



### **.Algorithms for Compiler Design**

by O.G. Kakde

ISBN:1584501006

[Charles River Media](#) © 2002(334 pages)

This text teaches the fundamental algorithms that underlie modern compilers, and focuses on the "front-end" of compiler design--lexical analysis, parsing, and syntax.

## **Table of Contents**

### [Algorithms for Compiler Design](#)

#### [Preface](#)

[Chapter 1](#) - Introduction

[Chapter 2](#) - Finite Automata and Regular Expressions

[Chapter 3](#) - Context-Free Grammar and Syntax Analysis

[Chapter 4](#) - Top-Down Parsing

[Chapter 5](#) - Bottom-up Parsing

[Chapter 6](#) - Syntax-Directed Definitions and Translations

[Chapter 7](#) - Symbol Table Management

[Chapter 8](#) - Storage Management

[Chapter 9](#) - Error Handling

[Chapter 10](#) - Code Optimization

[Chapter 11](#) - Code Generation

[Chapter 12](#) - Exercises

#### [Index](#)

#### [List of Figures](#)

#### [List of Tables](#)

#### [List of Examples](#)

## Back Cover

A compiler translates a high-level language program into a functionally equivalent low-level language program that can be understood and executed by the computer. Crucial to any computer system, effective compiler design is also one of the most complex areas of system development. Before any code for a modern compiler is even written, many programmers have difficulty with the high-level algorithms that will be necessary for the compiler to function. Written with this in mind, *Algorithms for Compiler Design* teaches the fundamental algorithms that underlie modern compilers. The book focuses on the “front-end” of compiler design: lexical analysis, parsing, and syntax. Blending theory with practical examples throughout, the book presents these difficult topics clearly and thoroughly. The final chapters on code generation and optimization complete a solid foundation for learning the broader requirements of an entire compiler design.

## FEATURES

- Focuses on the “front-end” of compiler design—lexical analysis, parsing, and syntax—topics basic to any introduction to compiler design
- Covers storage management, error handling, and recovery
- Introduces important “back-end” programming concepts, including code generation and optimization

# Algorithms for Compiler Design

O.G. Kakde

CHARLES RIVER MEDIA, INC.

Copyright © 2002, 2003 Laxmi Publications, LTD.

O.G. Kakde. *Algorithms for Compiler Design*

1-58450-100-6

No part of this publication may be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, or scanning, without prior permission in writing from the publisher.

**Publisher:** David Pallai

**Production:** Laxmi Publications, LTD.

**Cover Design:** The Printed Image

CHARLES RIVER MEDIA, INC.

20 Downer Avenue, Suite 3

Hingham, Massachusetts 02043

781-740-0400

781-740-8816 (FAX)

[info@charlesriver.com](mailto:info@charlesriver.com)

<http://www.charlesriver.com>

Original Copyright 2002, 2003 by Laxmi Publications, LTD.

O.G. Kakde. *Algorithms for Compiler Design*.

Original ISBN: 81-7008-100-6

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

02 7 6 5 4 3 2 First Edition

CHARLES RIVER MEDIA titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Special Sales Department at 781-740-0400.

## Acknowledgments

The author wishes to thank all of the colleagues in the Department of Electronics and Computer Science Engineering at Visvesvaraya Regional College of Engineering Nagpur, whose constant encouragement and timely help have resulted in the completion of this book. Special thanks go to Dr. C. S. Moghe, with whom the author had long technical discussions, which found their place in this book. Thanks are due to the institution for providing all of the infrastructural facilities and tools for a timely completion of this book. The author would particularly like to acknowledge Mr. P. S. Deshpande and Mr. A. S. Mokhade for their invaluable help and support from time to time. Finally, the author wishes to thank all of his students.

## Preface

This book on algorithms for compiler design covers the various aspects of designing a language translator in depth. The book is intended to be a basic reading material in compiler design.

Enough examples and algorithms have been used to effectively explain various tools of compiler design. The first chapter gives a brief introduction of the compiler and is thus important for the rest of the book.

Other issues like context free grammar, parsing techniques, syntax directed definitions, symbol table, code optimization and more are explained in various chapters of the book.

The final chapter has some exercises for the readers for practice.

## Chapter 1: Introduction

### 1.1 WHAT IS A COMPILER?

A compiler is a program that translates a high-level language program into a functionally equivalent low-level language program. So, a compiler is basically a translator whose source language (i.e., language to be translated) is the high-level language, and the target language is a low-level language; that is, a compiler is used to implement a high-level language on a computer.

## 1.2 WHAT IS A CROSS-COMPILER?

A cross-compiler is a compiler that runs on one machine and produces object code for another machine. The cross-compiler is used to implement the compiler, which is characterized by three languages:

1. The source language,
2. The object language, and
3. The language in which it is written.

If a compiler has been implemented in its own language, then this arrangement is called a "bootstrap" arrangement. The implementation of a compiler in its own language can be done as follows.

### Implementing a Bootstrap Compiler

Suppose we have a new language,  $L$ , that we want to make available on machines  $A$  and  $B$ . As a first step, we can write a small compiler:  $S_{CA}^A$ , which will translate an  $S$  subset of  $L$  to the object code for machine  $A$ , written in a language available on  $A$ . We then write a compiler  $S_{CS}^A$ , which is compiled in language  $L$  and generates object code written in an  $S$  subset of  $L$  for machine  $A$ . But this will not be able to execute unless and until it is translated by  $S_{CA}^A$ ; therefore,  $S_{CS}^A$  is an input into  $S_{CA}^A$ , as shown below, producing a compiler for  $L$  that will run on machine  $A$  and self-generate code for machine  $A$ :  $S_{CA}^A$ .

$$S_{CS}^A \rightarrow S_{CA}^A \rightarrow {}^L C_A^A$$

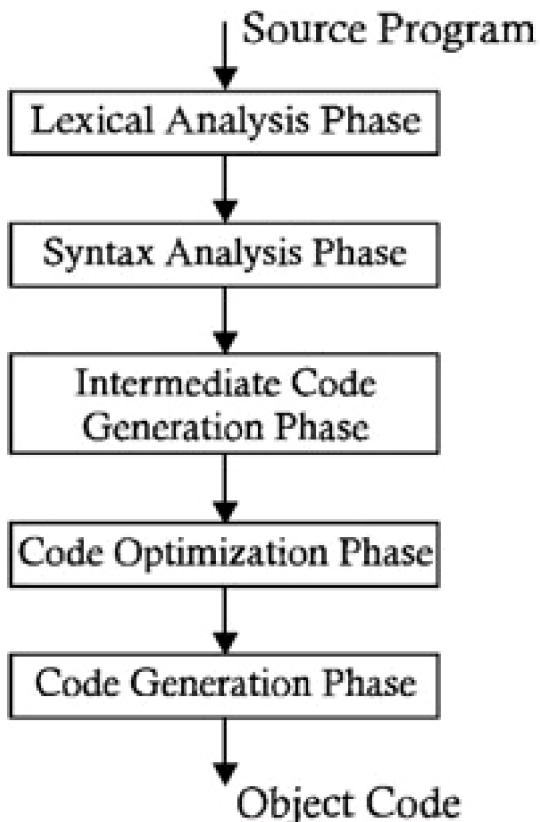
Now, if we want to produce another compiler to run on and produce code for machine  $B$ , the compiler can be written, itself, in  $L$  and made available on machine  $B$  by using the following steps:

$$\begin{array}{c} {}^L C_L^B \rightarrow {}^L C_A^A \rightarrow {}^L C_A^B \\ {}^L C_L^B \rightarrow {}^L C_A^B \rightarrow {}^L C_B^B \end{array}$$

## 1.3 COMPILATION

Compilation refers to the compiler's process of translating a high-level language program into a low-level language program. This process is very complex; hence, from the logical as well as an implementation point of view, it is customary to partition the compilation process into several phases, which are nothing more than logically cohesive operations that input one representation of a source program and output another representation.

A typical compilation, broken down into phases, is shown in [Figure 1.1](#).



**Figure 1.1:** Compilation process phases.

The initial process phases analyze the source program. The lexical analysis phase reads the characters in the source program and groups them into streams of tokens; each token represents a logically cohesive sequence of characters, such as identifiers, operators, and keywords. The character sequence that forms a token is called a "lexeme". Certain tokens are augmented by the lexical value; that is, when an identifier like xyz is found, the lexical analyzer not only returns id, but it also enters the lexeme xyz into the symbol table if it does not already exist there. It returns a pointer to this symbol table entry as a lexical value associated with this occurrence of the token id. Therefore, when internally representing a statement like X: = Y + Z, after the lexical analysis will be id<sub>1</sub>: = id<sub>2</sub> + id<sub>3</sub>.

The subscripts 1, 2, and 3 are used for convenience; the actual token is id. The syntax analysis phase imposes a hierarchical structure on the token string, as shown in [Figure 1.2](#).

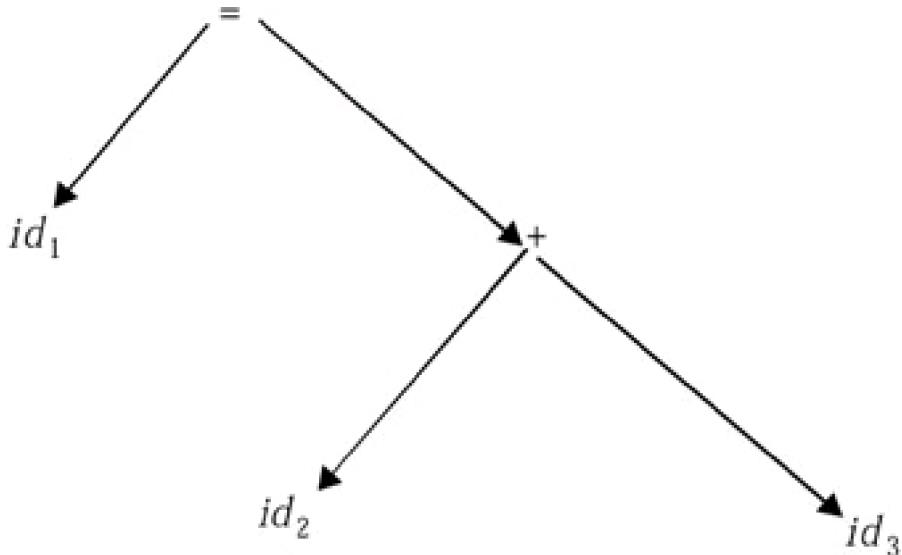


Figure 1.2: Syntax analysis imposes a structure hierarchy on the token string.

### Intermediate Code Generation

Some compilers generate an explicit intermediate code representation of the source program. The intermediate code can have a variety of forms. For example, a three-address code (TAC) representation for the tree shown in [Figure 1.2](#) will be:

$$\begin{aligned} T_1 &:= \text{id}_2 + \text{id}_3 \\ \text{id}_1 &:= T_2 \end{aligned}$$

where  $T_1$  and  $T_2$  are compiler-generated temporaries.

### Code Optimization

In the optimization phase, the compiler performs various transformations in order to improve the intermediate code. These transformations will result in faster-running machine code.

### Code Generation

The final phase in the compilation process is the generation of target code. This process involves selecting memory locations for each variable used by the program. Then, each intermediate instruction is translated into a sequence of machine instructions that performs the same task.

### Compiler Phase Organization

This is the logical organization of compiler. It reveals that certain phases of the compiler are heavily dependent on the source language and are independent of the code requirements of the target machine. All such phases, when grouped together, constitute the front end of the compiler; whereas those phases that are dependent on the target machine constitute the back end of the compiler. Grouping the compilation phases in the front and back ends facilitates the re-targeting of the code; implementation of the same source language on different machines can be done by rewriting only the back end.

**Note** Different languages can also be implemented on the same machine by rewriting the front end and using the same back end. But to do this, all of the front ends are required to produce the same intermediate code; and this is difficult, because the front end depends on the source language, and different languages are designed with different viewpoints. Therefore, it becomes difficult to write the front ends for different languages by using a common intermediate code.

Having relatively few passes is desirable from the point of view of reducing the compilation time. To reduce the number of passes, it is required to group several phases in one pass. For some of the phases, being grouped into one

pass is not a major problem. For example, the lexical analyzer and syntax analyzer can easily be grouped into one pass, because the interface between them is a single token; that is, the processing required by the token is independent of other tokens. Therefore, these phases can be easily grouped together, with the lexical analyzer working as a subroutine of the syntax analyzer, which is charge of the entire analysis activity.

Conversely, grouping some of the phases into one pass is not that easy. Grouping intermediate and object code-generation phases is difficult, because it is often very hard to perform object code generation until a sufficient number of intermediate code statements have been generated. Here, the interface between the two is not based on only one intermediate instruction—certain languages permit the use of a variable before it is declared. Similarly, many languages also permit forward jumps. Therefore, it is not possible to generate object code for a construct until sufficient intermediate code statements have been generated. To overcome this problem and enable the merging of intermediate and object code generation into one pass, the technique called "back-patching" is used; the object code is generated by leaving 'statementholes,' which will be filled later when the information becomes available.

### 1.3.1 Lexical Analysis Phase

In the lexical analysis phase, the compiler scans the characters of the source program, one character at a time. Whenever it gets a sufficient number of characters to constitute a token of the specified language, it outputs that token. In order to perform this task, the lexical analyzer must know the keywords, identifiers, operators, delimiters, and punctuation symbols of the language to be implemented. So, when it scans the source program, it will be able to return a suitable token whenever it encounters a token lexeme. (Lexeme refers to the sequence of characters in the source program that is matched by language's character patterns that specify identifiers, operators, keywords, delimiters, punctuation symbols, and so forth.) Therefore, the lexical analyzer design must:

1. Specify the token of the language, and
2. Suitably recognize the tokens.

We cannot specify the language tokens by enumerating each and every identifier, operator, keyword, delimiter, and punctuation symbol; our specification would end up spanning several pages—and perhaps never end, especially for those languages that do not limit the number of characters that an identifier can have. Therefore, token specification should be generated by specifying the rules that govern the way that the language's alphabet symbols can be combined, so that the result of the combination will be a token of that language's identifiers, operators, and keywords. This requires the use of suitable language-specific notation.

## Regular Expression Notation

Regular expression notation can be used for specification of tokens because tokens constitute a regular set. It is compact, precise, and contains a deterministic finite automata (DFA) that accepts the language specified by the regular expression. The DFA is used to recognize the language specified by the regular expression notation, making the automatic construction of recognizer of tokens possible. Therefore, the study of regular expression notation and finite automata becomes necessary. Some definitions of the various terms used are described below.

## 1.4 REGULAR EXPRESSION NOTATION/FINITE AUTOMATA DEFINITIONS

### String

A string is a finite sequence of symbols. We use a letter, such as  $w$ , to denote a string. If  $w$  is the string, then the length of string is denoted as  $|w|$ , and it is a count of number of symbols of  $w$ . For example, if  $w=xyz$ ,  $|w|=3$ . If  $|w|=0$ , then the string is called an "empty" string, and we use  $\in$  to denote the empty string.

### Prefix

A string's prefix is the string formed by taking any number of leading symbols of string. For example, if  $w=abc$ , then  $\in$ ,  $a$ ,  $ab$ , and  $abc$  are the prefixes of  $w$ . Any prefix of a string other than the string itself is called a "proper" prefix of the string.

### Suffix

A string's suffix is formed by taking any number of trailing symbols of a string. For example, if  $w=abc$ , then  $\in$ ,  $c$ ,  $bc$ , and  $abc$  are the suffixes of the  $w$ . Similar to prefixes, any suffix of a string other than the string itself is called a "proper" suffix of the string.

### Concatenation

If  $w_1$  and  $w_2$  are two strings, then the concatenation of  $w_1$  and  $w_2$  is denoted as  $w_1.w_2$ —simply, a string obtained by writing  $w_1$  followed by  $w_2$  without any space in between (i.e., a juxtaposition of  $w_1$  and  $w_2$ ). For example, if  $w_1=xyz$ , and  $w_2=abc$ , then  $w_1.w_2=xyzabc$ . If  $w$  is a string, then  $w.\in=\in$ , and  $\in.w=w$ . Therefore, we conclude that  $\in$  (empty string) is a concatenation identity.

### Alphabet

An alphabet is a finite set of symbols denoted by the symbol  $\Sigma$ .

### Language

A language is a set of strings formed by using the symbols belonging to some previously chosen alphabet. For example, if  $\Sigma = \{0, 1\}$ , then one of the languages that can be defined over this  $\Sigma$  will be  $L = \{\in, 0, 00, 000, 1, 11, 111, \dots\}$ .

### Set

A set is a collection of objects. It is denoted by the following methods:

1. We can enumerate the members by placing them within curly brackets ( $\{\}$ ). For example, the set  $A$  is defined by:  $A = \{0, 1, 2\}$ .
2. We can use a predetermined notation in which the set is denoted as:  $A = \{x | P(x)\}$ . This means that  $A$  is a set of all those elements  $x$  for which the predicate  $P(x)$  is true. For example, a set of all integers divisible by three will be denoted as:  $A = \{x | x \text{ is an integer and } x \bmod 3 = 0\}$ .

### Set Operations

- **Union:** If  $A$  and  $B$  are the two sets, then the union of  $A$  and  $B$  is denoted as:  $A \cup B = \{x | x \text{ is in } A \text{ or } x \text{ is in } B\}$ .
- **Intersection:** If  $A$  and  $B$  are the two sets, then the intersection of  $A$  and  $B$  is denoted as:  $A \cap B = \{x | x$

is in  $A$  and  $x$  is in  $B$  }.

- **Set difference:** If  $A$  and  $B$  are the two sets, then the difference of  $A$  and  $B$  is denoted as:  $A - B = \{ x \mid x \text{ is in } A \text{ but not in } B \}$ .
- **Cartesian product:** If  $A$  and  $B$  are the two sets, then the Cartesian product of  $A$  and  $B$  is denoted as:  $A \times B = \{ (a, b) \mid a \text{ is in } A \text{ and } b \text{ is in } B \}$ .
- **Power set:** If  $A$  is the set, then the power set of  $A$  is denoted as :  $2^A = P \mid P \text{ is a subset of } A \}$  (i.e., the set contains of all possible subsets of  $A$ .) For example:

$$A = \{ 0, 1 \}$$
$$2^A = \{ \emptyset, \{0\}, \{1\}, \{0, 1\} \}$$

- **Concatenation:** If  $A$  and  $B$  are the two sets, then the concatenation of  $A$  and  $B$  is denoted as:  $AB = \{ ab \mid a \text{ is in } A \text{ and } b \text{ is in } B \}$ . For example, if  $A = \{ 0, 1 \}$  and  $B = \{ 1, 2 \}$ , then  $AB = \{ 01, 02, 11, 12 \}$ .
- **Closure:** If  $A$  is a set, then closure of  $A$  is denoted as:  $A^* = A^0 \cup A^1 \cup A^2 \cup \dots \cup A^\infty$ , where  $A^i$  is the  $i$ th power of set  $A$ , defined as  $A^i = A \cdot A \cdot A \dots i \text{ times}$ .

$$A^0 = \{ \in \}$$

(i.e., the set of all possible combination of members of  $A$  of length 0)

$$A^1 = A$$

(i.e., the set of all possible combination of members of  $A$  of length 1)

$$A^2 = A \cdot A$$

(i.e., the set of all possible combinations of members of  $A$  of length 2)

Therefore  $A^*$  is the set of all possible combinations of the members of  $A$ . For example, if  $\Sigma = \{ 0, 1 \}$ , then  $\Sigma^*$  will be the set of all possible combinations of zeros and ones, which is one of the languages defined over  $\Sigma$ .

## 1.5 RELATIONS

Let  $A$  and  $B$  be the two sets; then the relationship  $R$  between  $A$  and  $B$  is nothing more than a set of ordered pairs  $(a, b)$  such that  $a$  is in  $A$  and  $b$  is in  $B$ , and  $a$  is related to  $b$  by relation  $R$ . That is:

$$R = \{ (a, b) \mid a \text{ is in } A \text{ and } b \text{ is in } B, \text{ and } a \text{ is related to } b \text{ by } R \}$$

For example, if  $A = \{ 0, 1 \}$  and  $B = \{ 1, 2 \}$ , then we can define a relation of 'less than,' denoted by  $<$  as follows:

$$< = \{ (0, 1), (0, 2), (1, 2) \}$$

A pair  $(1, 1)$  will not belong to the  $<$  relation, because one is not less than one. Therefore, we conclude that a relation  $R$  between sets  $A$  and  $B$  is the subset of  $A \times B$ .

If a pair  $(a, b)$  is in  $R$ , then  $aRb$  is true; otherwise,  $aRb$  is false.

$A$  is called a "domain" of the relation, and  $B$  is called a "range" of the relation. If the domain of a relation  $R$  is a set  $A$ , and the range is also a set  $A$ , then  $R$  is called as a relation on set  $A$  rather than calling a relation between sets  $A$  and  $B$ . For example, if  $A = \{ 0, 1, 2 \}$ , then a  $<$  relation defined on  $A$  will result in:  $< = \{ (0, 1), (0, 2), (1, 2) \}$ .

### 1.5.1 Properties of the Relation

Let  $R$  be some relation defined on a set  $A$ . Therefore:

1.  $R$  is said to be reflexive if  $aRa$  is true for every  $a$  in  $A$ ; that is, if every element of  $A$  is related with itself by relation  $R$ , then  $R$  is called as a reflexive relation.
2. If every  $aRb$  implies  $bRa$  (i.e., when  $a$  is related to  $b$  by  $R$ , and if  $b$  is also related to  $a$  by the same relation  $R$ ), then a relation  $R$  will be a symmetric relation.
3. If every  $aRb$  and  $bRc$  implies  $aRc$ , then the relation  $R$  is said to be transitive; that is, when  $a$  is related to  $b$  by  $R$ , and  $b$  is related to  $c$  by  $R$ , and if  $a$  is also related to  $c$  by relation  $R$ , then  $R$  is a transitive relation.

If  $R$  is reflexive and transitive, as well as symmetric, then  $R$  is an equivalence relation.

### Property Closure of a Relation

Let  $R$  be a relation defined on a set  $A$ , and if  $P$  is a set of properties, then the property closure of a relation  $R$ , denoted as  $P$ -closure, is the smallest relation,  $R'$ , which has the properties mentioned in  $P$ . It is obtained by adding every pair  $(a, b)$  in  $R$  to  $R'$ , and then adding those pairs of the members of  $A$  that will make relation  $R'$  have the properties in  $P$ . If  $P$  contains only transitivity properties, then the  $P$ -closure will be called as a transitive closure of the relation, and we denote the transitive closure of relation  $R$  by  $R^+$ ; whereas when  $P$  contains transitive as well as reflexive properties, then the  $P$ -closure is called as a reflexive-transitive closure of relation  $R$ , and we denote it by  $R^*$ .  $R^+$  can be obtained from  $R$  as follows:

$$R_{\text{old}}^+ = \Phi$$

$$R_{\text{new}}^+ = R$$

$$\text{While } (R_{\text{old}}^+ \neq R_{\text{new}}^+)$$

{

$$R_{\text{old}}^+ = R_{\text{new}}^+$$

for (every pair  $(a, b)$  and  $(b, c)$  in  $R^+ \dots$ ) do

add pair  $(a, c)$  to  $R^+_{\text{new}}$  if not already present  
}

$$R^+ = R^+_{\text{new}}$$

For example, if:

$$\begin{aligned} R &= \{ (0, 1), (1, 2), (3, 4) \} \text{ then} \\ R^+ &= \{ (0, 1), (1, 2), (3, 4), (0, 2) \} \\ R^* &= \{ (0, 1), (1, 2), (3, 4), (0, 2), (0, 0), \\ &\quad (1, 1), (2, 2), (3, 3), (4, 4) \} \\ R^* &= R^+ \cup \{ (a, a) \mid \text{for every } a \text{ in } A \} \end{aligned}$$

## Chapter 2: Finite Automata and Regular Expressions

### 2.1 FINITE AUTOMATA

A finite automata consists of a finite number of states and a finite number of transitions, and these transitions are defined on certain, specific symbols called input symbols. One of the states of the finite automata is identified as the initial state the state in which the automata always starts. Similarly, certain states are identified as final states.

Therefore, a finite automata is specified as using five things:

1. The states of the finite automata;
2. The input symbols on which transitions are made;
3. The transitions specifying from which state on which input symbol where the transition goes;
4. The initial state; and
5. The set of final states.

Therefore formally a finite automata is a five-tuple:

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

where:

- $Q$  is a set of states of the finite automata,
- $\Sigma$  is a set of input symbols, and
- $\delta$  specifies the transitions in the automata.

If from a state  $p$  there exists a transition going to state  $q$  on an input symbol  $a$ , then we write  $\delta(p, a) = q$ . Hence,  $\delta$  is a function whose domain is a set of ordered pairs,  $(p, a)$ , where  $p$  is a state and  $a$  is an input symbol, and the range is a set of states.

Therefore, we conclude that  $\delta$  defines a mapping whose domain will be a set of ordered pairs of the form  $(p, a)$  and whose range will be a set of states. That is,  $\delta$  defines a mapping from

$$Q \times \Sigma \text{ to } Q,$$

$q_0$  is the initial state, and  $F$  is a set of final states of the automata. For example:

$$M = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

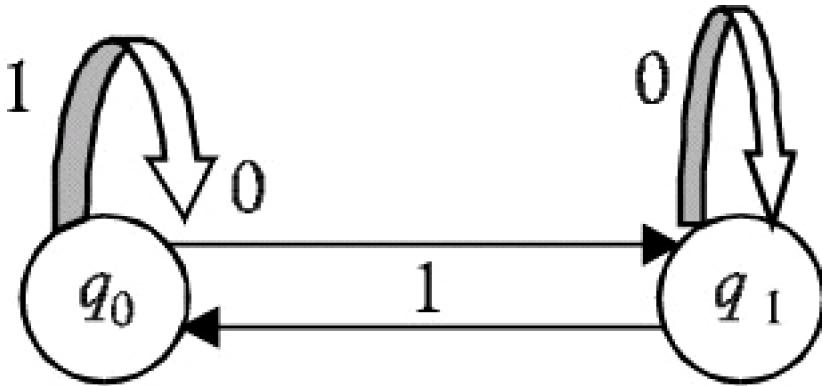
where

$$\begin{aligned}\delta(q_0, 0) &= q_1, \delta(q_0, 1) = q_0 \\ \delta(q_1, 0) &= q_1, \delta(q_1, 1) = q_0\end{aligned}$$

A directed graph exists that can be associated with finite automata. This

graph is called a "transition diagram of finite automata." To associate a graph with finite automata, the vertices of the graph correspond to the states of the automata, and the edges in the transition diagram are determined as follows.

If  $\delta(p, a) = q$ , then put an edge from the vertex, which corresponds to state  $p$ , to the vertex that corresponds to state  $q$ , labeled by  $a$ . To indicate the initial state, we place an arrow with its head pointing to the vertex that corresponds to the initial state of the automata, and we label that arrow "start." We then encircle the vertices twice, which correspond to the final states of the automata. Therefore, the transition diagram for the described finite automata will resemble [Figure 2.1](#).



**Figure 2.1:** Transition diagram for finite automata  $\delta(p, a) = q$ .

A tabular representation can also be used to specify the finite automata. A table whose number of rows is equal to the number of states, and whose number of columns equals the number of input symbols, is used to specify the transitions in the automata. The first row specifies the transitions from the initial state; the rows specifying the transitions from the final states are marked as \*. For example, the automata above can be specified as follows:

	0	1
$q_0$	$q_1$	$q_0$
$*q_1$	$q_1$	$q_0$

A finite automata can be used to accept some particular set of strings. If  $x$  is a string made of symbols belonging to  $\Sigma$  of the finite automata, then  $x$  is accepted by the finite automata if a path corresponding to  $x$  in a finite automata starts in an initial state and ends in one of the final states of the automata; that is, there must exist a sequence of moves for  $x$  in the finite automata that takes the transitions from the initial state to one of the final states of the automata. Since  $x$  is a member of  $\Sigma^*$ , we define a new transition function,  $\delta_1$ , which defines a mapping from  $Q \times \Sigma^*$  to  $Q$ . And if  $\delta_1(q_0, x) = a$  member of  $F$ , then  $x$  is accepted by the finite automata. If  $x$  is written as  $wa$ , where  $a$  is the last symbol of  $x$ , and  $w$  is a string of the remaining symbols of  $x$ , then:

$$\begin{aligned}\delta_1(q_0, x) &= \delta(\delta_1(q_0, w), a), \text{ Since } \delta_1 \text{ defines a mapping from } Q \times \Sigma^* \text{ to } Q \\ \delta_1(q_0, a) &= \delta(q_0, a)\end{aligned}$$

For example:

$$M = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

where

$$\begin{aligned}\delta(q_0, 0) &= q_1, \delta(q_0, 0) = q_0 \\ \delta(q_1, 0) &= q_1, \delta(q_1, 1) = q_0\end{aligned}$$

Let  $x$  be 010. To find out if  $x$  is accepted by the automata or not, we proceed as follows:

$$\delta_1(q_0, 0) = \delta(q_0, 0) = q_1$$

$$\text{Therefore, } \delta_1(q_0, 01) = \delta(\delta_1(q_0, 0), 1) = q_0$$

Therefore,  $\delta_1(q_0, 010) = \delta\{\delta_1(q_0, 0), 1\} = q_1$

Since  $q_1$  is a member of  $F$ ,  $x = 010$  is accepted by the automata.

If  $x = 0101$ , then  $\delta_1(q_0, 0101) = \delta\{\delta_1(q_0, 010), 1\} = q_0$

Since  $q_0$  is not a member of  $F$ ,  $x$  is not accepted by the above automata.

Therefore, if  $M$  is the finite automata, then the language accepted by the finite automata is denoted as  $L(M) = \{x \mid \delta_1(q_0, x) = \text{member of } F\}$ .

In the finite automata discussed above, since  $\delta$  defines mapping from  $Q \times \Sigma$  to  $Q$ , there exists exactly one transition from a state on an input symbol; and therefore, this finite automata is considered a deterministic finite automata (DFA).

Therefore, we define the DFA as the finite automata:

$M = (Q, \Sigma, \delta, q_0, F)$ , such that there exists exactly one transition from a state on a input symbol.

## 2.2 NON-DETERMINISTIC FINITE AUTOMATA

If the basic finite automata model is modified in such a way that from a state on an input symbol zero, one or more transitions are permitted, then the corresponding finite automata is called a "non-deterministic finite automata" (NFA). Therefore, an NFA is a finite automata in which there may exist more than one paths corresponding to  $x$  in  $\Sigma^*$  (because zero, one, or more transitions are permitted from a state on an input symbol). Whereas in a DFA, there exists exactly one path corresponding to  $x$  in  $\Sigma^*$ . Hence, an NFA is nothing more than a finite automata:

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

in which  $\delta$  defines mapping from  $Q \times \Sigma$  to  $2^Q$  (to take care of zero, one, or more transitions). For example, consider the finite automata shown below:

$$M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_3\})$$

where:

$$\begin{aligned}\delta(q_0, 0) &= \{q_1\}, & \delta(q_0, 1) &= \Phi \\ \delta(q_1, 0) &= \{q_1\}, & \delta(q_1, 1) &= \{q_1, q_2\} \\ \delta(q_2, 0) &= \Phi, & \delta(q_2, 1) &= \{q_3\} \\ \delta(q_3, 0) &= \{q_3\}, & \delta(q_3, 1) &= \{q_3\}\end{aligned}$$

The transition diagram of this automata is:

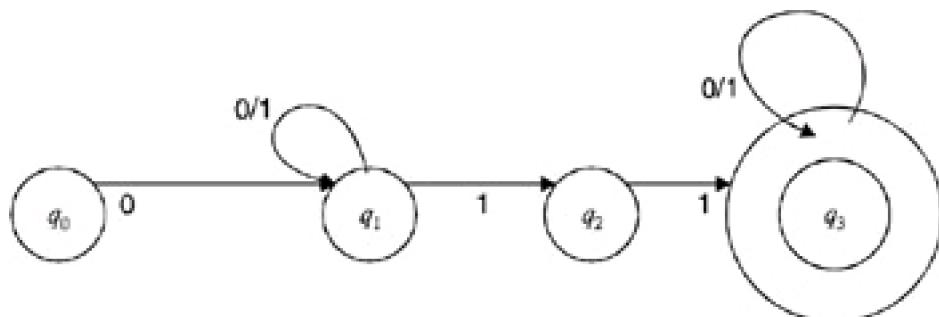


Figure 2.2: Transition diagram for finite automata that handles several transitions.

### 2.2.1 Acceptance of Strings by Non-deterministic Finite Automata

Since an NFA is a finite automata in which there may exist more than one path corresponding to  $x$  in  $\Sigma^*$ , and if this is, indeed, the case, then we are required to test the multiple paths corresponding to  $x$  in order to decide whether or not  $x$  is accepted by the NFA, because, for the NFA to accept  $x$ , at least one path corresponding to  $x$  is required in the NFA. This path should start in the initial state and end in one of the final states. Whereas in a DFA, since there exists exactly one path corresponding to  $x$  in  $\Sigma^*$ , it is enough to test whether or not that path starts in the initial state and ends in one of the final states in order to decide whether  $x$  is accepted by the DFA or not.

Therefore, if  $x$  is a string made of symbols in  $\Sigma$  of the NFA (i.e.,  $x$  is in  $\Sigma^*$ ), then  $x$  is accepted by the NFA if at least one path exists that corresponds to  $x$  in the NFA, which starts in an initial state and ends in one of the final states of the NFA. Since  $x$  is a member of  $\Sigma^*$  and there may exist zero, one, or more transitions from a state on an input symbol, we define a new transition function,  $\delta_1$ , which defines a mapping from  $2^Q \times \Sigma^*$  to  $2^Q$ ; and if  $\delta_1(\{q_0\}, x) = P$ , where  $P$  is a set

containing at least one member of  $F$ , then  $x$  is accepted by the NFA. If  $x$  is written as  $wa$ , where  $a$  is the last symbol of  $x$ , and  $w$  is a string made of the remaining symbols of  $x$  then:

$$\delta_1(\{q_0\}, x) = \delta_1(\delta_1(\{q_0\}, w), a) \text{ since } \delta_1 \text{ defines a mapping from } 2^Q \times \Sigma^* \text{ to } 2^Q$$

$$\delta_1(p, a) = \cup_{\text{for every } q \text{ in } p} \delta(q, a)$$

For example, consider the finite automata shown below:

$$M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_3\})$$

where:

$$\begin{aligned}\delta(q_0, 0) &= \{q_1\}, & \delta(q_0, 1) &= \Phi \\ \delta(q_1, 0) &= \{q_1\}, & \delta(q_1, 1) &= \{q_1, q_2\} \\ \delta(q_2, 0) &= \Phi, & \delta(q_2, 1) &= \{q_3\} \\ \delta(q_3, 0) &= \{q_3\}, & \delta(q_3, 1) &= \{q_3\}\end{aligned}$$

If  $x = 0111$ , then to find out whether or not  $x$  is accepted by the NFA, we proceed as follows:

$$\begin{aligned}\delta_1(\{q_0\}, 0) &= \delta(q_0, 0) = \{q_1\} \\ \text{Therefore } \delta_1(\{q_0\}, 01) &= \delta_1(\delta_1(\{q_0\}, 0), 1) \\ &= \delta_1(\{q_1\}, 1) = \delta(q_1, 1) \\ &= \{q_1, q_2\} \\ \text{Therefore } \delta_1(\{q_0\}, 011) &= \delta_1(\delta_1(\{q_0\}, 01), 1) \\ &= \delta_1(\{q_1, q_2\}, 1) \\ &= \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= \{q_1, q_2\} \cup \{q_3\} \\ &= \{q_1, q_2, q_3\} \\ \text{Therefore } \delta_1(\{q_0\}, 0111) &= \delta_1(\delta_1(\{q_0\}, 011), 1) \\ &= \delta_1(\{q_1, q_2, q_3\}, 1) \\ &= \delta(q_1, 1) \cup \delta(q_2, 1) \cup \delta(q_3, 1) \\ &= \{q_1, q_2\} \cup \{q_3\} \cup \{q_3\} \\ &= \{q_1, q_2, q_3\}\end{aligned}$$

Since  $\delta_1(\{q_0\}, 0111) = \{q_1, q_2, q_3\}$ , which contains  $q_3$ , a member of  $F$  of the NFA—, hence,  $x = 0111$  is accepted by the NFA.

Therefore, if  $M$  is a NFA, then the language accepted by NFA is defined as:

$$L(M) = \{x \mid \delta_1(\{q_0\}, x) = P, \text{ where } P \text{ contains at least one member of } F\}.$$

## 2.3 TRANSFORMING NFA TO DFA

For every non-deterministic finite automata, there exists an equivalent deterministic finite automata. The equivalence between the two is defined in terms of language acceptance. Since an NFA is nothing more than a finite automata in which zero, one, or more transitions on an input symbol is permitted, we can always construct a finite automata that will simulate all the moves of the NFA on a particular input symbol in parallel. We then get a finite automata in which there will be exactly one transition on an input symbol; hence, it will be a DFA equivalent to the NFA.

Since the DFA equivalent of the NFA simulates (parallels) the moves of the NFA, every state of a DFA will be a combination of one or more states of the NFA. Hence, every state of a DFA will be represented by some subset of the set of states of the NFA; and therefore, the transformation from NFA to DFA is normally called the "construction" subset. Therefore, if a given NFA has  $n$  states, then the equivalent DFA will have  $2^n$  number of states, with the initial state corresponding to the subset  $\{q_0\}$ . Therefore, the transformation from NFA to DFA involves finding all possible subsets of the set of states of the NFA, considering each subset to be a state of a DFA, and then finding the transition from it on every input symbol. But all the states of a DFA obtained in this way might not be reachable from the initial state; and if a state is not reachable from the initial state on any possible input sequence, then such a state does not play role in deciding what language is accepted by the DFA. (Such states are those states of the DFA that have outgoing transitions on the input symbols—but either no incoming transitions, or they only have incoming transitions from other unreachable states.) Hence, the amount of work involved in transforming an NFA to a DFA can be reduced if we attempt to generate only reachable states of a DFA. This can be done by proceeding as follows:

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be an NFA to be transformed into a DFA.

Let  $Q_1$  be the set states of equivalent DFA.

begin:

```

Q1old = Φ
Q1new = {q0}

While (Q1old ≠ Q1new)
{
    Temp = Q1new - Q1old
    Q1 = Q1new
    for every subset P in Temp do
        for every a in Σdo
            If transition from P on a goes to new subset S of Q
            then
                (transition from P on a is obtained by finding out
                the transitions from every member of P on a in a given

```

```

NFA
and then taking the union of all such transitions)
```

```

Q1 new = Q1 new ∪ S
}
Q1 = Q1new
end
```

A subset  $P$  in  $Q_1$  will be a final state of the DFA if  $P$  contains at least one member of  $F$  of the NFA. For example, consider the following finite automata:

$$M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_3\})$$

where:

$$\begin{aligned}
 \sigma(q_0, 0) &= \{q_1\}, & \sigma(q_0, 1) &= \emptyset \\
 \delta(q_1, 0) &= \{q_1\}, & \delta(q_1, 1) &= \{q_1, q_2\} \\
 \delta(q_2, 0) &= \Phi, & \delta(q_2, 1) &= \{q_3\} \\
 \delta(q_3, 0) &= \{q_3\}, & \delta(q_3, 1) &= \{q_3\}
 \end{aligned}$$

The DFA equivalent of this NFA can be obtained as follows:

	0	1
$\{q_0\}$	$\{q_1\}$	$\Phi$
$\{q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_1\}$	$\{q_1, q_2, q_3\}$
$\ast\{q_1, q_2, q_3\}$	$\{q_1, q_3\}$	$\{q_1, q_2, q_3\}$
$\ast\{q_1, q_3\}$	$\{q_1, q_3\}$	$\{q_1, q_2, q_3\}$
$\Phi$	$\Phi$	$\Phi$

The transition diagram associated with this DFA is shown in [Figure 2.3](#).

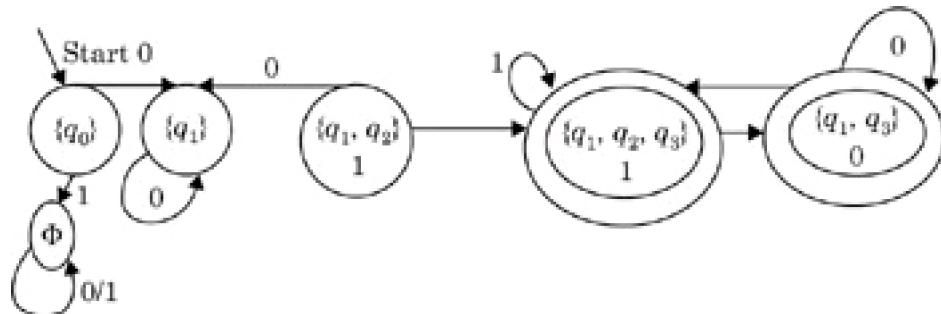
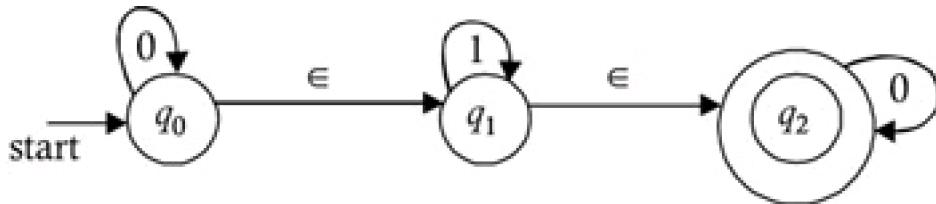


Figure 2.3: Transition diagram for  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_3\})$ .

## 2.4 THE NFA WITH $\epsilon$ -MOVES

If a finite automata is modified to permit transitions without input symbols, along with zero, one, or more transitions on the input symbols, then we get an NFA with ' $\epsilon$ -moves,' because the *transitions* made without symbols are called " $\epsilon$ -transitions."

Consider the NFA shown in [Figure 2.4](#).



**Figure 2.4:** Finite automata with  $\epsilon$ -moves.

This is an NFA with  $\epsilon$ -moves because it is possible to transition from state  $q_0$  to  $q_1$  without consuming any of the input symbols. Similarly, we can also transition from state  $q_1$  to  $q_2$  without consuming any input symbols. Since it is a finite automata, an NFA with  $\epsilon$ -moves will also be denoted as a five-tuple:

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

where  $Q$ ,  $\Sigma$ ,  $q_0$ , and  $F$  have the usual meanings, and  $\delta$  defines a mapping from

$$Q \times (\Sigma \cup \epsilon) \text{ to } 2^Q$$

(to take care of the  $\epsilon$ -transitions as well as the non  $\epsilon$ -transitions).

### Acceptance of a String by the NFA with $\epsilon$ -Moves

A string  $x$  in  $\Sigma^*$  will  $\epsilon$ -moves will be accepted by the NFA, if at least one path exists that corresponds to  $x$  starts in an initial state and ends in one of the final states. But since this path may be formed by  $\epsilon$ -transitions as well as non- $\epsilon$ -transitions, to find out whether  $x$  is accepted or not by the NFA with  $\epsilon$ -moves, we must define a function,  $\epsilon$ -closure( $q$ ), where  $q$  is a state of the automata.

The function  $\epsilon$ -closure( $q$ ) is defined follows:

$\epsilon$ -closure( $q$ )= set of all those states of the automata that can be reached from  $q$  on a path labeled by  $\epsilon$ .

For example, in the NFA with  $\epsilon$ -moves given above:

$$\epsilon\text{-closure}(q_0) = \{ q_0, q_1, q_2 \}$$

$$\epsilon\text{-closure}(q_1) = \{ q_1, q_2 \}$$

$$\epsilon\text{-closure}(q_2) = \{ q_2 \}$$

The function

$\epsilon$ -closure ( $q$ ) will never be an empty set, because  $q$  is always reachable from itself, without dependence on any input symbol; that is, on a path labeled by  $\epsilon$ ,  $q$  will always exist in  $\epsilon$ -closure( $q$ ) on that labeled path.

If  $P$  is a set of states, then the  $\epsilon$ -closure function can be extended to find  $\epsilon$ -closure( $P$ ), as follows:

$$\epsilon\text{-closure}(P) = \cup_{\text{for every } q \text{ in } P} \epsilon\text{-closure}(q)$$

#### 2.4.1 Algorithm for Finding $\epsilon$ -Closure(q)

Let  $T$  be the set that will comprise  $\epsilon$ -closure( $q$ ). We begin by adding  $q$  to  $T$ , and then initialize the stack by pushing  $q$  onto stack:

```

while (stack not empty) do
{
    p = pop (stack)
    R = δ(p, ε)
    for every member of R do
        if it is not present in T then
        {
            add that member to T
            push member of R on stack
        }
}

```

Since  $x$  is a member of  $\Sigma^*$ , and there may exist zero, one, or more transitions from a state on an input symbol, we define a new transition function,  $\delta_1$ , which defines a mapping from  $2^Q \times \Sigma^*$  to  $2^Q$ . If  $x$  is written as  $wa$ , where  $a$  is the last symbol of  $x$  and  $w$  is a string made of remaining symbols of  $x$  then:

$$\delta_1(\{q_0\}, x) = \delta_1(\delta_1(\{q_0\}, w), a)$$

since  $\delta_1$  defines a mapping from  $2^Q \times \Sigma^*$  to  $2^Q$ .

$$\epsilon\text{-closure}(\delta_1(\epsilon\text{-closure}(q_0), x)) = P$$

such that  $P$  contains at least one member of  $F$  and:

$$\begin{aligned} \epsilon\text{-closure}(\delta_1(\epsilon\text{-closure}(q_0), x)) \\ = \epsilon\text{-closure}(\epsilon\text{-closure}(\delta_1(\epsilon\text{-closure}(q_0), w)), a) \end{aligned}$$

For example, in the NFA with  $\epsilon$ -moves, given above, if  $x = 01$ , then to find out whether  $x$  is accepted by the automata or not, we proceed as follows:

$$\begin{aligned} \delta_1(\epsilon\text{-closure}(q_0), 0) &= \delta_1(\{q_0, q_1, q_2\}, 0) \\ &= \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) \\ &= \{q_0\} \cup \emptyset \cup \{q_2\} = \{q_0, q_2\} \\ \delta_1(\epsilon\text{-closure}(q_0), 01) &= \delta_1(\epsilon\text{-closure}(\delta_1(\epsilon\text{-closure}(q_0), 0)), 1) \\ &= \delta_1(\epsilon\text{-closure}(\{q_0, q_2\}), 1) \\ &= \delta_1(\{q_0, q_1, q_2\}), 1 \\ &= \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= \emptyset \cup \{q_1\} \cup \emptyset \\ &= \{q_1\} \end{aligned}$$

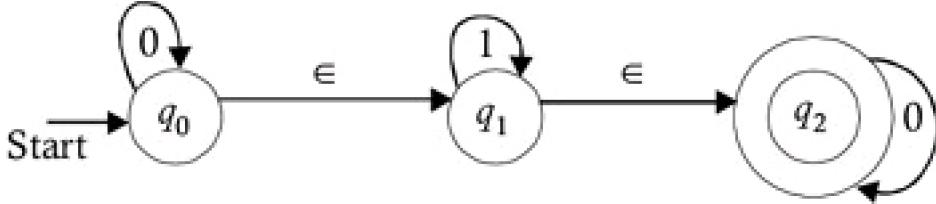
Therefore:

$\epsilon\text{-closure}(\delta_1(\epsilon\text{-closure}(q_0), 01)) = \epsilon\text{-closure}(\{q_1\}) = \{q_1, q_2\}$  Since  $q_2$  is a final state,  $x = 01$  is accepted by the automata.

#### Equivalence of NFA with $\epsilon$ -Moves to NFA Without $\epsilon$ -Moves

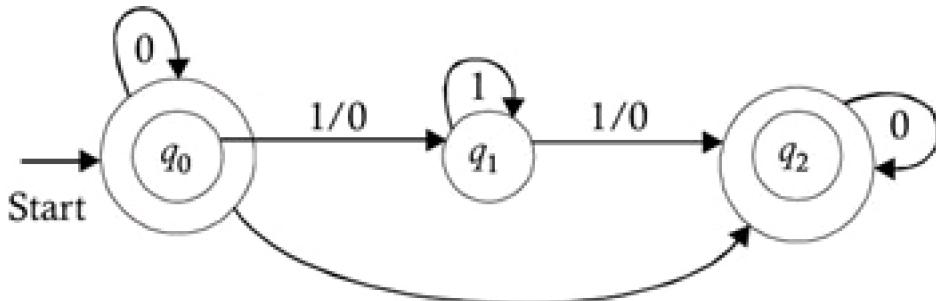
For every NFA with  $\epsilon$ -moves, there exists an equivalent NFA without  $\epsilon$ -moves that accepts the same language. To obtain an equivalent NFA without  $\epsilon$ -moves, given an NFA with  $\epsilon$ -moves, what is required is an elimination of  $\epsilon$ -transitions from a given automata. But simply eliminating the  $\epsilon$ -transitions from a given NFA with  $\epsilon$ -moves will change the language accepted by the automata. Hence, for every  $\epsilon$ -transition to be eliminated, we have to add some non- $\epsilon$ -transitions as substitutes in order to maintain the language's acceptance by the automata. Therefore, transforming an NFA with  $\epsilon$ -moves to an NFA without  $\epsilon$ -moves involves finding the non- $\epsilon$ -transitions that must be added to the automata for every  $\epsilon$ -transition to be eliminated.

Consider the NFA with  $\epsilon$ -moves shown in [Figure 2.5](#).



**Figure 2.5:** Transitioning from an  $\epsilon$ -move NFA to a non- $\epsilon$ -move NFA.

There are  $\epsilon$ -transitions from state  $q_0$  to  $q_1$  and from state  $q_1$  to  $q_2$ . To eliminate these  $\epsilon$ -transitions, we must add a transition on 0 from  $q_0$  to  $q_1$ , as well as from  $q_0$  to  $q_2$ . Similarly, a transition must be added on 1 from  $q_0$  to  $q_1$ , as well as from  $q_0$  to  $q_2$ , because the presence of these  $\epsilon$ -transitions in a given automata makes it possible to reach from  $q_0$  to  $q_1$  on consuming only 0, and it is possible to reach from  $q_0$  to  $q_2$  on consuming only 0. Similarly, it is possible to reach from  $q_0$  to  $q_1$  on consuming only 1, and it is possible to reach from  $q_0$  to  $q_2$  on consuming only 1. It is also possible to reach from  $q_1$  to  $q_2$  on consuming 0 as well as 1; and therefore, a transition from  $q_1$  to  $q_2$  on 0 and 1 is also required to be added. Since  $\epsilon$  is also accepted by the given NFA  $\epsilon$ -moves, to accept  $\epsilon$ , and initial state of the NFA without  $\epsilon$ -moves is required to be marked as one of the final states. Therefore, by adding these non- $\epsilon$ -transitions, and by making the initial state one of the final states, we get the automata shown in [Figure 2.6](#).



**Figure 2.6:** Making the initial state of the NFA one of the final states.

Therefore, when transforming an NFA with  $\epsilon$ -moves into an NFA without  $\epsilon$ -moves, only the transitions are required to be changed; the states are not required to be changed. But if a given NFA with  $q_0$  and  $\epsilon$ -moves accepts  $\epsilon$  (i.e., if the  $\epsilon$ -closure ( $q_0$ ) contains a member of  $F$ ), then  $q_0$  is also required to be marked as one of the final states if it is not already a member of  $F$ . Hence:

If  $M = (Q, \Sigma, \delta, q_0, F)$  is an NFA with  $\epsilon$ -moves, then its equivalent NFA without  $\epsilon$ -moves will be  $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$

where  $\delta_1(q, a) = \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q), a))$

and

$$F_1 = F \cup (q_0) \text{ if } \epsilon\text{-closure}(q_0) \text{ contains a member of } F$$

$$F_1 = F \text{ otherwise}$$

For example, consider the following NFA with  $\epsilon$ -moves:

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

where

$\delta$	0	1	$\epsilon$
$q_0$	$\{q_0\}$	$\emptyset$	$\{q_1\}$
$q_1$	$\emptyset$	$\{q_1\}$	$\{q_2\}$
$q_2$	$\emptyset$	$\{q_2\}$	$\emptyset$

Its equivalent NFA without  $\epsilon$ -moves will be:

$$M_1 = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_1, q_0, \{q_0, q_2\})$$

where

$\delta_1$	0	1
$q_0$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$q_1$	$\emptyset$	$\{q_1, q_2\}$
$q_2$	$\emptyset$	$\{q_2\}$

Since there exists a DFA for every NFA without  $\epsilon$ -moves, and for every NFA with  $\epsilon$ -moves there exists an equivalent NFA without  $\epsilon$ -moves, we conclude that for every NFA with  $\epsilon$ -moves there exists a DFA.

## 2.5 THE NFA WITH $\epsilon$ -MOVES TO THE DFA

There always exists a DFA equivalent to an NFA with  $\epsilon$ -moves which can be obtained as follows:

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be an NFA with  $\epsilon$ -moves.

A DFA equivalent to this NFA will be:

$M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ , where  $Q_1$  is a subset of  $2^Q$ ; that is, every state of a DFA corresponds to a subset of  $Q$ .

$q_1 = \epsilon\text{-closure}(q_0)$ , and it is the initial state of the DFA. We initially add  $q_1$  to  $Q_1$ , and then we find the transition from  $q_1$  as follows:

$$\delta_1(q_1, a) = \epsilon\text{-closure}(\delta(\text{subset representation of } q_1, a))$$

If this transition generates a new subset of  $Q$ , then it will be added to  $Q_1$ ; and next time transitions from it are found, we continue in this way until we cannot add any new states to  $Q_1$ . After this, we identify those states of the DFA whose subset representations contain at least one member of  $F$ . If  $\epsilon\text{-closure}(q_0)$  does not contain a member of  $F$ , and the set of such states of DFA constitute  $F_1$ , but if  $\epsilon\text{-closure}(q_0)$  contains a member of  $F$ , then we identify those members of  $Q_1$  whose subset representations contain at least one member of  $F$ , or  $q_0$  and  $F_1$  will be set as a member of these states.

Consider the following NFA with  $\epsilon$ -moves:

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

where

$\delta$	0	1	$\epsilon$
$q_0$	$\{q_0\}$	$\emptyset$	$\{q_1\}$
$q_1$	$\emptyset$	$\{q_1\}$	$\{q_2\}$
$q_2$	$\emptyset$	$\{q_2\}$	$\emptyset$

A DFA equivalent to this will be:

$$M_1 = (\{\{q_0, q_1, q_2\}, \{q_1, q_2\}, \emptyset\}, \{0, 1\}, \delta_1, \{q_0, q_1, q_2\}, \{\{q_0, q_1, q_2\}, \{q_1, q_2\}\})$$

where

$\delta_1$	0	1
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$

If we identify the subsets  $\{q_0, q_1, q_2\}$ ,  $\{q_0, q_1, q_2\}$  and  $\emptyset$  as  $A$ ,  $B$ , and  $C$ , respectively, then the automata will be:

$$M_1 = (\{A, B, C\}, \{0,1\}, \delta_1, A, \{A, B\})$$

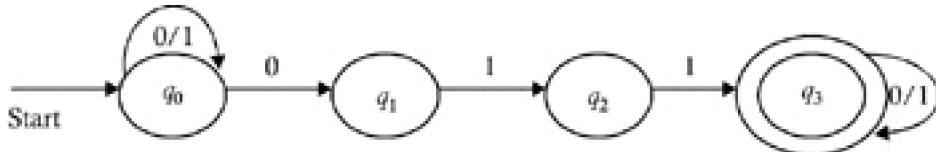
where

$\delta_1$	0	1
$A$	$A$	$B$
$B$	$C$	$B$
$C$	$C$	$C$

#### EXAMPLE 2.1

---

Obtain a DFA equivalent to the NFA shown in [Figure 2.7](#).



**Figure 2.7:** [Example 2.1](#) NFA.

---

A DFA equivalent to NFA in [Figure 2.7](#) will be:

	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_2, q_3\}^*$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$
$\{q_0, q_1, q_3\}^*$	$\{q_0, q_3\}$	$\{q_0, q_2, q_3\}$
$\{q_0, q_3\}^*$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$

where  $\{q_0\}$  corresponds to the initial state of the automata, and the states marked as \* are final states. If we rename the states as follows:

$\{q_0\}$	$A$
$\{q_0, q_1\}$	$B$
$\{q_0, q_2\}$	$C$
$\{q_0, q_2, q_3\}$	$D$
$\{q_0, q_1, q_3\}$	$E$
$\{q_0, q_3\}$	$F$

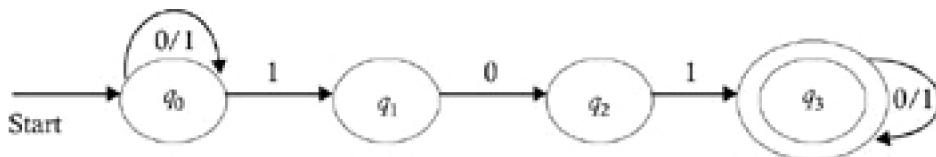
then the transition table will be:

	0	1
A	B	A
B	B	C
C	B	F
D*	E	F
E*	F	D
F*	E	F

### EXAMPLE 2.2

---

Obtain a DFA equivalent to the NFA illustrated in [Figure 2.8](#).



[Figure 2.8: Example 2.2](#) DFA equivalent to an NFA.

---

A DFA equivalent to the NFA shown in [Figure 2.8](#) will be:

	0	1
{q0}	{q0}	{q0, q1}
{q0, q1}	{q0, q2}	{q0, q1}
{q0, q2}	{q0}	{q0, q1, q3}
{q0, q1, q3}*	{q0, q2, q3}	{q0, q1, q3}
{q0, q2, q3}*	{q0, q3}	{q0, q1, q3}
{q0, q3}*	{q0, q3}	{q0, q1, q3}

where {q0} corresponds to the initial state of the automata, and the states marked as \* are final states. If we rename the states as follows:

{q0}	A
{q0, q1}	B
{q0, q2}	C
{q0, q2, q3}	D
{q0, q1, q3}	E
{q0, q3}	F

then the transition table will be:

	0	1
--	---	---

A	A	B
B	C	B
C	A	E
D*	F	E
E*	D	E
F*	F	E

**Team LiB**

PREVIOUS NEXT

## 2.6 MINIMIZATION/OPTIMIZATION OF A DFA

Minimization/optimization of a deterministic finite automata refers to detecting those states of a DFA whose presence or absence in a DFA does not affect the language accepted by the automata. Hence, these states can be eliminated from the automata without affecting the language accepted by the automata. Such states are:

- **Unreachable States:** Unreachable states of a DFA are not reachable from the initial state of DFA on any possible input sequence.
- **Dead States:** A dead state is a nonfinal state of a DFA whose transitions on every input symbol terminates on itself. For example,  $q$  is a dead state if  $q$  is in  $Q \setminus F$ , and  $\delta(q, a) = q$  for every  $a$  in  $\Sigma$ .
- **Nondistinguishable States:** Nondistinguishable states are those states of a DFA for which there exist no distinguishing strings; hence, they cannot be distinguished from one another.

Therefore, optimization entails:

1. Detection of unreachable states and eliminating them from DFA;
2. Identification of nondistinguishable states, and merging them together; and
3. Detecting dead states and eliminating them from the DFA.

### 2.6.1 Algorithm to Detect Unreachable States

Input  $M = (Q, \Sigma, \delta, q_0, F)$   
 Output = Set  $U$  (which is set of unreachable states)

{Let  $R$  be the set of reachable states of DFA. We take two  $R$ s,  $R_{\text{new}}$ , and  $R_{\text{old}}$  so that we will be able to perform iterations in the process of detecting unreachable states.}

```

begin
   $R_{\text{old}} = \emptyset$ 
   $R_{\text{new}} = \{q_0\}$ 
  while ( $R_{\text{old}} \neq R_{\text{new}}$ ) do
    begin
      temp1 =  $R_{\text{new}} - R_{\text{old}}$ 
       $R_{\text{old}} = R_{\text{new}}$ 
      temp2 =  $\emptyset$ 
      for every  $a$  in  $\Sigma$  do
        temp2 = temp2  $\cup$   $\delta(\text{temp1}, a)$ 
       $R_{\text{new}} = R_{\text{new}} \cup \text{temp2}$ 
    end
   $U = Q - R_{\text{new}}$ 
end
  
```

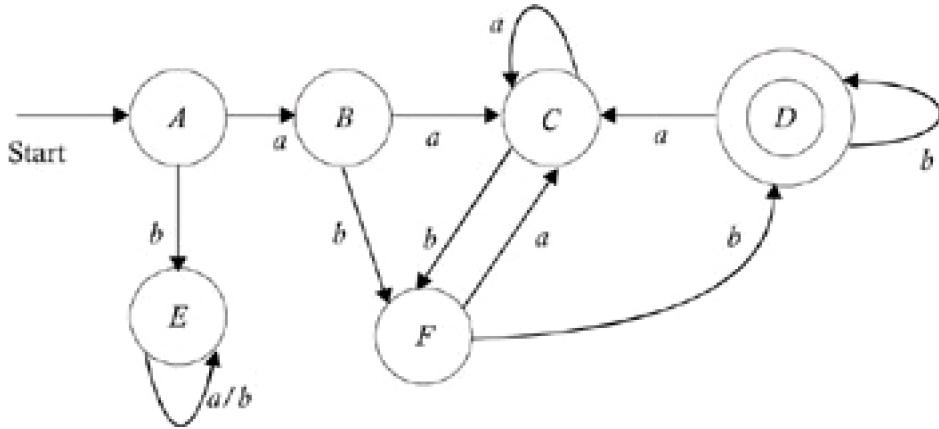
If  $p$  and  $q$  are the two states of a DFA, then  $p$  and  $q$  are said to be 'distinguishable' states if a distinguishing string  $w$  exists that distinguishes  $p$  and  $q$ .

A string  $w$  is a distinguishing string for states  $p$  and  $q$  if transitions from  $p$  on  $w$  go to a nonfinal state, whereas transitions from  $q$  on  $w$  go to a final state, or vice versa.

Therefore, to find nondistinguishable states of a DFA, we must find out whether some distinguishing string  $w$ , which distinguishes the states, exists. If no such string exists, then the states are nondistinguishable and can be merged together.

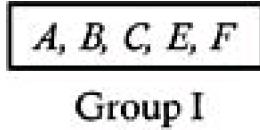
The technique that we use to find nondistinguishable states is the method of successive partitioning. We start with two

groups/partitions: one contains all nonfinal states, and other contains all the final state. This is because if every final state is known to be distinguishable from a nonfinal state, then we find transitions from members of a partition on every input symbol. If on a particular input symbol  $a$  we find that transitions from some of the members of a partition goes to one place, whereas transitions from other members of a partition go to another place, then we conclude that the members whose transitions go to one place are distinguishable from members whose transitions goes to another place. Therefore, we divide the partition in two; and we continue this partitioning until we get partitions that cannot be partitioned further. This happens when either a partition contains only one state, or when a partition contains more than one state, but they are not distinguishable from one another. If we get such a partition, we merge all of the states of this partition into a single state. For example, consider the transition diagram in [Figure 2.9](#).



**Figure 2.9:** Partitioning down to a single state.

Initially, we have two groups, as shown below:



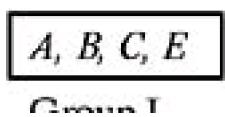
Since

$$\begin{aligned}
 \delta(A, a) &= B \\
 \delta(B, a) &= C \\
 \delta(C, a) &= C \\
 \delta(E, a) &= E \\
 \delta(F, a) &= C
 \end{aligned}$$

Partitioning of Group I is not possible, because the transitions from all the members of Group I go only to Group I. But since

$$\begin{aligned}
 \delta(A, b) &= E \\
 \delta(B, b) &= F \\
 \delta(C, b) &= F \\
 \delta(E, b) &= E \\
 \delta(F, b) &= D
 \end{aligned}$$

state  $F$  is distinguishable from the rest of the members of Group I. Hence, we divide Group I into two groups: one containing  $A, B, C, E$ , and the other containing  $F$ , as shown below:



Group I

Group II

Group III

Since

$$\delta(A, a) = B$$

$$\delta(B, a) = C$$

$$\delta(C, a) = C$$

$$\delta(E, a) = E$$

partitioning of Group I is not possible, because the transitions from all the members of Group I go only to Group I. But since

$$\delta(A, b) = E$$

$$\delta(B, b) = F$$

$$\delta(C, b) = F$$

$$\delta(E, b) = E$$

states A and E are distinguishable from states B and C. Hence, we further divide Group I into two groups: one containing A and E, and the other containing B and C, as shown below:

$A, E$

$B, C$

$F$

$D$

Group I

Group II

Group III

Group IV

Since

$$\delta(A, a) = B$$

$$\delta(E, a) = E$$

state A is distinguishable from state E. Hence, we divide Group I into two groups: one containing A and the other containing E, as shown below:

$A$

$E$

$B, C$

$F$

$D$

Group I

Group II

Group III

Group IV

Group V

Since

$$\delta(B, a) = C$$

$$\delta(C, a) = C$$

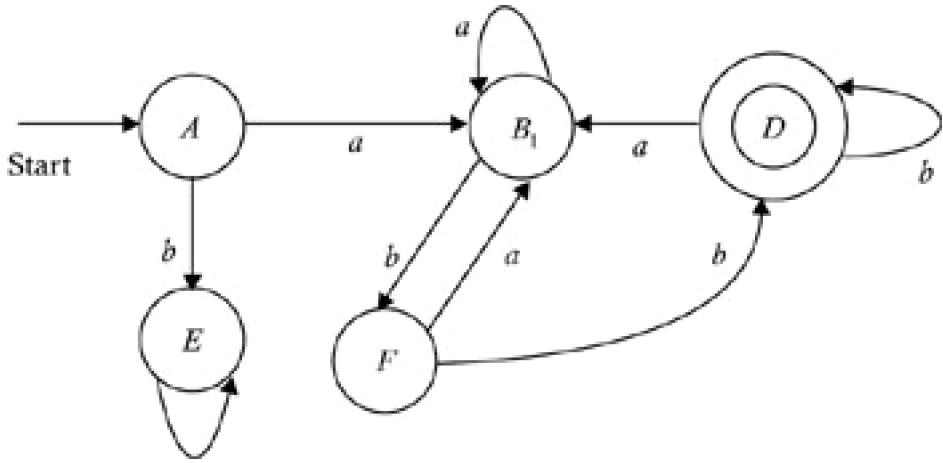
partitioning of Group III is not possible, because the transitions from all the members of Group III go to group III only. Similarly,

$$\delta(B, b) = F$$

$$\delta(C, b) = F$$

partitioning of Group III is not possible, because the transitions from all the members of Group III on b also only go to Group III.

Hence, B and C are nondistinguishable states; therefore, we merge B and C to form a single state,  $B_1$ , as shown in [Figure 2.10](#).



**Figure 2.10:** Merging nondistinguishable states B and C into a single state B<sub>1</sub>.

### 2.6.2 Algorithm for Detection of Dead States

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

Output = Set  $X$  (which is a set of dead states) {

{

$X = \emptyset$

for every  $q$  in  $(Q - F)$  do

{

flag = true;

for every  $a$  in  $\Sigma$  do

if  $(\delta(q, a) \neq q)$  then

{

flag = false

break

}

if flag = true then

$X = X \cup \{q\}$

}

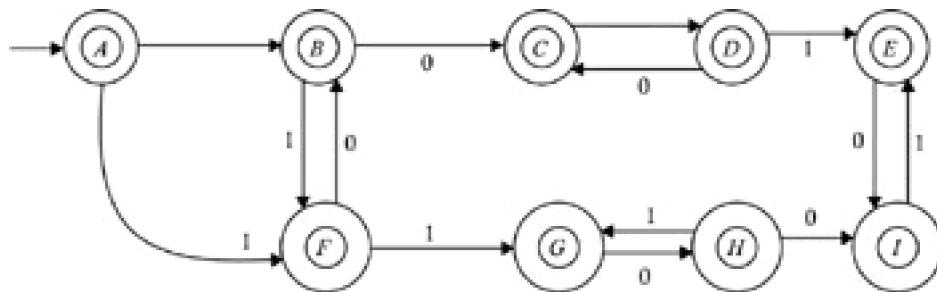
}

## 2.7 EXAMPLES OF FINITE AUTOMATA CONSTRUCTION

### EXAMPLE 2.3

Construct a finite automata accepting the set of all strings of zeros and ones, with at most one pair of consecutive zeros and at most one pair of consecutive ones.

A transition diagram of the finite automata accepting the set of all strings of zeros and ones, with at most one pair of consecutive zeros and at most one pair of consecutive ones is shown in [Figure 2.11](#).

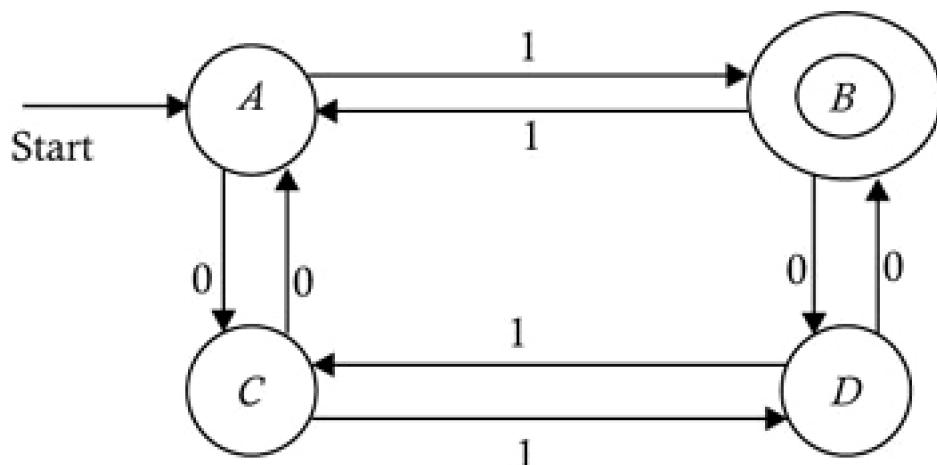


[Figure 2.11](#): Transition diagram for [Example 2.3](#) finite automata.

### EXAMPLE 2.4

Construct a finite automata that will accept strings of zeros and ones that contain even numbers of zeros and odd numbers of ones.

A transition diagram of the finite automata that accepts the set of all strings of zeros and ones that contains even numbers of zeros and odd numbers of ones is shown in [Figure 2.12](#).

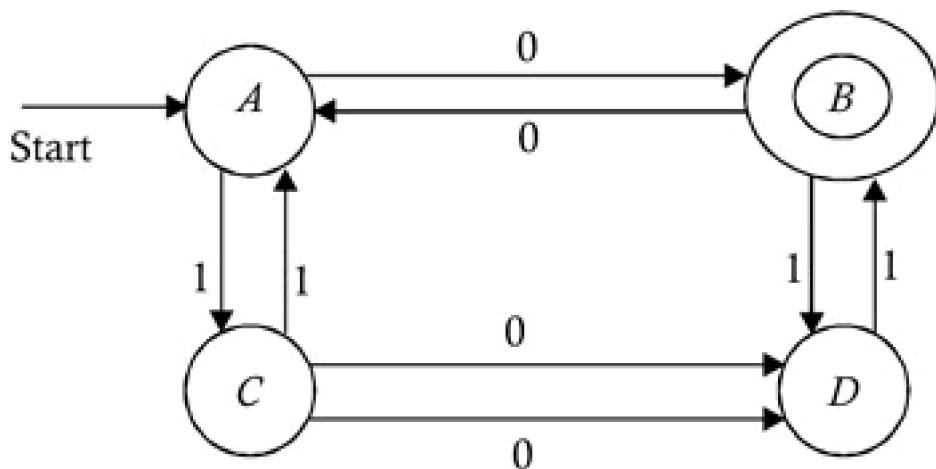


[Figure 2.12](#): Finite automata containing even number of zeros and odd number of ones.

### EXAMPLE 2.5

Construct a finite automata that will accept a string of zeros and ones that contains an odd number of zeros and an even number of ones.

A transition diagram of finite automata accepting the set of all strings of zeros and ones that contains an odd number of zeros and an even number of ones is shown in [Figure 2.13](#).

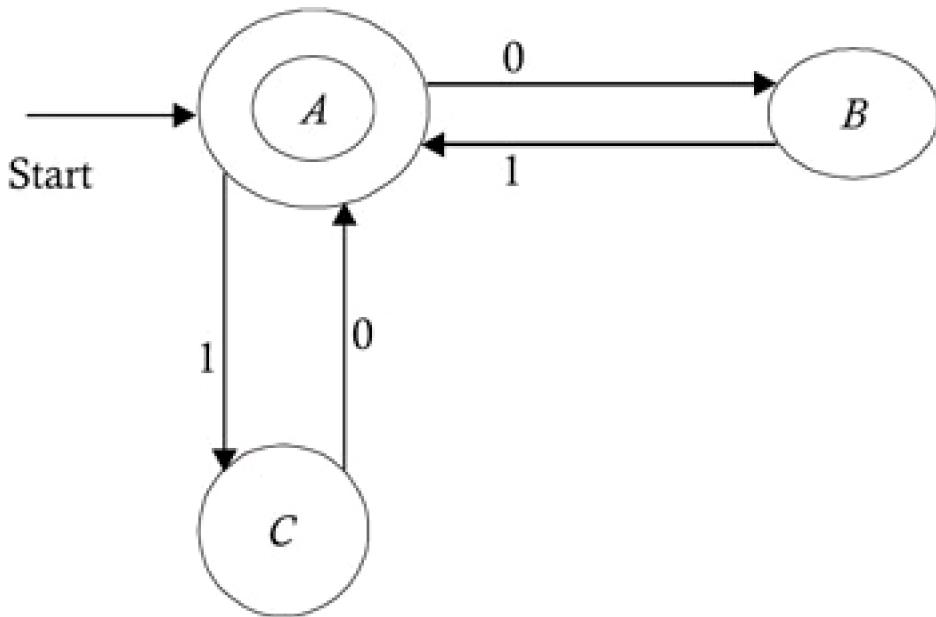


**Figure 2.13:** Finite automata containing odd number of zeros and even number of ones.

#### **EXAMPLE 2.6**

Construct the finite automata for accepting strings of zeros and ones that contain equal numbers of zeros and ones, and no prefix of the string should contain two more zeros than ones or two more ones than zeros.

A transition diagram of the finite automata that will accept the set of all strings of zeros and ones, contain equal numbers of zeros and ones, and contain no string prefixes of two more zeros than ones or two more ones than zeros is shown in [Figure 2.14](#).

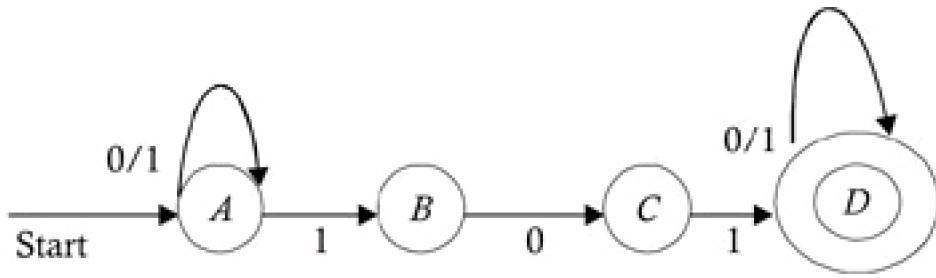


**Figure 2.14:** [Example 2.6](#) finite automata considers the set prefix.

#### **EXAMPLE 2.7**

Construct a finite automata for accepting all possible strings of zeros and ones that do not contain 101 as a substring.

[Figure 2.15](#) shows a transition diagram of the finite automata that accepts the strings containing 101 as a substring.



**Figure 2.15:** Finite automata accepts strings containing the substring 101.

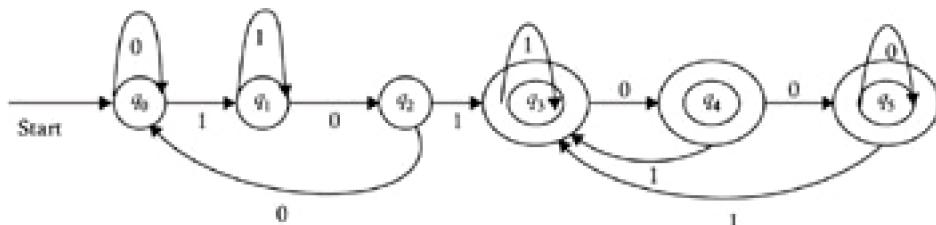
A DFA equivalent to this NFA will be:

	0	1
{A}	{A}	{A, B}
{A, B}	{A, C}	{A, B}
{A, C}	{A}	{A, B, D}
{A, B, D}*	{A, C, D}	{A, B, D}
{A, C, D}*	{A, D}	{A, B, D}
{A, C, D}*	{A, D}	{A, B, D}

Let us identify the states of this DFA using the names given below:

{A}	$q_0$
{A, B}	$q_1$
{A, C}	$q_2$
{A, B, D}	$q_3$
{A, C, D}	$q_4$
{A, D}	$q_5$

The transition diagram of this automata is shown in [Figure 2.16](#).



**Figure 2.16:** DFA using the names A-D and  $q_0$ - $q_5$ .

The complement of the automata in [Figure 2.16](#) is shown in [Figure 2.17](#).

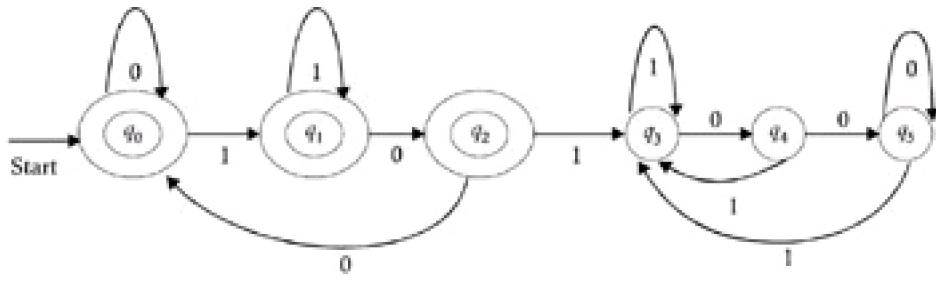


Figure 2.17: Complement to Figure 2.16 automata.

After minimization, we get the DFA shown in Figure 2.18, because states  $q_3$ ,  $q_4$ , and  $q_5$  are nondistinguishable states. Hence, they get combined, and this combination becomes a dead state and, can be eliminated.

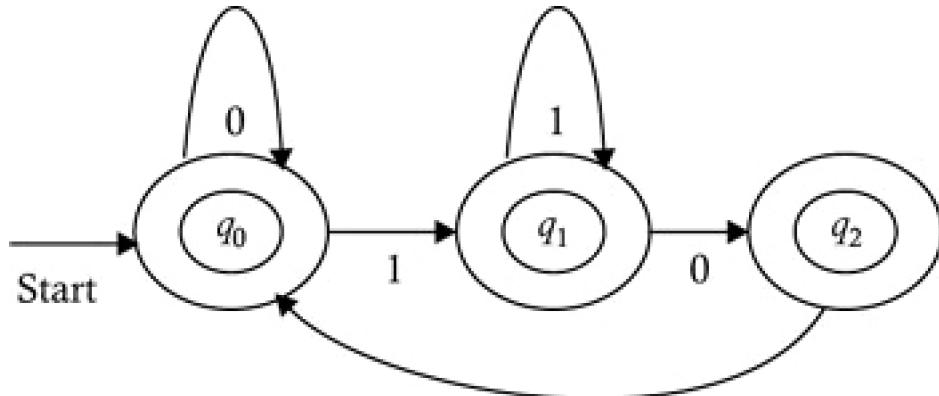


Figure 2.18: DFA after minimization.

#### EXAMPLE 2.8

---

Construct a finite automata that will accept those strings of decimal digits that are divisible by three (see Figure 2.19).

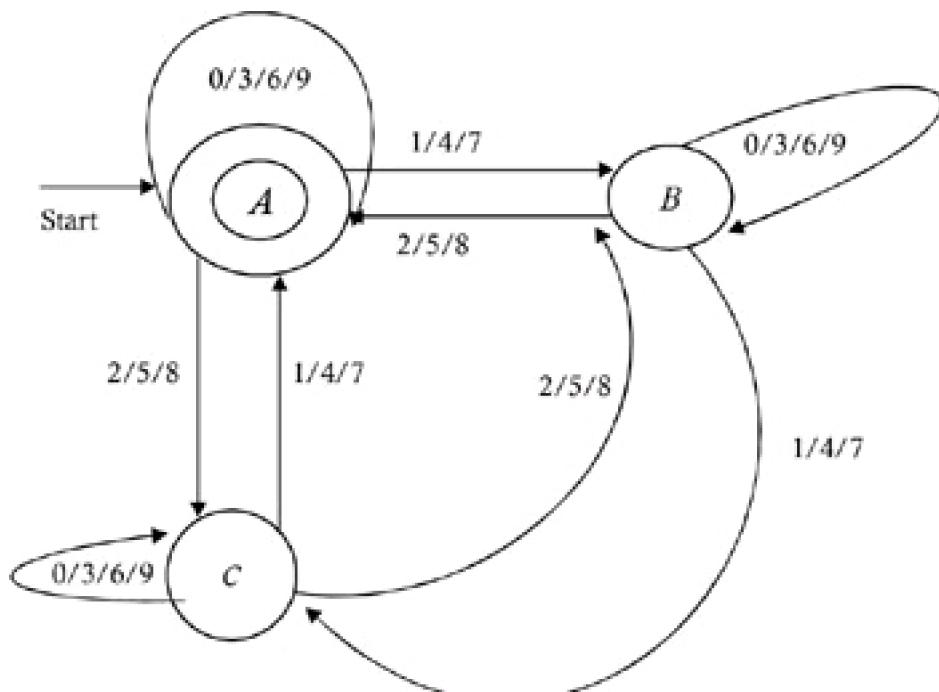


Figure 2.19: Finite automata that accepts strings divisible by three.

---

#### EXAMPLE 2.9

---

Figure 2.20

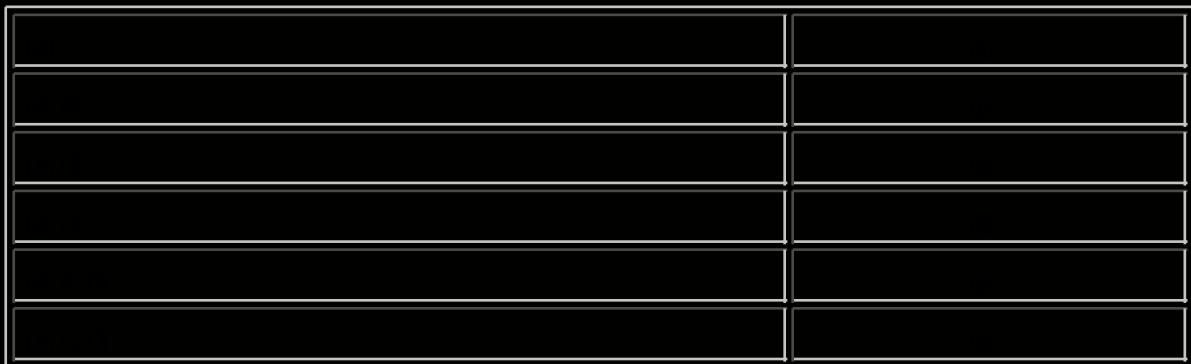
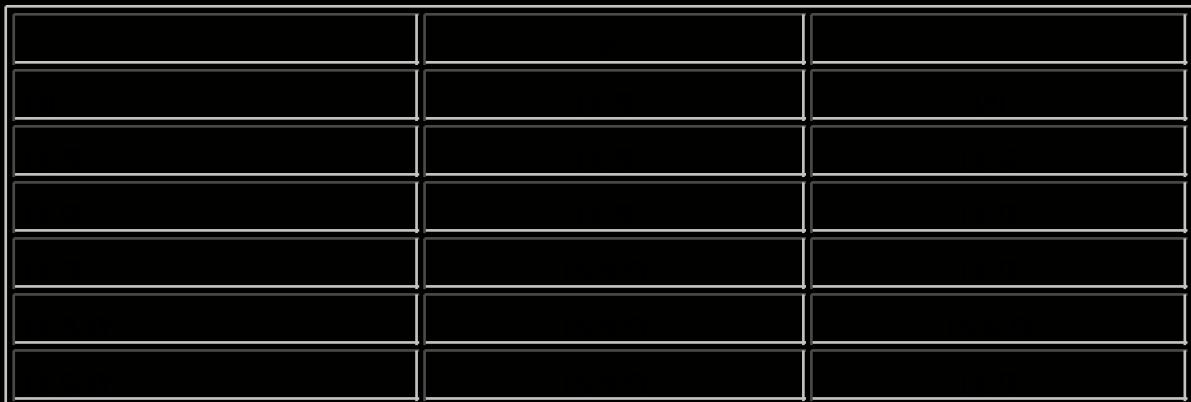
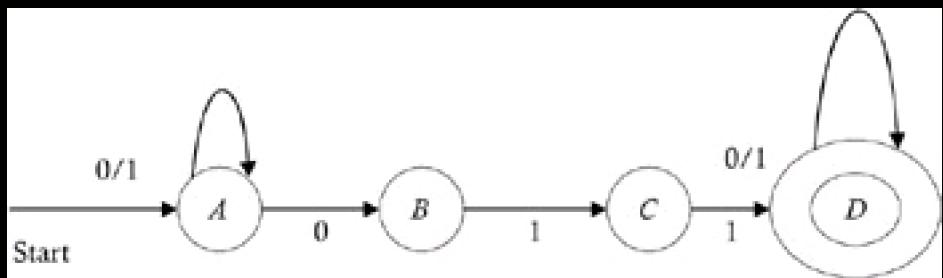


Figure 2.21

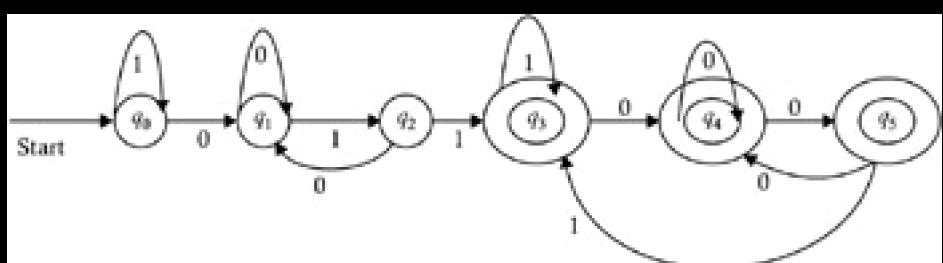
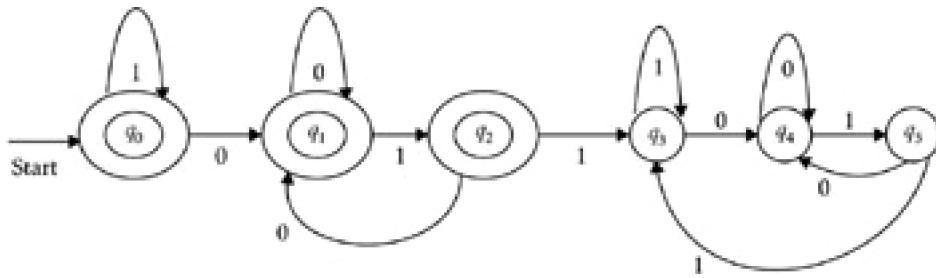


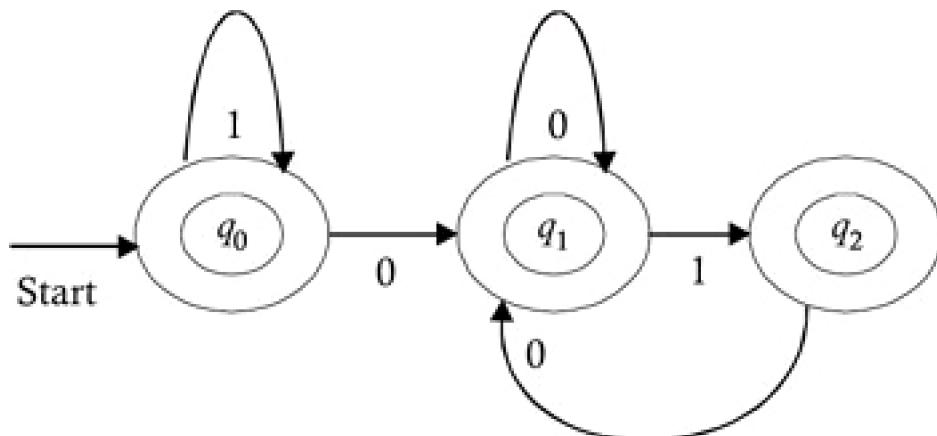
Figure 2.21

Figure 2.22



**Figure 2.22:** Complement to [Figure 2.21](#) automata.

After minimization, we get the DFA shown in [Figure 2.23](#), because the states  $q_3$ ,  $q_4$ , and  $q_5$  are nondistinguishable states. Hence, they get combined, and this combination becomes a dead state that can be eliminated.



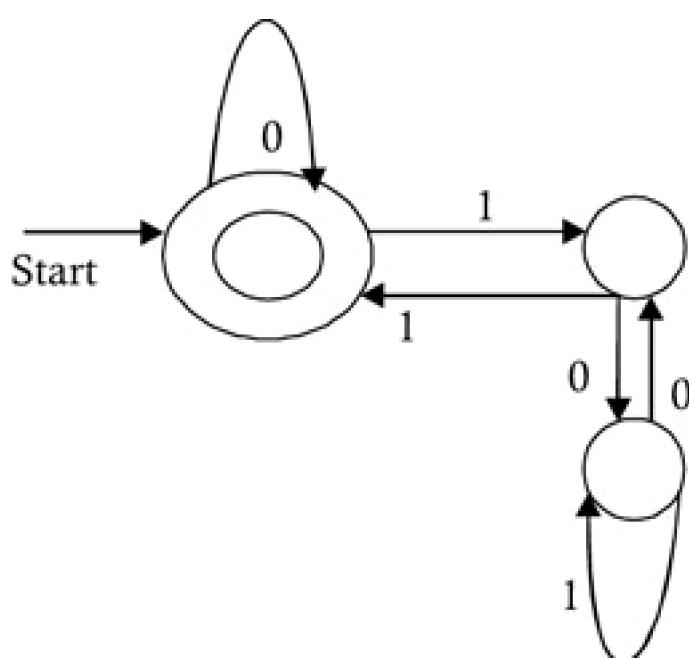
**Figure 2.23:** Minimization of nondistinguishable states of [Figure 2.22](#).

#### **EXAMPLE 2.10**

---

Construct a finite automata that will accept those strings of a binary number that are divisible by three.

The transition diagram of this automata is shown in [Figure 2.24](#).



**Figure 2.24:** Automata that accepts binary strings that are divisible by three.

---

[team 118](#)

[◀ PREVIOUS](#) [NEXT ▶](#)

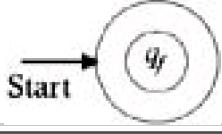
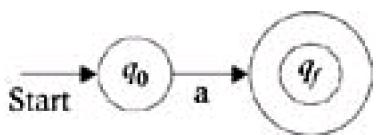
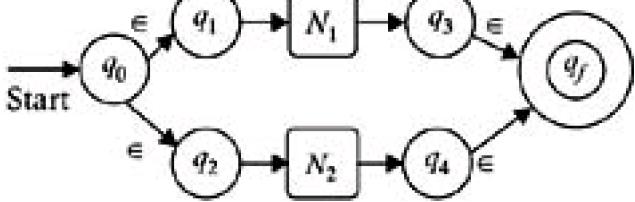
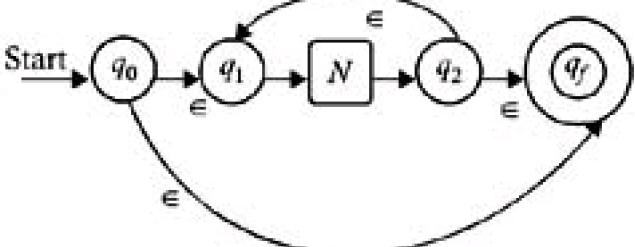
## 2.8 REGULAR SETS AND REGULAR EXPRESSIONS

### 2.8.1 Regular Sets

A regular set is a set of strings for which there exists some finite automata that accepts that set. That is, if  $R$  is a regular set, then  $R = L(M)$  for some finite automata  $M$ . Similarly, if  $M$  is a finite automata, then  $L(M)$  is always a regular set.

### 2.8.2 Regular Expression

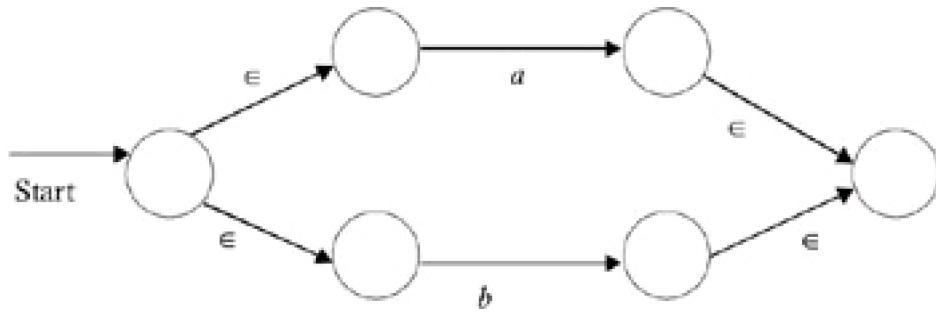
A regular expression is a notation to specify a regular set. Hence, for every regular expression, there exists a finite automata that accepts the language specified by the regular expression. Similarly, for every finite automata  $M$ , there exists a regular expression notation specifying  $L(M)$ . Regular expressions and the regular sets they specify are shown in the following table.

Regular expression	Regular Set	Finite automata
$\phi$	$\{\}$	
$\epsilon$	$\{\epsilon\}$	
Every $a$ in $\Sigma$ is a regular expression	$\{a\}$	
$r_1 + r_2$ or $r_1   r_2$ is a regular expression,	$R_1 \cup R_2$ (Where $R_1$ and $R_2$ are regular sets corresponding to $r_1$ and $r_2$ , respectively)	 <p>where <math>N_1</math> is a finite automata accepting <math>R_1</math>, and <math>N_2</math> is a finite automata accepting <math>R_2</math></p>
$r_1 \cdot r_2$ is a regular expression,	$R_1 \cdot R_2$ (Where $R_1$ and $R_2$ are regular sets corresponding to $r_1$ and $r_2$ , respectively)	 <p>where <math>N_1</math> is a finite automata accepting <math>R_1</math>, and <math>N_2</math> is a finite automata accepting <math>R_2</math></p>
$r^*$ is a regular expression,	$R^*$ (where $R$ is a regular set corresponding to $r$ )	 <p>where <math>N</math> is a finite automata accepting <math>R</math>.</p>

Hence, we only have three regular-expression operators:  $|$  or  $+$  to denote union operations,  $\cdot$  for concatenation operations, and  $*$  for closure operations. The precedence of the operators in the decreasing order is:  $*$ , followed by  $\cdot$ , followed by  $|$ . For example, consider the following regular expression:

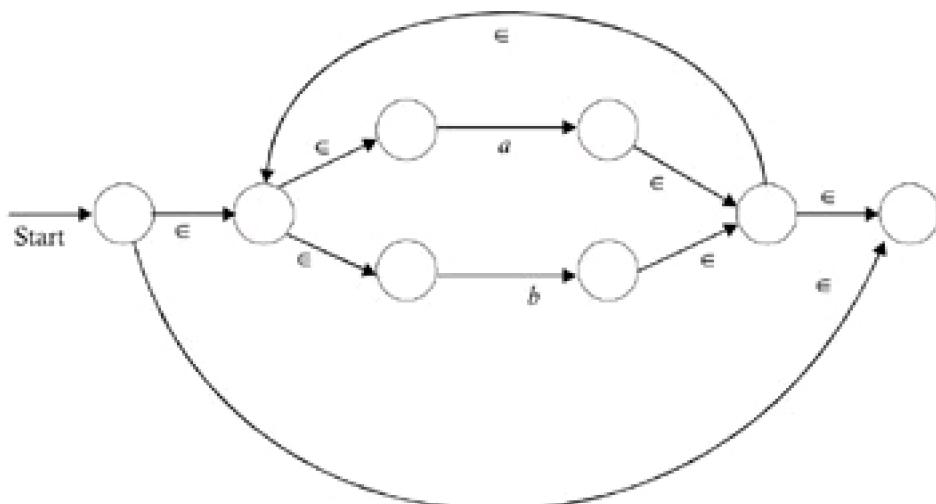
$$a. (a + b)^*. b.b$$

To construct a finite automata for this regular expression, we proceed as follows: the basic regular expressions involved are  $a$  and  $b$ , and we start with automata for  $a$  and automata for  $b$ . Since brackets are evaluated first, we initially construct the automata for  $a + b$  using the automata for  $a$  and the automata for  $b$ , as shown in [Figure 2.25](#).



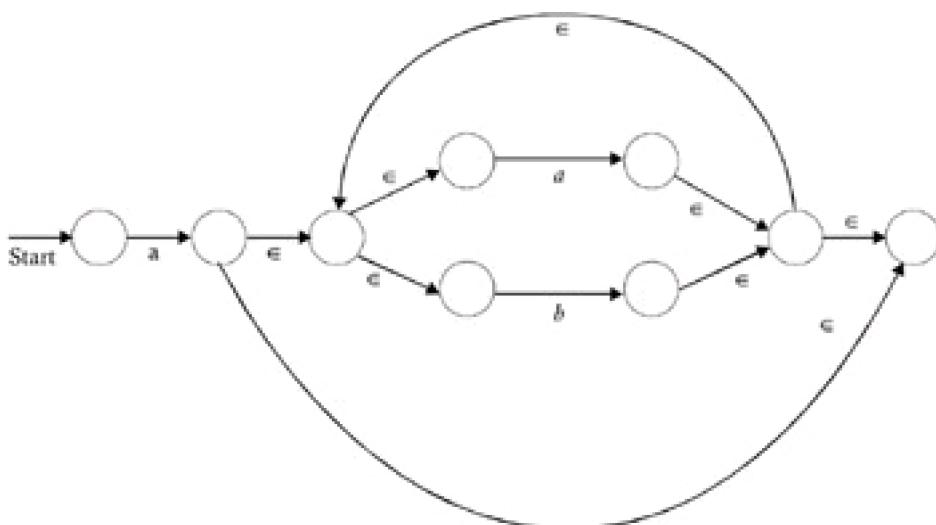
**Figure 2.25:** Transition diagram for  $(a + b)$ .

Since closure is required next, we construct the automata for  $(a + b)^*$ , using the automata for  $a + b$ , as shown in [Figure 2.26](#).



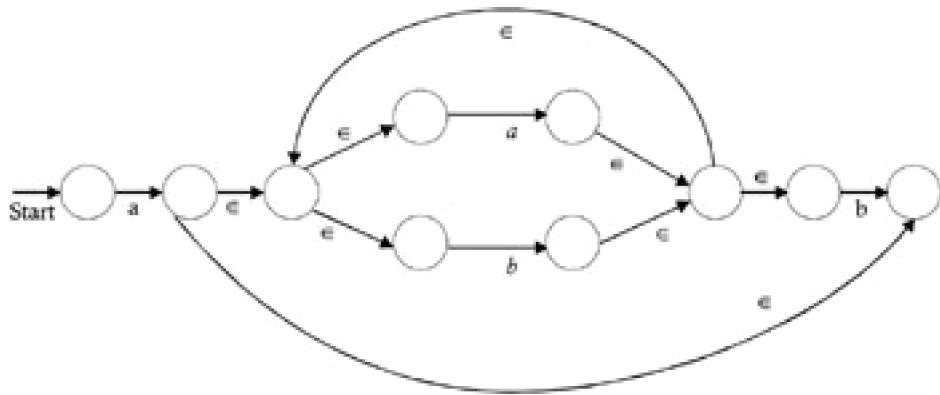
**Figure 2.26:** Transition diagram for  $(a + b)^*$ .

The next step is concatenation. We construct the automata for  $a.(a + b)^*$  using the automata for  $(a + b)^*$  and  $a$ , as shown in [Figure 2.27](#).



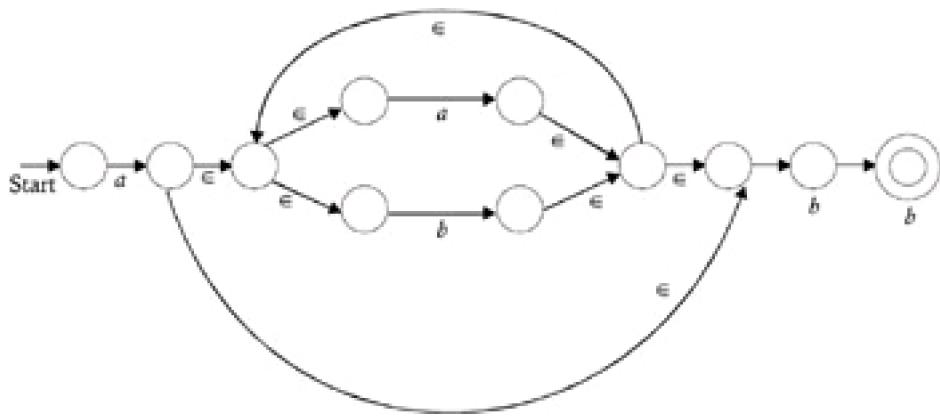
**Figure 2.27:** Transition diagram for  $a.(a + b)^*$ .

Next we construct the automata for  $a.(a + b)^*.b$ , as shown in [Figure 2.28](#).



**Figure 2.28:** Automata for  $a.(a + b)^*.b.$

Finally, we construct the automata for  $a.(a + b)^*.b.b$  ([Figure 2.29](#)).



**Figure 2.29:** Automata for  $a.(a + b)^*.b.b.$

This is an NFA with  $\epsilon$ -moves, but an algorithm exists to transform the NFA to a DFA. So, we can obtain a DFA from this NFA.

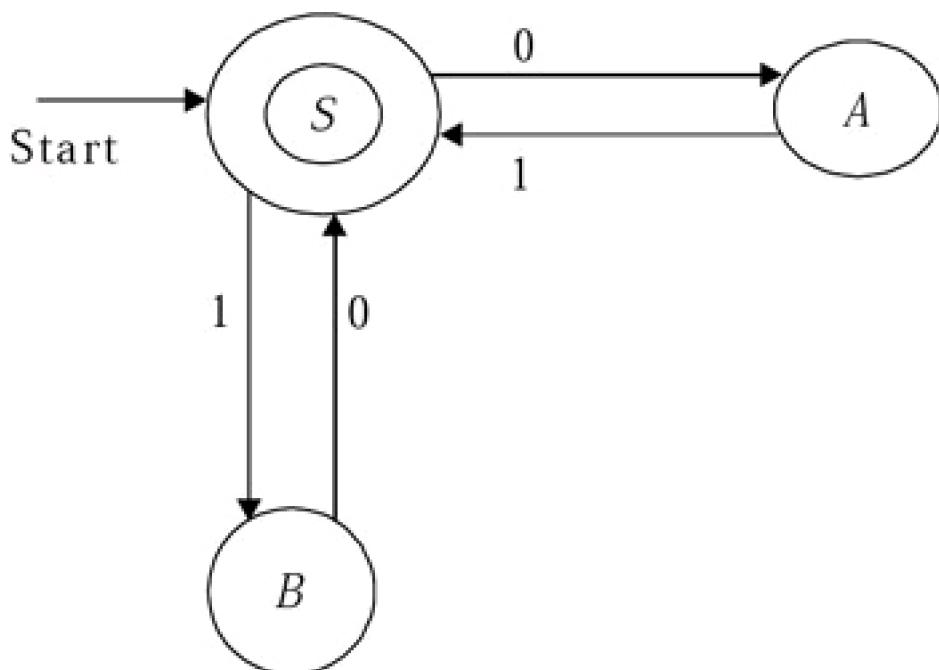
## 2.9 OBTAINING THE REGULAR EXPRESSION FROM THE FINITE AUTOMATA

Given a finite automata, to obtain a regular expression that specifies the regular set accepted by the given finite automata, the following steps are necessary:

1. Associate suitable variables (e.g., A, B, C, etc.) with the states of finite automata.
2. Form a set of equations using the following rules:
  - a. If there exists a transition from a state associated with variable A to a state associated with variable B on an input symbol a, then add the equation  $A = aB$  to the set of equation.
  - b. If the state associated with variable A is a final state, add  $A = \epsilon$  to the set of equations.
  - c. If we have the two equations  $A = ab$  and  $A = bc$ , then they can be combined as  $A = aB \mid bc$ .
3. Solve these equations to get the value of the variable associated with the starting state of the automata. In order to solve these equations, it is necessary to bring the equation in the following form:

$$S = aS/b$$

where S is a variable, and a and b are expressions that do not contain S. The solution to this equation is  $S = a^*b$ . (Here, the concatenation operator is between  $a^*$  and b, and is not explicitly shown.) For example, consider the finite automata whose transition diagram is shown in [Figure 2.30](#).



**Figure 2.30:** Deriving the regular expression for a regular set.

We use the names of the states of the automata as the variable names associated with the states.

The set of equations obtained by the application of the rules are:

$$S = 0A|1B| \in \quad \text{(I)}$$

$$A = 1S \quad \text{(II)}$$

$$B = 0S \quad \text{(III)}$$

To solve these equations, we do the substitution of (II) and (III) in (I), to obtain:

$$S = 01S|10S| \in$$

$$S = (01|10)S| \in$$

Therefore, the value of variable S comes out be:

$$S = (01|10)^* \in$$

$= (01|10)^*$  (because  $\in$  is a concatenation identity).

Therefore, the regular expression specifying the regular set accepted by the given finite automata is

$$(01|10)^*.$$

## 2.10 LEXICAL ANALYZER DESIGN

Since the function of the lexical analyzer is to scan the source program and produce a stream of tokens as output, the issues involved in the design of lexical analyzer are:

1. Identifying the tokens of the language for which the lexical analyzer is to be built, and to specify these tokens by using suitable notation, and
2. Constructing a suitable recognizer for these tokens.

Therefore, the first thing that is required is to identify what the keywords are, what the operators are, and what the delimiters are. These are the tokens of the language. After identifying the tokens of the language, we must use suitable notation to specify these tokens. This notation, should be compact, precise, and easy to understand. Regular expressions can be used to specify a set of strings, and a set of strings that can be specified by using regular-expression notation is called a "regular set." The tokens of a programming language constitutes a regular set. Hence, this regular set can be specified by using regular-expression notation. Therefore, we write regular expressions for things like operators, keywords, and identifiers. For example, the regular expressions specifying the subset of tokens of typical programming language are as follows:

```
operators = +|-|*|/|mod|div
keywords = if|while|do|then
letter = a|b|c|d|...|z|A|B|C|...|Z
digit = 0|1|2|3|4|5|6|7|8|9
identifier = letter (letter|digit)*
```

The advantage of using regular-expression notation for specifying tokens is that when regular expressions are used, the recognizer for the tokens ends up being a DFA. Therefore, the next step is the construction of a DFA from the regular expression that specifies the tokens of the language. But the DFA is a flow-chart (graphical) representation of the lexical analyzer. Therefore, after constructing the DFA, the next step is to write a program in suitable programming language that will simulate the DFA. This program acts as a token recognizer or lexical analyzer. Therefore, we find that by using regular expressions for specifying the tokens, designing a lexical analyzer becomes a simple mechanical process that involves transforming regular expressions into finite automata and generating the program for simulating the finite automata.

Therefore, it is possible to automate the procedure of obtaining the lexical analyzer from the regular expressions and specifying the tokens—and this is what precisely the tool LEX is used to do. LEX is a compiler-writing tool that facilitates writing the lexical analyzer, and hence a compiler. It inputs a regular expression that specifies the token to be recognized and generates a C program as output that acts as a lexical analyzer for the tokens specified by the inputted regular expressions.

### 2.10.1 Format of the Input or Source File of LEX

The LEX source file contains two things:

1. Auxiliary definitions having the format: name = regular expression.  
The purpose of the auxiliary definitions is to identify the larger regular expressions by using suitable names.  
LEX makes use of the auxiliary definitions to replace the names used for specifying the patterns of corresponding regular expressions.
2. The translation rules having the format:  
  
pattern {action}.

The 'pattern' specification is a regular expression that specifies the tokens, and '{action}' is a program fragment written in C to specify the action to be taken by the lexical analyzer generated by LEX when it encounters a string matching the pattern. Normally, the action taken by the lexical analyzer is to return a pair to the parser or syntax analyzer. The first member of the pair is a token, and the second member is the value or attribute of the token. For example, if the

token is an identifier, then the value of the token is a pointer to the symbol-table record that contains the corresponding name of the identifier. Hence, the action taken by the lexical analyzer is to install the name in the symbol table and return the token as an id, and to set the value of the token as a pointer to the symbol table record where the name is installed. Consider the following sample source program:

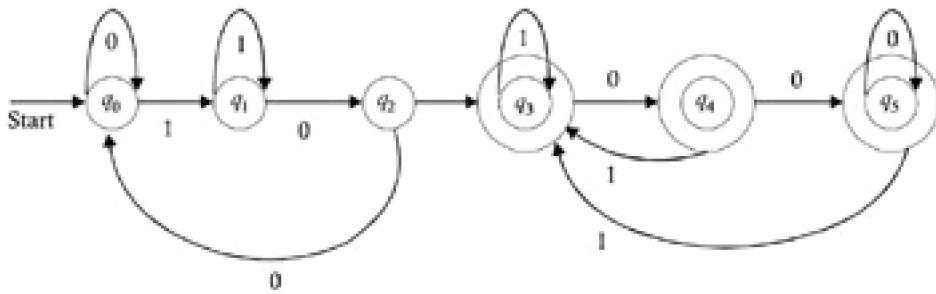
```
letter      [ a-z, A-Z ]
digit       [ 0-9 ]
%%
begin       { return ("BEGIN") }
end         { return ("END") }
if          {return ("IF")}
letter ( letter|digit)* { install ( );
                           return ("identifier")
                         }
<           { return ("LT") }
<=          { return ("LE") }
%%
definition of install()
```

In the above specification, we find that the keyword 'begin' can be matched against two patterns one specifying the keyword and the other specifying identifiers. In this case, pattern-matching is done against whichever pattern comes first in the physical order of the specification. Hence, 'begin' will be recognized as a keyword and not as an identifier. Therefore, patterns that specify keywords of the language are required to be listed before a pattern-specifying identifier; otherwise, every keyword will get recognized as identifier. A lexical analyzer generated by LEX always tries to recognize the longest prefix of the input as a token. Hence, if < = is read, it will be recognized as a token "LE" not "LT."

## 2.11 PROPERTIES OF REGULAR SETS

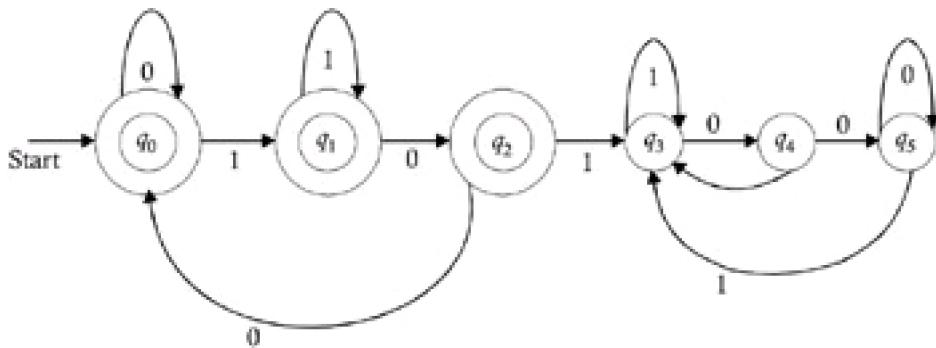
Since the union of two regular sets is always a regular set, regular sets are closed under the union operation. Similarly, regular sets are closed under concatenation and closure operations, because the concatenation of a regular sets is also a regular set, and the closure of a regular set is also a regular set.

Regular sets are also closed under the complement operation, because if  $L(M)$  is a language accepted by a finite automata  $M$ , then the complement of  $L(M)$  is  $\Sigma^* - L(M)$ . If we make all final states of  $M$  nonfinal, and we make all nonfinal states of  $M$  final, then the automata accepts  $\Sigma^* - L(M)$ ; hence, we conclude that the complement of  $L(M)$  is also a regular set. For example, consider the transition diagram in [Figure 2.31](#).



**Figure 2.31:** Transition diagram.

The transition diagram of the complement to the automata shown in [Figure 2.31](#) is shown in [Figure 2.32](#).



**Figure 2.32:** Complement to transition diagram in [Figure 2.31](#).

Since the regular sets are closed under complement as well as union operations, they are closed under intersection operations also, because intersection can be expressed in terms of both union and complement operations, as shown below:

$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$$

where  $L_1$  denotes the complement of  $L_1$ .

An automata for accepting  $L_1 \cap L_2$  is required in order to simulate the moves of an automata that accepts  $L_1$  as well as the moves of an automata that accepts  $L_2$  on the input string  $x$ . Hence, every state of the automata that accepts  $L_1 \cap L_2$  will be an ordered pair  $[p, q]$ , where  $p$  is a state of the automata accepting  $L_1$  and  $q$  is a state of the automata accepting  $L_2$ .

Therefore, if  $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  is an automata accepting  $L_1$ , and if  $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$  is an automata accepting  $L_2$ , then the automata accepting  $L_1 \cap L_2$  will be:  $M = (Q_1 \times Q_2, \Sigma, \delta, [q_1, q_2], F_1 \times F_2)$  where  $\delta([p, q], a) = [\delta_1(p, a), \delta_2(q, a)]$ . But all the members of  $Q_1 \times Q_2$  may not necessarily represent reachable states of  $M$ . Hence, to

reduce the amount of work, we start with a pair  $[q_1, q_2]$  and find transitions on every member of  $\Sigma$  from  $[q_1, q_2]$ . If some transitions go to a new pair, then we only generate that pair, because it will then represent a reachable state of  $M$ .

We next consider the newly generated pairs to find out the transitions from them. We continue this until no new pairs can be generated.

Let  $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  be a automata accepting  $L_1$ , and let  $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$  be a automata accepting  $L_2$ .  $M = (Q, \Sigma, \delta, q_0, F)$  will be an automata accepting  $L_1 \cap L_2$ .

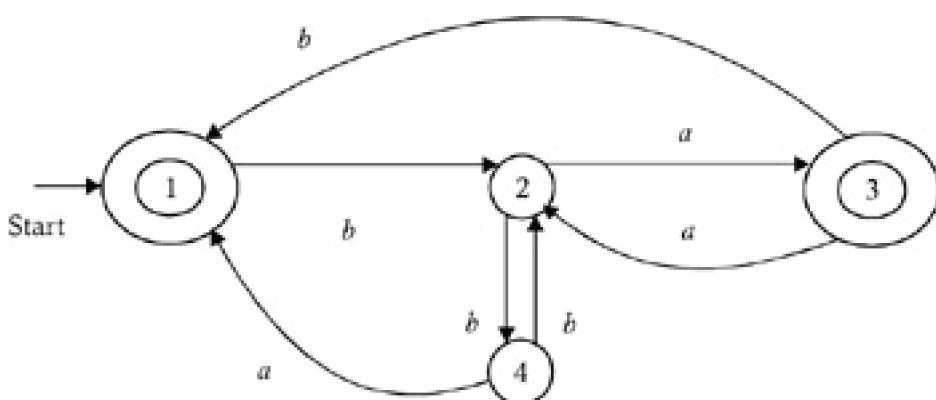
begin

```

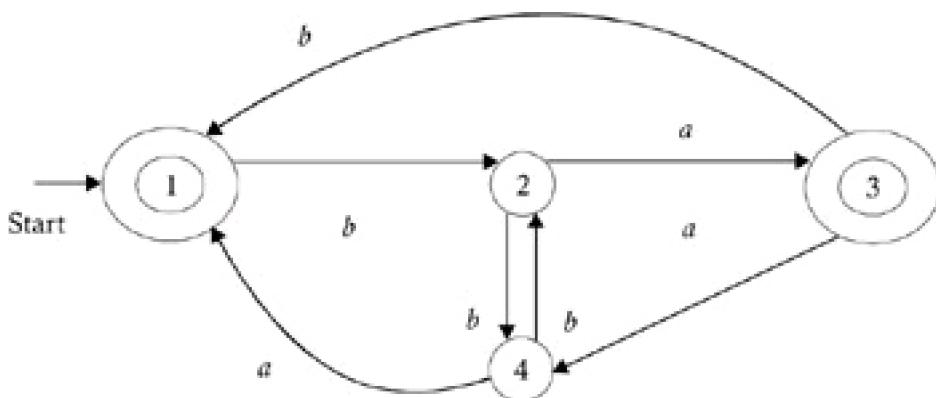
Qold = Φ
Qnew = { [ q1, q2 ] }
While ( Qold ≠ Qnew )
{
    Temp = Qnew - Qold
    Qold = Qnew
    for every pair [p, q] in Temp do
        for every a in Σ do
            Qnew = Qnew ∪ δ ([p, q], a)
}
Q = Qnew
end

```

Consider the automatas and their transition diagrams shown in [Figure 2.33](#) and [Figure 2.34](#).



**Figure 2.33:** Transition diagram of automata  $M_1$ .



**Figure 2.34:** Transition diagram of automata  $M_2$ .

The transition table for the automata accepting  $L(M_1) \cap L(M_2)$  is:

$\delta$	$A$	$b$
[1, 1]	[1, 1]	[2, 4]
[2, 4]	[3, 3]	[4, 2]
[3, 3]	[2, 2]	[1, 1]
[4, 2]	[1, 1]	[2, 4]
[2, 2]	[3, 1]	[4, 4]
[3, 1]	[2, 1]	[1, 4]
[4, 4]	[1, 3]	[2, 2]
[2, 1]	[3, 1]	[4, 4]
[1, 4]*	[1, 3]	[2, 2]
[1, 3]	[1, 2]	[2, 1]
[1, 2]*	[1, 1]	[2, 4]

We associate the names with states of the automata obtained, as shown below:

[1, 1]	$A$
[2, 4]	$B$
[3, 3]	$C$
[4, 2]	$D$
[2, 2]	$E$
[3, 1]	$F$
[4, 4]	$G$
[2, 1]	$H$
[1, 4]	$I$
[1, 3]	$J$
[1, 2]	$K$

The transition table of the automata using the names associated above is:

$\delta$	$a$	$B$
$A$	$A$	$B$
$B$	$C$	$D$
$C$	$E$	$A$
$D$	$A$	$B$
$E$	$F$	$G$
$F$	$H$	$I$
$G$	$J$	$E$

<i>H</i>	<i>F</i>	<i>G</i>
<i>I</i>	<i>J</i>	<i>E</i>
<i>J</i>	<i>K</i>	<i>H</i>
<i>K*</i>	<i>A</i>	<i>B</i>

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## 2.12 EQUIVALENCE OF TWO AUTOMATAS

Automatas  $M_1$  and  $M_2$  are said to be equivalent if they accept the same language; that is,  $L(M_1) = L(M_2)$ . It is possible to test whether the automatas  $M_1$  and  $M_2$  accept the same language—and hence, whether they are equivalent or not. One method of doing this is to minimize both  $M_1$  and  $M_2$ , and if the minimal state automatas obtained from  $M_1$  and  $M_2$  are identical, then  $M_1$  is equivalent to  $M_2$ .

Another method to test whether or not  $M_1$  is equivalent to  $M_2$  is to find out if:

$$(L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2)) = \emptyset$$

For this, complement  $M_2$ , and construct an automata that accepts both the intersection of language accepted by  $M_1$  and the complement of  $M_2$ . If this automata accepts an empty set, then it means that there is no string acceptable to  $M_1$  that is not acceptable to  $M_2$ . Similarly, construct an automata that accepts the intersection of language accepted by  $M_2$  and the complement of  $M_1$ . If this automata accepts an empty set, then it means that there is no string acceptable to  $M_2$  that is not acceptable to  $M_1$ . Hence, the language accepted by  $M_1$  is same as the language accepted by  $M_2$ .

## Chapter 3: Context-Free Grammar and Syntax Analysis

### 3.1 SYNTAX ANALYSIS

In the syntax-analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. If the tokens in a string are grouped according to the language's rules of syntax, then the string of tokens generated by the lexical analyzer is accepted as a valid construct of the language; otherwise, an error handler is called. Hence, two issues are involved when designing the syntax-analysis phase of a compilation process:

1. All valid constructs of a programming language must be specified; and by using these specifications, a valid program is formed. That is, we form a specification of what tokens the lexical analyzer will return, and we specify in what manner these tokens are to be grouped so that the result of the grouping will be a valid construct of the language.
2. A suitable recognizer will be designed to recognize whether a string of tokens generated by the lexical analyzer is a valid construct or not.

Therefore, suitable notation must be used to specify the constructs of a language. The notation for the construct specifications should be compact, precise, and easy to understand. The syntax-structure specification for the programming language (i.e., the valid constructs of the language) uses context-free grammar (CFG), because for certain classes of grammar, we can automatically construct an efficient parser that determines if a source program is syntactically correct. Hence, CFG notation is required topic for study.

## 3.2 CONTEXT-FREE GRAMMAR

CFG notation specifies a context-free language that consists of terminals, nonterminals, a start symbol, and productions. The terminals are nothing more than tokens of the language, used to form the language constructs. Nonterminals are the variables that denote a set of strings. For example,  $S$  and  $E$  are nonterminals that denote statement strings and expression strings, respectively, in a typical programming language. The nonterminals define the sets of strings that are used to define the language generated by the grammar.

They also impose a hierarchical structure on the language, which is useful for both syntax analysis and translation. Grammar productions specify the manner in which the terminals and string sets, defined by the nonterminals, can be combined to form a set of strings defined by a particular nonterminal. For example, consider the production  $S \rightarrow aSb$ . This production specifies that the set of strings defined by the nonterminal  $S$  are obtained by concatenating terminal  $a$  with any string belonging to the set of strings defined by nonterminal  $S$ , and then with terminal  $b$ . Each production consists of a nonterminal on the left-hand side, and a string of terminals and nonterminals on the right-hand side. The left-hand side of a production is separated from the right-hand side using the " $\rightarrow$ " symbol, which is used to identify a relation on a set  $(V \cup T)^*$ .

Therefore context-free grammar is a four-tuple denoted as:

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

where:

1.  $V$  is a finite set of symbols called as nonterminals or variables,
2.  $T$  is a set of symbols that are called as terminals,
3.  $P$  is a set of productions, and
4.  $S$  is a member of  $V$ , called as start symbol.

For example:

$$\begin{aligned} G &= (\{S\}, \{a, b\}, P, S) \text{ where } P \text{ contains:} \\ P &= \{ S \rightarrow asa, \\ &\quad S \rightarrow bsb, \\ &\quad S \rightarrow \epsilon \\ &\quad \} \end{aligned}$$

### 3.2.1 Derivation

Derivation refers to replacing an instance of a given string's nonterminal, by the right-hand side of the production rule, whose left-hand side contains the nonterminal to be replaced. Derivation produces a new string from a given string; therefore, derivation can be used repeatedly to obtain a new string from a given string. If the string obtained as a result of the derivation contains only terminal symbols, then no further derivations are possible. For example, consider the following grammar for a string  $S$ :

$$G = (\{S\}, \{a, b\}, P, S)$$

where  $P$  contains the following productions:

$$\begin{aligned} P &= \{ S \rightarrow aSa, \\ &\quad S \rightarrow bSb, \\ &\quad \dots \} \end{aligned}$$

$\boxed{S \rightarrow \epsilon}$

}

It is possible to replace the nonterminal  $S$  by a string  $aSa$ . Therefore, we obtain  $aSa$  from  $S$  by deriving  $S$  to  $aSa$ . It is possible to replace  $S$  in  $aSa$  by  $\epsilon$ , to obtain a string  $aa$ , which cannot be further derived.

If  $\alpha_1$  and  $\alpha_2$  are the two strings, and if  $\alpha_2$  can be obtained from  $\alpha_1$ , then we say  $\alpha_1$  is related to  $\alpha_2$  by "derives to relation," which is denoted by " $\rightarrow$ ". Hence, we write  $\alpha_1 \rightarrow \alpha_2$ , which translates to:  $\alpha_1$  derives to  $\alpha_2$ . The symbol  $\rightarrow$  denotes a derive to relation that relates the two strings  $\alpha_1$  and  $\alpha_2$  such that  $\alpha_2$  is a direct derivative of  $\alpha_1$  (if  $\alpha_2$  can be obtained from  $\alpha_1$  by a derivation of only one step). Therefore,  $\xrightarrow{+}$  will denote the transitive closure of derives to relation, and if we have the two strings  $\alpha_1$  and  $\alpha_2$  such that  $\alpha_2$  can be obtained from  $\alpha_1$  by derivation, but  $\alpha_2$  may not be a direct derivative of  $\alpha_1$ , then we write  $\alpha_1 \xrightarrow{+} \alpha_2$ , which translates to:  $\alpha_1$  derives to  $\alpha_2$  through one or more derivations.

Similarly,  $\xrightarrow{*}$  denotes the reflexive transitive closure of derives to relation; and if we have two strings  $\alpha_1$  and  $\alpha_2$  such that  $\alpha_1$  derives to  $\alpha_2$  in zero, one, or more derivations, then we write  $\alpha_1 \xrightarrow{*} \alpha_2$ . For example, in the grammar above, we find that  $S \rightarrow aSa \rightarrow abSba \rightarrow abba$ . Therefore, we can write  $S \xrightarrow{*} abba$ .

The language defined by a CFG is nothing but the set of strings of terminals that, in the case of the string  $S$ , can be generated from  $S$  as a result of derivations using productions of the grammar. Hence, they are defined as the set of those strings of terminals that are derivable from the grammar's start symbol. Therefore, if  $G = (V, T, P, S)$  is a grammar, then the language by the grammar is denoted as  $L(G)$  and defined as:

$$L(G) = \{ \omega \mid \omega \text{ is in } T^* \text{ and } S \xrightarrow{*} \omega \}$$

The above grammar can generate the string  $\epsilon, aa, bb, abba, \dots$ , but not  $aba$ .

### 3.2.2 Standard Notation

1. The capital letters toward the start of the alphabet are used to denote nonterminals (e.g.,  $A, B, C$ , etc.).
2. Lowercase letters toward the start of the alphabet are used to denote terminals (e.g.,  $a, b, c$ , etc.).
3.  $S$  is used to denote the start symbol.
4. Lowercase letters toward the end of the alphabet (e.g.,  $u, v, w$ , etc.) are used to denote strings of terminals.
5. The symbols  $\alpha, \beta, \gamma$ , and so forth are used to denote strings of terminals as well as strings of nonterminals.
6. The capital letters toward the end of alphabet (e.g.,  $X, Y$ , and  $Z$ ) are used to denote grammar symbols, and they may be terminals or nonterminals.

The benefit of using these notations is that it is not required to explicitly specify all four grammar components. A grammar can be specified by only giving the list of productions; and from this list, we can easily get information about the terminals, nonterminals, and start symbols of the grammar.

### 3.2.3 Derivation Tree or Parse Tree

When deriving a string  $w$  from  $S$ , if every derivation is considered to be a step in the tree construction, then we get the graphical display of the derivation of string  $w$  as a tree. This is called a "derivation tree" or a "parse tree" of string  $w$ . Therefore, a derivation tree or parse tree is the display of the derivations as a tree. Note that a tree is a derivation tree if it satisfies the following requirements:

1. All the leaf nodes of the tree are labeled by terminals of the grammar.

2. The root node of the tree is labeled by the start symbol of the grammar.
3. The interior nodes are labeled by the nonterminals.
4. If an interior node has a label  $A$ , and it has  $n$  descendants with labels  $X_1, X_2, \dots, X_n$  from left to right, then the production rule  $A \rightarrow X_1 X_2 X_3 \dots X_n$  must exist in the grammar.

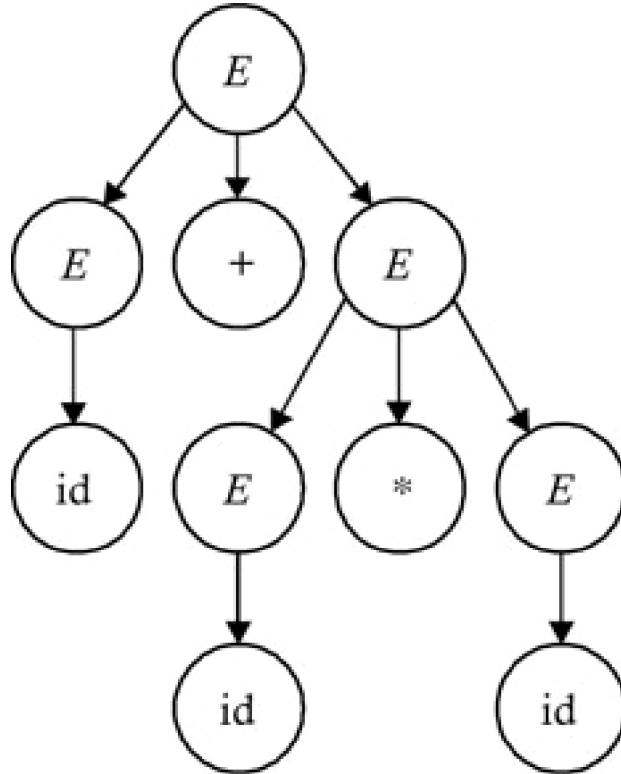
For example, consider a grammar whose list of productions is:

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow \text{id}$$

The tree shown in [Figure 3.1](#) is a derivation tree for a string  $\text{id} + \text{id} * \text{id}$ .



**Figure 3.1:** Derivation tree for the string  $\text{id} + \text{id} * \text{id}$ .

Given a parse (derivation) tree, a string whose derivation is represented by the given tree is one obtained by concatenating the labels of the leaf nodes of the parse tree in a left-to-right order.

Consider the parse tree shown in [Figure 3.2](#). A string whose derivation is represented by this parse tree is *abba*.

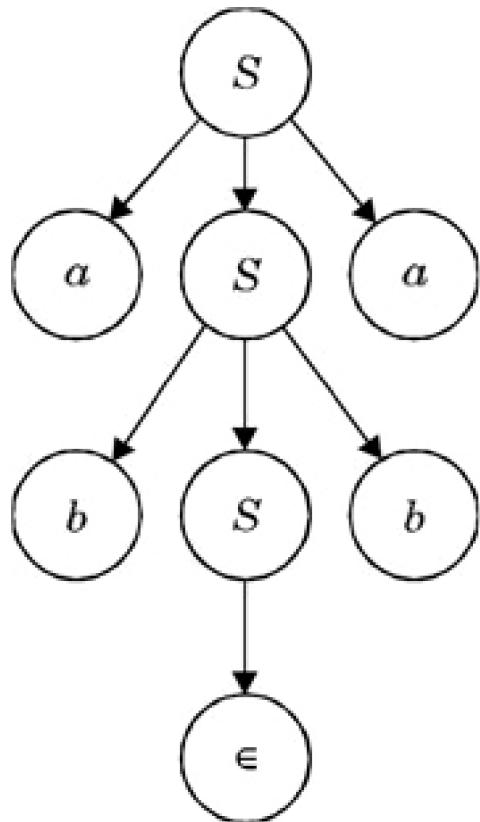


Figure 3.2: Parse tree resulting from leaf-node concatenation.

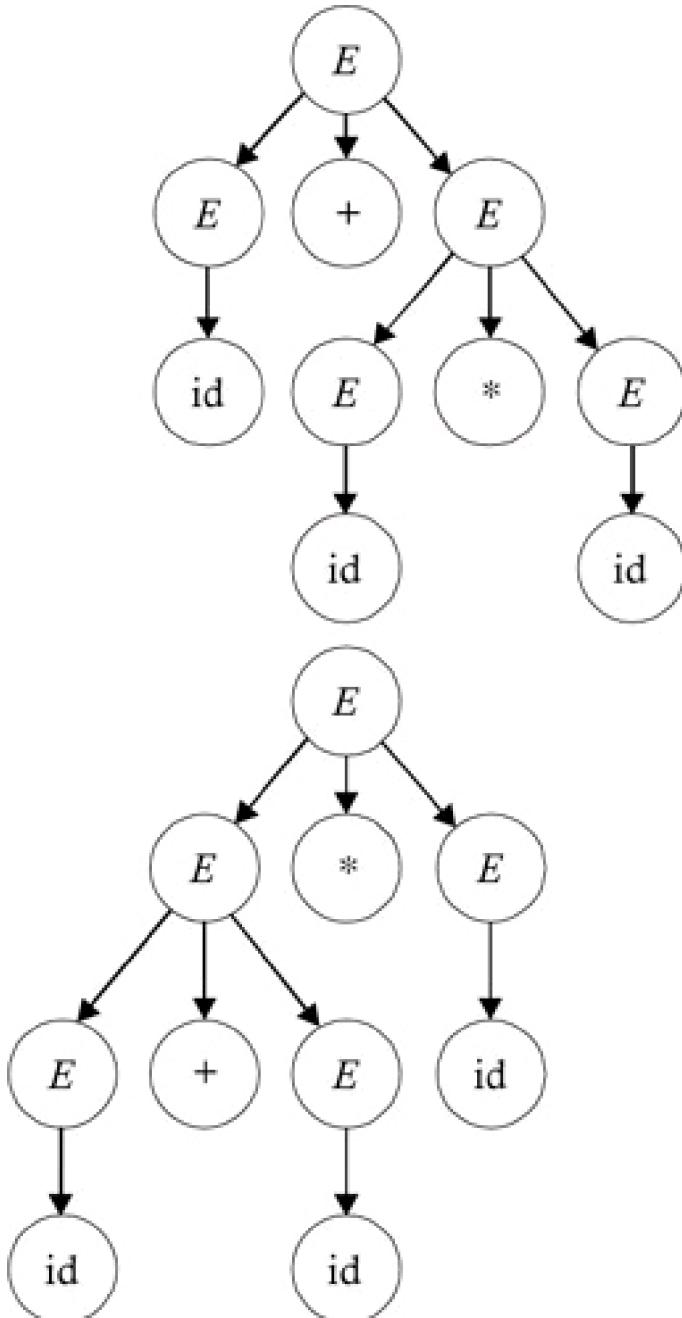
Since a parse tree displays derivations as a tree, given a grammar  $G = (V, T, P, S)$  for every  $w$  in  $T^*$ , and which is derivable from  $S$ , there exists a parse tree displaying the derivation of  $w$  as a tree. Therefore, we can define the language generated by the grammar as:

$$L(G) = \{ w \mid w \text{ is in } T^* \text{ and there exists at least one parse tree for } w \}$$

For some  $w$  in  $L(G)$ , there may exist more than one parse tree. That means that more than one way may exist to derive  $w$  from  $S$ , using the productions of the grammar. For example, consider a grammar having the productions listed below:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E^* E \\ E &\rightarrow \text{id} \end{aligned}$$

We find that for a string  $\text{id} + \text{id}^* \text{id}$ , there exists more than one parse tree, as shown in [Figure 3.3](#).



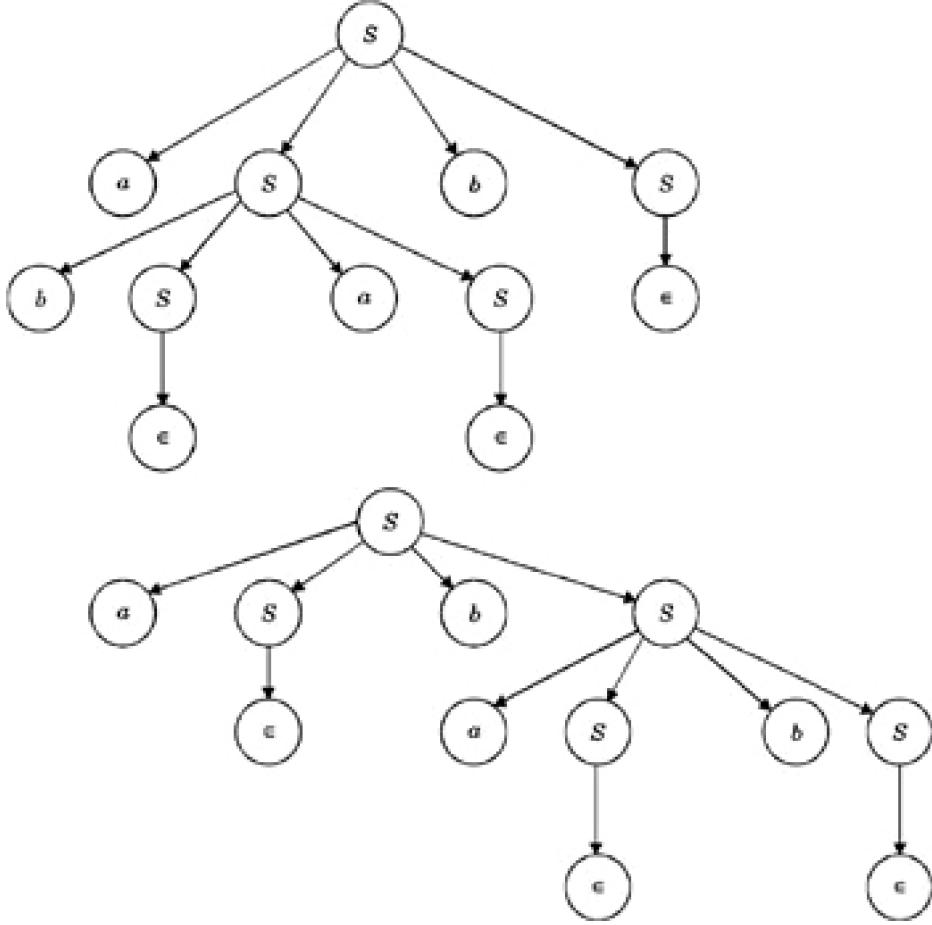
**Figure 3.3:** Multiple parse trees.

If more than one parse tree exists for some  $w$  in  $L(G)$ , then  $G$  is said to be an "ambiguous" grammar. Therefore, the grammar having the productions  $E \rightarrow E + E \mid E * E \mid id$  is an ambiguous grammar, because there exists more than one parse tree for the string  $id + id * id$  in  $L(G)$  of this grammar.

Consider a grammar having the following productions:

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

This grammar is also an ambiguous grammar, because more than one parse tree exists for a string  $abab$  in  $L(G)$ , as shown in [Figure 3.4](#).



**Figure 3.4:** Ambiguous grammar parse trees.

The parse tree construction process is such that the order in which the nonterminals are considered for replacement does not matter. That is, given a string  $w$ , the parse tree for that string (if it exists) can be constructed by considering the nonterminals for derivation in any order. The two specific orders of derivation, which are important from the point of view of parsing, are:

1. Left-most order of derivation
2. Right-most order of derivation

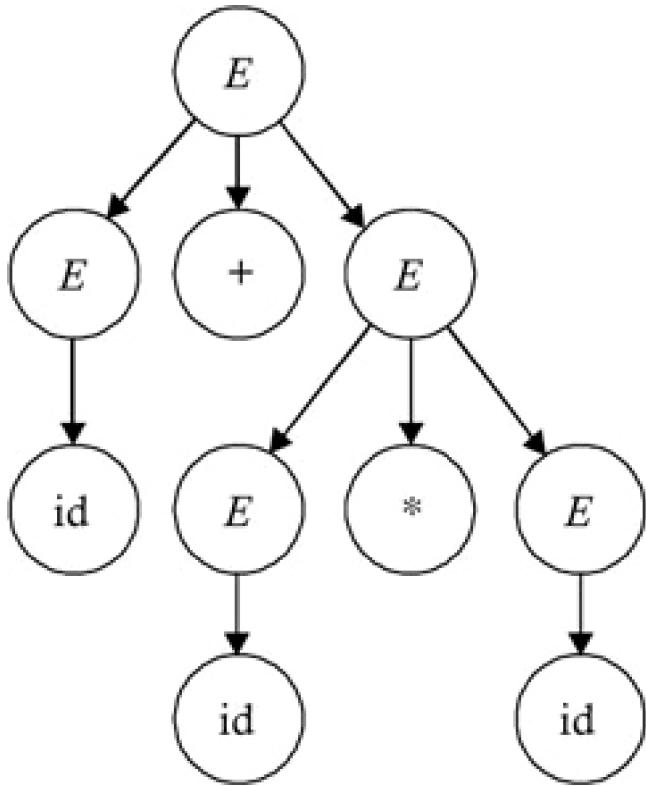
The left-most order of derivation is that order of derivation in which a left-most nonterminal is considered first for derivation at every stage in the derivation process. For example, one of the left-most orders of derivation for a string  $\text{id} + \text{id}^* \text{id}$  is:

$$E \rightarrow E + E \rightarrow \text{id} + E \rightarrow \text{id} + E^*E \rightarrow \text{id} + \text{id}^*E \rightarrow \text{id} + \text{id}^*\text{id}$$

In a right-most order of derivation, the right-most nonterminal is considered first. For example, one of the right-most orders of derivation for  $\text{id} + \text{id}^* \text{id}$  is:

$$E \rightarrow E + E \rightarrow E + E^*E \rightarrow E + E^*\text{id} \rightarrow E + \text{id}^*\text{id} \rightarrow \text{id} + \text{id}^*\text{id}$$

The parse tree generated by using the left-most order of derivation of  $\text{id} + \text{id}^* \text{id}$  and the parse tree generated by using the right-most order of derivation of  $\text{id} + \text{id}^* \text{id}$  are the same; hence, these orders are equivalent. A parse tree generated using these orders is shown in [Figure 3.5](#).



**Figure 3.5:** Parse tree generated by using both the right- and left-most derivation orders.

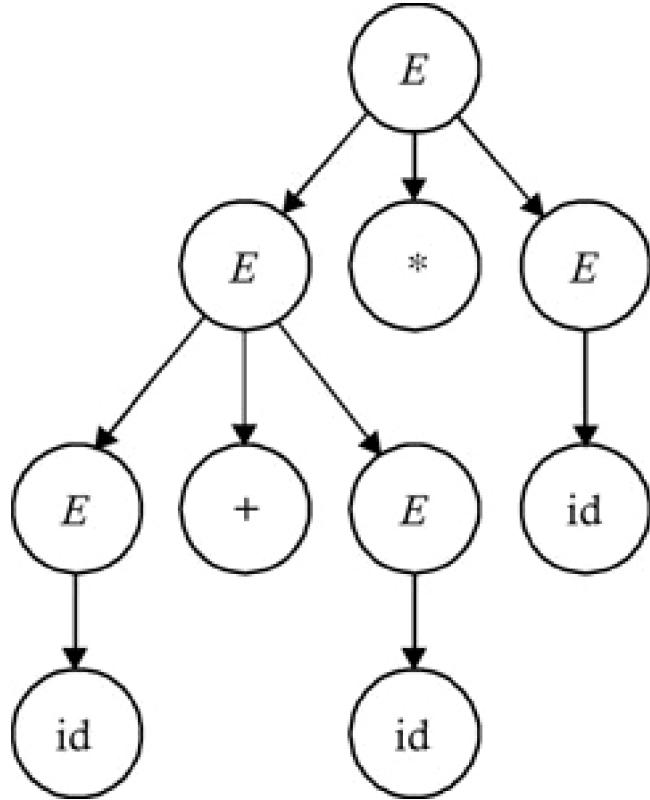
Another left-most order of derivation of  $\text{id} + \text{id}^* \text{id}$  is given below:

$$E \rightarrow E^* E \rightarrow E + E^* E \rightarrow \text{id} + E^* E \rightarrow \text{id} + \text{id}^* E \rightarrow \text{id} + \text{id}^* \text{id}$$

And here is another right-most order of derivation of  $\text{id} + \text{id}^* \text{id}$ :

$$E \rightarrow E^* E \rightarrow E^* \text{id} \rightarrow E + E^* \text{id} \rightarrow E + \text{id}^* \text{id} \rightarrow \text{id} + \text{id}^* \text{id}$$

The parse tree generated by using the left-most order of derivation of  $\text{id} + \text{id}^* \text{id}$  and the parse tree generated using the right-most order of derivation of  $\text{id} + \text{id}^* \text{id}$  are the same. Hence, these orders are equivalent. A parse tree generated using these orders is shown in [Figure 3.6](#).



**Figure 3.6:** Parse tree generated from both the left- and right-most orders of derivation.

Therefore, we conclude that for every left-most order of derivation of a string  $w$ , there exists an equivalent right-most order of derivation of  $w$ , generating the same parse tree.

**Note** If a grammar  $G$  is unambiguous, then for every  $w$  in  $L(G)$ , there exists exactly one parse tree. Hence, there exists exactly one left-most order of derivation and (equivalently) one right-most order of derivation for every  $w$  in  $L(G)$ . But if grammar  $G$  is ambiguous, then for some  $w$  in  $L(G)$ , there exists more than one parse tree. Therefore, there is more than one left-most order of derivation; and equivalently, there is more than one right-most order of derivation.

### 3.2.4 Reduction of Grammar

Reduction of a grammar refers to the identification of those grammar symbols (called "useless grammar symbols"), and hence those productions, that do not play any role in the derivation of any  $w$  in  $L(G)$ , and which we eliminate from the grammar. This has no effect on the language generated by the grammar. For example, a grammar symbol  $X$  is useful if and only if:

1. It derives to a string of terminals, and
2. It is used in the derivation of at least one  $w$  in  $L(G)$ .

Thus,  $X$  is useful if and only if:

1.  $X \xrightarrow{*} w$ , where  $w$  is in  $T^*$ , and
2.  $S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w$  in  $L(G)$ .

Therefore, reduction of a given grammar  $G$ , involves:

1. Identification of those grammar symbols that are not capable of deriving to a  $w$  in  $T^*$  and eliminating them from the grammar; and
2. Identification of those grammar symbols that are not used in any derivation and eliminating them from the grammar.

When identifying the grammar symbols that do not derive a  $w$  in  $T^*$ , only nonterminals need be tested, because every

terminal member of  $T$  will also be in  $T^*$ ; and by default, they satisfy the first condition. A simple, iterative algorithm can be used to identify those nonterminals that do not derive to  $w$  in  $T^*$ : we start with those productions that are of the form  $A \rightarrow w$  that is, those productions whose right side is  $w$  in  $T^*$ . We mark as nonterminal every  $A$  on the left side of every production that is capable of deriving to  $w$  in  $T^*$ , and then we consider every production of the form  $A \rightarrow X_1 X_2 \dots X_n$ , where  $A$  is not yet marked. If every  $X_i$  (for  $1 \leq i \leq n$ ) is either a terminal or a nonterminal that is already marked, then we mark  $A$  (nonterminal on the left side of the production).

We repeat this process until no new nonterminals can be marked. The nonterminals that are not marked are those not deriving to  $w$  in  $T^*$ . After identifying the nonterminals that do not derive to  $w$  in  $T^*$ , we eliminate all productions containing these nonterminals in order to obtain a grammar that does not contain any nonterminals that do not derive in  $T^*$ . The algorithm for identifying as well as eliminating the nonterminals that do not derive to  $w$  in  $T^*$  is given below:

```

Input:   G = (V, T, P, S)
Output:  G1 = (V1, T, P1, S)
        { where V1 is the set of nonterminals deriving to w in T*, we maintain V1 old and V1 new to continue
          iterations, and P1 is the set of productions that do not contain nonterminals that do not derive to w in T
          * }

```

Let  $U$  be the set of nonterminals that are not capable of deriving to  $w$  in  $T^*$ .

Then,

```

begin
  V1 old = ∅
  V1 new = ∅
  for every production of the form A → w do
    V1 new = V1 new ∪ { A }
  while (V1 old ≠ V1 new) do
    begin
      temp = V - V1 new
      V1 old = V1 new
      For every A in temp do
        for every A-production of the form A → X1 X2 ... Xn in P do
          if each Xi is either in T or in V1 old, then
            begin
              V1 new = V1 new ∪ { A }
              break;
            end
          end
        end
      V1 = V1 new
      U = V - V1
      for every production in P do
        if it does not contain a member of U then
          add the production to P1
    end

```

end

If  $S$  is itself a useless nonterminal, then the reduced grammar is a ‘null’ grammar.

When identifying the grammar symbols that are not used in the derivation of any  $w$  in  $L(G)$ , terminals as well as nonterminals must be tested. A simple, iterative algorithm can be used to identify those grammar symbols that are not used in the derivation of any  $w$  in  $L(G)$ : we start with  $S$ -productions and mark every grammar symbol  $X$  on the right side of every  $S$ -production. We then consider every production of the form  $A \rightarrow X_1 X_2 \dots X_n$ , where  $A$  is an already-marked nonterminal; and we mark every  $X$  on the right side of these productions. We repeat this process until no new nonterminals can be marked. We do not mark any terminals or nonterminals not used in the derivation of any  $w$  in  $L(G)$ . After identifying the terminals and nonterminals not used in the derivation of any  $w$  in  $L(G)$ , we eliminate all productions containing them; thus, we obtain a grammar that does not contain any useless symbols-hence, a reduced grammar.

The algorithm for identifying as well as eliminating grammar symbols that are not used in the derivation of any  $w$  in  $L(G)$  is given below:

```

Input:    $G_1 = (V_1, T, P_1, S)$ 
        { The grammar obtained after elimination of the nonterminals not deriving to  $w$  in  $T^*$  }

Output:   $G_2 = (V_2, T_2, P_2, S)$ 
        { where  $V_2$  is the set of nonterminals used in derivation of some  $w$  in  $L(G)$ , and  $T_2$  is set of terminals
          used in the derivation of some  $w$  in  $L(G)$ , and  $P_2$  is set of productions containing the members of  $V_2$ 
          and  $T_2$  only. We maintain  $V_2$  old and  $V_2$  new to continue iterations }

begin
     $T_2 = \emptyset$ 
     $V_2$  old =  $\emptyset$ 
     $P_2 = \emptyset$ 
     $V_2$  new = {  $S$  }
    While ( $V_2$  old #  $V_2$  new) do
        begin
            temp =  $V_2$  new -  $V_2$  old
             $V_2$  old =  $V_2$  new for every  $A$  in temp do
                for every  $A$ -production of the form  $A \rightarrow X_1 X_2 \dots X_n$  in  $P_1$  do
                    for each  $X_i$  ( $1 \leq i \leq n$ ) do
                        begin
                            if ( $X_i$  is in  $V_2$  old) then
                                 $V_2$  new =  $V_2$  new  $\cup$  {  $X_i$  }
                            if ( $X_i$  is in  $T$ ) then
                                 $T_2 = T_2 \cup \{ X_i \}$ 
                        end
                    end
                end
             $V_2 = V_2$  new
            temp1 =  $V_1 - V_2$ 
            temp2 =  $T_1 - T_2$ 
            for every production in  $P_1$  do add the production to  $P_2$  if it does
                not contain a member of temp1 as well as temp2
             $G_2 = (V_2, T_2, P_2, S)$ 
        end
    end
end

```

### **EXAMPLE 3.1**

---

Find the reduced grammar equivalent to CFG

$$G = (\{S, A, B, C\} \mid \{a, b, d\}, S, P)$$

where  $P$  contains

$$\begin{aligned} S &\rightarrow AC \mid SB \\ A &\rightarrow bASC \mid a \\ B &\rightarrow aSB \mid bbC \\ C &\rightarrow Bc \mid ad \end{aligned}$$


---

Since the productions  $A \rightarrow a$  and  $C \rightarrow ad$  exist in form  $A \rightarrow w$ , nonterminals  $A$  and  $C$  are derivable to  $w$  in  $T^*$ . The production  $S \rightarrow AC$  also exists, the right side of which contains the nonterminals  $A$  and  $C$ , which are derivable to  $w$  in  $T^*$ . Hence,  $S$  is also derivable to  $w$  in  $T^*$ . But since the right side of both of the  $B$ -productions contain  $B$ , the nonterminal  $B$  is not derivable to  $w$  in  $T^*$ .

Hence,  $B$  can be eliminated from the grammar, and the following grammar is obtained:

$$G_1 = (\{S, A, C\} \cup \{a, b, d\}, S, P_1)$$

where  $P_1$  contains

$$\begin{aligned} S &\rightarrow AC \\ A &\rightarrow bASC \mid a \\ C &\rightarrow ad \end{aligned}$$

Since the right side of the  $S$ -production of this grammar contains the nonterminals  $A$  and  $C$ ,  $A$  and  $C$  will be used in the derivation of some  $w$  in  $L(G)$ . Similarly, the right side of the  $A$ -production contains  $bASC$  and  $a$ ; hence, the terminals  $a$  and  $b$  will be used. The right side of the  $C$ -production contains  $ad$ , so terminal  $d$  will also be useful. Therefore, every terminal, as well as the nonterminal in  $G_1$ , is useful. So the reduced grammar is:

$$G_1 = (\{S, A, C\} \cup \{a, b, d\}, S, P_1)$$

where  $P_1$  contains

$$\begin{aligned} S &\rightarrow AC \\ A &\rightarrow bASC \mid a \\ C &\rightarrow ad \end{aligned}$$

### 3.2.5 Useless Grammar Symbols

A grammar symbol is a useless grammar symbol if it does not satisfy either of the following conditions:

$$\begin{aligned} Xw, \text{ where } w \text{ is in } T^* \\ S \alpha X \beta \text{ where } w \text{ is in } L(G) \end{aligned}$$

That is, a grammar symbol  $X$  is useless if it does not derive to terminal strings. And even if it does derive to a string of terminals,  $X$  is a useless grammar symbol if it does not occur in a derivation sequence of any  $w$  in  $L(G)$ . For example, consider the following grammar:

$$\begin{aligned} S &\rightarrow aB \mid bX \\ A &\rightarrow BAd \mid bSX \mid q \\ B &\rightarrow aSB \mid bBX \\ X &\rightarrow SBD \mid aBx \mid ad \end{aligned}$$

First, we find those nonterminals that do not derive to the string of terminals so that they can be separated out. The nonterminals  $A$  and  $X$  directly derive to the string of terminals because the production  $A \rightarrow q$  and  $X \rightarrow ad$  exist in a grammar. There also exists a production  $S \rightarrow bX$ , where  $b$  is a terminal and  $X$  is a nonterminal, which is already known to derive to a string of terminals. Therefore,  $S$  also derives to string of terminals, and the nonterminals that are capable of deriving to a string of terminals are:  $S$ ,  $A$ , and  $X$ .  $B$  ends up being a useless nonterminal; and therefore, the productions containing  $B$  can be eliminated from the given grammar to obtain the grammar given below:

$$\begin{aligned} S &\rightarrow bX \\ A &\rightarrow bSX \mid q \\ X &\rightarrow ad \end{aligned}$$

We next find in the grammar obtained those terminals and nonterminals that occur in the derivation sequence of some  $w$  in  $L(G)$ . Since every derivation sequence starts with  $S$ ,  $S$  will always occur in the derivation sequence of every  $w$  in  $L(G)$ . We then consider those productions whose left-hand side is  $S$ , such as  $S \rightarrow bX$ , since the right side of this

production contains a terminal  $b$  and a nonterminal  $X$ . We conclude that the terminal  $b$  will occur in the derivation sequence, and a nonterminal  $X$  will also occur in the derivation sequence. Therefore, we next consider those productions whose left-hand side is a nonterminal  $X$ . The production is  $X \rightarrow ad$ . Since the right side of this production contains terminals  $a$  and  $d$ , these terminals will occur in the derivation sequence. But since no new nonterminal is found, we conclude that the nonterminals  $S$  and  $X$ , and the terminals  $a$ ,  $b$ , and  $d$  are the grammar symbols that can occur in the derivation sequence. Therefore, we conclude that the nonterminal  $A$  will be a useless nonterminal, even though it derives to the string of terminals. So we eliminate the productions containing  $A$  to obtain a reduced grammar, given below:

$$\begin{aligned} S &\rightarrow bX \\ X &\rightarrow ad \end{aligned}$$

---

#### EXAMPLE 3.2

Consider the following grammar, and obtain an equivalent grammar containing no useless grammar symbols.

$$\begin{aligned} A &\rightarrow xyz \mid Xyzz \\ X &\rightarrow Xz \mid xYx \\ Y &\rightarrow yYy \mid XZ \\ Z &\rightarrow Zy \mid z \end{aligned}$$


---

Since  $A \rightarrow xyz$  and  $Z \rightarrow z$  are the productions of the form  $A \rightarrow w$ , where  $w$  is in  $T^*$ , nonterminals  $A$  and  $Z$  are capable of deriving to  $w$  in  $T^*$ . There are two  $X$ -productions:  $X \rightarrow Xz$  and  $X \rightarrow xYx$ . The right side of these productions contain nonterminals  $X$  and  $Y$ , respectively. Similarly, there are two  $Y$ -productions:  $Y \rightarrow yYy$  and  $Y \rightarrow XZ$ . The right side of these productions contain nonterminals  $Y$  and  $X$ , respectively. Hence, both  $X$  and  $Y$  are not capable of deriving to  $w$  in  $T^*$ . Therefore, by eliminating the productions containing  $X$  and  $Y$ , we get:

$$\begin{aligned} A &\rightarrow xyz \\ Z &\rightarrow Zy \mid z \end{aligned}$$

Since  $A$  is a start symbol, it will always be used in the derivation of every  $w$  in  $L(G)$ . And since  $A \rightarrow xyz$  is a production in the grammar, the terminals  $x$ ,  $y$ , and  $z$  will also be used in the derivation. But no nonterminal  $Z$  occurs on the right side of the  $A$ -production, so  $Z$  will not be used in the derivation of any  $w$  in  $L(G)$ . Hence, by eliminating the productions containing nonterminal  $Z$ , we get:

$$A \rightarrow xyz$$

which is a grammar containing no useless grammar symbols.

---

#### EXAMPLE 3.3

Find the reduced grammar that is equivalent to the CFG given below:

$$\begin{aligned} S &\rightarrow aC \mid SB \\ A &\rightarrow bSCa \\ B &\rightarrow aSB \mid bBC \\ C &\rightarrow aBC \mid ad \end{aligned}$$


---

Since  $C \rightarrow ad$  is the production of the form  $A \rightarrow w$ , where  $w$  is in  $T^*$ , nonterminal  $C$  is capable of deriving to  $w$  in  $T^*$ . The production  $S \rightarrow aC$  contains a terminal  $a$  on the right side as well as a nonterminal  $C$  that is known to be capable

of deriving to  $w$  in  $T^*$ .

Hence, nonterminal  $S$  is also capable of deriving to  $w$  in  $T^*$ . The right side of the production  $A \rightarrow bSCa$  contains the nonterminals  $S$  and  $C$ , which are known to be capable of deriving to  $w$  in  $T^*$ . Hence, nonterminal  $A$  is also capable of deriving to  $w$  in  $T^*$ . There are two  $B$ -productions:  $B \rightarrow aSB$  and  $B \rightarrow bBC$ . The right side of these productions contain the nonterminals  $S$ ,  $B$ , and  $C$ ; and even though  $S$  and  $C$  are known to be capable of deriving to  $w$  in  $T^*$ , nonterminal  $B$  is not. Hence, by eliminating the productions containing  $B$ , we get:

$$\begin{aligned}S &\rightarrow aC \\A &\rightarrow bSCa \\C &\rightarrow ad\end{aligned}$$

Since  $S$  is a start symbol, it will always be used in the derivation of every  $w$  in  $L(G)$ . And since  $S \rightarrow aC$  is a production in the grammar, terminal  $a$  as well as nonterminal  $C$  will also be used in the derivation. But since a nonterminal  $C$  occurs on the right side of the  $S$ -production, and  $C \rightarrow ad$  is a production, terminal  $d$  will be used along with terminal  $a$  in the derivation. A nonterminal  $A$ , though, occurs nowhere in the right side of either the  $S$ -production or the  $C$ -production; it will not be used in the derivation of any  $w$  in  $L(G)$ . Hence, by eliminating the productions containing nonterminal  $A$ , we get:

$$\begin{aligned}S &\rightarrow aC \\C &\rightarrow ad\end{aligned}$$

which is a reduced grammar equivalent to the given grammar, but it contains no useless grammar symbols.

---

#### EXAMPLE 3.4

---

Find the useless symbols in the following grammar, and modify the grammar so that it has no useless symbols.

$$\begin{aligned}S &\rightarrow 0 \mid A \\A &\rightarrow AB \\B &\rightarrow 1\end{aligned}$$

---

Since  $S \rightarrow 0$  and  $B \rightarrow 1$  are productions of the form  $A \rightarrow w$ , where  $w$  is in  $T^*$ , the nonterminals  $S$  and  $B$  are capable of deriving to  $w$  in  $T^*$ . The production  $A \rightarrow AB$  contains the nonterminals  $A$  and  $B$  on the right side; and even though  $B$  is known to be capable of deriving to  $w$  in  $T^*$ , nonterminal  $A$  is not capable of deriving to  $w$  in  $T^*$ . Therefore, by eliminating the productions containing  $A$ , we get:

$$\begin{aligned}S &\rightarrow 0 \\B &\rightarrow 1\end{aligned}$$

Since  $S$  is a start symbol, it will always be used in the derivation of any  $w$  in  $L(G)$ . And because  $S \rightarrow 0$  is a production in the grammar, terminal  $0$  will also be used in the derivation. But nonterminal  $B$  does not occur anywhere in the right side of the  $S$ -production, it will not be used in the derivation of any  $w$  in  $L(G)$ . Hence, by eliminating the productions containing nonterminal  $B$ , we get:

$$S \rightarrow 0$$

which is a grammar equivalent to the given grammar and contains no useless grammar symbols.

---

#### EXAMPLE 3.5

---

Find the useless symbols in the following grammar, and modify the grammar to obtain one that has no useless symbols.

$$S \rightarrow AB \mid CA$$

$B \rightarrow BC \mid AB$

$A \rightarrow a$

$C \rightarrow aB \mid b$

---

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

$\xrightarrow{*}$

end

---

**EXAMPLE 3.6**

---

Consider the following grammar and identify the nullable nonterminals.

$$\begin{aligned}S &\rightarrow ACB \mid CbB \mid Ba \\A &\rightarrow da \mid BC \\B &\rightarrow gC \mid \epsilon \\C &\rightarrow ha \mid \epsilon\end{aligned}$$

---

By applying the above algorithm, the results after each iteration are shown below:

Initially:

$$\begin{aligned}N_{\text{old}} &= \emptyset \\N_{\text{new}} &= \emptyset\end{aligned}$$

After the first execution of the *for* loop:

$$\begin{aligned}N_{\text{old}} &= \emptyset \\N_{\text{new}} &= \{ B, C \}\end{aligned}$$

After the first iteration of the *while* loop:

$$\begin{aligned}N_{\text{old}} &= \{ B, C \} \\N_{\text{new}} &= \{ B, C, A \}\end{aligned}$$

After the second iteration of the *while* loop:

$$\begin{aligned}N_{\text{old}} &= \{ B, C, A \} \\N_{\text{new}} &= \{ B, C, A, S \}\end{aligned}$$

After the third iteration of the *while* loop:

$$\begin{aligned}N_{\text{old}} &= \{ B, C, A, S \} \\N_{\text{new}} &= \{ B, C, A, S \}\end{aligned}$$

Therefore,  $N = \{ S, A, B, C \}$ ; and hence, all the nonterminals of the grammar are nullable.

### 3.2.7 Eliminating $\epsilon$ -Productions

Given a grammar  $G$  that contains  $\epsilon$ -productions, if  $L(G)$  does not contain  $\epsilon$ , then it is possible to eliminate all  $\epsilon$ -productions in the given grammar  $G$ . Whereas, if  $L(G)$  contains  $\epsilon$ , then elimination of all  $\epsilon$ -productions from  $G$  gives a grammar  $G$  in which  $L(G_1) = L(G) - \{ \epsilon \}$ . To eliminate the  $\epsilon$ -productions from a grammar, we use the following technique.

If  $A \rightarrow \epsilon$  is an  $\epsilon$ -production to be eliminated, then we look for all those productions in the grammar whose right side contains  $A$ , and we replace each occurrence of  $A$  in these productions. Thus, we obtain the non- $\epsilon$ -productions to be added to the grammar so that the language's generation remains the same. For example, consider the following grammar:

$$S \rightarrow aA$$

$$A \rightarrow b \mid \epsilon$$

To eliminate  $A \rightarrow \epsilon$  from the above grammar, we replace  $A$  on the right side of the production  $S \rightarrow aA$  and obtain a non- $\epsilon$ -production,  $S \rightarrow a$ , which is added to the grammar as a substitute in order to keep the language generated by the grammar the same. Therefore, the  $\epsilon$ -free grammar equivalent to the given grammar is:

$$\begin{aligned} S &\rightarrow aA \mid a \\ A &\rightarrow b \end{aligned}$$

#### **EXAMPLE 3.7**

---

Consider the following grammar, and eliminate all the  $\epsilon$ -productions from the grammar without changing the language generated by the grammar.

$$\begin{aligned} S &\rightarrow ABAC \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bB \mid \epsilon \\ C &\rightarrow c \end{aligned}$$


---

To eliminate  $A \rightarrow \epsilon$  from this grammar, the non- $\epsilon$ -productions to be added are obtained as follows: the list of the productions containing  $A$  on the right-hand side is:

$$\begin{aligned} S &\rightarrow ABAC \\ A &\rightarrow aA \end{aligned}$$

Replace each occurrence of  $A$  in each of these productions in order to obtain the non- $\epsilon$ -productions to be added to the grammar. The list of these productions is:

$$\begin{aligned} S &\rightarrow BAC \mid ABC \mid BC \\ A &\rightarrow a \end{aligned}$$

Add these productions to the grammar, and eliminate  $A \rightarrow \epsilon$  from the grammar. This gives us the following grammar:

$$\begin{aligned} S &\rightarrow ABAC \mid BAC \mid ABC \mid BC \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid \epsilon \\ C &\rightarrow c \end{aligned}$$

To eliminate  $B \rightarrow \epsilon$  from the grammar, the non- $\epsilon$ -productions to be added are obtained as follows. The productions containing  $B$  on the right-hand side are:

$$\begin{aligned} S &\rightarrow ABAC \mid BAC \mid ABC \mid BC \\ B &\rightarrow bB \end{aligned}$$

Replace each occurrence of  $B$  in these productions in order to obtain the non- $\epsilon$ -productions to be added to the grammar. The list of these productions is:

$$\begin{aligned} S &\rightarrow AAC \\ S &\rightarrow AC \end{aligned}$$

$$S \rightarrow C$$

$$\begin{array}{c} \curvearrowleft \quad \curvearrowright \\ B \rightarrow b \end{array}$$

Add these productions to the grammar, and eliminate  $A \rightarrow \epsilon$  from the grammar in order to obtain the following:

$$\begin{aligned} S &\rightarrow ABAC \mid BAC \mid ABC \mid BC \mid AAC \mid AC \mid C \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \\ C &\rightarrow c \end{aligned}$$

---

#### EXAMPLE 3.8

Consider the following grammar and eliminate all the  $\epsilon$ -productions without changing the language generated by the grammar.

$$\begin{aligned} S &\rightarrow AaA \\ A &\rightarrow Sb \mid bCC \mid \epsilon \\ C &\rightarrow CC \mid abb \end{aligned}$$


---

To eliminate  $A \rightarrow \epsilon$  from the grammar, the non- $\epsilon$ -productions to be added are obtained as follows: the list of productions containing  $A$  on right is:

$$S \rightarrow AaA$$

Replace each occurrence of  $A$  in this production to obtain the non- $\epsilon$ -productions to be added to the grammar. This are:

$$S \rightarrow aA \mid Aa \mid a$$

Add these productions to the grammar, and eliminate  $A \rightarrow \epsilon$  from the grammar to obtain the following:

$$\begin{aligned} S &\rightarrow AaA \mid aA \mid Aa \mid a \\ A &\rightarrow Sb \mid bCC \\ C &\rightarrow CC \mid abb \end{aligned}$$

### 3.2.8 Eliminating Unit Productions

A production of the form  $A \rightarrow B$ , where  $A$  and  $B$  are both nonterminals, is called a "unit production". Unit productions in the grammar increase the cost of derivations. The following algorithm can be used to eliminate unit productions from the grammar:

While there exist a unit production  $A \rightarrow B$  in the grammar do

{

  select a unit production  $A \rightarrow B$  such that there exists  
  at least one nonunit production

$B \rightarrow \alpha$

  for every nonunit production  $B \rightarrow \alpha$  do  
    add production  $A \rightarrow \alpha$  to the grammar  
    eliminate  $A \rightarrow B$  from the grammar

}

---

#### EXAMPLE 3.9

Given the grammar shown below, eliminate all the unit productions from the grammar.

$S \rightarrow AB$   
 $A \rightarrow a$   
 $B \rightarrow C \mid b$   
 $C \rightarrow D$   
 $D \rightarrow E$   
 $E \rightarrow a$

$B \rightarrow C$   
 $C \rightarrow D$   
 $D \rightarrow E$

---

$S \rightarrow AB$   
 $A \rightarrow a$   
 $B \rightarrow a \mid b$   
 $C \rightarrow a$   
 $D \rightarrow a$   
 $E \rightarrow a$

$S \rightarrow AE$   
 $A \rightarrow a$   
 $B \rightarrow a \mid b$

grammar by replacing  $A \rightarrow A\alpha_1 | \beta$  with the productions  $A \rightarrow \beta B$  and  $B \rightarrow \alpha_1 | \dots | \alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$ . In general, if a grammar contains productions:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

then the left-recursion can be eliminated by adding the following productions in place of the ones above.

$$\begin{aligned} A &\rightarrow \beta_1 B | \beta_2 B | \dots | \beta_n B \\ B &\rightarrow \alpha_1 B | \alpha_2 B | \dots | \alpha_m B | \epsilon \end{aligned}$$

---

#### **EXAMPLE 3.10**

Consider the following grammar:

$$\begin{aligned} S &\rightarrow aBDh \\ B &\rightarrow Bb | c \\ D &\rightarrow EF \\ E &\rightarrow g | \epsilon \\ F &\rightarrow f | \epsilon \end{aligned}$$


---

The grammar is left-recursive because it contains a pair of productions,  $B \rightarrow Bb | c$ . To eliminate the left-recursion from the grammar, replace this pair of productions with the following productions:

$$\begin{aligned} B &\rightarrow cC \\ C &\rightarrow bC | \epsilon \end{aligned}$$

Therefore, the grammar that we get after the elimination of left-recursion is:

$$\begin{aligned} S &\rightarrow aBDh \\ B &\rightarrow cC \\ C &\rightarrow bC | \epsilon \\ D &\rightarrow EF \\ E &\rightarrow g | \epsilon \\ F &\rightarrow f | \epsilon \end{aligned}$$

---

#### **EXAMPLE 3.11**

Consider the following grammar:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow Ad | Ae | aB | aC \\ B &\rightarrow bBC | f \\ C &\rightarrow g \end{aligned}$$


---

The grammar is left-recursive because it contains the productions  $A \rightarrow Ad | Ae | aB | aC$ . To eliminate the left-recursion from the grammar, replace these productions by the following productions:

$$A \rightarrow aBD | aCD$$

$$D \rightarrow dD \mid eD \mid \epsilon$$

Therefore, the resulting grammar after the elimination of left-recursion is:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow aBD \mid aCD \\ D &\rightarrow dD \mid eD \mid \epsilon \\ B &\rightarrow bBc \mid f \\ C &\rightarrow g \end{aligned}$$

---

**EXAMPLE 3.12**

---

Consider the following grammar:

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow L, S \mid S \end{aligned}$$

---

The grammar is left-recursive because it contains the productions  $L \rightarrow L, S \mid S$ . To eliminate the left-recursion from the grammar, replace these productions by the following productions:

$$\begin{aligned} L &\rightarrow SA \\ A &\rightarrow SA \mid \epsilon \end{aligned}$$

Therefore, after the elimination of left-recursion, we get:

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow SA \\ A &\rightarrow SA \mid \epsilon \end{aligned}$$

### 3.3 REGULAR GRAMMAR

Regular grammar is a context-free grammar in which every production is restricted to one of the following forms:

1.  $A \rightarrow aB$ , or
2.  $A \rightarrow w$ , where  $A$  and  $B$  are the nonterminals,  $a$  is a terminal symbol, and  $w$  is in  $T^*$ .

The  $\in$ -productions are permitted as a special case when  $L(G)$  contains  $\in$ . This grammar is called "regular grammar," because if the format of every production in CFG is restricted to  $A \rightarrow aB$  or  $A \rightarrow a$ , then the grammar can specify only regular sets. Hence, a finite automata exists that accepts  $L(G)$ , if  $G$  is regular grammar. Given a regular grammar  $G$ , a finite automata accepting  $L(G)$  can be obtained as follows:

1. The number of states of the automata will be equal to the number of nonterminals of the grammar plus one; that is, there will be a state corresponding to every nonterminal of the grammar. And one more state will be there, which will be the final state of the automata. The state corresponding to the start symbol of the grammar will be the initial state of the automata. If  $L(G)$  contains  $\in$ , then make the start state also the final state.
2. The transitions in the automata can be obtained as follows:

for every production  $A \rightarrow aB$  do

$$\text{make } \delta(A, a) = B$$

for every production of the form  $A \rightarrow a$  do

$$\text{make } \delta(A, a) = \text{final state}$$

#### EXAMPLE 3.13

Consider the regular grammar shown below and the transition diagram of the automata, shown in [Figure 3.7](#), that accepts the language generated by the grammar.

$$\begin{aligned} S &\rightarrow 0A \mid 1B \mid 0 \mid 1 \\ A &\rightarrow 0S \mid 1B \mid 1 \\ B &\rightarrow 0A \mid 1S \end{aligned}$$

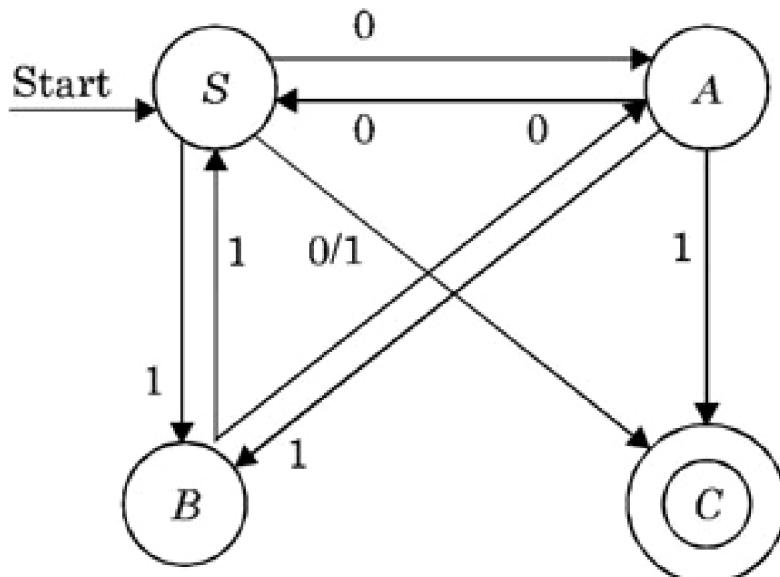




Figure 3.8

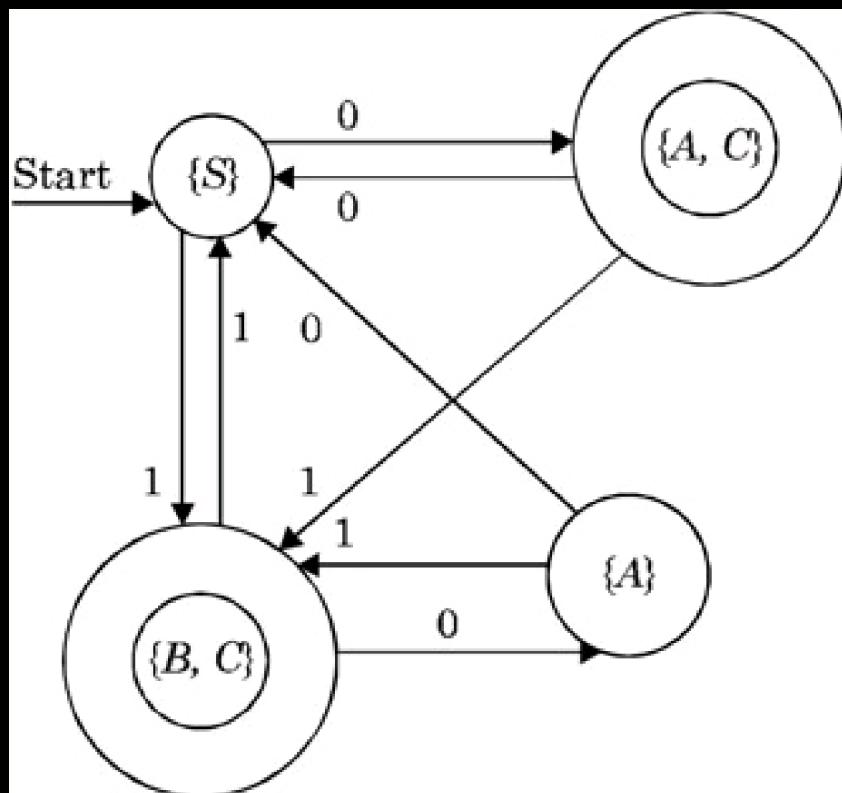
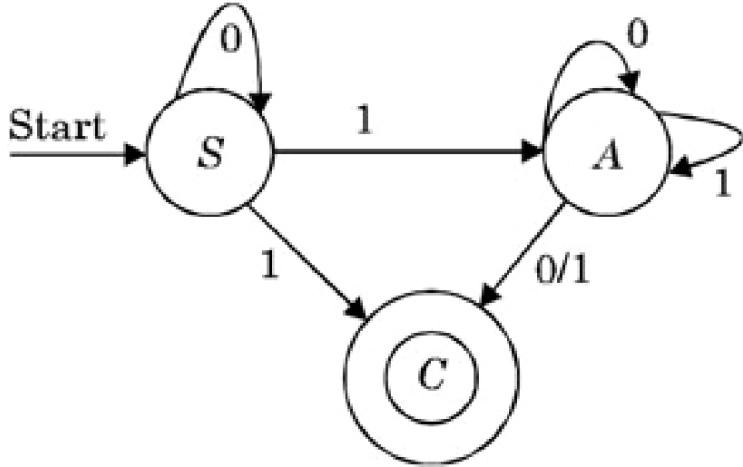


Figure 3.7

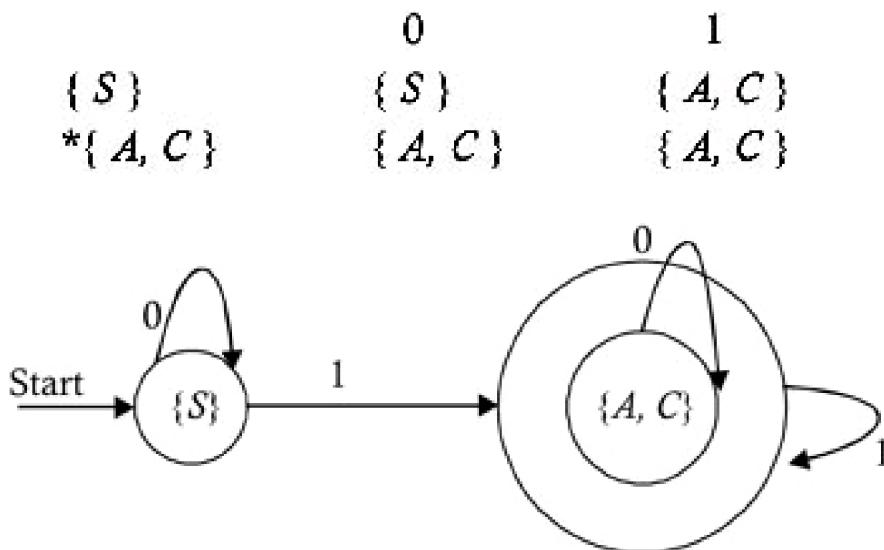
$$\begin{aligned} S &\rightarrow 0S \mid 1A \mid 1 \\ A &\rightarrow 0A \mid 1A \mid 0 \mid 1 \end{aligned}$$

Figure 3.9



**Figure 3.9:** Non-deterministic automata.

This is a non-deterministic automata. Its deterministic equivalent can be obtained as follows, and the transition diagram is shown in [Figure 3.10](#).

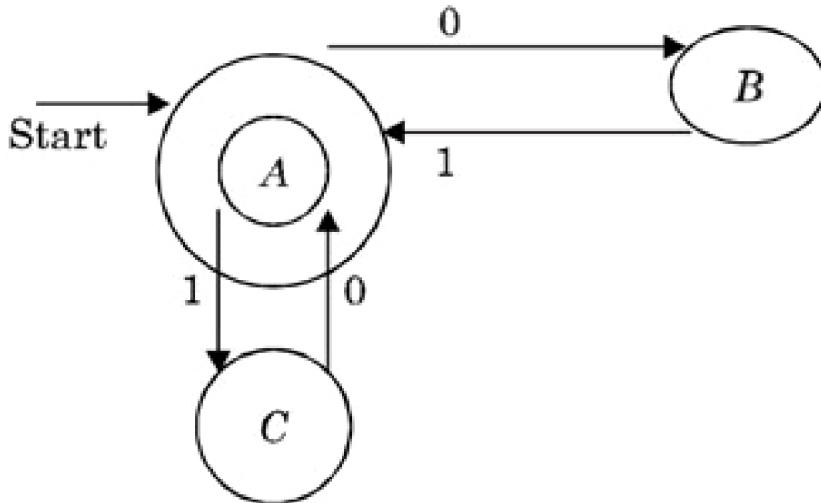


**Figure 3.10:** Transition diagram for deterministic automata equivalent shown in [Figure 3.9](#).

Given a finite automata  $M$ , a regular grammar  $G$  that generates  $L(M)$  can be obtained as follows:

1. Associate suitable variables like  $A, B, C$ , etc, with the states of the automata. The labels of the states can also be used as variable names.
2. Obtain the productions of the grammar as follows. If  $\delta(A, a) = B$ , then add a production  $A \rightarrow aB$  to the list of productions of the grammar. If  $B$  is a final state, then add either  $A \rightarrow a$  or  $B \rightarrow \epsilon$ , to the grammar's list of productions.
3. The variable associated with the initial state of the automata is the start symbol of the grammar.

For example consider the automata shown in [Figure 3.11](#).



**Figure 3.11:** Regular-grammar automata.

The regular grammar that generates the language accepted by the automata shown in [Figure 3.11](#) will have the following productions:

$$\begin{aligned}A &\rightarrow 0B \mid 1C \mid \epsilon \\B &\rightarrow 1A \\C &\rightarrow 0A\end{aligned}$$

or

$$\begin{aligned}A &\rightarrow 0B \mid 1C \\B &\rightarrow 1A \mid 1 \\C &\rightarrow 0A \mid 0\end{aligned}$$

where  $A$  is the start symbol. Both the grammars are same, but the first one contains  $\epsilon$ -productions, whereas the second is  $\epsilon$ -free.

#### **EXAMPLE 3.14**

---

Find out whether the following grammar generates the same language.

G1:

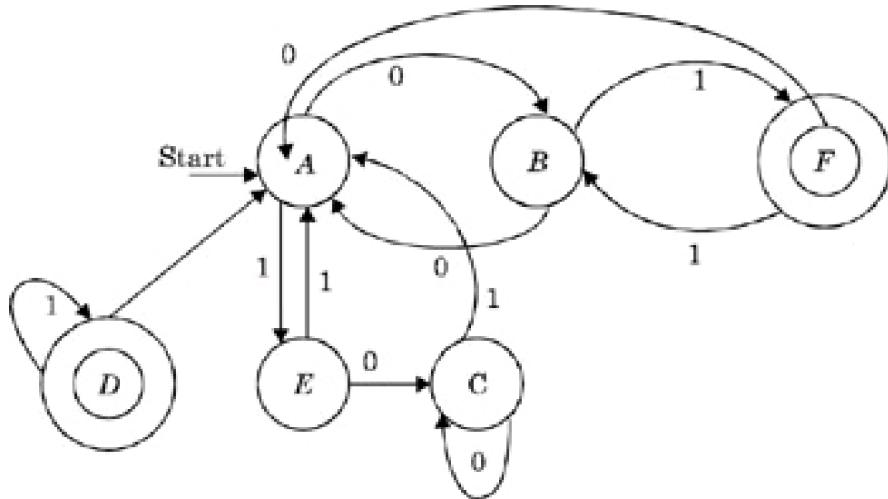
$$\begin{aligned}A &\rightarrow 0B \mid 1E \\B &\rightarrow 0A \mid 1F \mid \epsilon \\C &\rightarrow 0C \mid 1A \\D &\rightarrow 0A \mid 1D \mid \epsilon \\E &\rightarrow 0C \mid 1A \\F &\rightarrow 0A \mid 1B \mid \epsilon \\&\text{where } A \text{ is a start symbol}\end{aligned}$$

G2:

$$\begin{aligned}X &\rightarrow 0Y \mid 0 \mid 1Z \\Y &\rightarrow 0X \mid 1Y \mid 1 \\Z &\rightarrow 0Z \mid 1X \\&\text{where } X \text{ is a start symbol}\end{aligned}$$

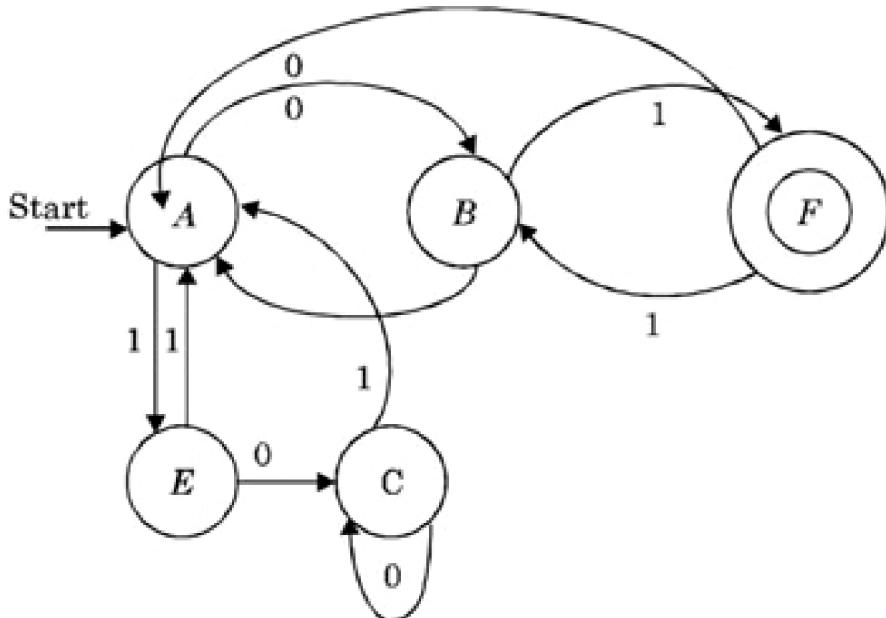

---

Since the grammars  $G_1$  and  $G_2$  are the regular grammars,  $L(G_1) = L(G_2)$  if the minimal state automata accepting  $L(G_1)$ , and the minimal state automata accepting  $L(G_2)$  are identical. The transition diagram of the automata accepting  $L(G_1)$  is shown in [Figure 3.12](#).



**Figure 3.12:** Transition diagram of automata that accepts  $L(G_1)$ .

The automata is deterministic. Hence, to minimize, it we proceed as follows. Since state  $D$  is an unreachable state, eliminate it first. So, after eliminating state  $D$ , we get the transition diagram shown in [Figure 3.13](#).



**Figure 3.13:** Transition diagram of automata after removal of state D.

We then identify the nondistinguishable states of the automata shown in [Figure 3.13](#), as follows. Initially, we have two groups:

**Group I**

**Group II**

**A, E, C**

**B, F**

Since

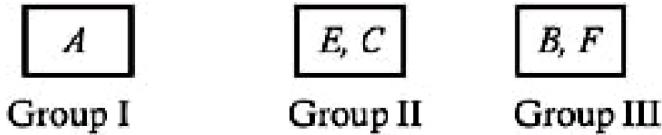
$$\delta(A, 0) = B$$

~~$\delta(E, 0) = C$~~

$$v_i \cup v_j = v$$

$$\delta(C, 0) = C$$

state  $B$  is distinguishable from rest of the members of Group I. Hence, we divide Group I into two groups—one containing  $A$ , and other containing  $E$  and  $C$ , as shown below:



Since

$$\delta(E, 0) = C$$

$$\delta(C, 0) = C$$

partitioning of Group II is not possible, because the transitions from all the members of Group II only go to Group II.

Similarly:

$$\delta(E, 1) = A$$

$$\delta(C, 1) = A$$

Partitioning of Group II is not possible, because the transitions from all the members of Group II only go to Group I. And since:

$$\delta(B, 0) = A$$

$$\delta(F, 0) = A$$

partitioning of Group III is not possible, because the transitions from all the members of Group III only go to Group I.

Similarly:

$$\delta(B, 1) = F$$

$$\delta(F, 1) = B$$

Partitioning of Group III is not possible, because the transitions from all the members of Group III only go to Group III. Hence, states  $E$  and  $C$  are nondistinguishable states. States  $B$  and  $F$  are also nondistinguishable states. Therefore, if we merge  $E$  and  $C$  to form a state  $E_1$ , and we merge  $B$  and  $F$  to form  $B_1$ , we get the automata shown in [Figure 3.14](#).

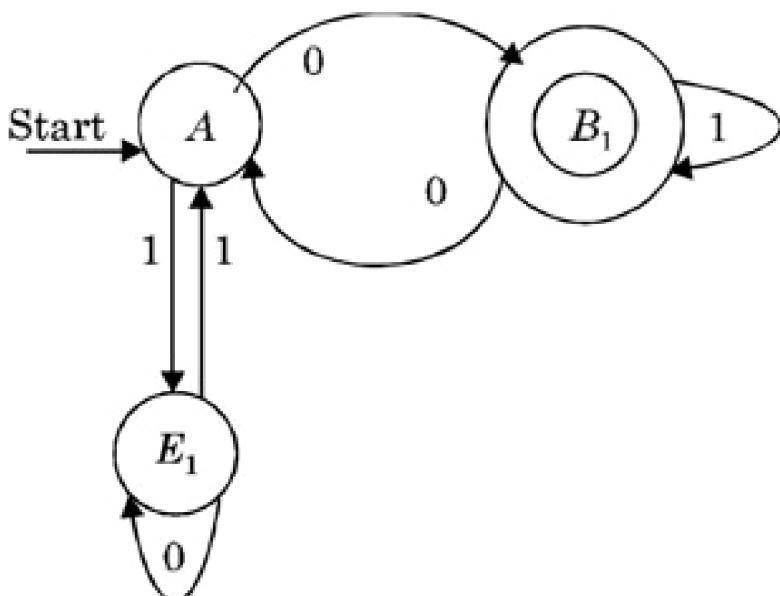
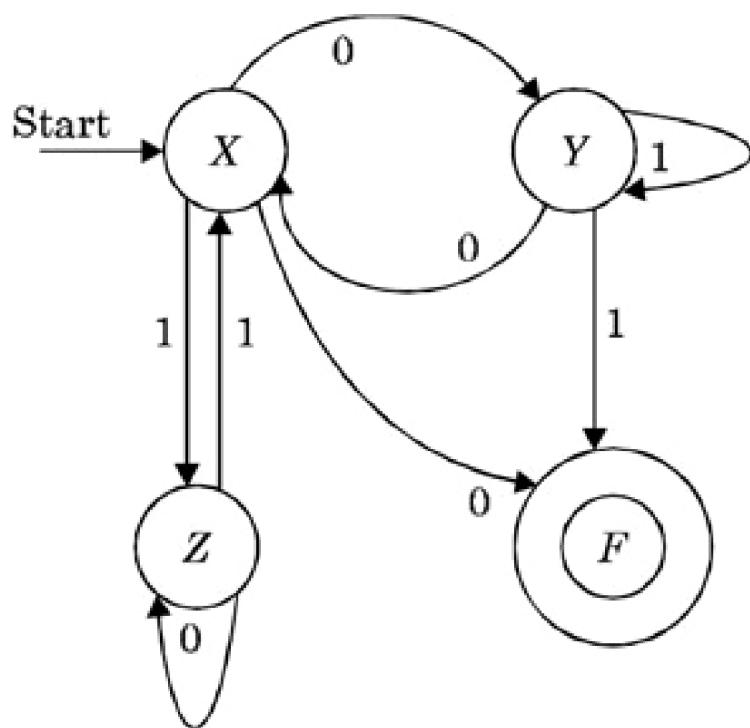


Figure 3.14: Transition diagram for the automata that results from merged states.

Since no dead states exist in the automata shown in [Figure 3.14](#), it is a minimal state automata that accepts  $L(G_1)$ . The transition diagram of the non-deterministic automata that accepts  $L(G_2)$  is shown in [Figure 3.15](#).



**Figure 3.15:** Non-deterministic automata that accepts  $L(G_2)$ .

Its equivalent deterministic automata is as follows, and the transition diagram is shown in [Figure 3.16](#).

0	1
<hr/>	
{X}	{Y, F}
*{Y, F}	{X}
{Z}	{Z}
	{X}

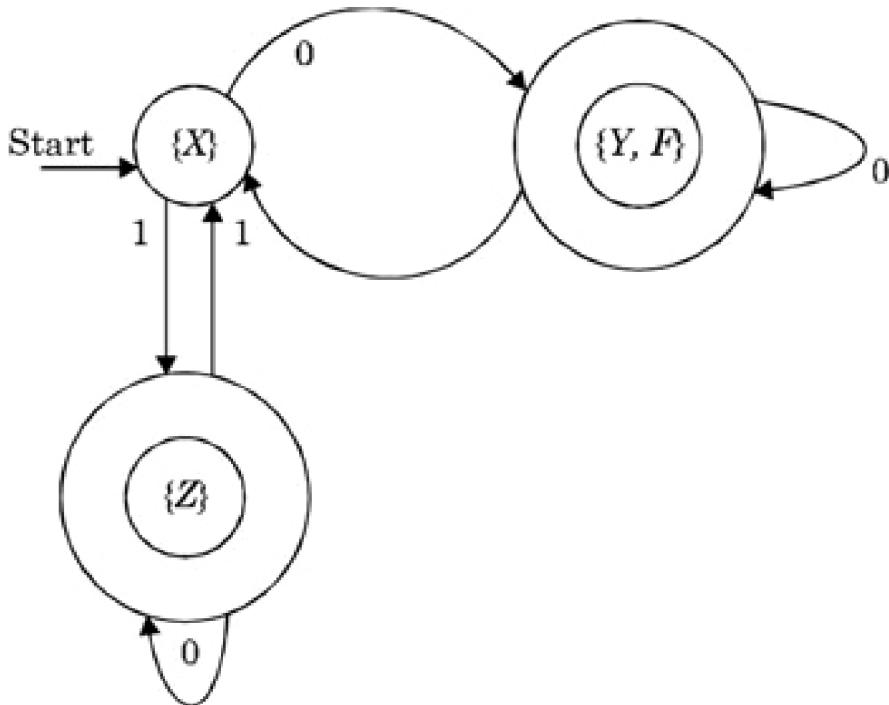


Figure 3.16: Transition diagram of the equivalent deterministic automata for Figure 3.15.

This automata does not contain unreachable, nondistinguishable states or dead states. Hence, it is a minimal state automata accepting  $L(G_2)$ , and since it is identical to the minimal state automata accepting  $L(G_1)$ ,  $L(G_2) = L(G_1)$ ; and therefore,  $G_1$  and  $G_2$  generate the same language.

### Obtaining a Regular Expression from the Regular Grammar

Given a regular grammar  $G$ , a regular expression that specifies  $L(G)$  can be directly obtained as follows:

1. Replace the " $\rightarrow$ " symbols in the grammar's productions with "=" symbols to get a set of equations.
2. Solve the set of equations obtained above to obtain the value of the variable  $S$ , where  $S$  is the start symbol of the grammar. The result is the regular expression specifying  $L(G)$ .

For example consider the following regular grammar:

$$\begin{aligned} S &\rightarrow 0A \mid 0 \mid 1B \\ A &\rightarrow 1A \mid 1 \\ B &\rightarrow 0B \mid 1S \end{aligned}$$

3. Replacing the " $\rightarrow$ " symbol in the productions of the grammar with the "=" symbol, we get the following set of equations:

$$S = 0A \mid 0 \mid 1B \tag{I}$$

$$A = 1A \mid 1 \tag{II}$$

$$B = 0B \mid 1S \tag{III}$$

From equation (III) we get:

$$B = 0^*1S$$

because equation (III) is of the form  $A = aA \mid b$ , where  $a$  and  $b$  are the expressions that do not contain variable  $A$ , and the solution of this is  $A = a^*b$ . Similarly, from equation (II) we get:

$$A = 1^*1$$

Substituting the values of  $A$  in (I) gives:

$$S = 01^*1 \mid 0 \mid 10^*1S$$

$$S = (10^*1)S \mid (01^*1 \mid 0)$$

Therefore,  $S = (10^*1)^*(01^*1 \mid 0)$ .

Hence, the required regular expression is:

$$(10^*1)^*(01^*1 \mid 0)$$

## 3.4 RIGHT LINEAR AND LEFT LINEAR GRAMMAR

### 3.4.1 Right Linear Grammar

Right linear grammar is a context-free grammar in which every production is restricted to one of the following forms:

1.  $A \rightarrow wB$
2.  $A \rightarrow w$ , where  $A$  and  $B$  are the nonterminals, and  $w$  is in  $T^*$

Since  $w$  is in  $T^*$ ,  $w$  can also be a single terminal; hence, every regular grammar, by default, satisfies this requirement of a right linear grammar. Therefore every regular grammar is a right linear grammar. Similarly, when  $|w| > 1$ , productions containing  $w$  on the right side can be split into more than one production. Each contains only one terminal and only one nonterminal on the right side by using additional nonterminals, because  $w$  can be written as  $ay$ , where  $a$  is the first terminal symbol of  $w$  and  $y$  is string made of the remaining symbols of  $w$ . Therefore, a production  $A \rightarrow wB$  can be split into the productions  $A \rightarrow aB_1$  and  $B_1 \rightarrow yB$  without affecting the language generated by the grammar. The production  $B_1 \rightarrow yB$  can be further split in a similar manner. And this can continue until  $|y|$  becomes one. A production  $A \rightarrow w$  can also be split into the productions  $A \rightarrow aB_1$  and  $B_1 \rightarrow y$  without affecting the language generated by the grammar. The production  $B_1 \rightarrow y$  can be further split in a similar manner, and this can continue until  $|y|$  becomes one, bringing the productions into the form required by the regular grammar. Therefore, we conclude that every right linear grammar can be rewritten in such a manner; every production of the grammar will satisfy the requirement of the regular grammar. For example, consider the following grammar:

$$\begin{aligned} S &\rightarrow aaB \mid ab \\ B &\rightarrow bB \mid bb \end{aligned}$$

The grammar is a right linear grammar; the production  $S \rightarrow aaB$  can be split into the productions  $S \rightarrow aC$  and  $C \rightarrow aB$  without affecting what is derived from  $S$ . Similarly, the production  $S \rightarrow ab$  can be split into the productions  $S \rightarrow aD$  and  $D \rightarrow b$ . The production  $B \rightarrow bb$  can also be split into the productions  $B \rightarrow bE$  and  $E \rightarrow b$ . Therefore, the above grammar can be rewritten as:

$$\begin{aligned} S &\rightarrow aC \\ C &\rightarrow aB \\ S &\rightarrow aD \\ D &\rightarrow a \\ B &\rightarrow bB \\ B &\rightarrow bE \\ E &\rightarrow b \end{aligned}$$

which is a regular grammar.

### 3.4.2 Left Linear Grammar

Left linear grammar is a context-free grammar in which every production is restricted to one of the following forms:

1.  $A \rightarrow Bw$
2.  $A \rightarrow w$ , where  $A$  and  $B$  are the nonterminals, and  $w$  is in  $T^*$

For every left linear grammar, there exists an equivalent right linear grammar that generates the same language, and vice versa. Hence, we conclude that every linear grammar (left or right) is a regular grammar. Given a right linear

grammar, an equivalent left linear grammar can be obtained as follows:

1. Obtain a regular expression for the language generated by the given grammar.
2. Reverse the regular expression obtained in step 1, above.
3. Obtain the regular, right linear grammar for the regular expression obtained in step 2.
4. Reverse the right side of every production of the grammar obtained in step 3. The resulting grammar will be an equivalent left linear grammar.

For example consider the right linear grammar given below:

$$\begin{aligned} S &\rightarrow 01B \mid 0 \\ B &\rightarrow 1B \mid 11 \end{aligned}$$

The regular expression for the above grammar is obtained as follows. Replace the  $\rightarrow$  by  $=$  in the above productions to obtain the equations:

$$S = 01B \mid 0 \quad (\text{I})$$

$$B = 1B \mid 11 \quad (\text{II})$$

Solving equation (II) gives:

$$B = 1^*(11)$$

By substituting the value of  $B$  in (I), we get:

$$S = 011^*11 \mid 0$$

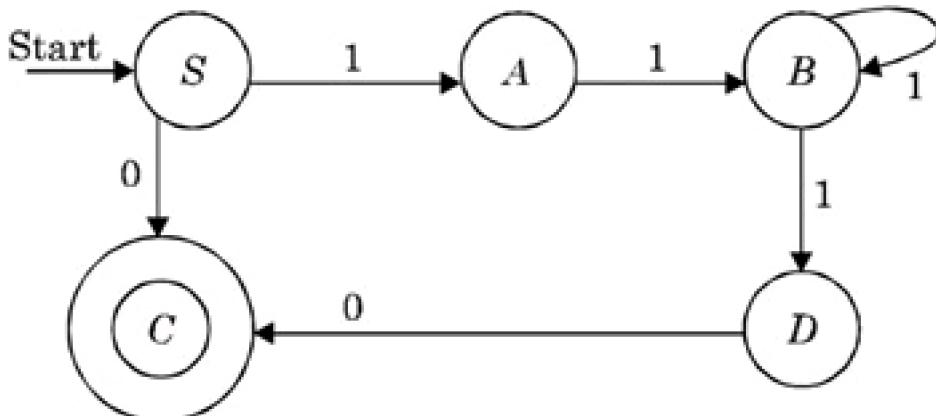
Therefore, the required regular expression is:

$$(011^*11 \mid 0)$$

And the reverse regular expression is:

$$(0 \mid 111^*10)$$

The finite automata accepting the language specified by the above regular expression is shown in [Figure 3.17](#).



**Figure 3.17:** Finite automata accepting the right linear grammar for a regular expression.

Therefore, the right linear grammar that generates the language accepted by the automata in [Figure 3.17](#) is:

$$S \rightarrow 1A \mid 0C \mid 0$$

$$A \rightarrow 1B$$

-----

$$B \rightarrow 1D \mid 1B$$

$$D \rightarrow 0C \mid 0$$

Since  $C$  is not useful, eliminating  $C$  gives:

$$S \rightarrow 1A \mid 0$$

$$A \rightarrow 1B$$

$$B \rightarrow 1D \mid 1B$$

$$D \rightarrow 0$$

which can be further simplified by replacing  $D$  in  $B \rightarrow 1D$ , using  $D \rightarrow 0$  to give:

$$S \rightarrow 1A \mid 0$$

$$A \rightarrow 1B$$

$$B \rightarrow 10 \mid 1B$$

Reversing the right side of the productions yields:

$$S \rightarrow A1 \mid 0$$

$$A \rightarrow B1$$

$$A \rightarrow 01 \mid B \mid$$

which is the equivalent left linear grammar. So, given a left linear grammar, an equivalent right linear grammar can be obtained as follows:

1. Reverse the right side of every production of the given grammar.
2. Obtain a regular expression for the language generated by the grammar obtained in step 1, above.
3. Reverse the regular expression obtained in the step 2.
4. Obtain the regular, right linear grammar for the regular expression obtained in the step 3.

The resulting grammar will be an equivalent left linear grammar. For example, consider the following left linear grammar:

$$S \rightarrow Sab \mid Aa$$

$$A \rightarrow Abb \mid bb$$

Reversing the right side of the productions gives us:

$$S \rightarrow baS \mid aA$$

$$A \rightarrow bba \mid bb$$

The regular expression that specifies the language generated by the above grammar can be obtained as follows.

Replace the  $\rightarrow$  symbols with  $=$  symbols in the productions of the above grammar to get the following set of equations:

$$S = baS \mid aA \tag{I}$$

$$A = bba \mid bb \tag{II}$$

From equation (II), we get:

$$A = (bb)^*(bb)$$

Substituting this value in (I) gives us:

$$S = baS \mid a(bb)^*(bb)$$

Therefore,

$$S = (ba)^*(a(bb)^*bb)$$

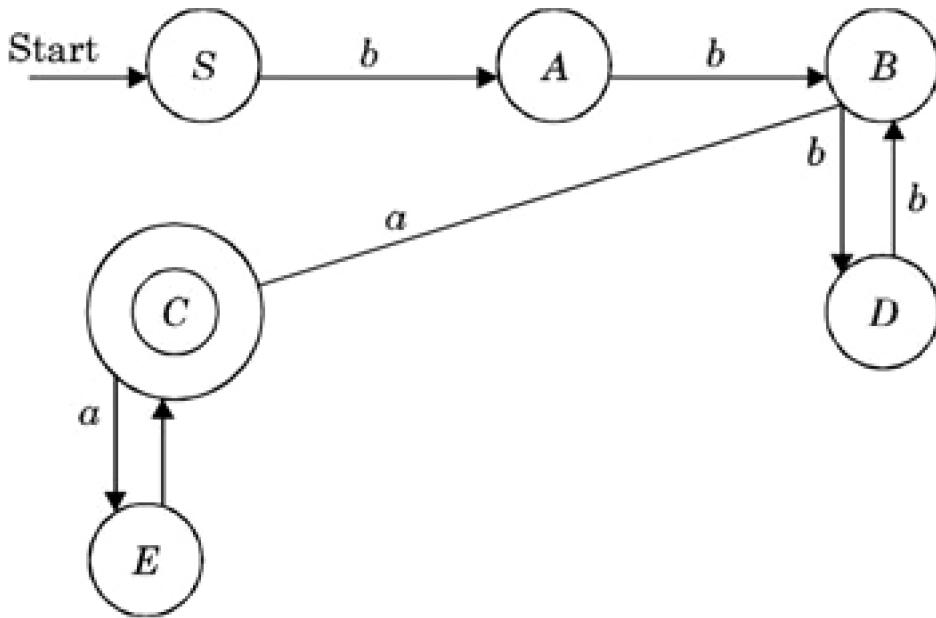
and the regular expression is:

$$(ba)^*(a(bb)^*bb)$$

The reversed regular expression is:

$$(bb(bb)^*a)(ab)^*$$

The finite automata that accepts the language specified by the reversed regular expression is shown in [Figure 3.18](#).



**Figure 3.18:** Transition diagram for a finite automata specified by a reversed regular expression.

Therefore, the regular grammar that generates the language accepted by the automata shown in [Figure 3.18](#) is:

$$\begin{aligned} S &\rightarrow bA \\ A &\rightarrow bB \\ B &\rightarrow aC \mid bD \mid a \\ D &\rightarrow bB \\ C &\rightarrow aE \\ E &\rightarrow bC \mid b \end{aligned}$$

which can be reduced to:

$$\begin{aligned} S &\rightarrow bbB \\ B &\rightarrow aaE \mid bbB \mid a \\ E &\rightarrow baE \mid b \end{aligned}$$

which is the required right linear grammar.

**EXAMPLE 3.15**

Consider the following grammar to obtain an equivalent left linear grammar.

$$\begin{aligned}S &\rightarrow gA \\A &\rightarrow aA \mid gB \mid g \\B &\rightarrow gA\end{aligned}$$

---

The regular expression for the above grammar is obtained as follows. Replace the  $\rightarrow$  by = in the above productions to obtain the equations:

$$S = gA \quad (\text{I})$$

$$A = aA \mid gB \mid g \quad (\text{II})$$

$$B = gA \quad (\text{III})$$

By substituting (III) in (II) we get:

$$A = aA \mid ggA \mid g$$

Therefore,  $A = (a \mid gg)A \mid g$  and  $A = (a \mid gg)^*g$ . By substituting this value in (I) we get:

$$S = a(a \mid gg)^*g$$

And the regular expression is:

$$a(a \mid gg)^*g$$

Therefore, the reversed regular expression is:

$$a(gg \mid a)^*g$$

But since  $(a \mid gg)^*$  is the same as  $(gg \mid a)^*$ , the reversed regular expression is same. Hence, the regular, right linear grammar that generates the language specified by the reversed regular expression is the given grammar itself.

Therefore, an equivalent left linear grammar can be obtained by reversing the right side of the productions of the given grammar:

$$\begin{aligned}S &\rightarrow Ag \\A &\rightarrow Aa \mid Bg \mid g \\B &\rightarrow Ag\end{aligned}$$

## Chapter 4: Top-Down Parsing

### INTRODUCTION

A syntax analyzer or parser is a program that performs syntax analysis. A parser obtains a string of tokens from the lexical analyzer and verifies whether or not the string is a valid construct of the source language—that is, whether or not it can be generated by the grammar for the source language. And for this, the parser either attempts to derive the string of tokens  $w$  from the start symbol  $S$ , or it attempts to reduce  $w$  to the start symbol of the grammar by tracing the derivations of  $w$  in reverse. An attempt to derive  $w$  from the grammar's start symbol  $S$  is equivalent to an attempt to construct the top-down parse tree; that is, it starts from the root node and proceeds toward the leaves. Similarly, an attempt to reduce  $w$  to the grammar's start symbol  $S$  is equivalent to an attempt to construct a bottom-up parse tree; that is, it starts with  $w$  and traces the derivations in reverse, obtaining the root  $S$ .

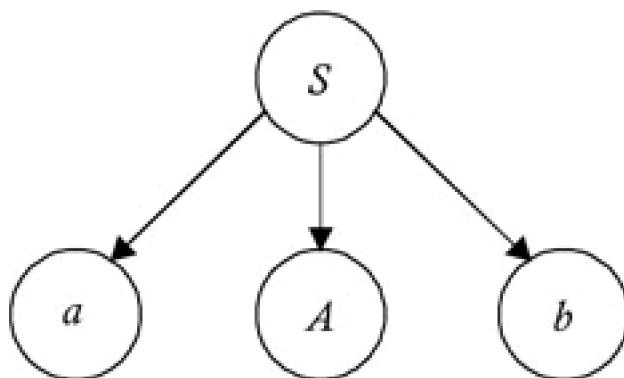
## 4.1 TOP-DOWN PARSING

Top-down parsing attempts to find the left-most derivations for an input string  $w$ , which is equivalent to constructing a parse tree for the input string  $w$  that starts from the root and creates the nodes of the parse tree in a predefined order. The reason that top-down parsing seeks the left-most derivations for an input string  $w$  and not the right-most derivations is that the input string  $w$  is scanned by the parser from left to right, one symbol/token at a time, and the left-most derivations generate the leaves of the parse tree in left-to-right order, which matches the input scan order.

Since top-down parsing attempts to find the left-most derivations for an input string  $w$ , a top-down parser may require backtracking (i.e., repeated scanning of the input); because in the attempt to obtain the left-most derivation of the input string  $w$ , a parser may encounter a situation in which a nonterminal  $A$  is required to be derived next, and there are multiple  $A$ -productions, such as  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ . In such a situation, deciding which  $A$ -production to use for the derivation of  $A$  is a problem. Therefore, the parser will select one of the  $A$ -productions to derive  $A$ , and if this derivation finally leads to the derivation of  $w$ , then the parser announces the successful completion of parsing. Otherwise, the parser resets the input pointer to where it was when the nonterminal  $A$  was derived, and it tries another  $A$ -production. The parser will continue this until it either announces the successful completion of the parsing or reports failure after trying all of the alternatives. For example, consider the top-down parser for the following grammar:

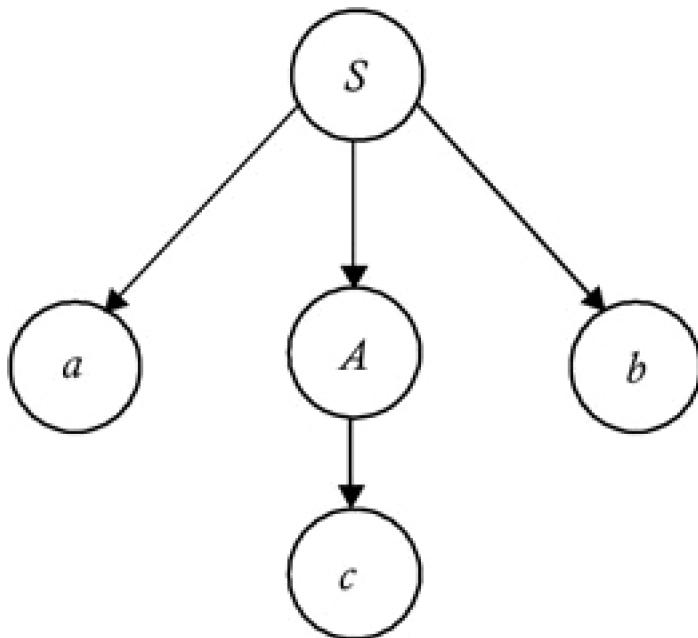
$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow cd \mid c \end{aligned}$$

Let the input string be  $w = acb$ . The parser initially creates a tree consisting of a single node, labeled  $S$ , and the input pointer points to  $a$ , the first symbol of input string  $w$ . The parser then uses the  $S$ -production  $S \rightarrow aAb$  to expand the tree as shown in [Figure 4.1](#).



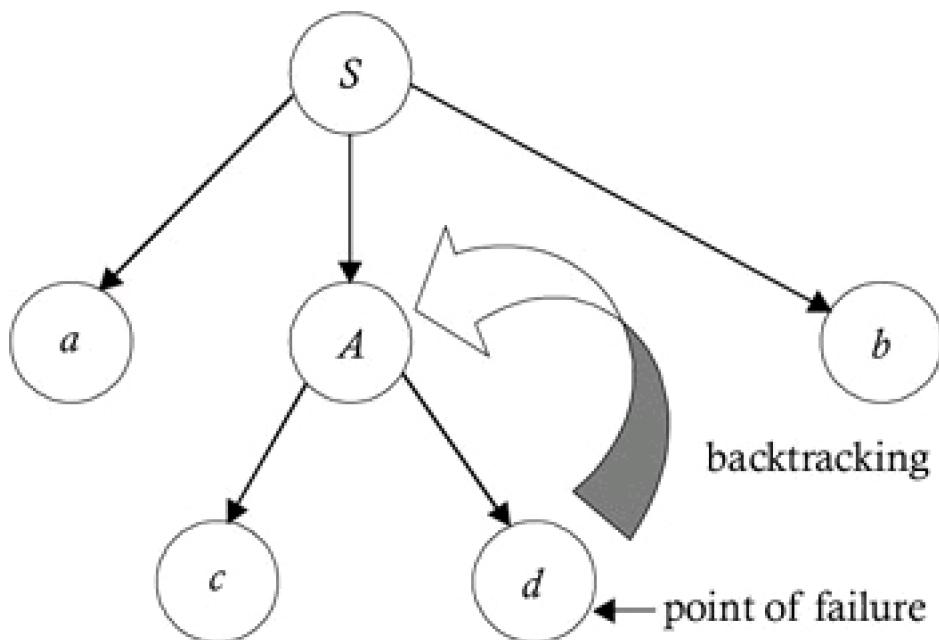
**Figure 4.1:** Parser uses the  $S$ -production to expand the parse tree.

The left-most leaf, labeled  $a$ , matches the first input symbol of  $w$ . Hence, the parser will now advance the input pointer to  $c$ , the second symbol of string  $w$ , and consider the next leaf labeled  $A$ . It will then expand  $A$ , using the first alternative for  $A$  in order to obtain the tree shown in [Figure 4.2](#).



**Figure 4.2:** Parser uses the first alternative for A in order to expand the tree.

The parser now has the match for the second input symbol. So, it advances the pointer to *b*, the third symbol of *w*, and compares it to the label of the next leaf. If the label does not match *d*, it reports failure and goes back (backtracks) to *A*, as shown in [Figure 4.3](#). The parser will also reset the input pointer to the second input symbol—the position it had when the parser encountered *A*—and it will try a second alternative to *A* in order to obtain the tree. If the leaf *c* matches the second symbol, and if the next leaf *b* matches the third symbol of *w*, then the parser will halt and announce the successful completion of parsing.



**Figure 4.3:** If the parser fails to match a leaf, the point of failure, *d*, reroutes (backtracks) the pointer to alternative paths from *A*.

## 4.2 IMPLEMENTATION

A top-down parser can be implemented by writing a set of recursive procedures to process the input. One procedure will take care of the left-most derivations for each nonterminal while processing the input. Each procedure should also provide for the storing of the input pointer in some local variable so that it can be reset properly when the parser backtracks. This implementation, called a "recursive descent parser," is a top-down parser for the above-described grammar that can be implemented by writing the following set of procedures:

```

S( )
{
if (input =='a' )
{
    advance( );
    if (A( ) != error)
        if (input =='b')
            { advance( );
            if (input == endmarker)
                return(success);
            else
                return(error);
            }
        else
            return(error);
    }
else
    return(error);
}

A( )
{
if (input =='c')
{
    advance( );
    if (input == 'd')
        advance( );}
    else
        return(error);
}

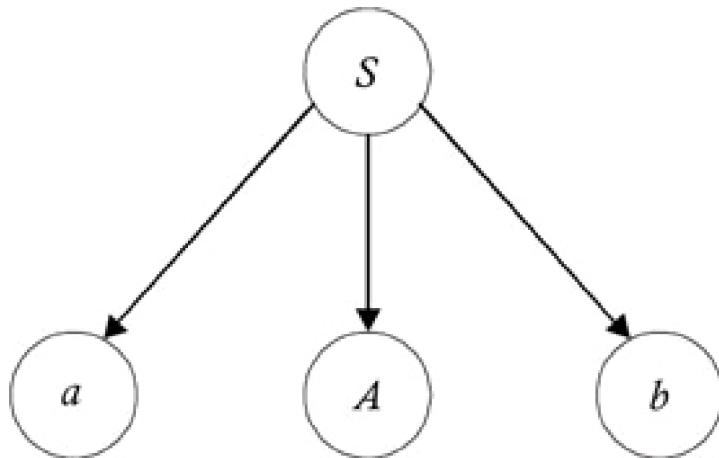
main( )
{
Append the endmarker to the string w to be parsed;
Set the input pointer to the left most token of w;
if ( S( ) != error)
    printf ("Successful completion of the parsing");
else
    printf ("Failure");
}

```

where `advance()` is a routine that, when called, advances the input's pointer to the next occurrence of the symbol `w`.

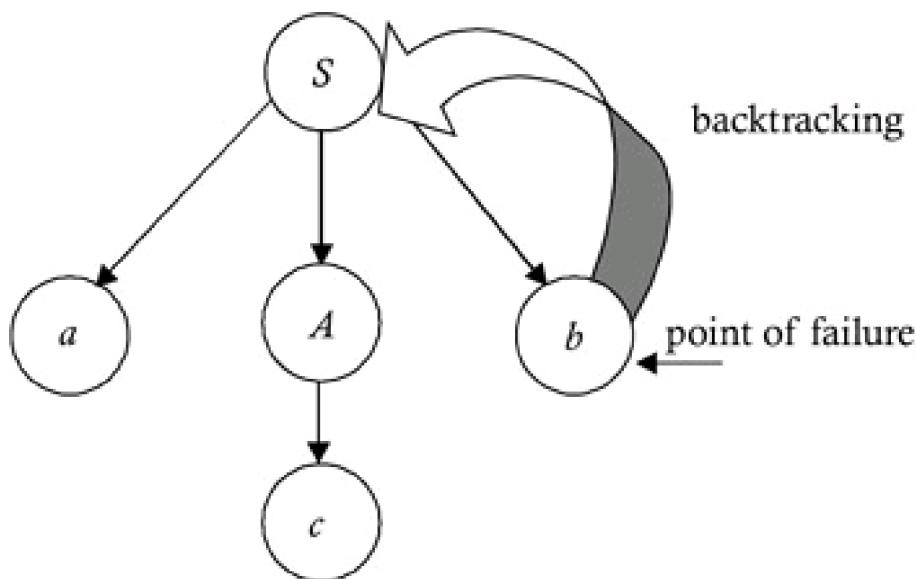
**Caution** In a backtracking parser, the order in which alternatives are tried affects the language accepted by the parser.

For example, in the above parser, if a production  $A \rightarrow c$  is tried before  $A \rightarrow cd$ , then the parser will fail to accept the string  $w = acdb$ , because it first expands  $S$ , as shown in [Figure 4.4](#).



**Figure 4.4:** The parser first expands S and fails to accept  $w = acdb$ .

The first input symbol matches the left-most leaf; and therefore, the parser will advance the pointer to  $c$  and consider the nonterminal  $A$  for expansion in order to obtain the tree shown in Figure 4.5.



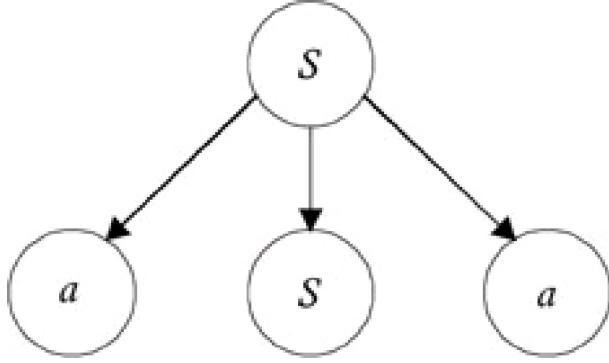
**Figure 4.5:** The parser advances to c and considers nonterminal A for expansion.

The second input symbol also matches. Therefore, the parser will advance the pointer to  $d$ , the third input symbol, and consider the next leaf, labeled  $b$  in [Figure 4.5](#). It finds that there is no match; and therefore, it will backtrack to  $S$  (as shown in [Figure 4.5](#) by the thick arrow). But since there is no alternative to  $S$  that can be tried, the parser will return failure. Because the point of mismatch is the descendent of a node labeled by  $S$ , the parser will backtrack to  $S$ . It cannot backtrack to  $A$ . Therefore, the parser will not accept the string  $acdb$ . Whereas, if the parser tries the alternative  $A \rightarrow cd$  first and  $A \rightarrow c$  second, then the parser is capable of accepting the string  $acdb$  as well as  $acb$  because, for the string  $w = acb$ , when the parser encounters a mismatch, it is at a node labeled by  $d$ , which is a descendent of a node labeled by  $A$ . Hence, it will backtrack to  $A$  and try  $A \rightarrow c$ , and end up in the parse tree for  $acb$ . Hence, we conclude that the order in which alternatives are tried in a backtracking parser affect the language accepted by the compiler or parser.

### EXAMPLE 4.1

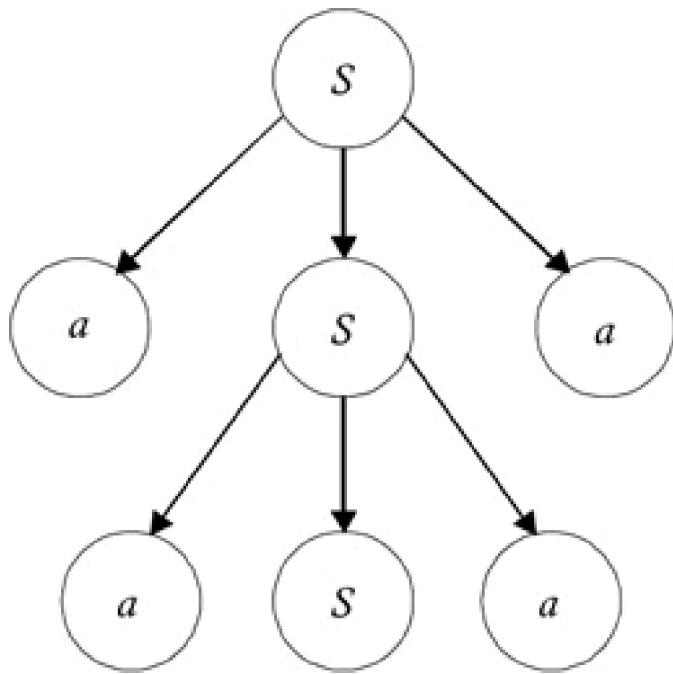
Consider a grammar  $S \rightarrow aa \mid aSa$ . If a top-down backtracking parser for this grammar tries  $S \rightarrow aSa$  before  $S \rightarrow aa$ , show that the parser succeeds on two occurrences of  $a$  and four occurrences of  $a$ , but not on six occurrences of  $a$ .

In the case of two occurrences of  $a$ , the parser will first expand  $S$ , as shown in Figure 4.6.



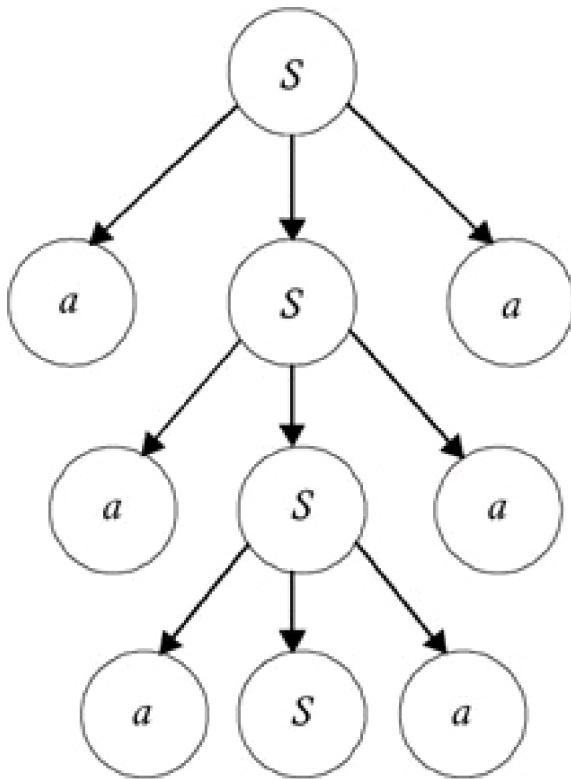
**Figure 4.6:** The parser first expands S.

The first input symbol matches the left-most leaf. Therefore, the parser will advance the pointer to a second a and consider the nonterminal S for expansion in order to obtain the tree shown in [Figure 4.7](#).



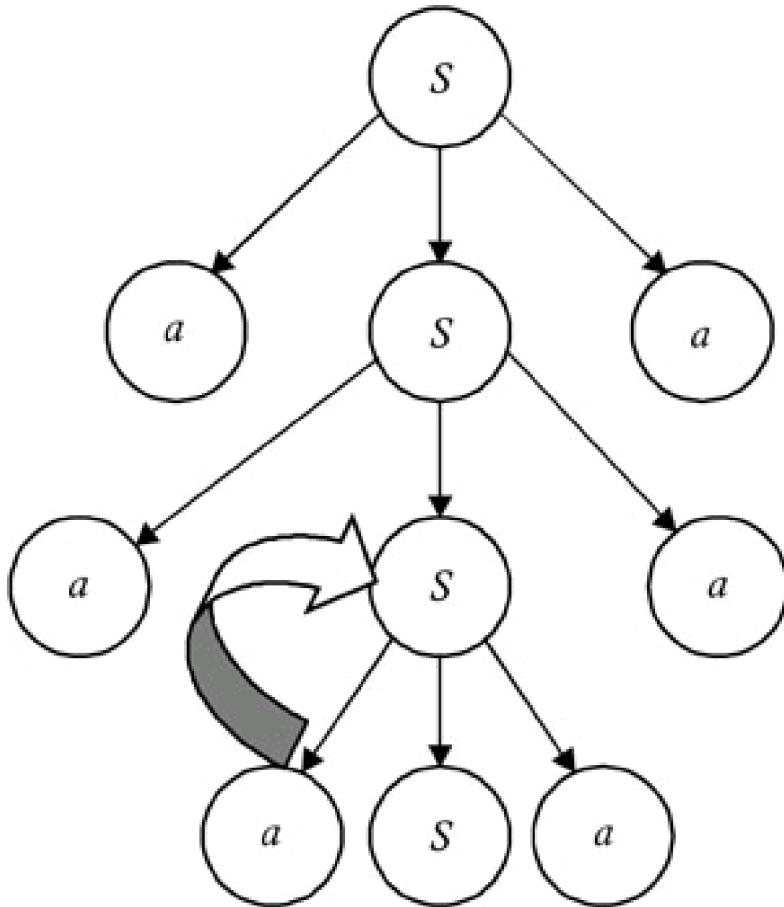
**Figure 4.7:** The parser advances the pointer to a second occurrence of a.

The second input symbol also matches. Therefore, the parser will consider the next leaf labeled S and expand it, as shown in [Figure 4.8](#).

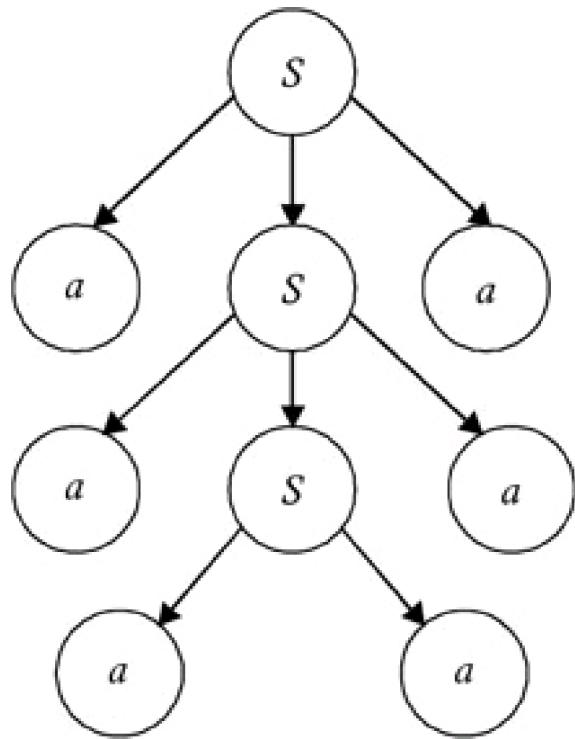


**Figure 4.8:** The parser expands the next leaf labeled S.

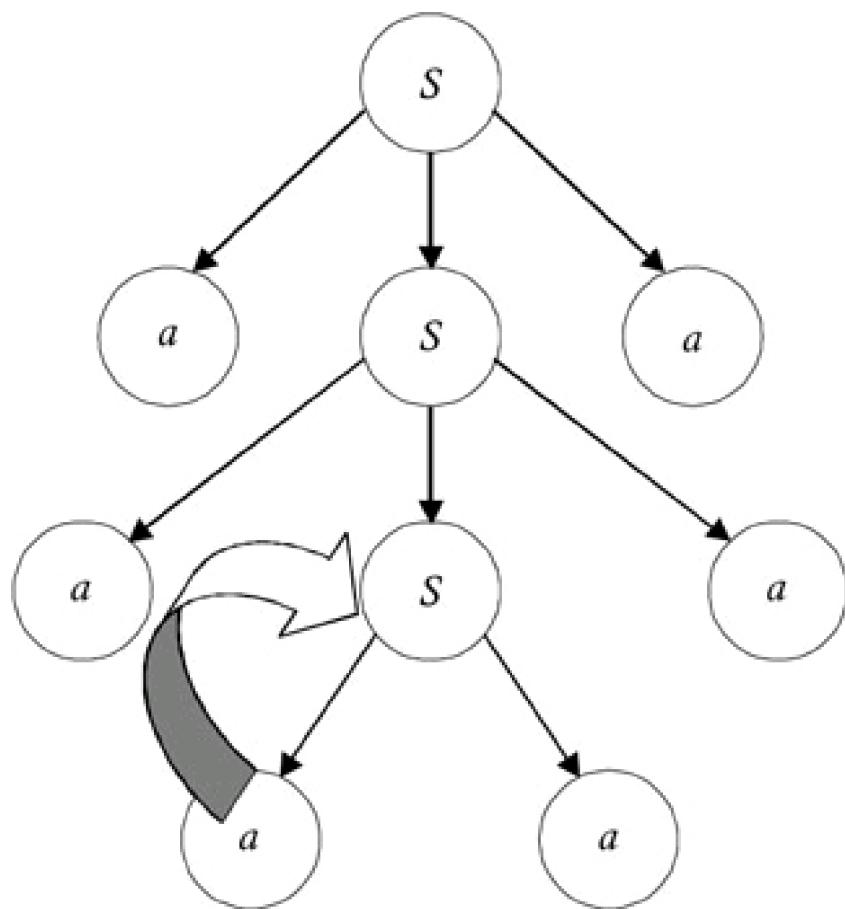
The parser now finds that there is no match. Therefore, it will backtrack to S, as shown by the thick arrow in [Figure 4.9](#). The parser then continues matching and backtracking, as shown in [Figures 4.10](#) through [4.15](#), until it arrives at the required parse tree, shown in [Figure 4.16](#).



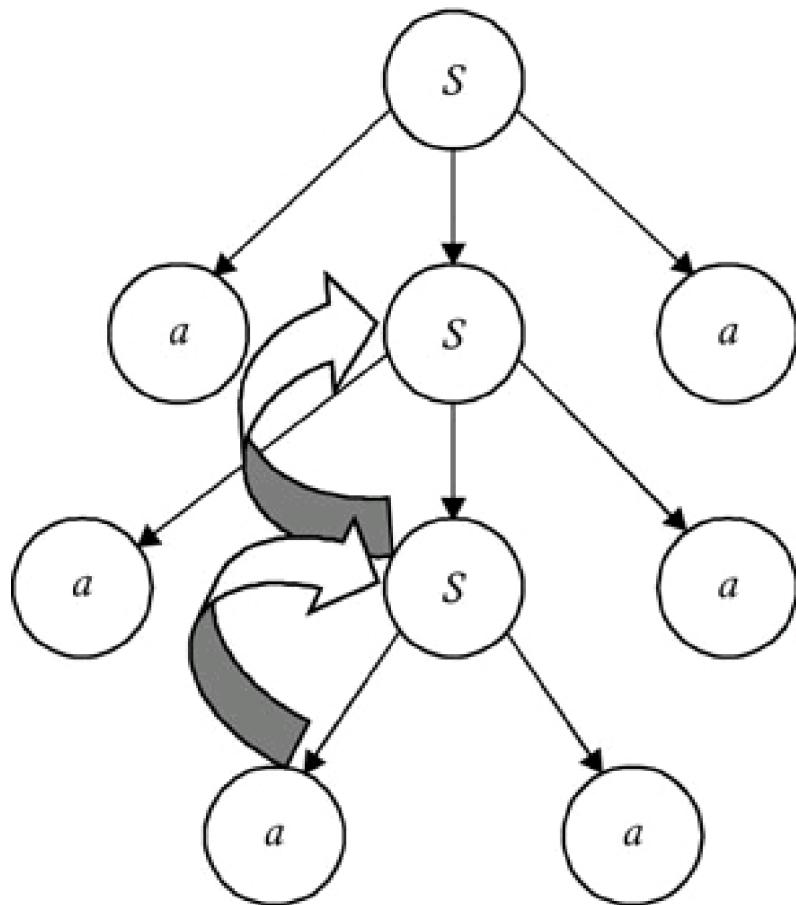
**Figure 4.9:** The parser finds no match, so it backtracks.



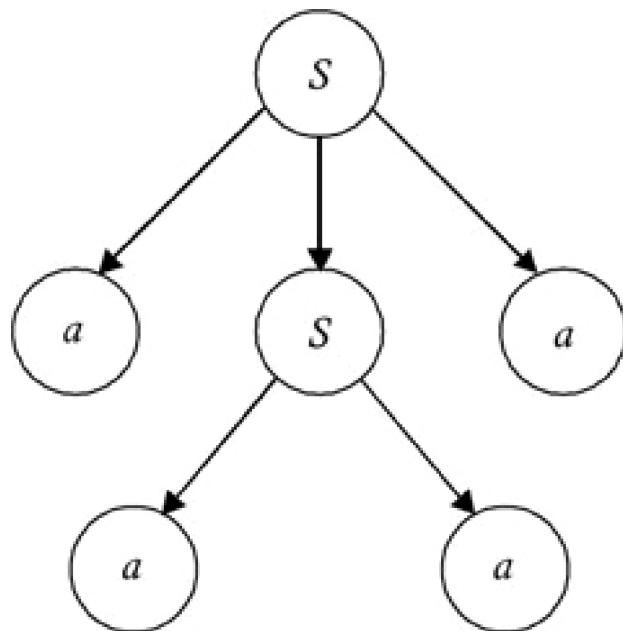
**Figure 4.10:** The parser tries an alternate aa.



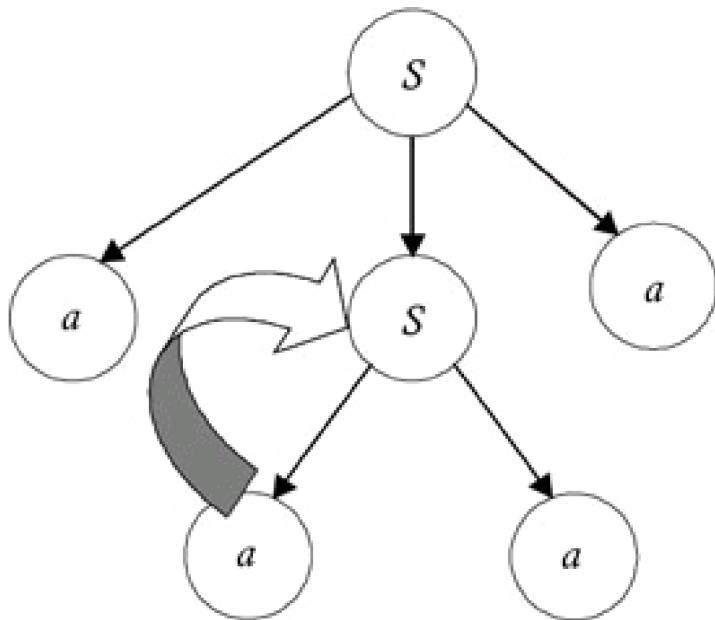
**Figure 4.11:** There is no further alternate of S that can be tried, so the parser will backtrack one more step.



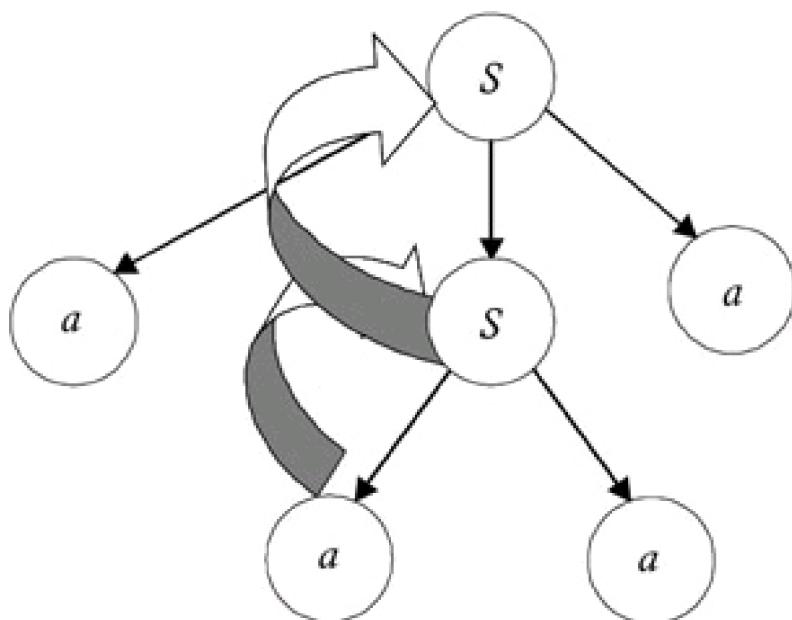
**Figure 4.12:** The parser again finds a mismatch; hence, it backtracks.



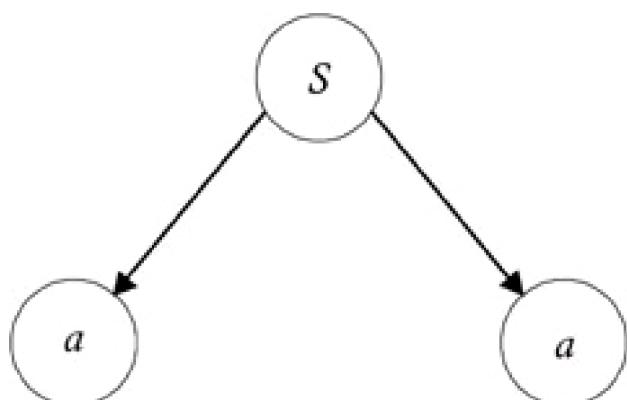
**Figure 4.13:** The parser tries an alternate aa.



**Figure 4.14:** Since no alternate of S remains to be tried, the parser backtracks one more step.

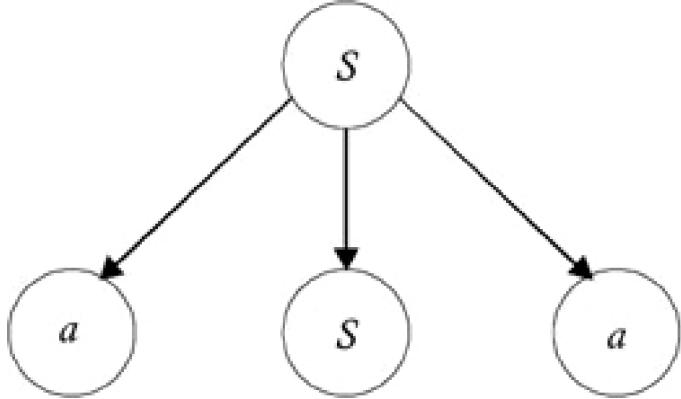


**Figure 4.15:** The parser tries an alternate aa.



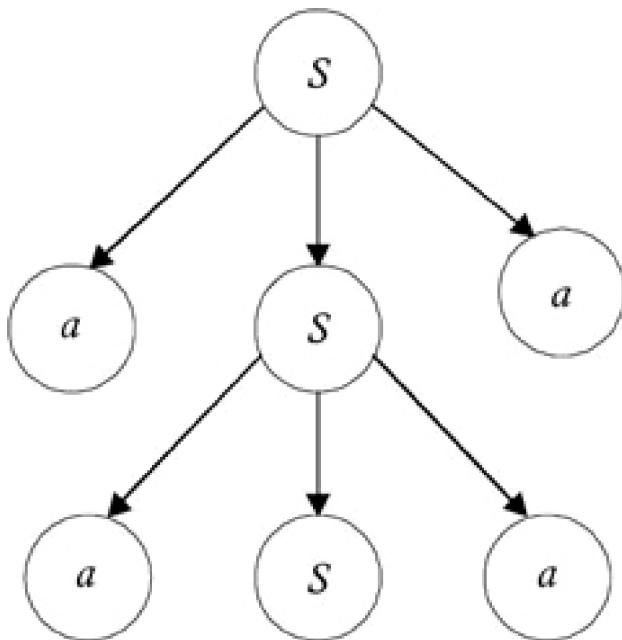
**Figure 4.16:** The parser arrives at the required parse tree.

Now, consider a string of four occurrences of a. The parser will first expand S, as shown in [Figure 4.17](#).



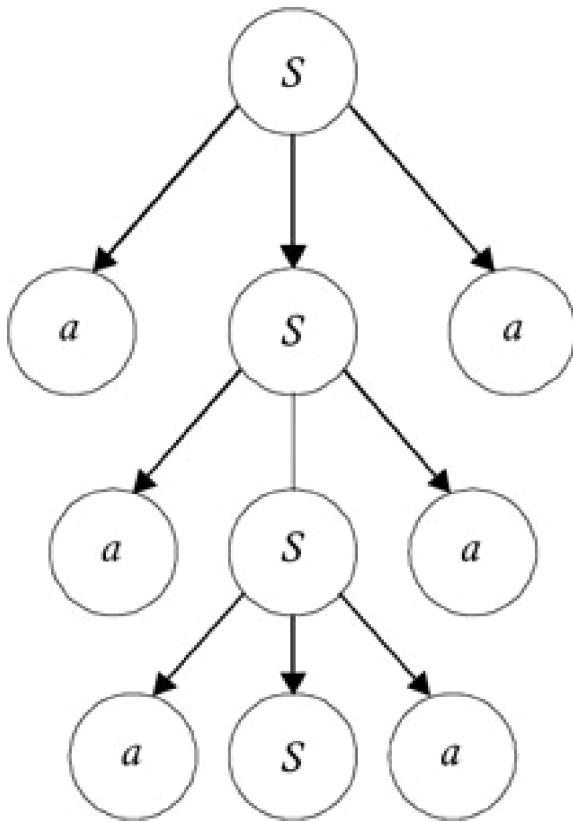
**Figure 4.17:** The parser first expands S.

The first input symbol matches the left-most leaf. Therefore, the parser will advance the pointer to a second a and consider the nonterminal S for expansion, obtaining the tree shown in [Figure 4.18](#).



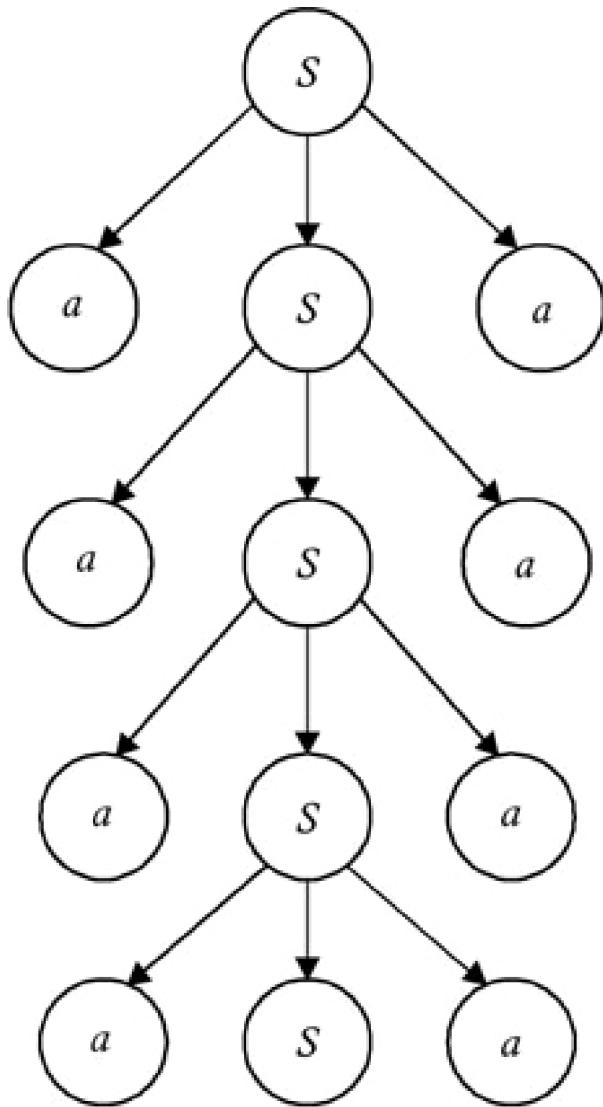
**Figure 4.18:** The parser advances the pointer to a second occurrence of a.

The second input symbol also matches. Therefore, the parser will consider the next leaf labeled by S and expand it, as shown in [Figure 4.19](#).



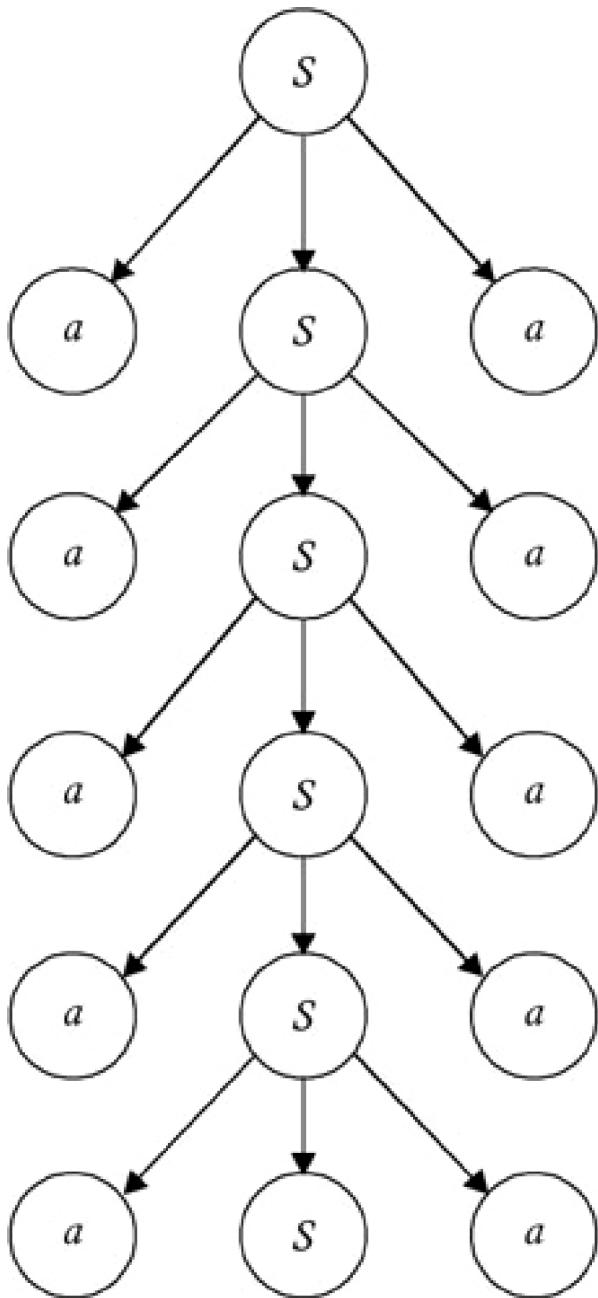
**Figure 4.19:** The parser considers the next leaf labeled by S.

The third input symbol also matches. So, the parser moves on to the next leaf labeled by S and expands it, as shown in [Figure 4.20](#).



**Figure 4.20:** The parser matches the third input symbol and moves on to the next leaf labeled by S.

The fourth input symbol also matches. Therefore, the next leaf labeled by S is considered. The parser expands it, as shown in [Figure 4.21](#).



**Figure 4.21:** The parser considers the fourth occurrence of the input symbol a.

Now it finds that there is no match. Therefore, it will backtrack to S ([Figure 4.22](#)) and continue backtracking, as shown in [Figures 4.23](#) through [4.30](#), until the parser finally arrives at the successful generation of a parse tree for *aaaa* in [Figure 4.31](#).

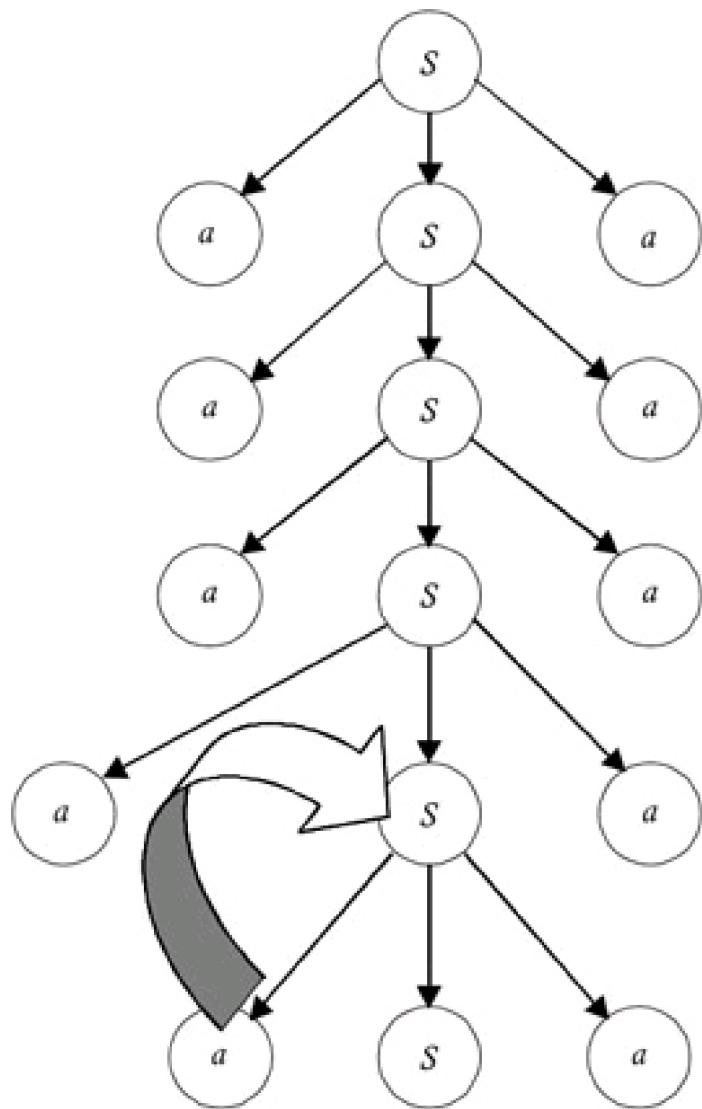


Figure 4.22: The parser finds no match, so it backtracks.

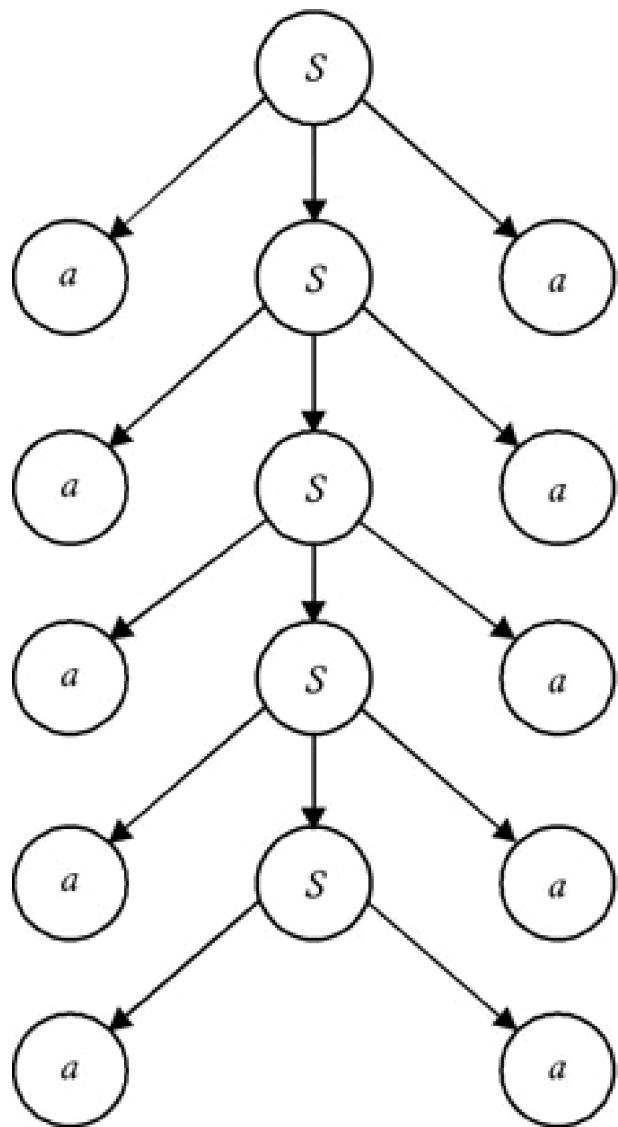
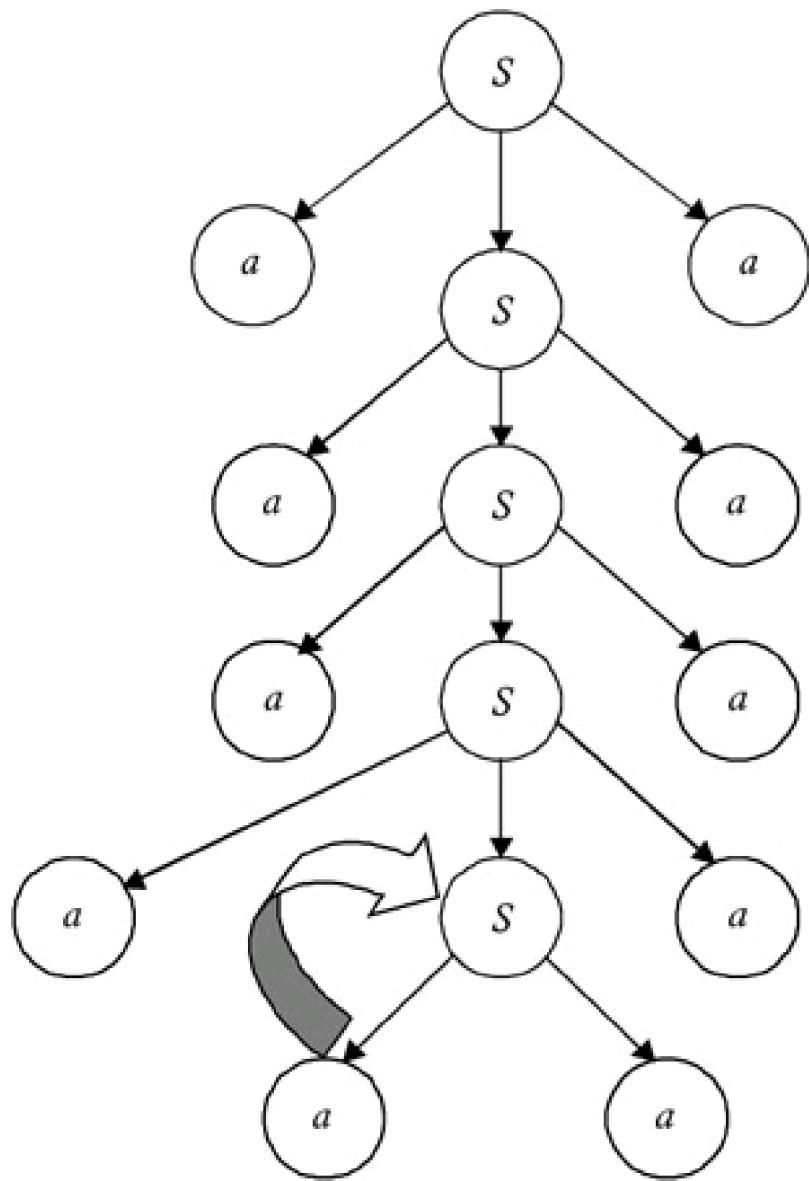


Figure 4.23: The parser tries an alternate aa.



**Figure 4.24:** No alternate of  $S$  can be tried, so the parser will backtrack one more step.

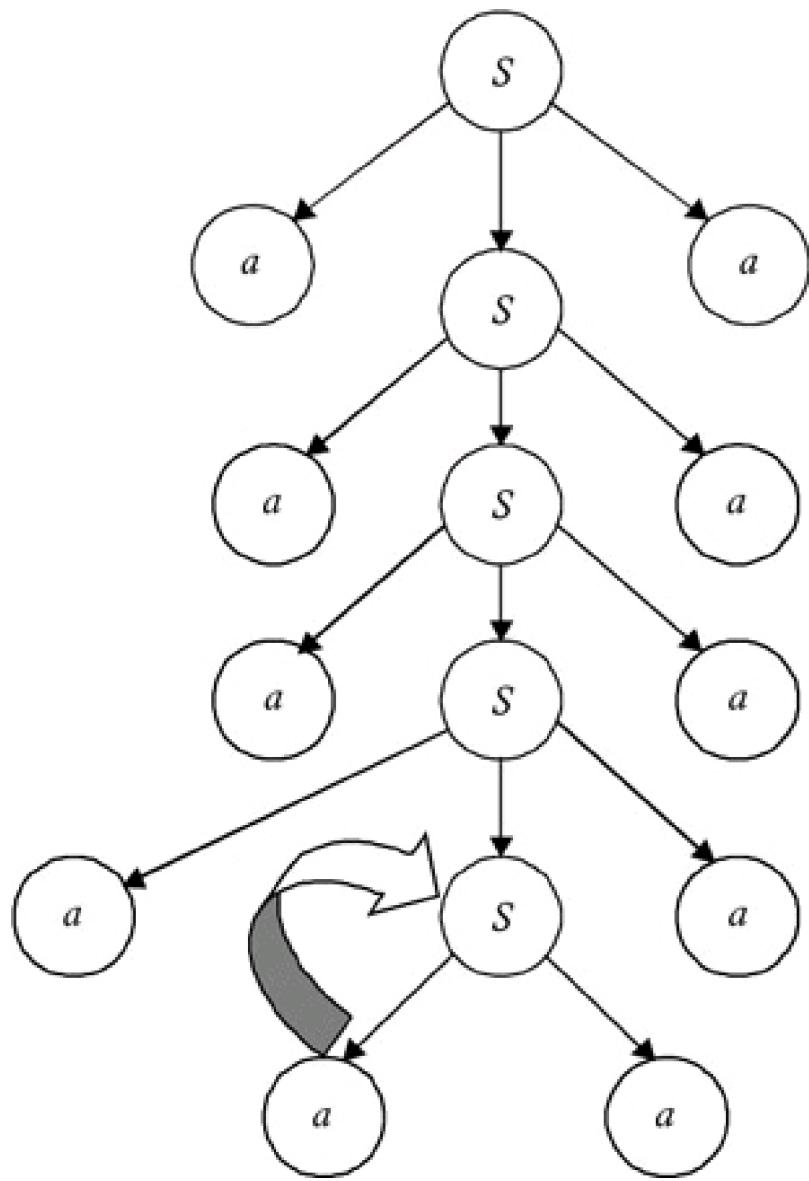


Figure 4.25: Again finding a mismatch, the parser backtracks.

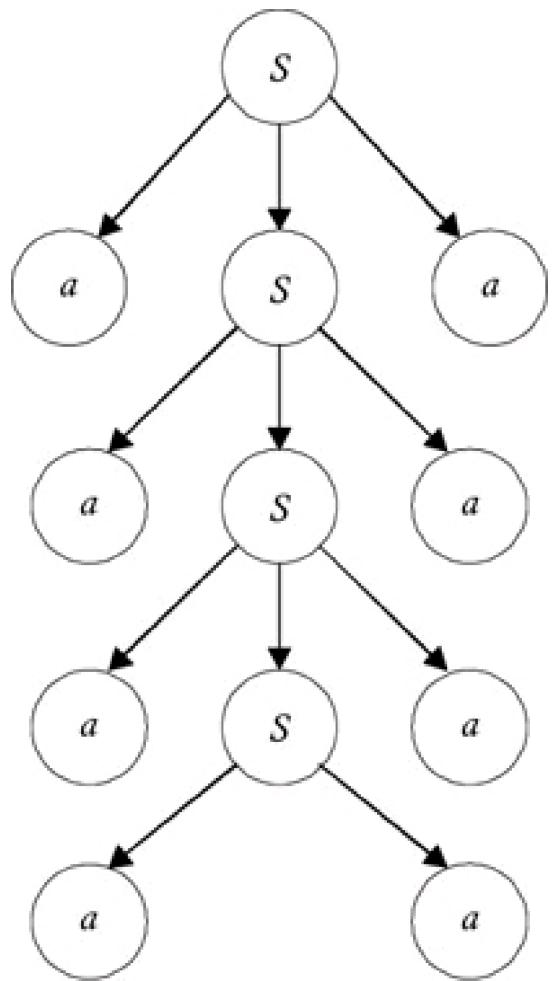
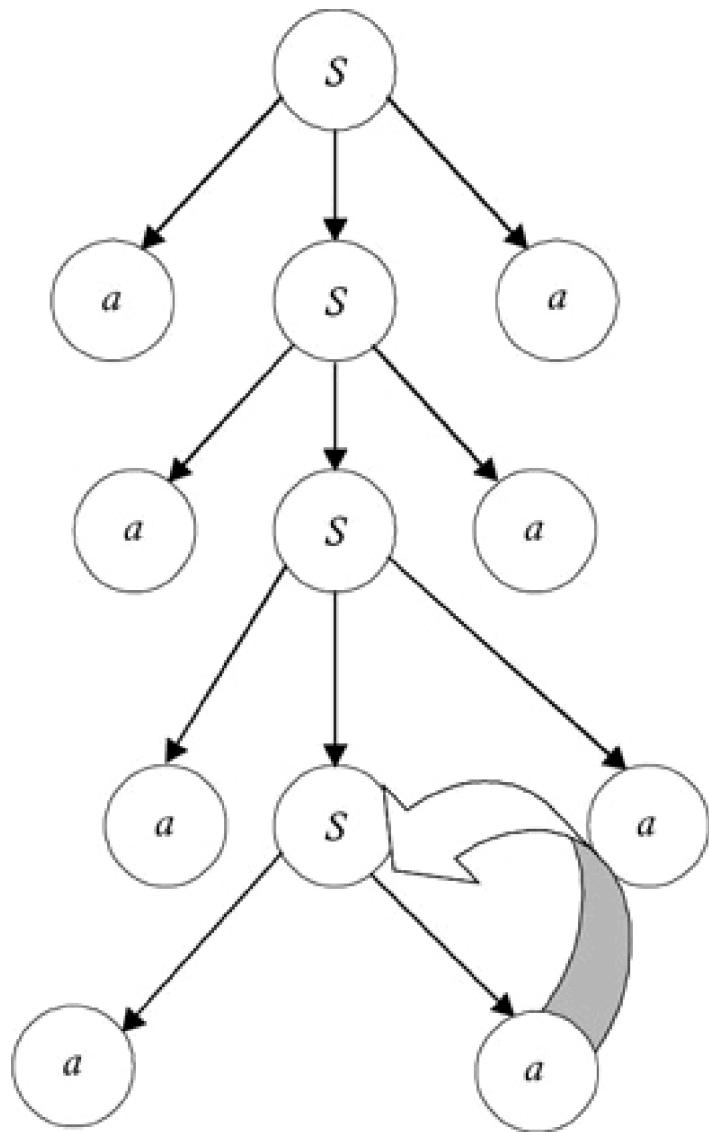


Figure 4.26: The parser then tries an alternate.



**Figure 4.27:** No alternate of S remains to be tried, so the parser will backtrack one more step.

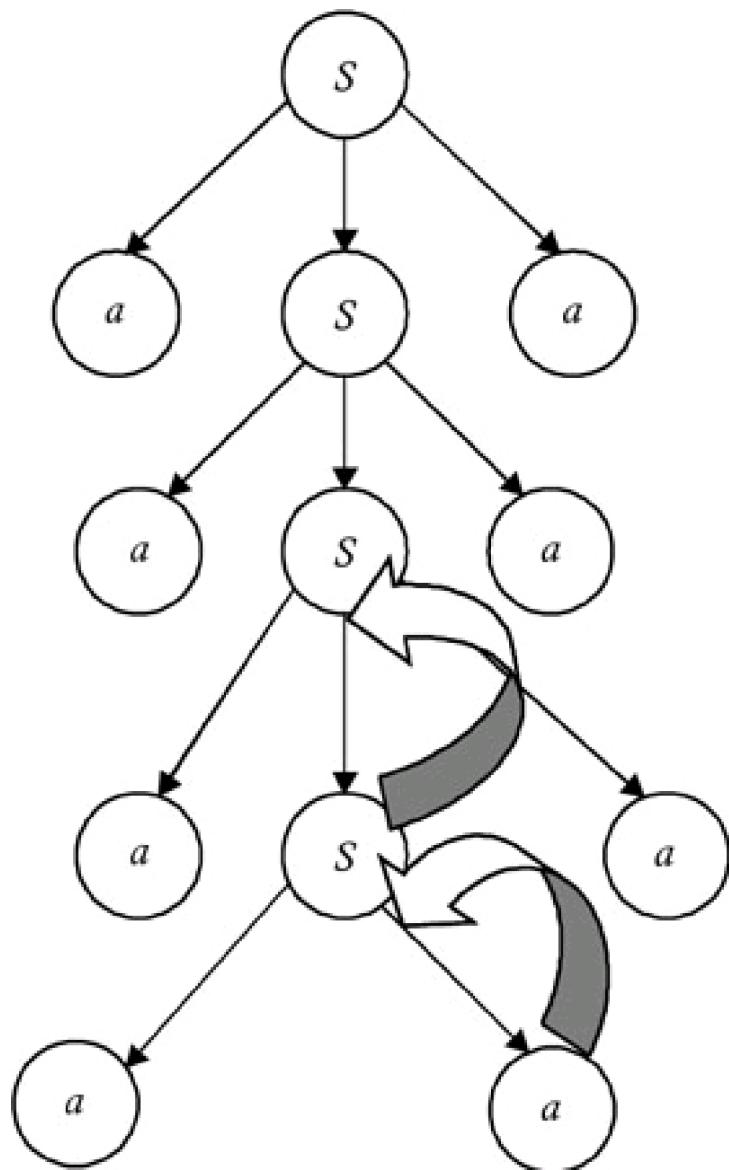
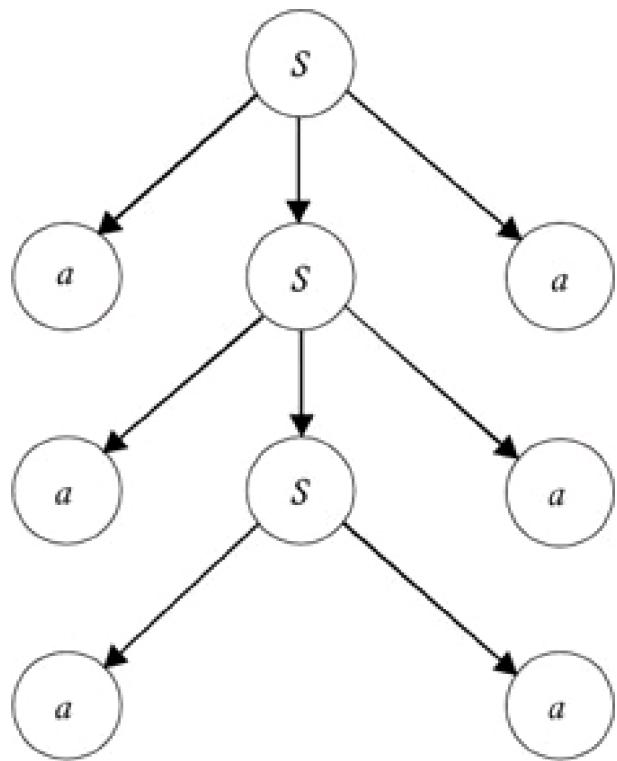
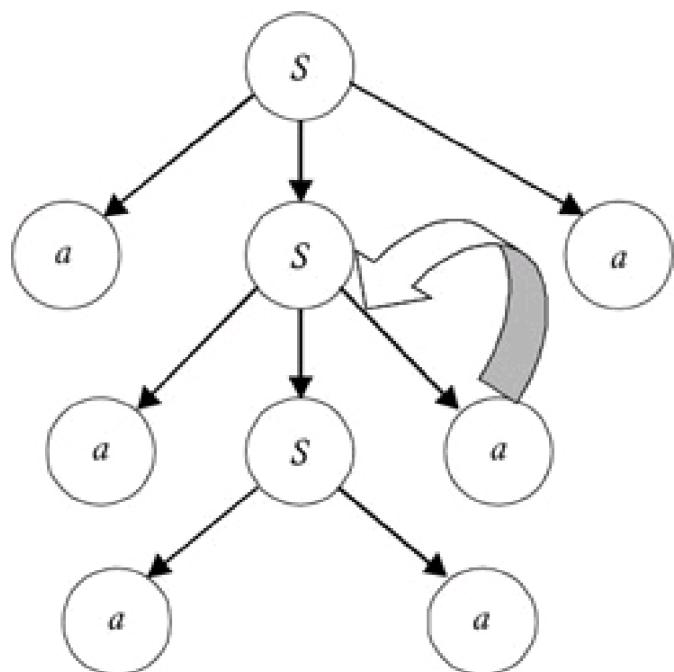


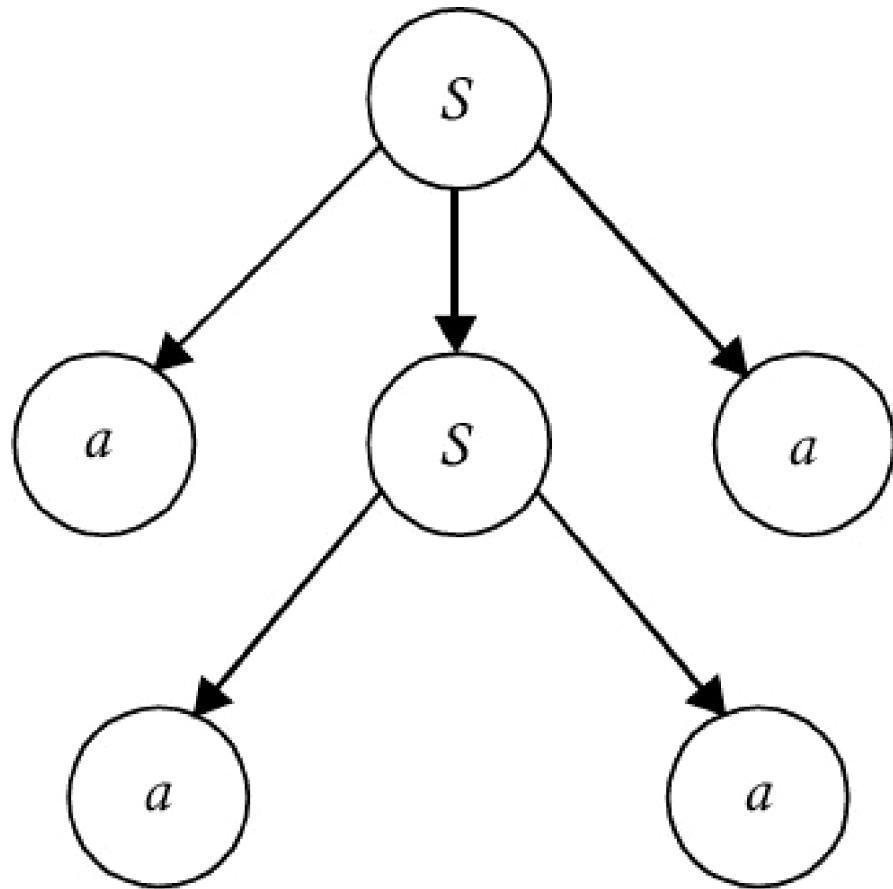
Figure 4.28: The parser again finds a mismatch; therefore, it backtracks.



**Figure 4.29:** The parser tries an alternate aa.

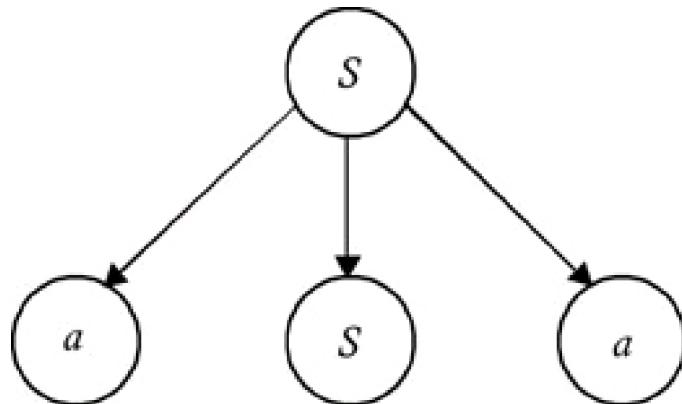


**Figure 4.30:** The parser then tries an alternate aa.



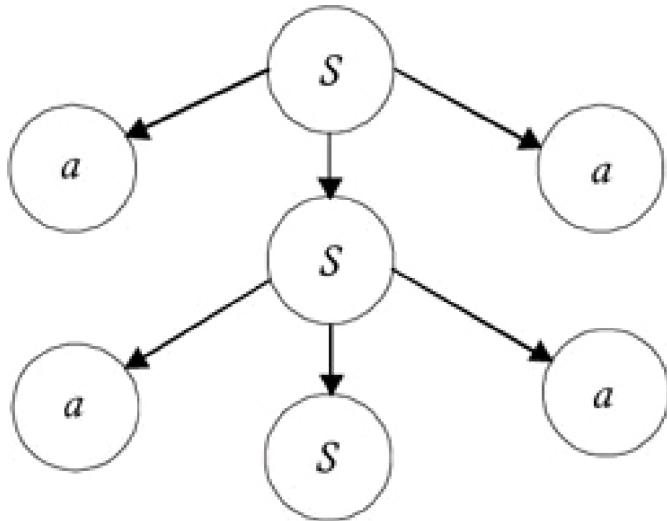
**Figure 4.31:** The parser successfully generates the parse tree for *aaaa*.

Now consider a string of six occurrences of *a*. The parser will first expand *S*, as shown in [Figure 4.32](#).



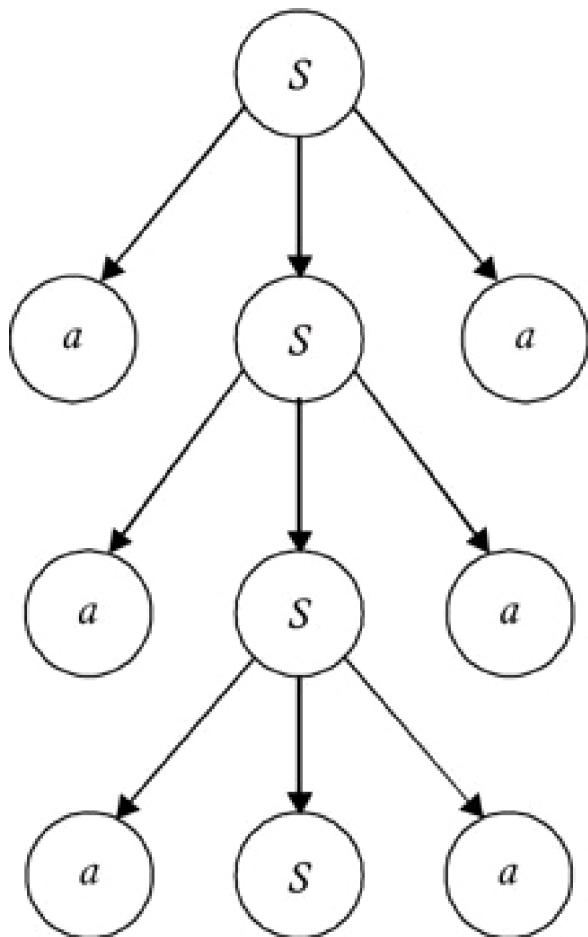
**Figure 4.32:** The parser expands *S*.

The first input symbol matches the left-most leaf. Therefore, the parser will advance the pointer to the second *a* and consider the nonterminal *S* for expansion. The tree shown in [Figure 4.33](#) is obtained.



**Figure 4.33:** The parser matches the first symbol, advances to the second occurrence of a, and considers S for expansion.

The second input symbol also matches. Therefore, the parser will consider next leaf labeled S and expand it, as shown in [Figure 4.34](#).



**Figure 4.34:** The parser finds a match for the second occurrence of a and expands S.

The third input symbol also matches, as do the fourth through sixth symbols. In each case, the parser will consider next leaf labeled S and expand it, as shown in [Figures 4.35](#) through [4.38](#).

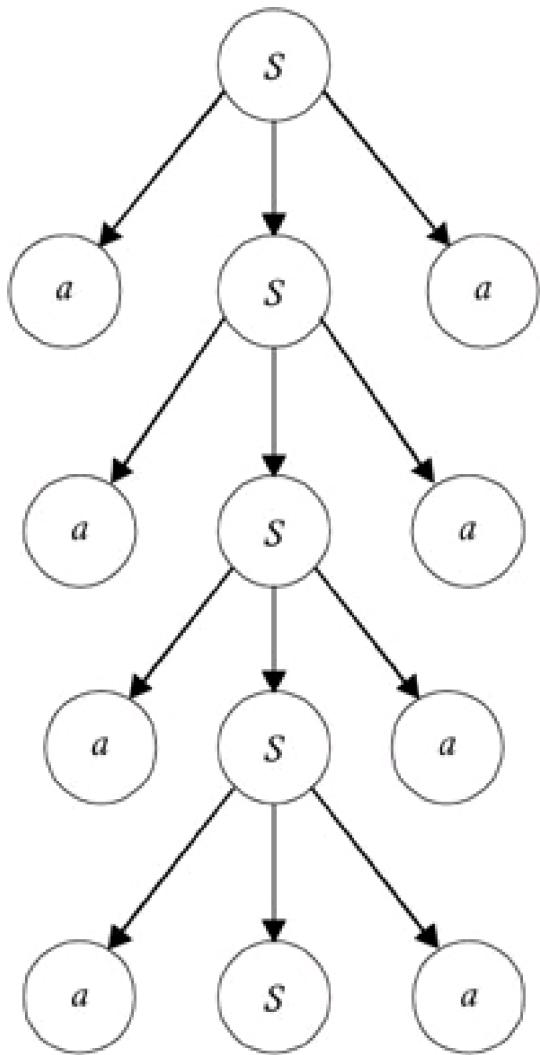


Figure 4.35: The parser matches the third input symbol, considers the next leaf, and expands S.

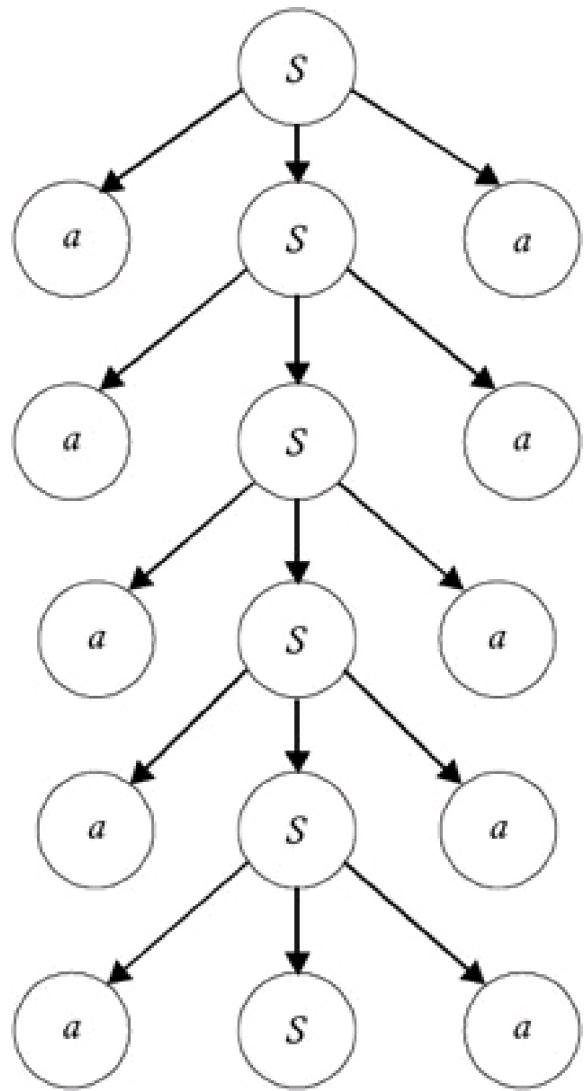


Figure 4.36: The parser matches the fourth input symbol, considers the next leaf, and expands S.

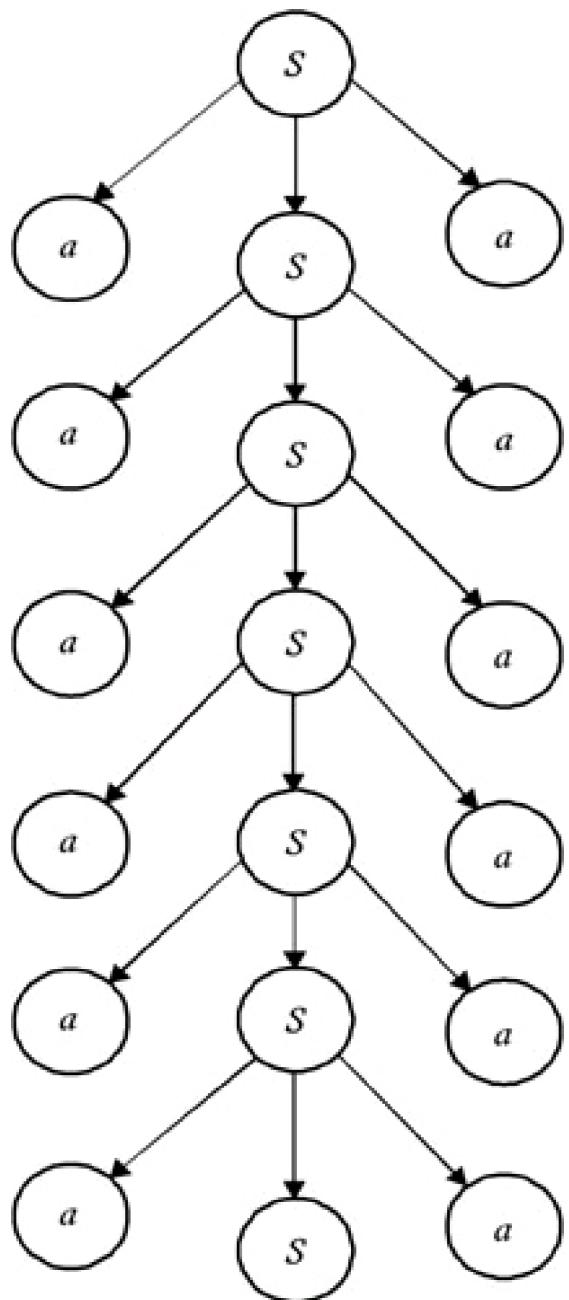
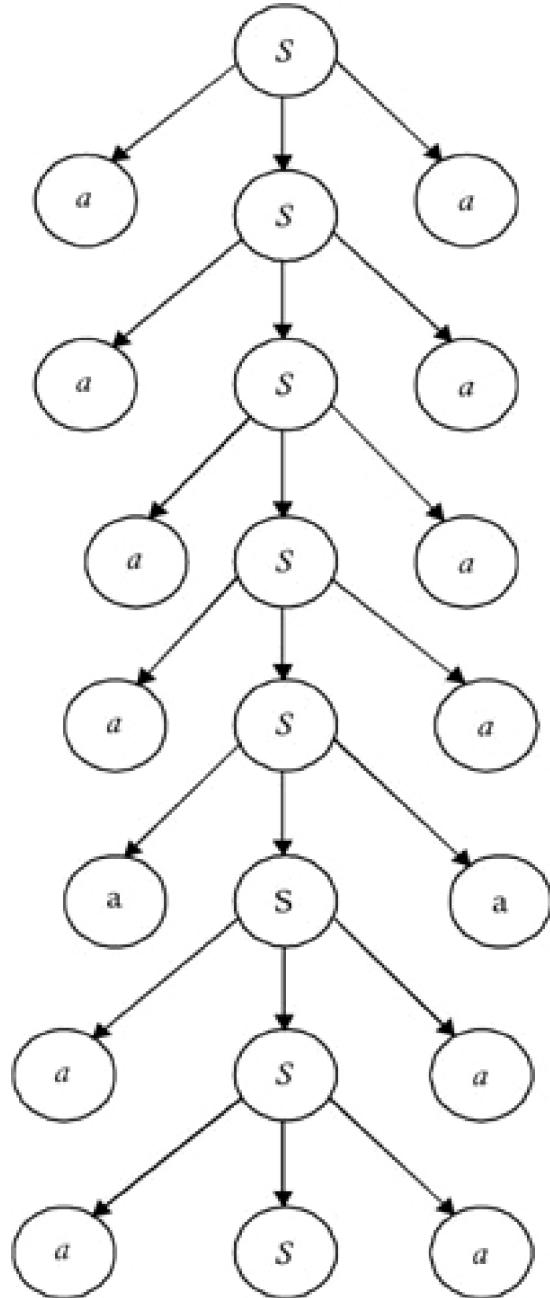
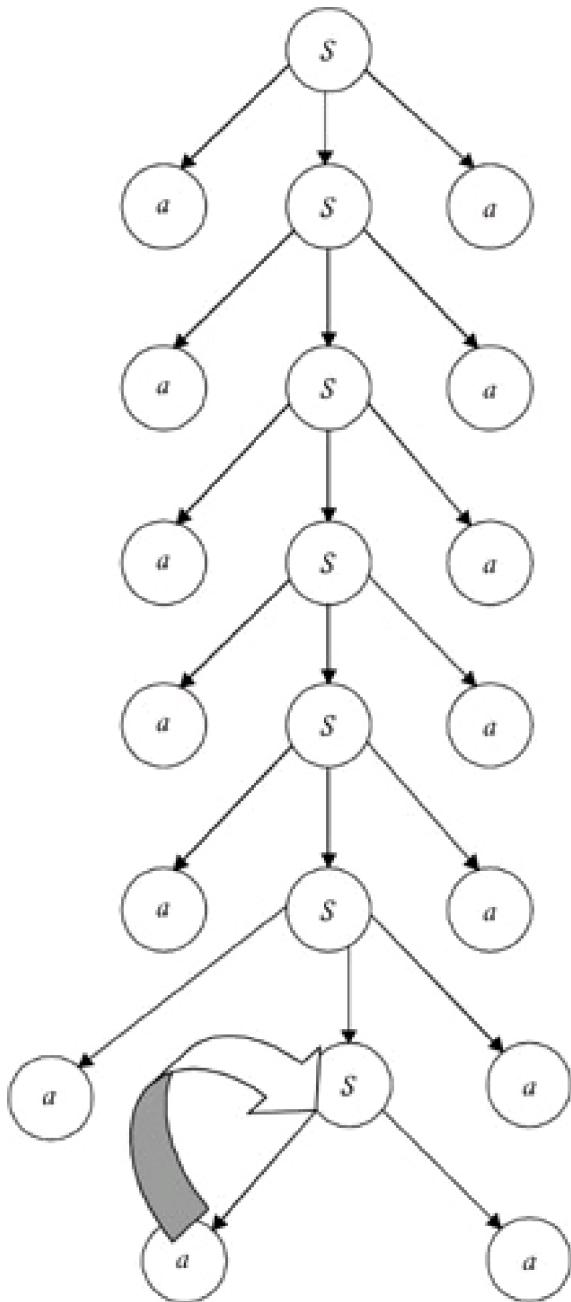


Figure 4.37: A match is found for the fifth input symbol, so the parser considers the next leaf, and expands S.



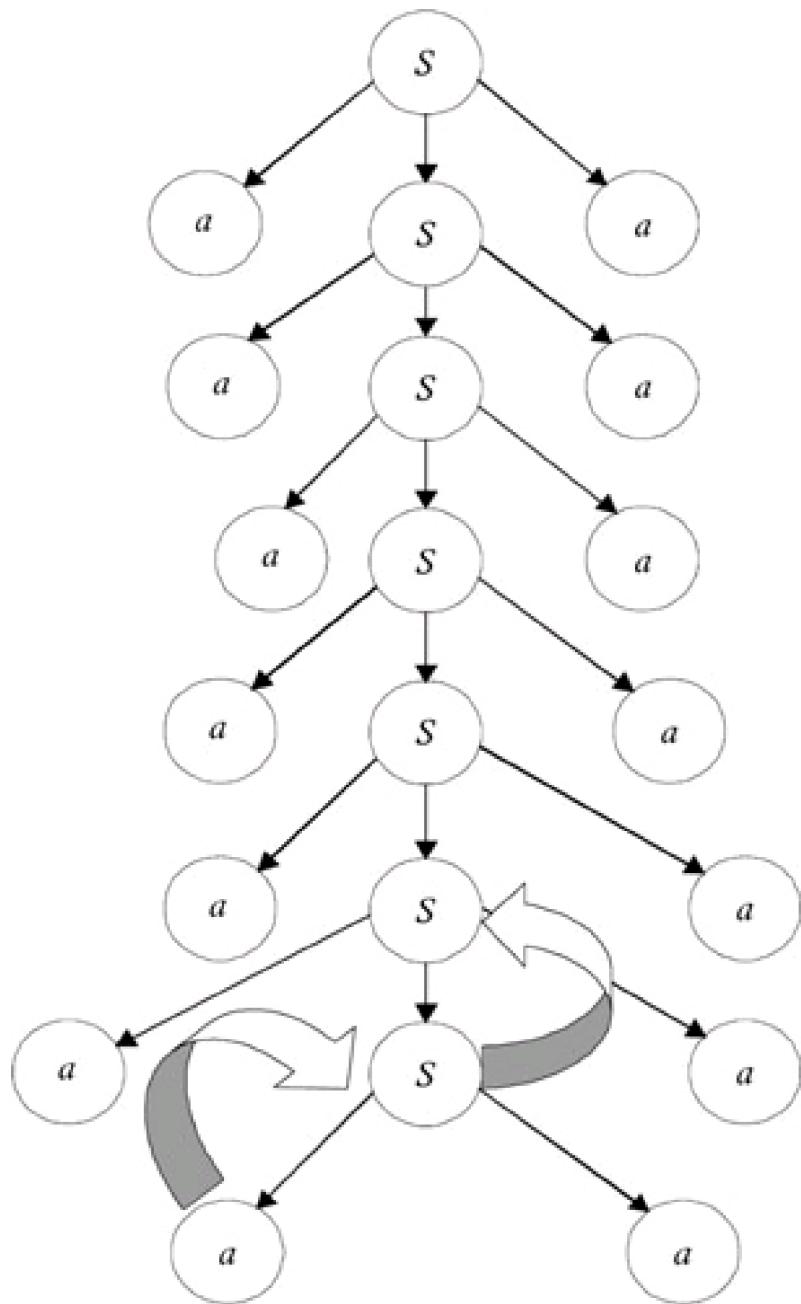
**Figure 4.38:** The sixth input symbol also matches. So the next leaf is considered, and S is expanded.

Now the parser finds that there is no match. Therefore, it will backtrack to S, as shown by the thick arrow in [Figure 4.39](#).



**Figure 4.39:** No match is found, so the parser backtracks to S.

Since there is no alternate of  $S$  that can be tried, the parser will backtrack one more step, as shown in [Figure 4.40](#). This procedure continues ([Figures 4.41](#) through [4.47](#)), until the parser tries the sixth alternate  $aa$  ([Figure 4.48](#)) and fails to find a match.



**Figure 4.40:** The parser backtracks one more step.

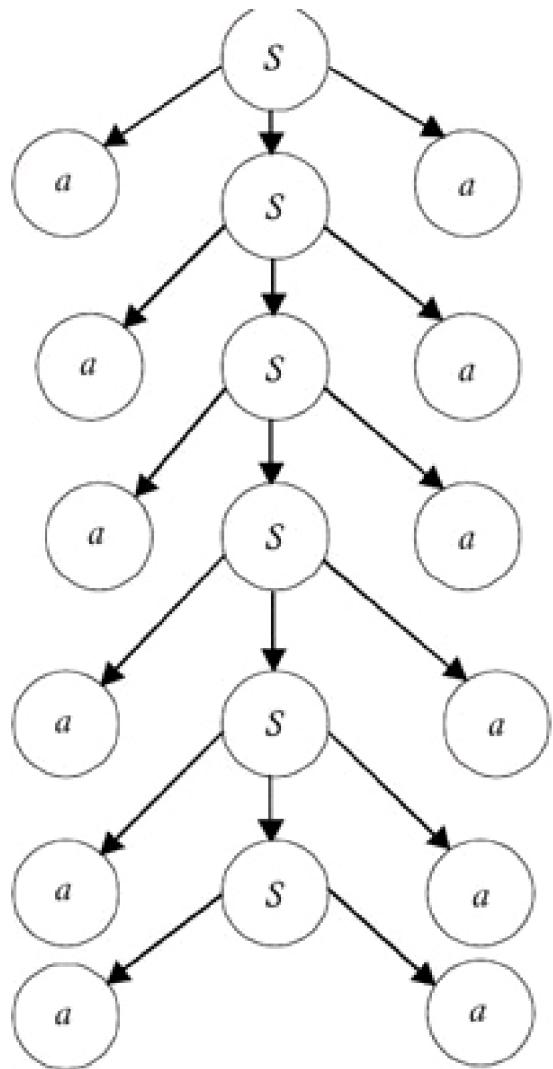
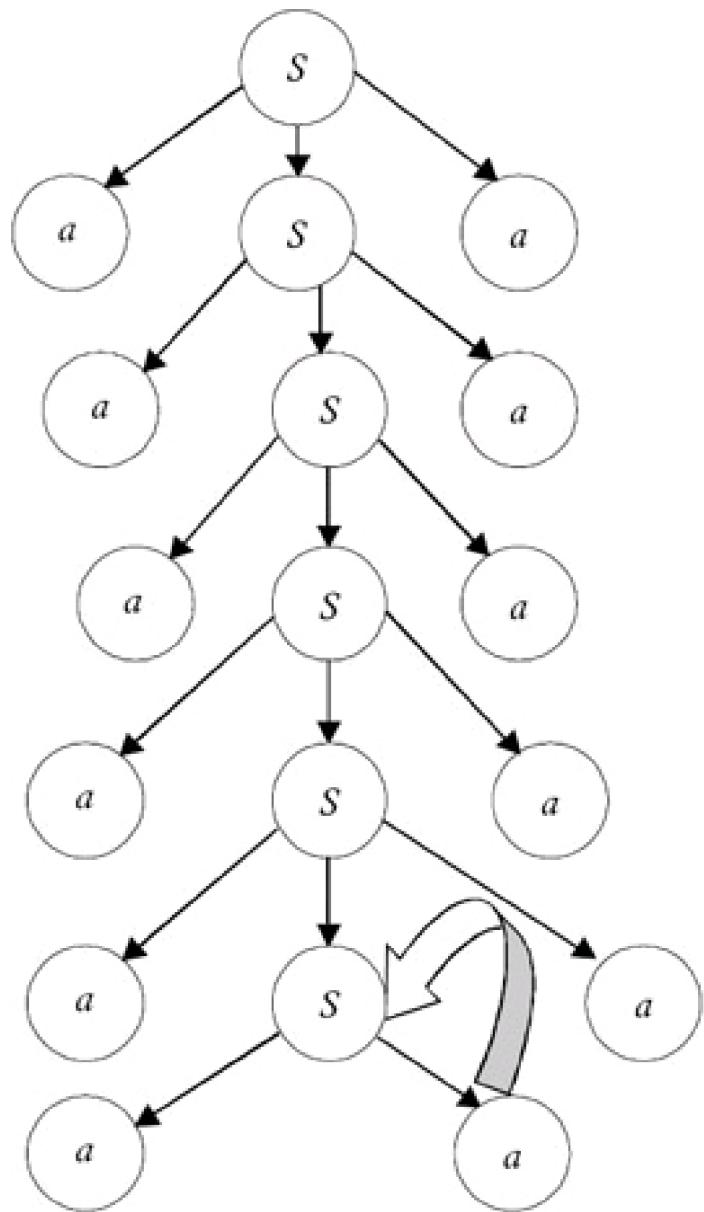
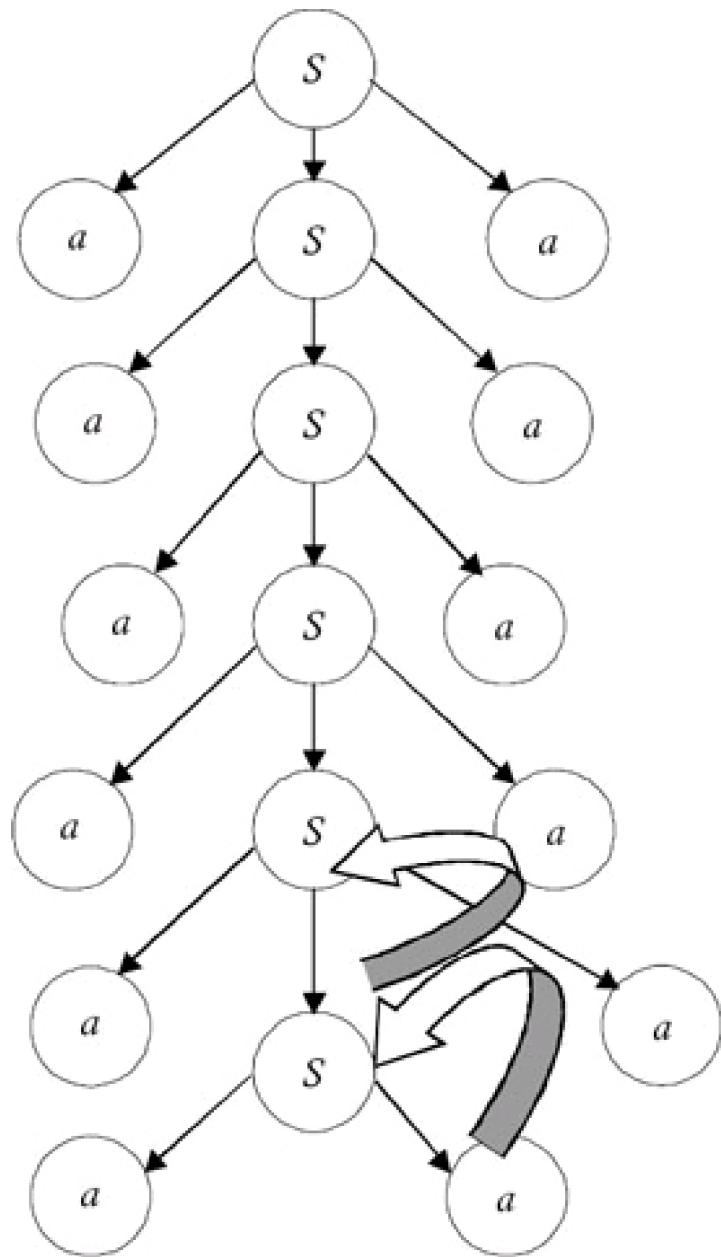


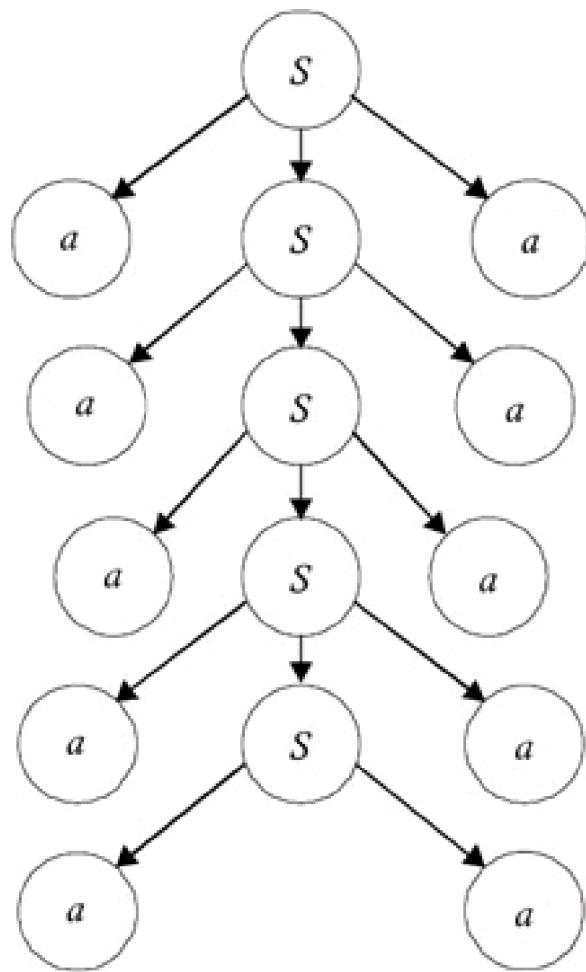
Figure 4.41: The parser tries the alternate aa.



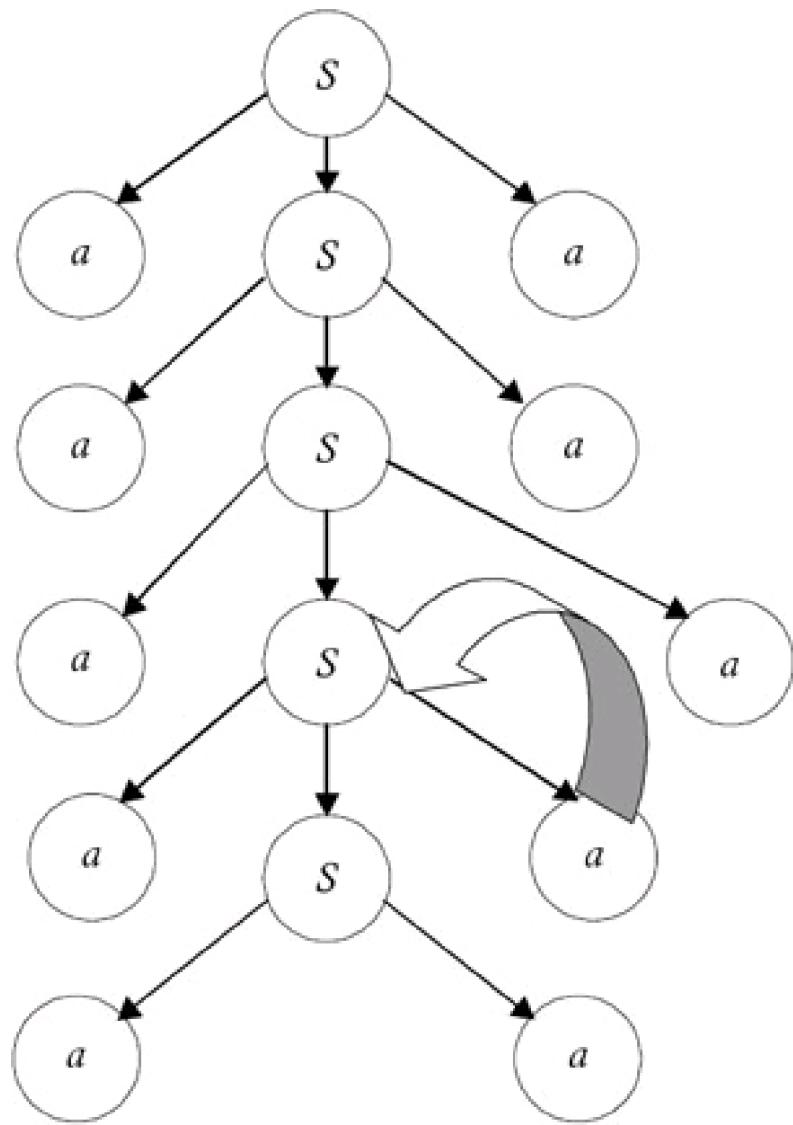
**Figure 4.42:** Again, a mismatch is found. So, the parser backtracks.



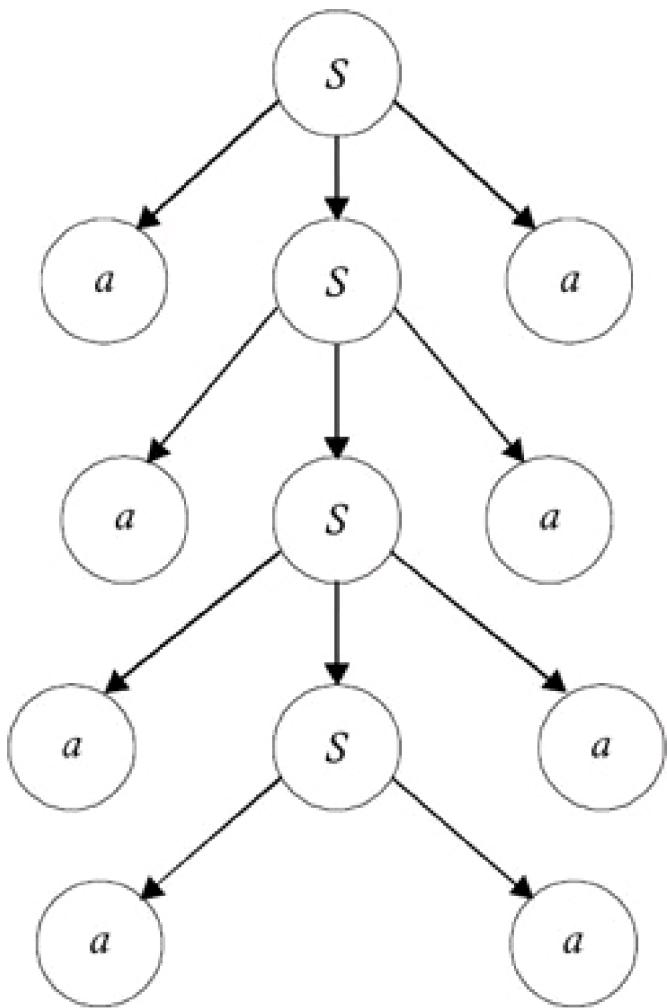
**Figure 4.43:** No alternate of  $S$  remains, so the parser will back-track one more step.



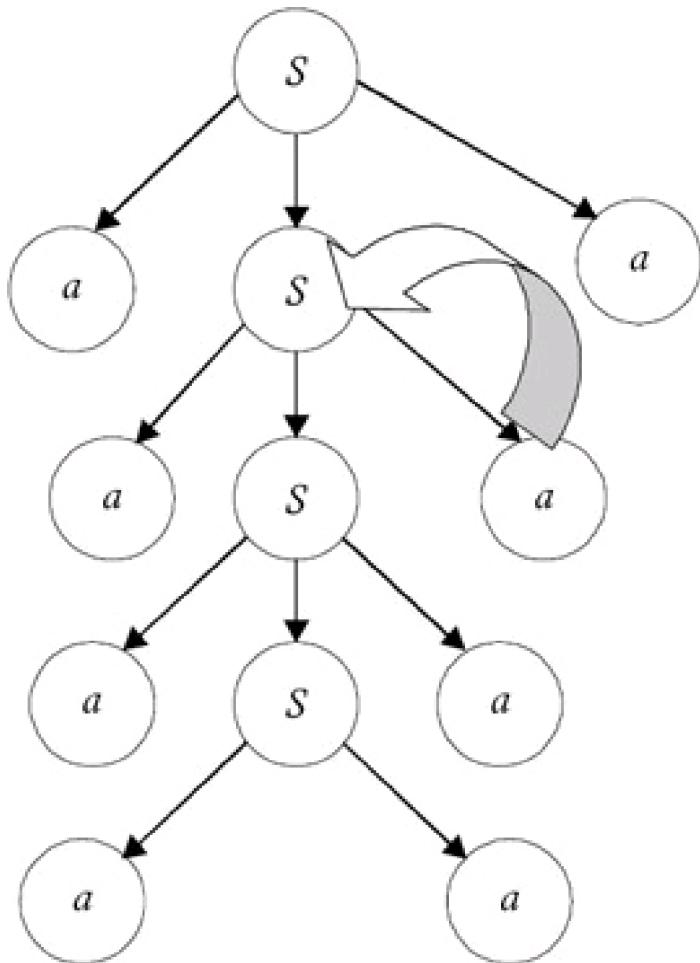
**Figure 4.44:** The parser tries an alternate aa.



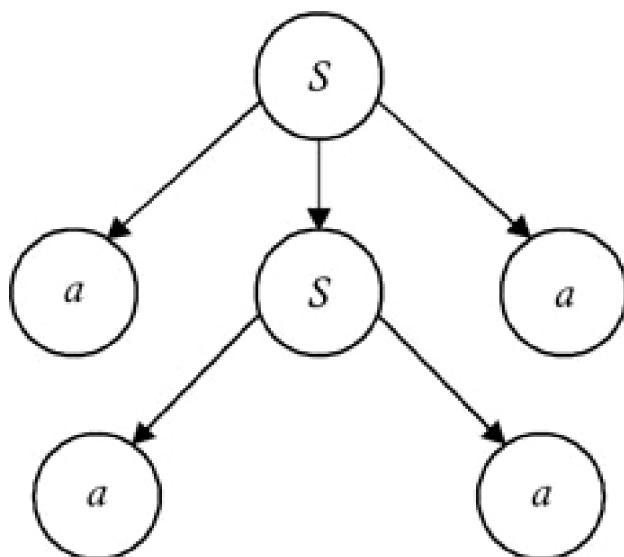
**Figure 4.45:** Again, a mismatch is found. The parser backtracks.



**Figure 4.46:** The parser then tries an alternate aa.



**Figure 4.47:** A mismatch is found, and the parser backtracks.



**Figure 4.48:** The parser tries for the alternate aa, fails to find a match, and cannot generate the parse tree for six occurrences of a.

### 4.3 THE PREDICTIVE TOP-DOWN PARSER

A backtracking parser is a non-deterministic recognizer of the language generated by the grammar. The backtracking problems in the top-down parser can be solved; that is, a top-down parser can function as a deterministic recognizer if it is capable of predicting or detecting which alternatives are right choices for the expansion of nonterminals (that derive to more than one alternative) during the parsing of input string  $w$ . By carefully writing a grammar, eliminating left recursion, and left-factoring the result, we obtain a grammar that can be parsed by a top-down parser. This grammar will be able to predict the right alternative for the expansion of a nonterminal during the parsing process; and hence, it need not backtrack.

If  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$  are the A-productions in the grammar, then a top-down parser can decide if a nonterminal  $A$  is to be expanded or not. And if it is to be expanded, the parser decides which  $A$ -production should be used. It looks at the next input symbol and finds out which of the  $\alpha_i$  derivatives to a string that start with the terminal symbol comes next in the input. If none of the  $\alpha_i$  derives to a string starting with a terminal symbol, the parser reports the failure; otherwise, it carries out the derivation of  $A$  using a production  $A \rightarrow \alpha_i$ , where  $\alpha_i$  derives to a string whose first terminal symbol is the symbol coming next in the input. Therefore, we conclude that if the set of first-terminal symbols of the strings derivable from  $\alpha_i$  is computed for each  $\alpha_i$ , and this set is made available to the parser, then the parser can predict the right choice for the expansion of nonterminal  $A$ . This information can be easily computed using the productions of the grammar. We define a function  $\text{FIRST}(\alpha)$ , where  $\alpha$  is in  $(V \cup T)^*$ , as follows:

$\text{FIRST}(\alpha)$  = Set of those terminals with which the strings derivable from  $\alpha$  start

If  $\alpha = XYZ$ , then  $\text{FIRST}(\alpha)$  is computed as follows:

$\text{FIRST}(\alpha) = \text{FIRST}(XYZ) = \{ X \}$  if  $X$  is terminal.

Otherwise,

$\text{FIRST}(\alpha) = \text{FIRST}(XYZ) = \text{FIRST}(X)$  if  $X$  does not derive to an empty string; that is, if

$\text{FIRST}(X)$  does not contain  $\epsilon$ .

If  $\text{FIRST}(X)$  contains  $\epsilon$ , then

$\text{FIRST}(\alpha) = \text{FIRST}(XYZ) = \text{FIRST}(X) - \{ \epsilon \} \cup \text{FIRST}(YZ)$

$\text{FIRST}(YZ)$  is computed in an identical manner:

$\text{FIRST}(YZ) = \{ Y \}$  if  $Y$  is terminal.

Otherwise,

$\text{FIRST}(YZ) = \text{FIRST}(Y)$  if  $Y$  does not derive to an empty string (i.e., if  $\text{FIRST}(Y)$  does not contain  $\epsilon$ ). If  $\text{FIRST}(Y)$  contains  $\epsilon$ , then

$\text{FIRST}(YZ) = \text{FIRST}(Y) - \{ \epsilon \} \cup \text{FIRST}(Z)$

For example, consider the grammar:

$$\begin{aligned} S &\rightarrow ACB \mid CbB \mid Ba \\ A &\rightarrow da \mid BC \\ B &\rightarrow g \mid \epsilon \\ C &\rightarrow h \mid \epsilon \end{aligned}$$

$\text{FIRST}(S) = \text{FIRST}(ACB) \cup \text{FIRST}(CbB) \cup$

$$\text{FIRST}(Ba) \quad (\text{I})$$

$$\text{FIRST}(A) = \text{FIRST}(da) \cup \text{FIRST}(BC)$$

$$= \{ d \} \cup \text{FIRST}(BC) \quad (\text{II})$$

$$\text{FIRST}(B) = \text{FIRST}(g) \cup \text{FIRST}(\in)$$

$$= \{ g, \in \}$$

$$\text{FIRST}(C) = \text{FIRST}(h) \cup \text{FIRST}(\in)$$

$$= \{ h, \in \}$$

Therefore:

$$\text{FIRST}(BC) = \text{FIRST}(B) - \{ \in \} \cup \text{FIRST}(C)$$

$$\begin{aligned} &= \{ g, \in \} - \{ \in \} \cup \{ h, \in \} \\ &= \{ g, h, \in \} \end{aligned}$$

Substituting in (II) we get:

$$\text{FIRST}(A) = \{ d \} \cup \{ g, h, \in \}$$

$$= \{ d, g, h, \in \}$$

$$\text{FIRST}(ACB) = \text{FIRST}(A) - \{ \in \} \cup \text{FIRST}(CB)$$

$$= \{ d, g, h, \in \} \cup \text{FIRST}(CB) \quad (\text{III})$$

$$\text{FIRST}(CB) = \text{FIRST}(C) - \{ \in \} \cup \text{FIRST}(B)$$

$$\begin{aligned} &= \{ h, \in \} - \{ \in \} \cup \{ g, \in \} \\ &= \{ g, h, \in \} \end{aligned}$$

Therefore, substituting in (III) we get:

$$\text{FIRST}(ACB) = \{ d, g, h, \in \} \cup \{ g, h, \in \}$$

$$= \{ d, g, h, \in \}$$

Similarly,

$$\text{FIRST}(CbB) = \text{FIRST}(C) - \{ \in \} \cup \text{FIRST}(bB)$$

$$\begin{aligned} &= \{ h, \in \} - \{ \in \} \cup \{ b \} \\ &= \{ b, h \} \{ b, h \} \end{aligned}$$

Similarly,

$$\text{FIRST}(Ba) = \text{FIRST}(B) - \{ \in \} \cup \text{FIRST}(a)$$

$$\begin{aligned} &= \{ g, \in \} - \{ \in \} \cup \{ a \} \\ &= \{ a, g \} \{ a, g \} \end{aligned}$$

Therefore, substituting in (I), we get:

$$\text{FIRST}(S) = \{ d, g, h, \in \} \cup \{ b, h, \in \} \cup \{ a, g, \in \}$$

$$= \{ a, b, d, g, h, \in \}$$

#### **EXAMPLE 4.2**

Consider the following grammar:

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow cd \mid ef \end{aligned}$$

$$\text{FIRST}(aAb) = \{ a \}$$

$$\text{FIRST}(cd) = \{ c \}, \text{ and}$$

$$\text{FIRST}(ef) = \{ e \}$$

Hence, while deriving  $S$ , the parser looks at the next input symbol. And if it happens to be the terminal  $a$ , then the parser derives  $S$  using  $S \rightarrow aAb$ . Otherwise, the parser reports an error. Similarly, when expanding  $A$ , the parser looks at the next input symbol; if it happens to be the terminal  $c$ , then the parser derives  $A$  using  $A \rightarrow cd$ . If the next terminal input symbol happens to be  $e$ , then the parser derives  $A$  using  $A \rightarrow ef$ . Otherwise, an error is reported.

Therefore, we conclude that if the right-hand FIRST for the production  $S \rightarrow aAb$  is computed, we can decide when the parser should do the derivation using the production  $S \rightarrow aAb$ . Similarly, if the right-hand FIRST for the productions  $A \rightarrow cd$  and  $A \rightarrow ef$  are computed, then we can decide when derivation is to be done using  $A \rightarrow cd$  and  $A \rightarrow ef$ , respectively. These decisions can be encoded in the form of table, as shown in [Table 4.1](#), and can be made available to the parser for the correct selection of productions for derivations during parsing.

**Table 4.1: Production Selections for Parsing Derivations**

	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>\$</b>
<b>S</b>	$S \rightarrow aAb$						
<b>A</b>			$A \rightarrow cd$		$A \rightarrow ef$		

The number of rows of the table are equal to the number of nonterminals, whereas the number of columns are equal to the number of terminals, including the end marker. The parser uses of the nonterminal to be derived as the row index of the table, and the next input symbol is used as the column index when the parser decides which production will be derived. Here, the production  $S \rightarrow aAb$  is added in the table at  $[S, a]$  because  $\text{FIRST}(aAb)$  contains a terminal  $a$ . Hence,  $S$  must be derived using  $S \rightarrow aAb$  if and only if the terminal symbol coming next in the input is  $a$ . Similarly, the production  $A \rightarrow cd$  is added at  $[A, c]$ , because  $\text{FIRST}(cd)$  contain  $c$ . Hence,  $A$  must be derived using  $A \rightarrow cd$  if and only if the terminal symbol coming next in the input is  $c$ . Finally,  $A$  must be derived using  $A \rightarrow ef$  if and only if the terminal symbol coming next in the input is  $e$ . Hence, the production  $A \rightarrow ef$  is added at  $[A, e]$ . Therefore, we conclude that the table can be constructed as follows:

for every production  $A \rightarrow \alpha$  do

    for every  $a$  in  $\text{FIRST}(\alpha)$  do

        TABLE[A, a] =  $A \rightarrow \alpha$

Using the above method, every production of the grammar gets added into the table at the proper place when the grammar is  $\in$ -free. But when the grammar is not  $\in$ -free,  $\in$ -productions will not get added to the table. If there is an  $\in$ -production  $A \rightarrow \in$  in the grammar, then deciding when  $A$  is to be derived to  $\in$  is not possible using the production's right-hand FIRST. Some additional information is required to decide where the production  $A \rightarrow \in$  is to be added to the table.

**Tip** The derivation by  $A \rightarrow \epsilon$  is a right choice when the parser is on the verge of expanding the nonterminal  $A$  and the next input symbol happens to be a terminal, which can occur immediately following  $A$  in any string occurring on the right side of the production. This will lead to the expansion of  $A$  to  $\epsilon$ , and the next leaf in the parse tree will be considered, which is labeled by the symbol immediately following  $A$  and, therefore, may match the next input symbol.

Therefore, we conclude that the production  $A \rightarrow \epsilon$  is to be added in the table at  $[A, b]$  for every  $b$  that immediately follows  $A$  in any of the production grammar's right-hand strings. To compute the set of all such terminals, we make use of the function  $\text{FOLLOW}(A)$ , where  $A$  is a nonterminal, as defined below:

$\text{FOLLOW}(A) = \text{Set of terminals that immediately follow } A \text{ in any string occurring on the right side of productions of the grammar}$

For example, if  $A \rightarrow \alpha B \beta$  is a production, then  $\text{FOLLOW}(B)$  can be computed using  $A \rightarrow \alpha B \beta$ , as shown below:

$\text{FOLLOW}(B) = \text{FIRST}(\beta)$  if  $\text{FIRST}(\beta)$  does not contain  $\epsilon$ .

$= \text{FIRST}(\beta) - \{\epsilon\} \text{ FOLLOW}(A)$   
when  $\text{FIRST}(\beta)$  contains  $\epsilon$ .

*/\* because when  $\beta$  derives to  $\epsilon$ , that time the terminal symbol immediately following  $A$ , will follow  $B^*$  \*/*

Therefore, we conclude that when the grammar is not  $\epsilon$ -free, then the table can be constructed as follows:

1. Compute FIRST and FOLLOW for every nonterminal of the grammar.

2. For every production  $A \rightarrow \alpha$ , do:

```
{
  for every non- $\epsilon$  member  $a$  in  $\text{FIRST}(\alpha)$  do
    TABLE[ $A, a$ ] =  $A \rightarrow \alpha$ 
    If  $\text{FIRST}(\alpha)$  contain  $\epsilon$  then
      For every  $b$  in  $\text{FOLLOW}(A)$  do
        TABLE[ $A, b$ ] =  $A \rightarrow \alpha$ 
}
```

Therefore, we conclude that if the table is constructed using the above algorithm, a top-down parser can be constructed that will be a nonbacktracking, or 'predictive' parser.

#### 4.3.1 Implementation of a Table-Driven Predictive Parser

A table-driven parser can be implemented using an input buffer, a stack, and a parsing table. The input buffer is used to hold the string to be parsed. The string is followed by a "\$" symbol that is used as a right-end marker to indicate the end of the input string. The stack is used to hold the sequence of grammar symbols. A "\$" indicates bottom of the stack. Initially, the stack has the start symbol of a grammar above the \$. The parsing table is a table obtained by using the above algorithm presented in the [previous section](#). It is a two-dimensional array  $\text{TABLE}[A, a]$ , where  $A$  is a nonterminal and  $a$  is a terminal, or \$ symbol. The parser is controlled by a program that behaves as follows:

1. The program considers  $X$ , the symbol on the top of the stack, and the next input symbol  $a$ .
2. If  $X = a = \$$ , then parser announces the successful completion of the parsing and halts.
3. If  $X = a \neq \$$ , then the parser pops the  $X$  off the stack and advances the input pointer to the next input symbol.
4. If  $X$  is a nonterminal, then the program consults the parsing table entry  $\text{TABLE}[x, a]$ . If  $\text{TABLE}[x, a] = x \rightarrow UVW$ , then the parser replaces  $X$  on the top of the stack by  $UVW$  in such a manner that  $U$  will come on the top. If  $\text{TABLE}[x, a] = \text{error}$ , then the parser calls the error-recovery routine.

For example consider the following grammar:

$$S \rightarrow aABb$$

$$A \rightarrow c \mid \epsilon$$

$$B \rightarrow d \mid \epsilon$$

$$\text{FOLLOW}(S) = \{ \$ \}$$

$$\text{FOLLOW}(A) = \text{FIRST}(Bb)$$

$$= \text{FIRST}(B) - \{ \epsilon \} \cup \text{FIRST}(b)$$

$$= \{ d, \epsilon \} - \{ \epsilon \} \cup \{ b \} = \{ d, b \}$$

$$\text{FOLLOW}(B) = \text{FIRST}(b) = \{ b \}$$

[Table 4.2](#)

Table 4.2: Production Selections for Parsing Derivations

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	\$

[Table 4.3](#)

Table 4.3: Steps Involved in Parsing the String *acdb*

Stack Contents	Unspent Input	Moves

Table 4.4

Table 4.4: Production Selections for String *ab* Parsing Derivations

Stack Contents	Unspent Input	Moves

Table 4.5

Table 4.5: Production Selections for Parsing Derivations for the String *adb*

Stack Contents	Unspent Input	Moves

the parser acts as a recognizer of only the members of  $L(G)$ ; hence, the grammar is  $LL(1)$ .

Therefore,  $LL(1)$  is the grammar for which an  $LL(1)$  parser can be constructed, which acts as a deterministic recognizer of  $L(G)$ . If a grammar is  $LL(1)$ , then a deterministic top-down table-driven recognizer can be constructed to recognize  $L(G)$ . A parsing table constructed for a given grammar  $G$  will have multiple entries if the grammar contains multiple productions that derive the same nonterminal—that is, the grammar contains the productions  $A \rightarrow \alpha \mid \beta$ , and both  $\alpha$  and  $\beta$  derive to a string that starts with the same terminal symbol. Therefore, one of the basic requirements for a grammar to be considered  $LL(1)$  is when the grammar contains multiple productions that derive the same nonterminal, such as:

for every pair of productions  $A \rightarrow \alpha \mid \beta$

$FIRST(\alpha) \cap FIRST(\beta) = \emptyset$  (i.e.,  $FIRST(\alpha)$  and  $FIRST(\beta)$  should be disjoint sets for every pair of productions  $A \rightarrow \alpha \mid \beta$ )

For a grammar to be  $LL(1)$ , the satisfaction of the condition above is necessary as well sufficient if the grammar is  $\in$ -free. When the grammar is not  $\in$ -free, then the satisfaction of the above condition is necessary but not sufficient, because either  $FIRST(\alpha)$  or  $FIRST(\beta)$  might contain  $\in$ , but not both. The above condition will still be satisfied; but if  $FIRST(\beta)$  contains  $\in$ , then production  $A \rightarrow \beta$  will be added in the table on all terminals in  $FOLLOW(A)$ . Hence, it also required that  $FIRST(\alpha)$  and  $FOLLOW(A)$  contain no common symbols. Therefore, an additional condition must be satisfied in order for a grammar to be  $LL(1)$ . When the grammar is not  $\in$ -free: for every pair of productions  $A \rightarrow \alpha \mid \beta$

if  $FIRST(\beta)$  contains  $\in$ , and  $FIRST(\alpha)$  does not contain  $\in$ , then

$FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

Therefore, for a grammar to be  $LL(1)$ , the following conditions must be satisfied:

For every pair of productions

```
{
  (1)  $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$ 
  and
  if  $FIRST(\beta)$  contains  $\in$ , and  $FIRST(\alpha)$  does not contain  $\in$ 
  then
  (1)  $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$ 
}
```

### 4.3.2 Examples

#### EXAMPLE 4.3

---

Test whether the grammar is  $LL(1)$  or not, and construct a predictive parsing table for it.

$$\begin{aligned} S &\rightarrow AaAb \mid BbBa \\ A &\rightarrow \in \\ B &\rightarrow \in \end{aligned}$$

Since the grammar contains a pair of productions  $S \rightarrow AaAb \mid BbBa$ , for the grammar to be  $LL(1)$ , it is required that:

$$\begin{aligned} FIRST(AaAb) \cap FIRST(BbBa) &= \emptyset \\ FIRST(AaAb) &= FIRST(A) - \{\in\} \cup FIRST(aAb) = \{a\} \\ FIRST(BbBa) &= FIRST(B) - \{\in\} \cup FIRST(bBa) = \{b\} \\ FIRST(AaAb) \cap FIRST(BbBa) &= \{a\} \cap \{b\} = \emptyset \end{aligned}$$

Hence, the grammar is  $LL(1)$ .

To construct a parsing table, the FIRST and FOLLOW sets are computed, as shown below:

$$\begin{aligned} FIRST(S) &= FIRST(AaAb) \cup FIRST(BbBa) \\ FIRST(S) &= \{a\} \cup \{b\} = \{a, b\} \end{aligned}$$

$\text{FIRST}(A) = \{ \in \}$

$\text{FIRST}(B) = \{ \in \}$   
 $\text{FOLLOW}(S) = \{ \$ \}$

$\text{FOLLOW}(A) = \text{FIRST}(aAb) = \{ a \}$ , and  
 $\text{FOLLOW}(A) = \text{FIRST}(b) = \{ b \}$ . Therefore,  
 $\text{FOLLOW}(A) = \{ a, b \}$

$\text{FOLLOW}(B) = \text{FIRST}(bBa) = \{ b \}$ , and  
 $\text{FOLLOW}(B) = \text{FIRST}(a) = \{ a \}$ . Therefore,  
 $\text{FOLLOW}(B) = \{ a, b \}$

---

Table 4.6: Production Selections for [Example 4.3](#) Parsing Derivations

	$a$	$b$	$\$$

**EXAMPLE 4.4**

---

$$\begin{aligned} S &\rightarrow 1AB \mid \in \\ A &\rightarrow 1AC \mid 0C \\ B &\rightarrow 0S \\ C &\rightarrow 1 \end{aligned}$$

---

$\text{FIRST}(1AB) \cap \text{FIRST}(\in) = \{ 1 \} \cap \{ \in \} = \emptyset$ , and  
 $\text{FIRST}(1AB) \cap \text{FOLLOW}(S) = \{ 1 \} \cap \{ \$ \} = \emptyset$

$\text{FIRST}(1AC) \cap \text{FIRST}(0C) = \{ 1 \} \cap \{ 0 \} = \emptyset$

If a grammar is left-recursive, then the set of productions will contain at least one pair of the form  $A \rightarrow A\alpha \mid \beta$ ; and hence,  $\text{FIRST}(A\alpha)$  and  $\text{FIRST}(\beta)$  will not be disjointed sets, because everything in the  $\text{FIRST}(\beta)$  will also be in the  $\text{FIRST}(A\alpha)$ . It thereby violates the condition for  $LL(1)$  grammar. Hence, a grammar containing a pair of productions  $A \rightarrow A\alpha \mid \beta$  (i.e., a left-recursive grammar) cannot be  $LL(1)$ .

Now, let  $X$  be a nullable nonterminal that derives to at least two terminal strings. Show that in  $LL(1)$  grammar, no production rule can have two consecutive occurrences of  $X$  on the right side of the production.

Since  $X$  is a nullable  $X \in \text{FOLLOW}(X)$ ,  $X$  is also deriving to at least two terminal strings  $Xw_1$  and  $Xw_2$  where  $w_1$  and  $w_2$  are the strings of terminals. Therefore, for a grammar using  $X$  to be  $LL(1)$ , it is required that:

$$\text{FIRST}(w_1) \cap \text{FIRST}(w_2) = \emptyset$$

$$\text{FIRST}(w_1) \cap \text{FOLLOW}(X) \text{ and } \text{FIRST}(w_2) \cap \text{FOLLOW}(X) = \emptyset$$

If this grammar contains a production rule  $A \rightarrow \alpha XX\beta$ -a production whose right side has two consecutive occurrences of  $X$ -then everything in  $\text{FIRST}(X)$  will also be in the  $\text{FOLLOW}(X)$ ; and since  $\text{FIRST}(X)$  contains  $\text{FIRST}(w_1)$  as well as  $\text{FIRST}(w_2)$ , the second condition will therefore not be satisfied. Hence, a grammar containing a production of the form  $A \rightarrow \alpha XX\beta$  will never be  $LL(1)$ , thereby proving that in  $LL(1)$  grammar, no production rule can have two consecutive occurrences of  $X$  on the right side of the production.

#### **EXAMPLE 4.5**

---

Construct a predictive parsing table for the following grammar where  $S|$  is a start symbol and  $\#$  is the end marker.

$$\begin{aligned} S| &\rightarrow S\# \\ S &\rightarrow qABC \\ A &\rightarrow a \mid bbD \\ B &\rightarrow a \mid \epsilon \\ C &\rightarrow b \mid \epsilon \\ D &\rightarrow c \mid \epsilon \end{aligned}$$


---

Here,  $\#$  is taken as one of the grammar symbols. And therefore, the initial configuration of the parser will be  $(S|, w\#)$ , where the first member of the pair is the contents of the stack and the second member is the contents of input buffer.

$$\begin{aligned} \text{FIRST}(S|) &= \text{FIRST}(S\#) && \text{(I)} \\ \text{FIRST}(S) &= \text{FIRST}(qABC) = \{ q \} \end{aligned}$$

Therefore, by substituting in (I), we get:

$$\begin{aligned} \text{FIRST}(S|) &= \{ q \} \\ \text{FIRST}(A) &= \text{FIRST}(a) \cup \text{FIRST}(bbD) = \{ a \} \cup \{ b \} \\ &= \{ a, b \} \\ \text{FIRST}(B) &= \text{FIRST}(a) \cup \text{FIRST}(\epsilon) = \{ a, \epsilon \} \\ \text{FIRST}(C) &= \text{FIRST}(b) \cup \text{FIRST}(\epsilon) = \{ b, \epsilon \} \\ \text{FIRST}(D) &= \text{FIRST}(c) \cup \text{FIRST}(\epsilon) = \{ c, \epsilon \} \end{aligned}$$

$$\text{FOLLOW}(S|) = \{ \ }$$

1. Using  $S| \rightarrow S\#$  we get:

$$\text{FOLLOW}(S) = \{ \# \}$$

2. Using  $S \rightarrow qABC$  we get:

$$\begin{aligned}\text{FOLLOW}(A) &= \text{FIRST}(BC) - \{\in\} \cup \text{FOLLOW}(S) \\ \text{FIRST}(BC) &= \text{FIRST}(B) - \{\in\} \cup \text{FIRST}(C) \\ &= \{a, \in\} - \{\in\} \{b\} = \{a, b, \in\}\end{aligned}\tag{II}$$

Substituting in (II) we get:

$$\begin{aligned}\text{FOLLOW}(A) &= \{a, b, \in\} - \{\in\} \cup \{\#\} = \{a, b, \#\} \\ \text{FOLLOW}(B) &= \text{FIRST}(C) - \{\in\} \cup \text{FOLLOW}(S) \\ &= \{b, \in\} - \{\in\} \cup \{\#\} = \{b, \#\} \\ \text{FOLLOW}(C) &= \text{FOLLOW}(S) = \{\#\}\end{aligned}$$

3. Using  $A \rightarrow bbD$  we get:

$$\text{FOLLOW}(D) = \text{FOLLOW}(A) = \{a, b, \#\}$$

Therefore, the parsing table is derived as shown in [Table 4.7](#).

**Table 4.7: Production Selections for Example 4.5 Parsing Derivations**

	<b><i>q</i></b>	<b><i>a</i></b>	<b><i>b</i></b>	<b><i>c</i></b>	<b><i>#</i></b>
<i>S</i>	$S \rightarrow S\#$				
<i>S</i>	$S \rightarrow qabc$				
<i>A</i>		$A \rightarrow a$	$A \rightarrow bbD$		
<i>B</i>		$B \rightarrow a$	$B \rightarrow \in$		$B \rightarrow \in$
<i>C</i>			$C \rightarrow b$		$C \rightarrow \in$
<i>D</i>		$D \rightarrow \in$	$D \rightarrow \in$	$D \rightarrow c$	$D \rightarrow \in$

#### **EXAMPLE 4.6**

---

Construct predictive parsing table for the following grammar:

$$\begin{aligned}S &\rightarrow A \\ A &\rightarrow aB \mid Ad \\ B &\rightarrow bBC \mid f \\ C &\rightarrow g \\ \text{FIRST}(S) &= \text{FIRST}(A) = \{a\} \\ \text{FIRST}(B) &= \{b, f\} \\ \text{FIRST}(C) &= \{g\}\end{aligned}$$


---

Since the grammar is  $\in$ -free, FOLLOW sets are not required to be computed in order to enter the productions into the parsing table. Therefore the parsing table is as shown in [Table 4.8](#).

**Table 4.8: Production Selections for Example 4.6 Parsing Derivations**

	<i>a</i>	<i>b</i>	<i>f</i>	<i>g</i>	<i>d</i>
<i>S</i>	$S \rightarrow A$				
<i>A</i>	$A \rightarrow aS$			$A \rightarrow d$	
<i>B</i>		$B \rightarrow bBC$	$B \rightarrow f$		
<i>C</i>				$C \rightarrow g$	

#### EXAMPLE 4.7

Construct a predictive parsing table for the following grammar, where  $S$  is a start symbol.

$$S \rightarrow iEtSS_1 \mid a$$

$$S_1 \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

$$\text{FIRST}(S) = \text{FIRST}(iEtSS_1) \cup \text{FIRST}(a) = \{ i, a \}$$

$$\text{FIRST}(S_1) = \text{FIRST}(eS) \cup \text{FIRST}(\epsilon) = \{ e, \epsilon \}$$

$$\text{FIRST}(E) = \text{FIRST}(b) = \{ b \}$$

$$\text{FOLLOW}(S) = \{ \$ \}$$

1. Using  $S \rightarrow iEtSS_1$ :

$$\text{FOLLOW}(E) = \{ t \}$$

$$\begin{aligned} \text{FOLLOW}(S) &= \text{FIRST}(S_1) - \{ \epsilon \} \cup \text{FOLLOW}(S) \\ &= \{ e, \epsilon \} - \{ \epsilon \} \cup \{ \$ \} \\ &= \{ e, \$ \} \end{aligned}$$

$$\text{FOLLOW}(S_1) = \text{FOLLOW}(S) = \{ e, \$ \}$$

2. Using  $S_1 \rightarrow eS$ :

$$\text{FOLLOW}(S) = \text{FOLLOW}(S_1) = \{ e, \$ \}$$

Therefore, the parsing table is as shown in [Table 4.9](#).

**Table 4.9: Production Selections for Example 4.7 Parsing Derivations**

	<i>i</i>	<i>a</i>	<i>b</i>	<i>e</i>	<i>T</i>	$\$$
<i>S</i>	$S \rightarrow iEtSS_1$	$S \rightarrow a$				
<i>S</i> <sub>1</sub>				$S_1 \rightarrow eS$		$S_1 \rightarrow \epsilon$
<i>S</i> <sub>1</sub>				$S_1 \rightarrow \epsilon$		
<i>E</i>			$E \rightarrow b$			

#### EXAMPLE 4.8

$S \rightarrow aBDh$   
 $B \rightarrow cC$   
 $C \rightarrow bC \mid \epsilon$   
 $D \rightarrow EF$   
 $E \rightarrow g \mid \epsilon$   
 $F \rightarrow f \mid \epsilon$

---

$\text{FIRST}(S) = \text{FIRST}(aBDh) = \{ a \}$   
 $\text{FIRST}(B) = \text{FIRST}(cC) = \{ c \}$   
 $\text{FIRST}(C) = \text{FIRST}(bC) \cup \text{FIRST}(\epsilon) = \{ b \} \cup \{ \epsilon \} = \{ b, \epsilon \}$   
 $\text{FIRST}(D) = \text{FIRST}(EF) = \text{FIRST}(E) - \{ \epsilon \} \cup \text{FIRST}(F)$  (I)  
 $\text{FIRST}(E) = \text{FIRST}(g) \cup \text{FIRST}(\epsilon) = \{ g, \epsilon \}$   
 $\text{FIRST}(F) = \text{FIRST}(f) \cup \text{FIRST}(\epsilon) = \{ f, \epsilon \}$

$\text{FIRST}(D) = \{ g, \epsilon \} - \{ \epsilon \} \cup \{ f, \epsilon \} = \{ g, f, \epsilon \}$   
 $\text{FOLLOW}(S) = \{ \$ \}$

$\text{FOLLOW}(B) = \text{FIRST}(Dh) = \text{FIRST}(D) - \{ \epsilon \} \cup \text{FIRST}(h)$   
 $= \{ g, f, \epsilon \} - \{ \epsilon \} \cup \{ h \}$   
 $= \{ g, f, h \}$   
 $\text{FOLLOW}(D) = \text{FIRST}(h) = \{ h \}$

$\text{FOLLOW}(C) = \text{FOLLOW}(B) = \{ g, f, h \}$

$\text{FOLLOW}(C) = \text{FOLLOW}(C) = \{ g, f, h \}$

$\text{FOLLOW}(E) = \text{FIRST}(F) - \{ \epsilon \} \cup \text{FOLLOW}(D)$   
 $= \{ f, \epsilon \} - \{ \epsilon \} \cup \{ h \} = \{ f, h \}$   
 $\text{FOLLOW}(F) = \text{FOLLOW}(D) = \{ h \}$

Table 4.10

**Table 4.10: Production Selections for Example 4.8 Parsing Derivations**

	<i>a</i>	<i>b</i>	<i>c</i>	<i>g</i>	<i>f</i>	<i>h</i>	\$
<i>S</i>	$S \rightarrow aBDh$						
<i>B</i>			$B \rightarrow cC$				
<i>C</i>		$C \rightarrow bC$		$C \rightarrow \epsilon$	$C \rightarrow \epsilon$	$C \rightarrow \epsilon$	
<i>D</i>				$D \rightarrow EF$	$D \rightarrow EF$	$D \rightarrow EF$	
<i>E</i>				$E \rightarrow g$	$E \rightarrow \epsilon$	$E \rightarrow \epsilon$	
<i>F</i>					$F \rightarrow f$	$F \rightarrow \epsilon$	

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Chapter 5: Bottom-up Parsing

### 5.1 WHAT IS BOTTOM-UP PARSING?

Bottom-up parsing can be defined as an attempt to reduce the input string  $w$  to the start symbol of a grammar by tracing out the right-most derivations of  $w$  in reverse. This is equivalent to constructing a parse tree for the input string  $w$  by starting with leaves and proceeding toward the root—that is, attempting to construct the parse tree from the bottom, up. This involves searching for the substring that matches the right side of any of the productions of the grammar. This substring is replaced by the left-hand-side nonterminal of the production if this replacement leads to the generation of the sentential form that comes one step before in the right-most derivation. This process of replacing the right side of the production by the left side nonterminal is called "reduction". Hence, reduction is nothing more than performing derivations in reverse. The reason why bottom-up parsing tries to trace out the right-most derivations of an input string  $w$  in reverse and not the left-most derivations is because the parser scans the input string  $w$  from the left to right, one symbol/token at a time. And to trace out right-most derivations of an input string  $w$  in reverse, the tokens of  $w$  must be made available in a left-to-right order. For example, if the right-most derivation sequence of some  $w$  is:

$$S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \dots \rightarrow \alpha_{n-1} \rightarrow w$$

then the bottom-up parser starts with  $w$  and searches for the occurrence of a substring of  $w$  that matches the right side of some production  $A \rightarrow \beta$  such that the replacement of  $\beta$  by  $A$  will lead to the generation of  $\alpha_{n-1}$ . The parser replaces  $\beta$  by  $A$ , then it searches for the occurrence of a substring of  $\alpha_{n-1}$  that matches the right side of some production  $B \rightarrow \gamma$  such that replacement of  $\gamma$  by  $B$  will lead to the generation of  $\alpha_{n-2}$ . This process continues until the entire  $w$  substring is reduced to  $S$ , or until the parser encounters an error.

Therefore, bottom-up parsing involves the selection of a substring that matches the right side of the production, whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a right-most derivation. That is, it leads to the generation of the previous right-most derivation. This means that selecting a substring that matches the right side of production is not enough; the position of this substring in the sentential form is also important.

**Tip** The substring should occur in the position and sentential form that is currently under consideration and, if it is replaced by the left-side nonterminal of the production, that it leads to the generation of the previous right-hand sentential form of the currently considered sentential form. Therefore, finding a substring that matches the right side of a production, as well as its position in the current sentential form, are both equally important. In order to take both of these factors into account, we will define a "handle" of the right sentential form.

## 5.2 A HANDLE OF A RIGHT SENTENTIAL FORM

A handle of a right sentential form  $\gamma$  is a production  $A \rightarrow \beta$  and a position of  $\beta$  in  $\gamma$ . The string  $\beta$  will be found and replaced by  $A$  to produce the previous right sentential form in the right-most derivation of  $\gamma$ . That is, if  $S \rightarrow \alpha A \beta \rightarrow \alpha \gamma \beta$ , then  $A \rightarrow \gamma$  is a handle of  $\alpha \gamma \beta$ , in the position following  $\alpha$ . Consider the grammar:

$$E \rightarrow E+E \mid E^*E \mid id$$

and the right-most derivation:

$$E \rightarrow E+E \rightarrow E+E^*E \rightarrow E+E+id \rightarrow E+id * id \rightarrow id + id * id$$

The handles of the sentential forms occurring in the above derivation are shown in [Table 5.1](#).

**Table 5.1: Sentential Form Handles**

Sentential Form	Handle
id + id * id	$E \rightarrow id$ at the position preceding +
E + id * id	$E \rightarrow id$ at the position following +
E + E * id	$E \rightarrow id$ at the position following *
E + E * E	$E \rightarrow E^*E$ at the position following +
E + E	$E \rightarrow E+E$ at the position preceding the end marker

Therefore, the bottom-up parsing is only an attempt to detect the handle of a right sentential form. And whenever a handle is detected, the reduction is performed. This is equivalent to performing right-most derivations in reverse and is called "handle pruning".

Therefore, if the right-most derivation sequence of some  $w$  is  $S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \dots \rightarrow \alpha_{n-1} \rightarrow w$ , then handle pruning starts with  $w$ , the  $n$ th right sentential form, the handle  $\beta_n$  of  $w$  is located, and  $\beta_n$  is replaced by the left side of some production  $A_n \rightarrow \beta_n$  in order to obtain  $\alpha_{n-1}$ . By continuing this process, if the parser obtains a right sentential form that consists of only a start symbol, then it halts and announces the successful completion of parsing.

### EXAMPLE 5.1

Consider the following grammar, and show the handle of each right sentential form for the string  $(a, (a, a))$ .

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow L, S \mid S \end{aligned}$$

The right-most derivation of the string  $(a, (a, a))$  is:

$$\begin{aligned} S &\rightarrow (L) \rightarrow (L, S) \rightarrow (L, (L)) \rightarrow (L, (L, S)) \rightarrow (L, (L, a)) \\ &\rightarrow (L, (S, a)) \rightarrow (L, (a, a)) \rightarrow (S, (a, a)) \rightarrow (a, (a, a)) \end{aligned}$$

[Table 5.2](#) presents the handles of the sentential forms occurring in the above derivation.

**Table 5.2: Sentential Form Handles**

Sentential Form	Handle

---

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## 5.3 IMPLEMENTATION

A convenient way to implement a bottom-up parser is to use a shift-reduce technique: a parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack. When a handle appears on the top of the stack, it performs reduction. This implementation makes use of a stack to hold grammar symbols and an input buffer to hold the string  $w$  to be parsed, which is terminated by the right endmarker  $\$$ , the same symbol used to mark the bottom of the stack. The configuration of the parser is given by a token pair—the first component of which is a stack content, and second component is an unexpanded input.

Initially, the parser will be in the configuration given by the pair  $(\$, w\$)$ ; that is, the stack is initially empty, and the buffer contains the entire string  $w$ . The parser shifts zero or more symbols from the input on to the stack until handle  $\alpha$  appears on the top of the stack. The parser then reduces  $\alpha$  to the left side of the appropriate production. This cycle is repeated until the parser either detects an error or until the stack contains a start symbol and the input is empty, giving the configuration  $(\$S, \$)$ . If the parser enters  $(\$S, \$)$ , then it announces the successful completion of parsing. Thus, the primary operation of the parser is to shift and reduce.

For example consider the bottom-up parser for the grammar having the productions:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow id \end{aligned}$$

and the input string:  $id + id * id$ . The various steps in parsing this string are shown in [Table 5.3](#) in terms of the contents of the stack and unspent input.

**Table 5.3: Steps in Parsing the String  $id + id * id$**

Stack Contents	Input	Moves
$\$$	$id + id * id \$$	shift $id$
$\$id$	$+ id * id \$$	reduce by $F \rightarrow id$
$\$F$	$+ id * id \$$	reduce by $T \rightarrow F$
$\$T$	$+ id * id \$$	reduce by $E \rightarrow T$
$\$E$	$+ id * id \$$	shift $+$
$\$E +$	$id * id \$$	shift $id$
$\$E + id$	$* id \$$	reduce by $F \rightarrow id$
$\$E + F$	$* id \$$	reduce by $T \rightarrow F$
$\$E + T$	$* id \$$	shift $*$
$\$E + T$	$* id \$$	shift $id$
$\$E + T * id$	$\$$	reduce by $F \rightarrow id$
$\$E + T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$E + T$	$\$$	reduce by $E \rightarrow E + T$
$\$E$	$\$$	accept

Shift-reduce implementation does not tell us anything about the technique used for detecting the handles; hence, it is possible to make use of any suitable technique to detect handles. Depending upon the technique that is used to detect handles, we get different shift-reduce parsers. For example, an operator-precedence parser is a shift-reduce parser that uses the precedence relationship between certain pairs of terminals to guide the selection of handles. Whereas LR parsers make use of a deterministic finite automata that recognizes the set of all viable prefixes; by reading the stack from bottom to top, it determines what handle, if any, is on the top of the stack.

## 5.4 THE LR PARSER

The LR parser is a shift-reduce parser that makes use of a deterministic finite automata, recognizing the set of all viable prefixes by reading the stack from bottom to top. It determines what handle, if any, is available. A viable prefix of a right sentential form is that prefix that contains a handle, but no symbol to the right of the handle. Therefore, if a finite-state machine that recognizes viable prefixes of the right sentential forms is constructed, it can be used to guide the handle selection in the shift-reduce parser.

Since the LR parser makes use of a DFA that recognizes viable prefixes to guide the selection of handles, it must keep track of the states of the DFA. Hence, the LR parser stack contains two types of symbols: state symbols used to identify the states of the DFA and grammar symbols. The parser starts with the initial state of a DFA 10 on the stack. The parser operates by looking at the next input symbol  $a$  and the state symbol  $l_i$  on the top of the stack. If there is a transition from the state  $l_i$  on  $a$  in the DFA going to state  $l_j$ , then it shifts the symbol  $a$ , followed by the state symbol  $l_j$ , onto the stack. If there is no transition from  $l_i$  on  $a$  in the DFA, and if the state  $l_i$  on the top of the stack recognizes, when entered, a viable prefix that contains the handle  $A \rightarrow \alpha$ , then the parser carries out the reduction by popping  $\alpha$  and pushing  $A$  onto the stack. This is equivalent to making a backward transition from  $l_i$  on  $\alpha$  in the DFA and then making a forward transition on  $A$ . Every shift action of the parser corresponds to a transition on a terminal symbol in the DFA. Therefore, the current state of the DFA and the next input symbol determine whether the parser shifts the next input symbol goes for reduction.

If we construct a table mapping every state and input symbol pair as either "shift," "reduce," "accept," or "error," we get a table that can be used to guide the parsing process. Such a table is called a parsing "action" table. When carrying out the reduction by  $A \rightarrow \alpha$ , the parser has to pop  $\alpha$  and push  $A$  onto the stack. This requires knowledge of where the transition goes in a DFA from the state brought onto the top of the stack after popping  $\alpha$  on the nonterminal  $A$ ; and hence, we require another table mapping of every state and nonterminal pair into a state. The table of transitions on the nonterminals in the DFA is called a "goto" table. Therefore, to create an LR parser we require an Action|GOTO table.

If the current state of a DFA has a transition on the terminal symbol  $a$  to the state  $l_j$ , then the next move will be to shift the symbol  $a$  and enter the state  $l_j$ . But if the current state of the DFA is one in which when entered recognizes that a viable prefix contains the handle, then the next move of the parser will be to reduce.

Therefore, an LR parser is comprised of an input buffer (which holds the input string  $w$  to be parsed and assumed to be terminated by the right endmarker \$), a stack holding the viable prefixes of the right sentential forms, and a parsing table that is obtain by mapping the moves of a DFA that recognizes viable prefixes and controls the parsing actions. The configuration of a parser is given by a token pair: the first component is a stack's content, and second component is unexpanded input. If, at a particular instant (and \$ is used as bottom-of-the-stack marker, also), a parser is configured as follows:

Stack Contents	Input
$\$I_0X_0I_1X_1 \dots X_mI_m$	$a_ia_{i+1} \dots a_n\$$

where  $I_i$  is a state symbol identifying the state of a DFA recognizing the viable prefixes, and  $X_i$  is the grammar symbol. The parser consults the parsing action table entry,  $[l_m, a_i]$ . If  $\text{action}[l_m, a_i] = S_j$ , then the parser shifts the next input symbol followed by the state  $l_j$  on the stack and enters into the configuration:

Stack Contents	Input
$\$I_0X_0I_1X_1 \dots X_mI_ma_i I_j$	$a_{i+1} \dots a_n\$$

If  $\text{action}[l_m, a_i] = \text{reduce by production } A \rightarrow \alpha$ , then the parser carries out the reduction as follows. If  $|\alpha| = r$ , then the parser pops two  $r$  symbols from the stack (because every shift action shifts a grammar symbol as well as state symbol), thereby bringing  $l_{m-r}$  on the top. It then consults the goto table entry,  $\text{goto}[l_{m-r}, A]$ . If  $\text{goto}[l_{m-r}, A] = l_k$ , then

it shifts  $A$  followed by  $I_k$  onto the stack, thereby entering into the configuration:

<b>Stack Contents</b>	<b>Input</b>
$\$I_0X_0I_1X_1 \dots X_{m-r}I_{m-r}AI_k$	$a_i a_{i+1} \dots a_n \$$

If  $\text{action}[I_m, a_i] = \text{accept}$ , then the parser halts and accepts the input string. If  $\text{action}[I_m, a_i] = \text{error}$ , then the parser invokes a suitable error-recovery routine. Initially the parser will be in the configuration given by the pair  $(\$, w\$)$ . Therefore, we conclude that parsing table construction involves constructing a DFA that recognizes the viable prefixes of the right sentential forms, using the given grammar, and then maps its moves into the form of the Action|GOTO table. To construct such a DFA, we make use of the items that are part of a grammar's productions. Here, an item called the " $LR(0)$ " of a production is a production with a dot placed at some position on the right side of the production. For example if  $A \rightarrow XYZ$  is a production, then the following items can be generated from it:

$$\begin{aligned} A &\rightarrow .XYZ \\ A &\rightarrow X.YZ \\ A &\rightarrow XY.Z \\ A &\rightarrow XYZ. \end{aligned}$$

If the length of the right side of the production is  $n$ , then there are  $(n+1)$  different positions on the right side of a production where a dot can be placed. Hence, the number of items that can be generated are  $(n+1)$ .

The dot's position on the right side tells us how much of the right-hand side of the production is seen in the process of parsing. For example, the item  $A \rightarrow X.YZ$  tells us that we have already seen a string derivable from  $X$  in the input and expect to see the string derivable from  $YZ$  next in the input.

#### 5.4.1 Augmented Grammar

To construct a DFA that recognizes the viable prefixes, we make use of augmented grammar, which is defined as follows: if  $G = (V, T, P, S)$  is a given grammar, then the augmented grammar will be  $G_1 = (V \cup \{S_1\}, T, P \cup \{S_1 \rightarrow S\}, S_1)$ ; that is, we add a unit production  $S_1 \rightarrow S$  to the grammar  $G$  and make  $S_1$  the new start symbol. The resulting grammar will be an augmented grammar. The purpose of augmenting the grammar is to make it explicitly clear to parser when to accept the string. Parsing will stop when the parser is on the verge of carrying out the reduction using  $S_1 \rightarrow S$ . A NFA that recognizes the viable prefixes will be a finite automata whose states correspond to the production items of the augmented grammar. Every item represents one state in the automata, with the initial state corresponding to an item  $S_1 \rightarrow S$ . The transitions in the automata are defined as follows:

$$\delta(A \rightarrow \alpha.X\beta, X) = A \rightarrow \alpha X.\beta$$

$\delta(A \rightarrow \alpha.B\beta, \in) = B \rightarrow \gamma$  (This transition is required, because if the current state is  $A \rightarrow \alpha.B\beta$ , that means we have not yet seen a string derivable from the nonterminal  $B$ ; and since  $B \rightarrow \gamma$  is a production of the grammar, unless we see  $\gamma$ , we will not get  $B$ . Therefore, we have to travel the path that recognizes  $\gamma$ , which requires entering into the state identified by  $B \rightarrow \gamma$  without consuming any input symbols.)

This NFA can then be transformed into a DFA using the subset construction method. For example, consider the following grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow \text{id} \end{aligned}$$

The augmented grammar is:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + T \mid T \end{aligned}$$

$$T \rightarrow T^* F \mid F$$

$$F \rightarrow \text{id}$$

The items that can be generated using these productions are:

$$S \rightarrow E$$

$$S \rightarrow E.$$

$$E \rightarrow .E + T$$

$$E \rightarrow E.+T$$

$$E \rightarrow E + .T$$

$$E \rightarrow E + T.$$

$$E \rightarrow .T$$

$$E \rightarrow T.$$

$$T \rightarrow .T^* F$$

$$T \rightarrow T.^* F$$

$$T \rightarrow T^*.F$$

$$T \rightarrow T^* F.$$

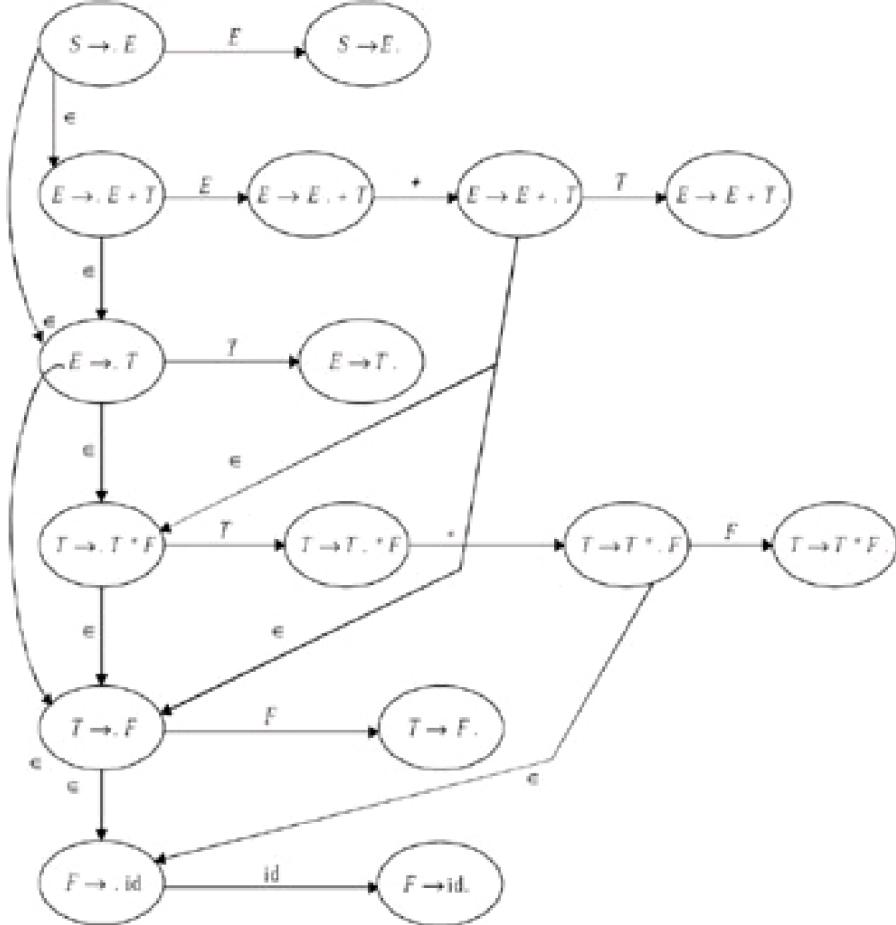
$$T \rightarrow .F$$

$$T \rightarrow F.$$

$$F \rightarrow .\text{id}$$

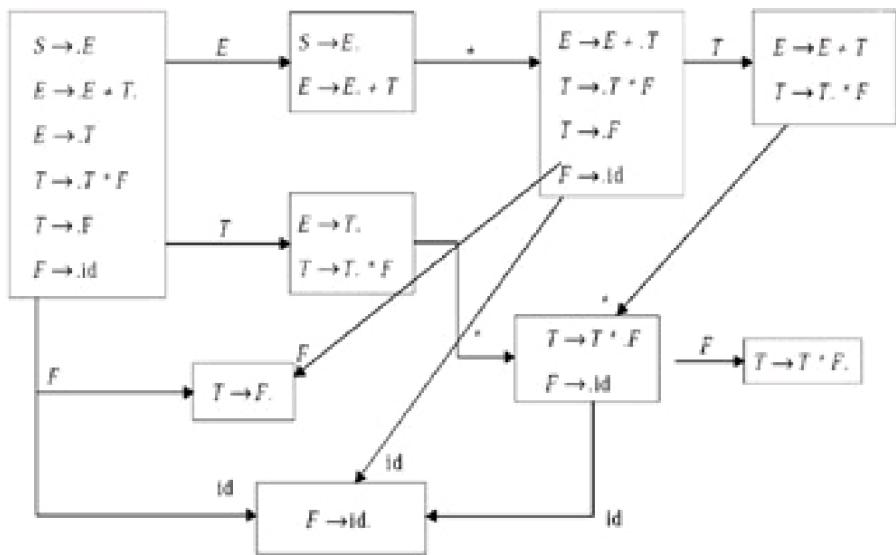
$$F \rightarrow \text{id}.$$

Therefore, the transition diagram of the NFA that recognizes viable prefixes is as shown in [Figure 5.1](#).



**Figure 5.1:** NFA transition diagram recognizes viable prefixes.

The DFA equivalent of the NFA shown in [Figure 5.1](#) is, by using subset construction, illustrated in [Figure 5.2](#).



**Figure 5.2:** Using subset construction, a DFA equivalent is derived from the transition diagram in [Figure 5.1](#).

Therefore, every state of the DFA that recognizes viable prefixes is a set of items; and hence, the set of DFA states will be a collection of sets of items—but any arbitrary collection of set of items will not correspond to the DFA set of states. A set of items that corresponds to the states of a DFA that recognizes viable prefixes is called a "canonical collection". Therefore, construction of a DFA involves finding canonical collection. An algorithm exists that directly obtains the canonical collection of LR(0) sets of items, thereby allowing us to obtain the DFA. Using this algorithm, we can directly obtain a DFA that recognizes the viable prefixes, rather than going through NFA to DFA transformation, as explained above. The algorithm for finding out the canonical collection of LR(0) sets of items makes use of the closure and goto functions. The set closure( $/$ ), where  $/$  is a set of items, is computed as follows:

1. Add every item in  $I$  to closure( $I$ )
2. Repeat
 

For every item of the form  $A \rightarrow \alpha.B\beta$  in closure( $I$ ) do

For every production  $B\beta \rightarrow$  do

Add  $B\beta \rightarrow$  to closure( $I$ )

Until no new item can be added to closure( $I$ )

For example, consider the following grammar:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T^* F \mid F$$

$$F \rightarrow \text{id}$$

$$\text{The closure}(\{E \rightarrow .E+T\}) = \{ E \rightarrow .E+T$$

$$E \rightarrow .T$$

$$T \rightarrow .T^* F$$

$$T \rightarrow .F$$

$$F \rightarrow .\text{id}$$

}

$$\text{goto}(I, X) = \text{closure}(\{A \rightarrow \alpha X \beta \mid A \rightarrow \alpha X \beta \text{ is in } I\})$$

That is, to find out goto from  $I$  on  $X$ , first identify all the items in  $I$  in which the dot precedes  $X$  on the right side. Then, move the dot in all the selected items one position to the right(i.e., over  $X$ ), and then take a closure of the set of these items.

**For example, if set  $I = \{ E \rightarrow .E+T$**

$$E \rightarrow .T$$

$$T \rightarrow .T^* F$$

$$T \rightarrow .F$$

$$F \rightarrow .\text{id}$$

}

**then  $\text{goto}(I, T) = \text{closure}(\{E \rightarrow T.$**

$$T \rightarrow T^* F$$

}

$$\}) = \{ E \rightarrow T.$$

$$T \rightarrow T^* F$$

}

#### 5.4.2 An Algorithm for Finding the Canonical Collection of Sets of LR(0) Items

/\* Let  $C$  be the canonical collection of sets of LR(0) items. We maintain  $C_{\text{new}}$  and  $C_{\text{old}}$  to continue the iterations\*/

Input: augmented grammar

Output: canonical collection of sets of LR(0) items (i.e., set  $C$ )

1.  $C_{\text{old}} = \emptyset$

2. add closure ( $\{S_1 \rightarrow .S\}$ ) to  $C$

3. while  $C_{\text{old}} \neq C_{\text{new}}$  do

```

{ temp =  $C_{\text{new}} - C_{\text{old}}$ 
 $C_{\text{old}} = C_{\text{new}}$ 
for every  $I$  in temp do
  for every  $X$  in  $V \cup T$  (i.e., for every grammar symbol  $X$ )
  do
    if goto( $I, X$ ) is not empty and not in  $C_{\text{new}}$  then
      add goto( $I, X$ ) to  $C_{\text{new}}$ 
}

```

4.  $C = C_{\text{new}}$

For example consider the following grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow \text{id} \end{aligned}$$

The augmented grammar is:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow \text{id} \end{aligned}$$

Initially,  $C_{\text{old}} = \emptyset$ . First we obtain:

$$\begin{aligned} \text{closure}(\{S \rightarrow .E\}) &= \{ S \rightarrow .E \\ &\quad E \rightarrow .E + T \\ &\quad E \rightarrow .T \\ &\quad T \rightarrow .T^* F \\ &\quad T \rightarrow .F \\ &\quad F \rightarrow .\text{id} \\ &\} \end{aligned}$$

We call it  $I_0$  and add it to  $C_{\text{new}}$ . Therefore:

$$\begin{aligned} C_{\text{new}} &= \{ I_0 \} \\ \text{Temp} &= \{ I_0 \} \\ C_{\text{old}} &= \{ I_0 \} \end{aligned}$$

In the first iteration, we obtain the goto from  $I_0$  on every grammar symbol, as shown below:

$$\begin{aligned} \text{goto}(I_0, E) &= \text{closure}(\{S \rightarrow E.\}) \\ &= \{S \rightarrow E. \\ &\quad E \rightarrow E. + T \\ &\quad E \rightarrow E. \\ &\quad F \rightarrow F. \} \end{aligned}$$

$I = I_1$

Add it to  $C_{\text{new}}$ :

$$\begin{aligned}\text{goto}(I_0, T) &= \text{closure}(\{E \rightarrow T. \\ &\quad T \rightarrow T.*F\} \\ &= \{E \rightarrow T. \\ &\quad T \rightarrow T.*F\} \\ &= I_2\end{aligned}$$

Add it to  $C_{\text{new}}$ :

$$\begin{aligned}\text{goto}(I_0, F) &= \text{closure}(\{T \rightarrow F. \\ &\quad \} \\ &= \{T \rightarrow F. \\ &\quad \} = I_3\end{aligned}$$

Add it to  $C_{\text{new}}$ :

$$\begin{aligned}\text{goto}(I_0, \text{id}) &= \text{closure}(\{F \rightarrow \text{id.}\} \\ &= \{F \rightarrow \text{id.}\} \\ &= I_4\end{aligned}$$

Add it to  $C_{\text{new}}$ :

$$\begin{aligned}\text{goto}(I_0, +) &= \emptyset, \text{ therefore, not added to } C_{\text{new}} \\ \text{goto}(I_0, *) &= \emptyset, \text{ therefore not added to } C_{\text{new}}\end{aligned}$$

Therefore, at the end of first iteration:

$$\begin{aligned}C_{\text{old}} &= \{I_0\} \\ C_{\text{new}} &= \{I_0, I_1, I_2, I_3, I_4\}\end{aligned}$$

In the second iteration:

$$\begin{aligned}\text{Temp} &= \{I_1, I_2, I_3, I_4\} \\ C_{\text{old}} &= \{I_0, I_1, I_2, I_3, I_4\}\end{aligned}$$

So, in the second iteration, we obtain goto from {1, 2, 3, 4} on every grammar symbol, as shown below:

$$\begin{aligned}\text{goto}(I_1, E) &= \emptyset, \text{ therefore not added to } C_{\text{new}} \\ \text{goto}(I_1, T) &= \emptyset, \text{ therefore not added to } C_{\text{new}} \\ \text{goto}(I_1, F) &= \emptyset, \text{ therefore not added to } C_{\text{new}} \\ \text{goto}(I_1, \text{id}) &= \emptyset, \text{ therefore not added to } C_{\text{new}}\end{aligned}$$

$$\begin{aligned}\text{goto}(I_1, *) &= \emptyset, \text{ therefore not added to } C_{\text{new}} \\ \text{goto}(I_1, +) &= \text{closure}(\{E \rightarrow E +.T\}) \\ &= \{E \rightarrow E +.T \\ &\quad T \rightarrow .T.*F \\ &\quad T \rightarrow .F\}\end{aligned}$$

$$F \rightarrow .id$$

$$\} = I_5$$

Add it to  $C_{\text{new}}$ :

$\text{goto}(I_2, E) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_2, T) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_2, F) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_2, id) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_2, +) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_2, *) = \text{closure}(\{T \rightarrow T^* . F$   
 $\})$   
 $\}) = \{ T \rightarrow T^* . F$   

$$F \rightarrow .id$$
  

$$\} = I_6$$

Add it to  $C_{\text{new}}$ :

$\text{goto}(I_3, E) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_3, T) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_3, F) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_3, id) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_3, +) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_3, *) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_4, E) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_4, T) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_4, F) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_4, id) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_4, +) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_4, *) = \phi$ , therefore not added to  $C_{\text{new}}$

Therefore, at the end of the second iteration:

$$C_{\text{old}} = \{ I_0, I_1, I_2, I_3, I_4 \}$$

$$C_{\text{new}} = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6 \}$$

In the third iteration:

$$\text{Temp} = \{ I_5, I_6 \}$$

$$C_{\text{old}} = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6 \}$$

In the third iteration, we obtain goto from  $\{I_5, I_6\}$  on every grammar symbol, as shown below:

$\text{goto}(I_5, E) = \phi$ , therefore not added to  $C_{\text{new}}$   
 $\text{goto}(I_5, T) = \text{closure}(\{E \rightarrow E + T.$   
 $T \rightarrow T^* F$

```

}
)
= { E → E + T.
    T → T.*F
} = I7

```

Add it to  $C_{\text{new}}$ :

```

goto(I5, F) = closure({ T → F.
}
) = {T → F.
} = same as I3,
hence, not added to  $C_{\text{new}}$ 
goto(I5, id) = closure({F → id.
}
) = {F → id.
} = same as I4,
hence, not added to  $C_{\text{new}}$ 
goto(I5, *) = φ, therefore not added to  $C_{\text{new}}$ 
goto(I5, +) = φ, therefore not added to  $C_{\text{new}}$ 
goto(I6, E) = φ, therefore not added to  $C_{\text{new}}$ 
goto(I6, T) = φ, therefore not added to  $C_{\text{new}}$ 
goto(I6, F) = closure({ T → T.*F.
}
) = { T → T.*F.
} = I8

```

Add it to  $C_{\text{new}}$ :

```

goto(I6, id) = closure({F → id.
}
) = {F → id.
} = same as I4,
hence not added to  $C_{\text{new}}$ 
goto(I5, *) = φ, therefore not added to  $C_{\text{new}}$ 
goto(I5, +) = φ, therefore not added to  $C_{\text{new}}$ 

```

Therefore, at the end of the third iteration:

$$\begin{aligned}
 C_{\text{old}} &= \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6 \} \\
 C_{\text{new}} &= \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8 \}
 \end{aligned}$$

In the fourth iteration:

$$\pi_{\text{new}} = \{ \text{r} \rightarrow \text{r} + \}$$

**temp = { I<sub>7</sub>, I<sub>8</sub> }**

$$C_{\text{old}} = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8 \}$$

So, in the fourth iteration, we obtain a goto from {I<sub>7</sub>, I<sub>8</sub>} on every grammar symbol, as shown below:

goto(I<sub>7</sub>, E) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>7</sub>, T) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>7</sub>, F) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>7</sub>, id) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>7</sub>, +) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>7</sub>, \*) = closure({T → T\*.F  
                  F → .id  
                  }  
                  )  
                  = {T → T\*.F.  
                  F → .id  
                  }; same as I<sub>6</sub>,  
hence not added to C<sub>new</sub>  
goto(I<sub>8</sub>, E) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>8</sub>, T) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>8</sub>, F) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>8</sub>, id) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>8</sub>, +) =  $\phi$ , therefore not added to C<sub>new</sub>  
goto(I<sub>8</sub>, \*) =  $\phi$ , therefore not added to C<sub>new</sub>

At the end of fourth iteration:

$$C_{\text{old}} = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8 \}$$

$$C_{\text{new}} = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8 \}$$

$$\text{Therefore, } C = \{ I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8 \}$$

The transition diagram of the DFA is shown in [Figure 5.3](#).

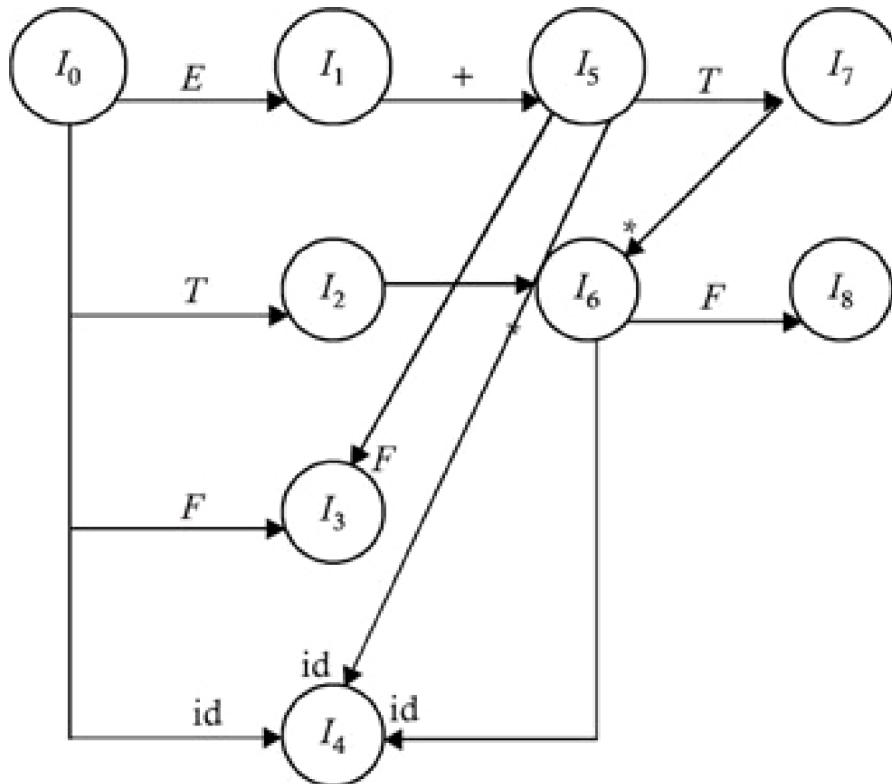


Figure 5.3: DFA transition diagram showing four iterations for a canonical collection of sets.

#### 5.4.3 Construction of a Parsing Action|GOTO Table for an SLR(1) Parser

The methods for constructing the parsing Action|GOTO table are described below.

##### Construction of the Action Table

1. For every state  $I_i$  in  $C$  do
  - for every terminal symbol  $a$  do
    - if  $\text{goto}(I_i, a) = I_j$ , then
      - make  $\text{action}[I_i, a] = S_j$  /\*for shift and enter into the state  $j$ \*/
2. For every state  $I_i$  in  $C$  whose underlying set of LR(0) items contains an item of the form  $A \rightarrow \alpha . \text{do}$ 
  - for every  $b$  in  $\text{FOLLOW}(A)$  do
    - make  $\text{action}[I_i, b] = R_k$  /\*where  $k$  is the number of the production  $A \rightarrow \alpha$  standing for reduce by  $A \rightarrow \alpha$  \*/
3. Make  $[I_i, \$] = \text{accept}$  if  $I_i$  contains an item  $S_1 \rightarrow S$ .

It is obvious that if a state  $I_i$  has a transition on a terminal  $a$  going to  $I_j$ , then the parser's next move will be to shift and enter into state  $j$ . Therefore, the shift entries in the action table are the mappings of the transitions in the DFA on terminals. Similarly, if state  $I_i$  corresponds to the viable prefix that contains the right side of the production  $A \rightarrow \alpha$ , then the parser will call a reduction by  $A \rightarrow \alpha$  on all those symbols that are in the  $\text{FOLLOW}(A)$ . This is because if the next input symbol happens to be a terminal symbol that can  $\text{FOLLOW}(A)$ , then only the reduction by  $A \rightarrow \alpha$  may lead to a previous right-most derivation. That is, if the next input symbol belongs to  $\text{FOLLOW}(A)$ , then the position of  $\alpha$  can be considered to be the one where, if it is replaced by  $A$ , we might get a previous right-most derivation. Whether or not  $A \rightarrow \alpha$  is a handle is decided in this manner.

The initial state is the one whose underlying set of items' representations contain an item  $S_1 \rightarrow .S$ . This method is called "SLR(1)"— $\alpha$  Simple LR; and the (1) indicates a length of one lookahead (the next symbol used by the parser to decide its next move) used. Therefore, this parsing table is an SLR parsing table. (When the parentheses are not specified, the length of the lookahead is assumed to be one.)

##### Construction of the Goto Table

A goto table is simply a mapping of transitions in the DFA on nonterminals. Therefore, it is constructed as follows:

For every  $l_i$  in  $C$  do

For every nonterminal  $A$  do

if  $\text{goto}(l_i, A) = l_j$  then

Make  $\text{GOTO}[l_i, A] = j$

Therefore, the SLR parsing table for the grammar having the following productions is shown in [Table 5.4](#).

$$S \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T^* F \mid F$$

$$F \rightarrow \text{id}$$

**Table 5.4: Action|GOTO SLR Parsing Table**

Action Table					GOTO Table		
	id	+	*	\$	$E$	$T$	$F$
$l_0$	$S_4$				1	2	3
$l_1$		$S_5$		Accept			
$l_2$		$R_2$	$S_6$	$R_2$			
$l_3$		$R_4$	$R_4$	$R_4$			
$l_4$		$R_5$	$R_5$	$R_5$			
$l_5$	$S_4$					7	3
$l_6$	$S_4$						8
$l_7$		$R_1$	$S_6$	$R_1$			
$l_8$		$R_3$	$R_3$	$R_3$			

The productions are numbered as:

$$E \rightarrow E + T \quad (1)$$

$$E \rightarrow T \quad (2)$$

$$T \rightarrow T^* F \quad (3)$$

$$T \rightarrow F \quad (4)$$

$$F \rightarrow \text{id} \quad (5)$$

#### EXAMPLE 5.2

Consider the following grammar:

$$S \rightarrow CC$$

-    -

$C \rightarrow cC$   
 $C \rightarrow d$

$S_1 \rightarrow S$   
 $S \rightarrow CC$   
 $C \rightarrow cC$   
 $C \rightarrow d$

---

$I_0 = \text{closure}(\{S_1 \rightarrow .S\}) = \{ S_1 \rightarrow .S$   
 $\quad \quad \quad S \rightarrow .CC$   
 $\quad \quad \quad C \rightarrow .cC$   
 $\quad \quad \quad C \rightarrow .d$   
 $\quad \quad \quad \}$   
 $\text{goto}(I_0, S) = \text{closure}(\{S_1 \rightarrow S.\}) = \{ S_1 \rightarrow S. \} = I_1$   
 $\text{goto}(I_0, C) = \text{closure}(\{S \rightarrow C.C\}) = \{ S \rightarrow C.C$   
 $\quad \quad \quad C \rightarrow .cC$   
 $\quad \quad \quad C \rightarrow .d \} = I_2$   
 $\text{goto}(I_0, c) = \text{closure}(\{C \rightarrow c.C\}) = \{ C \rightarrow c.C$   
 $\quad \quad \quad C \rightarrow .cC$   
 $\quad \quad \quad C \rightarrow .d \} = I_3$   
 $\text{goto}(I_0, d) = \text{closure}(\{C \rightarrow d.\}) = \{ C \rightarrow d. \} = I_4$   
 $\text{goto}(I_2, C) = \text{closure}(\{S \rightarrow CC.\}) = \{ S \rightarrow CC.$   
 $\quad \quad \quad \} = I_5$   
 $\text{goto}(I_2, c) = \text{closure}(\{C \rightarrow c.C\}) = \{ C \rightarrow c.C$   
 $\quad \quad \quad C \rightarrow .cC$   
 $\quad \quad \quad C \rightarrow .d$   
 $\quad \quad \quad \} = \text{same as } I_3$   
 $\text{goto}(I_2, d) = \text{closure}(\{C \rightarrow d.\}) = \{ C \rightarrow d.$   
 $\quad \quad \quad \} = \text{same as } I_4$   
 $\text{goto}(I_3, C) = \text{closure}(\{C \rightarrow c.C\}) = \{ C \rightarrow c.C$   
 $\quad \quad \quad \} = I_6$   
 $\text{goto}(I_3, c) = \text{closure}(\{C \rightarrow c.C\}) = \{ C \rightarrow c.C$   
 $\quad \quad \quad C \rightarrow .cC$   
 $\quad \quad \quad C \rightarrow .d$   
 $\quad \quad \quad \} = \text{same as } I_3$   
 $\text{goto}(I_3, d) = \text{closure}(\{C \rightarrow d.\}) = \{ C \rightarrow d.$   
 $\quad \quad \quad \} = \text{same as } I_4$

Figure 5.4

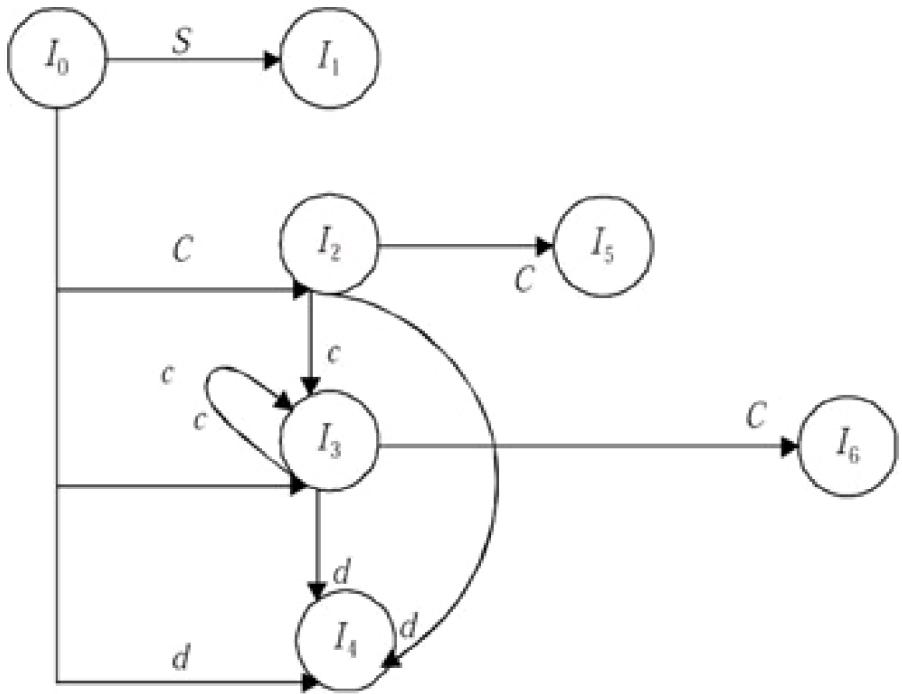


Figure 5.4: Transition diagram for [Example 5.2](#) DFA.

Therefore, the grammar has the following productions:

$$S_1 \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

which are numbered as:

$$S \rightarrow CC \quad (1)$$

$$C \rightarrow cC \quad (2)$$

$$C \rightarrow d \quad (3)$$

has an SLR parsing table as shown in [Table 5.5](#).

**Table 5.5: SLR Parsing Table**

Action Table				GOTO Table	
	c	d	\$	S	C
$I_0$	$S_3$	$S_4$		1	2
$I_1$			accept		
$I_2$	$S_3$	$S_4$			5
$I_3$	$S_3$	$S_4$			6
$I_4$	$R_3$	$R_3$	$R_3$		
$I_5$			$R_1$		
$I_6$	$R_2$	$R_2$	$R_2$		

By using the method discussed above, a parsing table can be obtained for any grammar. But the action table obtained from the method above will not necessarily be without multiple entries for every grammar. Therefore, we define a SLR(1) grammar as one in which we can obtain the action table without multiple entries by using the method described. If the action table contains multiple entries, then the grammar for which the table is obtained is not SLR(1) grammar.

For example, consider the following grammar:

$$\begin{aligned} S &\rightarrow AaAb \\ S &\rightarrow BbBa \\ A &\rightarrow \epsilon \\ B &\rightarrow \epsilon \end{aligned}$$

The augmented grammar will be:

$$\begin{aligned} S_1 &\rightarrow S \\ S &\rightarrow AaAb \\ S &\rightarrow BbBa \\ A &\rightarrow \epsilon \\ B &\rightarrow \epsilon \end{aligned}$$

The canonical collection sets of LR(0) items are computed as follows.

$$\begin{aligned} I_0 = \text{closure}(\{S_1 \rightarrow .S\}) &= \{S_1 \rightarrow .S \\ &\quad S \rightarrow .AaAb \\ &\quad S \rightarrow .BbBa \\ &\quad A \rightarrow . \\ &\quad B \rightarrow . \\ &\quad \} \end{aligned}$$

$$\begin{aligned} \text{goto}(I_0, S) &= \text{closure}(\{S_1 \rightarrow S.\}) = \{S_1 \rightarrow S.\} = I_1 \\ \text{goto}(I_0, A) &= \text{closure}(\{S \rightarrow A.aAb\}) = \{S \rightarrow A.aAb\} = I_2 \\ \text{goto}(I_0, B) &= \text{closure}(\{S \rightarrow B.bBa\}) = \{S \rightarrow B.bBa\} = I_3 \\ \dots &= \dots \end{aligned}$$

$\text{goto}(I_2, a) = \text{closure}(\{S \rightarrow au.av\}) = \{ S \rightarrow au.av \}$

$A \rightarrow \cdot$

$\} = I_4$

$\text{goto}(I_3, b) = \text{closure}(\{S \rightarrow Bb.Ba\}) = \{ S \rightarrow Bb.Ba \}$

$B \rightarrow \cdot$

$\} = I_5$

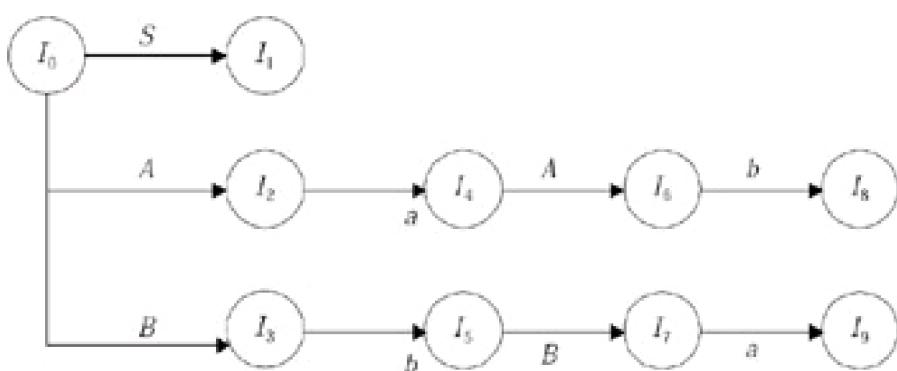
$\text{goto}(I_4, A) = \text{closure}(\{S \rightarrow AaA.b\}) = \{ S \rightarrow AaA.b \} = I_6$

$\text{goto}(I_5, B) = \text{closure}(\{S \rightarrow BbB.a\}) = \{ S \rightarrow BbB.a \} = I_7$

$\text{goto}(I_6, b) = \text{closure}(\{S \rightarrow AaAb.\}) = \{ S \rightarrow AaAb. \} = I_8$

$\text{goto}(I_7, a) = \text{closure}(\{S \rightarrow BbBa.\}) = \{ S \rightarrow BbBa. \} = I_9$

The transition diagram for the DFA is shown in [Figure 5.5](#).



**Figure 5.5:** DFA Transition diagram.

[Table 5.6](#) shows the SLR parsing table for the grammar having the following productions:

$S_1 \rightarrow S$

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

**Table 5.6: Action | GOTO SLR Parsing Table**

Action Table				GOTO Table		
	a	b	\$	S	A	B
$l_0$	$R_3/R_4$	$R_3/R_4$		1	2	3
$l_1$			Accept			
$l_2$	$S_4$					
$l_3$		$S_5$				
$l_4$	$R_3$	$R_3$			6	
$l_5$	$R_4$	$R_4$				7
$l_6$		$S_8$				
$l_7$	$S_9$					
$l_8$			$R_1$			
$l_9$			$R_2$			

The productions are numbered as follows:

$$S \rightarrow AaAb \quad (1)$$

$$S \rightarrow BbBa \quad (2)$$

$$A \rightarrow \epsilon \quad (3)$$

$$B \rightarrow \epsilon \quad (4)$$

Since the action table shown in [Table 5.6](#) contains multiple entries, the above grammar is not SLR(1).

SLR(1) grammars constitute a small subset of context-free grammars, so an SLR parser can only succeed on a small number of context-free grammars. That means an SLR parser is a less-powerful LR parser. (The power of the parser is measured in terms of the number of grammars on which it can succeed.) This is because when an SLR parser sees a right-hand-side production rule  $A \rightarrow \alpha$  on the top of the stack, it replaces this rule by the left-hand-side nonterminal  $A$  if the next input symbol can FOLLOW the nonterminal  $A$ . But sometimes this reduction may not lead to the generation of previous right-most derivations. For example, the parser constructed above can do the reduction by the production  $A \rightarrow \epsilon$  in the state  $l_0$  if the next input symbol happens to be either  $a$  or  $b$ , because both  $a$  and  $b$  are in the FOLLOW( $A$ ). But since the reduction by  $A \rightarrow \epsilon$  in  $l_0$  leads to the generation of a first instance of  $A$  in the sentential form  $AaAb$ , this reduction proves to be a proper one if the next input symbol is  $a$ . This is because the first instance of  $A$  in the sentential form  $AaAb$  is followed by  $a$ . But if the next input symbol is  $b$ , then this is not a proper reduction, because even though  $b$  follows  $A$ ,  $b$  follows a second instance of  $A$  in the sentential form  $AaAb$ . Similarly, if the parser carries out the reduction by  $A \rightarrow \epsilon$  in the state  $l_4$ , then it should be done if the next input symbol is  $b$ , because reduction by  $A \rightarrow \epsilon$  in  $l_4$  leads to the generation of a second instance of  $A$  in the sentential form  $AaAb$ .

Therefore, we conclude that if:

1. We let terminal  $a$  follow the first instance of  $A$  and let terminal  $b$  follow the second instance of  $A$  in the sentential form  $AaAb$ ;
2. We associate  $a$  with the item  $A \rightarrow .$  in  $l_0$  and terminal  $b$  with item  $A \rightarrow .$  in  $l_4$ ;
3. The parser has been asked to carry out a reduction by  $A \rightarrow \epsilon$  in  $l_0$  on those terminals associated

with the item  $A \rightarrow .$  in  $I_0$ , and carry out the reduction  $A \rightarrow \in$  in  $I_4$  on those terminals associated with the item  $A \rightarrow .$  in  $I_4$ ;

then there would have been no conflict and the parser would have been more powerful. But this requires associating a list of terminals (lookaheads) with the items. You may recall (see [Chapter 4](#)) that lookaheads are symbols that the parser uses to ‘look ahead’ in the input buffer to decide whether or not reduction is to be done. That is, we have to work with items of the form  $A \rightarrow \alpha.X\beta$ . The item  $a$  is called as an LR(1) item, because the length of the lookahead is one; therefore, an item without a lookahead is one with lookahead length of zero 0, an LR(0) item. In the SLR method, we were working with LR(0) items. Therefore, we define an LR( $k$ ) item to be an item using lookaheads of length  $k$ . So, an LR(1) item is comprised of two parts: the LR(0) item and the lookahead associated with the item.

**Note** We conclude that if we work with LR(1) items instead of using LR(0) items, then every state of the parser will correspond to a set of LR(1) items. When the parser looks ahead in the input buffer to decide whether reduction is to be done or not, the information about the terminals will be available in the state of the parser itself, which is not the case with the SLR parser state. Hence, with LR(1), we get a more powerful parser.

Therefore, if we modify the closure and the goto functions to work suitably with the LR(1) items, by allowing them to compute the lookaheads, we can obtain the canonical collection of sets of LR(1). And from this we can obtain the parsing Action|GOTO table. For example, closure( $I$ ), where  $I$  is a set of LR(1) items, is computed as follows:

1. Add every item in  $I$  to closure( $I$ ).

2. Repeat

For every item of the form  $A \rightarrow \alpha.B\beta$ ,  $a$  in closure( $I$ ) do

For every production  $B \rightarrow \gamma$  do

Add  $B \rightarrow \cdot\gamma$ , FIRST( $\beta a$ ) to closure( $I$ )

/\* because the reduction by  $B \rightarrow \gamma$  generates  $B$  preceding  $\beta$  in the right side of  $A \rightarrow \alpha B \beta$ ; and hence, the reduction by  $B \rightarrow \gamma$  is proper only on those symbols that are in the FIRST( $\beta$ ). But if  $\beta$  derives to an empty string, then  $a$  will also follow  $B$ , and the lookaheads of  $B \rightarrow \gamma$  will be FIRST( $\beta a$ )

until no new item can be added to closure( $I$ )

For example, consider the following grammar:

$S \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T^* F \mid F$

$F \rightarrow id$

The closure( $\{S \rightarrow .E, \$\}$ ) = {  $S \rightarrow .E, \$$

$E \rightarrow .E + T, \$ \mid +$

$E \rightarrow .T, \$ \mid +$

$T \rightarrow .T^* F, \$ \mid + \mid ^*$

$T \rightarrow .F, \$ \mid + \mid ^*$

$F \rightarrow .id, \$ \mid + \mid ^*$

}

$goto(I, X) = \text{closure}(\{A \rightarrow \alpha X \beta, a \mid A \rightarrow \alpha X \beta, a \text{ is in } I\})$

That is, to find out goto from  $I$  on  $X$ , first identify all the items in  $I$  with a dot preceding  $X$  in the LR(0) section of the item. Then, move the dot in all the selected items one position to the right (i.e., over  $X$ ), and then take this new set's closure. For example:

if set  $I = \{ S \rightarrow .E, \$$

```

 $L \rightarrow .L + I, \$ | +$ 
 $E \rightarrow .T, \$ | +$ 
 $T \rightarrow .T^*F, \$ | + | ^*$ 
 $T \rightarrow .F, \$ | + | ^*$ 
 $F \rightarrow .id, \$ | + | ^*$ 
}

then goto( $I, E$ ) = closure ( {  $S \rightarrow E.$ ,  $\$$ 
 $E \rightarrow E. + T, \$ | +$ 
}
) = {  $S \rightarrow E.$ ,  $\$$ 
 $E \rightarrow E. + T, \$ | +$ 
}

```

#### 5.4.4 An Algorithm for Finding the Canonical Collection of Sets of LR(1) Items

/\* Let  $C$  be the canonical collection of sets of LR(1) items. We maintain  $C_{new}$  and  $C_{old}$  to continue the iterations \*/

Input : augmented grammar

Output: canonical collection of sets of LR(1) items (i.e., set  $C$ )

1.  $C_{old} = \emptyset$
2. add closure({ $S_1 \rightarrow .S, \$$ }) to  $C$
3. while  $C_{old} \neq C_{new}$  do

```

temp =  $C_{new} - C_{old}$ 
 $C_{old} = C_{new}$ 
for every  $I$  in temp do
  for every  $X$  in  $V \cup T$  (i.e., for every grammar symbol  $X$ ) do
    if  $goto(I, X)$  is not empty and not in  $C_{new}$ , then
      add  $goto(I, X)$  to  $C_{new}$ 
}

```

4.  $C = C_{new}$

For example, consider the following grammar:

```

 $S \rightarrow AaAb$ 
 $S \rightarrow BbBa$ 
 $A \rightarrow \epsilon$ 
 $B \rightarrow \epsilon$ 

```

The augmented grammar will be:

```

 $S_1 \rightarrow S$ 
 $S \rightarrow AaAb$ 
 $S \rightarrow BbBa$ 
 $A \rightarrow \epsilon$ 
 $B \rightarrow \epsilon$ 

```

The canonical collection of sets of LR(1) items are computed as follows:

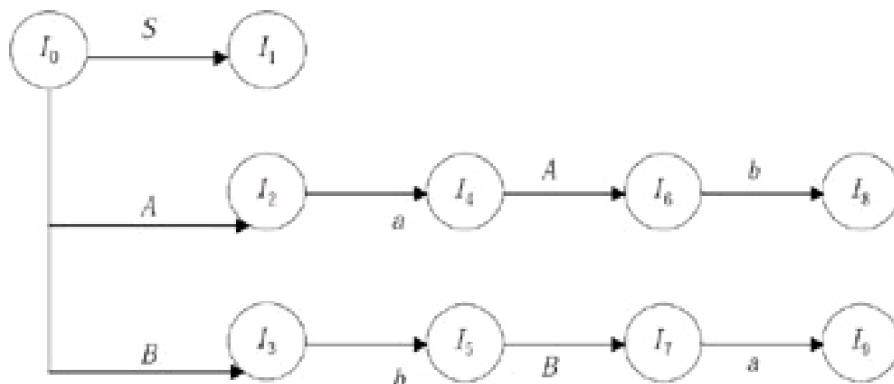
$$\begin{aligned}
I_0 = \text{closure}(\{S_1 \rightarrow .S, \$\}) &= \{ S_1 \rightarrow .S, \$ \\
&\quad S \rightarrow .AaAb, \$ \\
&\quad S \rightarrow .BbBa, \$ \\
&\quad A \rightarrow ., a \\
&\quad B \rightarrow ., b \\
&\quad \}
\end{aligned}$$

$$\begin{aligned}
\text{goto}(\{I_0, S\}) = \text{closure}(\{S_1 \rightarrow S., \$\}) &= \{ S_1 \rightarrow S., \$ \} = I_1 \\
\text{goto}(\{I_0, A\}) = \text{closure}(\{S \rightarrow A.aAb, \$\}) &= \{ S \rightarrow A.aAb, \$ \} = I_2 \\
\text{goto}(\{I_0, B\}) = \text{closure}(\{S \rightarrow B.bBa, \$\}) &= \{ S \rightarrow B.bBa, \$ \} = I_3 \\
\text{goto}(\{I_2, a\}) = \text{closure}(\{S \rightarrow Aa.Ab, \$\}) &= \{ S \rightarrow Aa.Ab, \$ \\
&\quad A \rightarrow ., b \\
&\quad \} = I_4
\end{aligned}$$

$$\begin{aligned}
\text{goto}(\{I_3, b\}) = \text{closure}(\{S \rightarrow Bb.Ba, \$\}) &= \{ S \rightarrow Bb.Ba, \$ \\
&\quad B \rightarrow ., a \\
&\quad \} = I_5
\end{aligned}$$

$$\begin{aligned}
\text{goto}(\{I_4, A\}) = \text{closure}(\{S \rightarrow AaA.b, \$\}) &= \{ S \rightarrow AaA.b, \$ \} = I_6 \\
\text{goto}(\{I_5, B\}) = \text{closure}(\{S \rightarrow BbB.a, \$\}) &= \{ S \rightarrow BbB.a, \$ \} = I_7 \\
\text{goto}(\{I_6, b\}) = \text{closure}(\{S \rightarrow AaAb., \$\}) &= \{ S \rightarrow AaAb., \$ \} = I_8 \\
\text{goto}(\{I_7, a\}) = \text{closure}(\{S \rightarrow BbBa., \$\}) &= \{ S \rightarrow BbBa., \$ \} = I_9
\end{aligned}$$

The transition diagram of the DFA is shown in [Figure 5.6](#).



[Figure 5.6](#): Transition diagram for the canonical collection of sets of LR(1) items.

#### 5.4.5 Construction of the Action|GOTO Table for the LR(1) Parser

The following steps will construct the parsing action table for the LR(1) parser:

1. for every state  $i$  in  $C$  do
  - for every terminal symbol  $a$  do
    - if  $\text{goto}(i, a) = j$  then
      - make  $\text{action}[i, a] = S_j /*$ for shift and enter into the state  $j*/$
2. for every state  $i$  in  $C$  whose underlying set of LR(1) items contains an item of the form  $A \rightarrow \alpha, a$ 
  - do
    - make  $\text{action}[i, a] = R_k /*$ where  $k$  is the number of

the production  $A \rightarrow \alpha$ , standing for reduce by  $A \rightarrow \alpha^*$  /

3. make  $[l_i, \$] = \text{accept}$  if  $l_i$  contains an item  $S_1 \rightarrow S, \$$

The goto table is simply a mapping of transitions in the DFA on nonterminals. Therefore, it is constructed as follows:

for every  $l_i$  in  $C$  do

for every nonterminal  $A$  do

if goto  $(l_i, A) = l_j$  then

make goto  $[l_i, A] = j$

This method is called as CLR(1) or LR(1), and is more powerful than SLR(1). Therefore, the CLR or LR parsing table for the grammar having the following productions is:

$$\begin{aligned}S_1 &\rightarrow S \\S &\rightarrow AaAb \\S &\rightarrow BbBa \\A &\rightarrow \epsilon \\B &\rightarrow \epsilon\end{aligned}$$

**Table 5.7: CLR/LR Parsing Action | GOTO Table**

Action Table				GOTO Table		
	a	b	\$	S	A	B
$l_0$	$R_3$	$R_4$		1	2	3
$l_1$			Accept			
$l_2$	$S_4$					
$l_3$		$S_5$				
$l_4$	$R_3$	$R_3$			6	
$l_5$	$R_4$	$R_4$				7
$l_6$		$S_8$				
$l_7$	$S_9$					
$l_8$			$R_1$			
$l_9$			$R_2$			

The productions are numbered as follows:

- $S \rightarrow AaAb$  (1)
- $S \rightarrow BbBa$  (2)
- $A \rightarrow \epsilon$  (3)
- $B \rightarrow \epsilon$  (4)

By comparing the SLR(1) parser with the CLR(1) parser, we find that the CLR(1) parser is more powerful. But the CLR(1) has a greater number of states than the SLR(1) parser; hence, its storage requirement is also greater than the

SLR(1) parser. Therefore, we can devise a parser that is an intermediate between the two; that is, the parser's power will be in between that of SLR(1) and CLR(1), and its storage requirement will be the same as SLR(1)'s. Such a parser, LALR(1), will be much more useful: since each of its states corresponds to the set of LR(1) items, the information about the lookaheads is available in the state itself, making it more powerful than the SLR parser. But a state of the LALR(1) parser is obtained by combining those states of the CLR parser that have identical LR(0) (core) items, but which differ in the lookaheads in their item set representations. Therefore, even if there is no reduce-reduce conflict in the states of the CLR parser that has been combined to form an LALR parser, a conflict may get generated in the state of LALR parser. We may be able to obtain a CLR parsing table without multiple entries for a grammar, but when we construct the LALR parsing table for the same grammar, it might have multiple entries.

#### 5.4.6 Construction of the LALR Parsing Table

The steps in constructing an LALR parsing table are as follows:

1. Obtain the canonical collection of sets of LR(1) items.
2. If more than one set of LR(1) items exists in the canonical collection obtained that have identical cores or LR(0)s, but which have different in lookaheads, then combine these sets of LR(1) items to obtain a reduced collection, C1, of sets of LR(1) items.
3. Construct the parsing table by using this reduced collection, as follows.

#### Construction of the Action Table

1. for every state  $i$  in  $C_1$  do
  - for every terminal symbol  $a$  do
    - if  $\text{goto}(i, a) = j$  then
      - make  $\text{action}[i, a] = S_j$  /\*for shift and enter
      - into the state  $j^*$
2. for every state  $i$  in  $C_1$  whose underlying set of LR(1) items contains an item of the form  $A \rightarrow \alpha.$ ,  $a$ , do
  - make  $\text{action}[i, a] = R_k$  /\*where  $k$  is the number of the production
  - $A \rightarrow \alpha$  standing for reduce by  $A \rightarrow \alpha$  \*/
3. make  $[i, \$] = \text{accept}$  if  $i$  contains an item  $S_1 \rightarrow S.., \$$

#### Construction of the Goto Table

The goto table simply maps transitions on nonterminals in the DFA. Therefore, the table is constructed as follows:

```
for every  $i$  in  $C_1$  do
for every nonterminal  $A$  do
  if  $\text{goto}(i, A) = j$  then
    make  $\text{goto}(i, A) = j$ 
```

For example, consider the following grammar:

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow aA \\ A &\rightarrow b \end{aligned}$$

The augmented grammar is:

$$\begin{aligned} S_1 &\rightarrow S \\ S &\rightarrow AA \\ A &\rightarrow aA \\ A &\rightarrow b \end{aligned}$$

The canonical collection of sets of LR(1) items are computed as follows:

$$I_0 = \text{closure}(\{S_1 \rightarrow .S, \$\}) = \{S_1 \rightarrow .S, \$$$

$$S \rightarrow .AA, \$$$

$$A \rightarrow .aA, a/b$$

$$A \rightarrow .b, a/b$$

}

$$\text{goto}(I_0, S) = \text{closure}(\{S_1 \rightarrow S., \$\}) = \{S_1 \rightarrow S., \$\} = I_1$$

$$\text{goto}(I_0, A) = \text{closure}(\{S \rightarrow AA, \$\}) = \{S \rightarrow AA, \$$$

$$A \rightarrow .aA, \$$$

$$A \rightarrow .b, \$$$

} = I\_2

$$\text{goto}(I_0, a) = \text{closure}(\{A \rightarrow a.A, a/b\}) = \{A \rightarrow a.A, a/b$$

$$A \rightarrow .aA, a/b$$

$$A \rightarrow .b, a/b$$

} = I\_3

$$\text{goto}(I_0, b) = \text{closure}(\{A \rightarrow b., a/b\}) = \{A \rightarrow b., a/b$$

} = I\_4

$$\text{goto}(I_2, A) = \text{closure}(\{S \rightarrow AA., \$\}) = \{S \rightarrow AA., \$$$

} = I\_5

$$\text{goto}(I_2, a) = \text{closure}(\{A \rightarrow a.A, \$\}) = \{A \rightarrow a.A, \$$$

$$A \rightarrow .aA, \$$$

$$A \rightarrow .b, \$$$

} = I\_6

$$\text{goto}(I_2, b) = \text{closure}(\{A \rightarrow b., \$\}) = \{A \rightarrow b., \$$$

} = I\_7

$$\text{goto}(I_3, A) = \text{closure}(\{A \rightarrow a.A., a/b\}) = \{A \rightarrow a.A., a/b$$

} = I\_8

$$\text{goto}(I_3, a) = \text{closure}(\{A \rightarrow a.A, a/b\}) = \{A \rightarrow a.A, a/b$$

$$A \rightarrow .aA, a/b$$

$$A \rightarrow .b, a/b$$

} = same as I<sub>3</sub>

$$\text{goto}(I_3, b) = \text{closure}(\{A \rightarrow b., a/b\}) = \{A \rightarrow b., a/b$$

} = same as I<sub>4</sub>

$$\text{goto}(I_6, A) = \text{closure}(\{A \rightarrow a.A., \$\}) = \{A \rightarrow a.A., \$$$

} = I<sub>9</sub>

$$\text{goto}(I_6, a) = \text{closure}(\{A \rightarrow a.A, \$\}) = \{A \rightarrow a.A, \$$$

$$A \rightarrow .aA, \$$$

$$A \rightarrow .b, \$$$

} = same as I<sub>6</sub>

$$\text{goto}(I_6, b) = \text{closure}(\{A \rightarrow b., \$\}) = \{ A \rightarrow b., \$ \\ \} = \text{same as } I_7$$

We see that the states  $I_3$  and  $I_6$  have identical LR(0) set items that differ only in their lookaheads. The same goes for the pair of states  $I_4, I_7$  and the pair of states  $I_8, I_9$ . Hence, we can combine  $I_3$  with  $I_6$ ,  $I_4$  with  $I_7$ , and  $I_8$  with  $I_9$  to obtain the reduced collection shown below:

$$I_0 = \text{closure} (\{S_1 \rightarrow .S, \$\}) = \{ S_1 \rightarrow .S, \$ \\ S \rightarrow .AA, \$ \\ A \rightarrow .aA, a/b \\ A \rightarrow .b, a/b \\ \}$$

$$I_1 = \{ S_1 \rightarrow S., \$ \}$$

$$I_2 = \{ S \rightarrow A.A, \$ \\ A \rightarrow .aA, \$ \\ A \rightarrow .b, \$ \\ \}$$

$$I_{36} = \{ A \rightarrow a.A, a/b/\$ \\ A \rightarrow .aA, a/b/\$ \\ A \rightarrow .b, a/b/\$ \\ \}$$

$$I_{47} = \{ A \rightarrow b., a/b/\$ \\ \}$$

$$I_5 = \{ S \rightarrow AA., \$ \\ \}$$

$$I_{89} = \{ A \rightarrow aA., a/b/\$ \\ \}$$

where  $I_{36}$  stands for combination of  $I_3$  and  $I_6$ ,  $I_{47}$  stands for the combination of  $I_4$  and  $I_7$ , and  $I_{89}$  stands for the combination of  $I_8$  and  $I_9$ . The transition diagram of the DFA using the reduced collection is shown in [Figure 5.7](#).

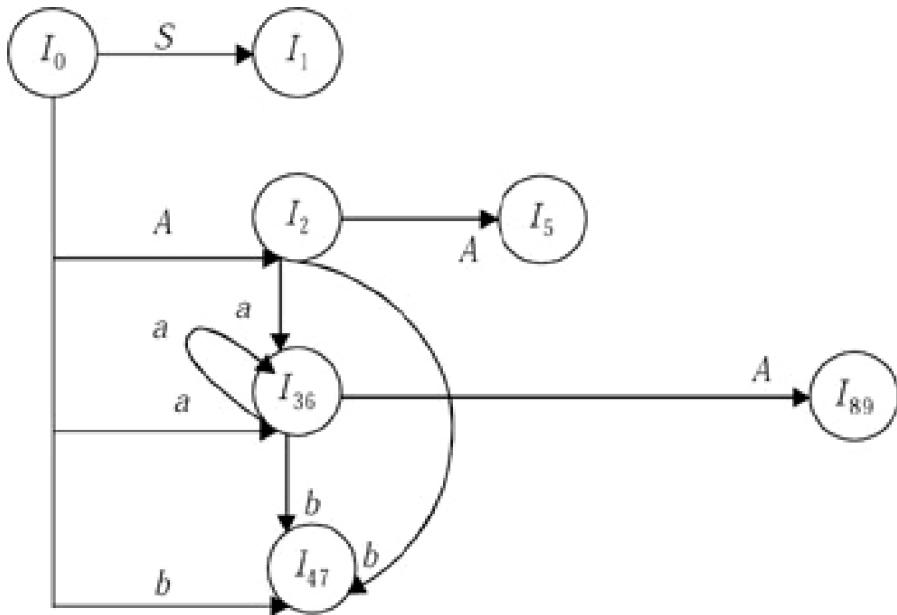


Figure 5.7: Transition diagram for a DFA using a reduced collection.

Therefore, [Table 5.8](#) shows the LALR parsing table for the grammar having the following productions:

$$\begin{aligned}S_1 &\rightarrow S \\S &\rightarrow AA \\A &\rightarrow aA \\A &\rightarrow b\end{aligned}$$

which are numbered as:

$$A \rightarrow AA \quad (1)$$

$$A \rightarrow aA \quad (2)$$

$$A \rightarrow b \quad (3)$$

**Table 5.8: LALR Parsing Table for a DFA Using a Reduced Collection**

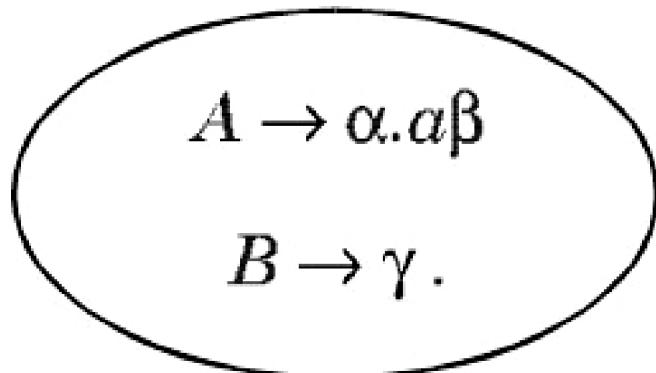
Action Table				GOTO Table	
	a	b	\$	S	A
I <sub>0</sub>	S <sub>36</sub>	S <sub>47</sub>		1	2
I <sub>1</sub>			Accept		
I <sub>2</sub>	S <sub>36</sub>	S <sub>47</sub>			5
I <sub>36</sub>	S <sub>36</sub>	S <sub>47</sub>			89
I <sub>47</sub>	R <sub>3</sub>	R <sub>3</sub>	R <sub>3</sub>		
I <sub>5</sub>			R <sub>1</sub>		
I <sub>89</sub>	R <sub>2</sub>	R <sub>2</sub>	R <sub>2</sub>		

#### 5.4.7 Parser Conflicts

An LR parser may encounter two types of conflicts: shift-reduce conflicts and reduce-reduce conflicts.

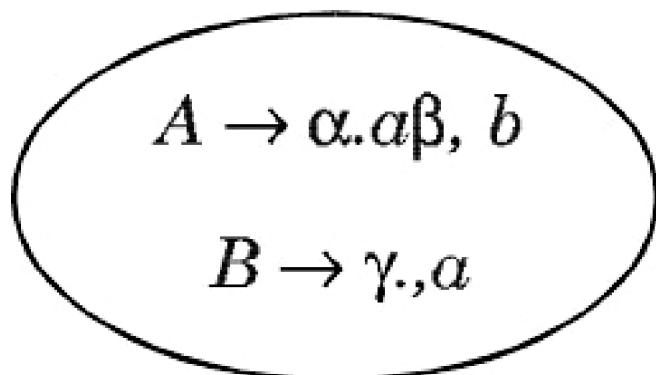
### Shift-Reduce Conflict

A shift-reduce ( $S\text{-}R$ ) conflict occurs in an SLR parser state if the underlying set of LR(0) item representations contains items of the form depicted in [Figure 5.8](#), and  $\text{FOLLOW}(B)$  contains terminal  $a$ .



[Figure 5.8](#): LR(0) underlying set representations that can cause SLR parser conflicts.

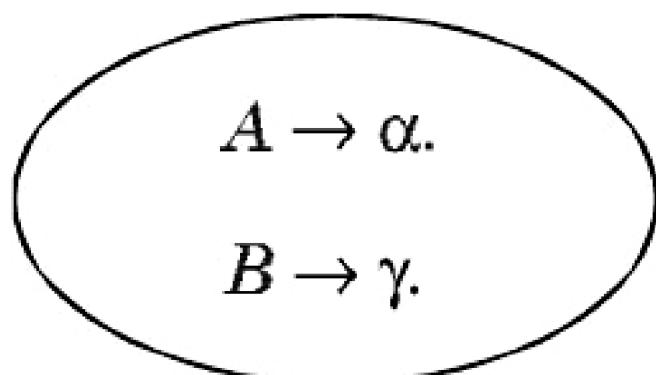
Similarly, an S-R conflict occurs in a state of the CLR or LALR parser if the underlying set of LR(1) items representation contains items of the form shown in [Figure 5.9](#).



[Figure 5.9](#): LR(1) underlying set representations that can cause CLR/LALR parser conflicts.

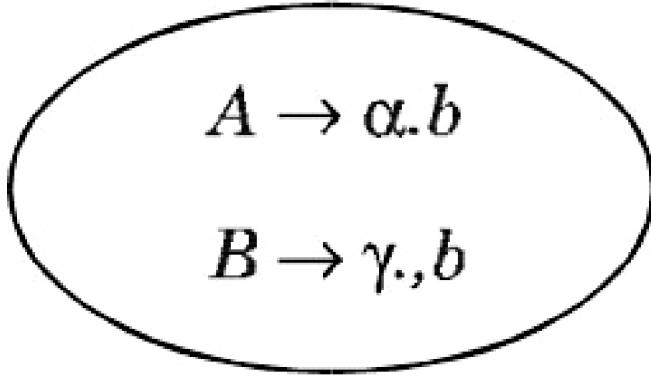
### Reduce-Reduce Conflict

A reduce-reduce ( $R\text{-}R$ ) conflict occurs if the underlying set of LR(0) items representation in a state of an SLR parser contains items of the form shown in [Figure 5.10](#), and  $\text{FOLLOW}(A)$  and  $\text{FOLLOW}(B)$  are not disjoint sets.



[Figure 5.10](#): LR(0) underlying set representations that can cause an SLR parser *reduce-reduce conflict*.

Similarly an  $R\text{-}R$  conflict occurs if the underlying set of LR(1) items representation in a state of a CLR or LALR parser contains items of the form shown in [Figure 5.11](#).



**Figure 5.11:** LR(1) underlying set representations that can cause an CLR/LALR parser.

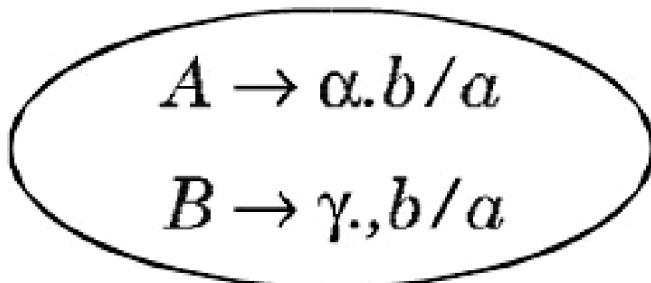
If a set of items' representation contains only nonfinal items, then there is no conflict in the corresponding state. (An item in which the dot is in the right-most position, like  $A \rightarrow XYZ.$ , is called as a final item; and an item in which the dot is not in the right-most position, like  $A \rightarrow X.YZ$ , is a nonfinal item).

Even if there is no *R-R* conflict in the states of a CLR parser, conflicts may be generated in the state of a LALR parser that is obtained by combining the states of the CLR parser. We combine the states of the CLR parser in order to form an LALR state. The items' lookaheads in the LALR parser state are obtained by combining the lookaheads of the corresponding items in the states of the CLR parser. And since reduction depends on the lookaheads, even if there is no *R-R* conflict in the states of the CLR parser, a conflict may become generated in the state of the LALR parser as a result of this state combination. For example, consider the sets of LR(1) items that represent the two different states of the CLR(1) parser, as shown in [Figure 5.12](#).



**Figure 5.12:** Sets of LR(1) items represent two different CLR(1) parser states.

There is no *R-R* conflict in these states. But when we combine these states to form an LALR, the state's set of items representation will be as shown in [Figure 5.13](#).



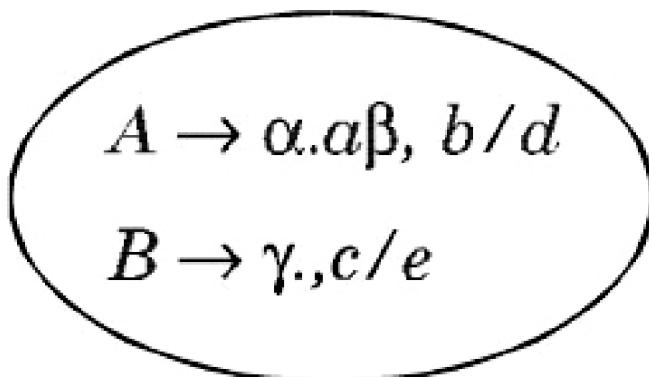
**Figure 5.13:** States are combined to form an LALR.

We see that there is an *R-R* conflict in this state, because the parser will call a reduction by  $A \rightarrow \alpha$  as well as by  $B \rightarrow \gamma$  on both  $a$  and  $b$ . If there is a *S-R* conflict in the CLR(1) states, it will never be reflected in the LALR(1) state obtained by combining the CLR(1) states. For example consider the sets of LR(1) items representing the two different states of the CLR(1) parser as shown in [Figure 5.14](#).



**Figure 5.14:** LR(1) items represent two different states of the CLR(1) parser.

There is no S-R conflict in these states. But when we combine these states, the resulting LALR state set will be as shown in [Figure 5.15](#). There is no S-R conflict in this state, as well.



**Figure 5.15:** LALR state set resulting from the combination of CLR(1) state sets.

#### 5.4.8 Handling Ambiguous Grammars

Since every ambiguous grammar fails to be LR, they will not belong to either the SLR, CLR, or LALR grammar classes. But some ambiguous grammars are quite useful for specifying languages. Hence, the question is how to deal with these grammars in the framework of LR parsing. For example, the natural grammar that specifies nonparenthesized expressions with + and \* operators is:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E^* E \\ E &\rightarrow \text{id} \end{aligned}$$

But this is ambiguous grammar, because the precedence and associations of the operators has not been specified. Even so, we prefer this grammar, because we can easily change the precedence and associations as required, thereby allowing us more flexibility. Similarly, if we use unambiguous grammar instead of the above grammar to specify the same language, it will have the following productions:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow \text{id} \end{aligned}$$

This parser will spend a substantial portion its time in carrying out reductions by the unit productions  $E \rightarrow T$  and  $T \rightarrow F$ . These production are in the grammar to enforce associations and precedence, thereby making the parsing time excessive. With an ambiguous grammar, every LR parser construction method will have conflicts. But these conflicts can be resolved by using the precedence and association information of + and \* as per the language's usage. For example, consider the SLR parser construction for the above grammar. The augmented grammar is:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + E \\ E &\rightarrow E^* E \\ E &\rightarrow \text{id} \end{aligned}$$

The canonical collection of sets of LR(0) items is shown below:

$$\begin{aligned} I_0 = \text{closure } (\{S \rightarrow .E\}) = \{ &S \rightarrow .E \\ &E \rightarrow .E + E \\ &E \rightarrow .E^* E \\ &E \rightarrow .\text{id} \\ & \vdots \end{aligned}$$

$\text{closure}(\{ S \rightarrow .E$   
 $E \rightarrow E. + E$   
 $E \rightarrow E. * E$   
 $\})$

$) = \{ S \rightarrow .E$   
 $E \rightarrow E. + E$   
 $E \rightarrow E. * E$   
 $\} = I_1$

$\text{closure}(\{ E \rightarrow \text{id}. \})$

$) = \{ E \rightarrow \text{id}. \}$   
 $\} = I_2$

$\text{closure}(\{ E \rightarrow E + .E \})$   
 $) = \{ E \rightarrow E + .E$   
 $E \rightarrow .E + E$   
 $E \rightarrow .E * E$   
 $E \rightarrow .\text{id}$   
 $\} = I_3$

$\text{closure}(\{ E \rightarrow E *.E \})$   
 $) = \{ E \rightarrow E *.E$   
 $E \rightarrow .E + E$   
 $E \rightarrow .E * E$   
 $E \rightarrow .\text{id}$   
 $\} = I_4$

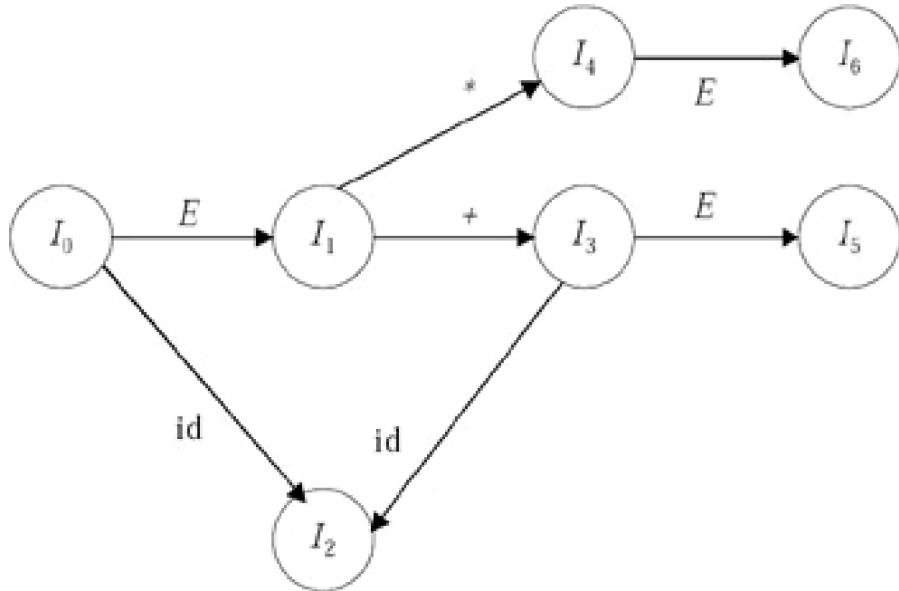
$\text{closure}(\{ E \rightarrow \text{id}. \})$   
 $) = \{ E \rightarrow \text{id}. \}$   
 $\} = \text{same as } I_2$

$\text{closure}(\{ E \rightarrow E + E. \})$   
 $E \rightarrow E. + E$   
 $E \rightarrow E. * E$   
 $\})$   
 $) = \{ E \rightarrow E + E.$   
 $E \rightarrow E. + E$   
 $E \rightarrow E. * E$   
 $\} = I_5$



$$\begin{aligned}
 ) = \{ & E \rightarrow E^* . E \\
 & E \rightarrow E + E \\
 & E \rightarrow E^* E \\
 & E \rightarrow .id \\
 \} = \text{same as } I_4
 \end{aligned}$$

The transition diagram of the DFA for the augmented grammar is shown in [Figure 5.16](#). The SLR parsing table is shown in [Table 5.9](#).



**Figure 5.16:** Transition diagram for augmented grammar DFA.

**Table 5.9: SLR Parsing Table for Augmented Grammar**

Action Table					GOTO Table
	+	*	id	\$	E
<i>I</i> <sub>0</sub>			S <sub>2</sub>		1
<i>I</i> <sub>1</sub>	S <sub>3</sub>	S <sub>4</sub>		accept	
<i>I</i> <sub>2</sub>	R <sub>3</sub>	R <sub>3</sub>		R <sub>3</sub>	
<i>I</i> <sub>3</sub>			S <sub>2</sub>		5
<i>I</i> <sub>4</sub>			S <sub>2</sub>		6
<i>I</i> <sub>5</sub>	S <sub>3/R</sub> 1	S <sub>4/R</sub> 1		R <sub>1</sub>	
<i>I</i> <sub>6</sub>	S <sub>3/R</sub> 2	S <sub>4/R</sub> 2		R <sub>2</sub>	

We see that there are shift-reduce conflicts in *I*<sub>5</sub> and *I*<sub>6</sub> on + as well as \*. Therefore, for an input string id + id + id\$, when the parser enters into the state *I*<sub>5</sub>, the parser will be in the configuration:

**Stack Contents**

*I*<sub>0</sub>E*I*<sub>1</sub> + *I*<sub>3</sub>E*I*<sub>5</sub>

**Unexpected Input**

+id\$

Hence, the parser can either reduce by  $E \rightarrow E + E$  or shift the  $+$  onto the stack and enter into the state  $I_3$ . To resolve this conflict, we make use of associations. If we want left-associativity, then a reduction by  $E \rightarrow E + E$  is the right choice. Whereas if we want right-associativity, then shift is a right choice.

Similarly, if the input string is  $id + id * id\$$  when the parser enters into the state  $I_5$ , it will be in the configuration:

<b>Stack Contents</b>	<b>Unexpended Input</b>
$I_0EI_1 + I_3EI_5$	$*id\$$

Hence, the parser can either reduce by  $E \rightarrow E + E$  or shift the  $*$  onto the stack and enter into the state  $I_4$  in order to resolve this conflict. We must make use of precedence if we want a higher precedence for  $+$  then the reduction by  $E \rightarrow E + E$ . If we want a higher precedence for  $*$ , then shift is a right choice.

Similarly if the input string is  $id * id + id\$$  when the parser enters into the state  $I_6$ , it will be in the configuration:

<b>Stack Contents</b>	<b>Unexpended Input</b>
$I_0EI_1 * I_4EI_6$	$+id\$$

Hence, the parser can either reduce by  $E \rightarrow E * E$  or shift the  $+$  onto the stack and enter into the state  $I_3$  in order to resolve this conflict. We have to make use of precedence if we want a higher precedence for  $*$ ; then reduction by  $E \rightarrow E * E$  is a right choice. Whereas if we want a higher precedence for  $+$ , then shift is right choice.

Similarly, if the input string is  $id * id * id\$$  when the parser enters into the state  $I_6$ , the parser will be in the configuration:

<b>Stack Contents</b>	<b>Unexpended Input</b>
$I_0EI_1 * I_4EI_6$	$*id\$$

The parser can either reduce by  $E \rightarrow E * E$  or shift the  $*$  onto the stack and enter into the state  $I_4$ . To resolve this conflict, we have to make use of associations. If we want left-associativity, then a reduction by  $E \rightarrow E * E$  is a right choice. If we want right-associativity, then shift is a right choice.

Therefore, for a higher precedence to  $*$ , and for left-associativity for both  $+$  and  $*$ , we get the SLR parsing table shown in [Table 5.10](#).

**Table 5.10: SLR Parsing Table Reflects Higher Precedence and Left-Associativity**

Action Table					GOTO Table
	+	*	id	\$	$E$
$I_0$			$S_2$		1
$I_1$	$S_3$	$S_4$		Accept	
$I_2$	$R_3$	$R_3$		$R_3$	
$I_3$			$S_2$		5
$I_4$			$S_2$		6
$I_5$	$R_1$	$S_4$		$R_1$	
$I_6$	$R_2$	$R_2$		$R_2$	

Therefore, we have a way to deal with ambiguous grammars. We can make use of nonambiguous rules to resolve parsing action conflicts.

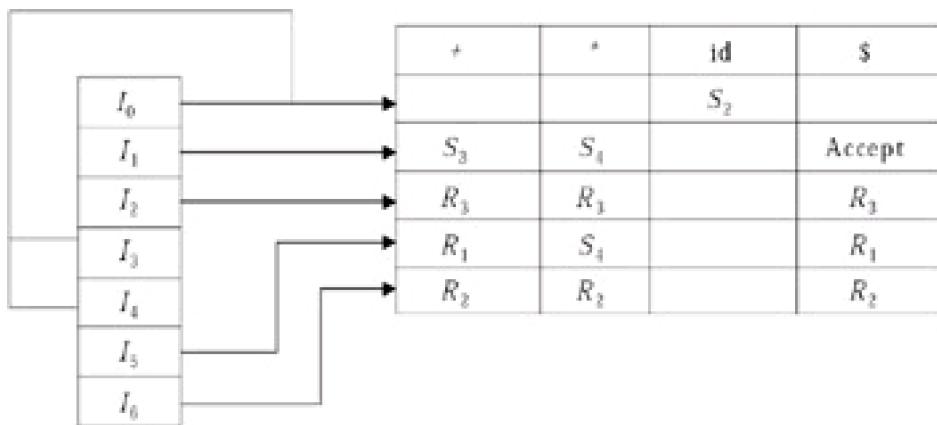


## 5.5 DATA STRUCTURES FOR REPRESENTING PARSING TABLES

Since there are only a few entries in the goto table, separate data structures must be used for the action table and the goto table. These data structures are described below.

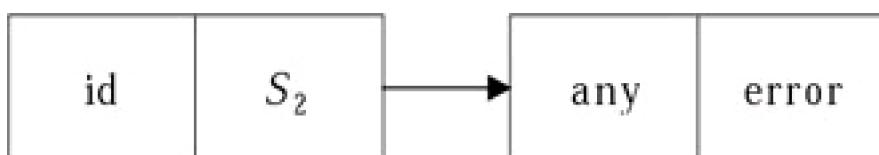
### Representing the Action Table

One of the simplest ways to represent the action table is to use a two-dimensional array. But since many rows of the action table are identical, we can save considerable space (and expend a negligible cost in processing time) by creating an array of pointers for each state. Then, pointers for states with the same actions will point to the same location, as shown in [Figure 5.17](#).



**Figure 5.17:** States with actions in common point to the same location via an array.

To access information, we assign each terminal a number from zero to one less than the number of terminals. We use this integer as an offset from the pointer value for each state. Further reduction in the space is possible at the expense of speed by creating a list of actions for each state. Each node on a list will be comprised of a terminal symbol and the action for that terminal symbol. It is here that the most frequent actions, like error actions, can be appended at the end of the list. For example, for the state  $I_0$  in [Table 5.10](#), the list will be as shown in [Figure 5.18](#).



**Figure 5.18:** List that incorporates the ability to append actions.

### Representing the GOTO Table

An efficient way to represent the goto table is to make a list of pairs for each nonterminal  $A$ . Each pair is of the form:

$$\text{goto}(\text{current-state}, A) = \text{next-state}$$

Since the error entries in the goto table are never consulted, we can replace each error entry by the most common nonerror entry in its column is represented by any in place of current-state.

## 5.6 WHY LR PARSING IS ATTRACTIVE

There are several reasons why LR parsers are attractive:

1. An LR parser can be constructed to recognize virtually all programming language constructs for which a CFG can be written.
2. The LR parsing method is the most general, nonbacktracking shift-reduce method known. Yet it can be implemented as efficiently as any other method.
3. The class of grammars that can be parsed by using the LR method is a proper superset of the class of grammars that can be parsed with a predictive parser.
4. The LR parser can quickly detect a syntactic error via the left-to-right scanning of input.

The main drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming language grammar. But fortunately, many LR parser generators are available that automatically generate the required LR parser.

## 5.7 EXAMPLES

The examples that follow further illustrate the concepts covered within this chapter.

### EXAMPLE 5.3

---

Construct an SLR(1) parsing table for the following grammar:

$$\begin{aligned} S &\rightarrow xAy \mid xBy \mid xAz \\ A &\rightarrow aS \mid q \\ B &\rightarrow q \end{aligned}$$

First, augment the given grammar by adding a production  $S_1 \rightarrow S$  to the grammar. Therefore, the augmented grammar is:

$$\begin{aligned} S_1 &\rightarrow S \\ S &\rightarrow xAy \mid xBy \mid xAz \\ A &\rightarrow aS \mid q \\ B &\rightarrow q \end{aligned}$$

Next, we obtain the canonical collection of sets of LR(0) items, as follows:

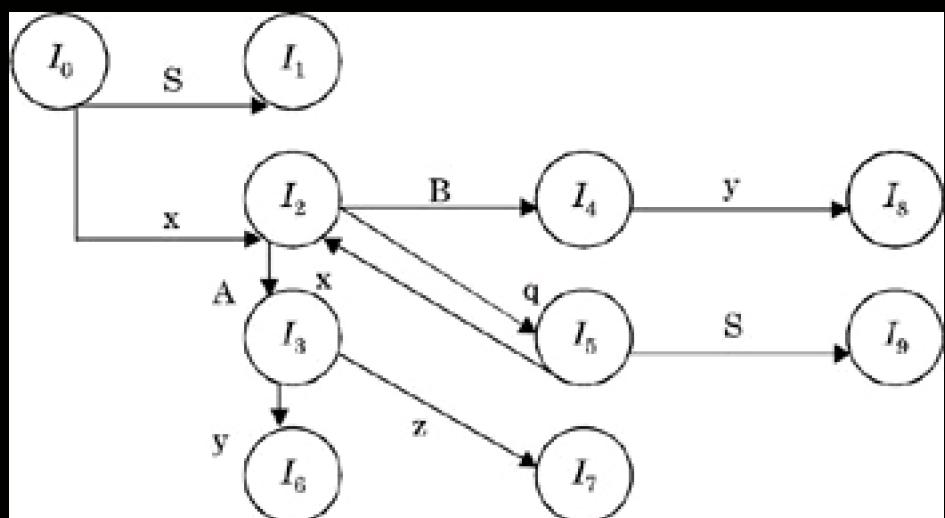
$$\begin{aligned} \text{closure}(\{S_1 \rightarrow .S\}) &= \{ S_1 \rightarrow .S \\ &\quad S \rightarrow .xAy \\ &\quad S \rightarrow .xBBy \\ &\quad S \rightarrow .xAz \\ &\} = I_0 \\ \text{goto}(I_0, S) &= \text{closure}(\{S_1 \rightarrow S.\}) = \{ S_1 \rightarrow S.\} = I_1 \\ \text{goto}(I_0, x) &= \text{closure}(\{S \rightarrow x.Ay \\ &\quad S \rightarrow x.BBy \\ &\quad S \rightarrow x.Az \\ &\}) = \{S \rightarrow x.Ay \\ &\quad S \rightarrow x.BBy \\ &\quad S \rightarrow x.Az \\ &\quad A \rightarrow .qS \\ &\quad A \rightarrow .q \\ &\quad B \rightarrow .q \\ &\} = I_2 \\ \text{goto}(I_2, A) &= \text{closure}(\{S \rightarrow xA.y \\ &\quad S \rightarrow xA.z\}) = \{S \rightarrow xA.y \\ &\quad S \rightarrow xA.z\} = I_3 \end{aligned}$$

$\text{goto}(I_2, B) = \text{closure}(\{S \rightarrow xB.y\}) = \{ S \rightarrow xB.y \} = I_4$   
 $\text{goto}(I_2, q) = \text{closure}(\{A \rightarrow q.S\})$   
 $A \rightarrow q.$

$B \rightarrow q.\}) = \{ A \rightarrow q.S$   
 $A \rightarrow q.$   
 $B \rightarrow q.$   
 $S \rightarrow .xAy$   
 $S \rightarrow .xBy$   
 $S \rightarrow .xAz$   
 $\} = I_5$

$\text{goto}(I_3, y) = \text{closure}(\{S \rightarrow xAy.\}) = \{ S \rightarrow xAy. \} = I_6$   
 $\text{goto}(I_3, z) = \text{closure}(\{S \rightarrow xAz.\}) = \{ S \rightarrow xAz. \} = I_7$   
 $\text{goto}(I_4, y) = \text{closure}(\{S \rightarrow xBy.\}) = \{ S \rightarrow xBy. \} = I_8$   
 $\text{goto}(I_5, S) = \text{closure}(\{A \rightarrow qS.\}) = \{ A \rightarrow qS. \} = I_9$   
 $\text{goto}(I_5, x) = \text{closure}(\{S \rightarrow x.Ay$   
 $S \rightarrow x.By$   
 $S \rightarrow x.Az$   
 $\}) = I_2$

Figure 5.19



Example 5.3

4. Using  $S \rightarrow xAz$ , we get  $\text{FOLLOW}(A) = \{z\}$

Therefore,  $\text{FOLLOW}(A) = \{y, z\}$ . Using  $A \rightarrow qS$ , we get  $\text{FOLLOW}(S) = \text{FOLLOW}(A) = \{y, z\}$ . Therefore,  $\text{FOLLOW}(S) = \{y, z, \$\}$ . Let the productions of the grammar be numbered as follows:

$$S \rightarrow xAy \quad (1)$$

$$S \rightarrow xBy \quad (2)$$

$$S \rightarrow xAz \quad (3)$$

$$A \rightarrow qS \quad (4)$$

$$A \rightarrow q \quad (5)$$

$$B \rightarrow q \quad (6)$$

The SLR parsing table for the productions above is shown in [Table 5.11](#).

**Table 5.11: SLR(1) Parsing Table**

Action Table						GOTO Table		
	x	Y	Z	q	\$	S	A	B
$l_0$	$S_2$	$R_3/R_4$				1		
$l_1$			Accept					
$l_2$				$S_5$			3	4
$l_3$		$S_6$	$S_7$					
$l_4$		$S_8$						
$l_5$	$S_2$	$R_5/R_6$	$R_5$			9		
$l_6$		$R_1$	$R_1$		$R_1$			
$l_7$		$R_3$	$R_3$		$R_3$			
$l_8$		$R_2$	$R_2$		$R_2$			
$l_9$		$R_4$	$R_4$					

#### EXAMPLE 5.4

Construct an SLR(1) parsing table for the following grammar:

$$S \rightarrow 0S0 \mid 1S1 \mid 10$$

First, augment the given grammar by adding the production  $S_1 \rightarrow S$  to the grammar. The augmented grammar is:

$$S_1 \rightarrow S$$

$$S \rightarrow 0S0 \mid 1S1 \mid 10$$

Next, we obtain the canonical collection of sets of LR(0) items, as follows:

$$\begin{aligned} \text{closure } (\{S_1 \rightarrow .S\}) &= \{ S_1 \rightarrow .S \\ &\quad S \rightarrow .0S0 \} \end{aligned}$$

$S \rightarrow .1S1$   
 $S \rightarrow .10$   
 $\} = I_0$   
 $\text{goto}(I_0, S) = \text{closure}(\{S_1 \rightarrow S.\}) = \{S_1 \rightarrow S.\} = I_1$   
 $\text{goto}(I_0, 0) = \text{closure}(\{S \rightarrow 0.S0\}) = \{S \rightarrow 0.S0\}$   
 $S \rightarrow .0S0$   
 $S \rightarrow .1S1$   
 $S \rightarrow .10$   
 $\} = I_2$   
 $\text{goto}(I_0, 1) = \text{closure}(\{S \rightarrow 1.S1$   
 $S \rightarrow 1.0$   
 $\}) = \{S \rightarrow 1.S1$   
 $S \rightarrow 1.0$   
 $S \rightarrow .0S0$   
 $S \rightarrow .1S1$   
 $S \rightarrow .10\} = I_3$   
 $\text{goto}(I_2, S) = \text{closure}(\{S \rightarrow 0S.0\}) = \{S \rightarrow 0S.0\} = I_4$   
 $\text{goto}(I_2, 0) = \text{closure}(\{S \rightarrow 0.S0$   
 $\}) = I_2$   
 $\text{goto}(I_2, 1) = \text{closure}(\{S \rightarrow 1.S1$   
 $S \rightarrow 1.0$   
 $\}) = I_3$   
 $\text{goto}(I_3, S) = \text{closure}(\{S \rightarrow 1S.1$   
 $\}) = I_5$   
 $\text{goto}(I_3, 0) = \text{closure}(\{S \rightarrow 10.$   
 $S \rightarrow 0.S0$   
 $\}) = \{S \rightarrow 10.$   
 $S \rightarrow 0.S0$   
 $S \rightarrow .0S0$   
 $S \rightarrow .1S1$   
 $S \rightarrow .10$   
 $\} = I_6$   
 $\text{goto}(I_3, 1) = \text{closure}(\{S \rightarrow 1S.1$   
 $S \rightarrow 1.0$   
 $\}) = I_3$   
 $\text{goto}(I_4, 0) = \text{closure}(\{S \rightarrow 0S0.$   
 $\}) = I_7$   
 $\text{goto}(I_5, 1) = \text{closure}(\{S \rightarrow 1S1.$   
 $\}) = I_8$   
 $\text{goto}(I_6, S) = \text{closure}(\{S \rightarrow 0S.0$   
 $\}) = I_4$   
 $\text{goto}(I_6, 0) = \text{closure}(\{S \rightarrow 0.S0$

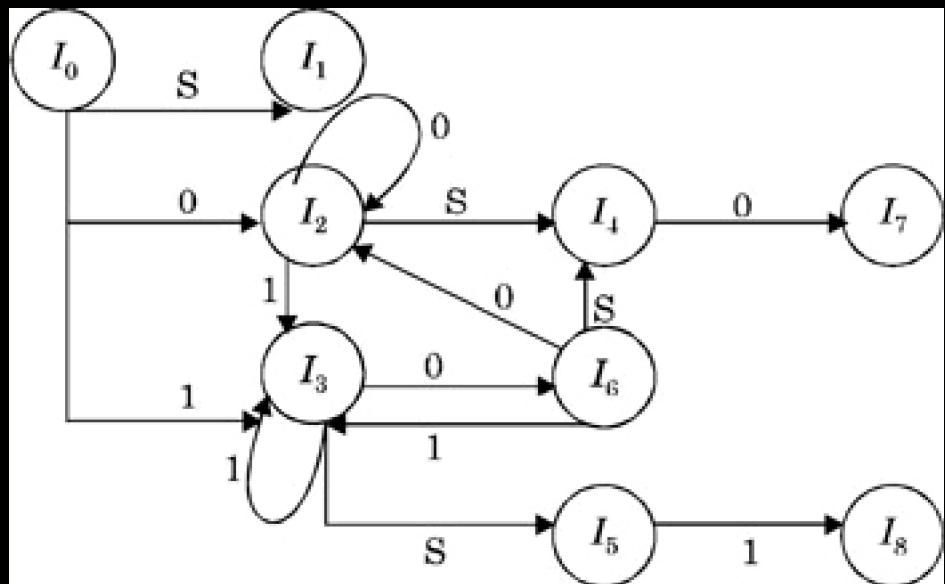
```

}) = I2
goto(I6, 1) = closure({S → 1.S1
                           S → 1.0
}) = I3

```

---

[Figure 5.20](#)



[Example 5.4](#)

$S \rightarrow 0S0$	(I)
$S \rightarrow 1S1$	(II)
$S \rightarrow 10$	(III)

[Table 5.12](#)

**Table 5.12: SLR Parsing Table for Example 5.4**

Action Table			GOTO Table	
	0	1	\$	S
$l_0$	$S_2$	$S_3$		1
$l_1$				accept
$l_2$	$S_2$	$S_3$		4
$l_3$	$S_6$	$S_3$		5
$l_4$	$S_7$			
$l_5$		$S_8$		
$l_6$	$S_2/R_3$	$S_3/R_3$	$R_3$	4
$l_7$	$R_1$	$R_1$		$R_1$
$l_8$	$R_2$	$R_2$		$R_2$

#### EXAMPLE 5.5

Consider the following grammar, and construct the LR(1) parsing table.

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

The augmented grammar is:

$$\begin{aligned} S\mid & \rightarrow S \\ S \rightarrow & aSbS \mid bSaS \mid \epsilon \end{aligned}$$

The canonical collection of sets of LR(1) items is:

$$\begin{aligned} I_0 = \{ & \\ & S\mid \rightarrow .S, \$ \\ & S \rightarrow .aSbS, \$ \\ & S \rightarrow .bSaS, \$ \\ & S \rightarrow ., \$ \\ & \} \\ \text{goto}(I_0, S) = \{ & S\mid \rightarrow S. , \$ \} = I_1 \\ \text{goto}(I_0, a) = \{ & S \rightarrow a.SbS, \$ \\ & S \rightarrow .aSbS, b \\ & S \rightarrow .bSaS, b \\ & S \rightarrow ., b \\ & \} = I_2 \\ \text{goto}(I_0, b) = \{ & S \rightarrow b.SaS, \$ \\ & S \rightarrow .aSbS, a \\ & S \rightarrow .bSaS, a \\ & S \rightarrow ., a \\ & \} = I_3 \\ \text{goto}(I_2, S) = \{ & S \rightarrow aS.bS, \$ \} = I_4 \end{aligned}$$

$\text{goto}(I_2, a) = \{ S \rightarrow a.SbS, b \\ S \rightarrow .aSbS, b \\ S \rightarrow .bSaS, b \\ S \rightarrow ., b \\ \} = I_5$   
 $\text{goto}(I_2, b) = \{ S \rightarrow b.SaS, b \\ S \rightarrow .aSbS, a \\ S \rightarrow .bSaS, a \\ S \rightarrow ., a \\ \} = I_6$   
 $\text{goto}(I_3, S) = \{ S \rightarrow bS.aS, \$ \} = I_7$   
 $\text{goto}(I_3, a) = \{ S \rightarrow a.SbS, a \\ S \rightarrow .aSbS, b \\ S \rightarrow .bSaS, b \\ S \rightarrow ., b \\ \} = I_8$

$\text{goto}(I_3, b) = \{ S \rightarrow b.SaS, a \\ S \rightarrow .aSbS, a \\ S \rightarrow .bSaS, a \\ S \rightarrow ., a \\ \} = I_9$   
 $\text{goto}(I_4, b) = \{ S \rightarrow aSbS, \$ \\ S \rightarrow .aSbS, \$ \\ S \rightarrow .bSaS, \$ \\ S \rightarrow ., \$ \\ \} = I_{10}$   
 $\text{goto}(I_5, S) = \{ S \rightarrow aS.bS, b \} = I_{11}$   
 $\text{goto}(I_5, a) = I_5$   
 $\text{goto}(I_5, b) = I_6$   
 $\text{goto}(I_6, S) = \{ S \rightarrow bS.aS, b \} = I_{12}$   
 $\text{goto}(I_6, a) = I_8$   
 $\text{goto}(I_6, b) = I_9$   
 $\text{goto}(I_7, a) = \{ S \rightarrow bSaS, \$ \\ S \rightarrow .aSbS, \$ \\ S \rightarrow .bSaS, \$ \\ S \rightarrow ., \$ \\ \} = I_{13}$   
 $\text{goto}(I_8, S) = \{ S \rightarrow aS.bS, a \} = I_{14}$   
 $\text{goto}(I_8, a) = I_5$   
 $\text{goto}(I_8, b) = I_6$   
 $\text{goto}(I_9, S) = \{ S \rightarrow bS.aS, a \} = I_{15}$   
 $\text{goto}(I_9, a) = I_8$   
 $\text{goto}(I_9, b) = I_9$   
 $\text{goto}(I_{10}, S) = \{ S \rightarrow aSbS, \$ \} = I_{16}$   
 $\text{goto}(I_{10}, a) = I_2$   
 $\text{goto}(I_{10}, b) = I_3$   
 $\text{goto}(I_{11}, b) = \{$

$$\begin{aligned} S &\rightarrow aSb.S, b \\ S &\rightarrow .aSbS, b \end{aligned}$$

$$\begin{aligned} S &\rightarrow .bSaS, b \\ S &\rightarrow ., b \\ } &= I_{17} \\ \text{goto}(I_{12}, a) &= \{ \\ S &\rightarrow bSa.S, b \\ S &\rightarrow .aSbS, b \\ S &\rightarrow .bSaS, b \\ S &\rightarrow ., b \\ } &= I_{18} \\ \text{goto}(I_{13}, S) &= \{ S \rightarrow bSaS, . \} = I_{19} \\ \text{goto}(I_{13}, a) &= I_2 \\ \text{goto}(I_{13}, b) &= I_3 \\ \text{goto}(I_{14}, b) &= \{ \\ S &\rightarrow aSb.S, a \\ S &\rightarrow .aSbS, a \\ S &\rightarrow .bSaS, a \\ S &\rightarrow ., a \\ } &= I_{20} \\ \text{goto}(I_{15}, a) &= \{ \\ S &\rightarrow bSa.S, a \\ S &\rightarrow .aSbS, a \\ S &\rightarrow .bSaS, a \\ S &\rightarrow ., a \\ } &= I_{21} \\ \text{goto}(I_{17}, S) &= \{ S \rightarrow aSbS, . \} = I_{22} \\ \text{goto}(I_{17}, a) &= I_5 \\ \text{goto}(I_{17}, b) &= I_6 \\ \text{goto}(I_{18}, S) &= \{ S \rightarrow bSaS, . \} = I_{23} \\ \text{goto}(I_{18}, a) &= I_5 \\ \text{goto}(I_{18}, b) &= I_6 \\ \text{goto}(I_{20}, S) &= \{ S \rightarrow aSbS, . \} = I_{24} \\ \text{goto}(I_{20}, a) &= I_3 \\ \text{goto}(I_{20}, b) &= I_9 \\ \text{goto}(I_{21}, S) &= \{ S \rightarrow bSaS, . \} = I_2 \\ \text{goto}(I_{21}, a) &= I_8 \\ \text{goto}(I_{21}, b) &= I_9 \end{aligned}$$

The parsing table for the production above is shown in [Table 5.13](#).

**Table 5.13: Parsing Table for Example 5.5**

Action Table			GOTO Table	
	A	B	\$	S
$l_0$	$S_2$	$S_3$	$R_3$	1
$l_1$			Accept	
$l_2$	$S_5$	$S_6/R_3$		4
$l_3$	$S_8/R_3$	$S_9$		7
$l_4$		$S_{10}$		
$l_5$	$S_5$	$S_6/R_3$		11
$l_6$	$S_8/R_3$	$S_9$		12
$l_7$	$S_{13}$			
$l_8$	$S_5$	$S_6/R_3$		14
$l_9$	$S_8/R_3$	$S_9$		15
$l_{10}$	$S_2$	$S_3$	$R_3$	16
$l_{11}$		$S_{17}$		
$l_{12}$	$S_{18}$			
$l_{13}$	$S_2$	$S_3$	$R_3$	19
$l_{14}$		$S_{20}$		
$l_{15}$		$S_{21}$		
$l_{16}$			$R_1$	
$l_{17}$	$S_5$	$S_6/R_3$		22
$l_{18}$	$S_5$	$S_6/R_3$		23
$l_{19}$			$R_2$	
$l_{20}$	$S_8/R_3$	$S_9$		24
$l_{21}$	$S_8/R_3$	$S_9$		25
$l_{22}$		$R_1$		
$l_{23}$		$R_2$		
$l_{24}$	$R_1$			
$l_{25}$	$R_2$			

The productions for the grammar are numbered as shown below:

$$S \rightarrow aSbS \quad (1)$$

$$S \rightarrow .bSaS \quad (2)$$

$$C \rightarrow \epsilon \quad (2\backslash)$$

**EXAMPLE 5.6**

Construct an LALR(1) parsing table for the following grammar:

$$\begin{aligned} S &\rightarrow Aa \mid bAc \mid dc \mid bda \\ A &\rightarrow d \end{aligned}$$

The augmented grammar is:

$$\begin{aligned} S \mid &\rightarrow S \\ S &\rightarrow Aa \\ S &\rightarrow bAc \\ S &\rightarrow dc \\ S &\rightarrow bda \\ A &\rightarrow d \end{aligned}$$

The canonical collection of sets of LR(1) items is:

$$\begin{aligned} I_0 = \{ &S \mid \rightarrow .S, \$ \\ &S \rightarrow .Aa, \$ \\ &S \rightarrow .bAc, \$ \\ &S \rightarrow .dc, \$ \\ &S \rightarrow .bda, \$ \\ &A \rightarrow .d, a \\ & \} \end{aligned}$$

$$\text{goto}(I_0, S) = \{ S \mid \rightarrow S., \$ \} = I_1$$

$$\text{goto}(I_0, A) = \{ S \rightarrow A.a, \$ \} = I_2$$

$$\text{goto}(I_0, b) = \{ S \rightarrow b.Ac, \$ \}$$

$$S \rightarrow b.da, \$$$

$$A \rightarrow .d, c$$

$$\} = I_3$$

$$\text{goto}(I_0, d) = \{ S \rightarrow d.c, \$ \}$$

$$A \rightarrow d., a$$

$$\} = I_4$$

$$\text{goto}(I_2, a) = \{ S \rightarrow Aa., \$ \} = I_5$$

$$\text{goto}(I_3, A) = \{ S \rightarrow bA.c, \$ \} = I_6$$

$$\text{goto}(I_3, d) = \{ S \rightarrow bd.a, \$ \}$$

$$A \rightarrow d., c$$

$$\} = I_7$$

$$\text{goto}(I_4, c) = \{ S \rightarrow dc., \$ \} = I_8$$

$$\text{goto}(I_6, c) = \{ S \rightarrow bAc., \$ \} = I_9$$

$$\dots$$

$\text{gutuv}(I_7, u) = (\omega \rightarrow \text{out}, \Phi) = I_{10}$

Table 5.14

Table 5.14: LALR(1) Parsing Table for [Example 5.5](#)

$$\begin{array}{l} S \rightarrow Aa \mid aAc \mid Bc \mid bBa \\ A \rightarrow d \\ B \rightarrow d \end{array}$$

$$\begin{array}{l} S \mid \rightarrow S \\ S \rightarrow Aa \mid aAc \mid Bc \mid bBa \\ A \rightarrow d \end{array}$$

$B \rightarrow d$

$I_0 = \{$
$S  \rightarrow .S, \$$
$S \rightarrow .Aa, \$$
$S \rightarrow .aAc, \$$
$S \rightarrow .Bc, \$$
$S \rightarrow .bBa, \$$
$A \rightarrow .d, a$
$B \rightarrow .d, c$
}
$\text{goto}(I_0, S) = \{ S  \rightarrow S., \$ \} = I_1$
$\text{goto}(I_0, A) = \{ S \rightarrow A.a, \$ \} = I_2$
$\text{goto}(I_0, B) = \{ S \rightarrow B.c, \$ \} = I_3$
$\text{goto}(I_0, a) = \{ S \rightarrow a.Ac, \$$
$A \rightarrow .d, c$
} = I_4
$\text{goto}(I_0, b) = \{$
$S \rightarrow b.Ba, \$$
$B \rightarrow .d, a$
} = I_5
$\text{goto}(I_0, d) = \{$
$A \rightarrow d., a$
$B \rightarrow d., c$
} = I_6
$\text{goto}(I_2, a) = \{ S \rightarrow Aa., \$ \} = I_7$
$\text{goto}(I_3, c) = \{ S \rightarrow Bc., \$ \} = I_8$
$\text{goto}(I_4, A) = \{ S \rightarrow aAc., \$ \} = I_9$
$\text{goto}(I_4, d) = \{ A \rightarrow d., c \} = I_{10}$
$\text{goto}(I_5, B) = \{ S \rightarrow bBa., \$ \} = I_{11}$
$\text{goto}(I_5, d) = \{ B \rightarrow d., a \} = I_{12}$
$\text{goto}(I_9, c) = \{ S \rightarrow aAc., \$ \} = I_{13}$
$\text{goto}(I_{11}, a) = \{ S \rightarrow bBa., \$ \} = I_{14}$

Table 5.15

**Table 5.15: LALR(1) Parsing Table for Example 5.6**

Action Table						GOTO Table		
	a	b	c	d	\$	S	A	B
$l_0$	$S_4$	$S_5$		$S_6$		1	2	3
$l_1$					Accept			
$l_2$	$S_7$							
$l_3$			$S_8$					
$l_4$				$S_{10}$		9		
$l_5$				$S_{12}$				11
$l_6$	$R_5$		$R_6$					
$l_7$					$R_1$			
$l_8$					$R_3$			
$l_9$			$S_{13}$					
$l_{10}$			$R_5$					
$l_{11}$	$S_{14}$							
$l_{12}$	$R_6$							
$l_{13}$					$R_2$			
$l_{14}$					$R_4$			

The productions of the grammar are numbered as shown below:

1.  $S \rightarrow Aa$
2.  $S \rightarrow aAc$
3.  $S \rightarrow Bc$
4.  $S \rightarrow bBa$
5.  $A \rightarrow d$
6.  $B \rightarrow d$

#### EXAMPLE 5.8

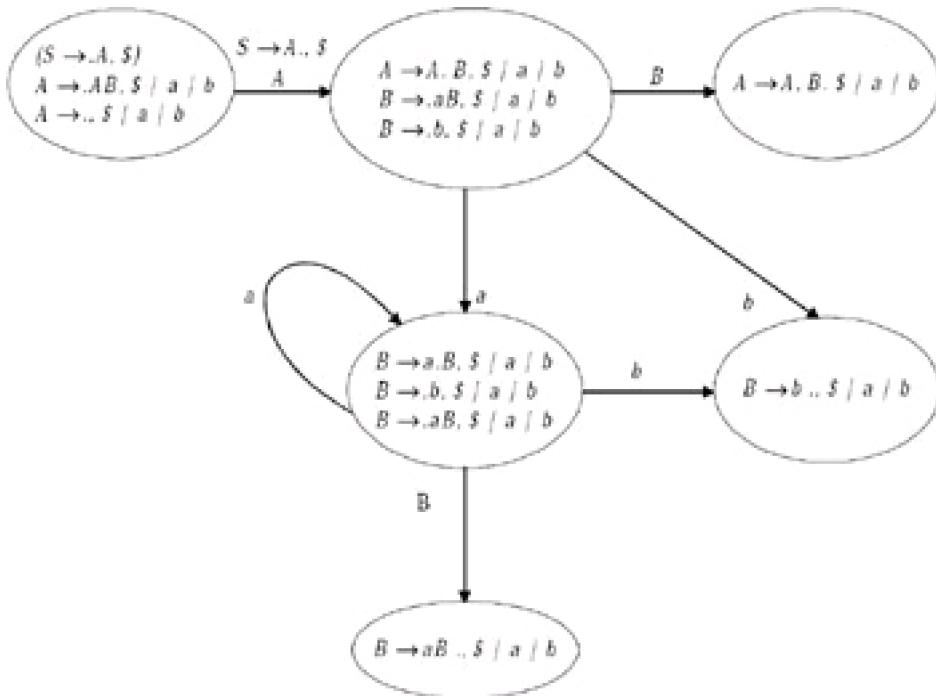
---

Construct the nonempty sets of LR(1) items for the following grammar:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow AB \mid \epsilon \\ B &\rightarrow aB \mid b \end{aligned}$$


---

The collection of nonempty sets of LR(1) items is shown in [Figure 5.21](#).



**Figure 5.21:** Collection of nonempty sets of LR(1) items for [Example 5.7](#).

Team LiB

◀ PREVIOUS

NEXT ▶

## Chapter 6: Syntax-Directed Definitions and Translations

### 6.1 SPECIFICATION OF TRANSLATIONS

The specification of a construct's translation in a programming language involves specifying what the construct is, as well as specifying the translating rules for the construct. Whenever a compiler encounters that construct in a program, it will translate the construct according to the rules of translation. Here, the term "translation" is used in a much broader sense. Translation does not necessarily mean generating either intermediate code or object code. Translation also involves adding information into the symbol table as well as performing construct-specific computations. For example, if a construct is a declarative statement, then its translation adds information about the construct's type attribute into the symbol table. Whereas, if the construct is an expression, then its translation generates the code for evaluating the expression.

When we specify what the construct is, we specify the syntactic structure of the construct; hence, syntactic specification is the part of the specification of the construct's translation. Therefore, if we suitably extend the notation that we use for syntactic specification so that it will allow for both the syntactic structure and the rules of translation that go along with it, then we can use this notation as a framework for the specification of the construct translation.

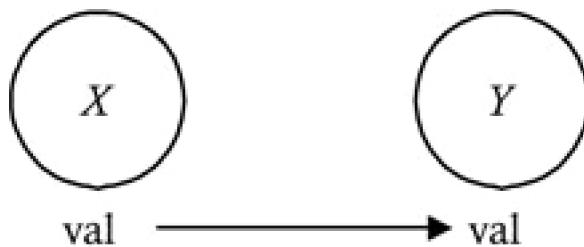
Translation of a construct involves manipulating the values of various quantities. For example, when translating the declarative statement `int a, b, c;` the compiler needs to extract the type `int` and add it to the symbol records of `a`, `b`, and `c`. This requires that the compiler keep track of the type `int`, as well as the pointers to the symbol records containing `a`, `b`, and `c`.

Since we use a context-free grammar to specify the syntactic structure of a programming language, we extend that context-free grammar by associating sets of attributes with the grammar symbols. These sets hold the values of the quantities, which a compiler is required to track, as well as the associated set of production rules of the grammar that specify the how the attributed values of the grammar symbols of the production are manipulated. These extensions allow us to specify the translations. Syntax-directed definitions and translation schemes are examples of these extensions of context-free grammars, allowing us to specify the translations.

Syntax-directed definitions use CFG to specify the syntactic structure of the construct. It associates a set of attributes with each grammar symbol; and with each production, it associates a set of semantic rules for computing the values of the attributes of the grammar symbols appearing in that production. Therefore, the grammar and the set of semantic rules constitute syntax-directed definitions.

## 6.2 IMPLEMENTATION OF THE TRANSLATIONS SPECIFIED BY SYNTAX-DIRECTED DEFINITIONS

Attributes are associated with the grammar symbols that are the labels of the parse tree nodes. They are thus associated with the construct's parse tree translation specification. Therefore, when a semantic rule is evaluated, the parser computes the value of an attribute at a parse tree node. For example, a semantic rule could specify the computation of the value of an attribute *val* that is associated with the grammar symbol *X* (a labeled parse tree node). To refer to the attribute *val* associated with the grammar symbol *X*, we use the notation *X.val*. Therefore, to evaluate the semantic rules and carry out translations, we must traverse the parse tree and get the values of the attributes at the nodes computed. The order in which we traverse the parse tree nodes depends on the dependencies of the attributes at the parse tree nodes. That is, if an attribute *val* at a parse tree node *X* depends on the attribute *val* at the parse tree node *Y*, as shown in [Figure 6.1](#), then the *val* attribute at node *X* cannot be computed unless the *val* attribute at *Y* is also computed.



**Figure 6.1:** The attribute value of node *X* is inherently dependent on the attribute value of node *Y*.

Hence, carrying out the translation specified by the syntax-directed definitions involves:

1. Generating the parse tree for the input string *W*,
2. Finding out the traversal order of the parse tree nodes by generating a dependency graph and doing a topological sort of that graph, and
3. Traversing the parse tree in the proper order and getting the semantic rules evaluated.

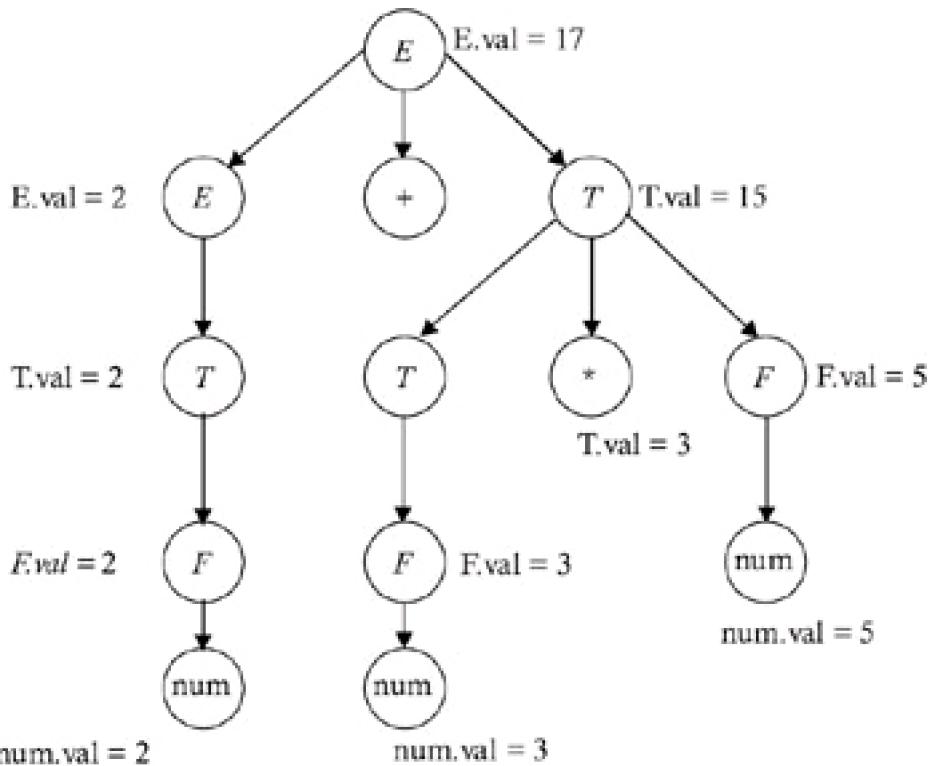
If the parse tree attribute's dependencies are such that an attribute of node *X* depends on the attributes of nodes generated before it in the parse tree-construction process, then it is possible to get *X*'s attribute value during the parsing itself; the parser is not required to generate an explicit parse tree, and the translations can be carried out along with the parsing. The attributes associated with a grammar symbol are classified into two categories: the synthesized and the inherited attributes of the grammar symbol.

### Synthesized Attributes

An attribute is said to be synthesized if its value at a parse tree node is determined by the attribute values at the child nodes. A synthesized attribute has a desirable property; it can be evaluated during a single bottom-up traversal of the parse tree. Synthesized attributes are, in practice, extensively used. Syntax-directed definitions that only use synthesized attributes are shown below:

$$\begin{array}{ll}
 E \rightarrow E_1 + T & E.val := E_1.val + T.val \\
 E \rightarrow T & E.val := T.val \\
 T \rightarrow T_2 * F & T.val := T_2.val * F.val \\
 T \rightarrow F & T.val := F.val \\
 F \rightarrow id & F.val := num.lexval
 \end{array}$$

These definitions specify the translations that must be carried by the expression evaluator. A parse tree, along with the values of the attributes at the nodes (called an "annotated parse tree"), for an expression  $2+3*5$  is shown in [Figure 6.2](#).



**Figure 6.2:** An annotated parse tree.

Syntax-directed definitions that only use synthesized attributes are known as "S-attributed" definitions. If translations are specified using S-attributed definitions, then the semantic rules can be conveniently evaluated by the *LR* parser itself during the parsing, thereby making translation more efficient. Therefore, S-attributed definitions constitute a subclass of the syntax-directed definitions that can be implemented using an *LR* parser.

### Inherited Attributes

Inherited attributes are those whose initial value at a node in the parse tree is defined in terms of the attributes of the parent and/or siblings of that node. For example, syntax-directed definitions that use inherited attributes are given below:

$D \rightarrow TL$	$L.type = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L1.id$	$L1.type = L.type$ $enter(id.prt, L.type)$
$L \rightarrow id$	$enter(id.prt, L.type)$

A parse tree, along with the attributes' values at the parse tree nodes, for an input string  $int\ id1,id2,id3$  is shown in [Figure 6.3](#).

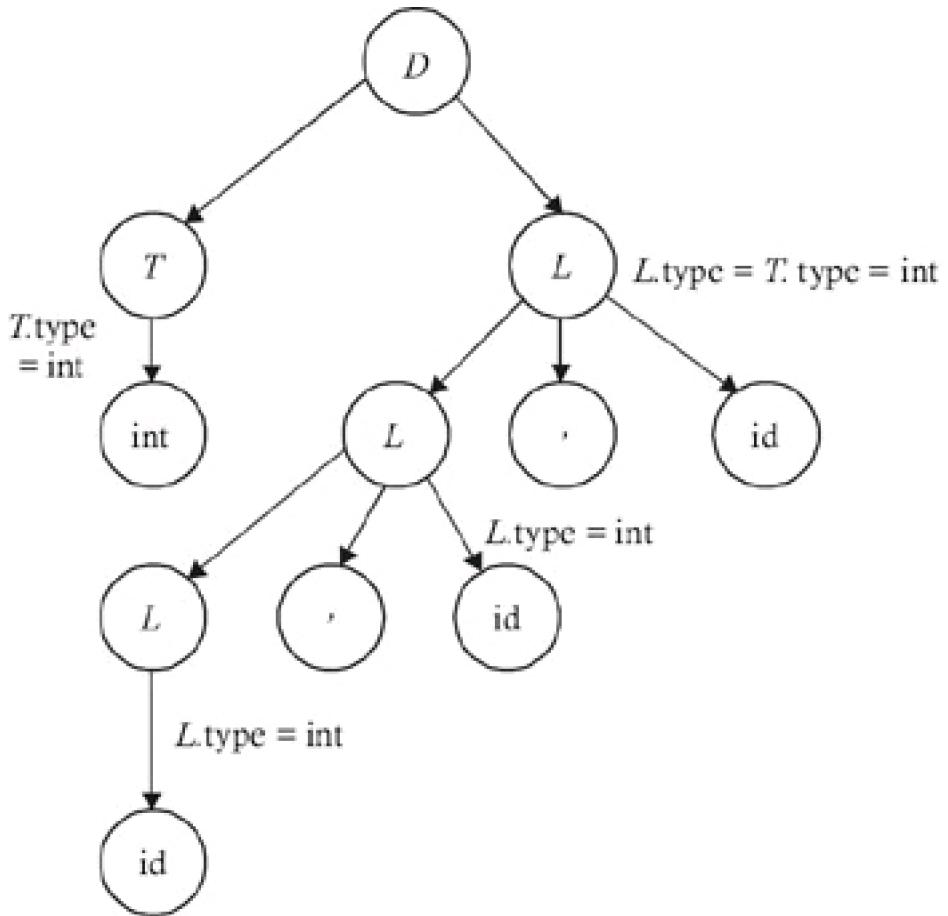


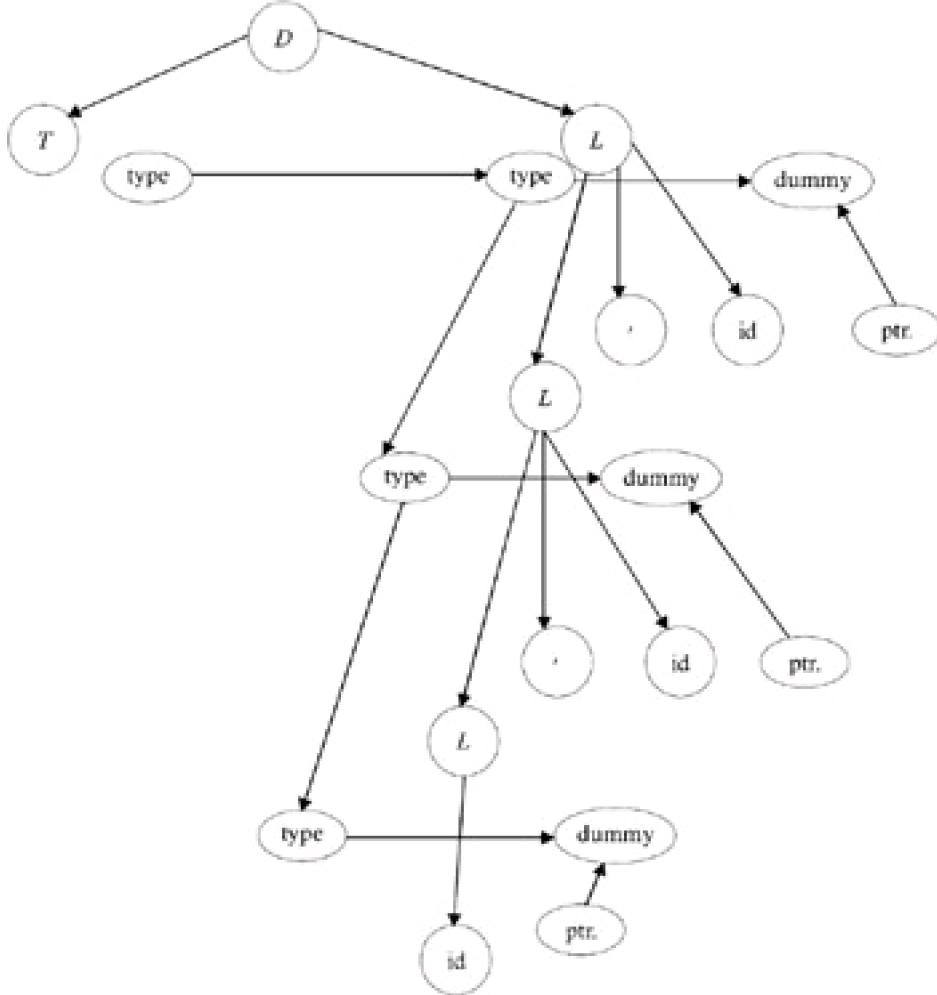
Figure 6.3: Parse tree with node attributes for the string `int id1,id2,id3`.

Inherited attributes are convenient for expressing the dependency of a programming language construct on the context in which it appears. When inherited attributes are used, then the interdependencies among the attributes at the nodes of the parse tree must be taken into account when evaluating their semantic rules, and these interdependencies among attributes are depicted by a directed graph called a "dependency graph". For example, if a semantic rule is of the form  $A.\text{val} = f(X.\text{val}, Y.\text{val}, Z.\text{val})$ —that is, if  $A.\text{val}$  is function of  $X.\text{val}$ ,  $Y.\text{val}$ , and  $Z.\text{val}$ —and is associated with a production  $A \rightarrow XYZ$ , then we conclude that  $A.\text{val}$  depends on  $X.\text{val}$ ,  $Y.\text{val}$ , and  $Z.\text{val}$ . Therefore, every semantic rule must adopt the above form (if it hasn't already) by introducing a dummy, synthesized attribute.

### Dummy Synthesized Attributes

If the semantic rule is in the form of a procedure call  $\text{fun}(a_1, a_2, a_3, \dots, a_k)$ , then we can transform it into the form  $b = \text{fun}(a_1, a_2, a_3, \dots, a_k)$ , where  $b$  is a dummy synthesized attribute. The dependency graph has a node for each attribute and an edge from node  $b$  to node  $a$  if attribute  $a$  depends on attribute  $b$ . For example, if a production  $A \rightarrow XYZ$  is used in the parse tree, then there will be four nodes in the dependency graph— $A.\text{val}$ ,  $X.\text{val}$ ,  $Y.\text{val}$ , and  $Z.\text{val}$ —with edges from  $X.\text{val}$ ,  $Y.\text{val}$ , and  $Z.\text{val}$  to  $A.\text{val}$ .

The dependency graph for such a parse tree is shown in [Figure 6.4](#). The ellipses denote the nodes of the dependency graph, and the circles denote the nodes of the parse tree.



**Figure 6.4:** Dependency graph with four nodes.

This topological sort of a dependency graph results in an order in which the semantic rules can be evaluated. But for reasons of efficiency, it is better to get the semantic rules evaluated (i.e., carry out the translation) during the parsing itself. If the translations are to be carried out during the parsing, then the evaluation order of the semantic rules gets linked to the order in which the parse tree nodes are created, even though the actual parse tree is not required to be generated by the parser. Many top-down as well as bottom-up parsers generate nodes in a depth-first left-to-right order; so the semantic rules must be evaluated in this same order if the translations are to be carried out during the parsing itself. A class of syntax-directed definitions, called "*L*-attributed" definitions, has attributes that can always be evaluated in depth-first, left-to-right order. Hence, if the translations are specified using *L*-attributed definitions, then it is possible to carry out translations during the parsing.

## 6.3 L-ATTRIBUTED DEFINITIONS

A syntax-directed definition is  $L$ -attributed if each inherited attribute of  $X_i$  for  $i$  between 1 and  $n$ , and on the right side of production  $A \rightarrow X_1X_2\dots X_n$ , depends only on:

1. The attributes (both inherited as well as synthesized) of the symbols  $X_1, X_2, \dots, X_{j-1}$  (i.e., the symbols to the left of  $X_j$  in the production, and
2. The inherited attributes of  $A$ .

The syntax-directed definition above is an example of the  $L$ -attributed definition, because the inherited attribute  $L.type$  depends on  $T.type$ , and  $T$  is to the left of  $L$  in the production  $D \rightarrow TL$ . Similarly, the inherited attribute  $L_1.type$  depends on the inherited attribute  $L.type$ , and  $L$  is parent of  $L_1$  in the production  $L \rightarrow L_1.id$ .

When translations carried out during parsing, the order in which the semantic rules are evaluated by the parser must be explicitly specified. Hence, instead of using the syntax-directed definitions, we use syntax-directed translation schemes to specify the translations. Syntax-directed definitions are more abstract specifications for translations; therefore, they hide many implementation details, freeing the user from having to explicitly specify the order in which translation takes place. Whereas the syntax-directed translation schemes indicate the order in which semantic rules are evaluated, allowing some implementation details to be specified.

## 6.4 SYNTAX-DIRECTED TRANSLATION SCHEMES

A syntax-directed translation scheme is a context-free grammar in which attributes are associated with the grammar symbols, and semantic actions, enclosed within braces ({}), are inserted in the right sides of the productions. These semantic actions are basically the subroutines that are called at the appropriate times by the parser, enabling the translation. The position of the semantic action on the right side of the production indicates the time when it will be called for execution by the parser. When we design a translation scheme, we must ensure that an attribute value is available when the action refers to it. This requires that:

1. An inherited attribute of a symbol on the right side of a production must be computed in an action immediately preceding (to the left of) that symbol, because it may be referred to by an action computing the inherited attribute of the symbol to the right of (following) it.
2. An action that computes the synthesized attribute of a nonterminal on the left side of the production should be placed at the end of the right side of the production, because it might refer to the attributes of any of the right-side grammar symbols. Therefore, unless they are computed, the synthesized attribute of a nonterminal on the left cannot be computed.

These restrictions are motivated by the  $L$ -attributed definitions. Below is an example of a syntax-directed translation scheme that satisfies these requirements, which are implemented during predictive parsing:

$$\begin{aligned} D \rightarrow T \{ L.type := T.type \} L; \\ T \rightarrow \text{int}\{ T.type := \text{int} \} \\ T \rightarrow \text{real}\{ T.type := \text{real} \} \\ L \rightarrow \{ L1.type = L.type \} L1, \text{id}\{\text{enter(id.prt, } L.type);\} \\ L \rightarrow \text{id}\{\text{enter(id.prt, } L.type);\} \end{aligned}$$

The advantage of a top-down parser is that semantic actions can be called in the middle of the productions. Thus, in the above translation scheme, while using the production  $D \rightarrow TL$  to expand  $D$ , we call a routine after recognizing  $T$  (i.e., after  $T$  has been fully expanded), thereby making it easier to handle the inherited attributes. Whereas a bottom-up parser reduces the right side of the production  $D \rightarrow TL$  by popping  $T$  and  $L$  from the top of the parser stack and replacing them by  $D$ , the value of the synthesized attribute  $T.type$  is already on the parser stack at a known position. It can be inherited by  $L$ . Since  $L.type$  is defined by a copy rule,  $L.type = T.type$ , the value of  $T.type$  can be used in place of  $L.type$ . Thus, if the parser stack is implemented as two parallel arrays—state and value—and state [ ] holds a grammar symbol  $X$ , then value [ ] holds a synthesized attribute of  $X$ . Therefore, the translation scheme implemented during bottom-up parsing is as follows, where [top] is value of stack top before the reduction and [newtop] is the value of the stack top after the reduction:

$$\begin{aligned} D \rightarrow Tl; \\ T \rightarrow \text{int}\{ \text{value[newtop]} = \text{int} \} \\ T \rightarrow \text{real}\{ \text{value[newtop]} = \text{real} \} \\ L \rightarrow L1, \text{id}\{\text{enter}(\text{value[top]}, \text{value[top-3]});\} \\ L \rightarrow \text{id}\{\text{enter}(\text{value[top]}, \text{value[top-1]});\} \end{aligned}$$

## 6.5 INTERMEDIATE CODE GENERATION

While translating a source program into a functionally equivalent object code representation, a parser may first generate an intermediate representation. This makes retargeting of the code possible and allows some optimizations to be carried out that would otherwise not be possible. The following are commonly used intermediate representations:

1. Postfix notation
2. Syntax tree
3. Three-address code

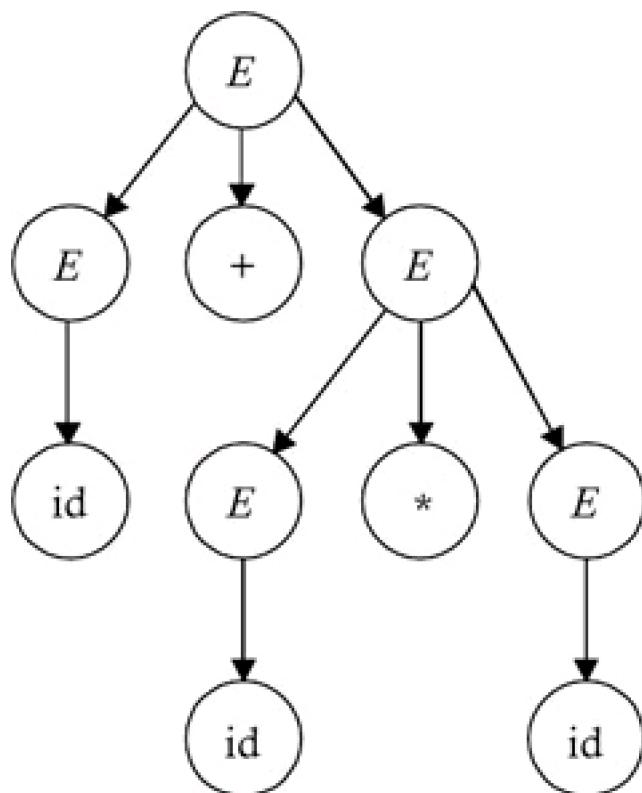
### Postfix Notation

In postfix notation, the operator follows the operand. For example, in the expression  $(a - b) * (c + d) + (a - b)$ , the postfix representation is:

$$Ab - Cd \times Ab - +$$

### Syntax Tree

The syntax tree is nothing more than a condensed form of the parse tree. The operator and keyword nodes of the parse tree ([Figure 6.5](#)) are moved to their parent, and a chain of single productions is replaced by single link ([Figure 6.6](#)).



**Figure 6.5:** Parse tree for the string  $\text{id} + \text{id} * \text{id}$ .

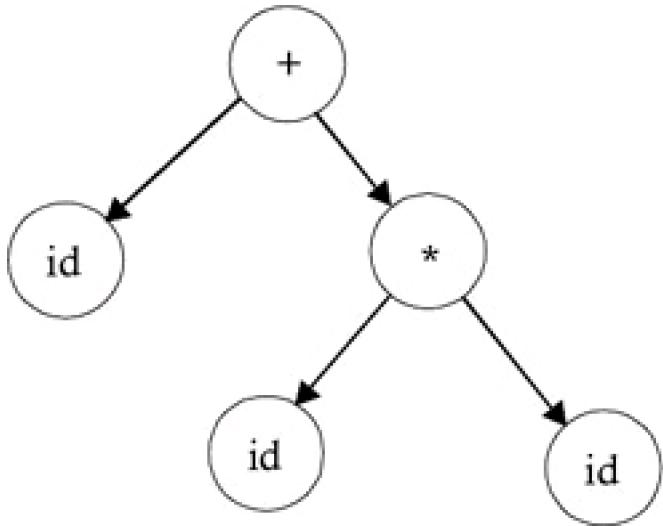


Figure 6.6: Syntax tree for  $\text{id} + \text{id} * \text{id}$ .

### Three-Address Code

Three address code is a sequence of statements of the form  $x = y \text{ op } z$ . Since a statement involves no more than three references, it is called a "three-address statement," and a sequence of such statements is referred to as three-address code. For example, the three-address code for the expression  $a + b * c + d$  is:

$$\begin{aligned} T_1 &= B * C \\ T_2 &= A + T_1 \\ T_3 &= T_2 + D \end{aligned}$$

Sometimes a statement might contain less than three references; but it is still called a three-address statement. The following are the three-address statements used to represent various programming language constructs:

- Used for representing arithmetic expressions:

$$\begin{aligned} X &= Y \text{ op } Z \\ X &= \text{op } Y \\ X &= Y \end{aligned}$$

- Used for representing Boolean expressions:

```

if A > B goto L
goto L

```

- Used for representing array references and dereferencing operations:

$$\begin{aligned} x &= y[i] \\ x[i] &= y \\ x &= *y \\ *x &= y \end{aligned}$$

- Used for representing a procedure call:

**param *T***

call  $p, n$

**Team LiB**

◀ PREVIOUS    **NEXT ▶**

## 6.6 REPRESENTING THREE-ADDRESS STATEMENTS

Records with fields for the operators and operands can be used to represent three-address statements. It is possible to use a record structure with four fields: the first holds the operator, the next two hold the operand1 and operand2, respectively, and the last one holds the result. This representation of a three-address statement is called a "quadruple representation".

### Quadruple Representation

Using quadruple representation, the three-address statement  $x = y \text{ op } z$  is represented by placing  $\text{op}$  in the operator field,  $y$  in the operand1 field,  $z$  in the operand2 field, and  $x$  in the result field. The statement  $x = \text{op } y$ , where  $\text{op}$  is a unary operator, is represented by placing  $\text{op}$  in the operator field,  $y$  in the operand1 field, and  $x$  in the result field; the operand2 field is not used. A statement like  $\text{param } t1$  is represented by placing  $\text{param}$  in the operator field and  $t1$  in the operand1 field; neither operand2 nor the result field are used. Unconditional and conditional jump statements are represented by placing the target labels in the result field. For example, a quadruple representation of the three-address code for the statement  $x = (a + b) * - c/d$  is shown in [Table 6.1](#). The numbers in parentheses represent the pointers to the triple structure.

**Table 6.1: Quadruple Representation of  $x = (a + b) * - c/d$**

	Operator	Operand1	Operand2	Result
(1)	+	a	b	t1
(2)	-	c		t2
(3)	*	t1	t2	t3
(4)	/	t3	d	t4
(5)	=	t4		x

### Triple Representation

The contents of the operand1, operand2, and result fields are therefore normally the pointers to the symbol records for the names represented by these fields. Hence, it becomes necessary to enter temporary names into the symbol table as they are created. This can be avoided by using the position of the statement to refer to a temporary value. If this is done, then a record structure with three fields is enough to represent the three-address statements: the first holds the operator value, and the next two holding values for the operand1 and operand2, respectively. Such a representation is called a "triple representation". The contents of the operand1 and operand2 fields are either pointers to the symbol table records, or they are pointers to records (for temporary names) within the triple representation itself. For example, a triple representation of the three-address code for the statement  $x = (a+b)*-c/d$  is shown in [Table 6.2](#).

**Table 6.2: Triple Representation of  $x = (a + b) * c/d$**

	Operator	Operand1	Operand2
(1)	+	a	b
(2)	-	c	
(3)	*	(1)	(2)
(4)	/	(3)	d
(5)	=	x	(4)

### Indirect Triple Representation

Another representation uses an additional array to list the pointers to the triples in the desired order. This is called an indirect triple representation. For example, a triple representation of the three-address code for the statement  $x = (a+b)*c/d$  is shown in [Table 6.3](#).

**Table 6.3: Indirect Triple Representation of  $x = (a + b) * c/d$**

		Operator	Operand1	Operand2
(1)	(1)	+	a	b
(2)	(2)	-	c	
(3)	(3)	*	(1)	(2)
(4)	(4)	/	(3)	d
(5)	(5)	=	x	(4)

### Comparison

By using quadruples, we can move a statement that computes A without requiring any changes in the statements using A, because the result field is explicit. However, in a triple representation, if we want to move a statement that defines a temporary value, then we must change all of the pointers in the operand1 and operand2 fields of the records in which this temporary value is used. Thus, quadruple representation is easier to work with when using an optimizing compiler, which entails a lot of code movement. Indirect triple representation presents no such problems, because a separate list of pointers to the triple structure is maintained. When statements are moved, this list is reordered, and no change in the triple structure is necessary; hence, the utility of indirect triples is almost the same as that of quadruples.

## 6.7 SYNTAX-DIRECTED TRANSLATION SCHEMES TO SPECIFY THE TRANSLATION OF VARIOUS PROGRAMMING LANGUAGE CONSTRUCTS

Specifying the translation of the construct involves specifying the construct's syntactic structure, using CFG, and associating suitable semantic actions with the productions of the CFG. For example, if we want to specify the translation of the arithmetic expressions into postfix notation so they can be carried along with the parsing, and if the parsing method is *LR*, then first we write a grammar that specifies the syntactic structure of the arithmetic expressions. We then associate suitable semantic actions with the productions of the grammar. The expressions used for these associations are covered below.

### 6.7.1 Arithmetic Expressions

The grammar that specifies the syntactic structure of the expressions in a typical programming language will have the following productions:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow id$$

Translating arithmetic expressions involves generating code to evaluate the given expression. Hence, for an expression  $a + b * c$ , the three-address code that is required to be generated is:

$$t1 = b * c$$

$$t2 = a + t1$$

where  $t1$  and  $t2$  are pointers to the symbol table records that contain compiler-generated temporaries, and  $a$ ,  $b$ , and  $c$  are pointers to the symbol table records that contain the programmer-defined names  $a$ ,  $b$ , and  $c$ , respectively.

Syntax-directed translation schemes to specify the translation of an expression into postfix notation are as follows:

$E \rightarrow E_1 + T$	{ $E.\text{code} = \text{concat}(E_1.\text{code}, T.\text{code}, "+");$ }
$E \rightarrow T$	{ $E.\text{code} = T.\text{code};$ }
$T \rightarrow T_1 * F$	{ $T.\text{code} = \text{concat}(T_1.\text{code}, F.\text{code}, "*");$ }
$T \rightarrow F$	{ $T.\text{code} = F.\text{code};$ }
$F \rightarrow id$	{ $F.\text{code} = \text{getname}(id.\text{place});$ }

where  $\text{code}$  is a string value attribute used to hold the postfix expression, and  $\text{place}$  is pointer value attribute used to link the pointer to the symbol record that contains the name of the identifier. The label  $\text{getname}$  returns the name of the identifier from the symbol table record that is pointed to by  $\text{ptr}$ , and  $\text{concat}(s1, s2, s3)$  returns the concatenation of the strings  $s1$ ,  $s2$ , and  $s3$ , respectively. For the string  $a+b*c$ , the values of the attributes at the parse tree node are shown in [Figure 6.7](#).

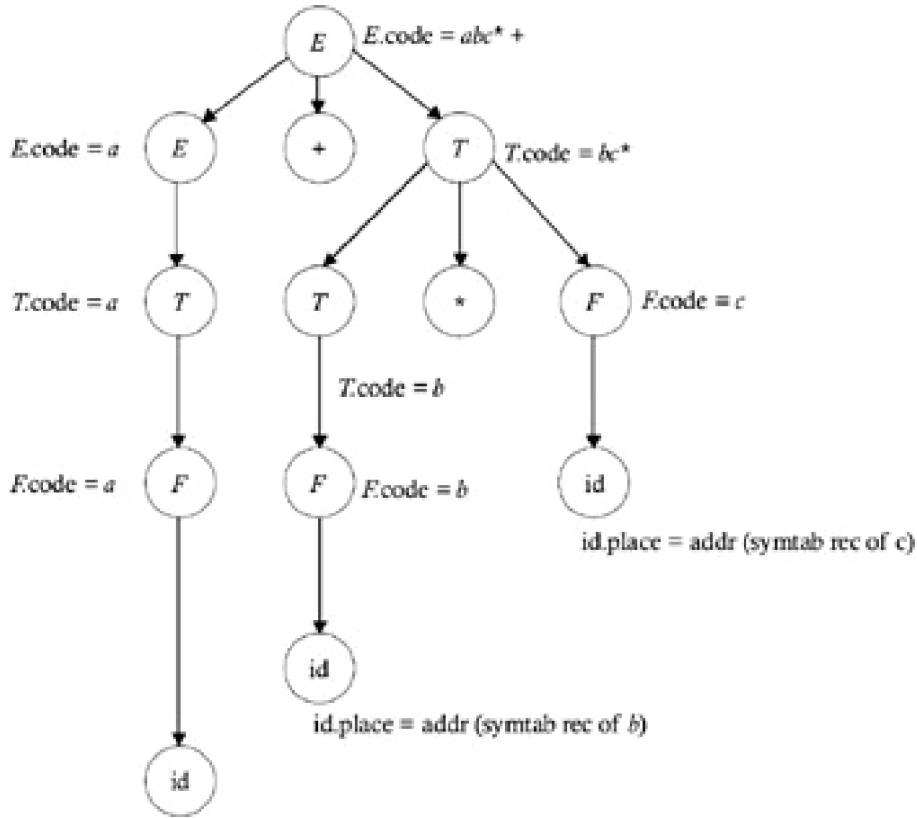


Figure 6.7: Values of attributes at the parse tree node for the string  $a + b * c$ .

#### **id.place = addr(symtab rec of a)**

Syntax-directed translation schemes to specify the translation of an expression into the syntax tree are as follows:

$$\begin{array}{ll}
 E \rightarrow E_1 + T & \{E.\text{ptr} = \text{mknnode} ('+', E_1.\text{ptr}, T.\text{ptr})\} \\
 E \rightarrow T & \{E.\text{ptr} = T.\text{ptr}\} \\
 T \rightarrow T_1 * F & \{T.\text{ptr} = \text{mknnode} ('\ast', T_1.\text{ptr}, F.\text{ptr})\} \\
 T \rightarrow F & \{T.\text{ptr} = F.\text{ptr}\} \\
 F \rightarrow \text{id} & \{F.\text{ptr} = \text{mkleaf} (\text{id.place})\}
 \end{array}$$

where ptr is pointer value attribute used to link the pointer to a node in the syntax tree, and place is pointer value attribute used to link the pointer to the symbol record that contains the name of the identifier. The mkleaf generates leaf nodes, and mknnode generates intermediate nodes.

For the string  $a+b*c$ , the values of the attributes at the parse tree nodes are shown in [Figure 6.8](#).

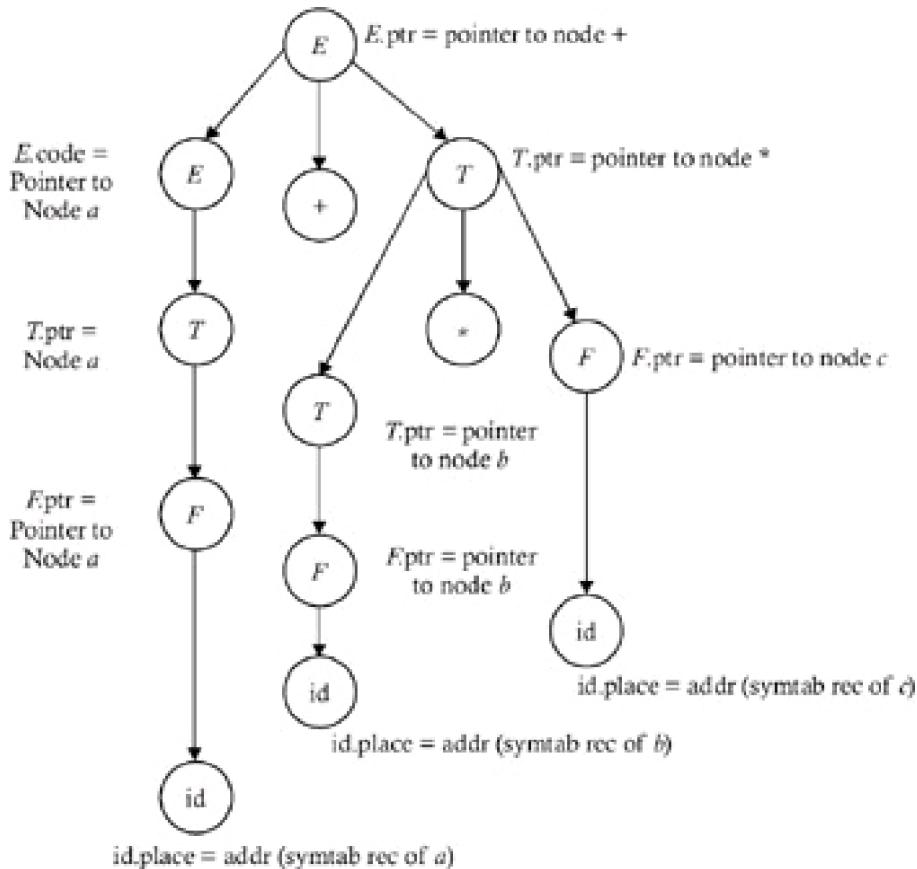


Figure 6.8: Values of the attributes at the parse tree nodes for  $a + b * c$ ,  $\text{id.place} = \text{addr}(\text{symtab rec of } a)$ .

### **id.place = addr(sumtab rec of a)**

Syntax-directed translation schemes specify the translation of an expression into three-address code, as follows:

$E \rightarrow E_1 + T$	$\{ \$_2 = \text{gentemp}();$
	$\text{gencode}('+' , E_1.\text{place}, T.\text{place});$
	$E.\text{place} = \$_2\}$
$E \rightarrow T$	$\{ E.\text{place} = T.\text{place}\}$
$T \rightarrow T_1 * F$	$\{ \$_1 = \text{gentemp}();$
	$\text{gencode}('*' , T_1.\text{place}, F.\text{place});$
	$T.\text{place} = \$_1\}$
$T \rightarrow F$	$\{ T.\text{ptr} = F.\text{ptr}\}$
$F \rightarrow \text{id}$	$\{ F.\text{ptr} = \text{id.place}\}$

where ptr is a pointer value attribute used to link the pointer to the symbol record that contains the name of the identifier, mkleaf generates leafnodes, and mknnode generates intermediate nodes. For the string  $a+b*c$ , the values of the attributes at the parse tree nodes are shown in [Figure 6.9](#).

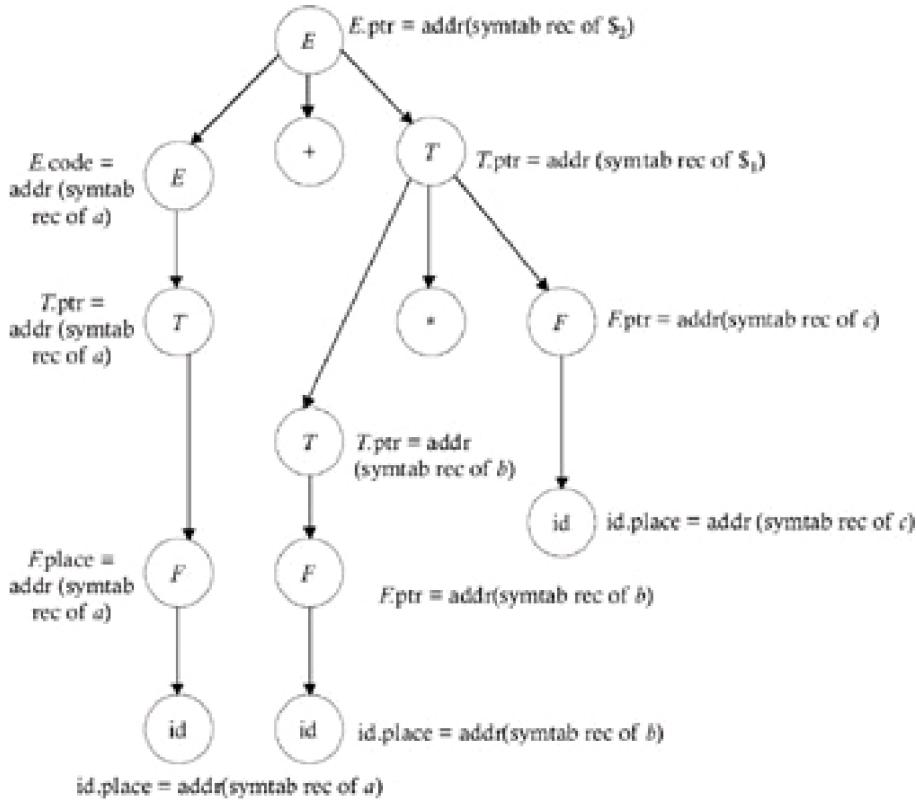


Figure 6.9: Values of the attributes at the parse tree nodes for  $a + b * c$ ,  $\text{id.place} = \text{addr}(\text{sumtab rec of } a)$ .

### 6.7.2 Boolean Expressions

One way of translating a Boolean expression is to encode the expression's true and false values as the integers one and zero, respectively. The code to evaluate the value of the expression in some temporary is generated as shown below:

```

 $E \rightarrow E_1 \text{ relop } E_2$ 
{
    t1 = gentemp();
    gencode(if  $E_1.\text{place}$  relop.val  $E_2.\text{place}$ 
    goto(nextquad + 3));
    gencode(t1 = 0);
    gencode(goto(nextquad+2))
    gencode(t1 = 1)
    E.place = t1;
}

```

where nextquad keeps track of the index in the code array. The next statement will be inserted by the gencode procedure, and will update the value of nextquad. The following translation scheme:

$\text{relop} \rightarrow <$	{ $\text{relop.val} = '<' $ }
$\text{relop} \rightarrow >$	{ $\text{relop.val} = '>' $ }
$\text{relop} \rightarrow <=$	{ $\text{relop.val} = '<=' $ }
$\text{relop} \rightarrow >=$	{ $\text{relop.val} = '>=' $ }
$\text{relop} \rightarrow ==$	{ $\text{relop.val} = '==' $ }
$\text{relop} \rightarrow !=$	{ $\text{relop.val} = '!='$ }

translates the expression  $a < b$  to the following three-address code:

i)      if  $a < b$  goto  $i + 3$   
 ii)      $a = 0$

```

i+1)    t1 = 0
i+2)    goto i+4
i+3)    t1 = 1
i+4)

```

Similarly, a Boolean expression formed by using logical operators involves generating code to evaluate those operators in some temporary form, as shown below:

```

E → E1 lop E2
{
  t1 = gentemp();
  gencode (t1 = E1.place lop.val E2.place);
  E.place = t1;
}

E → not E1
{
  t1 = gentemp();
  gencode (t1 = not E1.place)
  E.place = t1
}

lop → and { lop.val = and}
lop → or { lop.val = or}

```

The translation scheme above translates the expressions  $a < b$  and  $c > d$  to the following three-address code:

```

i)      if a < b goto i+3
i+1)   t1 = 0
i+2)   goto i+4
i+3)   t1 = 1
i+4)   if c > d goto i+7
i+5)   t2 = 0
i+6)   goto i+8
i+7)   t2 = 1
i+8)   t3 = t1 and t2

```

Another way to translate a Boolean expression is to represent its value by a position in the three-address code sequence. For example, if we point to the statement labeled  $L1$ , then the value of the expression is true (1); whereas if we point to the statement labeled  $L2$ , then the value of the expression is false (0). In this case, the use of a temporary to hold either a one or zero, depending upon the true or false value of the expression, becomes redundant. This also makes it possible to decide the value of the expression without evaluating it completely. This is called a "short circuit" or "jumping the code". To discover the true/false value of the expression  $a < b$  or  $c > d$ , it is not necessary to completely evaluate the expression; if  $a < b$  is true, then the entire expression will be true. Similarly to discover the true/false value of the expression  $a < b$  and  $c > d$ , it is not necessary to completely evaluate the expression, because if  $a < b$  is false, then the entire expression will be false.

**Tip** Therefore a Boolean expression can be translated into two to three address statements, a conditional jump, and an unconditional jump. But the targets of these jumps are known at the time of translating a Boolean expression; hence, these jumps are generated without their targets, which are filled in later on.

Therefore, we must remember the indices of these jumps in the code array by using suitable attributes of  $E$ . For this, we use two pointer value attributes:  $E.\text{true}$  and  $E.\text{false}$ . The attribute  $E.\text{true}$  will hold the pointer to the list that contains the index of the conditional jump in the code array, whereas the attribute  $E.\text{false}$  will hold the pointer to the list that contains the index of the unconditional jump. The translation scheme for the Boolean expression that uses relational operators is as follows:

```

 $E \rightarrow E_1 \text{ relop } E_2$ 
{
  E.true = mklist(nextquad);
  E.false = mklist(nextquad + 1);
  gencode (if  $E_1$ .place relop.val  $E_2$ .place goto);
  gencode (goto_);
}

```

where mklist(ind) is a procedure that creates a list containing ind and returns a pointer to the created list.

$\text{relop} \rightarrow <$	{ relop.val = '<' }
$\text{relop} \rightarrow >$	{ relop.val = '>' }
$\text{relop} \rightarrow <=$	{ relop.val = '<=' }
$\text{relop} \rightarrow >=$	{ relop.val = '>=' }
$\text{relop} \rightarrow ==$	{ relop.val = '==' }
$\text{relop} \rightarrow !=$	{ relop.val = '!=' }

The above translation scheme translates the expression  $a < b$  to the following three address code:

$E.\text{true}$	$\xrightarrow{I}$	if $a < b$ goto_
$E.\text{false}$	$\xrightarrow{I+2)$	goto_

### 6.7.3 Short-Circuit Code for Logical Expressions

There are several methods to adequately handle the various elements of Boolean operators. These are covered by type below.

#### AND

Logical expressions that use the 'and' operator are expressions defined by the production  $E \rightarrow E_1 \text{ and } E_2$ . Generating the short-circuit code for these logical expressions involves setting the true value of the first expression,  $E_1$ , to the start of the second expression,  $E_2$ , in the code array. We make the true value of  $E$  the same as the true value of expression  $E_2$ ; and we make the false value of  $E$  the same as the false values of both  $E_1$  and  $E_2$ . This requires remembering where  $E_2$  starts in the code array index, which means we must provision the memory of the nextquad value just before  $E_2$  is processed. This can be accomplished by introducing a nullable nonterminal  $M$  before  $E_2$  in the above production, providing for a reduction by  $M \rightarrow \epsilon$  just before the processing of  $E_2$ . Hence, we can get a semantic action associated with this production and executed at this point. We therefore have a method for remembering the value of nextquad just before the  $E_2$  code is generated.

```

 $E \rightarrow E_1 \text{ and } M E_2$  { backpatch(E1.true, M.quad);
  E.true = E2.true;
  E.false = merge(E1.false, E2.false);
}
 $M \rightarrow \epsilon$  {M.quad = nextquad; }

```

where backpatch(ptr,L) is a procedure that takes a pointer ptr to a list containing indices of the code array and fills the target of the statements at these indices in the code array by L.

#### OR

For an expression using the 'or' operator—that is, an expression defined by the production  $E \rightarrow E_1 \text{ or } E_2$ —generating the short-circuit code involves setting the false value of the first expression,  $E_1$ , to the start of  $E_2$  in the code array, and making the false value of  $E$  the same as the false value of  $E_2$ . The true value of  $E$  is assigned the same true value as both  $E_1$  and  $E_2$ . This requires remembering where  $E_2$  starts in the code array index, which requires making a provision for remembering the value of nextquad just before the expression  $E_2$  is processed. This can be achieved by introducing a nullable nonterminal  $M$  before  $E_2$  in the above production, providing for a reduction by  $M \rightarrow \epsilon$  just before

the processing of  $E_2$ . Hence, we obtain a semantic action that is associated with this production and executed at this point; therefore, we have provisioned the recall of the value of nextquad just before the  $E_2$  code is generated.

```

 $E \rightarrow E_1 \text{ or } M \ E_2 \quad \{ \text{ backpatch}(E_1.\text{false}, M.\text{quad});$ 
 $\quad \quad \quad E.\text{false} = E_2.\text{false};$ 
 $\quad \quad \quad E.\text{true} = \text{merge}(E_1.\text{true}, E_2.\text{true});$ 
 $\quad \}$ 
 $M \rightarrow \in \quad \{ M.\text{quad} = \text{nextquad}; \}$ 

```

## NOT

For the logical expression using the 'not' operator, that is, one defined by the production  $E \rightarrow \text{not } E_1$ , generating the short-circuit code involves making the false value of the expression  $E$  the same as the true value of  $E_1$ . And the true value of  $E$  is assigned the false value of  $E_1$ .

```

 $E \rightarrow \text{not } E_1 \quad \{$ 
 $\quad E.\text{true} = E_1.\text{false}$ 
 $\quad E.\text{false} = E_1.\text{true}$ 
 $\}$ 

```

The above translation scheme translates the expression  $a < b$  and  $c > d$  to the following three-address code:

$I)$ $E.\text{false}$	<b>if <math>a &lt; b</math> goto <math>I+2</math></b> <b><math>I+1)</math> goto_</b>
	<b><math>+2)</math> if <math>c &gt; d</math> goto_</b> <b><math>I+3)</math> goto_</b>
	<b><math>E.\text{true}</math></b>

For example, consider the following Boolean expression:

**not ( $P < Q$  and  $R < S$  or not ( $T < U$  and  $R < Q$ ))**

When the above translation scheme is used to translate this construct, the three-address code generated for it is as shown below, and the translation scheme is shown in [Figure 6.10](#).

$i)$ $i+1)$ $i+2)$ $i+3)$ $i+4)$ $i+5)$ $i+6)$ $i+7)$	<b>if <math>p &lt; q</math> goto(<math>i+2</math>)</b> <b>goto(<math>i+4</math>)</b> <b>if <math>r &lt; s</math> goto_</b> <b>goto(<math>i+4</math>)</b> <b>if <math>t &lt; u</math> goto(<math>i+6</math>)</b> <b>goto_</b> <b>if <math>r &lt; q</math> goto_</b> <b>goto_</b>
--	--



**Figure 6.10:** Translation scheme for a Boolean expression containing and, not, and or.

## IF-THEN-ELSE

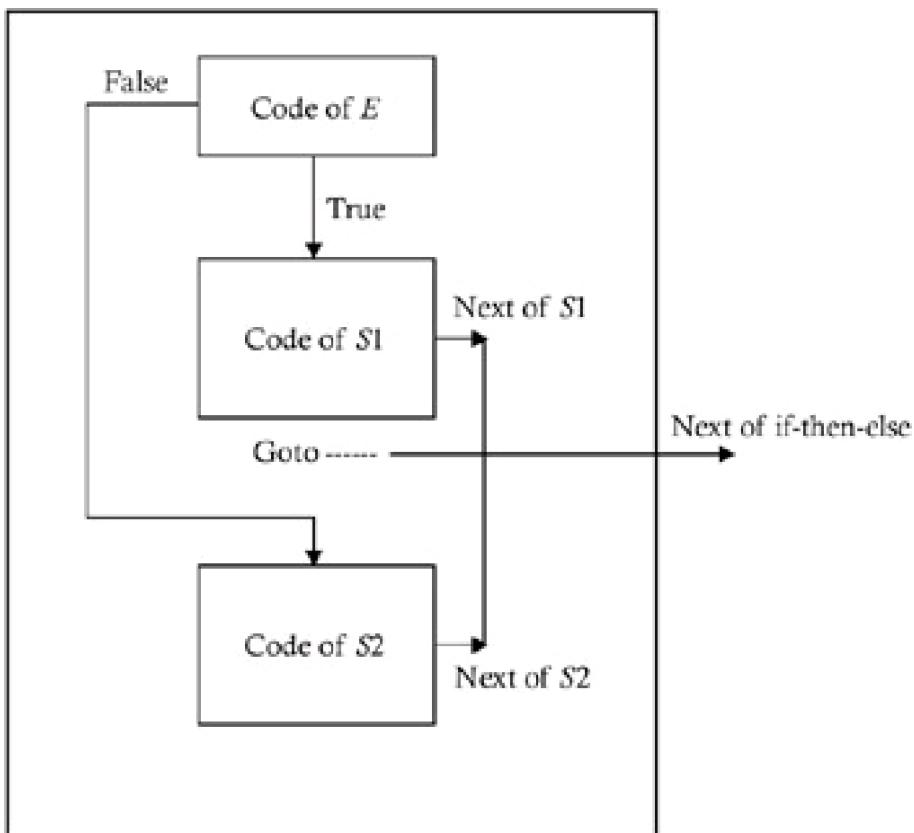
Since an if-then-else statement is composed of three components—a boolean expression  $E$ , a statement  $S1$  that is to be executed when  $E$  is true, and a statement  $S2$  that is to be executed when  $E$  is false—the translation of if-then-else involves making a provision for transferring control to the start of  $S1$  if  $E$  is true, for transferring control to the start of  $S2$  if  $E$  is false, and for transferring control to the next statement after the execution of  $S1$  and  $S2$  is over. This requires remembering where  $S1$  starts in the index of the code array as well as remembering where  $S2$  starts in the index of the code array.

This is achieved by introducing a nullable nonterminal  $M1$  before the  $S1$  and a nullable nonterminal  $M2$  before the  $S2$  in the above production, providing for the reduction by  $M1 \rightarrow \epsilon$  just before processing  $S1$ . Hence, we get a semantic action associated with this production and executed at this point, which enables the recall of the value of nextquad just before generating  $S1$  code. Similarly, it provides for the reduction by  $M2 \rightarrow \epsilon$  just before processing  $S2$ , and we get a semantic action associated with production executed at this point, enabling the recall of the value of nextquad just before generating  $S2$  code.

In addition, an unconditional jump is required at the end of  $S1$  in order to transfer control to the statement that follows the if-then-else statement. To generate this unconditional jump, we add a nullable nonterminal  $N$  after  $S1$  to the production and associate a semantic action with the production  $N \rightarrow \epsilon$ , which takes care of generating this unconditional jump, as shown in [Figure 6.11](#).

```

S → if E then M1 S1 N
  else M2 S2 {
    backpatch (E.true, M1.quad)
    backpatch (E.false, M2.quad)
    S.next:
      = merge (S1.next, S2.next, N.next)
  }
M1 → ε { M1.quad = nextquad;}
M2 → ε { M2.quad = nextquad}
N → ε {
  N.next = mklist (nextquad);
  gencode (goto...);
}
  
```



**Figure 6.11:** The addition of the nullable nonterminal N facilitates an unconditional jump.

Hence, for the statement if  $a < b$  then  $x = y + z$  else  $p = q + r$ , the three-address code that is required to be generated is:

- i)      **if  $a < b$  goto( $i + 2$ )**
- $i+1)$     **goto( $i + 5$ )**
- $i+2)$      **$t1 = y + z$**
- $i+3)$      **$x = t1$**
- $i+4)$     **goto...**
- $i+5)$      **$t2 = q + r$**
- $i+6)$      **$p = t2$**

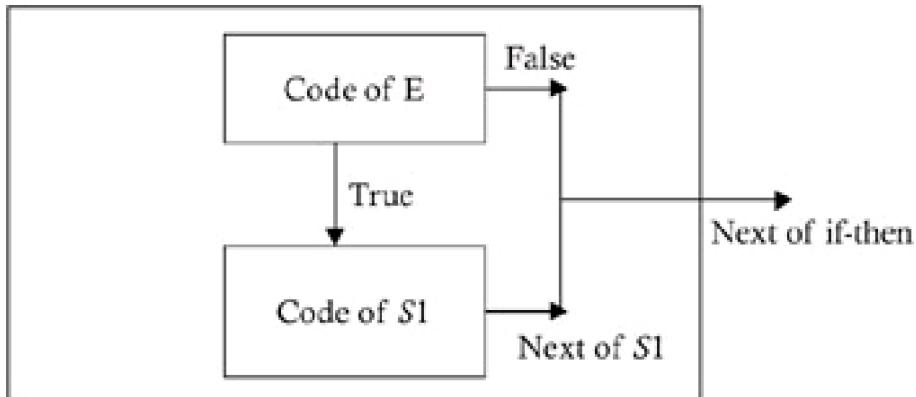
## IF-THEN

Since an if-then statement is comprised of two components, a Boolean expression  $E$  and an  $S1$  statement that will be executed when  $E$  is true, the translation of if-then involves making a provision for transferring control to the start of  $S1$  code if  $E$  is either true or false, and a provision is made for transferring control to the next statement after the execution of  $S1$  is over. This requires recalling the index of the start of  $S1$  in the code array, and can be achieved by introducing a nullable nonterminal  $M$  before  $S1$  in the production. This will provide for a reduction by  $M \rightarrow \epsilon$ , just before the processing of  $S1$ . Hence, we can get a semantic action associated with this production and executed at this point, which makes a provisioning the recall of for remembering the value of nextquad just before generating code of  $S1$  code is generated, as shown in [Figure 6.12](#) below:

```

S → if E then M S1  {
    backpatch (E.true, M.quad);
    S.next = merge(E.false, S1.next)
}
M → ε { M.quad = nextquad; }

```



**Figure 6.12:** A nullable nonterminal  $M$  provisions the translation of if-then.

Hence, for the statement if  $a < b$  then  $x = y + z$ , the three-address code that is required to be generated is:

- i)      **if  $a < b$  goto( $i + 2$ )**
- $i+1)$     **goto...**
- $i+2)$      **$t1 = y + z$**
- $i+3)$      **$x = t1$**

## WHILE

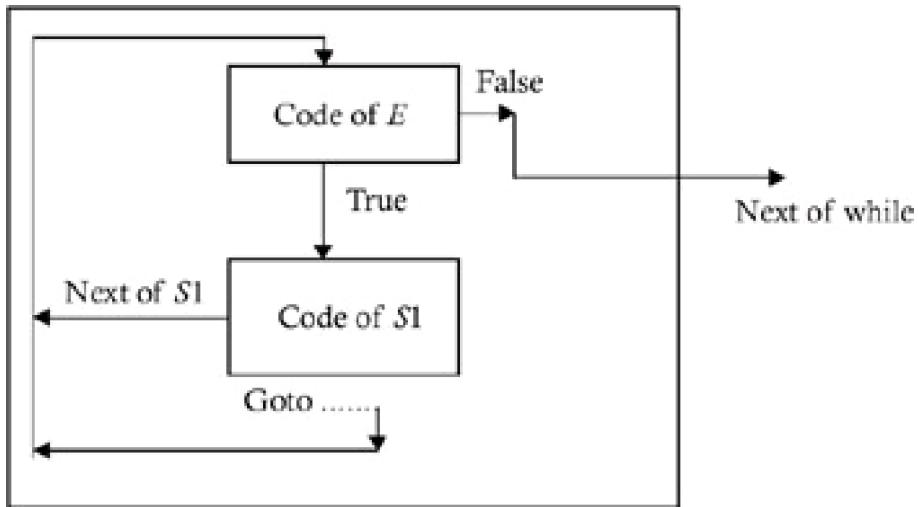
Since a while statement has two components, a Boolean expression  $E$  and a statement  $S1$ , which is the statement to

be executed repeatedly as long as  $E$  is true, the translation of while involves provisioning the transfer of control to the start of  $S1$  code if  $E$  is true. The expression must be tested again after  $S1$  execution is over, control must be transferred to the next statement if  $E$  is false. This requires remembering the index in the code array where  $S1$  code starts as well as where the  $E$  code starts. This can be achieved by introducing a nullable nonterminal  $M2$  before  $S1$  in the production. This will provide for the reduction by  $M2 \rightarrow \epsilon$  just before the processing of  $S1$ . Hence, a semantic action is associated with this production and is executed at this point, enabling the recall of the value of nextquad just before generating  $S$  code. Similarly, introducing a nullable nonterminal  $M1$  before  $E$  will provide for the reduction by  $M1 \rightarrow \epsilon$  just before the processing of  $E$ . Hence, a semantic action is now associated with this production and is executed at this point, provisioning the recall of the value of nextquad just before  $E$  code is generated, as shown in [Figure 6.13](#).

```

 $S \rightarrow \text{while } M1 \ E$ 
  do  $M2 \ S1 \ {$ 
    backpatch ( $E.\text{true}$ ,  $M2.\text{quad}$ )
    backpatch ( $S1.\text{next}$ ,  $M1.\text{quad}$ )
     $S.\text{next} = E.\text{false}$ 
    gencode (goto( $M1.\text{quad}$ ))
  }
 $M1 \rightarrow \epsilon \ \{ \ M1.\text{quad} = \text{nextquad}; \}$ 
 $M2 \rightarrow \epsilon \ \{ \ M2.\text{quad} = \text{nextquad}; \}$ 

```



**Figure 6.13:** The translation of the Boolean while statement is facilitated by a nullable nonterminal  $M$ .

Hence, for the statement  $\text{while } a < b \text{ do } x = y + z$ , the three-address code that is required to be generated is:

$i$ )	$\text{if } a < b \text{ goto}(i+2)$
$i+1$ )	$\text{goto} \dots$
$i+2$ )	$t1 = y + z$
$i+3$ )	$x = t1$
$i+4$ )	$\text{goto}(i)$

## DO-WHILE

Since a do-while statement is comprised of two components, a Boolean expression  $E$  and an  $S1$  statement that is executed repeatedly as long as  $E$  is true (as well as the test for whether  $E$  is true or false at the end of  $S1$  execution), translation involves provisioning the transfer of control to test the expression after the execution of  $S1$  is over. Control must also be transferred to the start of  $S1$  code if  $E$  is true, and conversely to the next statement if  $E$  is false.

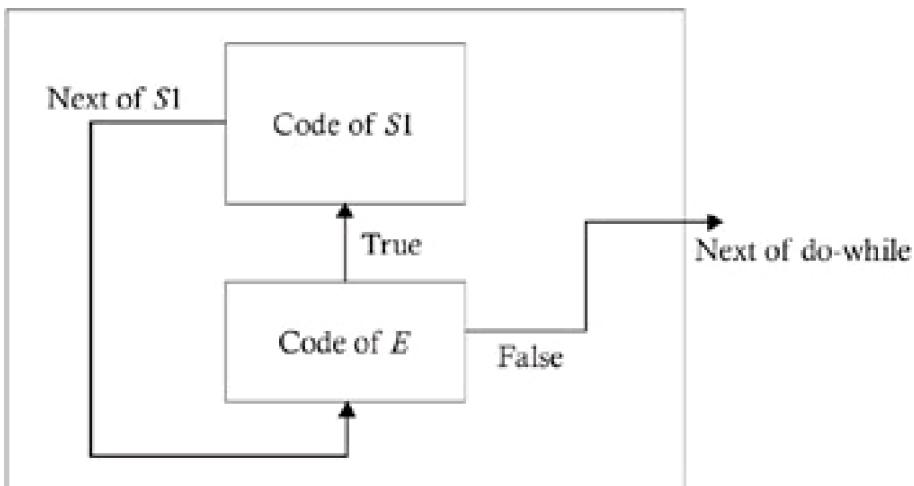
This requires recalling the  $S1$  start index in the code array as well as the  $E$  start index. We introduce a nullable nonterminal  $M1$  before  $S1$  in the production, providing for the reduction by  $M1 \rightarrow \epsilon$  just before the processing of  $S1$ . Hence, a semantic action is now associated with this production and is executed at this point, provisioning the recall of

the value of nextquad just before  $S1$  code generates. Similarly, introducing a nullable nonterminal  $M2$  before  $E$  will provide for the reduction by  $M2 \rightarrow \epsilon$  just before the processing of  $E$ . We then have a semantic action associated with this production and executed at this point, and which provisions the recall of the value of nextquad just before  $E$  code generates, as shown in [Figure 6.14](#).

```

 $S \rightarrow \text{do } M1 \ S1 \text{ while } M2 \ E \{$ 
    backpatch ( $E.\text{true}$ ,  $M1.\text{quad}$ )
    backpatch ( $S1.\text{next}$ ,  $M2.\text{quad}$ )
     $S.\text{next} = E.\text{false}$ 
}
 $M1 \rightarrow \epsilon \{ \ M1.\text{quad} = \text{nextquad}; \}$ 
 $M2 \rightarrow \epsilon \{ \ M2.\text{quad} = \text{nextquad}; \}$ 

```



**Figure 6.14:** Translation of the Boolean do-while.

Hence, for a statement  $\text{do } x = y + z \text{ while } a < b$ , the three-address code that is required to be generated is:

$i$	$t1 = y + z$
$i+1$	$x = t1$
$i+2$	$\text{if } a < b \text{ goto}(i)$
$i+3$	$\text{goto} \dots$

## REPEAT-UNTIL

Since a repeat-until statement has two components, a Boolean expression  $E$  and an  $S1$  statement that is executed repeatedly until  $E$  becomes true (as well as the test of whether  $E$  is true or false at the end of  $S1$ ), the translation of repeat-until involves provisioning transfer of control to a test of the expression after the execution of  $S1$  is over. We must also engineer a transfer a control to the start code of  $S1$  if  $E$  is false and to the next statement if  $E$  is true.

This requires recalling the index in the code array where  $S1$  code starts as well as the index in the code array where  $E$  code starts. We achieve this by introducing a nullable nonterminal  $M1$  before  $S1$  in the production. This will provide for the reduction by  $M1 \rightarrow \epsilon$ , just before the processing of  $S1$ . Hence, we can get a semantic action that is associated with this production and is executed at this point. This makes a provision for remembering the value of nextquad just before  $S$  code generates, and introduces a nullable non-terminal  $M2$  before  $E$ . This will provide for the reduction by  $M2 \rightarrow \epsilon$ , just before the processing of  $E$ . Now we can get a semantic action associated with this production and executed at this point, and which provisions the recall of the value of nextquad just before  $E$  code generates, as shown in [Figure 6.15](#).

```

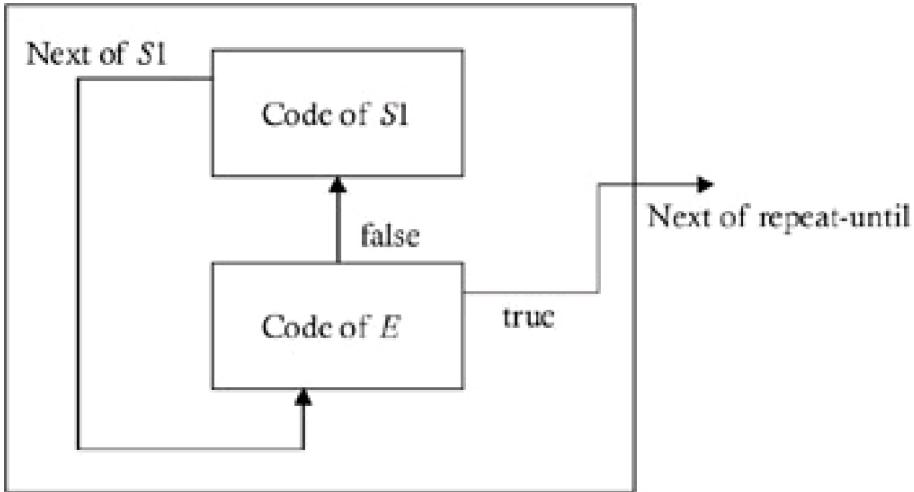
 $S \rightarrow \text{repeat } M1 \ S1$ 
     $\text{until } M2 \ E \ {$ 
        backpatch ( $E.\text{false}$ ,  $M1.\text{quad}$ )
        backpatch ( $S1.\text{next}$ ,  $M2.\text{quad}$ )
         $S.\text{next} = E.\text{true}$ 
}

```

```

    }
M1 → ∈ { M1.quad = nextquad; }
M2 → ∈ { M2.quad = nextquad; }

```



**Figure 6.15:** Translation of Boolean repeat-until.

Hence, for the Boolean statement `repeat x = y + z until a < b`, the three-address code that is required to be generated is:

- |             |                            |
|-------------|----------------------------|
| <i>i)</i>   | <i>t1 = y + z</i>          |
| <i>i+1)</i> | <i>x = t1</i>              |
| <i>i+2)</i> | <i>if a &lt; b goto...</i> |
| <i>i+3)</i> | <i>goto(i)</i>             |

## FOR

A for statement is composed of four components: an expression  $E_1$ , which is used to initialize the iteration variable; an expression  $E_2$ , which is a Boolean expression used to test whether or not the value of the iteration variable exceeds the final value; an expression  $E_3$ , which is used to specify the step by which the value of the iteration variable is to be incremented or decremented; and an  $S_1$  statement, which is the statement to be executed as long as the value of the iteration variable is less than or equal to the final value. Hence, the translation of a for statement involves provisioning the transfer a control to the start of  $S_1$  code if  $E_2$  is true, transferring control to the start of  $E_3$  code after the execution of  $S_1$  is over, transferring control to the start of  $E_2$  code after  $E_3$  code is executed, and transferring control to the next statement if  $E_2$  is false, as shown in [Figure 6.16](#).

$S \rightarrow \text{for } (E_1; M1 \ E_2; M2 \ E_3) M3 \ S1$

```

{
    backpatch (E2.true, M3.quad)
    backpatch (M3.next, M1.quad)
    backpatch (S1.next, M2.quad)
    gencode (goto(M2.quad))
    S.next = E2.false
}

```

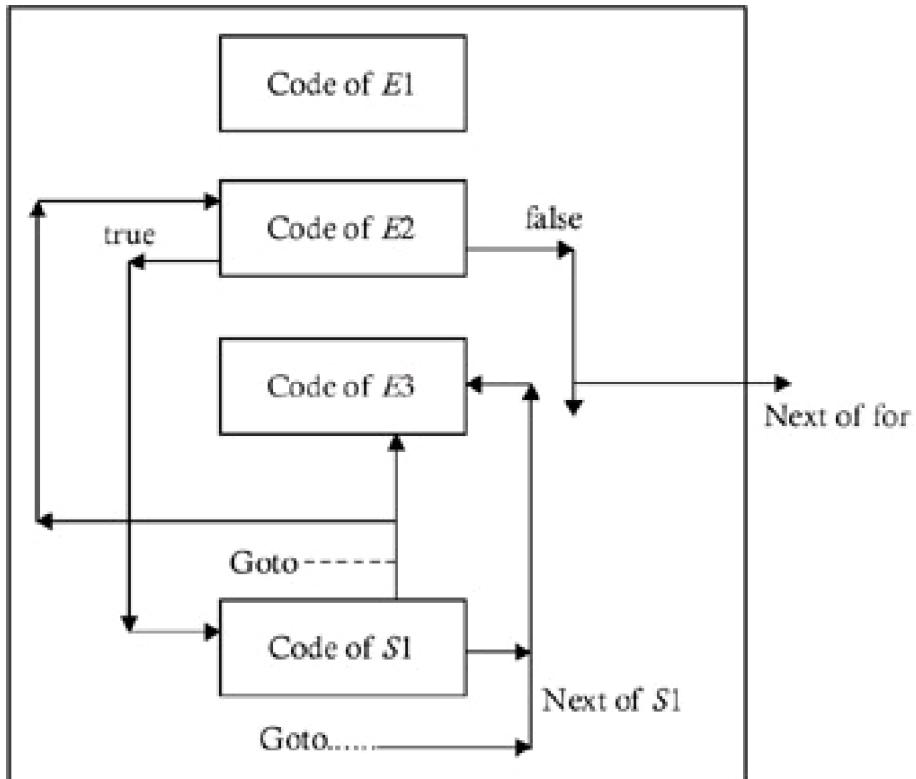
$M1 \rightarrow \in \{ M1.\text{quad} = \text{nextquad}; \}$

$M2 \rightarrow \in \{ M2.\text{quad} = \text{nextquad}; \}$

```

M3 \rightarrow \in {
    M3.next: = mklist (nextquad)
    gencode (goto...)
    M3.quad = nextquad;
}

```



**Figure 6.16:** Handling the translation of the Boolean for.

Hence, for a statement `for(i = 1; i <= 20; i++) x = y + z;`, the three-address code that is required to be generated is:

<i>j</i> )	<i>i</i> = 1
<i>j+1</i> )	if <i>i</i> <= 20 goto( <i>j+6</i> )
<i>j+2</i> )	goto...
<i>j+3</i> )	<i>t1</i> = <i>i</i> +1
<i>j+4</i> )	<i>i</i> = <i>t1</i>
<i>j+5</i> )	goto( <i>j+1</i> )
<i>j+6</i> )	<i>t2</i> = <i>y</i> + <i>z</i>
<i>j+7</i> )	<i>x</i> = <i>t2</i>
<i>j+8</i> )	goto( <i>j+3</i> )

## 6.8 IMPLEMENTATION OF INCREMENT AND DECREMENT OPERATORS

```
L → id++ {  
    t1 = gentemp();  
    t2 = gentemp();  
    gencode(t1 = id.place);  
    gencode(t2 = id.place +1);  
    gencode(id.place = t2);  
    L.place = t1;  
}  
L → ++id {  
    t1 = gentemp();  
    gencode(t1 = id.place +1);  
    gencode(id.place = t1);  
    L.place = t1;  
}  
L → id- - {  
    t1 = gentemp();  
    t2 = gentemp();  
    gencode(t1 = id.place);  
    gencode(t2 = id.place -1);  
    gencode(id.place = t2);  
    L.place = t1;  
}  
L → - -id {  
    t1 = gentemp();  
    gencode(t1 = id.place -1);  
    gencode(id.place = t1);  
    L.place = t1;  
}
```

## 6.9 THE ARRAY REFERENCE

An array reference is an expression with an *l*-value. Therefore, to capture its syntactic structure, we add the following productions to the grammar:

$$\begin{aligned} L &\rightarrow \text{id}[elist] \\ elist &\rightarrow \text{elist} / E \mid E \end{aligned}$$

An array reference in a source program is replaced by the *l*-value of an expression that specifies the arrayreference to an element of the array. Computing the *l*-value involves finding the offset of the referred element of the array and then adding it to the base. But since deriving an offset depends on the subscripts used in an array reference, and the values of these subscripts are not known during the compilation, unless the subscripts are constant expressions, a compiler has to generate the code for evaluating the *l*-value of an expression that specifies the reference to an element of an array. This *l*-value computation is achieved as follows:

$$\begin{aligned} \text{L-value}(a[i_1, i_2, i_3, \dots, i_k]) &= \text{addr}(a) + \text{offset} \\ \text{Offset} &= [(i_1 - lb_1)(ub_2 - lb_2+1)(ub_3 - lb_3+1) \\ &\quad \dots(ub_k - lb_k+1) + (i_2 - lb_2)(ub_3 - lb_3+1) \\ &\quad (ub_4 - lb_4+1)\dots(ub_k - lb_k+1) + \dots + (i_k - lb_k)] * \\ &\quad \text{size of element} \end{aligned}$$

where  $lb_i$  and  $ub_i$  are the lower and upper bounds of the *i*th dimension.

If the lower bound of each dimension is one, and the upper bound of the *i*th dimension is  $d_i$ , then the offset computing formula becomes:

$$\begin{aligned} \text{Offset} &= [(i_1 - 1)*d_2*d_3* \\ &\quad \dots *d_k + (i_2 - 1)*d_3*d_4* \\ &\quad \dots *d_k + \dots + (i_k - 1)) * bpw \\ \text{Offset} &= [i_1*d_2*d_3*\dots*d_k + i_2*d_3*d_4* \\ &\quad \dots *d_k + \dots + i_k) * bpw \\ &\quad [d_2*d_3* \\ &\quad \dots *d_k + d_3*d_4*\dots*d_k + \\ &\quad \dots + d_k] * bpw \end{aligned}$$

The  $[i_1*d_2*d_3*\dots*d_k + i_2*d_3*d_4*\dots*d_k + \dots + i_k] * bpw$  is a variable part of the offset computation, whereas  $[d_2*d_3*\dots*d_k + d_3*d_4*\dots*d_k + \dots + d_k] * bpw$  is a constant part of the offset computation and is not required to be computed for every reference to an array *a*. It can be computed once while processing the declaration of the array *a*. We call this value "constant C". Therefore:

$$\text{Offset} = V - C$$

where *V* is the variable part, and

$$\text{L-value}(a[i_1, i_2, i_3, \dots, i_k]) = \text{addr}(a) + V - C$$

Since  $\text{addr}(a)$  is fixed, we can combine *C* with  $\text{addr}(a)$  and store this value in an attribute, *L.place*, and we can store *V* in another attribute, *L.off*, so that:

$$i\text{-value}(a[i_1, i_2, i_3, \dots, i_k]) = L.\text{place}[L.\text{off}]$$

Hence, the translation of an array reference involves generating code for computing  $V$ , and  $V$  is made a value of attribute  $L.\text{off}$ . We compute  $\text{addr}(a) - C$  and make it the value of the attribute  $L.\text{place}$ . Computing  $V$  involves evaluating the expression:

$$[i_1] * d_2 * d_3 * \dots * d_k + i_2 * d_3 * d_4 * \dots * d_k + \dots + i_k * bpw$$

This expression can be rewritten as:

$$(((i_1)d_2 + i_2)d_3 + i_3)d_4 + \dots + 1) * bpw$$

Therefore, the three-address code that is required to be generated for computing  $V$  is:

$$\begin{aligned} t1 &= i_1 \\ t1 &= t1 * d_2 \\ t1 &= t1 + i_2 \\ t1 &= t1 * d_3 \\ t1 &= t1 + i_3 \\ &\vdots \\ t1 &= t1 * d_k \\ t1 &= t1 + i_k \\ V &= t1 * bpw \end{aligned}$$

Therefore, the translation scheme is:

```

elist → E      (Initialize queue by adding E.place)
elist → elist1, E (Append E.place to queue)
L → id[elist] { T1 := gentemp()
                 elist.Ndim = 1
                 gencode(T1 = retrieve());
                 while (queue not empty) do
                 {
                     gencode(T1 = T1 * limit(id.place, elist.Ndim))

                     gencode(T1 := T1 + retrieve())
                     elist.Ndim = elist.Ndim + 1
                 }
                 V = gentemp();
                 U = gentemp();
                 gencode(V := T1 * bpw)
                 gencode(U := id.place - C)
                 L.off = V
                 L.place := U
             }

```

where `retrieve()` is a function that retrieves a value from the queue, and `limit()` returns the upper bound of the dimension of the array.

In this translation scheme, the attribute `id.place` cannot be accessed in the semantic action associated with the production `elist → E` or in the semantic action associated with the production `elist → elist I, E`. So it is not possible to make use of the value of the subscript that is available in `E.place` to get the required three-address statements generated. Hence, a queue is necessary in order to maintain the subscripts' storage. These subscripts are used later on for generating the code for computing the offset.

Another way to approach this is to modify the grammar to make it suitable for translation. This requires rewriting the

productions in such a manner that both id and  $E$  exist in the same production so that the pointer to the symbol table record of the array name is available in id.place. This can be used to retrieve the upper-bound dimension information of the array. And the value of the subscript is available  $E.place$ ; so by using both of these, the required three-address statements can be generated, and the value of the subscript does not need to be stored. Therefore, the modified grammar, along with the semantic actions, is:

```

L → elist {      U = newtemp(); V = newtemp()
                V = elist.place * bpw
                U = gencode (elist.array - C)
                L.place = U
                L.off = V
            }
elist → id      E {elist.place = E.place
                    elist.array = id.place
                    elist.Ndim = 1; }

elist → elist, E {   T1 = newtemp ();
                    gencode (T1 = elist.place *
                            limit (elist.array, elist.Ndim +1))
                    gencode (T1 = T1 + E.place)
                    elist.array = elist1.array
                    elist.place = T1,
                    elist.Ndim = elist.Ndim +1
            }

```

For example, consider the following assignment statement:

$$c[a[i]] = b[i, j] + c[a[i, j]] + d[i + j]$$

where  $a$  and  $b$  are arrays of size  $30 \times 40$ , and  $c$  and  $d$  are arrays of size 20.

There are four bytes per word, and the arrays are allocated statically. When the above translation scheme is used to translate this construct, the three-address code generated is:

```

t1 = i * 40
t1 = t1 + j
t1 = t1 * 4
t1 = addr(a) - 164
t3 = t2[t1]
t3 = t3 * 4
t4 = addr(c) - 4
t5 = i * 40
t5 = t5 + j
t5 = t5 * 4
t6 = addr(b) - 164
t7 = t6[t5]
t8 = i * 40
t8 = t8 + j
t8 = t8 * 4
t9 = addr(a) - 164
t10 = t9[t8]
t10 = t10 * 4
t11 = addr(c) - 4
t12 = t11[t10]
t13 = i + j
t14 = addr(d) - 4
t15 = t14[t13]

```

---  
 $t16 = t7 + t12$   
 $t16 = t16 + t15$   
 $t4[t3] = t16$

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## 6.10 SWITCH/CASE

To capture the syntactic structure of the switch statement, we add the following productions to the grammar. Here, break is assumed to be a part of statement that is derivable from a nonterminal S.

```
S → switch E { caselist}
caselist → caselist case V: S
caselist → case V: S
caselist → default: S
caselist → caselist default: S
```

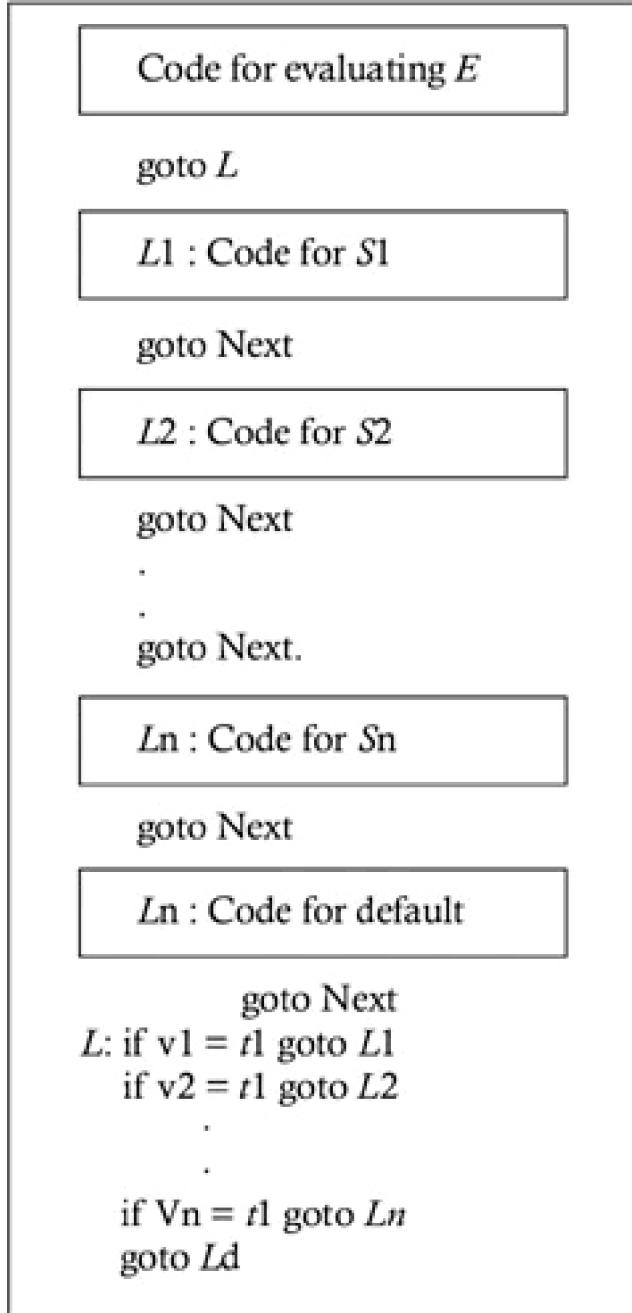
A switch statement is comprised of two components: an expression  $E$ , which is used to select a particular case from the list of cases; and a caselist, which is a list of  $n$  number of cases, each of which corresponds to one of the possible values of the expression  $E$ , perhaps including a default case.

**Note** A case statement can be implemented in a variety of different ways. If the number of cases is not too great, then a case statement can be implemented by generating a sequence of conditional jumps, each of which tests for an individual value and transfers to the code for the corresponding statement. If the number of cases is large, then it is more efficient to construct a hash table for the case values with the labels of the various statements as entries.

A syntax-directed translation scheme that translates a case statement into a sequence of conditional jumps, each of which tests for an individual value and transfers to the code for the corresponding statement, is considered below. We begin with a typical switch statement:

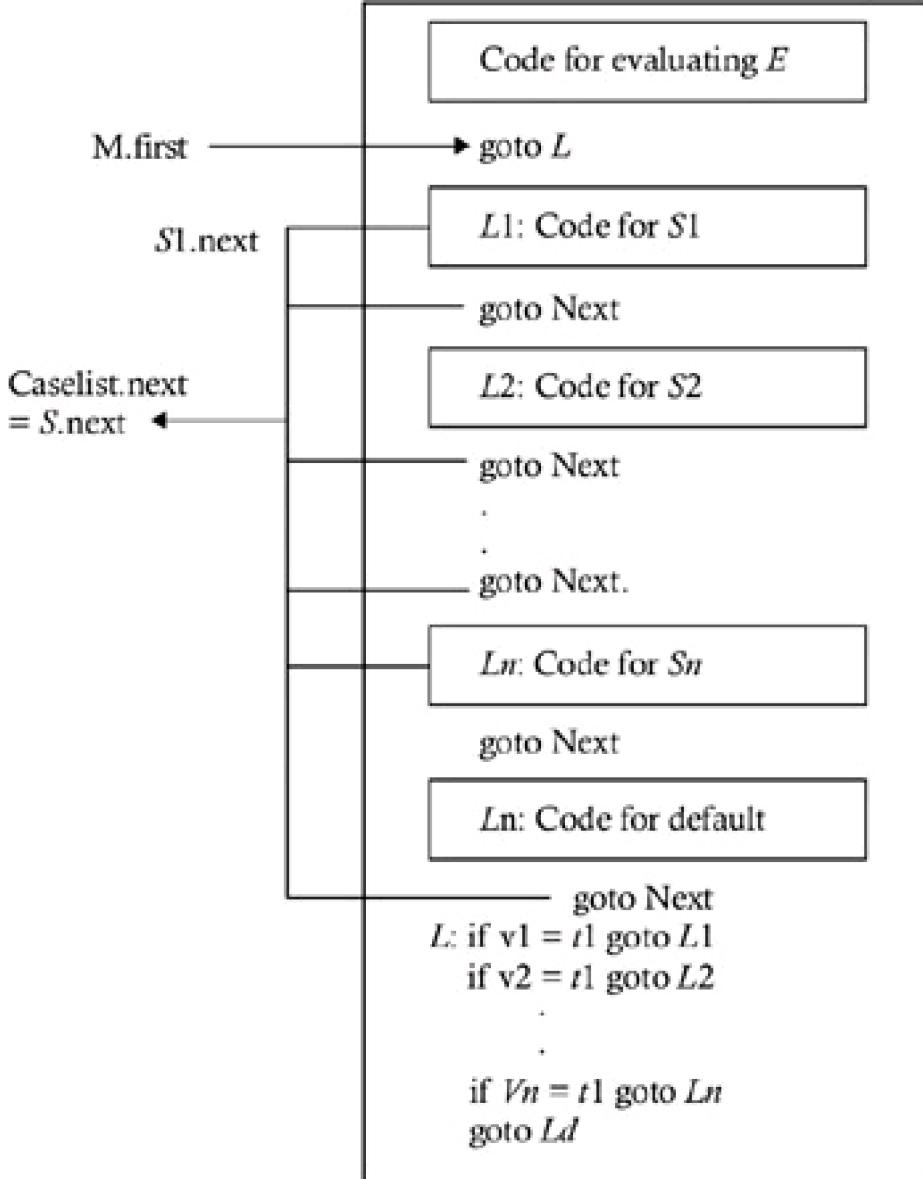
```
switch (E)
{
    case V1: S1
    case V2: S2
    .
    .
    .
    case Vn:Sn
}
```

The generated three-address that is required for the statement is shown in [Figure 6.17](#). Here, next is the label of the code for the statement that comes next in the switch statement execution order.



**Figure 6.17:** A switch/case three-address translation.

Therefore, switch statement translation involves generating an unconditional jump after the code of every  $S_1, S_2, \dots, S_n$  statement in order to transfer control to the next element of the switch statement, as well as to remember the code start of  $S_1, S_2, \dots, S_n$ , and to generate the conditional jumps. Each of these jumps tests for an individual value and transfers to the code for the corresponding statement. This requires introducing nullable nonterminals before  $S_1$ , as shown in [Figure 6.18](#).



**Figure 6.18:** Nullable nonterminals are introduced into a switch statement translation.

#### EXAMPLE 6.1

Consider the following switch statement:

```

switch (i + j)
{
  case 1: x = y + z
  default: p = q + r
  case 2: u = v + w
}
  
```

The above translation scheme translates into the following three-address code, which is also shown in [Figure 6.19](#):

i)  $t1 = i + j$   
 i + 1) goto( $i + 11$ )  
 i + 2)  $t1 = y + z$   
 i + 3)  $x = t1$   
 i + 4) goto\_  
 i + 5)  $t2 = q+r$   
 i + 6)  $p = t2$   
 i + 7) goto\_  
 i + 8)  $t3 = u+w$   
 i + 9)  $u = t3$   
 i + 10) goto\_  
 i + 11) if  $t1 = 1$  goto( $i+2$ )  
 i + 12) if  $t1 = 2$  goto( $i+8$ )  
 i + 13) goto( $i + 5$ )



Figure 6.19: Contents of queue during the translation.

#### EXAMPLE 6.2

---

Using the above translation scheme translates the following switch statement:

```

switch (a+b)
{
  case 2: { x = y; break; }
  case 5: {switch x
  {
    case 0: { a = b + 1; break; }
    case 1: { a = b + 3; break; }
    default: { a = 2; break; }
  }
  break;
  case 9: { x = y - 1; break; }
  default: { a = 2; break; }
}
  
```

---

The three address code is:

1.  $t1 = a + b$
2. goto(23)
3.  $x = y$
4. goto NEXT
5. goto(14)
6.  $t3 = b + 1$

7.  $a = t3$   
8. goto NEXT  
9.  $t4 = b + 3$   
10.  $a = t4$   
11. goto NEXT  
12.  $a = 2$   
13. goto NEXT  
14. if  $x = 0$  goto(6)  
15. if  $x = 1$  goto(9)  
16. goto(12)  
17. goto NEXT  
18.  $t5 = y - 1$   
19.  $x = t5$   
20. goto NEXT  
21.  $a = 2$   
22. goto NEXT  
23. if  $t1 = 2$  goto(3)  
24. if  $t1 = 5$  goto(5)  
25. if  $t1 = 9$  goto(18)  
26. goto(21)

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## 6.11 THE PROCEDURE CALL

```
S → call id (arglist)
{   for every value T in queue generate
    Param T gencode
    (call id.place, arglist.count)
}
arglist → arglist, E{ append (queue, Eplace)
                    arglist.count:= arglist. count + 1}
arglist → E{ initialize queue by Eplace
            arglist.count:= 1}
```

## 6.12 EXAMPLES

Following are additional examples of syntax-directed definitions and translations.

### EXAMPLE 6.3

Generate the three-address code for the following C program:

```
main()
{ int i = 1;
  int a[10];
  while(i <= 10)
    a[i] = ;
}
```

The three-address code for the above C program is:

1.  $i = 1$
2. if  $i \leq 10$  goto(4)
3. goto(8)
4.  $t1 = i * \text{width}$
5.  $t2 = \text{addr}(a) - \text{width}$
6.  $t2[t1] = 0$
7. goto(2)

where width is the number of bytes required for each element.

### EXAMPLE 6.4

Generate the three-address code for the following program fragment:

```
while (A < C and B > D) do
  if A = 1 then C = C+1
  else
    while A <= D do
      A = A + 3
```

The three-address code is:

1. if  $a < c$  goto(3)
2. goto(16)
3. if  $b > d$  goto(5)
4. goto(16)
5. if  $a = 1$  goto(7)
6. goto(10)
7.  $t1 = c+1$

8.  $c = t1$
9. goto(1)
10. if  $a \leq d$  goto
11. goto(1)
12.  $t2 = a + 3$
13.  $a = t2$
14. goto(10)
15. goto(1)

#### **EXAMPLE 6.5**

---

Generate the three-address code for the following program fragment, where  $a$  and  $b$  are arrays of size  $20 \times 20$ , and there are four bytes per word.

---

```

begin
add = 0;
i = 1;
j = 1;
do
begin
add = add + a[i,j] * b[j,i]
i = i + 1;
j = j + 1;
end
while i <= 20 and j <= 20;
end

```

The three-address code is:

1.  $add = 0$
2.  $i = 1$
3.  $j = 1$
4.  $t1 = i * 20$
5.  $t1 = t1 + j$
6.  $t1 = t1 * 4$
7.  $t2 = \text{addr}(a) - 84$
8.  $t3 = t2[t1]$
9.  $t4 = j * 20$
10.  $t4 = t4 + i$
11.  $t4 = t4 * 4$
12.  $t5 = \text{addr}(b) - 84$
13.  $t6 = t5[t4]$
14.  $t7 = t3 * t6$
15.  $t7 = add + t7$

16.  $t8 = i + 1$
17.  $i = t8$
18.  $t9 = j + 1$
19.  $j = t9$
20. if  $i \leq 20$  goto(22)
21. goto NEXT
22. if  $j \leq 20$  goto(4)
23. goto NEXT

#### **EXAMPLE 6.6**

---

Consider the program fragment:

```
sum = 0
for(i = 1; i<= 20; i++)
    sum = sum + a[i] + b[i];
```

and generate the three-address code for it. There are four bytes per word.

---

The three address code is:

1.  $\text{sum} = 0$
2.  $i = 1$
3. if  $i \leq 20$  goto(8)
4. goto NEXT
5.  $t1 = i + 1$
6.  $i = t1$
7. goto(3)
8.  $t2 = i * 4$
9.  $t3 = \text{addr}(a) - 4$
10.  $t4 = t3[t2]$
11.  $t5 = i * 4$
12.  $t6 = \text{addr}(b) - 4$
13.  $t7 = t6[t5]$
14.  $t8 = \text{sum} + t4$
15.  $t8 = t8 + t7$
16.  $\text{sum} = t8$
17. goto(5)

## Chapter 7: Symbol Table Management

### 7.1 THE SYMBOL TABLE

A symbol table is a data structure used by a compiler to keep track of scope/ binding information about names. This information is used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements. The symbol table is searched every time a name is encountered in the source text. When a new name or new information about an existing name is discovered, the content of the symbol table changes. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table.

For efficiency, our choice of the implementation data structure for the symbol table and the organization its contents should be stress a minimal cost when adding new entries or accessing the information on existing entries. Also, if the symbol table can grow dynamically as necessary, then it is more useful for a compiler.

## 7.2 IMPLEMENTATION

Each entry in a symbol table can be implemented as a record that consists of several fields. These fields are dependent on the information to be saved about the name. But since the information about a name depends on the usage of the name (i.e., on the program element identified by the name), the entries in the symbol table records will not be uniform. Hence, to keep the symbol table records uniform, some of the information about the name is kept outside of the symbol table record, and a pointer to this information is stored in the symbol table record, as shown in [Figure 7.1](#). Here, the information about the lower and upper bounds of the dimension of the array named *a* is kept outside of the symbol table record, and the pointer to this information is stored within the symbol table record.

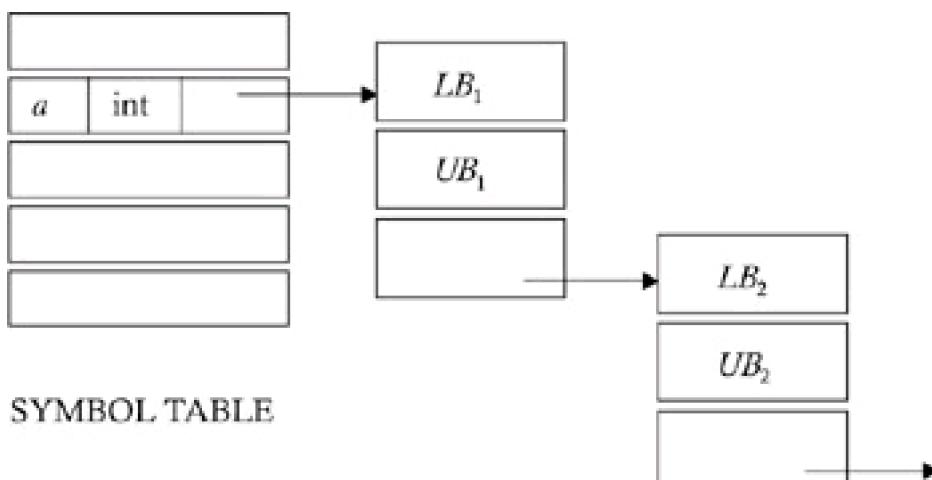


Figure 7.1: A pointer steers the symbol table to remotely stored information for the array *a*.

## 7.3 ENTERING INFORMATION INTO THE SYMBOL TABLE

Information is entered into the symbol table in various ways. In some cases, the symbol table record is created by the lexical analyzer as soon as the name is encountered in the input, and the attributes of the name are entered when the declarations are processed. But very often, the same name is used to denote different objects, perhaps even in the same block. For example, in C programming, the same name can be used as a variable name and as a member name of a structure, both in the same block. In such cases, the lexical analyzer only returns the name to the parser, rather than a pointer to the symbol table record. That is, a symbol table record is not created by the lexical analyzer; the string itself is returned to the parser, and the symbol table record is created when the name's syntactic role is discovered.

## 7.4 WHERE SHOULD NAMES BE HELD?

If there is a modest upper bound on the length of the name, then the name can be stored in the symbol table record itself. But if there is no such limit, or if the limit is rarely reached, then an indirect scheme of storing name is used. A separate array of characters, called a "string table," is used to store the name, and a pointer to the name is kept in the symbol table record, as shown in [Figure 7.2](#).

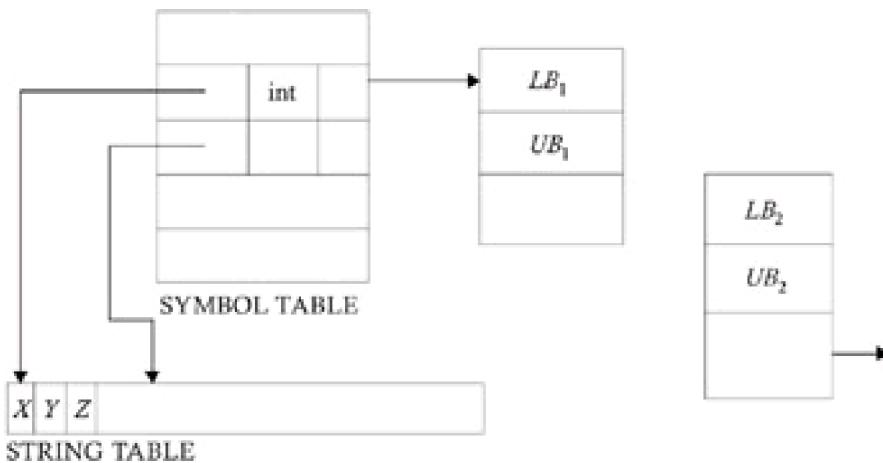


Figure 7.2: Symbol table names are held either in the symbol table record or in a separate string table.

## 7.5 INFORMATION ABOUT THE RUNTIME STORAGE LOCATION

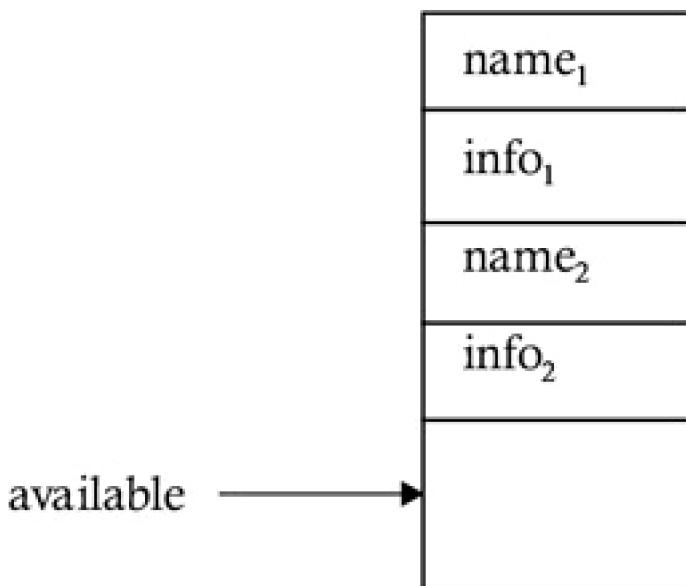
The information about the runtime, name storage location is kept in the symbol table. If the compiler is going to be generating assembly code, then the assembler takes care of the storage locations of the various names. After generating the assembly code, the compiler scans the symbol table and generates the assembly language data definitions. These are appended to the assembly language code for each name. But if machine code is being generated, then the compiler must ascertain the position of each data object relative to a fixed origin.

## 7.6 VARIOUS APPROACHES TO SYMBOL TABLE ORGANIZATION

There are several methods of organizing the symbol table. These methods are discussed below.

### 7.6.1 The Linear List

A linear list of records is the easiest way to implement a symbol table. The new names are added to the table in the order that they arrive. Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether or not the name is already present in the table. If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer, as shown in the [Figure 7.3](#).



**Figure 7.3:** A new record is added to the linear list of records.

To retrieve the information about the name, the table is searched sequentially, starting from the first record in the table. The average number of comparisons,  $p$ , required for search are  $p = (n + 1)/2$  for successful search and  $p = n$  for an unsuccessful search, where  $n$  is the number of records in symbol table. The advantage of this organization is that it takes less space, and additions to the table are simple. This method's disadvantage is that it has a higher accessing time.

### 7.6.2 Search Trees

A search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record, and these links point to the record in the search tree. Whenever a name is to be added, first the name is searched in the tree. If it does not exist, then a record for the new name is created and added at the proper position in the search tree. This organization has the property of alphabetical accessibility; that is, all the names accessible from name<sub>1</sub> will, by following a left link, precede name<sub>1</sub> in alphabetical order. Similarly, all the names accessible from name<sub>1</sub> will follow name<sub>1</sub> in alphabetical order by following the right link (see [Figure 7.4](#)). The expected time needed to enter  $n$  names and to make  $m$  queries is proportional to  $(m + n) \log_2 n$ ; so for greater numbers of records (higher  $n$ ) this method has advantages over linear list organization.

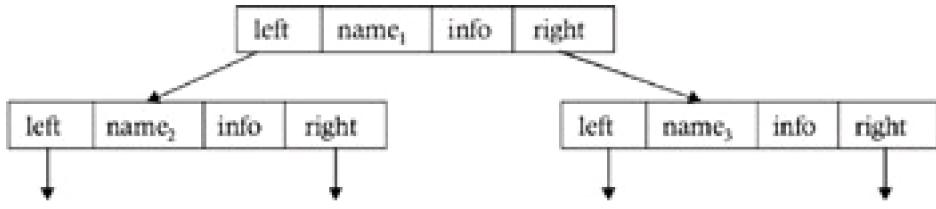


Figure 7.4: The search tree organization approach to a symbol table.

### 7.6.3 Hash Tables

A hash table is a table of  $k$  pointers numbered from zero to  $k-1$  that point to the symbol table and a record within the symbol table. To enter a name into symbol table, we find out the hash value of the name by applying a suitable hash function. The hash function maps the name into an integer between zero and  $k-1$ , and using this value as an index in the hash table, we search the list of the symbol table records that is built on that hash index. If the name is not present in that list, we create a record for name and insert it at the head of the list. When retrieving the information associated with the name, the hash value of the name is first obtained, and then the list that was built on this hash value is searched for information about the name (Figure 7.5).

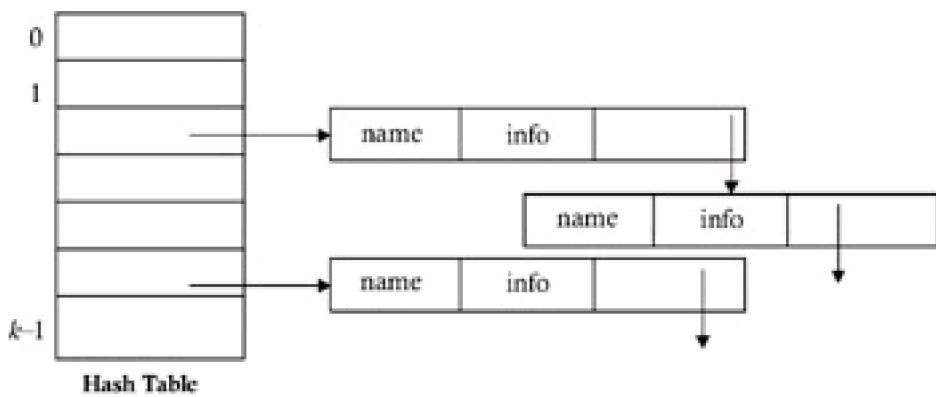


Figure 7.5: Hash table method of symbol table organization.

## 7.7 REPRESENTING THE SCOPE INFORMATION IN THE SYMBOL TABLE

Every name possesses a region of validity within the source program, called the "scope" of that name. The rules governing the scope of names in a block-structured language are as follows:

1. A name declared within a block  $B$  is valid only within  $B$ .
2. If block  $B_1$  is nested within  $B_2$ , then any name that is valid for  $B_2$  is also valid for  $B_1$ , unless the identifier for that name is re-declared in  $B_1$ .

These scope rules require a more complicated symbol table organization than simply a list of associations between names and attributes. One technique that can be used is to keep multiple symbol tables, one for each active block, such as the block that the compiler is currently in. Each table is list of names and their associated attributes, and the tables are organized into a stack. Whenever a new block is entered, a new empty table is pushed onto the stack for holding the names that are declared as local to this block. And when a declaration is compiled, the table on the stack is searched for a name. If the name is not found, then the new name is inserted. When a reference to a name is translated, each table is searched, starting from the top table on the stack, ensuring compliance with static scope rules. For example, consider following program structure. The symbol table organization will be as shown in [Figure 7.6](#).

```
Program main
Var x,y : integer :
Procedure P :
Var x,a : boolean;
Procedure q
Var x,y,z : real;
Begin
.
.
end
begin :
end
begin :
end
```

Symbol table for block  $q$

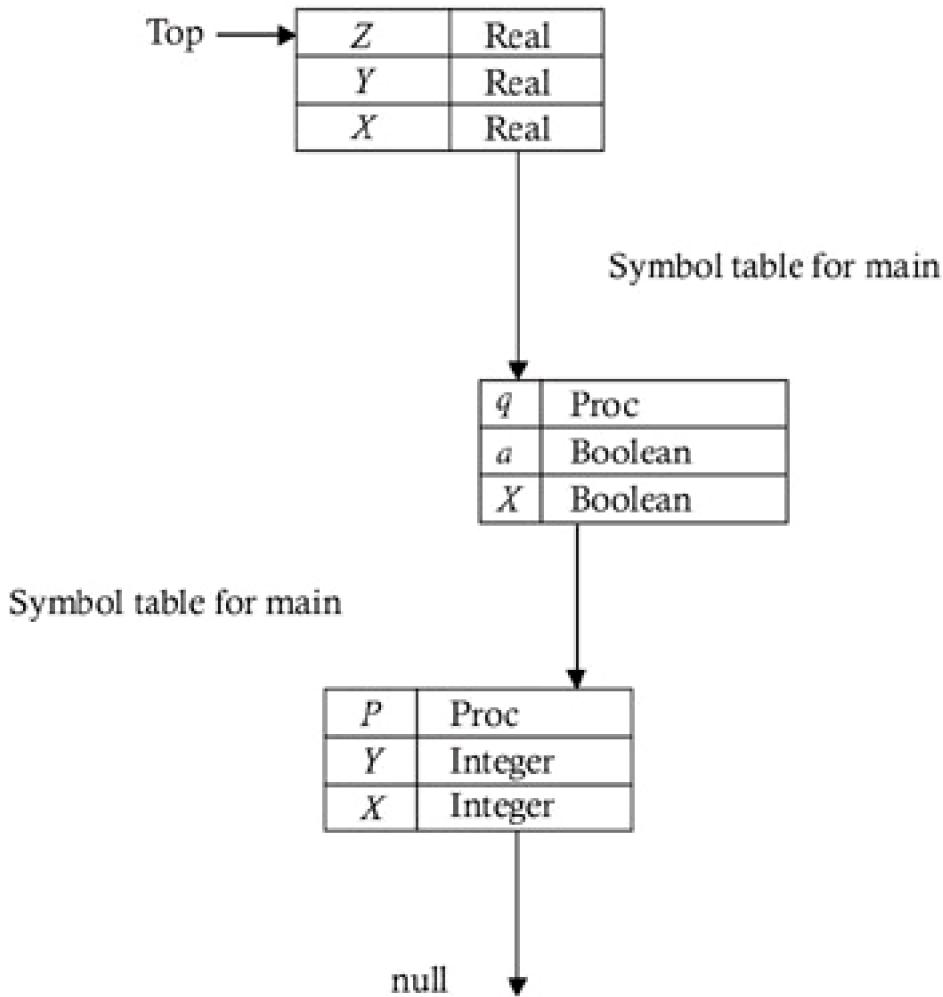


Figure 7.6: Symbol table organization that complies with static scope information rules.

Another technique can be used to represent scope information in the symbol table. We store the nesting depth of each procedure block in the symbol table and use the [procedure name, nesting depth] pair as the key to accessing the information from the table. A nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure. This number is basically a count of how many procedures are there in the referencing environment of the procedure.

For example, refer to the program code structure above. The symbol table's contents are shown in [Table 7.1](#).

**Table 7.1: Symbol Table Contents Using a Nesting Depth Approach**

X	1	real
Y	1	real
Z	1	real
q	3	proc
a	3	Boolean
X	3	Boolean
P	2	proc
Y	2	integer
X	2	integer

**Team LiB**

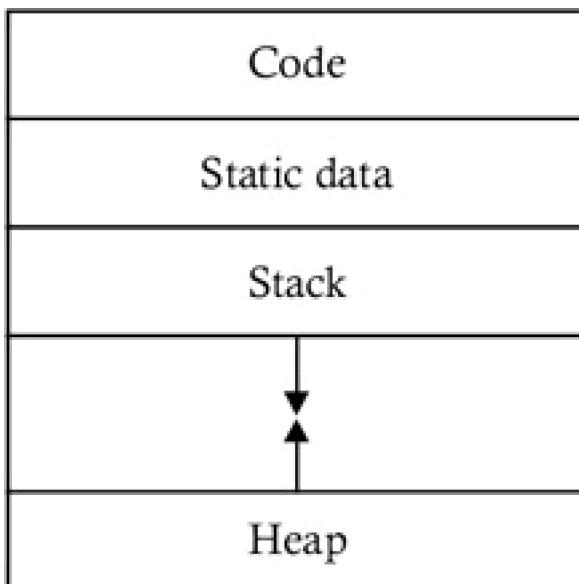
**◀ PREVIOUS** **NEXT ▶**

## Chapter 8: Storage Management

### 8.1 STORAGE ALLOCATION

One of the important tasks that a compiler must perform is to allocate the resources of the target machine to represent the data objects that are being manipulated by the source program. That is, a compiler must decide the run-time representation of the data objects in the source program. Source program run-time representations of the data objects, such as integers and real variables, usually take the form of equivalent data objects at the machine level; whereas data structures, such as arrays and strings, are represented by several words of machine memory.

The strategies that can be used to allocate storage to the data objects are determined by the rules defining the scope and duration of the names in the programming language. The simplest strategy is static allocation, which is used in languages like FORTRAN. With static allocation, it is possible to determine the run-time size and relative position of each data object during compilation. A more-complex strategy for dynamic memory allocation that involves stacks is required for languages that support recursion: an entry to a new block or procedure causes the allocation of space on a stack, which is freed on exit from the block or procedure. An even more-complex strategy is required for languages, which allows the allocation and freeing of memory for some data in a non-nested fashion. This storage space can be allocated and freed arbitrarily from an area called a "heap". Therefore, implementation of languages like PASCAL and C allow data to be allocated under program control. The run-time organization of the memory will be as shown in [Figure 8.1](#).



**Figure 8.1:** Heap memory storage allows program-controlled data allocation.

The run-time storage has been subdivided to hold the generated target code and the data objects, which are allocated statically for the stack and heap. The sizes of the stack and heap can change as the program executes.

## 8.2 ACTIVATION OF THE PROCEDURE AND THE ACTIVATION RECORD

Each execution of a procedure is referred to as an activation of the procedure. This is different from the procedure definition, which in its simplest form is the association of an identifier with a statement; the identifier is the name of the procedure, and the statement is the body of the procedure.

If a procedure is non-recursive, then there exists only one activation of procedure at any one time. Whereas if a procedure is recursive, several activations of that procedure may be active at the same time. The information needed by a single execution or a single activation of a procedure is managed using a contiguous block of storage called an "activation record" or *activation frame* consisting of the collection of fields. (Very often, registers take the place of one or more of the fields in the activation record.) The activation record contains the following information:

1. Temporary values, such as those arising during the evaluation of the expression.
2. Local data of a procedure.
3. The information about the machine state (i.e., the machine status) just before a procedure is called, including PC values and the values of these registers that must be restored when control is relinquished after the procedure.
4. Access links (optional) referring to non-local data that is held in other activation records. This is not required for a language like FORTRAN, because non-local data is kept in fixed place. But it is required for Pascal.
5. Actual parameters (i.e., the parameters supplied to the called procedure). These parameters may also be passed in machine registers for greater efficiency.
6. The return value used by called procedure to return a value to calling procedure. Again, for greater efficiency, a machine register may be used for returning values.

The size of almost all of the fields of the activation record can be determined at compile time. An exception is if a called procedure has a local array whose size is determined by the values of the actual parameters.

The information in the activation record is organized in a manner that enables easy access at execution time. A pointer to the activation record is required. This pointer is called the current environment pointer (CEP), and it points to one of the fixed fields in the activation record. Using the proper offset from this pointer, and depending upon the format of the activation record, the contents of the activation record can be accessed. [Figure 8.2](#) shows the organization of information in a typical activation record.

Return value
Actual parameters
Control links
Access links
Machine status
Local data
Temporaries



Direction of growth

Figure 8.2: Typical format of an activation record.

## 8.3 STATIC ALLOCATION

In static allocation, the names are bound to specific storage locations as the program is compiled. These storage locations cannot be changed during the program's execution. Since the binding does not change at run time, every time a procedure is called, its names are bound to the same storage locations. Hence, if the local names are allocated statically, then their values will be retained throughout the activation of a procedure. The compiler uses the name type to determine the amount of storage to set aside for that name. The address of this storage consists of an offset from an end-of-activation record for the procedure. The compiler must decide where the activation records go relative to the target code and relative to other activation records. Once this decision is made, the storage position for each name in the record is fixed. Therefore, at compile time, it is possible to fill in both the address at which the target code can find the data and the address at which information is saved. However, there are some limitations to using static allocation:

1. The size of the data object and any constraints on its position in memory must be known at compile time.
2. Recursive procedures cannot be permitted, because all activations of a procedure use the same binding for local names.
3. Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

## 8.4 STACK ALLOCATION

In stack allocation, storage is organized as a stack, and activation records are pushed and popped as the activation of procedures begin and end, respectively, thereby permitting recursive procedures. The storage for the locals in each procedure call is contained in the activation record for that call. Hence, the locals are bound to fresh storage in each activation, because a new activation record is pushed onto stack when a call is made. The storage values of locals are deleted when the activation ends.

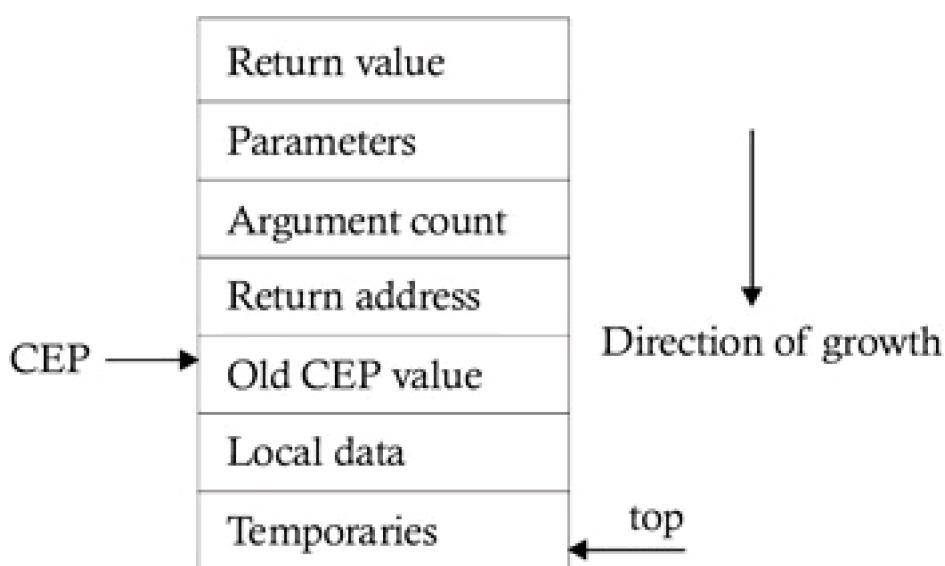
### 8.4.1 The Call and Return Sequence

Procedure calls are implemented by generating what is called a "call sequence and return sequence" in the target code. The job of a call sequence is to set up an activation record. Setting up an activation record means entering the information into the fields of the activation record if the storage for the activation record is allocated statically. When the storage for the activation record is allocated dynamically, storage is allocated for it on the stack, and the information is entered in its fields.

On the other hand, the job of a return sequence is to restore the state of machine so that the machine's calling procedure can continue executing. This also involves destroying the activation record if it was allocated dynamically on the stack.

The code in a call sequence is often divided between the caller and the callee. But there is no exact division of run-time tasks between the caller and callee. It depends on the source language, the target machine, and the operating system. Hence, even when using a common language, the call sequence may differ from implementation to implementation. But it is desirable to put as much of the calling sequence into the callee as possible, because there may be several calls for a procedure. And even though that portion of the calling sequence is generated for each call by the various callers, this portion of the calling sequence is shared within the callee, so it is generated only once.

[Figure 8.3](#) shows the format of a typical activation record. Here, the contents of the activation record are accessed using the CEP pointer.



**Figure 8.3:** The CEP pointer is used to access the contents of the activation record.

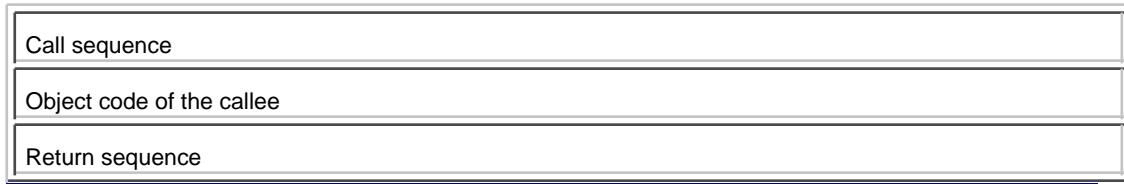
The stack is assumed to be growing from higher to lower addresses. A positive offset will be used to access the contents of the activation record when we want to go in a direction opposite to that of the growth of the stack (in [Figure 8.3](#), the field pointed to by the CEP). A negative offset will be used to access the contents of the activation record when we want to go in the same direction as the growth of stack. A typical call sequence for caller code to evaluate parameters is as follows:

```

push ( ) /* for return value
push (T1) /* T1 is holding the first argument
push (T2) /* T2 is holding the second argument
.
.
.
push (Tn) /* Tn is holding the nth argument
push (n) /* n is the count of arguments
push (return address)
push (CEP)
goto start of code segment of callee

```

A typical callee code segment is shown in [Figure 8.4](#).



**Figure 8.4:** Typical callee code segment.

A typical call sequence in the callee will be:

CEP = top /\*

Code for pushing the local data of  
the callee

And a typical return sequence is:

```

top = CEP + 1
1 = *top /* for retrieving return address
top = top + 1
CEP = *CEP / for resetting the CEP to point to the activation record of the caller
top = top + *top + 2 /* for resetting top to point to the top of the activation record of caller goto1

```

#### 8.4.2 Access to Nonlocal Names

The way that the nonlocals are accessed depends on the scope rules of the language (see [Chapter 7](#)). There are two different types of scope rules: static scope rules and dynamic scope rules.

Static scope rules determine which declaration a name's reference will be associated with, depending upon the program's language, thereby determining from where the name's value will be obtained at run time. When static scope rules are used during compilation, the compiler knows how the declarations are bound to the name references, and hence, from where their values will be obtained at run time. What the compiler has to do is to provision the retrieval of the nonlocal name value when it is accessed at run time.

Whereas when dynamic scope rules are used, the values of nonlocal names are retrieved at run time by scanning down the stack, starting at the top-most activation record. The rule for associating a nonlocal reference to a declaration is simple when procedure nesting is not permitted. In the absence of nested procedures, the storage for all names declared outside any procedure can be allocated statically. The position of this storage is known at compile time, so if a name is nonlocal in some procedure's body, its statically determined address is used; whereas if a name is local, it is assessed via a CEP pointer using the suitable offset.

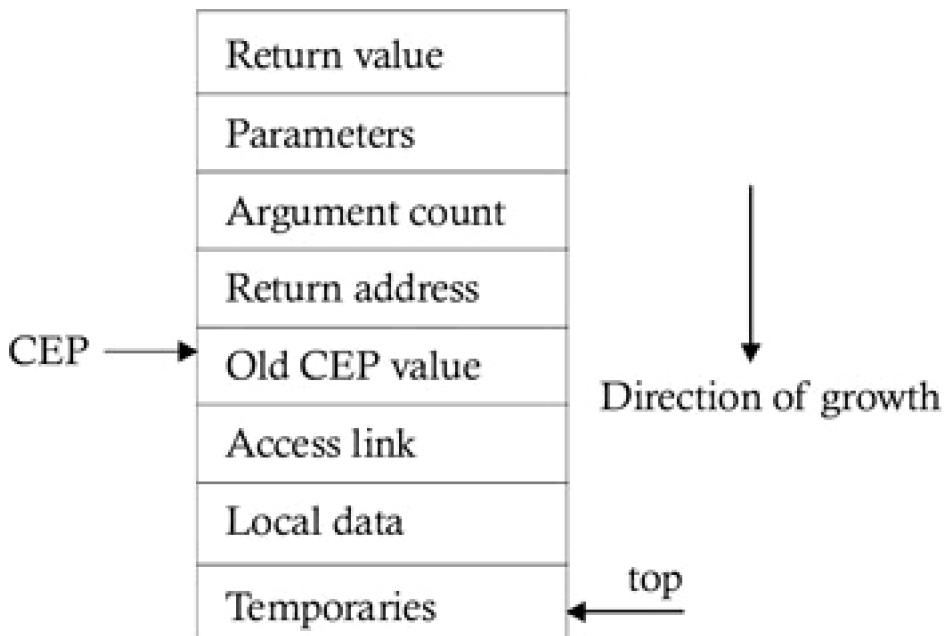
An important benefit of static allocation for nonlocals is that declared procedures can be freely passed as parameters and returned as results. For example, a function *inCis* passed by address; that is, a pointer is passed to it. When the procedures are nested, declarations are bound to name references according to the following rule: if a name *x* is not declared in a procedure *P*, then an occurrence of *x* in *P* is in the scope of a declaration of *x* in an enclosing procedure

$P_1$  such that:

1. The enclosing procedure  $P_1$  has a declaration of  $x$ , and
2.  $P_1$  is more closely nested around  $P$  than any other procedure with a declaration of  $x$ .

Therefore, a reference to a nonlocal name  $x$  is resolved by associating it with the declaration of  $x$  in  $P_1$ , and the compiler is required to provision getting the value of  $x$  at run time from the most-recent activation record of  $P_1$  by generating a suitable call sequence.

One of the ways to implement this is to add a pointer, called an "access link," to each activation record. And if a procedure  $P$  is nested immediately within  $Q$  in the source text, then make the access link in the activation record  $P$ , pointing to the most-recent activation record of  $Q$ . This requires an activation record with a format like that shown in [Figure 8.5](#).

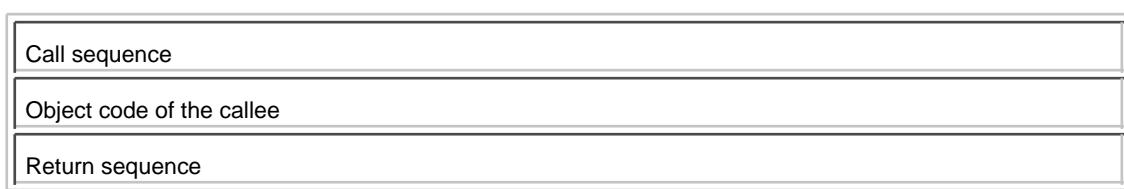


**Figure 8.5:** An activation record that deals with nonlocal name references.

The modified call and return sequence, required for setting up the activation record shown in [Figure 8.5](#), is:

```
push () /* for return value  
push (T1) /* T1 is holding the first argument  
push (T2) /* T2 is holding the second argument  
. . .  
push (Tn) /* Tn is holding the nth argument  
push(n) /* n is the count of arguments  
push (return address)  
push (CEP)  
code to set up access link  
goto start of code segment of callee
```

A typical callee segment is shown in [Figure 8.6](#).



**Figure 8.6:** A typical callee segment.

A typical call sequence in the callee is:

CEP = top+1/\* code for pushing the local data of the callee

A typical return sequence is:

top = CEP+1

1 = \*top /\* for retrieving return address

top = top+1

CEP = \*CEP / for resetting the CEP to point to the activation record of caller

top = top + \*top +2/\* for resetting top to point to the top of the activation record of caller goto 1

#### 8.4.3 Setting Up the Access Link

To generate the code for setting up the access link, a compiler makes use of the following information: the nesting depth of the caller procedure and the nesting depth of the callee procedure. A procedure's nesting depth is number that is obtained by starting with value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure. This number is basically a count of how many procedures are there in the referencing environment of the procedure.

Suppose that procedure  $p$  at a nesting depth  $N_p$  calls a procedure at nesting depth  $N_q$ . Then the access link in the activation record of procedure  $q$  is set up as follows:

if ( $N_q > N_p$ ) then

The access link in the activation record of procedure  $q$  is set to point to the activation record of procedure  $p$ .

else

if ( $N_q = N_p$ ) then

Copy the access link in the activation record of procedure  $p$  into the activation record of procedure  $q$ .

else

if ( $N_q < N_p$ ) then

Follow ( $N_p - N_q$ ) links to reach to the activation record, and copy the access link of this activation record into the activation record of procedure  $q$ .

### The Block Statement

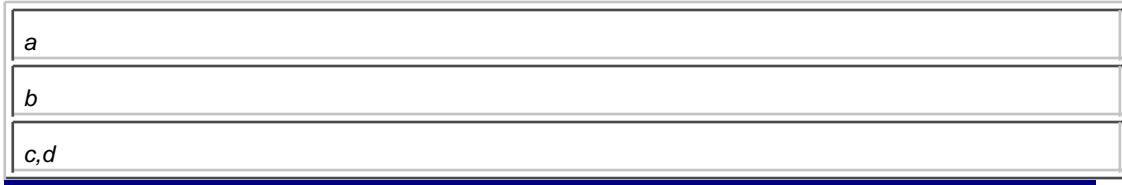
A block is a statement that contains its own local data declarations. Blocks can either be independent—like  $B_1$  begin and  $B_1$  end, then  $B_2$  begin and  $B_2$  end—or they can be nested—like  $B_1$  begin and  $B_2$  begin, then  $B_2$  end and  $B_1$  end. This nesting property is sometimes called a "block structure". The scope of a declaration in a block-structured language is given by the most closely nested rule:

1. The scope of a declaration in a block  $B$  includes  $B$ .
2. If a name  $X$  is not declared in a block  $B$ , then an occurrence of  $X$  in  $B$  is in the scope of a declaration of  $X$  in an enclosing block  $B'$ , such that:
  - a.  $B'$  has a declaration of  $X$ , and
  - b.  $B'$  is more closely nested around  $B$  than any other block with a declaration of  $X$ .

Block structure can be implemented using stack allocation. Space is allocated for declared names. The block is entered by pushing an activation record, and it is de-allocated when control leaves the block and the activation record is destroyed. That is, a block is treated like a parameter-less procedure, called only at the entry to the block and returned upon exit from the block. An alternative is to allocate storage for a complete procedure body at one time. If there are blocks within the procedure, then an allowance is made for the storage needed by the declarations within the block, as shown in [Figure 8.7](#). For example, consider the following program structure:

```
main ()  
{  
    int a;  
    {  
        int b;  
        {  
            int c;  
            printf ("%d %d\n", b,c);  
        }  
        {  
            int d;  
            printf("%d %d\n", b, d);  
        }  
    }  
    printf("%d\n",a);  
}
```

---



**Figure 8.7:** Storage for declared names.

## Chapter 9: Error Handling

### 9.1 ERROR RECOVERY

One of the important tasks that a compiler must perform is the detection of and recovery from errors. Recovery from errors is important, because the compiler will be scanning and compiling the entire program, perhaps in the presence of errors; so as many errors as possible need to be detected.

Every phase of a compilation expects the input to be in a particular format, and whenever that input is not in the required format, an error is returned. When detecting an error, a compiler scans some of the tokens that are ahead of the error's point of occurrence. The fewer the number of tokens that must be scanned ahead of the point of error occurrence, the better the compiler's error-detection capability. For example, consider the following statement:

if  $a = b$  then  $x = y + z;$

The error in the above statement will be detected in the syntactic analysis phase, but not before the syntax analyzer sees the token "then"; but the first token, itself, is in error.

After detecting an error, the first thing that a compiler is supposed to do is to report the error by producing a suitable diagnostic. A good error diagnostic should possess the following properties.

1. The message should be produced in terms of the original source program rather than in terms of some internal representation of the source program. For example, the message should be produced along with the line numbers of the source program.
2. The error message should be easy to understand by the user.
3. The error message should be specific and should localize the problem. For example, an error message should read, "x is not declared in function fun," and not just, "missing declaration".
4. The message should not be redundant; that is, the same message should not be produced again and again.

Therefore, a compiler should report errors by generating messages with the above properties. The errors captured by the compiler can be classified as either syntactic errors or semantic errors. Syntactic errors are those errors that are detected in the lexical or syntactic analysis phase by the compiler. Semantic errors are those errors detected by the compiler.

## 9.2 RECOVERY FROM LEXICAL PHASE ERRORS

The lexical analyzer detects an error when it discovers that an input's prefix does not fit the specification of any token class. After detecting an error, the lexical analyzer can invoke an error recovery routine. This can entail a variety of remedial actions.

The simplest possible error recovery is to skip the erroneous characters until the lexical analyzer finds another token. But this is likely to cause the parser to read a deletion error, which can cause severe difficulties in the syntax analysis and remaining phases. One way the parser can help the lexical analyzer can improve its ability to recover from errors is to make its list of legitimate tokens (in the current context) available to the error recovery routine. The error-recovery routine can then decide whether a remaining input's prefix matches one of these tokens closely enough to be treated as that token.

## 9.3 RECOVERY FROM SYNTACTIC PHASE ERRORS

A parser detects an error when it has no legal move from its current configuration. The  $LL(1)$  and  $LR(1)$  parsers use the valid prefix property; therefore, they are capable of announcing an error as soon as they read an input that is not a valid continuation of the previous input's prefix. This is earliest time that a left-to-right parser can announce an error. But there are a variety of other types of parsers that do not necessarily have this property.

The advantages of using a parser with a valid-prefix-property capability is that it reports an error as soon as possible, and it minimizes the amount of erroneous output passed to subsequent phases of the compiler.

### Panic Mode Recovery

Panic mode recovery is an error recovery method that can be used in any kind of parsing, because error recovery depends somewhat on the type of parsing technique used. In panic mode recovery, a parser discards input symbols until a statement delimiter, such as a semicolon or an end, is encountered. The parser then deletes stack entries until it finds an entry that will allow it to continue parsing, given the synchronizing token on the input. This method is simple to implement, and it never gets into an infinite loop.

## 9.4 ERROR RECOVERY IN LR PARSING

A systematic method for error recovery in *LR* parsing is to scan down the stack until a state *S* with a goto on a particular nonterminal *A* is found, and then discard zero or more input symbols until a symbol *a* is found that can legitimately follow *A*. The parser then shifts the state goto  $[S, A]$  on the stack and resumes normal parsing.

There might be more than one choice for the nonterminal *A*. Normally, these would be nonterminals representing major program pieces, such as statements.

Another method of error recovery that can be implemented is called "phrase level recovery". Each error entry in the *LR* parsing table is examined, and, based on language usage, an appropriate error-recovery procedure is constructed. For example, to recover from an construct error that starts with an operator, the error-recovery routine will push an imaginary id onto the stack and cover it with the appropriate state. While doing this, the error entries in a particular state that call for a particular reduction on some input symbols are replaced by that reduction. This has the effect of postponing the error detection until one or more reductions are made; but the error will still be caught before a shift.

A phrase level error-recovery implementation for an *LR* parser is shown below. The parsing table's grammar is:

$$E \rightarrow E + E \mid E^* E \mid \text{id}$$

The *SLR* parsing table for the above grammar is shown in [Table 9.1](#).

**Table 9.1: Parsing Table for  $E \rightarrow E + E \mid E^* E \mid \text{id}$**

	id	+	*	\$	E
<i>l</i> <sub>0</sub>	<i>S</i> <sub>2</sub>				1
<i>l</i> <sub>1</sub>		<i>S</i> <sub>3</sub>	<i>S</i> <sub>4</sub>	Accept	
<i>l</i> <sub>2</sub>		<i>R</i> <sub>3</sub>	<i>R</i> <sub>3</sub>	<i>R</i> <sub>3</sub>	
<i>l</i> <sub>3</sub>	<i>S</i> <sub>2</sub>				5
<i>l</i> <sub>4</sub>	<i>S</i> <sub>2</sub>				6
<i>l</i> <sub>5</sub>		<i>S</i> <sub>3</sub> / <i>R</i> <sub>1</sub>	<i>S</i> <sub>4</sub> / <i>R</i> <sub>1</sub>	<i>R</i> <sub>1</sub>	
<i>l</i> <sub>6</sub>		<i>S</i> <sub>3</sub> / <i>R</i> <sub>2</sub>	<i>S</i> <sub>4</sub> / <i>R</i> <sub>2</sub>	<i>R</i> <sub>2</sub>	

The conflict is resolved by giving higher precedence to \* and using leftassociativity, as shown in [Table 9.2](#).

**Table 9.2: Higher Precedent \* and Left-Associativity**

	id	+	*	\$	E
$l_0$	$S_2$				1
$l_1$		$S_3$	$S_4$	Accept	
$l_2$		$R_3$	$R_3$	$R_3$	
$l_3$	$S_2$				5
$l_4$	$S_2$				6
$l_5$		$R_1$	$S_4$	$R_1$	
$l_6$		$R_2$	$R_2$	$R_2$	

The parsing table with error routines is shown in [Table 9.3](#), where routine  $e_1$  is called from states  $l_0$ ,  $l_3$ , and  $l_4$ , which pushes an imaginary id onto the stack and covers it with state  $l_2$ . The routine  $e_2$  is called from state  $l_1$ , which pushes + onto stack and covers it with state  $l_3$ .

**Table 9.3: Parsing Table with Error Routines**

	id	+	*	\$	E
$l_0$	$S_2$	$e_1$	$e_1$	$e_1$	1
$l_1$	$E_2$	$S_3$	$S_4$	Accept	
$l_2$	$R_3$	$R_3$	$R_3$	$R_3$	
$l_3$	$S_2$	$e_1$	$E_1$	$E_1$	5
$l_4$	$S_2$	$E_1$	$E_1$	$E_1$	6
$l_5$	$R_1$	$R_1$	$S_4$	$R_1$	
$l_6$	$R_2$	$R_2$	$R_2$	$R_2$	

For example, if we trace the behavior of the parser described above for the input id + \*id \$:

Stack Contents	Unspent Input	Moves
\$ $l_0$	id + *id \$	shift and enter into state 2
\$ $l_0$ id $l_2$	+ *id \$	reduce by production number 3
\$ $l_0$ E $l_1$	+ *id \$	shift and enter into state 3
\$ $l_0$ E $l_1$ + $l_3$ id $l_2$	*id \$	call error routine $e_1$
\$ $l_0$ E $l_1$ + $l_3$ id $l_2$	*id \$	reduce by production number 3
<b>(id <math>l_2</math> pushed by <math>e_1</math>)</b>		
\$ $l_0$ E $l_1$ + $l_3$ E $l_5$	*id \$	shift and enter into state 4
\$ $l_0$ E $l_1$ + $l_3$ E $l_5$ * $l_4$	id \$	shift and enter into state 2
\$ $l_0$ E $l_1$ + $l_3$ E $l_5$ * $l_4$ id $l_2$	\$	reduce by production number 3

Stack Contents	Unspent Input	Moves
\$ \$0E1+\$3E1\$ \$4E1\$	\$	reduce by production number 2
\$ \$0E1+\$3E1\$	\$	reduce by production number 1
\$ \$0E1	\$	accept

Similarly, if we trace the behavior of the parser for the input id id\*id \$:

Stack Contents	Unspent Input	Moves
\$ \$0	id id*id\$	shift and enter into state 2
\$ \$0id\$	id*id\$	reduce by production number 3
\$ \$0E1	id*id\$	call error routine e2
\$ \$0E1+\$3	id*id\$	shift and enter into state 2
<b>(3 pushed by e2)</b>		
\$ \$0E1+\$3id\$	*id\$	reduce by production number 3
\$ \$0E1+\$3E1\$	*id\$	shift and enter into state 4
\$ \$0E1+\$3E1\$*\$4	id\$	shift and enter into state 2
\$ \$0E1+\$3E1\$*\$4id\$	\$	reduce by production number 3
\$ \$0E1+\$3E1\$*\$4E1\$	\$	reduce by production number 2
\$ \$0E1+\$3E1\$	\$	reduce by production number 1
\$ \$0E1	\$	accept

## 9.5 AUTOMATIC ERROR RECOVERY IN YACC

The tool YACC can generate a parser with the ability to automatically recover from the errors. Major nonterminals, such as those for program blocks or statements, are identified; and then error productions of the form  $A \rightarrow \text{error } \alpha$  are added to the grammar, where  $\alpha$  is usually  $\epsilon$ .

When YACC-generated parser encounters an error, it finds the top-most state on its stack, whose underlying set of items includes an item of the form  $A \rightarrow .\text{error}$ . Therefore, the parser shifts the token error, and a reduction to  $A$  is immediately possible. The parser then invokes a semantic action associated with production  $A \rightarrow \text{error}$ , and this semantic action takes care of recovering from the error.

## 9.6 PREDICTIVE PARSING ERROR RECOVERY

An error is detected during predictive parsing when the terminal on the top of the stack does not match the next input symbol, or when nonterminal A is on top of the stack and a is the next input symbol.  $M[A,a]$  is the error entry used to for recovery. Panic mode recovery can be used to recover from an error detected by the *LL* parser. The effectiveness of panic mode recovery depends on the choice of the synchronizing token. Several heuristics can be used when selecting the synchronizing token in order to ensure quick recovery from common errors:

1. All the symbols in the  $\text{FOLLOW}(A)$  must be kept in the set of synchronizing tokens, because if we skip until an a symbol in  $\text{FOLLOW}(A)$  is read, and we pop A from the stack, it is likely that the parsing can continue.
2. Since the syntactic structure of a language is very often hierarchical, we add the symbols that begin higher constructs to the synchronizing set of lower constructs. For example, we add keywords to the synchronizing sets of nonterminals that generate expressions.
3. We also add the symbols in  $\text{FIRST}(A)$  to the synchronizing set of nonterminal A. This provides for a resumption of parsing according to A if a symbol in  $\text{FIRST}(A)$  appears in the input.
4. A derivation by an  $\epsilon$ -production can be used as a default. Error detection will be postponed, but the error will still be captured. This method reduces the number of nonterminals that must be considered during error recovery.

**Note** Another method of error recovery that can be implemented is called "phrase level recovery". In phrase level recovery, each error entry in the *LL* parsing table is examined, and based on language usage, an appropriate error-recovery procedure is constructed. For example, to recover from a construct error that starts with an operator, the error-recovery routine will insert an imaginary id into the input. Then, if some state terminal symbols are derived using an  $\epsilon$ -production, the error entries in that state are replaced by the derivation using the imaginary-id  $\epsilon$ -production. This has the effect of postponing error detection.

A phrase level error-recovery implementation for an *LR* parser is shown in [Tables 9.4](#) and [9.5](#). The parsing table is constructed for the following grammar:

$$\begin{aligned} E &\rightarrow TE_1 \\ E_1 &\rightarrow +TE_1 \mid \epsilon \\ T &\rightarrow FT_1 \\ T_1 &\rightarrow *FT_1 \mid \epsilon \\ F &\rightarrow \text{id} \end{aligned}$$

**Table 9.4: LR Parsing Table**

	id	+	*	\$
$E$	$E \rightarrow TE_1$			
$T$	$T \rightarrow FT_1$			
$F$	$F \rightarrow id$			
$E_1$		$E_1 \rightarrow +TE_1$		$E_1 \rightarrow \epsilon$
$T_1$		$T_1 \rightarrow \epsilon$	$T_1 \rightarrow *FT_1$	$T_1 \rightarrow \epsilon$
id	pop			
+		pop		
*			pop	
\$				accept

The modified table is shown in [Table 9.5](#). Routine  $e_1$ , when called, pushes an imaginary id into the input; and routine  $e_2$ , when called, removes all the remaining symbols from the input.

**Table 9.5: Phrase Level Error-Recovery Implementation**

	id	+	*	\$
$E$	$E \rightarrow TE_1$	$e_1$	$e_1$	$e_1$
$T$	$T \rightarrow FT_1$	$e_1$	$e_1$	$e_1$
	$F \rightarrow id$	$e_1$	$e_1$	$e_1$
$E_1$	$E_1 \rightarrow \epsilon$	$E_1 \rightarrow +TE_1$	$E_1 \rightarrow \epsilon$	$E_1 \rightarrow \epsilon$
$T_1$	$T_1 \rightarrow \epsilon$	$T_1 \rightarrow \epsilon$	$T_1 \rightarrow *FT_1$	$T_1 \rightarrow \epsilon$
id	pop			
+		pop		
*			pop	
\$	$e_2$	$e_2$	$e_2$	accept

For example, if we trace the behavior of the parser shown in [Table 9.5](#) for the input id + \*id \$:

Stack Contents	Unspent Input	Moves
\$E	id + *id \$	derive using $E \rightarrow TE_1$
\$E_1 T	id + *id \$	derive using $T \rightarrow FT_1$
\$E_1 T_1 F	id + *id \$	derive using $F \rightarrow id$
\$E_1 T_1 id	id + *id \$	pop
\$E_1 T_1	+ *id \$	derive using $T_1 \rightarrow \epsilon$

Stack Contents	Unspent Input	Moves
\$E1	+*id\$	derive using $E_1 \rightarrow +TE_1$
\$E1T+	+*id\$	pop
\$E1T	*id\$	call error routine e1
\$E1T	id*id\$	derive using $T \rightarrow FT_1$
(imaginary id is pushed by e1)		
\$E1T1F	id*id\$	derive using $F \rightarrow id$
\$E1T1id	id*id\$	pop
\$E1T1	*id\$	derive using $T_1 \rightarrow *FT_1$
\$E1T1F	id\$	derive using $F \rightarrow id$
\$E1T1id	id\$	pop
\$E1T1	\$	derive using $T_1 \rightarrow \epsilon$
\$E1	\$	derive using $E_1 \rightarrow \epsilon$
\$	\$	accept

Similarly, if we trace the behavior for the input id id\*id \$:

Stack Contents	Unspent Input	Moves
\$E	id id*id\$	derive using $E \rightarrow TE_1$
\$E1T	id+*id\$	derive using $T \rightarrow FT_1$
\$E1T1F	id+*id\$	derive using $F \rightarrow id$
\$E1T1id	id+*id\$	pop
\$E1T1	id*id\$	derive using $T_1 \rightarrow \epsilon$
\$E1	id*id\$	derive using $E_1 \rightarrow \epsilon$
\$	id*id\$	call error routine e2
(id*id\$ is removed by e2)		
\$	\$	accept

## 9.7 RECOVERY FROM SEMANTIC ERRORS

The primary sources of semantic errors are undeclared names and type incompatibilities. Recovery from an undeclared name is rather straightforward. The first time the undeclared name is encountered, an entry can be made in the symbol table for that name with an attribute that is appropriate to the current context. For example, if missing declaration error of  $x$  is encountered, then the error-recovery routine enters the appropriate attribute for  $x$  in  $x$ 's symbol table, depending on the current context of  $x$ . A flag is then set in the  $x$  symbol table record to indicate that an attribute has been added, and to recover from an error or not in response to the declaration of  $x$ .

## Chapter 10: Code Optimization

### 10.1 INTRODUCTION TO CODE OPTIMIZATION

The translation of a source program to an object program is basically one of many mappings; that is, there are many object programs for the same source program, which implement the same computations. Some of these object-translated source programs may be better than other object programs when it comes to storage requirements and execution speeds. Code optimization refers to techniques a compiler can employ in order to produce an improved object code for a given source program.

How beneficial the optimization is depends upon the situation. For a program that is only expected to be run a few times, and which will then be discarded, no optimization is necessary. Whereas if a program is expected to run indefinitely, or if it is expected to run many times, then optimization is useful, because the effort spent on improving the program's execution time will be paid back, even if execution time is only reduced by a small percentage.

What follows are some optimization techniques that are useful when designing optimizing compilers.

## 10.2 WHAT IS CODE OPTIMIZATION?

Code optimization refers to the techniques used by the compiler to improve the execution efficiency of the generated object code. It involves a complex analysis of the intermediate code and the performance of various transformations; but every optimizing transformation must also preserve the semantics of the program. That is, a compiler should not attempt any optimization that would lead to a change in the program's semantics.

Optimization can be machine-independent or machine-dependent. Machine-independent optimizations can be performed independently of the target machine for which the compiler is generating code; that is, the optimizations are not tied to the target machine's specific platform or language. Examples of machine-independent optimizations are: elimination of loop invariant computation, induction variable elimination, and elimination of common subexpressions.

On the other hand, machine-dependent optimization requires knowledge of the target machine. An attempt to generate object code that will utilize the target machine's registers more efficiently is an example of machine-dependent code optimization. Actually, code optimization is a misnomer; even after performing various optimizing transformations, there is no guarantee that the generated object code will be optimal. Hence, we are actually performing code improvement. When attempting any optimizing transformation, the following criteria should be applied:

1. The optimization should capture most of the potential improvements without an unreasonable amount of effort.
2. The optimization should be such that the meaning of the source program is preserved.
3. The optimization should, on average, reduce the time and space expended by the object code.

## 10.3 LOOP OPTIMIZATION

Loop optimization is the most valuable machine-independent optimization because a program's inner loops are good candidates for improvement. The important loop optimizations are elimination of loop invariant computations and elimination of induction variables. A loop invariant computation is one that computes the same value every time a loop is executed. Therefore, moving such a computation outside the loop leads to a reduction in the execution time. Induction variables are those variables used in a loop; their values are in lock-step, and hence, it may be possible to eliminate all except one.

### 10.3.1 Eliminating Loop Invariant Computations

To eliminate loop invariant computations, we first identify the invariant computations and then move them outside loop if the move does not lead to a change in the program's meaning. Identification of loop invariant computation requires the detection of loops in the program. Whether a loop exists in the program or not depends on the program's control flow, therefore, requiring a control flow analysis. For loop detection, a graphical representation, called a "program flow graph," shows how the control is flowing in the program and how the control is being used. To obtain such a graph, we must partition the intermediate code into basic blocks. This requires identifying leader statements, which are defined as follows:

1. The first statement is a leader statement.
2. The target of a conditional or unconditional goto is a leader.
3. A statement that immediately follows a conditional goto is a leader.

A basic block is a sequence of three-address statements that can be entered only at the beginning, and control ends after the execution of the last statement, without a halt or any possibility of branching, except at the end.

### 10.3.2 Algorithm to Partition Three-Address Code into Basic Blocks

To partition three-address code into basic blocks, we must identify the leader statements in the three-address code and then include all the statements, starting from a leader, and up to, but not including, the next leader. The basic blocks into which the three-address code is partitioned constitute the nodes or vertices of the program flow graph. The edges in the flow graph are decided as follows. If  $B_1$  and  $B_2$  are the two blocks, then add an edge from  $B_1$  to  $B_2$  in the program flow graph, if the block  $B_2$  follows  $B_1$  in an execution sequence. The block  $B_2$  follows  $B_1$  in an execution sequence if and only if:

1. The first statement of block  $B_2$  immediately follows the last statement of block  $B_1$  in the three-address code, and the last statement of block  $B_1$  is not an unconditional goto statement.
2. The last statement of block  $B_1$  is either a conditional or unconditional goto statement, and the first statement of block  $B_2$  is the target of the last statement of block  $B_1$ .

For example, consider the following program fragment:

```
Fact(x)
{
    int f = 1;
    for(i = 2; i<=x; i++)
        f = f*i;
    return(f);
}
```

The three-address-code representation for the program fragment above is:

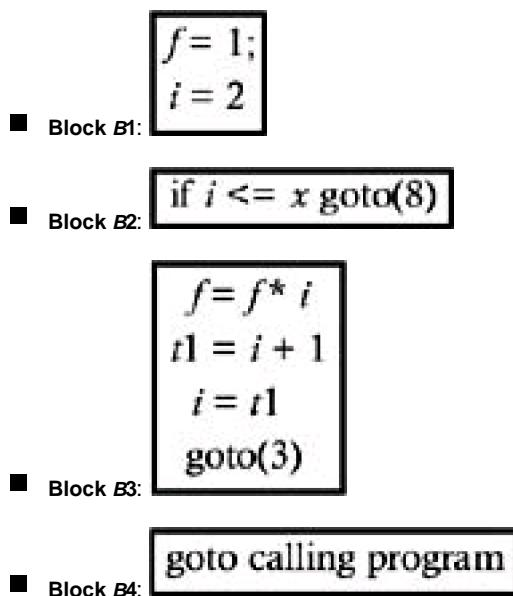
1.  $f = 1;$
2.  $i = 2$

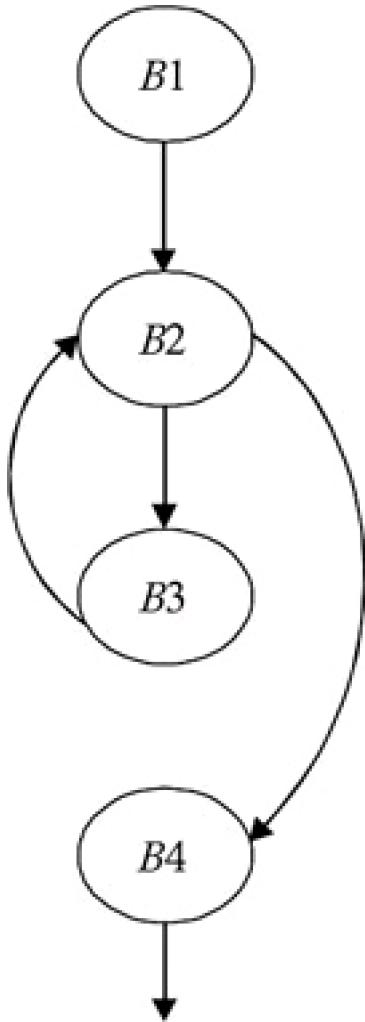
3. if  $i \leq x$  goto(8)
4.  $f = f * i$
5.  $t1 = i + 1$
6.  $i = t1$
7. goto(3)
8. goto calling program

The leader statements are:

- Statement number 1, because it is the first statement.
- Statement number 3, because it is the target of a goto.
- Statement number 4, because it immediately follows a conditional goto statement.
- Statement number 8, because it is a target of a conditional goto statement.

Therefore, the basic blocks into which the above code can be partitioned are as follows, and the program flow graph is shown in [Figure 10.1](#).





**Figure 10.1:** Program flow graph.

### 10.3.3 Loop Detection

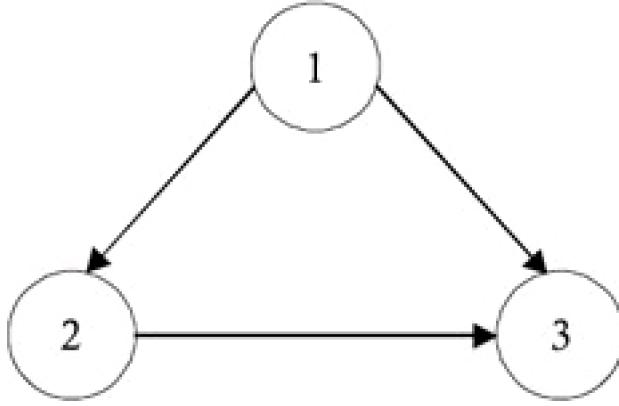
A loop is a cycle in the flow graph that satisfies two properties:

1. It should have a single entry node or header, so that it will be possible to move all of the loop invariant computations to a unique place, called a "preheader," which is a block/node placed outside the loop, just in front of the header.
2. It should be strongly connected; that is, it should be possible to go from any node of the loop to any other node while staying within the loop. This is required until at least some of the loops get executed repeatedly.

If the flow graph contains one or more back edges, then only one or more loops/ cycles exist in the program. Therefore, we must identify any back edges in the flow graph.

### 10.3.4 Identification of the Back Edges

To identify the back edges in the flow graph, we compute the dominators of every node of the program flow graph. A node  $a$  is a dominator of node  $b$  if all the paths starting at the initial node of the graph that reach to node  $b$  go through  $a$ . For example, consider the flow graph in [Figure 10.2](#). In this flow graph, the dominator of node 3 is only node 1, because all the paths reaching up to node 3 from node 1 do not go through node 2.



**Figure 10.2:** The flow graph back edges are identified by computing the dominators.

Dominator (dom) relationships have the following properties:

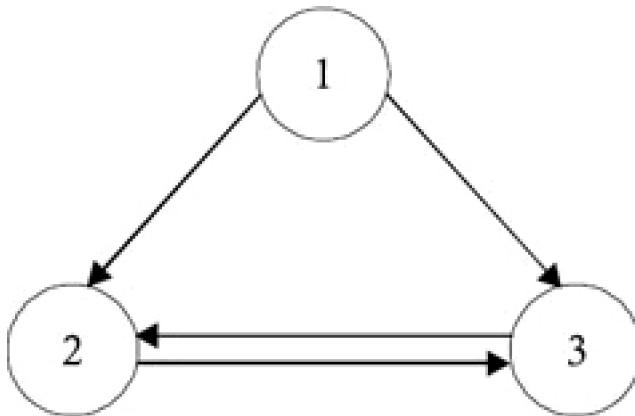
1. They are reflexive; that is, every node dominates itself.
2. That are transitive; that is, if  $a \text{ dom } b$  and  $b \text{ dom } c$ , this implies  $a \text{ dom } c$ .

### 10.3.5 Reducible Flow Graphs

Several code-optimization transformations are easy to perform on reducible flow graphs. A flow graph  $G$  is reducible if and only if we can partition the edges into two disjointed groups, forward edges and back edges, with the following two properties:

1. The forward edges form an acyclic graph in which every node can be reached from the initial node  $G$ .
2. The back edges consist only of edges whose heads dominate their tails.

For example, consider the flow graph shown in [Figure 10.3](#). This flow graph has no back edges, because no edge's head dominates the tail of that edge. Hence, it could have been a reducible graph if the entire graph had been acyclic. But that is not the case. Therefore, it is not a reducible flow graph.



**Figure 10.3:** A flow graph with no back edges.

After identifying the back edges, if any, the natural loop of every back edge must be identified. The natural loop of a back edge  $a \rightarrow b$  is the set of all those nodes that can reach  $a$  without going through  $b$ , including node  $b$  itself. Therefore, to find a natural loop of the back edge  $n \rightarrow d$ , we start with node  $n$  and add all the predecessors of node  $n$  to the loop. Then we add the predecessors of the nodes that were just added to the loop; and we continue this process until we reach node  $d$ . These nodes plus node  $d$  constitute the set of all those nodes that can reach node  $n$  without going through node  $d$ . This is the natural loop of the edge  $n \rightarrow d$ . Therefore, the algorithm for detecting the natural loop of a back edge is:

Input : back edge  $n \rightarrow d$ .  
Output: set loop, which is a set of nodes forming the natural  
loop of the back edge  $n \rightarrow d$ .

```

main()
{
    loop = { d } /* Initialize by adding node d to the set loop*/
    insert(n); /* call a procedure insert with the node n */
}
procedure insert(m)
{
    if m is not in the loop then
    {
        loop = loop ∪ { m }
        for every predecessor p of m do
            insert(p);
    }
}

```

For example in the flow graph shown in [Figure 10.1](#), the back edges are edge  $B3 \rightarrow B2$ , and the loop is comprised of the blocks  $B2$  and  $B3$ .

After the natural loops of the back edges are identified, the next task is to identify the loop invariant computations. The three-address statement  $x = y \text{ op } z$ , which exists in the basic block  $B$  (a part of the loop), is a loop invariant statement if all possible definitions of  $b$  and  $c$  that reach upto this statement are outside the loop, or if  $b$  and  $c$  are constants, because then the calculation  $b \text{ op } c$  will be the same each time the statement is encountered in the loop. Hence, to decide whether the statement  $x = b \text{ op } c$  is loop invariant or not, we must compute the  $u-d$  chaining information. The  $u-d$  chaining information is computed by doing a global data flow analysis of the flow graph. All of the definitions that are capable of reaching to a point immediately before the start of the basic block are computed, and we call the set of all such definitions for a block  $B$  the  $\text{IN}(B)$ . The set of all the definitions capable of reaching to a point immediately after the last statement of block  $B$  will be called  $\text{OUT}(B)$ . We compute both  $\text{IN}(B)$  and  $\text{OUT}(B)$  for every block  $B$ ,  $\text{GEN}(B)$  and  $\text{KILL}(B)$ , which are defined as:

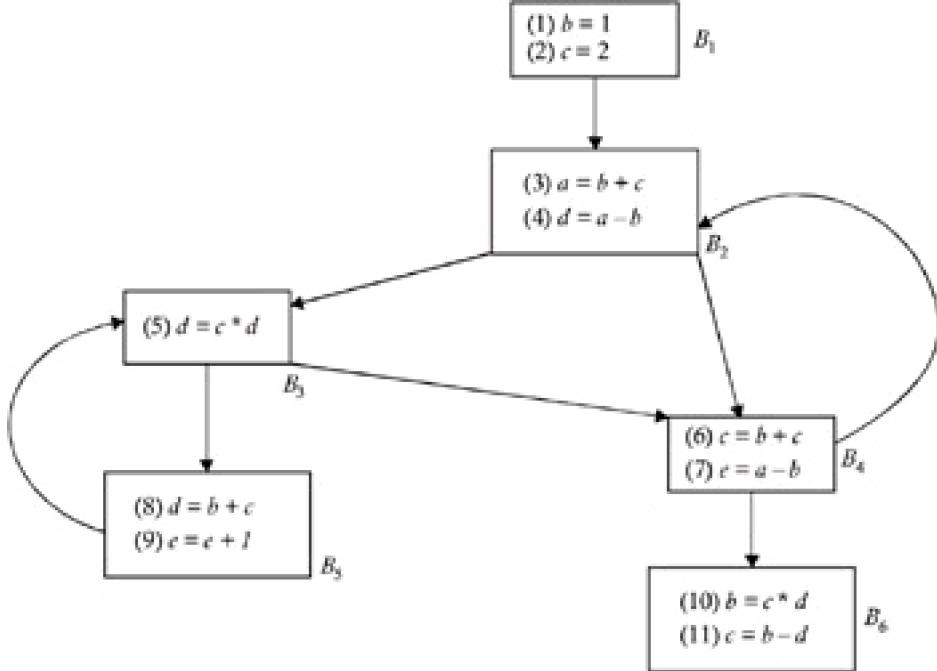
- $\text{GEN}(B)$ : The set of all the definitions generated in block  $B$ .
- $\text{KILL}(B)$ : The set of all the definitions outside block  $B$  that define the same variables as are defined in block  $B$ .

Consider the flow graph in [Figure 10.4](#).

The GEN and KILL sets for the basic blocks are as shown in [Table 10.1](#).

**Table 10.1: GEN and KILL sets for [Figure 10.4](#) Flow Graph**

Block	GEN	KILL
$B1$	{1,2}	{6,10,11}
$B2$	{3,4}	{5,8}
$B3$	{5}	{4,8}
$B4$	{6,7}	{2,9,11}
$B5$	{8,9}	{4,5,7}
$B6$	{10,11}	{1,2,6}

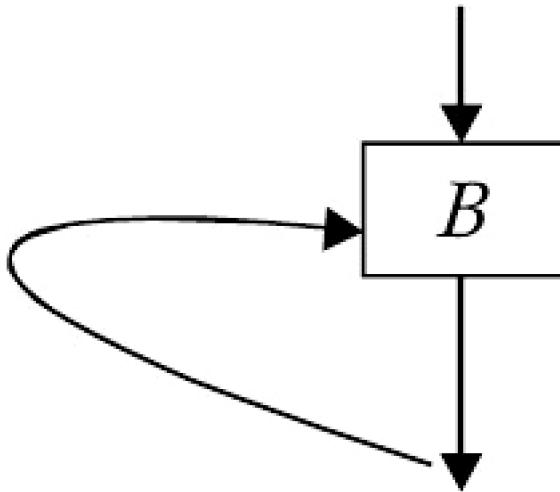


**Figure 10.4:** Flow graph with GEN and KILL block sets.

$\text{IN}(B)$  and  $\text{OUT}(B)$  are defined by the following set of equations, which are called "data flow equations":

$$\begin{aligned}\text{IN}(B) &= \cup \text{OUT}(P) \\ \text{OUT}(B) &= \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)\end{aligned}$$

The next step, therefore, is to solve these equations. If there are  $n$  nodes, there will be  $2n$  equations in  $2n$  unknowns. The solution to these equations is not generally unique. This is because we may have a situation like that shown in [Figure 10.5](#), where a block  $B$  is a predecessor of itself.



**Figure 10.5:** Nonunique solution to a data flow equation, where  $B$  is a predecessor of itself.

If there is a solution to the data flow equations for block  $B$ , and if the solution is  $\text{IN}(B) = \text{IN}_0$  and  $\text{OUT}(B) = \text{OUT}_0$ , then  $\text{IN}_0 \cup \{d\}$  and  $\text{OUT}_0 \cup \{d\}$ , where  $d$  is any definition not in  $\text{IN}_0$ .  $\text{OUT}_0$  and  $\text{KILL}(B)$  also satisfy the equations, because if we take  $\text{OUT}_0 \cup \{d\}$  as the value of  $\text{OUT}(B)$ , since  $B$  is one of the predecessors of itself according to  $\text{IN}(B) = \cup \text{OUT}(P)$ ,  $d$  gets added to  $\text{IN}(B)$ , because  $d$  is not in the  $\text{KILL}(B)$ . Hence, we get  $\text{IN}(B) = \text{IN}_0 \cup \{d\}$ . And according to  $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$ ,  $\text{OUT}(B) = \text{OUT}_0 \cup \{d\}$  gets satisfied. Therefore,  $\text{IN}_0, \text{OUT}_0$  is one of the solutions, whereas  $\text{IN}_0 \cup \{d\}, \text{OUT}_0 \cup \{d\}$  is another solution to the equations—no unique solution. What we are interested in is finding smallest solution, that is, the smallest  $\text{IN}(B)$  and  $\text{OUT}(B)$  for every block  $B$ , which consists of values that are in all solutions. For example, since  $\text{IN}_0$  is in  $\text{IN}_0 \cup \{d\}$ , and  $\text{OUT}_0$  is in  $\text{OUT}_0 \cup \{d\}$ ,  $\text{IN}_0, \text{OUT}_0$  is the smallest solution. And

this is what we want, because the smallest  $\text{IN}(B)$  turns out to be the set of all definitions reaching the point just before the beginning of  $B$ . The algorithm for computing the smallest  $\text{IN}(B)$  and  $\text{OUT}(B)$  is as follows:

```

1. For each block B do
{
     $\text{IN}(B) = \emptyset$ 
     $\text{OUT}(B) = \text{GEN}(B)$ 
}

2. flag = true

3. while (flag) do
{
    flag = false
    for each block B do
    {
         $\text{IN}_{\text{new}}(B) = \Phi$ 
        for each predecessor  $P$  of  $B$ 
         $\text{IN}_{\text{new}}(B) = \text{IN}_{\text{new}}(B) \cup \text{OUT}(P)$ 
        if  $\text{IN}_{\text{new}}(B) \neq \text{IN}(B)$  then
        {
            flag = true
             $\text{IN}(B) = \text{IN}_{\text{new}}(B)$ 
             $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$ 
        }
    }
}
}

```

Initially, we take  $\text{IN}(B)$  for every block that is to be an empty set, and we take  $\text{OUT}(B)$  for  $\text{GEN}(B)$ , and we compute  $\text{IN}_{\text{new}}(B)$ . If it is different from  $\text{IN}(B)$ , we compute a new  $\text{OUT}(B)$  and go for the next iteration. This is continued until  $\text{IN}(B)$  comes out to be the same for every  $B$  in a previous or current iteration.

For example, for the flow graph shown in [Figure 10.5](#), the IN and OUT iterations for the blocks are computed using above algorithm, as shown in [Tables 10.2–10.6](#).

**Table 10.2: IN and OUT Computation for [Figure 10.5](#)**

Block	IN	OUT
$B_1$	$\Phi$	{1,2}
$B_2$	$\Phi$	{3,4}
$B_3$	$\Phi$	{5}
$B_4$	$\Phi$	{6,7}
$B_5$	$\Phi$	{8,9}
$B_6$	$\Phi$	{10,11}

**Table 10.3: First Iteration for the IN and OUT Values**

Block	IN	OUT
B1	$\Phi$	{1,2}
B2	{1,2,6,7}	{1,2,3,4,6,7}
B3	{3,4,8,9}	{3,5,9}
B4	{3,4,5}	{3,4,5,6,7}
B5	{5}	{8,9}
B6	{6,7}	{7,10,11}

**Table 10.4: Second Iteration for the IN and OUT Values**

Block	IN	OUT
B1	$\Phi$	{1,2}
B2	{1,2,3,4,5,6,7}	{1,2,3,4,6,7}
B3	{1,2,3,4,6,7,8,9}	{1,2,3,5,6,7,9}
B4	{1,2,3,4,5,6,7,9}	{1,3,4,5,6,7}
B5	{3,5,9}	{3,8,9}
B6	{3,4,5,6,7}	{3,4,5,7,10,11}

**Table 10.5: Third Iteration for the IN and OUT Values**

Block	IN	OUT
B1	$\Phi$	{1,2}
B2	{1,2,3,4,5,6,7}	{1,2,3,4,6,7}
B3	{1,2,3,4,6,7,8,9}	{1,2,3,5,6,7,9}
B4	{1,2,3,4,5,6,7,9}	{1,3,4,5,6,7}
B5	{1,2,3,5,6,7,9}	{1,2,3,6,8,9}
B6	{1,3,4,5,6,7}	{1,3,4,5,7,10,11}

**Table 10.6: Fourth Iteration for the IN and OUT Values**

Block	IN	OUT
B1	$\Phi$	{1,2}
B2	{1,2,3,4,5,6,7}	{1,2,3,4,6,7}
B3	{1,2,3,4,6,7,8,9}	{1,2,3,5,6,7,9}
B4	{1,2,3,4,5,6,7,9}	{1,3,4,5,6,7}
B5	{1,2,3,5,6,7,9}	{1,2,3,6,8,9}
B6	{1,3,4,5,6,7}	{1,3,4,5,7,10,11}

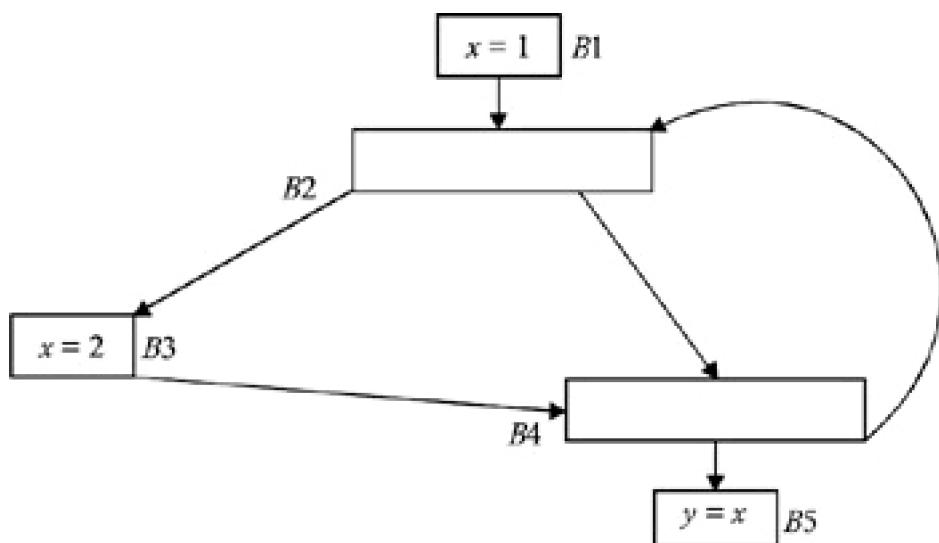
The next step is to compute the  $u-d$  chains from the reaching definitions information, as follows.

If the use of  $A$  in block  $B$  is preceded by its definition, then the  $u-d$  chain of  $A$  contains only the last definition prior to this use of  $A$ . If the use of  $A$  in block  $B$  is not preceded by any definition of  $A$ , then the  $u-d$  chain for this use consists of all definitions of  $A$  in  $IN(B)$ .

For example, in the flow graph for which IN and OUT were computed in [Tables 10.2–10.6](#), the use of  $a$  in definition 4, block  $B2$  is preceded by definition 3, which is the definition of  $a$ . Hence, the  $u-d$  chain for this use of  $a$  only contains definition 3. But the use of  $b$  in  $B2$  is not preceded by any definition of  $b$  in  $B2$ . Therefore, the  $u-d$  chain for this use of  $B$  will be {1}, because this is the only definition of  $b$  in  $IN(B2)$ .

The  $u-d$  chain information is used to identify the loop invariant computations. The next step is to perform the code motion, which moves a loop invariant statement to a newly created node, called "preheader," whose only successor is a header of the loop. All the predecessors of the header that lie outside the loop will become predecessors of the preheader.

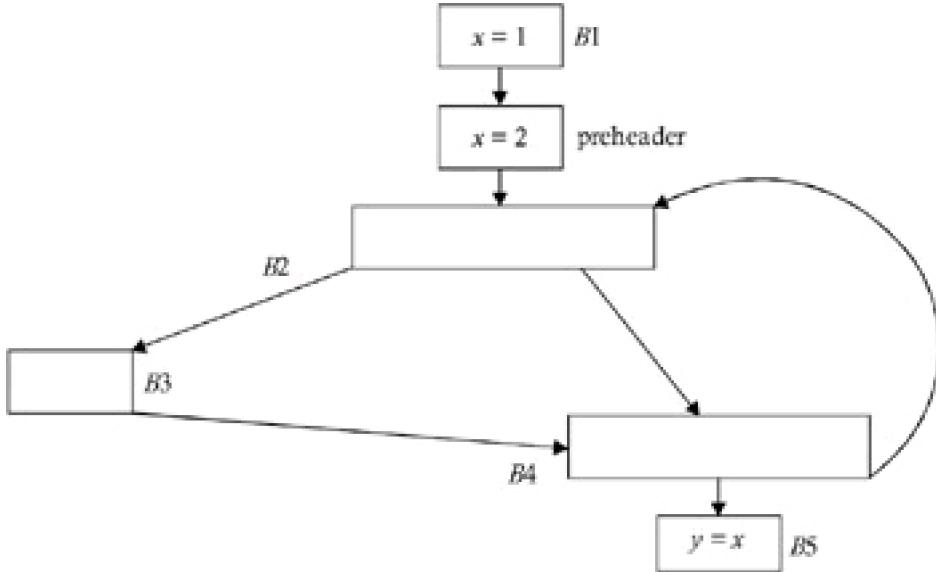
But sometimes the movement of a loop invariant statement to the preheader is not possible because such a move would alter the semantics of the program. For example, if a loop invariant statement exists in a basic block that is not a dominator of all the exits of the loop (where an exit of the loop is the node whose successor is outside the loop), then moving the loop invariant statement in the preheader may change the semantics of the program. Therefore, before moving a loop invariant statement to the preheader, we must check whether the code motion is legal or not. Consider the flow graph shown in [Figure 10.6](#).



**Figure 10.6:** A flow graph containing a loop invariant statement.

In the flow graph shown in [Figure 10.6](#),  $x = 2$  is the loop invariant. But since it occurs in  $B3$ , which is not the dominator

of the exit of loop, if we move it to the preheader, as shown in [Figure 10.7](#), a value of two will always get assigned to  $y$  in  $B5$ ; whereas in the original program,  $y$  in  $B5$  may get value one as well as two.

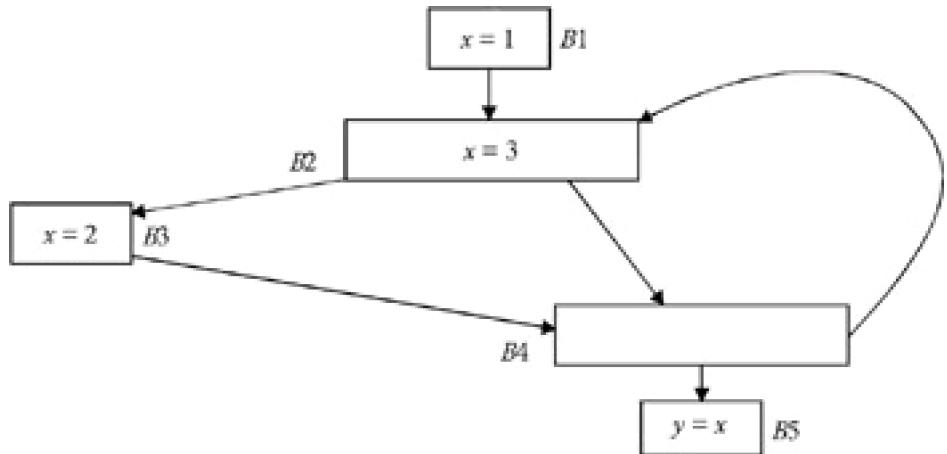


**Figure 10.7:** Moving a loop invariant statement changes the semantics of the program.

### After Moving $x = 2$ to the Preheader

In the flow graph shown in [Figure 10.7](#), if  $x$  is not used outside the loop, then the statement  $x = 2$  can be moved to the preheader. Therefore, for a code motion to be legal, the following conditions must be met, even if no errors are encountered:

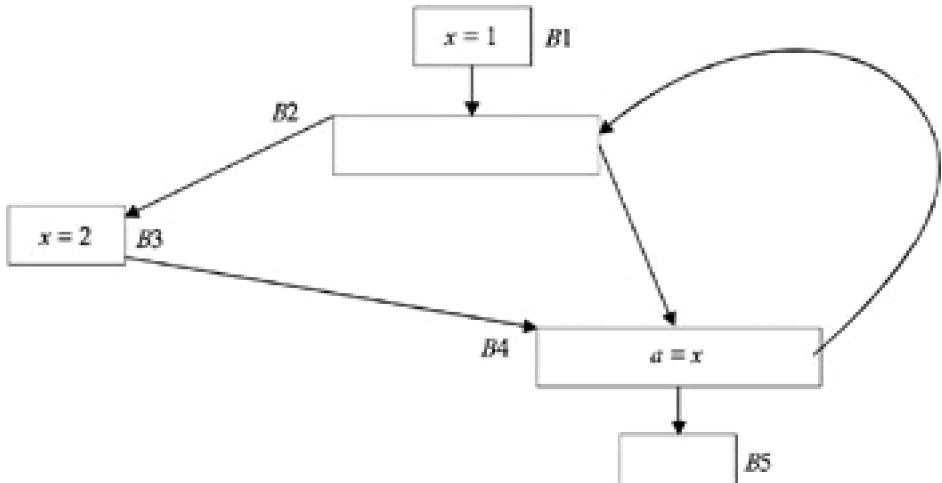
1. The block in which a loop invariant statement occurs should be a dominator of all exits of the loop, or the name assigned to the block should not be used outside the loop.
2. We cannot move a loop invariant statement assigned to  $A$  into preheader if there is another statement in the loop that assigns to  $A$ . For example, consider the flow graph shown in [Figure 10.8](#).



**Figure 10.8:** Moving the preheader changes the meaning of the program.

Even though the statement  $x = 3$  in  $B2$  satisfies condition (1), moving it to the preheader will change the meaning of the program. Because if  $x = 3$  is moved to the preheader, then the value that will be assigned to  $y$  in  $B5$  will be two if the execution path is  $B1-B2-B3-B4-B2-B4-B5$ . Whereas for the same execution path, the original program assigns a three to  $y$  in  $B5$ .

3. The move is illegal if  $A$  is used in the loop, and  $A$  is reached by any definition of  $A$  other than the statement to be moved. For example, consider the flow graph shown in [Figure 10.9](#).



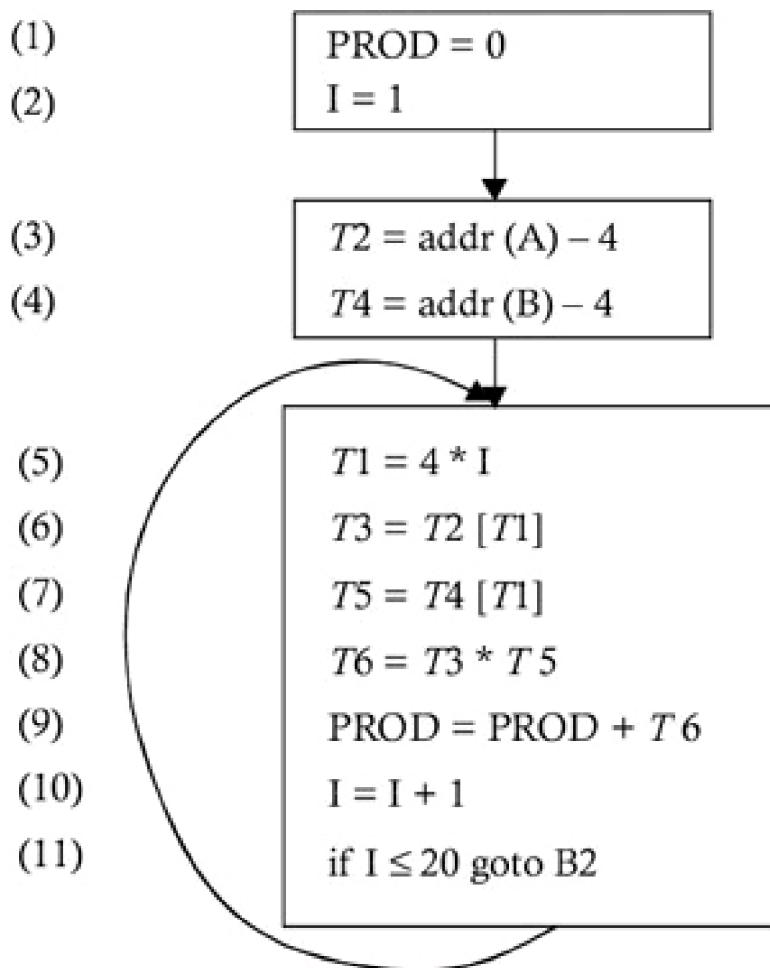
**Figure 10.9:** Moving a value to the preheader changes the original meaning of the program.

Even though  $x$  is not used outside the loop, the statement  $x = 2$  in the block  $B2$  cannot be moved to the preheader, because the use of  $x$  in  $B4$  is also reached by the definition  $x = 1$  in  $B1$ . Therefore, if we move  $x = 2$  to the preheader, then the value that will get assigned to  $a$  in  $B4$  will always be a one, which is not the case in the original program.

## 10.4 ELIMINATING INDUCTION VARIABLES

We define basic induction variables of a loop as those names whose only assignments within the loop are of the form  $I = I \pm C$ , where  $C$  is a constant or a name whose value does not change within the loop. A basic induction variable may or may not form an arithmetic progression at the loop header.

For example, consider the flow graph shown in [Figure 10.10](#). In the loop formed by  $B2$ ,  $I$  is a basic induction variable.



**Figure 10.10:** Flow graph where  $I$  is a basic induction variable.

We then define an induction variable of loop  $L$  as either a basic induction variable or a name  $J$  for which there is a basic induction variable  $I$ , such that each time  $J$  is assigned in  $L$ ,  $J$ 's value is some linear function or value of  $I$ . That is, the value of  $J$  in  $L$  should be  $C_1 I + C_2$ , where  $C_1$  and  $C_2$  could be functions of both constants and loop invariant names. For example, in loop  $L$ ,  $I$  is a basic induction variable; and  $T1$  is also an induction variable, because the only assignment of  $T1$  in the loop assigns a value to  $T1$  that is a linear function of  $I$ , computed as  $4 * I$ .

### Algorithm for Detecting and Eliminating Induction Variables

An algorithm exists that will detect and eliminate induction variables. Its method is as follows:

1. Find all of the basic induction variables by scanning the statements of loop  $L$ .
2. Find any additional induction variables, and for each such additional induction variable  $A$ , find the family of some basic induction  $B$  to which  $A$  belongs. (If the value of  $A$  at the point of assignment is expressed as  $C_1 B + C_2$ , then  $A$  is said to belong to the family of basic induction variable  $B$ ). Specifically, we search for names  $A$  with single assignments to  $A$  within loop  $L$ , and which have

one of the following forms:

$$A = B * C$$

$$A = C * B$$

$$A = B/C$$

$$A = B \pm C$$

$$A = C \pm B$$

where  $C$  is a loop constant, and  $B$  is an induction variable, basic or otherwise. If  $B$  is basic, then  $A$  is in the family of  $B$ . If  $B$  is not basic, let  $B$  be in the family of  $D$ , then the additional requirements to be satisfied are:

- a. There must be no assignment to  $D$  between the lone point of assignment to  $B$  in  $L$  and the assignment to  $A$ .
  - b. There must be no definition of  $B$  outside of  $L$  reaches  $A$ .
3. Consider each basic induction variable  $B$  in turn. For every induction variable  $A$  in the family of  $B$ :
- a. Create a new name,  $\text{temp}$ .
  - b. Replace the assignment to  $A$  in the loop with  $A = \text{temp}$ .
  - c. Set  $\text{temp}$  to  $C_1 B + C_2$  at the end of the preheader by adding the statements:

$$\text{temp} = C_1 * B$$

$$\text{temp} = \text{temp} + C_2 /* \text{omit if } C_2 = 0 */$$

- d. Immediately after each assignment  $B = B + D$ , where  $D$  is a loop invariant, append:

$$\text{temp} = \text{temp} + C_1 * D$$

If  $D$  is a loop invariant name, and if  $C_1 \neq 1$ , create a new loop invariant name for  $C_1 * D$ , and add the statements:

$$\text{temp1} = C_1 * D$$

$$\text{temp} = \text{temp} + \text{temp1}$$

- e. For each basic induction variable  $B$  whose only uses are to compute other induction variables in its family and in conditional branches, take some  $A$  in  $B$ 's family, preferably one whose function expresses its value simply, and replace each test of the form  $B$  reloop  $X$  goto  $Y$  by:

$$\text{temp2} = C_1 * X$$

$$\text{temp2} = \text{temp2} + C_2 /* \text{omit if } C_2 = 0 */$$

$$\text{if temp reloop temp2 goto } Y$$

Delete all assignments to  $B$  from the loop, as they will now be useless.

- f. If there is no assignment to  $\text{temp}$  between the introduced statement  $A = \text{temp}$  (step 1) and the only use of  $A$ , then replace all uses of  $A$  by  $\text{temp}$  and delete the statement  $A = \text{temp}$ .

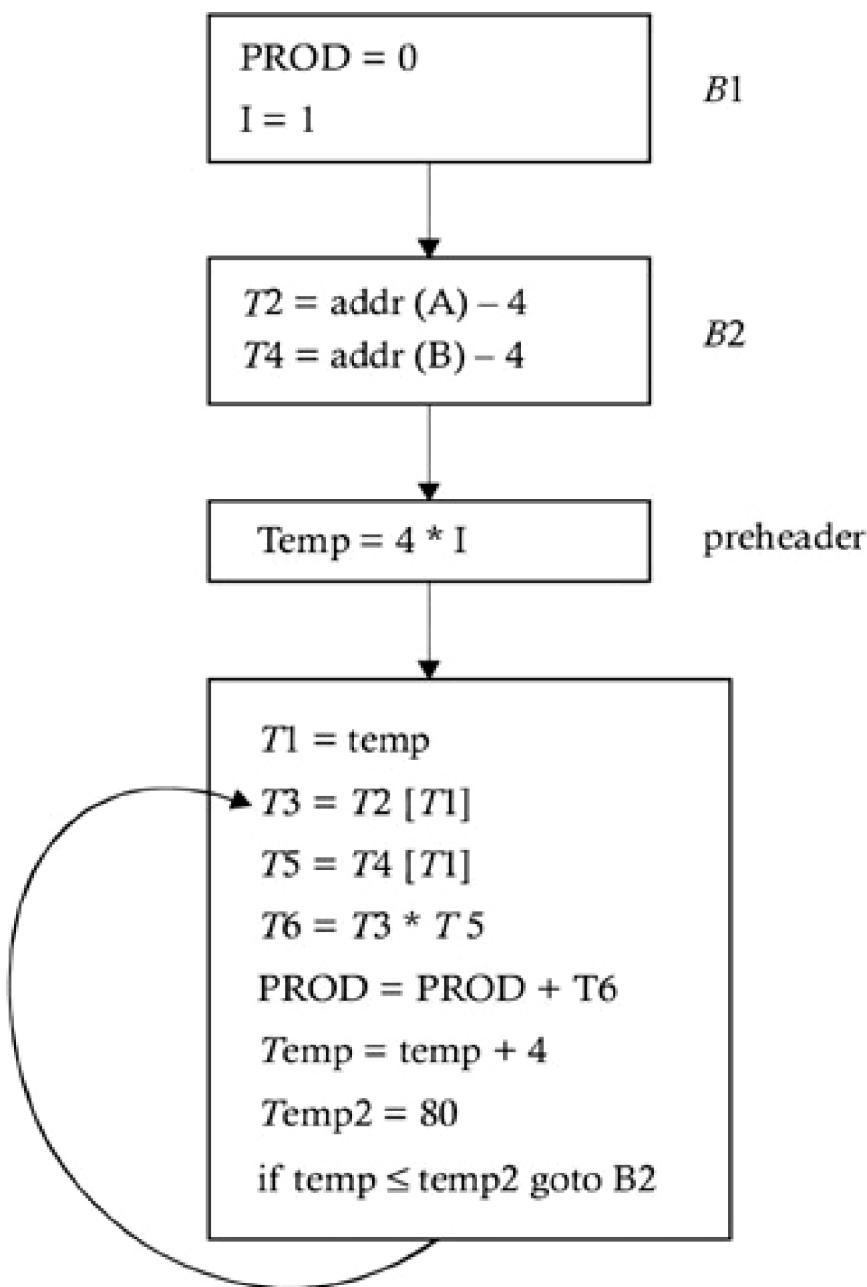
In the flow graph shown in [Figure 10.10](#), we see that  $I$  is a basic induction variable, and  $T1$  is the additional induction variable in the family of  $I$ , because

the value of  $T1$  at the point of assignment in the loop is expressed as  $T1 = 4 * I$ . Therefore, according to step 3b, we replace  $T1 = 4 * I$  by  $T1 = \text{temp}$ . And according to step 3c, we add  $\text{temp} = 4 * I$  to the preheader. We then append the statement  $\text{temp} = \text{temp} + 4$  after [Figure 10.10](#) statement (10), as per step 3d. And according to step 3e, we replace the statement if  $I \leq 20$  goto B2 by:

**temp1 = 80**

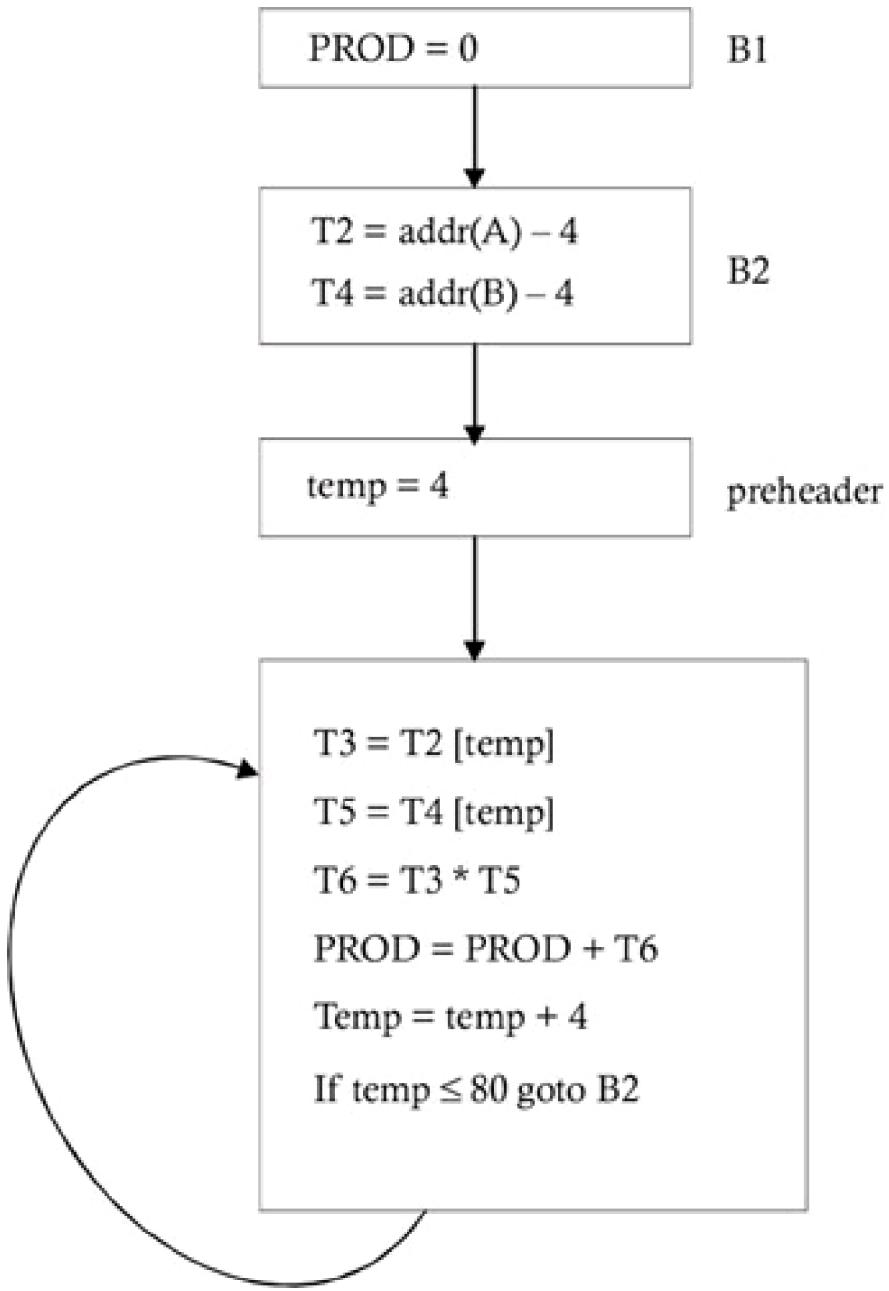
**if (temp ≤ temp1) goto B2, and delete  $I = I + 1$**

The results of these modifications are shown in [Figure 10.11](#).



**Figure 10.11:** Modified flow graph.

By step 3f, replace  $T1$  by  $\text{temp}$ . And by copy propagation,  $\text{temp} = 4 * I$ , in the preheader, can be replaced by  $\text{temp} = 4$ , and the statement  $I = 1$  can be eliminated. In B1, the statement if  $\text{temp} \leq \text{temp2 goto B2}$  can be replaced by if  $\text{temp} \leq 80 \text{ goto B2}$ , and we can eliminate  $\text{temp2} = 80$ , as shown in [Figure 10.12](#).



**Figure 10.12:** Flow graph preheader modifications.

## 10.5 ELIMINATING LOCAL COMMON SUBEXPRESSIONS

The first step in eliminating local common subexpressions is to detect the common subexpression in a basic block. The common subexpressions in a basic block can be automatically detected if we construct a directed acyclic graph (DAG).

### DAG Construction

For constructing a basic block DAG, we make use of the function `node(id)`, which returns the most recently created node associated with id. For every three-address statement  $x = y \ op \ z$ ,  $x = op \ y$ , or  $x = y$  in the block we:

```
do
{
    1. If node(y) is undefined, create a leaf labeled y, and let node(y) be this node. If node(z) is
       undefined, create a leaf labeled z, and let that leaf be node(z). If the statement is of the form x =
          op y or x = y, then if node(y) is undefined, create a leaf labeled y, and let node(y) be this node.

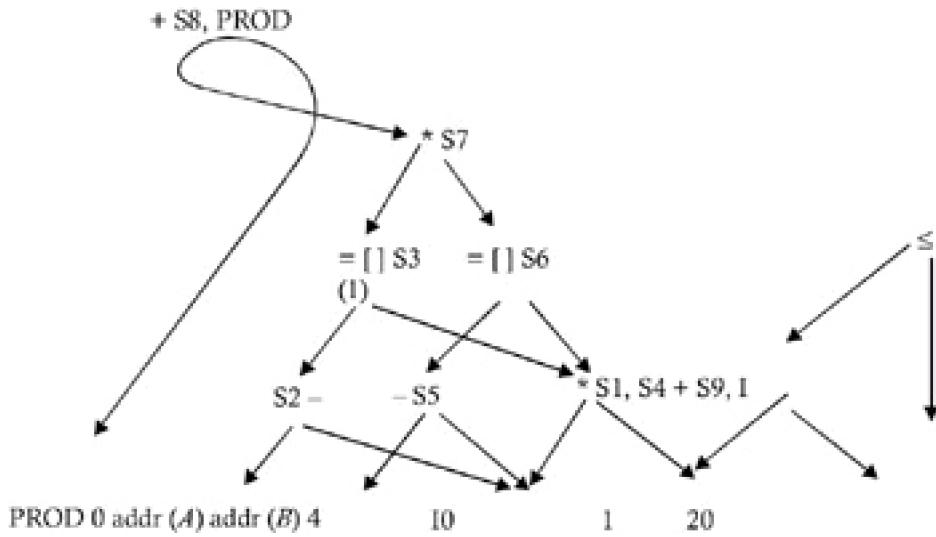
    2. If a node exists that is labeled op whose left child is node(y) and whose right child is node(z) (to
       catch the common subexpressions), then return this node. Otherwise, create such a node, and
       return it. If the statement is of the form x = op y, then check if a node exists that is labeled op
       whose only child is node(y). Return this node. Otherwise, create such a node and return. Let the
       returned node be n.

    3. Append x to the list of identifiers for the node n returned in step 2. Delete x from the list of
       attached identifiers for node(x), and set node(x) to be node n.

}
```

Therefore, we first go for a DAG representation of the basic block. And if the interior nodes in the DAG have more than one label, then those nodes of the DAG represent the common subexpressions in the basic block. After detecting these common subexpressions, we eliminate them from the basic block. The following example shows the elimination of local common subexpressions, and the DAG is shown in [Figure 10.13](#).

1. S1 := 4 \* I
2. S2 : addr(A) - 4
3. S3 : S2 [S1]
4. S4 : 4 \* I
5. S5 := addr(B) - 4
6. S6 := S5 [S4]
7. S7 := S3 \* S6
8. S8 : PROD + S7
9. PROD := S8
10. S9 := I + 1
11. I = S9
12. if I ≤ 20 goto (1).



**Figure 10.13:** DAG representation of a basic block.

In [Figure 10.13](#), PROD 0 indicates the initial value of PROD, and  $I_0$  indicates the initial value of  $I$ . We see that the same value is assigned to  $S8$  and PROD. Similarly, the value assigned to  $S9$  is the same as  $I$ . And the value computed for  $S1$  and  $S4$  are the same; hence, we can eliminate these common subexpressions by selecting one of the attached identifiers (one that is needed outside the block). We assume that none of the temporaries is needed outside the block. The rewritten block will be:

1.  $S1 := 4 * I$
2.  $S2 := \text{addr}(A) - 4$
3.  $S3 := S2 [S1]$
4.  $S5 := \text{addr}(B) - 4$
5.  $S6 := S5 [S1]$
6.  $S7 := S3 * S6$
7.  $\text{PROD} := \text{PROD} + S7$
8.  $I := I + 1$
9. if  $I \leq 20$  goto (1)

## 10.6 ELIMINATING GLOBAL COMMON SUBEXPRESSIONS

Global common subexpressions are expressions that compute the same value but in different basic blocks. To detect such expressions, we need to compute available expressions.

### 10.6.1 Available Expressions

An expression  $x \text{ op } y$  is available at a point  $p$  if every path from the initial node of the flow graph reaching to  $p$  evaluates  $x \text{ op } y$ , and if after the last such evaluation and prior to reaching  $p$  there are no subsequent assignments to  $x$  or  $y$ . To eliminate global common subexpressions, we need to compute the set of all the expressions available at the point just before the start of every block. This requires computing the set all the expressions available at a point just after the end of every block. We call these sets  $\text{IN}(b)$  and  $\text{OUT}(b)$ , respectively. The computation of  $\text{IN}(b)$  and  $\text{OUT}(b)$  requires computing the set of all expressions generated by the basic block and the set of all expressions killed by the basic block, respectively:

- A block kills an expression  $x \text{ op } y$  if it assigns to  $x$  or  $y$  and if does not subsequently recompute as  $\text{op } y$ .
- A block generates an expression  $x \text{ op } y$  if it evaluates  $x \text{ op } y$  and subsequently does not redefine  $x$  or  $y$ .

To compute the available expressions, we solve the following equations:

$$\text{OUT}(b) = \text{IN}(b) - \text{KILL}(b)$$

$$\text{IN}(b) = \cap \text{OUT}(p)$$

Here, also, we obtain the smallest solution.

The algorithm for computing the smallest  $\text{IN}(b)$  and  $\text{OUT}(b)$  is given below, where  $b_1$  is the initial block, and  $U$  is a "universal" set of all expressions appearing on the right of one or more statements of the program.

```

1.  $\text{IN}(b_1) = \emptyset$ 
    $\text{OUT}(b_1) = \text{GEN}(b_1);$ 

2. for ( $i=2; i \leq n; i++$ )
   {
       $\text{IN}(b) = U$ 
       $\text{OUT}(b) = U - \text{GEN}(b)$ 
   }

3. flag = true

4. while (flag) do
   {
      flag = false
      for ( $i=2; i \leq n; i++$ )
      {
          $\text{IN}_{\text{new}}(b_i) = \Phi$ 
         for each predecessor  $p$  of  $b_i$ 
          $\text{IN}_{\text{new}}(b_i) = \text{IN}_{\text{new}}(b_i) \cap \text{OUT}(p)$ 
         if  $\text{IN}_{\text{new}}(b_i) \neq \text{IN}(b_i)$  then
            {
               flag = true
                $\text{IN}(b_i) = \text{IN}_{\text{new}}(b_i)$ 
            }
      }
   }

```

```

    OUT( $b_i$ ) = IN( $b_i$ ) - KILL( $b_i$ )  $\cup$  GEN( $b_i$ )
}
}
```

After computing  $\text{IN}(b)$  and  $\text{OUT}(b)$ , eliminating the global common subexpressions is done as follows. For every statement  $s$  of the form  $x = y \text{ op } z$  such that  $y \text{ op } z$  is available at the beginning of the block containing  $s$ , and neither  $y$  nor  $z$  is defined prior to the statement  $x = y \text{ op } z$  in that block, do:

1. Find all definitions reaching up to the  $s$  statement block that have  $y \ op \ z$  on the right.
  2. Create a new temp.
  3. Replace each statement  $U = y \ op \ z$  found in step 1 by:

`temp = y op z  
U = temp`

4. Replace the statement  $x = y \text{ op } z$  in block by  $x = \text{temp}$ .

Team LiB

**◀ PREVIOUS** **NEXT ▶**

## 10.7 LOOP UNROLLING

Loop unrolling involves replicating the body of the loop to reduce the required number of tests if the number of iterations are constant. For example consider the following loop:

```
I = 1
while (I <= 100)
{
    X[I] = 0;
    I++;
}
```

In this case, the test  $I \leq 100$  will be performed 100 times. But if the body of the loop is replicated, then the number of times this test will need to be performed will be 50. After replication of the body, the loop will be:

```
I = 1
while(I<= 100)
{
    X[I] = 0;
    I++;
    X[I] = 0;
    I++;
}
```

It is possible to choose any divisor for the number of times the loop is executed, and the body will be replicated that many times. Unrolling once—that is, replicating the body to form two copies of the body—saves 50% of the maximum possible executions.

## 10.8 LOOP JAMMING

Loop jamming is a technique that merges the bodies of two loops if the two loops have the same number of iterations and they use the same indices. This eliminates the test of one loop. For example, consider the following loop:

```
{  
for (I = 0; I < 10; I++)  
    for (J = 0; J < 10; J++)  
        X[I,J] = 0;  
    for (I = 0; I < 10; I++)  
        X[I,I] = 1;  
}
```

Here, the bodies of the loops on  $I$  can be concatenated. The result of loop jamming will be:

```
{  
for (I = 0; I < 10; I++)  
{  
    for (J = 0; J < 10; J++)  
        X[I,J] = 0;  
    X[I,I] = 1;  
}  
}
```

The following conditions are sufficient for making loop jamming legal:

1. No quantity is computed by the second loop at the iteration  $I$  if it is computed by the first loop at iteration  $J \geq I$ .
2. If a value is computed by the first loop at iteration  $J \geq I$ , then this value should not be used by second loop at iteration  $I$ .

## Chapter 11: Code Generation

### 11.1 AN INTRODUCTION TO CODE GENERATION

Code generation is the last phase in the compilation process. Being a machine-dependent phase, it is not possible to generate good code without considering the details of the particular machine for which the compiler is expected to generate code. Even so, a carefully selected code-generation algorithm can produce code that is twice as fast as code generated by an ill-considered code-generation algorithm.

In this chapter, we first discuss straightforward code generation from a sequence of three-address statements. This is followed by a discussion of the code-generation algorithm that takes into account the flow of control structures in the program when assigning registers to names. Then we will look at a code-generation algorithm that is capable of generating reasonably good code from a basic block. Finally, various machine-dependent optimizations that are capable of improving the efficiency of object code are discussed. Throughout our discussion, we assume that the input to the code-generation algorithm is a sequence of three-address statements partitioned into basic blocks.

## 11.2 PROBLEMS THAT HINDER GOOD CODE GENERATION

There are three main difficulties that we face when attempting to generate efficient object code, namely:

1. Selection of the most-efficient instructions to represent the computation specified by the three-address statement;
2. Deciding on a computation order that leads to the generation of the more-efficient object code; and
3. Deciding which registers to use.

### Selecting the Most-Efficient Instructions to Represent the Computation Specified by the Three-Address Statement

Many machines allow certain computations to be done in more than one way. For example, if a machine permits an instruction AOS for incrementing the contents of a storage location directly, then for a three-address statement  $a = a + 1$ , it is possible to generate the instruction AOS  $a$ , rather than a sequence of instructions like the following:

```
MOVE a, R
ADD #1, R
MOVE R, a
```

Now, deciding which instruction sequence is better is the problem. This decision requires an extensive knowledge about the context in which these three-address statements will appear.

### Deciding on the Computation Order that Will Lead to the Generation of More-Efficient Object Code

Some computation orders require fewer registers to hold intermediate results than others. Now, deciding the best order is very difficult. For example, consider the basic block:

$$\begin{aligned}t1 &= a + b \\t2 &= c + d \\t3 &= e - t2 \\t4 &= t1 - t3\end{aligned}$$

If the order of computation used is the one given in the basic block  $t1-t2-t3-t4$ , then the number of registers required for holding the intermediate result is more than when the order  $t2-t3-t1-t4$  is used.

### Deciding on Registers

Deciding which register should handle the computation is another problem that stands in the way of good code generation. The problem is further complicated when a machine requires register-pairs for some operands and results.

## 11.3 THE MACHINE MODEL

Being a machine-dependent phase, we will need to describe some of the features of a typical computer in order to discuss the various issues involved in code generation. For this purpose, we describe a hypothetical machine model, as follows.

We assume that the machine is byte-addressable with two bytes per word, having  $2^{16}$  bytes, and eight general-purpose registers,  $R0$  to  $R7$ , that are capable of holding a 16-bit quantity. The format of the instruction is an op source destination with four-bit opcode, and the source and destination are each six-bit fields. Since a six-bit field is not capable of holding a memory address (a memory address is a 16-bit), when sources and destinations are memory addresses, then these six-bit fields hold certain bit patterns that specify that the words following an instruction contain memory addresses used as source and destination operands, respectively. The following addressing modes are assumed to be supported by the machine model:

1.  $r$  (register addressing)
2.  $*r$  (indirect register)
3.  $X$  (absolute address)
4.  $\#data$  (immediate)
5.  $X(r)$  (indexed address)
6.  $*X(r)$  (indirect indexed address)

We assume that opcodes like the one listed below are available:

- MOV (for moving source to destination),
- ADD (for adding source to destination), and
- SUB (for subtracting source from destination), and so on.

The cost of the instruction is considered to be its length, because generating a shorter instruction not only reduces the storage requirement of the object code, but it also reduces the time taken to perform the operation. This is because most machines spend more time fetching words from memory than they spend in executing the instruction. Hence, by minimizing the instruction length, we minimize the time taken to perform the instruction, as well.

For example, length of the instruction `MOV R0, R1` is one memory word, because, three-bit code is enough for uniquely identifying each of the registers. Therefore, the six-bit fields, each for source and destination operand, can easily hold the three-bit codes for the registers shown in [Table 11.1](#).

**Table 11.1: Six-Bit Registers for the Instruction `MOV R0, R1`**

MOV	R0	R1
-----	----	----

Similarly, the length of the instruction `MOV R0, M` is two memory words, because since the destination operand is a memory address, it will occupy the word following an instruction, as shown in [Table 11.2](#).

**Table 11.2: Six-Bit Registers for the Instruction MOV R0, R2**

MOV	R0	bit pattern
M		

Similarly, the length of the instruction MOV M1, M2 is three memory words, because the source and the destination operands, being memory addresses, will occupy the words following the instruction, as shown in [Table 11.3](#).

**Table 11.3: Six-Bit Registers for the Instruction MOV M1, M2**

MOV	bit pattern	bit pattern
M1		
M2		

For example, consider a three-address statement,  $a = b + c$ . We can generate the following different instruction sequences for this statement, depending upon where the values of operand  $b$  and  $c$  can be found.

If the values of  $b$  and  $c$  can be found in the memory locations of the same name, then the following instruction sequences can be generated:

1. MOV b, R0  
ADD c, R0  
MOV R0, a      length = six words
2. MOV b, a  
ADD c, a      length = six words

If the addresses of  $a$ ,  $b$ , and  $c$  are assumed to be in registers  $R0$ ,  $R1$ , and  $R2$ , respectively then the following instruction sequence can be generated:

3. MOV \*R1, \*R0  
ADD \*R2, \*R0      length = two words

If the values of  $b$  and  $c$  are assumed to be in registers  $R0$  and  $R1$ , respectively, then the following instruction sequence can be generated:

4. ADD R2, R1  
MOV R1, a      length = three words

Therefore, we conclude that for generating good code, we must utilize the addressing capabilities of the machine efficiently. And this will be possible if we keep the one-value or the r-value of the name in the register if it is going to be used in the future.

## 11.4 STRAIGHTFORWARD CODE GENERATION

Given a sequence of three-address statements partitioned into basic blocks, straightforward code generation involves generating code for each three-address statement in turn by taking the advantage of any of the operands of the three-address statements that are in the register, and leaving the computed result in the register as long as possible. We store it only if the register is needed for another computation or just before a procedure call, jump, or labeled statement, such as at the end of a basic block. The reason for this is that after leaving a basic block, we may go to several different blocks, or we may go to one particular block that can be reached from several others. In either case, we cannot assume that a datum used by a block appears in the same register, no matter how the program's control reached that block. Hence, to avoid possible error, our code-generation strategy stores everything across the basic block boundaries.

When generating code by using the above strategy, we need to keep track of what is currently in each register. For this, we maintain what is called a "register descriptor," which is simply a pointer to a list that contains information about what is currently in each of the registers. Initially, all of the registers are empty.

We also need to keep track of the locations for each name—where the current value of the name can be found at run time. For this, we maintain what is called an "address descriptor" for each name in the block. This information can be stored in the symbol table.

We also need a location to perform the computation specified by each of the three-address statements. For this, we make use of the function `getreg()`. When called, `getreg()` returns a location specifying the computation performed by a three-address statement. For example, if  $x = y \text{ op } z$  is performed, `getreg()` returns a location  $L$  where the computation  $y \text{ op } z$  should be performed; and if possible, it returns a register.

### Algorithm for the Function Getreg()

What follows is an algorithm for storing and returning the register locations for three-address statements by using the function `getreg()`.

```
{
For every three-address statement of the form x = y op z
in the basic block do
{
    1. Call getreg() to obtain the location L in which the computation y op z should be performed. /* This
       requires passing the three-address statement x = y op z as a parameter to getreg(), which can be
       done by passing the index of this statement in the quadruple array.

    2. Obtain the current location of the operand y by consulting its address descriptor, and if the value
       of y is currently both in the memory location as well as in the register, then prefer the register. If
       the value of y is currently not available in L, then generate an instruction MOV y, L (where y as
       assumed to represent the current location of y).

    3. Generate the instruction OP z, L, and update the address descriptor of x to indicate that x is now
       available in L, and if L is in a register, then update its descriptor to indicate that it will contain the
       run-time value of x.

    4. If the current values of y and /or z are in the register, we have no further uses for them, and they
       are not live at the end of the block, then alter the register descriptor to indicate that after the
       execution of the statement x = y op z, those registers will no longer contain y and /or z.

}
Store all the results.
}
```

The function `getreg()`, when called upon to return a location where the computation specified by the three-address

statement  $x = y \text{ op } z$  should be performed, returns a location  $L$  as follows:

1. First, it searches for a register already containing the name  $y$ . If such a register exists, and if  $y$  has no further use after the execution of  $x = y \text{ op } z$ , and if it is not live at the end of the block and holds the value of no other name, then return the register for  $L$ .
2. Otherwise, getreg() searches for an empty register; and if an empty register is available, then it returns it for  $L$ .
3. If no empty register exists, and if  $x$  has further use in the block, or  $\text{op}$  is an operator such as indexing that requires a register, then getreg() finds a suitable, occupied register. The register is emptied by storing its value in the proper memory location  $M$ , the address descriptor is updated, the register is returned for  $L$ . (The least-recently used strategy can be used to find a suitable, occupied register to be emptied.)
4. If  $x$  is not used in the block or no suitable, occupied register can be found, getreg() selects a memory location of  $x$  and returns it for  $L$ .

#### **EXAMPLE 11.1**

---

Consider the expression:

$$x = (a + b) - ((c + d) - e)$$

The three-address code for this is:

$$\begin{aligned} t1 &= a + b \\ t2 &= c + d \\ t3 &= t2 - e \\ x &= t1 - t3 \end{aligned}$$

Applying the algorithm above results in [Table 11.4](#).

---

**Table 11.4: Computation for the Expression  $x = (a + b) - ((c + d) - e)$**

Statement	L	Instructions Generated	Register Descriptor	Address Descriptor
			All registers empty	
$t1 = a + b$	R0	MOV a, R0 ADD b, R0	R0 will hold $t1$	$t1$ is in R0
$t2 = c + d$	R1	MOV c, R1 ADD d, R1	R1 will hold $t2$	$t2$ is in R1
$t3 = t2 - e$	R1	SUB e, R1	R1 will hold $t3$	$t3$ is in R1
$x = t1 - t3$	R0	SUB R1, R0	R0 will hold $x$	$x$ is in R0
		MOV R0, x		$x$ is in R0 and memory

The algorithm makes use of the next-use information of each name in order to make more-informed decisions regarding register allocation. Therefore, it is required to compute the next-use information. If:

- A statement at the index  $i$  in a block assigns a value to name  $x$ ,
- And if a statement at the index  $j$  in the same block uses  $x$  as an operand,
- And if the path from the statement at index  $i$  to the statement at index  $j$  is a path without any intervening assignment to name  $x$ , then

we say that the value of  $x$  computed by the statement at index  $i$  is used in the statement at index  $j$ . Hence, the next use of the name  $x$  in the statement  $i$  is statement  $j$ . For each three-address statement  $i$ , we need to compute information about those three-address statements in the block that are the next uses of the names coming in statement  $i$ . This requires the backward scanning of the basic block, which will allow us to attach to every statement  $i$  under consideration the information about those statements that are the next uses of each name in the statement  $i$ . The algorithm is as follows:

```
For each statement i of the form  $x = y \text{ op } z$  do
{
    attach information about the next uses of  $x$ ,  $y$ , and  $z$ 
    to statement  $i$ 
    set the information for  $x$  to no next-use /* This information
    can be kept into the symbol table */
    set the information for  $y$  and  $z$  to be the next use
    in statement  $i$ 
}
```

Consider the basic block:

$$\begin{aligned} t1 &= a + b \\ t2 &= c + d \\ t3 &= e - t2 \\ t4 &= t1 - t3 \end{aligned}$$

When straightforward code generation is done using the above algorithm, and if only two registers,  $R0$  and  $R1$ , are available, then the generated code is as shown in [Table 11.5](#).

**Table 11.5: Generated Code with Only Two Available Registers, R0 and R1**

Statement	L	Instructions Generated	Cost	Register Descriptor	Address Descriptor
				R0 and R1 empty	
$t1 = a + b$	R0	MOV a, R0 ADD b, R0	2 words 2 words	R0 will hold	$t1$ is in $t1$ R0
$t2 = c + d$	R1	MOV c, R1 ADD d, R1	2 words 2 words	R1 will hold $t2$	$t2$ is in R1
$t3 = e - t2$		MOV R0, $t1$ (generated memory by getreg())	2 words		$t1$ is in
					$t3$ is in R0
	R0	MOV e, R0 SUB R1, R0	2 words 1 word	R0 will hold $t3$ R1 will be empty because $t2$ has no next use	
$x = t1 - t3$	R1	MOV $t1$ , R1 SUB R0, R1	2 words 1 word	R1 will hold x R0 will be empty because $t3$ has no next use	x is in R1
		MOV R1, x	2 words		x is in R1 and memory

We see that the total length of the instruction sequence generated is 18 memory words. If we rearrange the final computations as:

$$\begin{aligned}
 t2 &= c + d \\
 t3 &= e - t2 \\
 t1 &= a + b \\
 t4 &= t1 - t3
 \end{aligned}$$

and then generate the code, we get [Table 11.6](#).

**Table 11.6: Generated Code with Rearranged Computations**

Statement	L	Instructions Generated	Cost	Register Descriptor	Address Descriptor
				$R0$ and $R1$ empty	
$t2 = c + d$	$R0$	MOV $c$ , $R0$ ADD $d$ , $R0$	2 words 2 words	$R0$ will hold $t2$	$t2$ is in $R0$
$t3 = e - t2$	$R1$	MOV $e$ , $R1$ SUB $R0$ , $R1$	2 words 1 word	$R1$ will hold $t3$ $R0$ will be empty because $t2$ has no next use	$t3$ is in $R1$
$t1 = a + b$	$R0$	MOV $a$ , $R0$ ADD $b$ , $R0$	2 words 2 words	$R0$ will hold $t1$	$t1$ is in $R0$
$x = t1 - t3$	$R1$	SUB $R1$ , $R0$	1 word	$R0$ will hold $x$ $R1$ will be empty because $t3$ has no next use	$x$ is in $R0$
		MOV $R0$ , $x$	2 words		$x$ is in $R0$ and memory

Here, the length of the instruction sequence generated is 14 memory words. This indicates that the order of the computation is a deciding factor in the cost of the code generated. In the above example, the cost is reduced when the order  $t2-t3-t1-t4$  is used, because  $t1$  gets computed immediately before the statement that computes  $t4$ , which uses  $t1$  as its left operand. Hence, no intermediate store-and-load is required, as is the case when the order  $t1-t2-t3-t4$  is used. Good code generation requires rearranging the final computation order, and this can be done conveniently with a DAG representation of a basic block rather than with a linear sequence of three-address statements.

## 11.5 USING DAG FOR CODE GENERATION

To rearrange the final computation order for more-efficient code-generation, we first obtain a DAG representation of the basic block, and then we order the nodes of the DAG using heuristics. Heuristics attempts to order the nodes of a DAG so that, if possible, a node immediately follows the evaluation of its left-most operand.

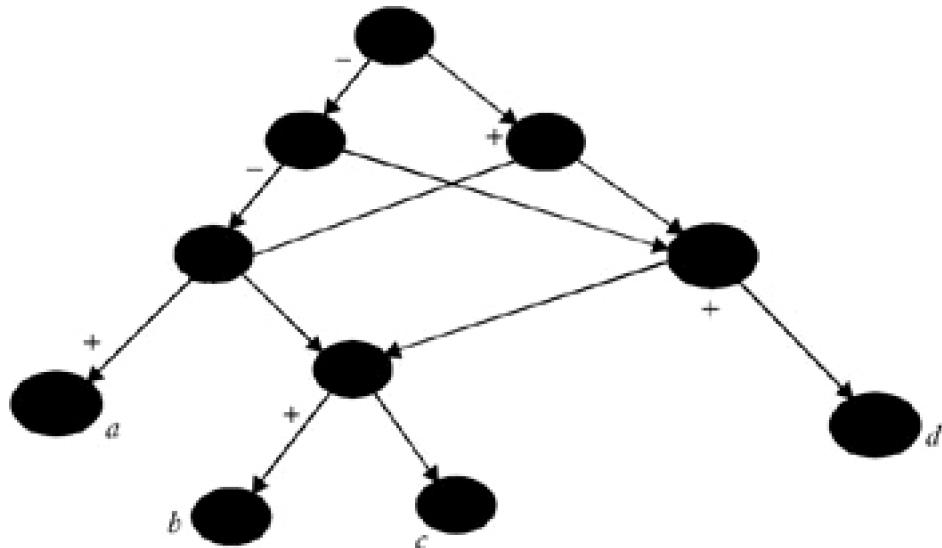
### 11.5.1 Heuristic DAG Ordering

The algorithm for heuristic ordering is given below. It lists the nodes of a DAG such that the node's reverse listing results in the computation order.

```
{
While there exists an unlisted interior node do
{
select an unlisted node  $n$  whose parents have been listed
list  $n$ 
while there exists a left-most child  $m$  of  $n$  that has no
unlisted parents and  $m$  is not a leaf do
{
list  $m$ 
 $m = n$ 
}
}
order = reverse of the order of listing of nodes
}
```

#### EXAMPLE 11.2

Consider the DAG shown in [Figure 11.1](#).



**Figure 11.1:** DAG Representation.

The order in which the nodes are listed by the heuristic ordering is shown in [Figure 11.2](#).

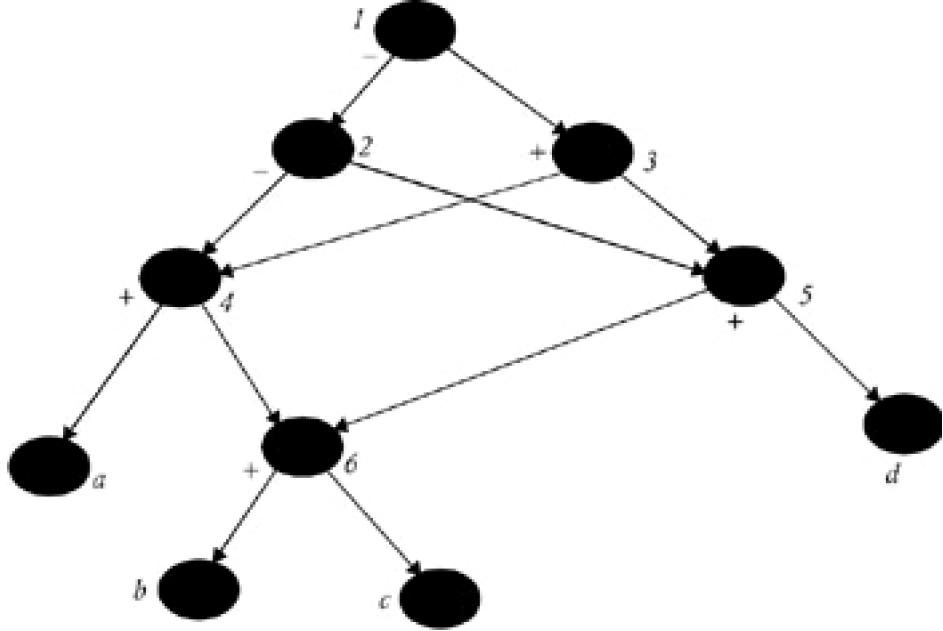


Figure 11.2: DAG Representation with heuristic ordering.

Therefore, the computation order is:

$$\begin{aligned}
 T_6 &= b + c \\
 T_5 &= t_6 + d \\
 T_4 &= a + t_6 \\
 T_3 &= t_4 + t_5 \\
 T_2 &= t_4 - t_5 \\
 T_1 &= t_2 - t_3
 \end{aligned}$$

If the DAG representation turns out to be a tree, then for the machine model described above, we can obtain the optimal order using the algorithm described in [Section 11.5.2](#), below. Here, an optimal order means the order that yields the shortest instruction sequence.

### 11.5.2 The Labeling Algorithm

This algorithm works on the tree representation of a sequence of three-address statements. It could also be made to work if the intermediate code form was a parse tree. This algorithm has two parts: the first part labels each node of the tree from the bottom up, with an integer that denotes the minimum number of registers required to evaluate the tree and with no storing of intermediate results. The second part of the algorithm is a tree traversal that travels the tree in an order governed by the computed labels in the first part, and which generates the code during the tree traversal.

```

{
if n is a leaf node then
    if n is the left-most child of its parent then
        label(n) = 1
    else
        label(n) = 0
    else
        label(n) = max[label(ni) + (i - 1)]
        for i = 1 to k
/* where n1, n2, ..., nk are the children of n, ordered by their labels; that is,
label(n1) ≥ label(n2) ≥ ... ≥ label(nk) */
}

```

For  $k = 2$ , the formula  $\text{label}(n) = \max[\text{label}(n_i) + (i - 1)]$  becomes:

$$\text{label}(n) = \max[11, 12 + 1]$$

where 11 is  $\text{label}(n_1)$ , and 12 is  $\text{label}(n_2)$ . Since either 11 or 12 will be same, or since there will be a difference of at least the difference between 11 and 12 (i.e.,  $11 - 12$ ), which is greater than or equal to one, we get:

$$\begin{aligned}\text{label}(n) &= l_1 + 1 \text{ if } l_1 = l_2 \\ &= \max(l_1, l_2) \text{ if } l_1 \neq l_2\end{aligned}$$

### EXAMPLE 11.3

---

Consider the following three-address code and its DAG representation, shown in [Figure 11.3](#):

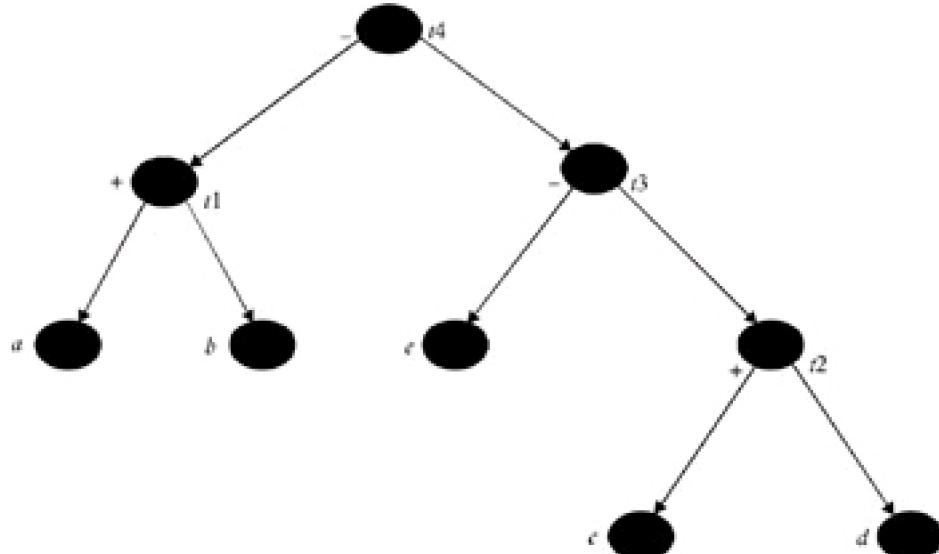
$$t1 = a + b$$

$$t2 = c + d$$

$$t3 = e - t2$$

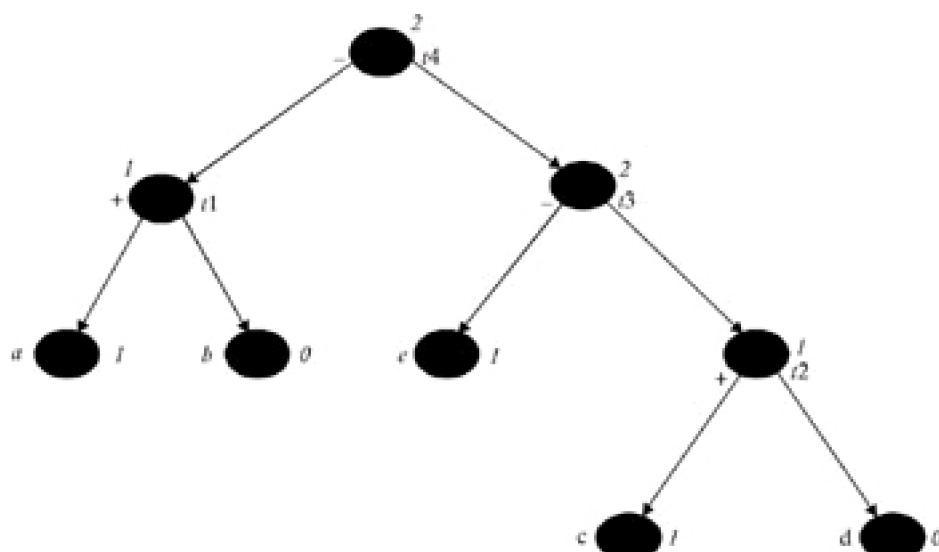
$$t4 = t1 - t3$$


---



**Figure 11.3:** DAG representation of three-address code for [Example 11.3](#).

The tree, after labeling, is shown in [Figure 11.4](#).



**Figure 11.4:** DAG representation tree after labeling.

### 11.5.3 Code Generation by Traversing the Labeled Tree

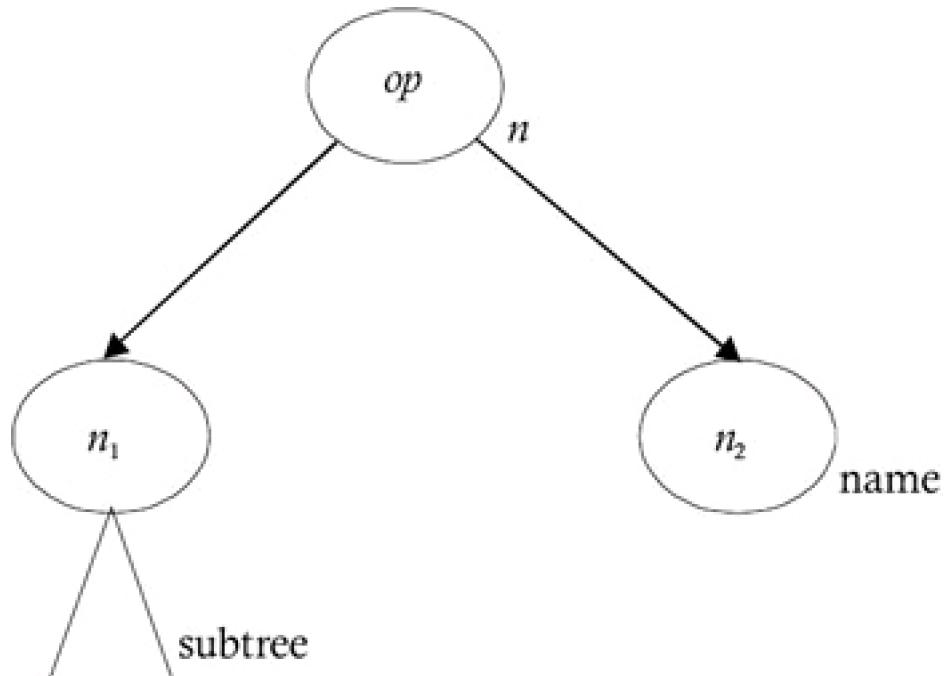
We will now examine an algorithm that traverses the labeled tree and generates machine code to evaluate the tree in the register  $R0$ . The content of  $R0$  can then be stored in the appropriate memory location. We assume that only binary operators are used in the tree. The algorithm uses a recursive procedure,  $\text{gencode}(n)$ , to generate the code for evaluating into a register a subtree that has its root in node  $n$ . This procedure makes use of RSTACK to allocate registers.

Initially, RSTACK contains all available registers. We assume the order of the registers to be  $R0, R1, \dots$ , from top to bottom. A call to  $\text{gencode}()$  may find a subset of registers, perhaps in a different order in RSTACK, but when  $\text{gencode}()$  returns, it leaves the registers in RSTACK in the same order in which they were found. The resulting code computes the value of the tree in the top register of RSTACK. It also makes use of TSTACK to allocate temporary memory locations. Depending upon the type of node  $n$  with which  $\text{gencode}()$  is called,  $\text{gencode}()$  performs the following:

1. If  $n$  is a leaf node and is the left-most child of its parent, then  $\text{gencode}()$  generates a load instruction for loading the top register of RSTACK by the label of node  $n$ :

**MOV name, RSTACK[top]**

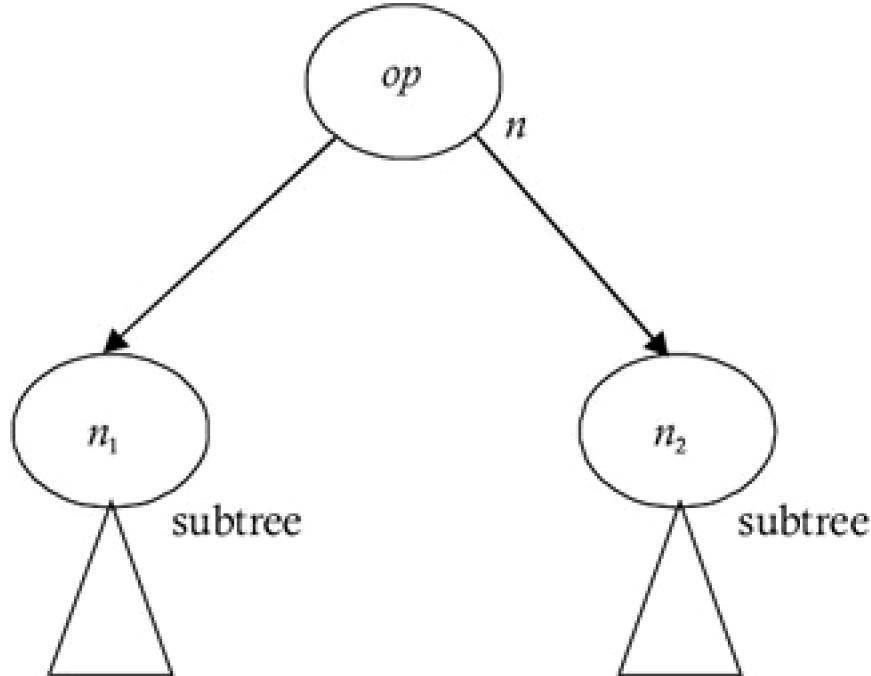
2. If  $n$  is an interior node, it will be an operator node labeled by  $op$  with the children  $n_1$  and  $n_2$ , and  $n_2$  is a simple operand and not a root of the subtree, as shown in [Figure 11.5](#).



**Figure 11.5:** The node  $n$  is an operand and not a subtree root.

In this case,  $\text{gencode}()$  will first generate the code to evaluate the subtree rooted at  $n_1$  in the top[RSTACK]. It will then generate the instruction,  $OP\ name, RSTACK[top]$ .

3. If  $n$  is an interior node, it will be an operator node labeled by  $op$  with the children  $n_1$  and  $n_2$ , and both  $n_1$  and  $n_2$  are roots of subtrees, as shown in [Figure 11.6](#).



**Figure 11.6:** The node  $n$  is an operator, and  $n_1$  and  $n_2$  are subtree roots.

In this case, gencode() examines the labels of  $n_1$  and  $n_2$ . If  $\text{label}(n_2) > \text{label}(n_1)$ , then  $n_2$  requires a greater number of registers to evaluate without storing the intermediate results than  $n_1$  does. Therefore, gencode() checks whether the total number of registers available to  $r$  is greater than the label( $n_1$ ). If it is, then the subtree rooted at  $n_1$  can be evaluated without storing the intermediate results. It first swaps the top two registers of RSTACK, then generates the code for evaluating the subtree rooted at  $n_2$ , which is harder to evaluate in RSTACK[top]. It removes the top-most register from RSTACK and stores it in  $R$ , then generates code for evaluating the subtree rooted at  $n_1$  in RSTACK[top]. An instruction, OP R, RSTACK[top], is generated, pushing  $R$  onto RSTACK. The top two registers are swapped so that the register holding the value of  $n$  will be in the top register of RSTACK.

4. If  $\text{label}(n_2) \leq \text{label}(n_1)$ , then  $n_1$  requires a greater number of register to evaluate without storing the intermediate results than  $n_2$  does. Therefore, gencode() checks whether the total number of registers available to  $r$  is greater than  $\text{label}(n_2)$ . If it is, then the subtree rooted at  $n_2$  can be evaluated without storing the intermediate results. Hence, it first generates the code for evaluating subtree rooted at  $n_1$ , which is harder to evaluate in RSTACK[top], removes the top-most register from RSTACK, and stores it in  $R$ . It then generates code for evaluating the subtree rooted at  $n_2$  in RSTACK[top]. An instruction, OP RSTACK[top], R, is generated that pushes register  $R$  onto RSTACK. In this case, the top register, after pushing  $R$  onto RSTACK, holds the value of  $n$ . Therefore, swapping and reswapping is needed.
5. If  $\text{label}(n_1)$  as well as  $\text{label}(n_2)$  are greater than or equal to  $r$  (i.e., both subtrees require  $r$  or more registers to evaluate without intermediate storage), a temporary memory location is required. In this case, gencode() first generates the code for evaluating  $n_2$  in a temporary memory location, then generates code to evaluate  $n_1$ , followed by an instruction to evaluate root  $n$  in the top register of RSTACK.

### Algorithm for Implementing Gencode()

The procedure for gencode() is outlined as follows:

```

Procedure gencode(n)
{
    if n is a leaf node and the left-most child of its parent then
        generate MOV name, RSTACK[top]
}

```

```

if  $n$  is an interior node with children  $n_1$  and  $n_2$ , with
    label( $n_2$ ) = 0 then
{
    gencode( $n_1$ )
    generate op name RSTACK[top] /* name is the operand
    represented by  $n_2$  and op is the operator
    represented by  $n^*$ /
}

if  $n$  is an interior node with children  $n_1$  and  $n_2$ ,
label( $n_2$ ) > label( $n_1$ ), and label( $n_1$ ) < r then
{
    swap top two registers of RSTACK
    gencode( $n_2$ )
     $R$  = pop(RSTACK)
    gencode( $n_1$ )
    generate op  $R$ , RSTACK[top] /* op is the operator
    represented by  $n^*$ /
    PUSH( $R$ , RSTACK)
    swap top two registers of RSTACK
}

if  $n$  is an interior node with children  $n_1$  and  $n_2$ ,
    label( $n_2$ ) <= label( $n_1$ ), and label( $n_2$ ) < r then
{
    gencode( $n_1$ )
     $R$  = pop(RSTACK)
    gencode( $n_2$ )
    generate op RSTACK[top],  $R$  /* op is the operator
    represented by  $n^*$ /
    PUSH( $R$ , RSTACK)
}

if  $n$  is an interior node with children  $n_1$  and  $n_2$ ,
label( $n_2$ ) <= label( $n_1$ ), and label( $n_1$ ) > r as well as
label( $n_2$ ) > r then
{
    gencode( $n_2$ )
     $T$  = pop(TSTACK)
    generate MOV RSTACK[top],  $T$ 
    gencode( $n_1$ )
    PUSH( $T$ , TSTACK)
    generate op  $T$ , RSTACK[top] /* op is the operator
    represented by  $n^*$ /
}
}

```

The algorithm above can be used when the DAG represented is a tree; but when there are common subexpressions in the basic block, the DAG representation will no longer be a tree, because the common subexpressions will correspond to nodes with more than one parent. These are called "shared nodes". In this case, we can apply the labeling and the gencode() algorithm by partitioning the DAG into a set of trees. We find, for each shared node as well as root  $n$ , the maximal subtree with  $n$  as a root that includes no other shared nodes, except as leaves. For example, consider the DAG shown in [Figure 11.7](#). It is not a tree, but it can be partitioned into the set of trees shown in [Figure 11.8](#). The procedure gencode() can be used to generate code for each node of this tree.

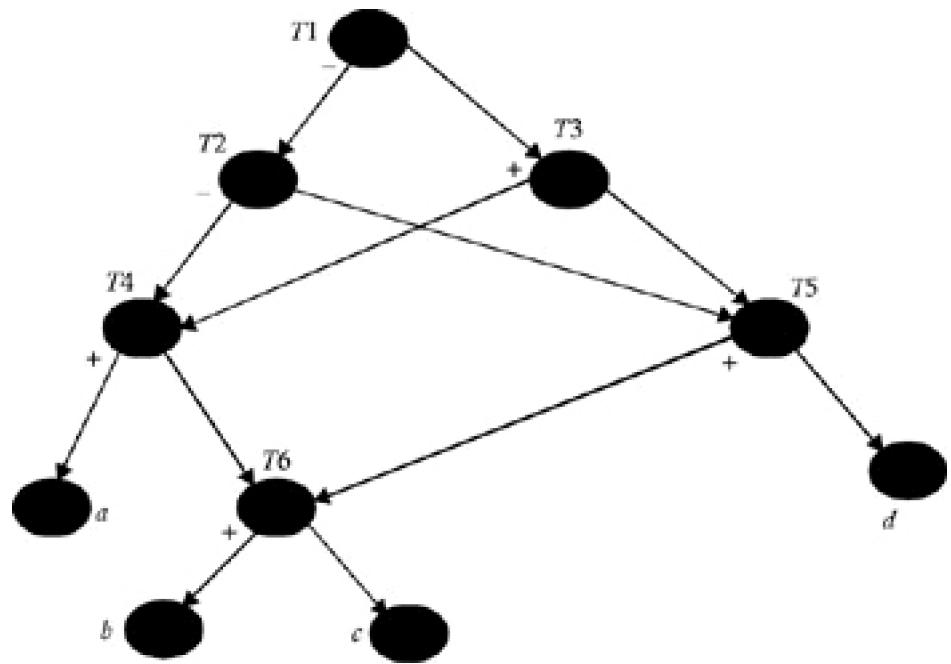


Figure 11.7: A nontree DAG.

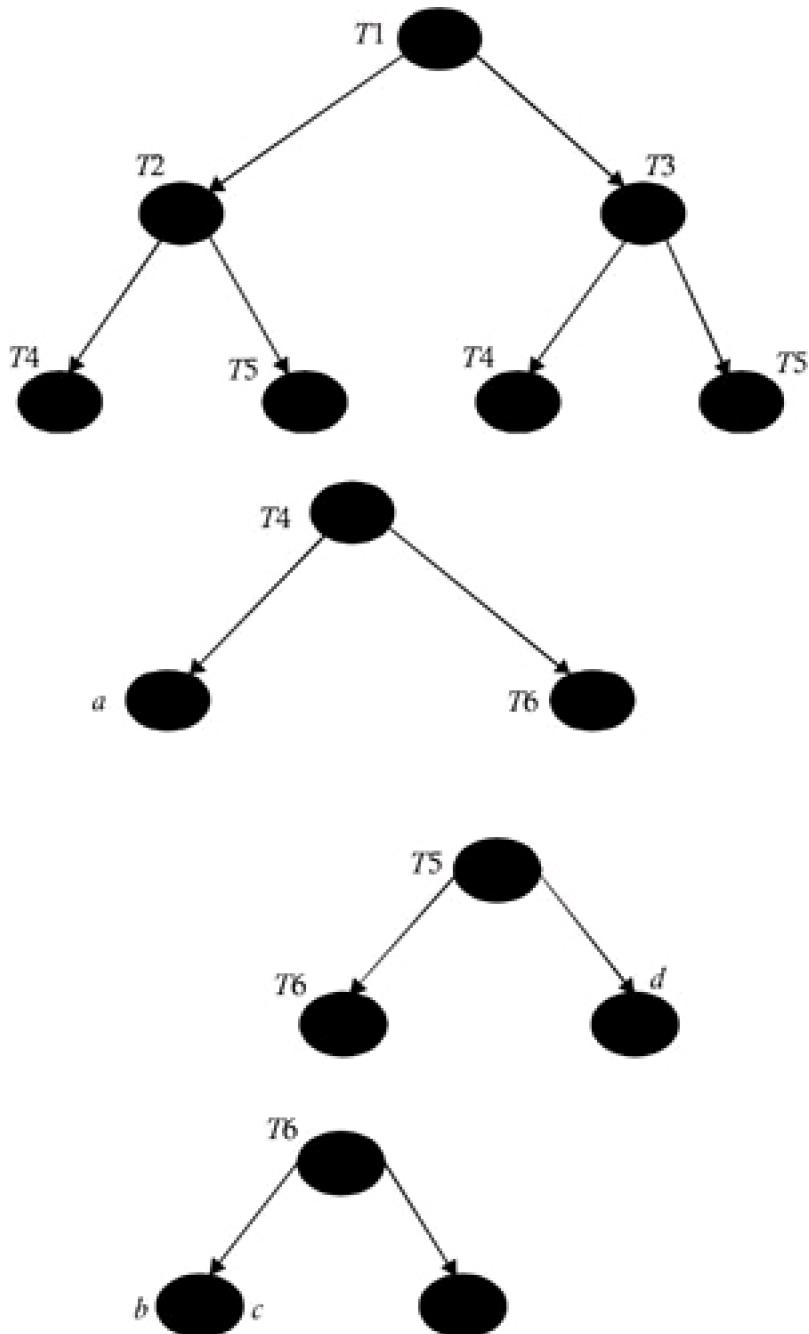


Figure 11.8: A DAG that has been partitioned into a set of trees.

---

**EXAMPLE 11.4**

---

Consider the labeled tree shown in [Figure 11.9](#).

---

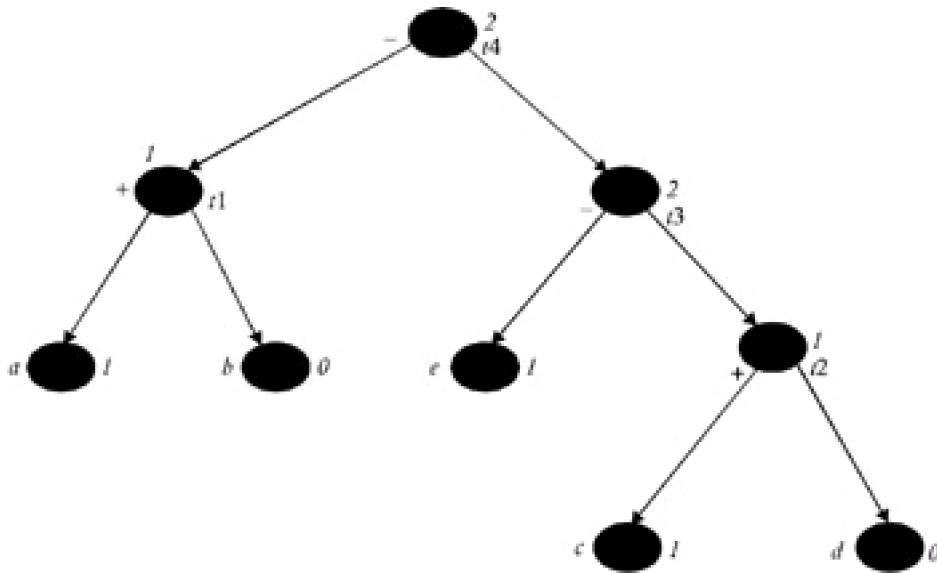


Figure 11.9: Labeled tree for [Example 11.4](#).

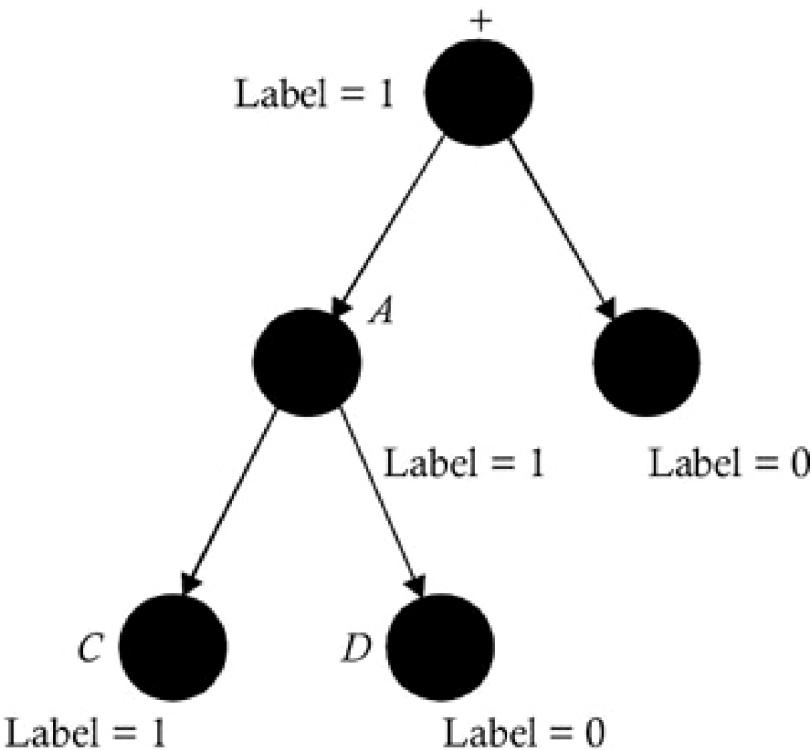
The code generated by gencode() when this tree is given as input along with the recursive calls of gencode is shown in [Table 11.7](#). It starts with call to gencode() of 4. Initially, the top two registers will be R0 and R1.

**Table 11.7: Recursive Gencode Calls**

Call to Gencode()	Action Taken	RSTACK Contents Top Two Registers	Code Generated
		R0, R1	
gencode(4)	Swap top two registers Call gencode(3) Pop R1 Call gencode(t1) Generate an instruction SUB R1,R0 Push R1 Swap top two registers	R1, R0 R0, R1  R1, R0 R0, R1	MOV E, R1 MOV C, R0 ADD D, R0 SUB R0, R1 MOV A, R0 ADD B, R0 SUB R1, R0
gencode(3)	Call gencode(E) Pop R1 Call gencode(t2) Generate an instruction SUB R0,R1 Push R1	R1, R0  R0  R1, R0	MOV E, R1 MOV C, R0 ADD D, R0 SUB R0, R1
gencode(E)	Generate an instruction MOV E, R1	R1, R0	MOV E, R1
gencode(t2)	gencode(c) Generate an instruction ADD D, R0	R0	MOV C, R0 ADD D, R0
gencode(c)	Generate an instruction MOV C, R0	R0	
gencode(t1)	gencode(A) Generate an instruction ADD B, R0	R0	MOV A, R0 ADD B, R0
gencode(A)	Generate an instruction MOV A, R0	R0	MOV A, R0

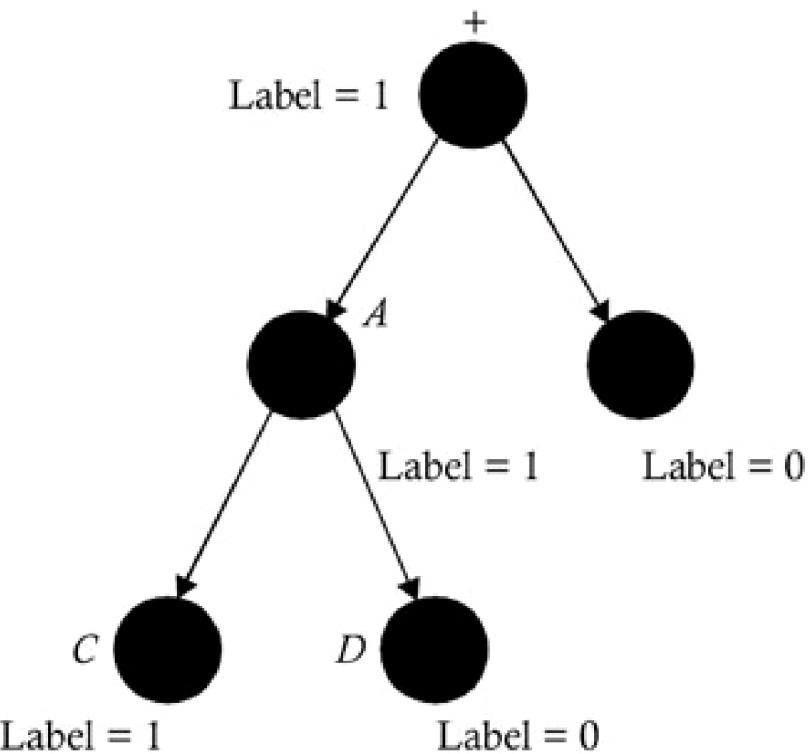
## 11.6 USING ALGEBRAIC PROPERTIES TO REDUCE THE REGISTER REQUIREMENT

It is possible to make use of algebraic properties like operator commutativity and associativity to reduce the register requirements of the tree. For example, consider the tree shown in [Figure 11.10](#).



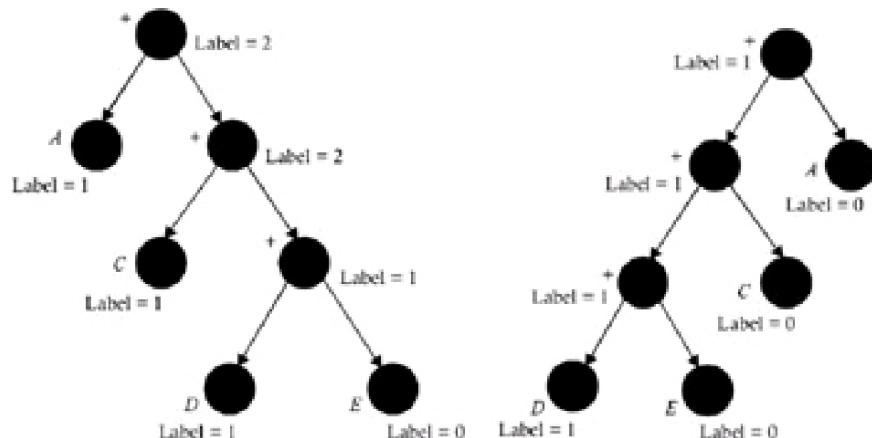
**Figure 11.10:** Tree with a label of two.

The label of the tree in [Figure 11.10](#) is two, but since  $+$  is a commutative operator, we can interchange the left and the right subtrees, as shown in [Figure 11.11](#). This brings the register requirement of the tree down to one.



**Figure 11.11:** The left and right subtrees have been interchanged, reducing the register requirement to one.

Similarly, associativity can be used to reduce the register requirement. Consider the tree shown in [Figure 11.12](#).



**Figure 11.12:** Associativity is used to reduce a tree's register requirement.

## 11.7 PEEPHOLE OPTIMIZATION

Code generated by using the statement-by-statement code-generation strategy contains redundant instructions and suboptimal constructs. Therefore, to improve the quality of the target code, optimization is required. Peephole optimization is an effective technique for locally improving the target code. Short sequences of target code instructions are examined and replacement by faster sequences wherever possible. Typical optimizations that can be performed are:

- Elimination of redundant loads and stores
- Elimination of multiple jumps
- Elimination of unreachable code
- Algebraic simplifications
- Reducing for strength
- Use of machine idioms

### Eliminating Redundant Loads and Stores

If the target code contains the instruction sequence:

1. MOV R, a
2. MOV a, R

we can delete the second instruction if it is an unlabeled instruction. This is because the first instruction ensures that the value of *a* is already in the register *R*. If it is labeled, there is no guarantee that step 1 will always be executed before step 2.

### Eliminating Multiple Jumps

If we have jumps to other jumps, then the unnecessary jumps can be eliminated in either intermediate code or the target code. If we have a jump sequence:

```
goto L1
...
L1: goto L2
```

then this can be replaced by:

```
goto L2
...
L1: goto L2
```

If there are now no jumps to *L1*, then it may be possible to eliminate the statement, provided it is preceded by an unconditional jump. Similarly, the sequence:

```
if a < b goto L1
...
L1: goto L2
```

can be replaced by:

```
if a < b goto L2
...
L1: goto L2
```

### Eliminating Unreachable Code

An unlabeled instruction that immediately follows an unconditional jump can possibly be removed, and this operation can be repeated in order to eliminate a sequence of instructions. For debugging purposes, a large program may have within it certain segments that are executed only if a debug variable is one. For example, the source code may be:

```
#define debug 0
...
if (debug)
{
    print debugging information
}
```

This if statement is translated in the intermediate code to:

**if debug = 1 goto L1**

goto L2

L1 : print debugging information

L2 :

One of the optimizations is to replace the pair:

if debug = 1 goto L1

goto L2

within the statements with a single conditional goto statement by negating the condition and changing its target, as shown below:

**if debug ≠ 1 goto L2**

Print debugging information

L2 :

Since debug is a constant zero by constant propagation, this code will become:

if 0 ≠ 1 goto L2

Print debugging information

L2 :

Since  $0 \neq 1$  is always true this will become:

goto L2

Print debugging information

L2 :

Therefore, the statements that print the debugging information are unreachable and can be eliminated, one at a time.

## Algebraic Simplifications

If statements like:

$$\begin{aligned}a &= a + 0 \\a &= a * 1\end{aligned}$$

are generated in the code, they can be eliminated, because zero is an additive identity, and one is a multiplicative identity.

## Reducing Strength

Certain machine instructions are considered to be cheaper than others. Hence, if we replace expensive operations by equivalent cheaper ones on the target machine, then the efficiency will be better. For example,  $x^2$  is invariably cheaper to implement as  $x * x$  than as a call to an exponentiation routine. Similarly, fixed-point multiplication or division by a power of two is cheaper to implement as a shift.

## Using Machine Idioms

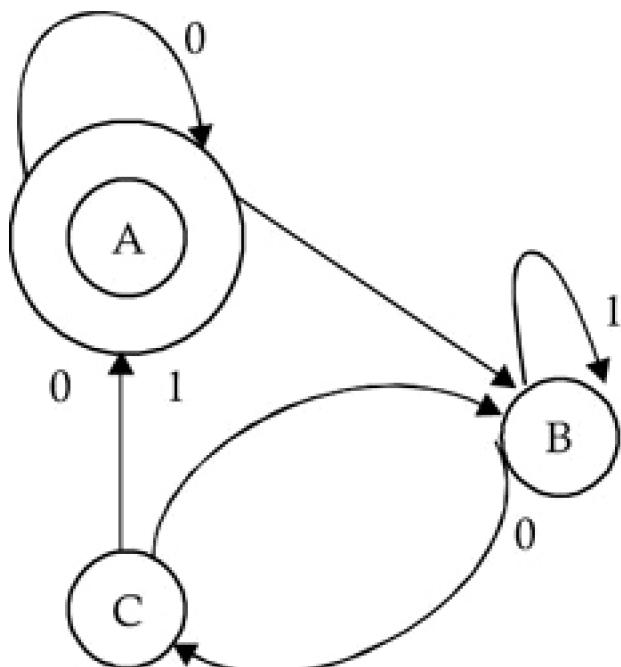
The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. Using these modes can greatly improve the quality of the code when pushing or popping a stack. These modes can also be used for implementing statements like  $a = a + 1$ .

## Chapter 12: Exercises

The exercises that follow are designed to provide further examples of the concepts covered in this book. Their purpose is to put these concepts to work in practical contexts that will enable you, as a programmer, to better and more-efficiently use algorithms when designing your compiler.

### EXERCISE 12.1

Construct the regular expression that corresponds to the state transition diagram shown in [Figure 12.1](#).



**Figure 12.1:** State transition diagram.

### EXERCISE 12.2

Prove that regular sets are closed under intersection. Present a method for constructing a DFA with an intersection of two regular sets.

### EXERCISE 12.3

Transform the following NFA into an optimal/minimal state DFA.

	0	1	$\epsilon$
A	A, C	B	D
B	B	D	C
C	C	A, C	D
D	D	A	-

---

**EXERCISE 12.4**

---

Obtain the canonical collection of sets of LR(1) items for the following grammar:

$$\begin{aligned}S &\rightarrow SA \mid Ba \\A &\rightarrow Ab \mid \epsilon \\B &\rightarrow aA \mid c\end{aligned}$$

---

---

**EXERCISE 12.5**

---

Construct an LR(1) parsing table for the following grammar:

$$\begin{aligned}T &\rightarrow \text{int} \\L &\rightarrow L, \text{id} \mid \text{id}\end{aligned}$$

---

---

**EXERCISE 12.6**

---

Construct an LALR(1) parsing table for the following grammar:

$$\begin{aligned}D &\rightarrow L : T \\L &\rightarrow L, \text{id} \mid \text{id} \\T &\rightarrow \text{integer}\end{aligned}$$

---

---

**EXERCISE 12.7**

---

Construct an SLR(1) parsing table for the following grammar:

$$\begin{aligned}S &\rightarrow A) \\S &\rightarrow A, P \mid ( P, P \\P &\rightarrow \{\text{num}, \text{num}\}\end{aligned}$$

---

---

**EXERCISE 12.8**

---

Consider the following code fragment. Generate the three-address-code for it.

```
if a < b then
    while c > d do
        x = x + y
    else
        do
            p = p + q
        while e <= f
```

---

---

**EXERCISE 12.9**

---

Consider the following code fragment. Generate the three-address code for it.

```
for (i = 1; i <= 10; i++)
```

---

**EXERCISE 12.10**

---

$S \rightarrow AE$   
 $S \rightarrow DS \text{ while}$   
 $D \rightarrow \text{do}$

---

**EXERCISE 12.12**

---

$L \rightarrow \text{elist}$   
 $\text{elist} \rightarrow \text{elist}[E] \mid [E]$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T^* F \mid F$   
 $F \rightarrow \text{id}$

---

**EXERCISE 12.13**

---

---

**EXERCISE 12.14**

---

**Team LiB**

◀ PREVIOUS | NEXT ▶

## Index

### A

Action specification in LEX, [46-47](#)

Action tables

Action | GOTO tables, [140](#)

arrays to represent, [178-179](#)

LALR parsing tables, [165-169](#)

for LR(1) parser, [163-165](#)

for SLR(1) parser, [152-161](#)

Activation records, [248-249](#)

Addressing modes, machine model and, [297-299](#)

Algebraic properties, register requirements reduced with, [317-318](#)

Alphabet, defined for lexical analysis, [6](#)

Ambiguous grammars and bottom-up parsing, [172-177](#)

AND operator and translation, [214-215](#)

Arithmetic expressions, translation of, [208-211](#)

Array references, [225-229](#)

Arrays, to represent action tables, [178-179](#)

Attributes

defined, [196](#)

dummy synthesized attributes, [199-201](#)

inherited attributes, [198-199](#)

synthesized attributes, [197-198](#)

Augmented grammars, [142-146, 175-176](#)

Automatas, equivalence of, [51-52](#)

## Index

### B

Back end compilers, [4](#)  
Back-patching, [5](#)  
Backtracking parsers, [95](#)  
    recursive descent parsers, [94-118](#)  
Block statements and stack allocation, [256-257](#)  
Boolean expressions, translation of, [211-214](#)  
Bootstrap compilers, defined, [1-2](#)  
  
Bottom-up parsing  
    Action | GOTO tables, [140](#)  
    ambiguous grammars, [172-177](#)  
    canonical collection of sets algorithm, [146-152](#)  
    defined and described, [135-136](#)  
    handles of right sentential form, [136-138](#)  
    implementation of, [138-140](#)  
    LALR parsing, [165-166](#), [190-194](#)  
    LR parsers, [140-142](#)  
    LR(1) parsing, [163-165](#), [179-194](#)  
Braces {} in syntax-directed translation schemes, [202-203](#)

## Index

### C

Call and return sequences, stack allocation and, [250-253](#)

Canonical collection of sets

algorithm for, [146-152](#)

exercises, [324](#)

of LR(1), algorithm, [161-163](#)

Cartesian products, set operation, [7](#)

CASE statements, [229-234](#)

Closure

property closure of a relation, [9](#)

set operation, [7-8](#)

Closure operations, regular sets and, [47](#)

Code generation phase, [2, 3, 4](#)

DAGs and, [305-316](#)

difficulties encountered during, [296-297](#)

getreg() function and, [300-305](#)

labeled trees and, [307-316](#)

straightforward strategy for, [299-305](#)

Code optimization phase, [2, 3](#)

algebraic properties to reduce register requirements, [317-318](#)

algebraic simplifications, [320](#)

defined and described, [269-270](#)

global common subexpressions, eliminating, [290-292](#)

jumps, eliminate multiple, [319](#)

loads and stores, eliminating redundancy, [319](#)

local common subexpressions, eliminating, [288-290](#)

loop optimization, [270-284](#)

machine idioms and, [321](#)

partitioning three-address code into basic blocks, [271-273](#)

peephole optimization, [318-321](#)

reducible flow graphs and, [274-284](#)

strength reduction, [321](#)

unreachable code, eliminating, [319-320](#)

Compilation, process described, [2-5](#)

Compilers

defined, [1](#)

front-end vs. back-end compilers, [4](#)

organization of, [4](#)

Computational order, [296](#)

Concatenation

defined, [6](#)

set operation, [7](#)

Concatenation operation, regular sets and, [47](#)

Context-free grammars (CFGs)

algorithm for identifying useless symbols, [64](#)

defined and described, [54](#)  
derivation in, [55-56](#)  
 $\epsilon$  -productions and, [70-73](#)  
left linear grammar, [86-90](#)  
left-recursive grammar, [75-77](#)  
productions (P) in, [54](#)  
reduction of grammar, [61-70](#)  
regular grammar as, [77-85](#)  
right linear grammar, [85-86](#)  
SLR(1) grammars, [152-161](#)  
start symbol (S) in, [54](#)  
in syntax analysis phase, [53-54](#)  
terminals (T) in, [54](#)  
unit productions and, [73-75](#)  
variables (V) or nonterminals in, [54, 56](#)

Cross-compilers, defined, [1-2](#)

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

## Index

### D

- DAGs. See [Directed acyclic graphs \(DAGs\)](#)
- Data storage. See [Storage management](#)
- Data structures for representing parsing tables, [178-179](#)
- Dead states of DFAs, [27](#)  
detection of, [31](#)
- Decrement operators, implementation of, [224-225](#)
- Dependency graphs, [199-201](#)
- Derivation  
in context-free grammar, [55-56](#)  
derivation trees in CFG, [56-61](#)
- Detection, of DFA unreachable and dead states, [28-31](#)
- Deterministic finite automata (DFA)  
Action | GOTO tables, [141-142](#)  
augmented grammar and, [142-146](#)  
equivalent to NFAs with  $\epsilon$ -moves, [23-27](#)  
exercises, [323-324](#)  
minimization of, [27-31](#)  
minimization/optimization of, [27-31](#)  
transforming NFAs into, [16-18](#)
- DFA. See [Deterministic finite automata \(DFA\)](#)
- Directed acyclic graphs (DAGs), [288-290](#)  
code generation and, [305-316](#)  
heuristic DAG ordering, [305-307](#)  
labeling algorithm and, [307-309](#)
- DO-WHILE statements and translation, [220-221](#)
- Dummy synthesized attributes, [199-201](#)

## Index

### E

$\in$ -closure( $q$ ), finding, [19-20](#)  
 $\in$ -moves  
    acceptance of strings by NFAs with, [19](#)  
    equivalence of NFAs with and without, [21-22](#)  
    finding  $\in$ -closure( $q$ ), [19-20](#)  
    NFAs with, [18-27](#)  
 $\in$ -productions  
    defined, [70](#)  
    eliminating, [71-73](#)  
    and nonnullable nonterminals, [70-71](#)  
    regular grammar and, [77-84](#)  
 $\in$ -transitions, [18](#)  
Equivalence of automata, [51-52](#)

### Error handling

detection and report of errors, [259-260](#)  
exercises, [325](#)  
lexical phase errors, [260](#)  
in LR parsing, [261-264](#)  
panic mode recovery, [261](#)  
phase level recovery, [261-264](#)  
predictive parsing error recovery, [264-267](#)  
semantic errors and, [268](#)  
YACC and, [264](#)

Errors. See [Error handling](#)

## Index

### F

Finite automata

construction of, [31-38](#)

defined, [11](#)

exercises, [326](#)

non-deterministic finite automata (NFA), [14-16](#)

specification of, [11-14](#)

strings and, [13, 15-16](#)

FOR statements and translation, [223-224](#)

Front-end compilers, [4](#)

## Index

### G

- Gencode() function, [313-316](#)
- Getreg() function, [300-305](#)
- Global common subexpressions, eliminating, [290-292](#)
- GOTO tables, [140](#)
  - construction of, [152-161](#)
  - for LR(1) parser, [163-165](#)
- Grammars, exercises
  - ambiguous grammars, [172-177](#)
  - augmented grammar, [142-146](#), [175-176](#)
  - left-recursive grammar, [75-77](#)
  - useless grammar symbols (reduction of), [61-70](#)

## Index

### H

Handle pruning, [137](#)

Hash tables for organization of symbol tables, [243-244](#)

## Index

I

IF-THEN-ELSE statements and translation, [216-218](#)

IF-THEN statements and translation, [218-219](#)

Increment operators, implementation of, [224-225](#)

Indirect triple representation, [206-207](#)

Induction variables of loops

    defined, [284-285](#)

    detecting and eliminating, [285-288](#)

Inherited attributes, [198-199](#)

Input files, LEX, [46-47](#)

Intermediate code generation phase, [2](#), [3](#)

Intersection, set operation, [7](#)

## Index

### J-K

Jumps

    and Boolean translation, [213-214](#)

    eliminating multiple, [319](#)

## Index

### L

- LALR parsing, [165-166](#), [190-194](#)
- Language, defined for lexical analysis, [6](#)
- Language tokens, lexical analysis and, [5](#)
- L-attributed definitions, [201](#)
- Left linear grammar, [86-90](#)
- LEX compiler-writing tool, [45-46](#)
  - action specification in, [46-47](#)
  - format for input or source files, [46-47](#)
  - pattern specification in, [46-47](#)
- Lexemes, [5](#)
- Lexical analysis
  - design of lexical analyzers, [45-47](#)
  - phase of compiling, [2-3](#), [5](#), [260](#)
- Lexical analyzers, design of, [45-47](#)
- Lexical phase, [2-3](#), [5](#)
  - error recovery, [260](#)
- Linear lists for organization of symbol tables, [242](#)
- Local common subexpressions, eliminating, [288-290](#)
- Logical expressions
  - AND operator, [214-215](#)
  - DO-WHILE statements, [220-221](#)
  - FOR statements, [223-224](#)
  - IF-THEN-ELSE statements, [216-218](#)
  - IF-THEN statements, [218-219](#)
  - NOT operator, [215-216](#)
  - OR operator, [215](#)
  - REPEAT statements, [222-223](#)
  - translation and, [214-224](#)
  - WHILE statements, [219-220](#)
- Loop invariant computations, [271](#)
- Loop jamming, [293-294](#)
- Loop optimizations, [270-284](#)
  - back edge identification, [273-274](#)
  - induction variables, reduction of, [284-288](#)
  - loop detection, [273](#)
  - loop jamming, [293-294](#)
  - loop unrolling, [292-293](#)
  - reducible flow graphs and, [274-284](#)
- Loop unrolling, [292-293](#)
- LR parsers and parsing, [140-142](#), [179-194](#)
- LR(1) parsers and parsing
  - action tables, [163-165](#)
  - exercises, [324](#)

**Team LiB**

◀ PREVIOUS | **NEXT ▶**

## Index

### M

Machine model described, [297-299](#)

Memory. See [Storage management](#)

Memory addresses, machine model and, [297-299](#)

## Index

### N

#### Names

access to nonlocal names, [253-255](#)

address descriptors and, [299](#)

held in symbol tables, [241](#)

runtime name storage, [241](#)

scope of name, [244-246](#)

#### Non-deterministic finite automata (NFA)

defined and described, [14](#)

DFA equivalents of, [23-27](#)

with  $\in$ -moves, [18-27](#)

equivalence and  $\in$ -moves, [21-22](#)

strings and, [15-16](#)

transformation into deterministic (DFA), [16-18](#)

#### Nondistinguishable states of DFAs, [27](#)

#### Nonlocal names, [253-255](#)

#### Nonterminals in context-free grammar, [54, 56](#)

#### NOT operator and translation, [215-216](#)

## Index

### O

Opcodes, machine model and, [297-299](#)

Operators

for regular expressions, [40](#)

translation and Boolean operators, [214-216](#)

Optimizations

of DFAs, [27-31](#) see also [Code optimization phase](#)

OR operator and translation, [215](#)

## Index

### P

Panic mode recovery, [261](#)

Parsers and parsing

action tables, [140](#)

backtracking parsers, [95](#)

conflicts, [169-171](#)

data structures for representing parsing tables, [178-179](#)

defined and described, [91](#)

LALR parsing, [165-169](#), [190-194](#)

LR parsers, [140-142](#)

LR(1) parsers, action tables, [163-165](#)

predictive top-down parsers, [118-133](#)

table-driven predictive parsers, [123-133](#)

see also [Bottom-up parsing](#); [Parse trees](#); [Syntax analysis phase](#); [Top-down parsing](#)

Parse trees

in CFG, [56-61](#)

derivation trees in CFG, [56-61](#)

labeled trees and code generation, [307-316](#)

node labeling algorithm, [307-309](#)

symbol table organization with, [242-243](#)

syntax trees, [203-204](#)

Pattern specification in LEX, [46-47](#)

Peephole optimization, [318-321](#)

Postfix notation, [203](#)

Power set, set operation, [7](#)

Predictive parsing

error recovery and, [264-267](#)

predictive top-down parsers, [118-133](#)

Predictive top-down parsers, [118-133](#)

Prefixes, defined, [6](#)

Procedure calls, [234-235](#)

Productions (P) in context-free grammar, [54](#)

## Index

### **Q**

Quadruple representation, [205-206](#), [207](#)

## Index

### R

- Recursion, eliminating left recursion, [75-77](#)
- Recursive descent parsers, implementation, [94-118](#)
- Reduce-reduce conflicts, [170-171](#)
- Reducible flow graphs
  - and code optimization, [274-284](#)
  - loop invariant statements and, [282-283](#)
- Reduction of grammar, [61-70](#)
  - algorithm for identifying useless symbols, [64](#)
  - bottom-up parsing and, [135-136](#)
- Registers
  - algebraic properties to reduce requirements for, [317-318](#)
  - register descriptors, [299](#)
  - RSTACK to allocate, [309-313](#)
  - selecting for computation, [297](#)
- Regular expression notation
  - finite automata definitions, [6-8](#)
  - role in lexical analysis, [5](#)
- Regular expressions
  - defined and described, [39-43](#)
  - exercise, [323](#)
  - lexical analyzer design and, [45](#)
  - obtained from finite automata, [43-44](#)
  - obtained from regular grammar, [84-85](#)
  - operators for, [40](#) see also [Regular expression notation](#)
- Regular grammar, [77-85](#)
  - defined, [77](#)
  - $\in$ -productions and, [77](#)
  - regular expressions from, [84-85](#)
- Regular sets, [39](#)
  - exercises, [326](#)
  - lexical analyzer design and, [45](#)
  - properties of, [47-51](#)
- Relations
  - defined and described, [8](#)
  - properties of, [8-9](#)
  - property closure of, [9](#)
  - symbol for in CFG, [54](#)
- REPEAT statements and translation, [222-223](#)
- Return sequences, stack allocation and, [250-253](#)
- Right linear grammar, [85-86](#)
- RSTACKs, allocating registers with, [309-313](#)

## Index

### S

- Scope rules and scope information, [244-246](#), [253](#)
- Search trees for organization of symbol tables, [242-243](#)
- Sentential form handles, [136-138](#)
- Set difference, set operation, [7](#)
- Set operations, defined, [7](#)
- Sets
  - defined, [7](#)
  - regular sets, [39](#), [45](#), [47-51](#)
  - relations between, [8-9](#)
- Shift-reduce conflicts, [169](#)
- SLR(1)
  - exercises, [324](#)
  - grammars, [152-161](#)
- SLR parsing, [151-162](#), [176-177](#), [180-190](#)
- Source files, LEX, [46-47](#)
- Stack allocation
  - access link set up, [255-257](#)
  - access to nonlocal names and, [253-255](#)
  - block statements and, [256-257](#)
  - call and return sequences, [250-253](#)
- Start symbol (S) in context-free grammar, [54](#)
- Storage management
  - heap memory storage, [247-248](#)
  - procedure activation and activation records, [248-249](#)
  - stack allocation, [250-257](#)
  - static allocation, [250](#)
  - storage allocation, [247-248](#)
- Strings, defined, [6](#)
- Suffixes, defined, [6](#)
- SWITCH statements, translation of, [229-234](#)
- Symbol tables
  - defined and described, [239](#)
  - exercises, [326](#)
  - hash tables for organization of, [243-244](#)
  - implementation of, [239-240](#)
  - information entry for, [240](#)
  - linear lists for organization of, [242](#)
  - names held in, [241](#)
  - scope information, [244-246](#)
  - search trees for organization of, [242-243](#)
- Syntactic phase error recovery, [260-261](#)

Syntax analysis phase, [2-3](#)

    context-free grammar and, [53-54](#)

    error recovery during syntactic phase, [260-261](#)

Syntax-directed definitions

    L-attributed definitions, [201](#)

    translation and, [195-201](#)

Syntax directed translations and translation schemes, [202-203](#)

Syntax trees, [203-204](#)

Synthesized attributes, [197-198](#)

    dummy synthesized attributes, [199-201](#)

## Index

### T

Table-driven predictive parsers, implementation, [123-133](#)

Terminals (T) in context-free grammar, [54](#)

Three-address code, [204-205](#)

exercises, [324-325](#)

partitioning into basic blocks, [271-273](#)

Three-address statements, representation of, [205-207](#), [296](#)

Tokens, lexical analysis and, [5](#)

Top-down parsing

defined and described, [91-92](#)

exercises, [326](#)

implementation, [94-118](#)

predictive top-down parsers, [118-133](#)

Translations and translation schemes

of arithmetic expressions, [208-211](#)

of array references, [225-229](#)

of Boolean expressions, [211-214](#)

of decrement and increment operators, [224-225](#)

examples of, [235-238](#)

exercises, [325](#)

intermediate code generation and, [203-205](#)

of logical expressions, [214-224](#)

procedure calls and, [234-235](#)

specification of, [195-196](#)

of SWITCH / CASE statements, [229-234](#)

syntax-directed definitions, [195-201](#)

Trees. See [Parse trees](#)

Triple representation, [206](#)

## Index

### U

- Union set operation, [7](#)
  - and regular sets, [47](#)
- Unit productions defined, [73](#)
  - elimination of, [73-75](#)
- Unreachable states of DFAs, [27](#)
  - detecting, [28-31](#)

## Index

### V

Variables (V) in context-free grammar, [54](#), [56](#)

## Index

### W-X

WHILE statements and translation, [219-220](#)

## Index

### Y-Z

YACC, error handling and, [264](#)

## List of Figures

### Chapter 1: Introduction

[Figure 1.1:](#) Compilation process phases.

[Figure 1.2:](#) Syntax analysis imposes a structure hierarchy on the token string.

### Chapter 2: Finite Automata and Regular Expressions

[Figure 2.1:](#) Transition diagram for finite automata  $\delta(p, a) = q$ .

[Figure 2.2:](#) Transition diagram for finite automata that handles several transitions.

[Figure 2.3:](#) Transition diagram for  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\} \delta, q_0, \{q_3\})$ .

[Figure 2.4:](#) Finite automata with  $\epsilon$ -moves.

[Figure 2.5:](#) Transitioning from an  $\epsilon$ -move NFA to a non- $\epsilon$ -move NFA.

[Figure 2.6:](#) Making the initial state of the NFA one of the final states.

[Figure 2.7:](#) Example 2.1 NFA.

[Figure 2.8:](#) Example 2.2 DFA equivalent to an NFA.

[Figure 2.9:](#) Partitioning down to a single state.

[Figure 2.10:](#) Merging nondistinguishable states B and C into a single state B1.

[Figure 2.11:](#) Transition diagram for Example 2.3 finite automata.

[Figure 2.12:](#) Finite automata containing even number of zeros and odd number of ones.

[Figure 2.13:](#) Finite automata containing odd number of zeros and even number of ones.

[Figure 2.14:](#) Example 2.6 finite automata considers the set prefix.

[Figure 2.15:](#) Finite automata accepts strings containing the substring 101.

[Figure 2.16:](#) DFA using the names A-D and q0–5.

[Figure 2.17:](#) Complement to Figure 2.16 automata.

[Figure 2.18:](#) DFA after minimization.

[Figure 2.19:](#) Finite automata that accepts string decimals that are divisible by three.

[Figure 2.20:](#) Finite automata accepts strings containing 101.

[Figure 2.21:](#) Finite automata identified by the name states A-D and q0–5.

[Figure 2.22:](#) Complement to Figure 2.21 automata.

[Figure 2.23:](#) Minimization of nondistinguishable states of Figure 2.22.

[Figure 2.24:](#) Automata that accepts binary strings that are divisible by three.

[Figure 2.25:](#) Transition diagram for  $(a + b)$ .

[Figure 2.26](#): Transition diagram for  $(a + b)^*$ .

[Figure 2.27](#): Transition diagram for  $a.(a + b)^*$ .

[Figure 2.28](#): Automata for  $a.(a + b)^* . b$ .

[Figure 2.29](#): Automata for  $a.(a + b)^*.b.b$ .

[Figure 2.30](#): Deriving the regular expression for a regular set.

[Figure 2.31](#): Transition diagram.

[Figure 2.32](#): Complement to transition diagram in Figure 2.31.

[Figure 2.33](#): Transition diagram of automata M<sub>1</sub>.

[Figure 2.34](#): Transition diagram of automata M<sub>2</sub>.

## Chapter 3: Context-Free Grammar and Syntax Analysis

[Figure 3.1](#): Derivation tree for the string id + id \* id.

[Figure 3.2](#): Parse tree resulting from leaf-node concatenation.

[Figure 3.3](#): Multiple parse trees.

[Figure 3.4](#): Ambiguous grammar parse trees.

[Figure 3.5](#): Parse tree generated by using both the right- and left-most derivation orders.

[Figure 3.6](#): Parse tree generated from both the left- and right-most orders of derivation.

[Figure 3.7](#): Transition diagram for automata that accepts the regular grammar of Example 3.13.

[Figure 3.8](#): Deterministic equivalent of the non-deterministic automata shown in Figure 3.7.

[Figure 3.9](#): Non-deterministic automata.

[Figure 3.10](#): Transition diagram for deterministic automata equivalent shown in Figure 3.9.

[Figure 3.11](#): Regular-grammar automata.

[Figure 3.12](#): Transition diagram of automata that accepts  $L(G_1)$ .

[Figure 3.13](#): Transition diagram of automata after removal of state D.

[Figure 3.14](#): Transition diagram for the automata that results from merged states.

[Figure 3.15](#): Non-deterministic automata that accepts  $L(G_2)$ .

[Figure 3.16](#): Transition diagram of the equivalent deterministic automata for Figure 3.15.

[Figure 3.17](#): Finite automata accepting the right linear grammar for a regular expression.

[Figure 3.18](#): Transition diagram for a finite automata specified by a reversed regular expression.

## Chapter 4: Top-Down Parsing

[Figure 4.1](#): Parser uses the S-production to expand the parse tree.

[Figure 4.2](#): Parser uses the first alternative for A in order to expand the tree.

[Figure 4.3](#): If the parser fails to match a leaf, the point of failure, d, reroutes (backtracks) the pointer to

alternative paths from A.

[Figure 4.4](#): The parser first expands S and fails to accept  $w = acdb$ .

[Figure 4.5](#): The parser advances to c and considers nonterminal A for expansion.

[Figure 4.6](#): The parser first expands S.

[Figure 4.7](#): The parser advances the pointer to a second occurrence of a.

[Figure 4.8](#): The parser expands the next leaf labeled S.

[Figure 4.9](#): The parser finds no match, so it backtracks.

[Figure 4.10](#): The parser tries an alternate aa.

[Figure 4.11](#): There is no further alternate of S that can be tried, so the parser will backtrack one more step.

[Figure 4.12](#): The parser again finds a mismatch; hence, it backtracks.

[Figure 4.13](#): The parser tries an alternate aa.

[Figure 4.14](#): Since no alternate of S remains to be tried, the parser backtracks one more step.

[Figure 4.15](#): The parser tries an alternate aa.

[Figure 4.16](#): The parser arrives at the required parse tree.

[Figure 4.17](#): The parser first expands S.

[Figure 4.18](#): The parser advances the pointer to a second occurrence of a.

[Figure 4.19](#): The parser considers the next leaf labeled by S.

[Figure 4.20](#): The parser matches the third input symbol and moves on to the next leaf labeled by S.

[Figure 4.21](#): The parser considers the fourth occurrence of the input symbol a.

[Figure 4.22](#): The parser finds no match, so it backtracks.

[Figure 4.23](#): The parser tries an alternate aa.

[Figure 4.24](#): No alternate of S can be tried, so the parser will backtrack one more step.

[Figure 4.25](#): Again finding a mismatch, the parser backtracks.

[Figure 4.26](#): The parser then tries an alternate.

[Figure 4.27](#): No alternate of S remains to be tried, so the parser will backtrack one more step.

[Figure 4.28](#): The parser again finds a mismatch; therefore, it backtracks.

[Figure 4.29](#): The parser tries an alternate aa.

[Figure 4.30](#): The parser then tries an alternate aa.

[Figure 4.31](#): The parser successfully generates the parse tree for  $aaaa$ .

[Figure 4.32](#): The parser expands S.

[Figure 4.33](#): The parser matches the first symbol, advances to the second occurrence of a, and considers S for expansion.

[Figure 4.34](#): The parser finds a match for the second occurrence of a and expands S.

[Figure 4.35](#): The parser matches the third input symbol, considers the next leaf, and expands S.

[Figure 4.36](#): The parser matches the fourth input symbol, considers the next leaf, and expands S.

[Figure 4.37](#): A match is found for the fifth input symbol, so the parser considers the next leaf, and expands S.

[Figure 4.38](#): The sixth input symbol also matches. So the next leaf is considered, and S is expanded.

[Figure 4.39](#): No match is found, so the parser backtracks to S.

[Figure 4.40](#): The parser backtracks one more step.

[Figure 4.41](#): The parser tries the alternate aa.

[Figure 4.42](#): Again, a mismatch is found. So, the parser backtracks.

[Figure 4.43](#): No alternate of S remains, so the parser will back-track one more step.

[Figure 4.44](#): The parser tries an alternate aa.

[Figure 4.45](#): Again, a mismatch is found. The parser backtracks.

[Figure 4.46](#): The parser then tries an alternate aa.

[Figure 4.47](#): A mismatch is found, and the parser backtracks.

[Figure 4.48](#): The parser tries for the alternate aa, fails to find a match, and cannot generate the parse tree for six occurrences of a.

## Chapter 5: Bottom-up Parsing

[Figure 5.1](#): NFA transition diagram recognizes viable prefixes.

[Figure 5.2](#): Using subset construction, a DFA equivalent is derived from the transition diagram in Figure 5.1.

[Figure 5.3](#): DFA transition diagram showing four iterations for a canonical collection of sets.

[Figure 5.4](#): Transition diagram for Example 5.2 DFA.

[Figure 5.5](#): DFA Transition diagram.

[Figure 5.6](#): Transition diagram for the canonical collection of sets of LR(1) items.

[Figure 5.7](#): Transition diagram for a DFA using a reduced collection.

[Figure 5.8](#): LR(0) underlying set representations that can cause SLR parser conflicts.

[Figure 5.9](#): LR(1) underlying set representations that can cause CLR/LALR parser conflicts.

[Figure 5.10](#): LR(0) underlying set representations that can cause an SLR parser *reduce-reduce conflict*.

[Figure 5.11](#): LR(1) underlying set representations that can cause an CLR/LALR parser.

[Figure 5.12](#): Sets of LR(1) items represent two different CLR(1) parser states.

[Figure 5.13](#): States are combined to form an LALR.

[Figure 5.14](#): LR(1) items represent two different states of the CLR(1) parser.

[Figure 5.15](#): LALR state set resulting from the combination of CLR(1) state sets.

[Figure 5.16](#): Transition diagram for augmented grammar DFA.

[Figure 5.17](#): States with actions in common point to the same location via an array.

[Figure 5.18](#): List that incorporates the ability to append actions.

[Figure 5.19](#): Transition diagram for the canonical collection of sets of LR(0) items in Example 5.3.

[Figure 5.20](#): DFA transition diagram for Example 5.4.

[Figure 5.21](#): Collection of nonempty sets of LR(1) items for Example 5.7.

## **Chapter 6: Syntax-Directed Definitions and Translations**

[Figure 6.1](#): The attribute value of node X is inherently dependent on the attribute value of node Y.

[Figure 6.2](#): An annotated parse tree.

[Figure 6.3](#): Parse tree with node attributes for the string int id1,id2,id3.

[Figure 6.4](#): Dependency graph with four nodes.

[Figure 6.5](#): Parse tree for the string id+id\*id.

[Figure 6.6](#): Syntax tree for id+id\*id.

[Figure 6.7](#): Values of attributes at the parse tree node for the string  $a + b * c$ .

[Figure 6.8](#): Values of the attributes at the parse tree nodes for  $a + b * c$ , id.place = addr(symtab rec of a).

[Figure 6.9](#): Values of the attributes at the parse tree nodes for  $a + b * c$ , id.place = addr(sumtab rec of a).

[Figure 6.10](#): Translation scheme for a Boolean expression containing and, not, and or.

[Figure 6.11](#): The addition of the nullable nonterminal N facilitates an unconditional jump.

[Figure 6.12](#): A nullable nonterminal M provisions the translation of if-then.

[Figure 6.13](#): The translation of the Boolean while statement is facilitated by a nullable nonterminal M.

[Figure 6.14](#): Translation of the Boolean do-while.

[Figure 6.15](#): Translation of Boolean repeat-until.

[Figure 6.16](#): Handling the translation of the Boolean for.

[Figure 6.17](#): A switch/case three-address translation.

[Figure 6.18](#): Nullable nonterminals are introduced into a switch statement translation.

[Figure 6.19](#): Contents of queue during the translation.

## **Chapter 7: Symbol Table Management**

[Figure 7.1](#): A pointer steers the symbol table to remotely stored information for the array a.

[Figure 7.2](#): Symbol table names are held either in the symbol table record or in a separate string table.

[Figure 7.3](#): A new record is added to the linear list of records.

[Figure 7.4](#): The search tree organization approach to a symbol table.

[Figure 7.5](#): Hash table method of symbol table organization.

[Figure 7.6](#): Symbol table organization that complies with static scope information rules.

## **Chapter 8: Storage Management**

[Figure 8.1](#): Heap memory storage allows program-controlled data allocation.

[Figure 8.2](#): Typical format of an activation record.

[Figure 8.3](#): The CEP pointer is used to access the contents of the activation record.

[Figure 8.4](#): Typical callee code segment.

[Figure 8.5](#): An activation record that deals with nonlocal name references.

[Figure 8.6](#): A typical callee segment.

[Figure 8.7](#): Storage for declared names.

## **Chapter 10: Code Optimization**

[Figure 10.1](#): Program flow graph.

[Figure 10.2](#): The flow graph back edges are identified by computing the dominators.

[Figure 10.3](#): A flow graph with no back edges.

[Figure 10.4](#): Flow graph with GEN and KILL block sets.

[Figure 10.5](#): Nonunique solution to a data flow equation, where  $B$  is a predecessor of itself.

[Figure 10.6](#): A flow graph containing a loop invariant statement.

[Figure 10.7](#): Moving a loop invariant statement changes the semantics of the program.

[Figure 10.8](#): Moving the preheader changes the meaning of the program.

[Figure 10.9](#): Moving a value to the preheader changes the original meaning of the program.

[Figure 10.10](#): Flow graph where  $I$  is a basic induction variable.

[Figure 10.11](#): Modified flow graph.

[Figure 10.12](#): Flow graph preheader modifications.

[Figure 10.13](#): DAG representation of a basic block.

## **Chapter 11: Code Generation**

[Figure 11.1](#): DAG Representation.

[Figure 11.2](#): DAG Representation with heuristic ordering.

[Figure 11.3](#): DAG representation of three-address code for Example 11.3.

[Figure 11.4](#): DAG representation tree after labeling.

[Figure 11.5](#): The node  $n$  is an operand and not a subtree root.

[Figure 11.6](#): The node  $n$  is an operator, and  $n_1$  and  $n_2$  are subtree roots.

[Figure 11.7](#): A nontree DAG.

[Figure 11.8](#): A DAG that has been partitioned into a set of trees.

[Figure 11.9](#): Labeled tree for Example 11.4.

[Figure 11.10](#): Tree with a label of two.

[Figure 11.11](#): The left and right subtrees have been interchanged, reducing the register requirement to one.

[Figure 11.12](#): Associativity is used to reduce a tree's register requirement.

## Chapter 12: Exercises

[Figure 12.1](#): State transition diagram.

[Team LiB](#)

[◀ PREVIOUS](#) [NEXT ▶](#)

## List of Tables

### Chapter 4: Top-Down Parsing

[Table 4.1](#): Production Selections for Parsing Derivations

[Table 4.2](#): Production Selections for Parsing Derivations

[Table 4.3](#): Steps Involved in Parsing the String *acdb*

[Table 4.4](#): Production Selections for String *ab* Parsing Derivations

[Table 4.5](#): Production Selections for Parsing Derivations for the String *adb*

[Table 4.6](#): Production Selections for Example 4.3 Parsing Derivations

[Table 4.7](#): Production Selections for Example 4.5 Parsing Derivations

[Table 4.8](#): Production Selections for Example 4.6 Parsing Derivations

[Table 4.9](#): Production Selections for Example 4.7 Parsing Derivations

[Table 4.10](#): Production Selections for Example 4.8 Parsing Derivations

### Chapter 5: Bottom-up Parsing

[Table 5.1](#): Sentential Form Handles

[Table 5.2](#): Sentential Form Handles

[Table 5.3](#): Steps in Parsing the String *id + id \* id*

[Table 5.4](#): Action|GOTO SLR Parsing Table

[Table 5.5](#): SLR Parsing Table

[Table 5.6](#): Action | GOTO SLR Parsing Table

[Table 5.7](#): CLR/LR Parsing Action | GOTO Table

[Table 5.8](#): LALR Parsing Table for a DFA Using a Reduced Collection

[Table 5.9](#): SLR Parsing Table for Augmented Grammar

[Table 5.10](#): SLR Parsing Table Reflects Higher Precedence and Left-Associativity

[Table 5.11](#): SLR(1) Parsing Table

[Table 5.12](#): SLR Parsing Table for Example 5.4

[Table 5.13](#): Parsing Table for Example 5.5

[Table 5.14](#): LALR(1) Parsing Table for Example 5.5

[Table 5.15](#): LALR(1) Parsing Table for Example 5.6

### Chapter 6: Syntax-Directed Definitions and Translations

[Table 6.1](#): Quadruple Representation of  $x = (a + b)^* - c/d$

[Table 6.2](#): Triple Representation of  $x = (a + b)^* - c/d$

[Table 6.3](#): Indirect Triple Representation of  $x = (a + b)^* - c/d$

## **Chapter 7: Symbol Table Management**

[Table 7.1](#): Symbol Table Contents Using a Nesting Depth Approach

## **Chapter 9: Error Handling**

[Table 9.1](#): Parsing Table for  $E \rightarrow E + E \mid E^* E \mid id$

[Table 9.2](#): Higher Precedent \* and Left-Associativity

[Table 9.3](#): Parsing Table with Error Routines

[Table 9.4](#): LR Parsing Table

[Table 9.5](#): Phrase Level Error-Recovery Implementation

## **Chapter 10: Code Optimization**

[Table 10.1](#): GEN and KILL sets for Figure 10.4 Flow Graph

[Table 10.2](#): IN and OUT Computation for Figure 10.5

[Table 10.3](#): First Iteration for the IN and OUT Values

[Table 10.4](#): Second Iteration for the IN and OUT Values

[Table 10.5](#): Third Iteration for the IN and OUT Values

[Table 10.6](#): Fourth Iteration for the IN and OUT Values

## **Chapter 11: Code Generation**

[Table 11.1](#): Six-Bit Registers for the Instruction MOV R0, R1

[Table 11.2](#): Six-Bit Registers for the Instruction MOV R0, R2

[Table 11.3](#): Six-Bit Registers for the Instruction MOV M1, M2

[Table 11.4](#): Computation for the Expression  $x = (a + b) - ((c + d) - e)$

[Table 11.5](#): Generated Code with Only Two Available Registers, R0 and R1

[Table 11.6](#): Generated Code with Rearranged Computations

[Table 11.7](#): Recursive Gencode Calls

## List of Examples

### Chapter 2: Finite Automata and Regular Expressions

[EXAMPLE 2.1](#)

[EXAMPLE 2.2](#)

[EXAMPLE 2.3](#)

[EXAMPLE 2.4](#)

[EXAMPLE 2.5](#)

[EXAMPLE 2.6](#)

[EXAMPLE 2.7](#)

[EXAMPLE 2.8](#)

[EXAMPLE 2.9](#)

[EXAMPLE 2.10](#)

### Chapter 3: Context-Free Grammar and Syntax Analysis

[EXAMPLE 3.1](#)

[EXAMPLE 3.2](#)

[EXAMPLE 3.3](#)

[EXAMPLE 3.4](#)

[EXAMPLE 3.5](#)

[EXAMPLE 3.6](#)

[EXAMPLE 3.7](#)

[EXAMPLE 3.8](#)

[EXAMPLE 3.9](#)

[EXAMPLE 3.10](#)

[EXAMPLE 3.11](#)

[EXAMPLE 3.12](#)

[EXAMPLE 3.13](#)

[EXAMPLE 3.14](#)

[EXAMPLE 3.15](#)

### Chapter 4: Top-Down Parsing

[EXAMPLE 4.1](#)

[EXAMPLE 4.2](#)

[EXAMPLE 4.3](#)

[EXAMPLE 4.4](#)

[EXAMPLE 4.5](#)

[EXAMPLE 4.6](#)

[EXAMPLE 4.7](#)

[EXAMPLE 4.8](#)

## **Chapter 5: Bottom-up Parsing**

[EXAMPLE 5.1](#)

[EXAMPLE 5.2](#)

[EXAMPLE 5.3](#)

[EXAMPLE 5.4](#)

[EXAMPLE 5.5](#)

[EXAMPLE 5.6](#)

[EXAMPLE 5.7](#)

[EXAMPLE 5.8](#)

## **Chapter 6: Syntax-Directed Definitions and Translations**

[EXAMPLE 6.1](#)

[EXAMPLE 6.2](#)

[EXAMPLE 6.3](#)

[EXAMPLE 6.4](#)

[EXAMPLE 6.5](#)

[EXAMPLE 6.6](#)

## **Chapter 11: Code Generation**

[EXAMPLE 11.1](#)

[EXAMPLE 11.2](#)

[EXAMPLE 11.3](#)

[EXAMPLE 11.4](#)

## **Chapter 12: Exercises**

[EXERCISE 12.1](#)

[EXERCISE 12.2](#)

[EXERCISE 12.3](#)

[EXERCISE 12.4](#)

[EXERCISE 12.5](#)

[EXERCISE 12.6](#)

[EXERCISE 12.7](#)

[EXERCISE 12.8](#)

[EXERCISE 12.9](#)

[EXERCISE 12.10](#)

[EXERCISE 12.11](#)

[EXERCISE 12.12](#)

[EXERCISE 12.13](#)

[EXERCISE 12.14](#)

**Team LiB**

**◀ PREVIOUS**