a) = \delta(q, b, b) = \{(q, \Lambda)\}$$

(iii) Define $B = (Q', \Sigma, \Gamma', \delta_B, q'_0, Z'_0, F)$, where

$$Q' = \{q_0, q'_0, q_f\}, \qquad \Gamma' = (S, a, b, Z'_0)$$

$F = \{q_f\}$. (We apply Theorem 7.1 to (ii)).

δ_B is given by

$\delta_B(q'_0, \Lambda, Z'_0) = \{(q, Z_0Z'_0)\}$

$\delta_B(q, \Lambda, S) = \{(q, aSb), (q, aS), (q, a)\}$

$\delta_B(q, a, a) = \{(q, \Lambda)\} = \delta_B(q, b, b)$

$\delta_B(q, \Lambda, Z'_0) = \{(q_f, \Lambda)\}$

(***Note:*** $\delta_B(q, a, S) = \delta(q, a, S) = \emptyset$ and $\delta_B(q, b, S) = \delta(q, b, S) = \emptyset$)

7.7 (i) Define $G = (\{S\}, \{a, b\}, P, S)$, where P consists of

$$S \to SaSbSaS,\ S \to SaSaSbS,\ S \to SbSaSaS,\ S \to \Lambda$$

(Refer to Exercise 4.10(e).) G is the required grammar. (ii) Apply Theorem 7.3. (iii) Apply Theorem 7.1 to the pda obtained in (ii).

7.8 Let $A = (\{q_0, q_1\}, \{a, b\}, \{Z_0, a, b\}, \delta, q_0, Z_0, \emptyset)$, where δ is given by

$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$, $\quad \delta(q_0, b, Z_0) = \{(q_0, bZ_0)\}$

$\delta(q_0, a, b) = \{(q_0, ab)\}$, $\quad \delta(q_0, b, a) = \{(q_0, ba)\}$

$\delta(q_0, a, a) = \{(q_0, aa), (q_1, \Lambda)\}$

$\delta(q_0, b, b) = \{(q_0, bb), (q, \Lambda)\}$

$\delta(q_0, \Lambda, Z_0) = \{(q_1, \Lambda)\}$

$\delta(q_1, a, a) = \{(q_1, \Lambda)\}$, $\quad \delta(q_1, b, b) = \{(q_1, \Lambda)\}$

$\delta(q_1, \Lambda, Z_0) = \{(q_1, A)\}$

A makes a guess whether it has reached the centre of the string. A reaches the centre only when the input symbol and the topmost symbol on PDS are the same. This explains the definition of $\delta(q_0, a, a)$ and $\delta(q_0, b, b)$. A accepts the given set by null store.

7.9 Example 7.6 gives a pda A accepting the given set by empty store. The only problem is that it is not deterministic. We have $\delta(q, a, Z_0) = \{(q, aZ_0)\}$ and $\delta(q, \Lambda, Z_0) = \{(q, \Lambda)\}$. So A is not deterministic (refer to Definition 7.5). But the construction can be modified as follows:

$$A_1 = (\{q, q_1\}, \{a, b\}, \{Z_0, a, b\}, \delta, q, Z_0, \emptyset)$$

where δ is defined by the following rules:

$\delta(q, a, Z_0) = \{(q_1, aZ_0)\}$, $\quad \delta(q, b, Z_0) = \{(q_1, bZ_0)\}$

$\delta(q_1, a, a) = \{(q_1, aa)\}$, $\quad \delta(q_1, b, b) = \{(q_1, bb)\}$

$\delta(q_1, a, b) = \{(q_1, \Lambda)\}$, $\quad \delta(q_1, b, a) = \{(q_1, \Lambda)\}$

$\delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$

A_1 is deterministic and accepts the given set by empty store.

7.10 The S-productions are

$$S \to [q_0, Z_0, q_0]\,|\,[q_0, Z_0, q_1]$$

$$\delta(q_1, b, a) = \{(q_1, \Lambda)\}, \quad \delta(q_1, \Lambda, Z_0) = \{(q_1, \Lambda)\}$$

and

$$\delta(q_0, b, a) = \{(q_1, \Lambda)\}$$

Now these induce $[q_1, a, q_1] \to b$, $[q_1, Z_0, q_1] \to \Lambda$ and $[q_0, a, q_1] \to b$, respectively. $\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$ induces

$[q_0, Z_0, q_0] \rightarrow a[q_0, a, q_0][q_0, Z_0, q_0]$

$[q_0, Z_0, q_0] \rightarrow a[q_0, a, q_1][q_1, Z_0, q_0]$

$[q_0, Z_0, q_1] \rightarrow a[q_0, a, q_0][q_0, Z_0, q_1]$

$[q_0, Z_0, q_1] \rightarrow a[q_0, a, q_1][q_1, Z_0, q_1]$

$\delta(q_0, a, a) = \{(q_0, aa)\}$ induces

$[q_0, a, q_0] \rightarrow a[q_0, a, q_0][q_0, a, q_0]$

$[q_0, a, q_0] \rightarrow a[q_0, a, q_1][q_1, a, q_0]$

$[q_0, a, q_1] \rightarrow a[q_0, a, q_0][q_0, a, q_1]$

$[q_0, a, q_1] \rightarrow a[q_0, a, q_1][q_1, a, q_1]$

7.13 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting a given regular set. Define a pda A by $A = (Q, \Sigma, \{Z_0\}, \delta_1, q_0, Z_0, F)$. δ is given by the following rule:

$$\delta_1(q, a, Z_0) = \{(q'_0, Z_0)\} \qquad \text{if } \delta(q, a) = q'$$

It is easy to see that $T(M) = T(A)$. Let $w \in T(M)$. Then $\delta(q_0, w) = q' \in F$.

$\delta(q_0, w, Z_0) = \{(q', Z_0)\}$. So $w \in T(A)$, i.e., $T(M) \subseteq T(A)$. The proof that $T(A) \subseteq T(M)$ is similar.

7.15 If $\delta(q, a, z)$ contains $(q', Z_1Z_2 \ldots Z_n)$, $n \geq 3$, we introduce new states $q_1, q_2, \ldots, q_{n-2}$. We define new transitions involving new states as follows:

(i) $(q_1, Z_{n-1}Z_n)$ is included in $\delta(q, a, Z)$

(ii) $\delta(q_i, \Lambda, Z_{n-i}) = \{(q_{i+1}, Z_{n-i-1}\ Z_{n-i})\}$ for $i = 1, 2, \ldots, n-3$

(iii) $\delta(q_{n-2}, \Lambda, Z_2) = \{(q', Z_1Z_2)\}$

This construction is repeated for every transition given by $(q', \gamma) \in \delta(q, a, Z)$, $|\gamma| \geq 3$. Deleting such transitions and adding the new transitions induced by them we get a pda which never adds more than one symbol at a time.

Chapter 8

8.1 For a sentential form such as $a^{n+1}b^n$, $A \rightarrow a$ is the production applied in the last step only when a is followed by ab. So $A \rightarrow a$ is a handle production if and only if the symbol to the right of a is scanned and found to be b. Similarly, $A \rightarrow aAb$ is a handle production if and only if the symbol to the right of aAb is b. Also, $S \rightarrow aAb$ is a handle production if and only if the symbol to the right of aAb is Λ. Therefore, the grammar is LR(1), but not LR(0).

8.2 We can actually show that the given grammar is not LR(k) for any $k \geq 0$. Suppose it is LR(k) for some k. Consider the rightmost

derivations of $01^{2k+1}2$ and $01^{2k+3}2$ given by:

$$S \underset{R}{\overset{*}{\Rightarrow}} 01^kA1^k2 \underset{R}{\Rightarrow} 01^{2k+1}2 = \alpha\beta w \tag{A8.1}$$

where $\alpha = 01^k$, $\beta = a$, $w = 1^k2$.

$$S \underset{R}{\overset{*}{\Rightarrow}} 01^{k+1}A1^{k+1}2 \underset{R}{\Rightarrow} 01^{2k+3}\,2 \Rightarrow \alpha'\beta'w' \tag{A8.2}$$

where $\alpha' = 01^{k+1}$, $\beta' = a$, $w' = 1^{k+1}2$. As the strings formed by the first $2k + 1$ symbols (note $|\alpha\beta| + k = 2k + 1$) of $\alpha\beta w$ and $\alpha'\beta'w'$ are the same. $\alpha = \alpha'$, i.e. $01^k = 01^{k+1}$, which is a contradiction. Thus the given grammar is not LR(k) for any k.

8.3 The given grammar is ambiguous and hence is not LR(k) for any k. For example, there are two derivation trees for ab.

8.4 As $a^nb^nc^n$ appears in both the sets, it admits two different derivation trees. So the set cannot be generated by an unambiguous grammar.

Chapter 9

9.2 The set of quintuples representing the TM consists of q_1b1Lq_2, q_100Rq_1, q_2bbRq_3, q_200Lq_2, q_211Lq_2, q_30bRq_4, q_31bRq_5, q_4b0Rq_5, q_400Rq_4, q_411Rq_4, q_5b0Lq_2.

9.3 The computation for the first symbol 1 is $q_111b11 \vdash bq_2b11$. Afterwards it halts.

9.4 The computation sequence for the substring 12 of 1213 is

$$q_11213 \vdash bq_2213 \vdash bbq_313.$$

As $\delta(q_3, 1)$ is not defined, the TM halts. For 2133 and 312 the TM does not start.

9.6 Modify the construction given in Example 9.7.

9.8 We have the following steps for processing the even-length palindromes:

(a) The Turing machine M scans the first symbol of the input tape (0 or 1), erases it and changes state (q_1 or q_2).
(b) M scans the remaining part without changing the tape symbol until it encounters b.
(c) The R/W head moves to the left. If the rightmost symbol tallies with the leftmost symbol (which can be erased but remembered), the rightmost symbol is erased. Otherwise M halts.
(d) The R/W head moves to the left until b is encountered.

Steps (a), (b), (c), (d) are repeated after changing the states suitably. The transition table is defined by Table A9.1.

TABLE A9.1 Transition Table for Exercise 9.8

Present state	Input symbol		
	0	1	b
$\rightarrow q_0$	bRq_1	bRq_2	bRq_7
q_1	$0Rq_1$	$1Rq_1$	bLq_3
q_2	$0Rq_1$	$1Rq_2$	bLq_4
q_3	bLq_5		
q_4		bLq_6	
q_5	$0Lq_5$	$1Lq_5$	bRq_0
q_6	$0Lq_6$	$1Lq_6$	bRq_0
(q_7)			

9.9 We have three states q_0, q_1, q_f, where q_0 is the initial state used to remember that even number of 1's have been encountered so far. q_1 is used to remember that odd number of 1's have been encountered so far. q_f is the final state. The transition table is defined by Table A9.2.

TABLE A9.2 Transition Table for Exercise 9.9

Present state	0	1	b
$\rightarrow q_0$	$0Rq_0$	$1Rq_1$	bRq_f
q_1	$0Rq_1$	$1Rq_0$	
(q_f)			

9.10 The construction given in Example 9.7 can be modified. As the number of occurrences of c is independent of that of a or b, after scanning the rightmost c, the R/W head can move to the left and erase c.

9.11 Assume that the input tape has 0^m10^n where $m \div n$ is required. We have the following steps:

(a) The leftmost 0 is replaced by b and the R/W head moves to the right.

(b) The R/W head replaces the first 0 after 1 by 1 and moves to the left. On reaching the blank at the left end the cycle is repeated.

(c) Once the 0's to the left of 1's are exhausted, M replaces all 0's and 1's by b's. $a \div b$ is the number of 0's left over in the input tape and equal to 0.

(d) Once the 0's to the right of 1's are exhausted, n 0's have been changed to 1's and $n + 1$ of m 0's have been changed to b. M replaces 1's (there are $n + 1$ 1's) by one 0 and n b's. The number of 0's remaining gives the values of $a \div b$. The transition table is defined by Table A9.3.

TABLE A9.3 Transition Table for Exercise 9.11

Present state	Input symbol		
	0	1	b
q_0	bRq_1	bRq_5	
q_1	$0Rq_1$	$1Rq_2$	
q_2	$1Lq_3$	$1Rq_2$	bLq_4
q_3	$0Lq_3$	$1Lq_3$	bRq_0
q_4	$0Lq_4$	bLq_4	$0Rq_6$
q_5	bRq_5	bRq_5	bRq_6
(q_6)			

Chapter 10

10.2 1. (B, w) is an input to M.
2. Convert B to an equivalent DFA A.
3. Run the Turing machine M_1 for A_{DFA} on input (A, w)
4. If M_1 accepts, M accepts; otherwise M rejects

10.3 Construct a TM M as follows:
1. (A) is an input to M.
2. Mark the initial state of A (q_0 marked as q_0^*, a new symbol).
3. Repeat until no new states are marked: a new state is marked if there is a transition from a state already marked to the new state.
4. If a final state is marked, M accepts (A); otherwise it rejects.

10.4 Let $L = (T(A_1) - T(A_2)) \cup (T(A_2) - T(A_1))$. L is regular and $L = T(A')$. Apply E_{DFA} to (A').

10.8 Use Examples 10.4 and 10.5.

10.9 A_{TM} is regarding a given Turing machine accepting an input, that is, reaching an accepting state after scanning w and halting $HALT_{TM}$ is regarding a given TM halting on an input (or M need not accept w in this case).

10.10 Represent a number between 0 and 1 as $0 \cdot a_1a_2 \ldots$ where $a_1, a_2, \ldots$ are binary digits. Assume the set to be a sequence, apply diagonalization process and get a contradiction.

10.11 When a problem is undecidable, we can modify or take a particular case of the problem and try for algorithms. Studying undecidable problems may kindle an imagination to get better ideas on computation.

10.12 Suppose the problem is solvable. Then there is an algorithm to decide whether a given terminal string w is in L. Let M be a TM. Then there is a grammar G such that $L(G)$ is the same as the set accepted by M. Then $w \in L(G)$ if and only if M halts on w. This means that the halting problem of TM is solvable, which is a contradiction. Hence the recursiveness of a type 0 grammar is unsolvable.

10.14 Suppose there exists a Turing Machine M over $\{0, 1\}$ and a state q_m such that the problem of determining whether or not M will enter q_m is unsolvable. Define a new Turing machine M' which simulates M and has the additional transition given by $\delta(q_m, \Lambda) = (q_m, 1, R)$. Then M enters q_m when it starts with a given tape configuration if and only if M' prints 1 when it starts with a given tape configuration. Hence the given problem is unsolvable.

10.17 According to Church's thesis we can construct a Turing Machine which can execute any given algorithm. Hence the given statement is false. (Of course, the Church's thesis is not proved. But there is enough evidence to justify its acceptance.)

10.18 Let $\Sigma = \{a\}$. Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$, where $x_i = a^{k_i}$, $i = 1, 2, \ldots, n$ and $y_j = a^{l_j}$, $j = 1, 2, \ldots, n$. Then $(x_1)^{l_1}(x_2)^{l_2} \ldots (x_n)^{l_n} = (y_1)^{k_1}(y_2)^{k_2} \ldots (y_n)^{k_n}$. Both are equal to $a^{\Sigma k_i l_i}$. Hence PCP is solvable when $|\Sigma| = 1$.

10.20 $x_1 = 01$, $y_1 = 011$, $x_2 = 1$, $y_2 = 10$, $x_3 = 1$, $y_3 = 1$. Hence $|x_i| < |y_i|$ for $i = 1, 2, 3$. So $x_{i_1}x_{i_2} \ldots x_{i_m} \neq y_{i_1}y_{i_2} \ldots y_{i_m}$ for no choice of i's. ***Note:*** $|x_{i_1}x_{i_2} \ldots x_{i_m}| < |y_{i_1}y_{i_2} \ldots y_{i_m}|$. Hence the PCP with the given lists has no solution.

10.21 $x_1 = 0$, $y_1 = 10$, $x_2 = 110$, $y_2 = 000$, $x_3 = 001$, $y_3 = 10$. Here no pair (x_1, y_1), (x_2, y_2) or (x_3, y_3) has common nonempty initial substring. So $x_{i_1}x_{i_2} \ldots x_{i_m} \neq y_{i_1}y_{i_2} \ldots y_{i_m}$ for no choice of i_j's. Hence the PCP with the given lists has solution.

10.22 As $x_1 = y_1$, the PCP with the given lists has a solution.

10.23 In this problem, $x_1 = 1$, $x_2 = 10$, $x_3 = 1011$, $y_1 = 111$, $y_2 = 0$, $y_3 = 10$. Then, $x_3x_1x_1x_2 = y_3y_1y_1y_2 = 101111110$. Hence the PCP with the given lists has a solution. Repeating the sequence 3, 1, 1, 2, we can get more solutions.

10.24 Both (a) and (b) are possible. One of them is possible by Church's thesis. Find out which one?

Chapter 11

11.1 (a) The function is defined for all natural numbers divisible by 3.
(b) $x = 2$
(c) $x \geq 2$
(d) all natural numbers
(e) all natural numbers

11.2 (a) $\chi_{\{0\}}(0) = 1$, $\chi_{\{0\}}(x + 1) \doteq \chi_{\{0\}}\text{sgn}(p(x))$

(b) $f(x + 1) = x^2 + 2x + 1$

So, $f(x + 1) = f(x) + S(S(Z(x))) * U_1^1(x) + S(Z(x))$

Hence f is obtained by recursion and addition of primitive recursive functions

(c) $f(x, y) = y + (x \dot{-} y)$

(d) Define parity function $P_r(y)$ by

$$P_r(0) = P_r(2) = \cdots = 0, \qquad P_r(1) = P_r(3) = \cdots = 1$$

P_r is primitive recursive since $P_r(0) = 0$, $P_r(x + 1) = \chi_{\{0\}}(U_2^2(x), P_r(x))$. Define f by $f(0) = 0$, $f(x + 1) = f(x) + P_r(x)$.

(e) $\text{sgn}(0) = Z(0)$, $\text{sgn}(x + 1) = S(Z(U_2^2\ (x, \text{sgn}\ (x))))$

(f) $L(x, y) = \text{sgn}(x \dot{-} y)$

(g) $E(x, y) = \chi_{\{0\}}((x \dot{-} y) + (y \dot{-} x))$

All the functions (a)–(g) are obtained by applying composition and recursion to known primitive functions and hence primitive recursive functions.

11.3 $A(1, y) = A(1 + 0, y - 1 + 1) = A(0, A(1, y - 1))$ using (11.10) of Example 11.11. Using (11.8), we get

$$A(1, y) = 1 + A(1, y - 1).$$

Repeating the argument, we have

$A(1, y) = y - 1 + A(1, 1) = y + 2$ (By Example 11.12, $A(1, 1) = 3$).

This result is used in evaluating $A(3, 1)$.

$A(2, 3) = A(1 + 1, 2 + 1) = A(1, A(2, 2)) = A(1, 7)$ using Example 11.12. Using $A(1, y) = y + 2$, we get $A(2, 3) = 2 + 7 = 9$. Then using (11.10), $A(3, 1) = A(2 + 1, 0 + 1) = A(2, A(3, 0))$ By Example 11.12, $A(2, 1) = 5$. Also, $A(3, 0) = A(2, 1)$ by (11.9). Hence $A(3, 1) = A(2, 5) = A(1 + 1, 4 + 1) = A(1, A(2, 4))$. Since $A(1, y) = y + 2$, $A(3, 1) = 2 + A(2, 4)$. Applying (11.10), we have $A(2, 4) = A(1, A(2, 3)) = 2 + A(2, 3) = 2 + 9 = 11$.

Hence, $A(3, 1) = 2 + 11 = 13$.

$A(3, 2) = A(2, A(3, 1)) = A(2, 13) = A(1, A(2, 12)) = 2 + A(2, 12)$ $= 2 + A(1, A(I, 11)) = 2 + A(1, 13) = 2 + 2 + 13 = 17$.

To evaluate $A(3, 3)$, we prove $A(2, y + 1) = 2y + A(2, 1)$. Now,

$$A(2, y + 1) = A(1 + 1, y + 1) = A(1, A(2, y) = 2 + A(2, y).$$

Repeating this argument, $A(2, y + 1) = 2y + A(2, 1)$. Now,

$A(3, 3) = A(2 + 1, 2 + 1) = A(2, A(3, 2)) = A(2, 17) = 2(16) +$ $A(2, 1) = 32 + 5 = 37$.

11.4 (b) It is clear that $r(x, 0) = 0$. Also, $r(x, y)$ increases by 1 when y is increased by 1 and $r(x, y) = 0$ when $y = x$. Using these observations we see that $r(x, y + 1) = S(r(x, y)) * \text{sgn}(x \dot{-} S(r(x, y)))$. Hence $r(x, y)$ is 1 defined by

$$r(x, 0) = 0$$

$$r(x, y + 1) = S(r(x, y)) * \text{sgn}(x \dot{-} S(r(x, y)))$$

11.5 $f(x)$ is the smallest value of y for which $(y + 1)^2 > x$. Therefore, $f(x) = \mu_y(\chi_{\{0\}}((y + 1)^2 \dot{-} x))$, f is partial recursive since it is obtained from primitive recursive functions by application of minimization.

11.8 The constant function $f(x) = 1$ is primitive recursive for $f(0) = 1$ and $f(x + 1) = U_2^2(x, f(x))$. Now χ_{A^c}, $\chi_{A \cap B}$ and $\chi_{A \cup B}$ are recursive for $\chi_{A^c} = 1 \dot{-} \chi_A$, $\chi_{A \cap B} = \chi_A * \chi_B$ and $\chi_{A \cup B} = \chi_A + \chi_B \dot{-} \chi_{A \cap B}$

(Addition and proper subtraction are primitive recursive functions and the given functions are obtained from recursive functions using composition.)

11.9 Let E denote the set of all even numbers. $\chi_E(0) = 0$, $\chi_E(n + 1) = 1 \dot{-} \text{sgn}(U_2^2(n, \chi_E(n)))$. The sign function and proper subtraction function are primitive recursive. Thus E is obtained from primitive recursive functions using recursion. Hence χ_E is primitive recursive and hence recursive. To prove the other part use Exercise 11.8.

11.11 Define f by $f(0) = k$, $f(n + 1) = U_2^2(n, f(n))$. Hence f is primitive recursive.

11.12 $\chi_{\{a_1, a_2, \ldots, a_n\}} = \chi_{\{a_1\}} + \chi_{\{a_2\}} + \ldots + \chi_{\{a_n\}}$. As $\chi_{\{a_1\}}$ is primitive recursive. (Refer to Exercise 11.2(a)) and the sum of primitive recursive functions is primitive recursive, $\chi_{\{a_1, a_2, \ldots, a_n\}}$ is primitive recursive.

11.13 Represent x and y in tally notation. Using Example 9.6 we can compute concatenation of strings representing x and y which is precisely $x + y$ in tally notation.

11.14 Let M in the Post notation have $\{q_1, q_2, \ldots, q_n\}$ and $\{a_1, a_2, \ldots, a_m\}$ as Q and Σ respectively. Let $Q' = \{q_1, \ldots, q_n, q_{n+1}, \ldots, q_{2n}\}$, where $q_{n+1} \ldots, q_{2n}$ are new states. Let a quadruple of the form $q_i a_j R q_k$ induce the quintuple $q_i a_j a_k R q_k$. Let a quadruple of the form $q_i a_j L q_k$ induce the quintuple $q_i a_j a_j L q_k$. Finally, let $q_i a_j a_k q_l$ induce $q_i a_j a_k R q_{n+i}$. We introduce quintuples $q_{n+i} a_t a_t L q_i$ for $i = 1, 2, \ldots, n$ and $t = 1, 2, 3, \ldots, m$. The required TM has Q' as the set of states and the set of quintuples represent δ.

11.15 $q_0 1111 x_1 by \overset{*}{\vdash} 1111 q_0 x_1 by \vdash 1111 q_1 xby$. As b lies between x_1 and y, $Z(4) = 0$ (given by b).

11.16 In Section 11.4.5 we obtained $q_0 1 x_1 by \overset{*}{\vdash} q_5 1 x_1, bly$. Similarly, $q_0 111 x_1 by \overset{*}{\vdash} q_5 111 x_1 b1y \vdash 1 q_0 11 x_1 bly$. Proceeding further, $1 q_0 11 x_1 b1y \overset{*}{\vdash} 1 q_5 11 x_1 b11y \vdash 11 q_0 1 x_1 b11y \overset{*}{\vdash} 111 q_9 x_1 1111y$ (as in Section 11.4.5). Hence $S(3) = 4$.

11.18 Represent the argument x in tally notation. $f(x) = S(S(x))$. Using the construction given in Section 11.4.7, we can construct a TM which gives the value $S(S(x))$.

11.19 $f(x_1, x_2) = S(S(U_1^2(x_1, x_2)))$. Use the construction in Section 11.4.7.

11.20 Represent (x_1, x_2) by $1^{x_1} b 1^{x_2}$. By taking the input as $\$1^{x_1} b 1^{x_2}$($ is representing the left-end) and suitably modifying the TM given in Example 9.6, we get the value of $x_1 + x_2$ to the right of $.

Chapter 12

12.1 Denote $f(n) = \sum_{i=0}^{k} a_i n^i$ and $g(n) = \sum_{j=0}^{l} b_j n^j$, where a_i, b_j, are positive integers. Assume $k \geq l$. Then $f(n) + g(n) = \sum_{i=0}^{k} (a_i + b_i) n^i$, where $b_i = 0$ for $i > l$. $f(n) + g(n)$ is a polynomial of degree k. Hence $f(n)g(n) = O(n^{k+l})$.

12.2 As n^2 dominates $n \log n$ and $n^2 \log n$ dominates n^2, the growth rate of $h(n)$ > growth rate of $g(n)$. Note $f(n) = g(n) = O(n^2)$.

12.3 As $\sum_{i=0}^{n} i = n(n+1)/2$, $\sum_{i=0}^{n} i^2 = n(n+1)(2n+1)/6$ and $\sum_{i=0}^{n} i^3 = (n(n+1)/2)^2$, the answers for (i) and (ii) are $O(n^3)$ and $O(n^2)$. (iii) $a(1 - r^n)/1 - r = O(r^n/r) = O(r^{n-1})$. (iv) $\frac{n}{2}[2a + (n-1)d] = O(n^2)$.

12.4 As $\log_2 n$, $\log_3 n$, $\log_e n$, differ by a constant factor, $f(n) = O(n^2 \log n)$

12.5 gcd = 3.

12.6 The principal disjunctive normal form of the boolean expression has 5 terms (refer to Example 1.13). $P \wedge Q \wedge R$ is one such term. So (T, T, T) satisfies the given expression. Similar assignments for the other four terms.

12.7 No.

12.8 (T, T, F, F) makes the given expression satisfiable.

12.9 Only if: Take an *NP*-complete problem L. Then $\overline{L}$ is in **CO-NP = NP**.

if: Let P be NP-complete and $\overline{P} \in \mathbf{NP}$. Let L be any language in **NP**. We get a polynomial reduction ϕ of L to P and hence a polynomial reduction ψ of $\overline{L}$ to $\overline{P}$. We prove $\mathbf{NP} \subset \mathbf{CO\text{-}NP}$. Combine ψ and nondeterministic polynomial-time algorithms for $\overline{P}$ to get a nondeterministic polynomial-time algorithm for $\overline{L}$. So $\overline{L} \in \mathbf{NP}$ or $L \in \mathbf{CO\text{-}NP}$. This proves $\mathbf{NP} \subset \mathbf{CO\text{-}NP}$. The other inclusion is similar.

Further Reading

Chandrasekaran, N., Automata and Computers, *Proceedings of the KMA National Seminar on Discrete Mathematics and Applications*, St. Thomas College, Kozhencherry, January, 9–11, 2003.

Davis, M.D. and E.J. Weyuker, *Computability, Complexity and Languages, Fundamentals of Theoretical Computer Science*, Academic Press, New York, 1983.

Deo, N., *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall of India, New Delhi, 2001.

Ginsburg, S., *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966.

Glorioso, R.M., *Engineering Cybernetics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

Gries, D., *The Science of Programming,* Narosa Publishing House, New Delhi, 1981.

Harrison, M.A., *Introduction to Formal Language Theory*, Addison-Wesley, Reading (Mass.), 1978.

Hein, J.L., *Discrete Structures, Logic and Computability*, Narosa Publishing House, New Delhi, 2004.

Hopcroft, J.E. and J.D. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Welsey, Reading (Mass.), 1969.

Hopcroft J.E., and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Narosa Publishing House, New Delhi, 1987.

Hopcroft, J.E., J. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Pearson Education, Asia, 2002.

Kain, R.Y., *Automata Theory: Machines and Languages,* McGraw-Hill, New York, 1972.

Kohavi, ZVI, *Switching and Finite Automata Theory*, Tata McGraw-Hill, New Delhi, 1986.

Korfhage, R.R., *Discrete Computational Structures*, Academic Press New York, 1984.

Krishnamurthy, E.V., *Introductory Theory of Computer Science*, Affiliated East-West Press, New Delhi, 1984.

Levy, L.S., *Discrete Structures of Computer Science*, Wiley Eastern, New Delhi, 1988.

Lewis, H.R. and C.L. Papadimitrou, *Elements of the Theory of Computation,* 2nd ed., Prentice-Hall of India, New Delhi, 2003.

Linz, P., *An Introduction to Formal Languages and Automata*, Narosa Publishing House, New Delhi, 1997.

Mandelson, E., *Introduction to Mathematical Logic*, D. Van Nostrand, New York, 1964.

Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill Kogakusha, Tokyo, 1974.

Martin, J.H., *Introduction to Languages and the Theory of Computation*, McGraw-Hill International Edition, New York, 1991.

Minsky, M., *Computation: Finite and Infinite Machines,* Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

Nelson, R.J., *Introduction to Automata*, Wiley, New York, 1968.

Preparata, F.P. and R.T. Yeh, *Introduction to Discrete Structures*, Addison-Wesley, Reading (Mass.), 1973.

Rani Siromoney, *Formal Languages and Automata*, The Chiristian Literature Society, Madras, 1979.

Revesz, G.E., *Introduction to Formal Languages,* McGraw-Hill, New York, 1986.

Sahni, D.F. and D.F. McAllister, *Discrete Mathematics in Computer Science,* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

Sipser, M., *Introduction to the Theory of Computation,* Brooks/Cole, Thomson Learning, London, 2003.

Tremblay, J.P. and R. Monohar, *Discrete Mathematical Structures with Applications to Computer Science,* McGraw-Hill, New York, 1975.

Ullman, J.D., *Fundamental Concepts of Programming Systems,* Addison-Wesley, Reading (Mass.), 1976.

Index

Abelian group, 38
Acceptance
  by finite automaton, 77
  by NDFA, 79
  by pda by final state, 233–240
  by pda by null store, 234–240
  by Turing machine, 284
Accepting state (*see* Final state)
Ackermann's function, 330
Algebraic system, 39
Algorithm, 124, 309, 310
Alphabet, 54
Ambiguous grammar, 188
Ancestor of a vertex, 51
Arden's theorem, 139
Associativity, 38
A-tree, 183
Automaton, 71, 72
  minimization of, 91–97

Bijection, 45
Binary operation, 38
Binary trees, 51
Boolean expressions, 353
Bottom-up parsing, 258

Chain production (*see* Unit production)
Characteristic function, 344
Chomsky classification, 220
Chomsky normal form (*see* Normal form)
Church–Turing thesis, 362
Circuit, 49
Closure, 37
  properties, 126–128, 165–167, 272
  of relations, 43
Commutativity, 38, 39
Comparison method, 152
Complexity, 346–371
Composition of functions, 324
Concatenation, 54
Congruence relation, 41
Conjunction (AND), 2
Conjunctive normal form, 14
Connectives, 2–6
Construction of reduced grammar, 190–196
Context-free grammar (language), 122, 180–218
  decision algorithms for, 217
  normal forms for, 201–213
  and pda, 240–251
  simplification of, 189–201
Context-sensitive grammar (language), 120, 299
Contradiction, 8
Cook's theorem, 354
CSAT, 359

Decidability, 310
Decidable languages, 311

Decision algorithms (*see* Context-free grammar)
Derivation, 110, 111
  leftmost, 187
  rightmost, 187
Derivation tree (parse tree), 181, 184
  definition of, 181
  subtree of, 182
  yield of, 182
Descendant, 51
Deterministic
  finite automaton, 73
  pda, 236
Directed graph (or digraph), 47
Dirichlet drawer principle, 46
Disjunction (OR), 3
Disjunctive normal form, 11
Distributivity, 39

Elementary product, 11
Equivalence
  class, 42
  of DFA and NDFA, 80–84
  of finite automata, 157, 158
  of regular expressions, 160
  relation, 41
  of states, 91
  of well-formed formulas, 9
Euclidean algorithm, 349

Fibonacci numbers, 69
Field, 39
Final state, 73, 74, 77
Finite automaton
  deterministic, 73
  minimization of, 91–97
  nondeterministic, 78
  and regular expression, 153
Function (or map), 45
  by minimization, 330
  partial, 322
  by recursion, 328
  total, 322

Gödel, Kurt, 332
Grammar, 109
  monotonic, 121
  self-embedding, 226
Graph, 47
  connected, 49
  representation, 47
Greibach normal form (*see* Normal form)
Group, 38
Growth rate of functions, 346

Halting problem of Turing machine, 314–315
Hamiltonian circuit problem, 359
Handle production, 268
Hierarchy of languages, 120–122

ID (Instantaneous description)
  of pushdown automaton, 229
  of Turing machine, 279
Identities
  logical, 10
  for regular expressions, 138
Identity element, 38, 55
If and only if, 4
Implication (IF...THEN...), 4
Inclusion relation, 123
Initial function, 323
Induction, 57, 58, 60
Initial state, 73–74, 78, 228, 278
Internal vertex, 50
Inverse, 38

Kleene's theorem, 142
λ-move, 140
  elimination of, 141

Language(s)
  and automaton, 128
  classification of, 120
  generated by a grammar, 110
Leaf of a tree, 50
Length
  of path, 51
  of string, 55
Levi's theorem, 56
Linear bounded automaton (LBA), 297–299, 301–303
  and languages, 299–301
Logical connectives (*see* Connectives)
LR(k) grammar, 267

Map, 45, 46
  bijection (one-to-one correspondence), 45
  one-to-one (injective), 45
  onto (surjective), 45
Maxterm, 14
Mealy machine, 84
  transformation into Moore machine, 85–87
Minimization of automata, 91–97
Minterm, 12
Modus ponens, 16
Modus tollens, 16
Monoid, 38
Moore machine, 84
  transformation into Mealy machine, 87–89
Move relation
  in pda, 230
  in Turing machine, 280

NAND, 33
Negation (NOT), 2
Nondeterministic finite automaton, 78
  conversion to DFA, 80
Nondeterministic Turing machine, 295–297
NOR, 33
Normal form
  Chomsky, 201–203
  Greibach, 206–213
  of well-formed formulas, 11
NP-complete problem, 352
  importance of, 352
Null productions, 196
  elimination of, 196–199

One-to-one correspondence (*see also* Bijection), 45
Operations on languages, 126–128
Ordered directed tree, 50

Palindrome, 55, 113
Parse tree (*see* Derivation tree)
Parsing and pda, 251–260
Partial recursive function, 330
  and Turing machine, 332–340
Partition, 37
Path, 48
  acceptance by, 231–240
  pda, 230
Phrase structure grammar (*see* Grammar)
Pigeonhole principle, 46
Post correspondence problem (PCP), 315–317
Power set, 37
Predecessor, 48
Predicate, 19
Prefix of a string, 55
Primitive recursive function, 323–329
Principal conjunctive normal form, 15
  construction to obtain, 15
Principle of induction, 57
Production (or production rule), 109
Proof
  by contradiction, 61
  by induction, 57
  by modified method, 58
  by simultaneous induction, 60
Proposition (or Statement), 1
Propositional variable, 6
Pumping lemma
  for context-free languages and applications, 213, 216
  for regular sets and applications, 162–163
Push-down automaton, 227–251, 254
  and context-free languages, 240–251

Quantum computation, 360
Quantum computers, 361
Quantum bit (qubit), 361
Quantifier
  existential, 20
  universal, 20

Recursion, 37
Recursive definition of a set, 37
Recursive function, 329
Recursive set, 124
Recursively enumerable set, 124, 310
Reduction technique, 351
Reflexive-transitive closure, 43
Regular expressions, 136
  finite automata and, 140
  identities for, 138

Regular grammar, 122
Regular sets, 137
  closure properties of, 165–167
  and regular grammar, 167
Relations
  reflexive, 41
  symmetric, 41
  transitive, 41
Right-linear grammar, 226
Ring, 39
Root, 50
Russels paradox, 320

SAT problem (satisfiability problem), 353
Self-embedding grammar, 226
Semigroup, 38
Sentence, 110
Sentential form, 110
Sets, 36, 37, 38, 39, 40
  complement of, 37
  intersection of, 37
  union of, 37
Simple graph, 70
Start symbol, 109
Statement (*see* Proposition)
String
  empty, 54
  length of, 55
  operations on, 54
  prefix of, 55
  suffix of, 55
Strong Church–Turing thesis, 363
Subroutines, 290
Successor, 48
Symmetric difference, 68

Tautology, 8
Time complexity, 294, 349
Top-down parsing, 252
Top-down parsing, using deterministic pda's 256
Transition function, 73, 78, 228
  properties of, 75–76
Transition system, 74
  containing A-moves, 140
  and regular grammar, 169
Transitive closure, 43
Transpose, 55
Travelling salesman problem, 359
Tree, 49
  height of, 51
  properties of, 49–50
Turing-computable functions, 333
Turing machine, 277
  construction of, to compute the projection function, 336
  construction of, to compute the successor function, 335
  construction of, to compute the zero function, 334
  construction of, to pertorm composition, 338
  construction of, to perform minimization, 340
  construction of, to perform recursion, 339
  description of, 289
  design of, 284
  multiple track, 290
  multitape, 292
  nondeterministic, 295
  representation by ID, 279
  representation by transition diagram, 281
  representation by transition table, 280
  and type 0 grammar, 299–301
Type 0 grammar (*see* Grammar)
Type 1 grammar (*see* Context-sensitive grammar)
Type 2 grammar (*see* Context-free grammar)
Type 3 grammar (*see* Regular grammar)

Unambiguous grammar, 271
Undecidable language, 313
Unit production, 199
  elimination of, 199–201

Valid
  argument, 15
  predicate formula, 22
Variable, 109
Vertex, 47
  ancestor of, 51
  degree of, 48
  descendant of, 51
  son of, 51

Well-formed formula, 6
  or predicate calculus, 21

Theory of Computer Science

Third Edition

Automata, Languages and Computation

K.L.P. Mishra • N. Chandrasekaran

This Third Edition, in response to the enthusiastic reception given by academia and students to the previous edition, offers a cohesive presentation of all aspects of theoretical computer science, namely **automata**, **formal languages**, **computability**, and **complexity**. Besides, it includes coverage of mathematical preliminaries.

New to this Edition

- Expanded sections on pigeonhole principle and the principle of induction (both in Chapter 2)
- A rigorous proof of Kleene's theorem (Chapter 5)
- Major changes in the chapter on Turing machines (TMs)
  - A new section on high-level description of TMs
  - Techniques for the construction of TMs
  - Multitape TM and nondeterministic TM
- A new chapter (Chapter 10) on decidability and recursively enumerable languages
- A new chapter (Chapter 12) on complexity theory and NP-complete problems
- A section on quantum computation in Chapter 12.

Key Features

- Objective-type questions in each chapter—with answers provided at the end of the book.
- Eighty-three additional solved examples—added as Supplementary Examples in each chapter.
- Detailed solutions at the end of the book to chapter-end exercises.

The book is designed to meet the needs of the undergraduate and postgraduate students of computer science and engineering as well as those of the students offering courses in computer applications.

About the Authors

K.L.P. MISHRA (Ph.D., Leningrad), had a distinguished career as Professor of Electrical and Electronics Engineering, and Principal, Regional Engineering College, Tiruchirapalli.

N. CHANDRASEKARAN, Ph.D., is Professor of Mathematics at St. Joseph's College, Tiruchirapalli (an autonomous college, nationally accredited with five stars and a college selected for potential for excellence). He is a regular faculty at Bharathidasan Institute of Management, Tiruchirapalli and formerly served as a visiting professor at National Institute of Technology, Tiruchirapalli.

ISBN:978-81-203-2968-3

9 788120 329683

Rs. 225.00

www.phindia.com