COMPILER CONSTRUCTION

K. V. N. Sunitha

ALWAYS LEARNING

PEARSON

Compiler Construction

This page is intentionally left blank.

Compiler Construction

K. V. N. Sunitha
Principal
BVRIT Hyderabad
College of Engineering for Women
Bachupally, Hyderabad

PEARSON

Chennai • Delhi

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd.
Licensees of Pearson Education in South Asia

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version. The publisher reserves the right to remove any material in this eBook at any time.

ISBN 9789332500297
eISBN 9789332520127

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India
Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

Dedicated to

My parents

Late Sri K. Subbaiah and Smt. K. Subba Lakshmamma for inculcating the thirst for knowledge in me

My husband

Sri M. Chidambara Moorthy for his inspiration and motivation

And my sons

Charan and Uday for their love and cooperation

This page is intentionally left blank.

Brief Contents

Chapter 1:	Introduction	1
Chapter 2:	Lexical Analyzer	31
Chapter 3:	Syntax Definition – Grammars	79
Chapter 4:	Syntax Analysis—Top-Down Parsers	125
Chapter 5:	Bottom-Up Parsers	171
Chapter 6:	Syntax-Directed Translation	241
Chapter 7:	Semantic Analysis	291
Chapter 8:	Intermediate Code Generation	309
Chapter 9:	Symbol Table	337
Chapter 10:	Code Optimization	375
Chapter 11:	Code Generation	425
Recommended Readings and Websites		447
Index		449

This page is intentionally left blank.

CONTENTS

Preface *xvii*

1. INTRODUCTION 1

1.1 What Is a Compiler—2
1.1.1 History 2
1.1.2 What Is the Challenge? 3
1.2 Compiler vs. Interpreter—4
1.3 Typical Language Processing System—6
1.3.1 Preprocessor 6
1.3.2 Loader/Linker 9
1.4 Design Phases—10
1.4.1 The Lexical Analysis 10
1.4.2 Intermediate Code Generator 13
1.4.3 Code Optimizer 13
1.4.4 Target Code Generator 14
1.4.5 Symbol Table Manager and Error Handler 15
1.4.6 Compiler Front End 16
1.4.7 Compiler Back End 17
1.5 Design Passes—19
1.6 Retargeting—20
1.7 Bootstrapping—20
1.7.1 T-diagram 21
1.7.2 Advantages of Bootstrapping 22
1.8 Compiler Design Tools—23
1.9 Modern Compilers—Design Need for Compilers—24
1.10 Application of Compiler Design Principles—24
Solved Problems 25
Summary 27
Fill in the Blanks 28
Objective Question Bank 29
Exercises 30
Key for Fill in the Blanks 30
Key for Objective Question Bank 30

2. Lexical Analyzer 31

2.1 Introduction—32
2.2 Advantages of Separating Lexical Analysis from Syntax Analysis—33
2.3 Secondary Tasks of a Lexical Analyzer—33
2.4 Error Recovery in Lexical Analysis—33
2.5 Tokens, Patterns, Lexemes—34
2.6 Strategies for Implementing a Lexical Analyzer—37
2.7 Input Buffering—38
2.8 Specification of Tokens—40
2.8.1 Operations on Language 40
2.9 Recognition of Tokens—42
2.10 Finite State Machine—43
2.10.1 Finite Automaton Model 44
2.10.2 Properties of the Transition Function "δ" 45
2.10.3 Transition Diagram 45
2.10.4 Transition Table 45
2.10.5 Language Acceptance 46
2.10.6 Finite Automation Is of Two Types 47
2.10.7 Deterministic Finite Dutomaton (DFA) 48
2.10.8 Nondeterministic Finite Automaton (NFA) 49
2.10.9 Equivalence of DFAs and NFAs 50
2.10.10 Converting NFA (MN) to DFA (MD)—Subset Construction 51
2.10.11 NFA with Epsilon (ε)-Transitions 54
2.10.12 Epsilon Closure (ε-closure) 54
2.10.13 Eliminating ε-Transitions 55
2.10.14 Converting NFA with ε-Transition to NFA Without ε-Transition 55
2.10.15 Converting NFA with ε-Transition to DFA 56
2.10.16 Comparison Method for Testing Equivalence of Two FAs 57
2.10.17 Reduction of the Number of States in FA 58
2.10.18 Minimization of DFA 59
2.10.19 Minimization of DFA Using the Myhill Nerode Theorem 61
2.11 Lex Tool: Lexical Analyzer Generator—63
2.11.1 Introduction 63
Solved Problems 64
Summary 66
Fill in the Blanks 66
Objective Question Bank 67
Exercises 68
Key for Fill in the Blanks 70
Key for Objective Question Bank 70

3. Syntax Definition – Grammars 79

3.1 Introduction—79
3.2 Types of Grammars—Chomsky Hierarchy—81

3.3 Grammar Representations—84
3.4 Context Free Grammars—87
3.5 Derivation of CFGs—89
3.6 Language Defined by Grammars—91
3.6.1 Leftmost and Rightmost Derivation 92
3.6.2 Derivation Tree 93
3.6.3 Equivalence of Parse Trees and Derivations 94
3.7 Left Recursion—96
3.8 Left-Factoring—98
3.9 Ambiguous Grammar—100
3.10 Removing Ambiguity——103
3.11 Inherent Ambiguity—105
3.12 Non-context Free Language Constructs—106
3.13 Simplification of Grammars—106
3.14 Applications of CFG—112
Solved Problems 112
Summary 116
Fill in the Blanks 116
Objective Question Bank 117
Exercises 120
Key for Fill in the Blanks 123
Key for Objective Question Bank 123

4. SYNTAX ANALYSIS—TOP-DOWN PARSERS 125

4.1 Introduction—126
4.2 Error Handling in Parsing—126
4.2.1 Panic Mode Error Recovery 127
4.2.2 Phrase Level Recovery 127
4.2.3 Error Productions 127
4.2.4 Global Correction 127
4.3 Types of Parsers—128
4.3.1 Universal Parsers 128
4.3.2 Top-Down Parsers (TDP) 128
4.3.3 Bottom-Up Parsers 130
4.4 Types of Top-Down Parsers—131
4.4.1 Brute Force Technique 131
4.5 Predictive Parsers—133
4.5.1 Recursive Descent Parser 134
4.5.2 Nonrecursive Descent Parser—LL(1) Parser 136
4.5.3 Algorithm for LL(1) Parsing 137
4.5.4 First(α), Where α Is Any String of Grammar Symbols 139
4.5.5 Follow(A) Where 'A' is a Nonterminal 141
4.6 Construction of Predictive Parsing Tables—144
4.7 LL(1) Grammar—145

4.8 Error Recovery in Predictive Parsing—150
Solved Problems 152
Summary 157
Fill in the Blanks 158
Objective Question Bank 158
Exercises 161
Key for Fill in the Blanks 163
Key for Objective Question Bank 164

5. Bottom-Up Parsers 171

5.1 Bottom-Up Parsing—172
5.2 Handle—173
5.3 Why the Name SR Parser—174
5.4 Types of Bottom-Up Parsers—175
5.5 Operator Precedence Parsing—176
5.5.1 Precedence Relations 177
5.5.2 Recognizing Handles 177
5.5.3 Parsing Algorithm for Operator Precedence Parser 178
5.5.4 Construction of the Precedence Relation Table 179
5.5.5 Mechanical Method of Constructing Operator Precedence Table 181
5.5.6 Calculating Operator Precedence Relation <· ·> = 182
5.5.7 Error Recovery in Operator Precedence Parser 184
5.5.8 Procedure for Converting Precedence Relation Table to Precedence Function Table 186
5.6 LR Grammar—187
5.7 LR Parsers—187
5.8 LR Parsing Algorithm—188
5.8.1 Task of LR Parser: Detect Handle and Reduce Handle 188
5.9 Construction of the LR Parsing Table—191
5.9.1 Augmented Grammar 192
5.9.2 LR(0) Item 192
5.9.3 Closure(I) 193
5.9.4 Goto(I,X) 194
5.9.5 Creating Canonical Collection "C" of LR(0) Items 195
5.9.6 Construction of DFA with a Set of Items 195
5.10 LR(0) Parser—197
5.10.1 Advantages of the LR(0) Parser 199
5.10.2 Disadvantages of the LR(0) Parser 199
5.10.3 LR(0) Grammar 199
5.10.4 Conflicts in Shift-Reduce Parsing 200
5.11 SLR(1) Parser—204
5.12 Canonical LR(1) Parsers CLR(1)/LR(1)—209
5.12.1 Closure(I) Where I Is a Set of LR(1) Items 209
5.12.2 Goto(I,X) 210

5.12.3 Creating Canonical Collection "C" of LR(1) Items 211
5.12.4 Constructing CLR(1) Parsing Table 212
5.12.5 CLR(1) Grammar 213
5.13 LALR(1) Parser—215
5.14 Comparison of Parsers: Top-Down Parser vs. Bottom-Up Parser—223
5.15 Error Recovery in LR Parsing—224
5.16 Parser Construction with Ambiguous Grammars—225
Solved Problems 227
Summary 233
Fill in the Blanks 234
Objective Question Bank 235
Exercises 237
Key for Fill in the Blanks 239
Key for Objective Question Bank 239

6. Syntax-Directed Translation 241

6.1 Introduction—241
6.2 Attributes for Grammar Symbols—242
6.3 Writing Syntax-Directed Translation—243
6.4 Bottom-Up Evaluation of SDT—250
6.5 Creation of the Syntax Tree—260
6.6 Directed Acyclic Graph (DAG)—262
6.7 Types of SDTs—264
6.8 S-Attributed Definition—265
6.9 Top-Down Evaluation of S-Attributed Grammar—265
6.10 L-Attributed Definition—269
6.11 Converting L-Attributed to S-Attributed Definition—276
6.12 YACC—278
Solved Problems 284
Summary 288
Fill in the Blanks 289
Objective Question Bank 289
Key for Fill in the Blanks 290
Key for Objective Question Bank 290

7. Semantic Analysis 291

7.1 Introduction—291
7.2 Type Systems—293
7.3 Type Expressions—293
7.4 Design of Simple Type Checker—295
7.5 Type Checking of Expressions—296
7.6 Type Checking of Statements—296
7.7 Type Checking of Functions—297

7.8 Equivalence of Type Expressions—297
7.8.1 Structural Equivalence 297
7.8.2 Encoding of Type Expressions 298
7.8.3 Name Equivalence 299
7.8.4 Type Graph 300
7.9 Type Conversion—302
7.10 Overloading of Functions and Operators—302
7.11 Polymorphic Functions—303
Solved Problems 304
Summary 305
Fill in the Blanks 306
Objective Question Bank 306
Exercises 307
Key for Fill in the Blanks 308
Key for Objective Question Bank 308

8. INTERMEDIATE CODE GENERATION 309

8.1 Introduction—309
8.2 Intermediate Languages—309
8.2.1 Syntax Trees 310
8.2.2 Directed Acyclic Graph (DAG) 312
8.2.3 Postfix Notation 312
8.2.4 Three Address Code 313
8.3 Types of Three Address Statements—314
8.4 Representation of Three Address Code—315
8.4.1 Quadruple 316
8.4.2 Triple 316
8.4.3 Indirect Triples 317
8.4.4 Comparison of Representations 318
8.5 Syntax-Directed Translation into Three Address Code—318
8.5.1 Assignment Statement 318
8.5.2 Addressing Array Elements 320
8.5.3 Logical Expression 322
8.5.4 Control Statements 324
Solved Problems 327
Summary 332
Fill in the Blanks 333
Objective Question Bank 333
Exercises 334
Key for Fill in the Blanks 335
Key for Objective Question Bank 335

9. SYMBOL TABLE 337

9.1 Introduction—337

9.2 Symbol Table Entries—339
9.3 Operations on the Symbol Table—340
9.4 Symbol Table Organization—341
9.5 Non-block Structured Language —342
9.5.1 Linear List in Non-block Structured Language 342
9.5.2 Linked List or Self-organizing Tables 344
9.5.3 Hierarchical List 347
9.5.4 Hash Tables 352
9.6 Block Structured Language—356
9.6.1 Stack Symbol Tables 358
9.6.2 Stack-Implemented Tree-structured Symbol Tables 363
9.6.3 Stack-Implemented Hash-structured Symbol Table 365
Summary 368
Fill in the Blanks 368
Objective Question Bank 369
Exercises 371
Key for Fill in the Blanks 374
Key for Objective Question Bank 374

10. CODE OPTIMIZATION 375

10.1 Introduction—375
10.2 Where and How to Optimize—377
10.3 Procedure to Identify the Basic Blocks—378
10.4 Flow Graph—380
10.5 DAG Representation of Basic Block—381
10.6 Construction of DAG—381
10.6.1 Algorithm for Construction of DAG 382
10.6.2 Application of DAG 384
10.7 Principle Source of Optimization—386
10.8 Function-Preserving Transformations—386
10.8.1 Common Sub-expression Elimination 386
10.8.2 Copy Propagation 389
10.8.3 Dead Code Elimination 394
10.8.4 Constant Propagation 397
10.9 Loop Optimization—397
10.9.1 A Loop Invariant Computation 398
10.9.2 Induction Variables 399
10.10 Global Flow Analysis—403
10.10.1 Points and Paths 404
10.10.2 Reaching Definition 405
10.10.3 Use Definition Chains 405
10.10.4 Live Variable Analysis 405
10.10.5 Definition Use Chains 406
10.10.6 Data Flow Analysis of Structured Programs 406
10.10.7 Representation of Sets 406

10.10.8 Iterative Algorithm for Reaching Definition 408
10.11 Machine-Dependent Optimization—411
10.11.1 Redundant Loads and Stores 411
10.11.2 Algebraic Simplification 412
10.11.3 Dead Code Elimination 412
10.11.4 Flow-of-Control Optimization 413
10.11.5 Reduction in Strength 413
10.11.6 Use of Machine Idioms 413
Solved Problems 414
Summary 419
Fill in the Blanks 420
Objective Question Bank 420
Exercises 421
Key for Fill in the Blanks 424
Key for Objective Question Bank 424

11. CODE GENERATION 425

11.1 Introduction—425
11.2 Issues in the Design of a Code Generator—425
11.2.1 Input to the Code Generator 426
11.2.2 Target Programs 426
11.2.3 Memory Management 427
11.2.4 Instruction Selection 427
11.2.5 Register Allocation 428
11.2.6 Choice of Evaluation Order 429
11.3 Approach to Code Generation—429
11.3.1 Algorithm for Code Generation Using Three Address Code 430
11.4 Instruction Costs—432
11.5 Register Allocation and Assignment—433
11.5.1 Fixed Registers 434
11.5.2 Global Register Allocation 434
11.5.3 Usage Count 434
11.5.4 Register Assignment for Outer Loop 435
11.5.5 Graph Coloring for Register Assignment 436
11.6 Code Generation Using DAG—436
Solved Problems 439
Summary 442
Fill in the Blanks 442
Objective Question Bank 443
Exercises 444
Key for Fill in the Blanks 445
Key for Objective Question Bank 445

Recommended Readings and Websites 447
Index 449

PREFACE

Every computing device built today needs a compiler. It enables us to use a programming language by translating programs into machine code. It is essential for a good programmer to understand how a compiler works. The study of compilers entails an analysis of theoretical ideas in translation and optimization with sparse resources. The purpose of this book is to cover the underlying concepts and techniques used in compiler design. Some of these techniques can also be used in software design and natural language processing.

Intended as an introductory reading material in the subject, the book uses enough examples and algorithms to effectively explain the various phases in compiler design, besides covering the design of each component of a compiler in detail. As part of the course, students can implement each phase of a compiler. The programming languages used in the examples are in C.

The eleven chapters in the book are organized as explained below:

- Chapter 1 gives an overview of compiler design and covers all the phases in a compiler.
- Chapter 2 covers lexical analysis, formal language theory, regular expressions and finite-state machines and the automated tool LEX.
- Chapter 3 discusses syntax definition. The basic notations and concepts of grammars and languages such as ambiguity, left recursion and left factoring are unraveled. This is useful in the design of parsers.
- Chapter 4 on syntax analysis elaborates on the types of parsing and types of top-down parsers and its design.
- Chapter 5 describes bottom-up parsers and their design. Problems and solution in each design are explained.
- Chapter 6 delineates syntax-directed translation. Methods to perform top-down translation and bottom-up translations are examined.
- Chapter 7 based on semantic analysis provides the main ideas for static type checking, typing rules, type expressions and the design of simple type checker.
- Chapter 8 expounds on intermediate code generation and the translation of different language constructs into their address code. It also elucidates runtime support required for design of compilers and the different storage allocations.
- Chapter 9 analyzes symbol tables. The important functions of a symbol table and symbol table organization for block-structured and non-block structured languages are spelt out in detail.
- Chapter 10 on optimization reveals the various machine-dependent and machine-independent optimization techniques. It also depicts register allocation using graph coloring and data flow equations.

- Chapter 11 rationalizes target code generation, enumerates the methods to design a simple code generator and points out the issues in design.

The problems denoted by asterisks in the examples and exercises have appeared in GATE examinations.

Acknowledgements

I sincerely thank my management—the Chairman Sri K. V. Vishnu Raju, the Vice-Chairman Sri Ravi Chandran Raja Gopal and the Director Sri Ram Kumar of Sri Vishnu Educational Society—for their encouragement and support in completing this book. I thank my research scholar Dr N. Kalyani for her review comments. I also thank the publisher Pearson Education for their best efforts in bringing out this book in time.

K. V. N. Sunitha

CHAPTER 1

Introduction

A compiler is a translator that translates programs written in one language to another language, that is, translates a high-level language program into a functionally equivalent low-level language program that can be understood and later executed by a computer. Designing an effective compiler for a computer system is also one of the most complex areas of system development.

CHAPTER OUTLINE

1.1 What Is a Compiler?
1.2 Compiler vs. Interpreter
1.3 Typical Language-Processing System
1.4 Design Phases
1.5 Design of Passes
1.6 Retargeting
1.7 Bootstrapping
1.8 Compiler Design Tools
1.9 Modern Compilers—Design Need for Compilers
1.10 Application of Compiler Design Principles

Compiler construction is a broad field. The demand for compilers will always remain as long as there are programming languages. In early compilers, little was understood about the underlying principles and techniques and all tasks were done by hand. Hence, compilers were messy and complex systems. As programming languages started evolving rapidly, more compilers were in demand; so people realized the importance of compilers and started understanding the principles and introduced new tools for designing a compiler.

In early days, much of the effort in compilation was focused on how to implement high-level language constructs. Then, for a long time, the main emphasis was on improving the efficiency of generated codes. These topics remain important even today, but many new technologies have caused them to become more specialized. Compiler writing techniques can be used to construct translators for a wide variety of languages. The process of compilation is introduced in this chapter. The importance of a compiler in a typical language-processing system and the challenges in designing a compiler are discussed in detail. The design phases of a compiler along with the modern compiler tools are explained.

1.1 What Is a Compiler?

The software that accepts text in some language as input and produces text in another language as output while preserving the actual meaning of the original text is called a *translator*. The input language is called the *source language*; the output language is called the *target* or *object language*. A compiler is basically a translator. It translates a source program written in some high-level programming language such as Pascal/C/C++ into machine language for computers, such as the Intel Pentium IV/AMD processor machine, as shown in Figure 1.1. The generated machine code can be used for execution as many times as needed. In addition to this, a compiler even reports about errors in the source program.

1.1.1 History

In the early days of computers, programmers had to be exceptionally skilled since all the coding was done in machine language. Machine language code is written in binary bit patterns (or hexadecimal patterns). With the evolution of assembly language, programming was made simpler. These languages used mnemonics, which has a close resemblance with symbolic instructions and executable machine codes. A programmer must pay attention to far more detail and must have complete knowledge of the processor in use.

Toward the end of the 1950s, machine-independent languages were introduced. These languages had a high level of abstraction. Typically, a single high-level instruction is translated into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions. Grace Murray Hopper conceptualized the first translator for the A-0 programming language from his experimental studies. This was coined as a compiler in the early 1950s. John Backus at IBM took 18 man years to introduce the first complete compiler for FORTRAN in 1957.

In many application domains, the idea of using a higher-level language quickly caught on for various reasons.

- High-level programming languages are much easier to learn and understand. No background knowledge in hardware is necessary.
- They can be easily debugged.
- They are relatively machine independent.
- While writing programs, programmers need not consider the internal structure of a system.

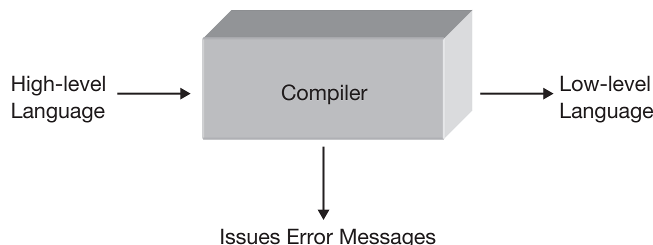


Figure 1.1 Compiler Input and Output

- High-level languages (HLL) offer more powerful control and data structure facilities than low-level languages.
- High-level languages allow faster development than in assembly language, even with highly skilled programmers. Development time increases and it becomes 10 to 100 times faster.
- Programs written in high-level languages are much easier and less expensive to maintain than similar programs written in assembly language or low-level languages.

Computers only understand machine language. As people started writing programs in high-level languages, to translate them into machine understandable form, compilers were invented. Hence, knowledge of compilers became the basic necessity for any programmer.

Initially compilers were written in the assembly language. In 1962, Tim Hart and Mike Levin at MIT created the first *self-hosting* compiler for Lisp. A self-hosting compiler is capable of compiling its own source code in a high-level language. The initial C compiler was written in C at AT & T Bell labs. Since the 1970s, it has become common practice to implement a compiler in the language it compiles. Building a self-hosting compiler is a bootstrapping problem.

1.1.2 What Is the Challenge?

The complexity of compilers is increasing day by day because of the increasing complexity of computer architectures and expanding functionality supported by newer programming languages.

So to design a compiler, the main challenge lies with variations in issues like:

- Programming languages (e.g., FORTRAN, C++, Java)
- Programming paradigms (e.g., object-oriented, functional, logic)
- Computer architectures (e.g., MIPS, SPARC, Intel, alpha)
- Operating systems (e.g., Linux, Solaris, Windows)

Qualities of a Compiler (in the order of importance)

1. The compiler itself must be bug-free.
2. It must generate the correct machine code.
3. The generated machine code must run fast.
4. The compiler itself must run fast (compilation time must be proportional to program size).
5. The compiler must be portable (i.e., modular, supporting separate compilation).
6. It must print good diagnostics and error messages.
7. The generated code must work well with existing debuggers.
8. The compiler must have consistent and predictable optimization.

Building a compiler requires knowledge of

- Programming languages (parameter passing techniques supported, scope of variables, memory allocation techniques supported, etc.)
- Theory (automata, context-free languages, grammars, etc.)
- Algorithms and data structures (hash tables, graph algorithms, dynamic programming, etc.)
- Computer architecture (assembly programming)
- Software engineering.

The first compiler was built for FORTRAN at IBM in the 1950s. It took nearly 18 man years to do this. Today, with automated tools, we can build a simple compiler in just a few months. Crafting an efficient and a reliable compiler is always a challenging task.

1.2 Compiler vs. Interpreter

An interpreter is another way of implementing a programming language. An interpreter works similar to a compiler in most of the phases like Lexical, Syntax, and Semantic. This analysis is performed for each single statement and when the statement is error free, instead of generating the corresponding code, that statement is executed and the result of execution is displayed. In an interpreter, since every time the program has to be processed for and has to be checked for errors, it is slower than the compiler program. Writing an interpreter program is a simpler task than writing a compiler program because a compiler processes the program as a whole, whereas an interpreter processes a program one line at a time. If speed is not a constraint, an interpreter can be moved on to any machine easily.

An *interpreter* first reads the source program line by line, written in a high-level programming language; it also reads the data for this program; then it runs or executes the program directly against the data to produce results. This is shown in Figure 1.2. It does not produce machine code or object code. One example is the Unix shell interpreter, which runs operating system commands interactively.

Some languages, like Java, combine both compiler and interpreter. The Java program was first compiled by a compiler to produce a byte code. This byte code is interpreted while execution. In some systems, the compiler may generate a syntax tree and syntax tree may be interpreted directly. The choice of compiler and interpreter depends on the compromise made on speed and space. The compiled code is bigger than the syntax tree, but the running speed is less for the compiled code than for the syntax tree.

The interpreter is a good choice, if the program has to be modified while testing. It starts to run the program quickly and works on the representation that is close to the source code. When an error occurs, it displays informative messages and allows users to correct the errors immediately.

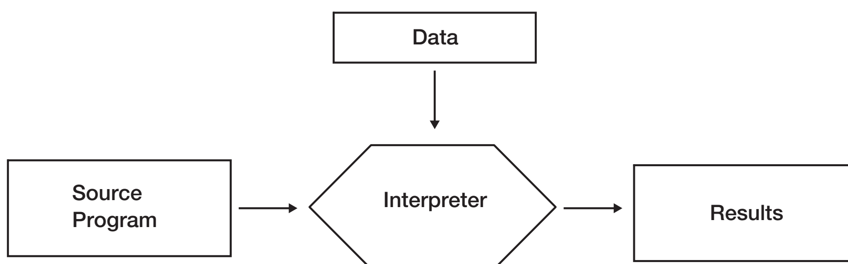


Figure 1.2 Interpreter

Note that both interpreters and compilers (like any other program) are written in some high-level programming language (which may be different from the language they accept) and they are translated into machine code. For example, a Java interpreter can be completely written in C or C++ or even in Java. Because an interpreter does not generate machine code as output for the input source code, it is machine independent.

- ✓ C, C++, Fortran, Pascal, C#
  - ❍ Usually compiled
- ✓ Basic, JavaScript, Lisp, LOGO, Perl, Python, SMALL TALK, APL
  - ❍ Usually interpreted

Generally a compiler is faster than an interpreter because interpreter reads and interprets each statement in a program as many times as the number of evaluations of the statement. For example, when a for/while loop is interpreted, the statements inside the body of the loop will be analyzed and evaluated on every iteration. Some languages, such as Java and Lisp, are provided with both an interpreter and a compiler. Java programs are generally compiled and interpreted. Java compiler (javac) converts programs written in java , that is, Java classes with .java extension into byte-code files with .class extension. The Java Virtual Machine, which is known as JVM is a java interpreter. It may internally compile the byte code into machine code or interpret the byte code directly and then execute the resulting code.

Advantages of Interpreters

1. If a program is simple and is to be executed once, an interpreter is preferred.

   For example, let us suppose that there is a program with 5000 statements. For a test run, only 50 lines need to be visited. The time taken by a compiler and then an interpreter to run the program is as follows:

   Compiler ⟶ $5000\, t_c + 50\, t_e$
   Interpreter ⟶ $50\, t_i$

   where t_c is time taken to compile, t_e is time taken for execution, and t_i is time taken for interpreting.
   Hence, if it is single execution, an interpreter is preferred to a compiler.
   To execute a database, OS commands interpreter is preferred to a compiler.

2. Interpreters can be made portable than compilers.

   To understand the portability of an interpreter, consider the following scenario. Assume that we are writing an interpreter in some high-level language (like Java), which supports portability. Whatever you write using that language is portable. Hence, the interpreter is also portable. However, if a compiler is written in the same language, then it is not portable. Because the output of a compiler (whatever may be the implementation language) is always machine language, it runs only on a specific machine.

3. Certain language features are supported by interpreters.

   For example, consider a language feature like dynamic types. In SNOBOL, the type of a variable is dependent on a value that is assigned to it anytime.

If $x = 20$, then x is of the integer type.

If $x = 2.5$, then x is real type.

If x = "hai," then x is literal type.

In languages like FORTRAN, the variables are assigned the type based on first character. If variable name starts with i, j, k, l, m, n, it is treated as an integer and otherwise as real. All compilers do not support dynamic types but can be supported by interpreters.

Therefore, in the above situation, we prefer an interpreter rather than a compiler.

Compilers and interpreters are not the only examples of translators. There are many other translators. Here are a few more examples:

Table 1.1 List of translators

Translator	Source code	Target code
Text editors	English text	Meaningful sentence
LEX	Regular expression	A scanner
YACC	SDT	Parser generator
Javac compiler	Java	Java byte code
Cross compiler	Java	C++ code
Database query optimizer	SQL	Query evaluation plan
Text formater	LaTeX	PostScript

This book discusses the design of a compiler for programs in high-level languages. But the same techniques can also be applied to design interpreters.

1.3 Typical Language-Processing System

To understand the importance of a compiler, let us take a typical language-processing system shown in Figure 1.3. Given a high-level language program, let us see how we get the executable code. In addition to a compiler, other programs (translators) are needed to generate an *executable code*. The different software required in this process are shown below in the figure. The first translator needed here is the preprocessor.

1.3.1 Preprocessor

A source program may be divided into modules stored in separate files and may consist of macros. A preprocessor produces input to a compiler. A preprocessor processes the source code before the compilation and produces a code that can be more efficiently used by the compiler. It is not necessary to consider a preprocessor as part of a compiler as preprocessing requires a complete pass. It cannot be included as part of a single pass compiler.

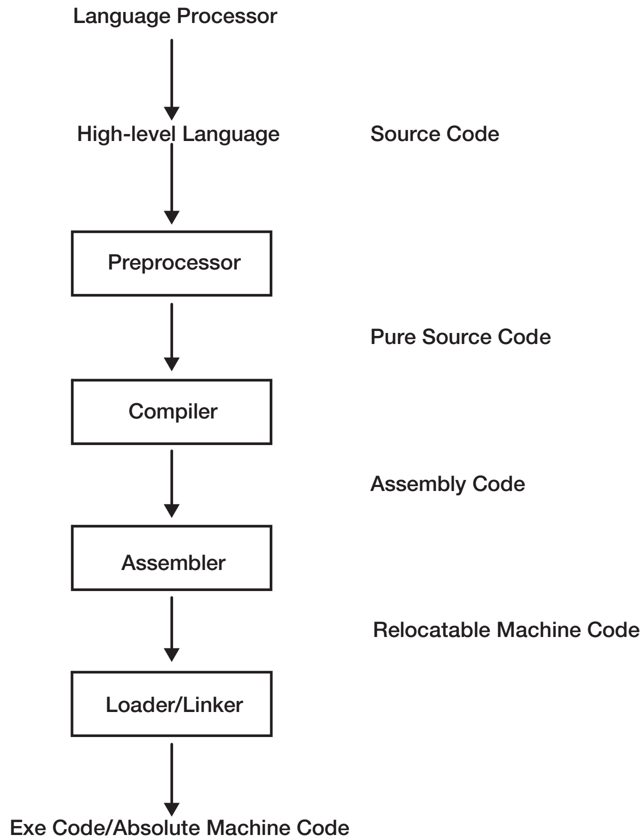


Figure 1.3 Typical Language Processing System

A preprocessor performs the following functions:

a. ***Macro processing***: Macros are shorthands for longer constructs. A macro processor has to deal with two types of statements—macro definition and macro expansion. Macro definitions have the name of the macro and a body defining the macro. They contain formal parameters. During the expansion of the macro, these formal parameters are substituted for the actual parameters. All macros (i.e., *#define* statements) are identified and substituted with their respective values.
b. ***File inclusion***: Code from all files is appended in text while preserving line numbers from individual files.
c. ***Rational preprocessor***: They augment older languages with modern flow of control and data-structuring facilities.
d. ***Language extensions***: A preprocessor can add new functionalities to extend the language by built-in macros—for example, adding database query facilities to the language.

```
#include <stdio.h>
#define min(x, y)((x)<(y))?(x):(y).. other
declarations…

void fun()
{
int a = 1;
int b = 2;
int c;
c = min(a,b);
}
```

Sample C program

```
extern intprintf (char *,...);
/* ... many more declarations from stdio.h */
.. other declarations…

void fun()
{
int a = 1;
int b = 2;
int c;
c = ((a)<(b))?(a):(b);
}
```

Preprocessor output

Figure 1.4 Preprocessor Example

Figure 1.4 shows an example of a code before and after preprocessing.

If the C language program is an input for a preprocessor, then it produces the output as a C program where there are no #includes and macros, that is, a C program with only C statements (pure HLL). This phase/translator is not mandatory for every high-level language program. For example, if the source code is a Pascal program, preprocessing is not required. This is an optional phase. So the output of this translator is a pure HLL program.

Compiler: To convert any high-level language to machine code, one translator is mandatory and that is nothing but a *compiler*. A ***compiler*** is a translator that converts a high-level language to a low-level language (e.g., assembly code or machine code).

Assembler: Assembly code is a mnemonic version of machine code in which names rather than binary values for machine instructions and memory addresses are used. An assembler needs to assign memory locations or addresses to symbols/identifiers. It should use these addresses in generating the target language, that is, the machine language. The assembler should ensure that the same address must be used for all the occurrences of a given identifier and no two identifiers are assigned with the same address. A simple mechanism to accomplish this is to make two passes over the input. During the first pass whenever a new identifier is encountered, assign an address to it. Store the identifier along with the address in a symbol table. During the second pass, whenever an identifier is seen, then its address is retrieved from the symbol table and that value is used in the generated machine code.

Consider the following example:

Example 1: Consider the following C code for adding two numbers:

```
main()
 {
  int x,y,z;
 scanf("%d, %d",&x,&y);
      z=x+y;
     }
```

The equivalent assembly code for adding two numbers is as follows:

```
.BEGIN
IN X
LOAD X
IN Y
ADD Y
STORE Z
HALT
.END
X: .DATA 0
Y: .DATA 0
Z: .DATA 0
```

Assembler is a translator that converts assembly code to machine code. Machine code is of two types. One is absolute machine code and other is relocatable machine code. The machine code with actual memory addresses is called the absolute machine code. Generally, the assembler produces the machine code with relative addresses, which is called the relocatable machine code since it needs to be relocated in memory for execution.

The equivalent machine relocatable code for adding two numbers is as follows:

```
Address  Machine  Code
0000     1101     0110
0001     0000     0110
0010     1101     0111
0011     0011     0111
0100     0001     1000
0101     1111     0000

0110     0000     0000
0111     0000     0000
1000     0000     0000
```

1.3.2 Loader/Linker

To convert the relocatable machine code to the executable code, one more translator is required and this is called the loader/linker.

A loader/linker or link editor is a translator that takes one or more object modules generated by a compiler and combines them into a single executable program called the exe code.

The terms loader and linker are used synonymsly on Unix environments. This program is known as a linkage editor in IBM mainframe OS. However, in some operating systems, the same program handles both the tasks of object linking and physical loading of a program. Some systems use linking for the former and loading for the latter. The definitions of linking and loading are as follows:

Linking: This is a process where a linker takes several object files and libraries as input and produces one executable object file shown in Figure 1.5. It retrieves from the input files (and

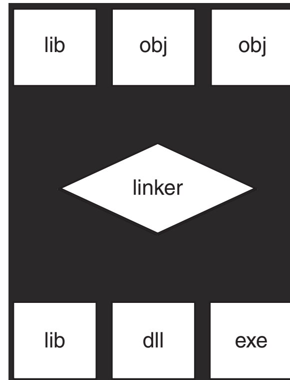


Figure 1.5 Linker

combines them in the executable code) the code of all the procedures that are referenced and resolves all external references to actual machine addresses. The libraries include language-specific libraries, operating system libraries, and user-defined libraries.

Loading: This is a process where a loader loads an executable file into memory, initializes the registers, heap, data, etc., and starts the execution of the program.

If we look at the design of all these translators, designing a preprocessor or assembler or loader/linker is simple. It can be taken up as a one-month project. But among all, the most complex translator is the compiler. The design of the first FORTRAN compiler took 18 man years. The complexity of the design of a compiler mainly depends on the source language. Currently, many automated tools are available. With modern compiler tools like YACC (yet another compiler compiler), LEX (lexical analyzer), and data flow engines, the design of a compiler is made easy.

1.4 Design Phases

Designing a compiler is a complex task; the design is divided into simpler subtasks called "phases." A phase converts one form of representation to another form. The task of a compiler can be divided into six phases as shown in Figure 1.6.

1.4.1 The Lexical Analysis

Lexical analyzer is also called **scanner**. The **scanner** groups the input stream of characters into a stream of tokens and constructs a **symbol table**, which is used later for contextual analysis. The tokens in any high-level language can include

- key words,
- identifiers,
- operators,
- constants: numeric, character and special characters
- comments,
- punctuation symbols.

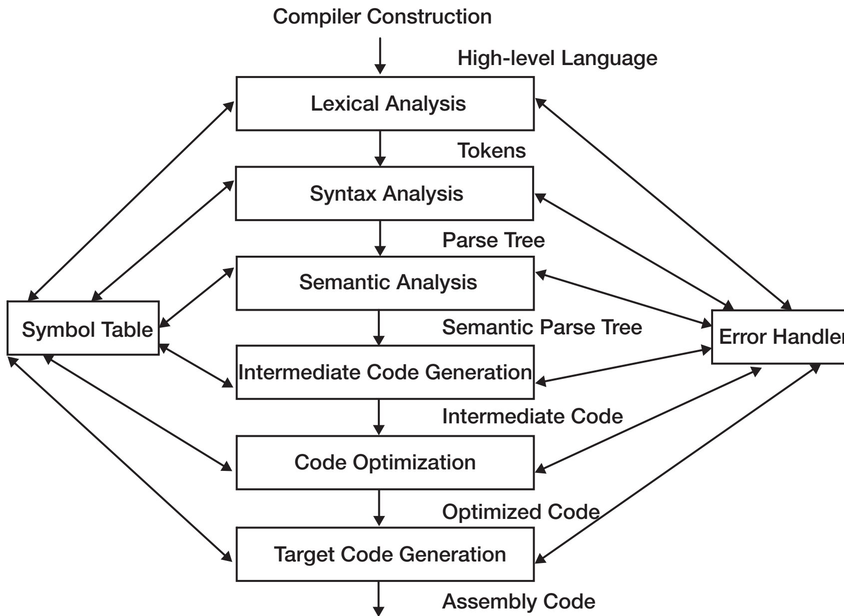


Figure 1.6 Phases of a Compiler

The **lexical analyzer** groups input stream of characters into logical units called tokens. The input to the lexical analysis is a character stream and the output is a stream of tokens. The lexical analysis is shown in Figure 1.7. Regular expressions are used to define the rules for recognizing the tokens by a scanner (or lexical analyzer). The scanner is implemented as a finite state automata. Automated scanner generators tools are Lex and Flex. Flex is a faster version of Lex.

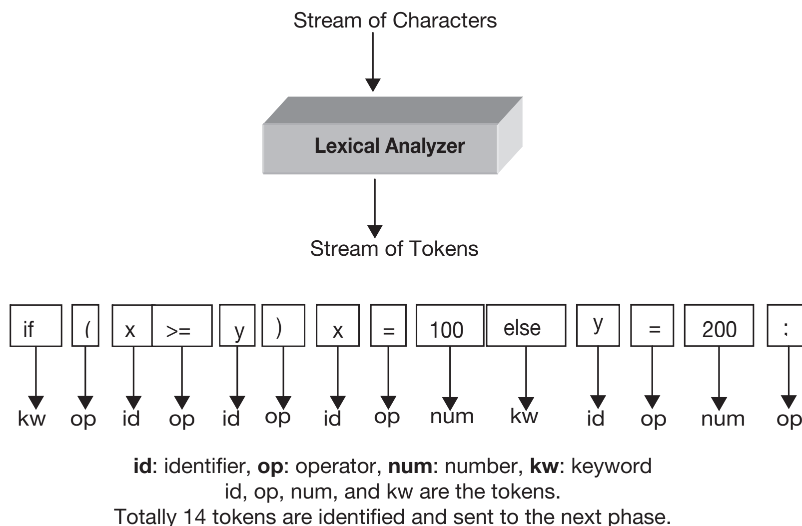


Figure 1.7 Lexical Analysis

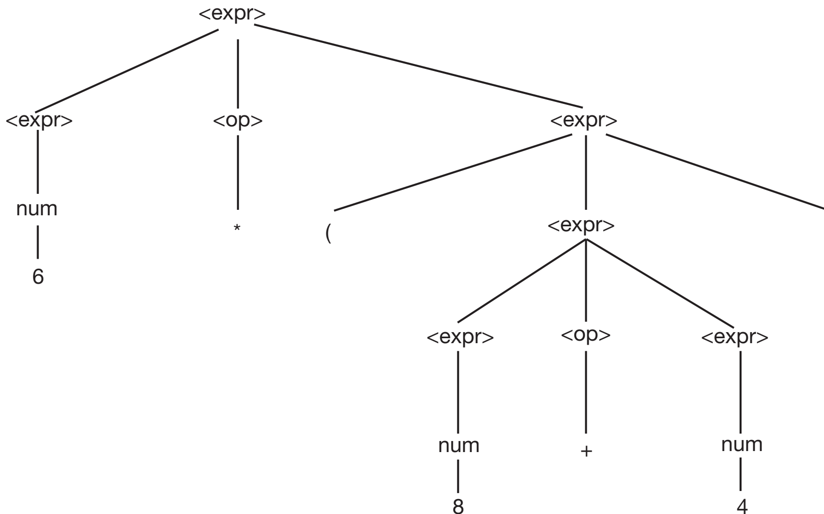


Figure 1.8 Parser tree for 6 ∗ (8 + 4)

Ex: If (x > = y) x = 100 else y = 200;

Syntax Analysis: Syntax Analyzer checks the syntax of a given statement. This is also called parser**.** The parser groups stream of tokens into a hierarchical structure called the parse tree. In order to check the syntax, first the standard syntax of all such statements is to be known. Generally the syntax rules for any HLL are given by context-free grammars (CFG) or Backus-Naur Form (BNF). The CFG, which describes all assignment statements with arithmetic expressions, is as follows:

Example 2: To check the syntax of the statement 6 × (8 + 4)
CFG is

Expr → Expr op Expr
| (expr)
| num
op → + | - | * | /

Using this grammar, the syntax of the given statement is verified by deriving the statement from grammar. This is shown in Figure 1.8.

So the output of the parser is a parse tree representation of the program. The parser is generally implemented as a push-down automaton.

Many automated tools are available for designing parsers. YACC and Bison are widely used tools. They are used for generating bottom-up parsers in C language. Bison is a faster version of YACC.

Semantic Analysis: Checks are performed to ensure that parts of a program fit together meaningfully. The semantic analyzer checks semantics, that is, whether the language constructs are meaningful.

For example, given a statement A + B, let us see how a parser and a semantic analyzer work.

On seeing the above expression, parsers look at binary addition and check whether two operands are present. The presence of two operands confirms that the syntax is right. However, it does not recognize the operand type, whether it is A or B. Even if A is an array and B is a function, it says that the syntax is right. It is the semantic analyzer that checks whether the two operands are type-compatible or not. Static type checking is the basic function of the semantic analyzer. The output of the semantic analysis phase is an annotated parse tree. Attribute grammars are used to describe the semantics of a program.

This phase is often combined with the syntax analysis. During parsing, information concerning variables and other objects is stored in a ***symbol table***. This information is used to perform the context-sensitive checking in the semantic analysis phase.

1.4.2 Intermediate Code Generator

This phase converts the hierarchical representation of source text, that is, this phase converts the parse tree into a linear representation called **Intermediate Representation** (**IR**) of the program. A well-designed intermediate representation of source text facilitates the independence of the analysis and syntheses (front- and back-end) phases. Intermediate representations may be the assembly language or an abstract syntax tree.

An ***intermediate language/code*** is often used by many compilers for analyzing and optimizing the source program. Intermediate language should have two important properties: (a) it should be simple and easy to generate and (b) it should be easy to translate to the target program.

IR can be represented in different ways: (1) as a syntax tree, (2) as a directed acyclic graph (DAG), (3) as a postfix notation, and (4) as a three-address code.

Example of IR in three-address code is as follows:

```
temp1 = int2float(sqr(t));
temp2 = g * temp1;
temp3 = 0.5 * temp2;
 dist = temp3;
```

The main purpose of IR is to make target code generation easier.

1.4.3 Code Optimizer

Compile time code optimization involves static analysis of the intermediate code to remove extraneous operations. The main goal of a code optimizer is to improve the efficiency. Hence, it always tries to reduce the number of instructions. Once the number of instructions is reduced, the code runs faster and occupies less space.

Examples of code optimizations are as follows:

Constant folding

I := 4 + J – 5; can be optimized as I := J – 1;

or

I = 3; J = I + 2; can be optimized as I = 3; J = 5

```
while (COUNT < MAX) do
   INPUT SALES;
   SVALUE := SALES * ( PRICE + TAX );
   OUTPUT := SVALUE;
   COUNT := COUNT + 1;
end;
```

can be optimized as

```
T :=PRICE + TAX;
while (COUNT < MAX) do
   INPUT SALES;
   SVALUE := SALES * T;
   OUTPUT := SVALUE;
   COUNT := COUNT + 1;
end;
```

Common sub-expression elimination
From:

```
   A := 100 * (B + C);
   D := 200 + 8 * (B + C);
   E := A * (B + C);
```

can be optimized as:

```
TEMP := B + C;
   A := 100 * TEMP;
   D := 200 + 8 * TEMP;
   E := A * TEMP;

   E := A * TEMP;
```

Strength reduction

```
2*x can be optimized as x + x
2*x can be optimized as shift left x
```

Mathematical identities

```
a * b + a * c can be optimized as a*(b + c)
a - b can be optimized as a + (-b)
```

1.4.4 Target Code Generator

The code generator's main function is to translate the intermediate representation of the source code to the target machine code. The machine code may be an executable binary or assembly code, or another high-level language. To design a target code generator, the

designer should have the complete knowledge about the target machine. Producing low-level code requires familiarity with machine-level issues such as

- data handling
- machine instruction syntax
- variable allocation
- program layout
- registers
- instruction set

This phase takes each IR statement and generates an equivalent machine instruction.

For the following source text

```
if (x <= 3) then y := 2 * x;
else y := 5;
```

The target code generated is

Address	Instruction
10	CMP x 3
11	JG 14
12	MUL 2 x temp
13	JMP 16
14	MOV y 5

To design a compiler, all the above phases are necessary; additionally, two more routines are also important. They are symbol table manager and error handler.

1.4.5 Symbol Table Manager and Error Handler

In addition to passing the data from one phase to other, additional information acquired during a phase may be needed by a later phase. The **symbol table manager** is used to store the record for each identifier, procedure encountered in the source program with relevant attributes. The attributes of identifier can be name, token, type, address, and so on. For procedures, a record is stored in the symbol table with attributes like return type, number of parameters, type of each parameter, and so on. Whenever any phase encounters new information, records are stored in the symbol table and whenever any phase requires any information, records are retrieved from the symbol table. Syntax analyzer stores the type of information; this is later used by the semantic analyzer for type checking.

The **error handler** is used to report and recover from errors encountered in the source program. A good compiler should not stop on seeing the very first error. Each phase may encounter errors, so to deal with errors and to continue further, error handlers are required in each phase.

The process of compilation can be viewed in the analysis and synthesis model shown in Figure 1.9.

Two-Stage Model of Compilation

1. **Analysis**: The analysis part of compiler analyses breaks up the source program into constituent pieces and creates an intermediate representation of source text.

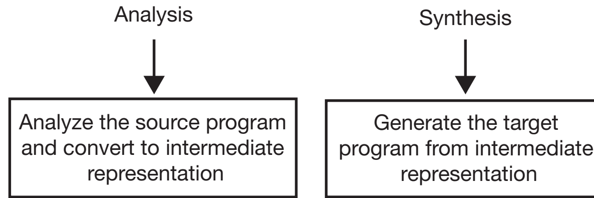


Figure 1.9 Phases of a Compiler Compilation

2. **Synthesis**: Synthesis part of compiler constructs the desired target machine code from the intermediate representation.

Most compilers are designed as two-stage systems. The first stage focuses on analyzing the source program and transforming it into intermediate representation. This is also called the front end of the system. The second stage focuses on the synthesis part where the intermediate code is converted to the target code.

The two-pass model simplifies the design of the compiler and helps in generating a new compiler for a new source language or to a new target machine with ease. The front end is designed with a focus on the source language to check for errors and generate a common intermediate representation that is error free to the back end. The back end can be designed to take the intermediate code that may be from any source language and generate the target code that is efficient and correct. Both front end and back end can use the common optimizer, which works on intermediate representation.

Analysis of the Source Program
Analysis deals with analyzing the source program and preparing the intermediate form.

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

Synthesis
Synthesis concerns issues involving generating code in the target language. It usually consists of the following phases:

- Intermediate code generation
- Code optimization
- Final code generation

1.4.6 Compiler Front End

The compiler front end shown in Figure 1.10 consists of multiple phases, each provided by the formal language theory:

1. Lexical analysis—breaking the source code text into small pieces called tokens, each representing a single atomic unit of the language, for instance, a keyword, identifier, or symbol name. The tokens are specified typically as regular expressions. From a regular

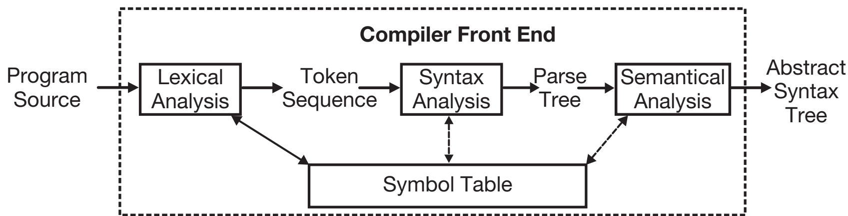


Figure 1.10 The Front End of a Compiler

expression, a finite state machine is constructed. This machine can be used to recognize tokens. This phase is also called scanner or lexical analysis.
2. Syntax analysis—Syntax analyzer checks the syntactic structures of the source code. It focuses only on the form or structure. In other words, it identifies the order of tokens and builds hierarchical structure called parse tree. This phase is also called parsing.
3. Semantic analysis is to recognize the *meaning* of programming constructs and start to prepare for output. In that phase, static type checking is done and most of the compilers show these semantic errors as incompatible types.
4. Intermediate representation—source program is transformed to an equivalent of the original program called intermediate representation (IR). This IR makes the target code generation easier. Intermediate code can be a data structure (typically a tree or graph) or an intermediate language.

1.4.7 Compiler Back End

Compiler front end alone is necessary for a few applications like language verification tools. A real compiler carries the intermediate code produced by the front end to the back end, which produces a functional equivalent program in the output language. This is done in multiple steps.

1. **Compiler**—The process of collecting program information from the intermediate representation of the source code is called compiler analysis. Typical analysis includes variable u-d (use-define) and d-u (define-use) chains, alias analysis, dependency analysis, and so on. The best possible way for any compiler optimization is an accurate variable analysis. During the analysis phase call graph and control flow graph are usually built.
2. **Optimization**—The optimizer optimizes the code. It transforms the intermediate language representation into functionally equivalent faster and smaller code.
3. **Code generation**—This is the transformation from intermediate code to the target/machine code. This involves the proper usage of registers, memory, and resources and proper selection of the machine instructions and addressing modes that reduce the running cost.

Prerequisite for any compiler optimization is compiler analysis and they tightly work together. The dependency of compiler front end and back end is shown in Figure 1.11(a).

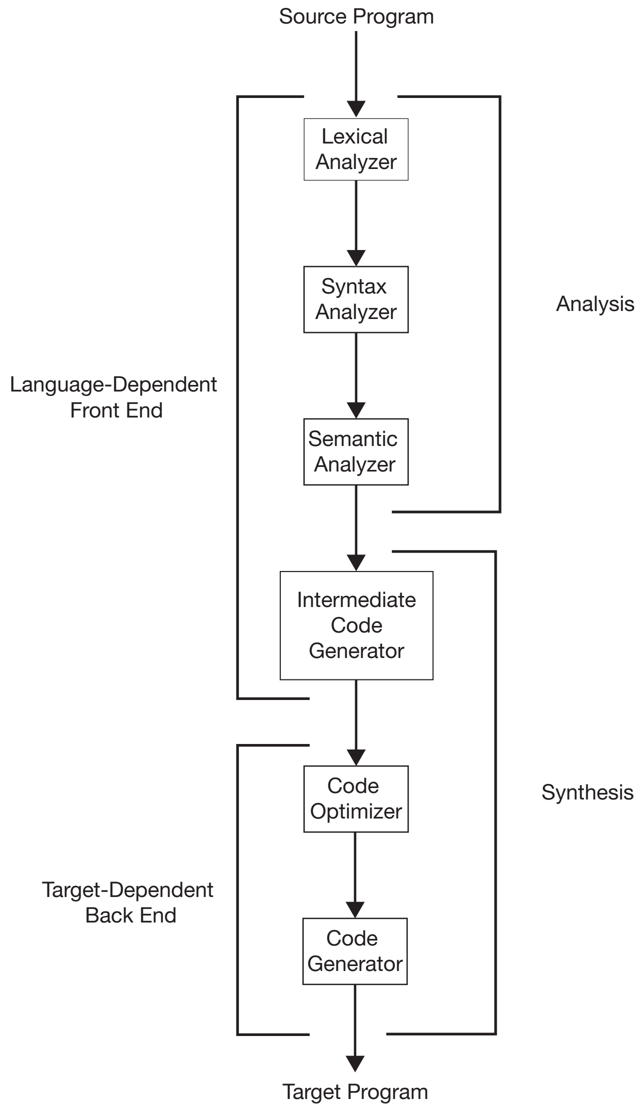


Figure 1.11(a) Compiler Front End and Back End

Example 3: Compiler phases:

```
Int a,b,c;
Real d;
---
a = b * c + d
```

output of each phase is shown below in Figure 1.11(b).

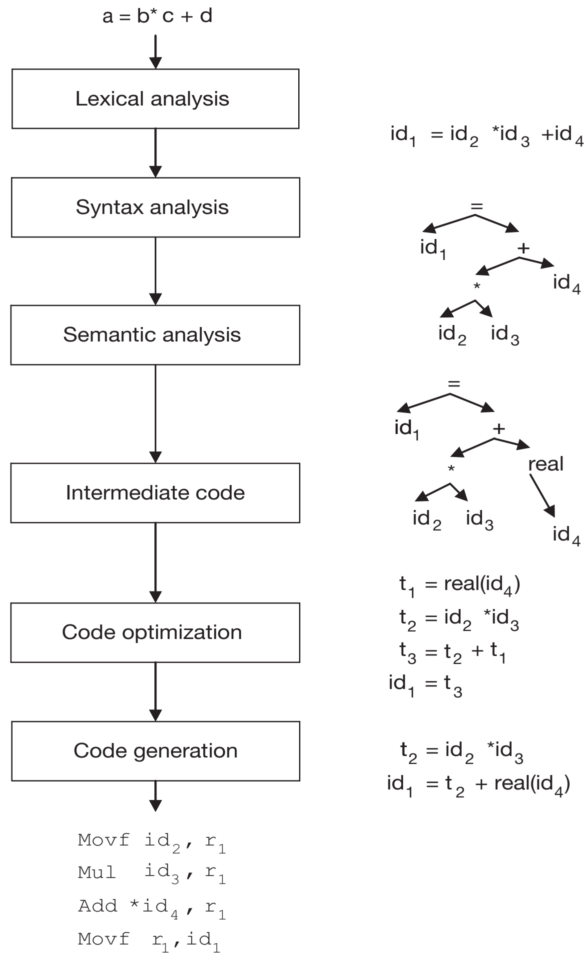


Figure 1.11(b) Result of Each Phase

1.5 Design of Passes

Reading the source text once is called a pass. All translators are designed in passes.

For example, the assembler is designed in two passes. Suppose there is a program as follows:

```
     Add R1,R2
     Jmp L
     ------
     ------
L:
     ------
```

Here Jmp L cannot be translated in first pass as address of L is not known. It is defined later. This is called forward reference problem. Because of this forward reference, the assembler is designed in two passes. In the first pass, the assembler collects all the labels and stores their addresses in the symbol table. In the second pass, the assembler completes the translation to machine code.

The assembler can be designed even in one pass. The process is to scan the input text line by line and translate. If the statement is seen with a forward reference, store it in some table-"not_yet_assembled" and continue with next statement. While doing so, if any label is seen, store it in the symbol table. At the end, since all the labels are available in the table, take each statement in the "not_yet_assembled" table and translate. But the problem here is if every statement has a forward reference, the size of the table becomes a problem. This is the reason why generally the assembler is designed in two passes.

A macro processor or loader is also designed in two passes.

Coming to the compiler, if we want to complete all the phases in one pass, it requires too much of space. This is the reason why a compiler is designed in two passes. Lexical analysis to intermediate code generation, that is, lexical analysis, syntax analysis, semantic analysis, and intermediate code generation is carried out in the first pass and target code generation and optimization in the second pass. This is how we can group phases into passes.

1.6 Retargeting

Creating more and more compilers for the same source language but for different machines is called retargeting**.** Retargetable compiler, also called *cross compiler,* is a compiler that can be modified easily to generate code for different architecture. The code generated is less efficient.

For example: If there is a C compiler that produces code for poor machine T800, 8 bit processor machine, then we want a compiler for a better machine, that is, i860–64 bit processor machine. To design a C compiler for i860, we do not have to start the design from lexical analysis. As front end is independent of target machine and completely dependent on source language, take the front end of T800 and design the back end for i860 machine. Add these two to get the C compiler for i860. This is the advantage of retargeting.

Typically the design of a compiler divides the functionality so that the code generation is separate from parsing and semantic checking. Polymorphism in the code-generating section is used in the design and implementation of a re-targetable compiler, so that a code can be generated for a processor of any type.

Examples of retargetable compilers:

- GCC
- lcc
- vbcc

1.7 Bootstrapping

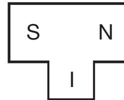
A compiler is a complex program and it is not desirable to use the assembly language for its development. Rather, a high-level language is used for writing compilers.

The process by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is known

as **bootstrapping**. Using the facilities offered by a language to compile itself is the essence of bootstrapping.

1.7.1 T-diagram

For bootstrapping purposes, a compiler is characterized by three languages: the source language that it compiles (S), the target language it generates code for (N), and the implementation/host language (I). This characteristic can be represented by using a diagram known as the T-diagram.



A compiler may run on one machine and produce code for another machine. Such a compiler is called a **cross compiler**.

Suppose we write a cross compiler in S in implementation language I to generate code for machine N, i.e., we create $S_I N$. If an existing compiler for I runs on machine M and generates code for M, it is characterized by $I_M M$. If $S_I N$ runs on $I_M M$, we get a compiler $S_M N$, that is, a compiler from S to N runs on M. This is illustrated by putting T-Diagrams together as shown in Figure 1.12.

$$S_I N + I_M M = S_M N$$

Note that the implementation language I of the source compiler and the source language of the running compiler must be the same.

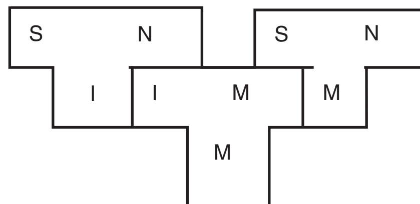


Figure 1.12 Cross Compiler

Example 4: Suppose we have a Pascal translator written in C, which takes pascal code and produces C as output.

If we want to have the same Pascal compiler in C++, then run the Pascal compiler under the C compiler, which produces C++ as output. Implementation language can be any language like Java. Once we run the Pascal compiler under the C compiler, we get Pascal compiler in C++. This is shown as a T diagram in Figure 1.13.

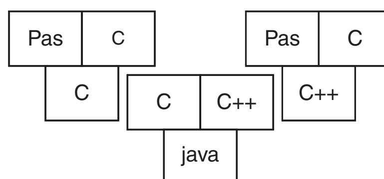
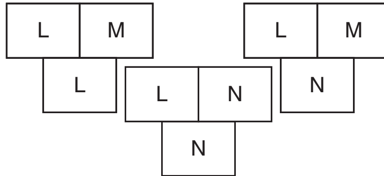


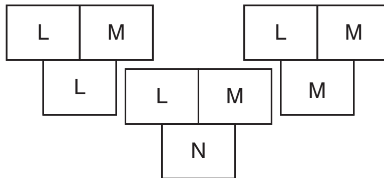
Figure 1.13 Pascal Compiler in C++

1.7.2 Advantages of Bootstrapping

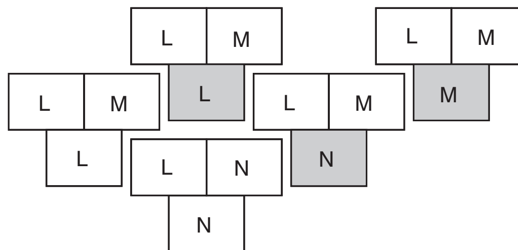
1. It builds up a compiler for larger and larger subsets of a language.
   Suppose we write a compiler $L_N N$ for language L in L to generate code for machine N. Development takes place on machine M, where an existing compiler $L_L M$ for L runs and generates code for M. By first compiling $L_L M$ with $L_N N$, we obtain a cross compiler $L_N M$ that runs on N, but produces code for M.



   The compiler $L_N M$ can be compiler second time, this time using generated cross compiler. The result of second compilation is a compiler $L_M M$ that runs on N and generates code for M.



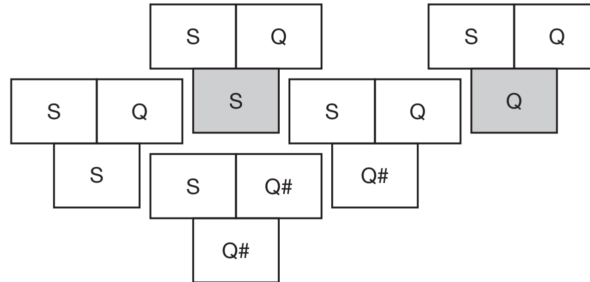
   The two steps can be combined as shown in the figure below.



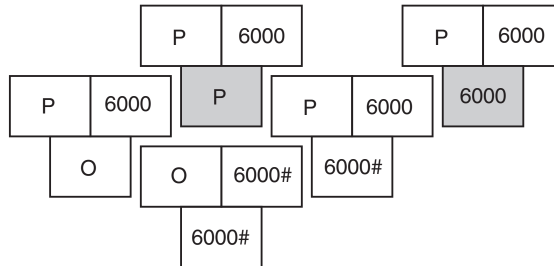
2. Using bootstrapping an optimizing compiler can optimize itself.

   Suppose all development is done on machine Q. We have $S_S Q$, a good optimizing compiler for language S written in S for machine Q. We want $S_Q Q$ a good optimizing compiler for S written in Q.

   We create $S_{Q\#} Q\#$, a quick and dirty compiler for S on Q that not only generates poor code but also takes a long time to do so. (Q# indicates poor implementation in Q. $S_{Q\#} Q\#$ is a poor implementation of the compiler that generates poor code.) However, we can use the compiler $S_{Q\#} Q\#$ to obtain a good compiler for S in two steps shown in the following T diagram.



Example 5:
Let us see how the process of bootstrapping gives us a clean implementation of Pascal. A fresh compiler was written in 1972 for CDC 6000 series machines by revising Pascal. In the following diagram, O represents "old Pascal " and P represents "revised Pascal."



A compiler for revised Pascal language was written in old Pascal and translated into CDC as 6000#, that is, P6000#6000#. The old compiler did not generate efficient code. Therefore, the compiler speed of 6000#, that is, P6000#6000# was rather moderate and its storage requirements were high. Revisions to Pascal were small enough that compiler P06000 and run through the inefficient compiler P6000#6000#. This gives a clean implementation of a compiler for the required language.

1.8 Compiler Design Tools

Many automated tools are available to design and enhance the performance of a compiler. Some of them are as follows:

1. **Automatic Parser Generators**

   With the help of CFG, these generators produce syntax analyzers (parse tree). Out of all phases of a compiler, the most complex is parsing. In old compilers, syntax analysis consumed not only a fraction of the running time of a compiler but also a large fraction of the intellectual effort of writing a compiler.
   For example, YACC, Bison

2. **Scanner Generators**

   These generators automatically generate lexical analyzers (the stream of tokens), normally from a specification based on regular expression.
   For example, Lex, Flex

3. **Syntax-Directed Translation Engines**

   These engines take the parse tree as input and produce intermediate code with the three-address format.

4. **Automatic Code Generators**

   These take a collection of rules that define the translation of each operation of the intermediate language for the target machine.

5. **Data-Flow Engines**

   To perform good code optimization, we need to understand how information flows across different parts of a program. The "data flow engines" gather the information about how values are transmitted from one part to other parts of the program. This is very useful for good code optimization.

6. **Global Optimizers**

   Global optimizer is language and compiler independent. It can be retargeted and it supports a number of architectures. It is useful if you need a back end for the compiler that handles code optimization,

These are the various compiler construction tools, which are widely used.

1.9 Modern Compilers—Design Need for Compilers

A compiler is one of the most important system software that translates the programs written in high-level language to low-level language. Most of the techniques are for procedure-oriented languages like C, Pascal and so on. Researchers are working in the direction of generating compilers for new paradigms of the language and generating an efficient code in terms of execution time, power consumption, and space requirement. Modern compilers have to be designed with different orientation as the languages are object based or object oriented. It becomes essential for the compiler designer to focus on the following issues:

- Focus on essential traditional and advanced techniques common to all language paradigms
- Consider programming types—procedural, object oriented, functional, logic and distributed languages
- Understand different optimizing techniques and tools available
- Understand various computer architectures and formats

1.10 Application of Compiler Design Principles

The important component of programming is language processing. Many application software and systems software require structured input. Few examples are databases (query language processing), OS (command line processing), Equation editors, XML- and html-based

systems, typesetting systems like Troff, Latex, editors like vi,Emacs, Awk, Sed, Form processing, extracting information automatically from forms. If input of any application has a structure, then language processing can be done on the input. The whole spectrum of the language-processing technology is used by compilers.

By studying the complete structure of compilers and how various parts are composed together to get a compiler, one can become confident of using the language-processing technology for various software developments. This knowledge helps to design, develop, understand, modify/enhance, and maintain compilers for (even complex!) programming languages.

Solved Problems

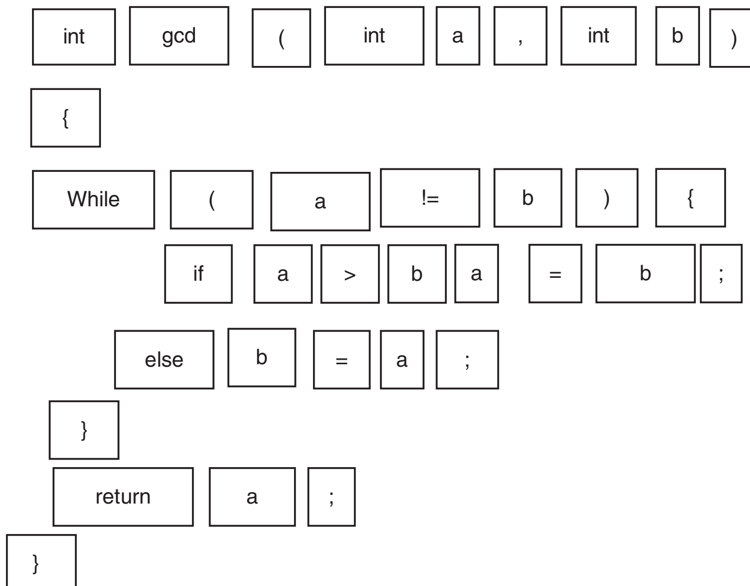
1. Show the output of each phase of compiler on the following input.

```
int gcd(int a, int b)
{
while (a != b) {
   if (a > b) a =b;
   else b =a;
}
return a;
}
```

Solution:

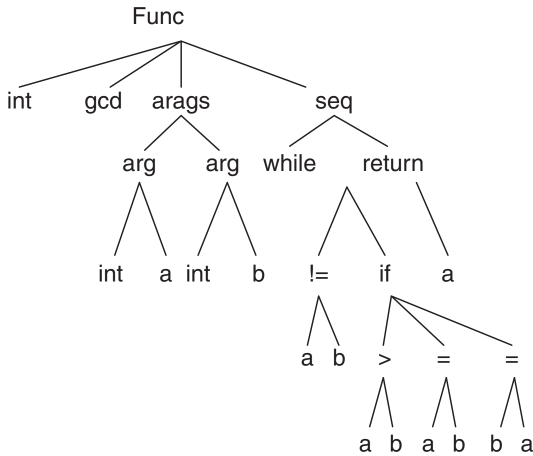
The first phase: Lexical Analysis Gives Tokens

Input to LA:



A stream of tokens. Whitespace, comments are removed.

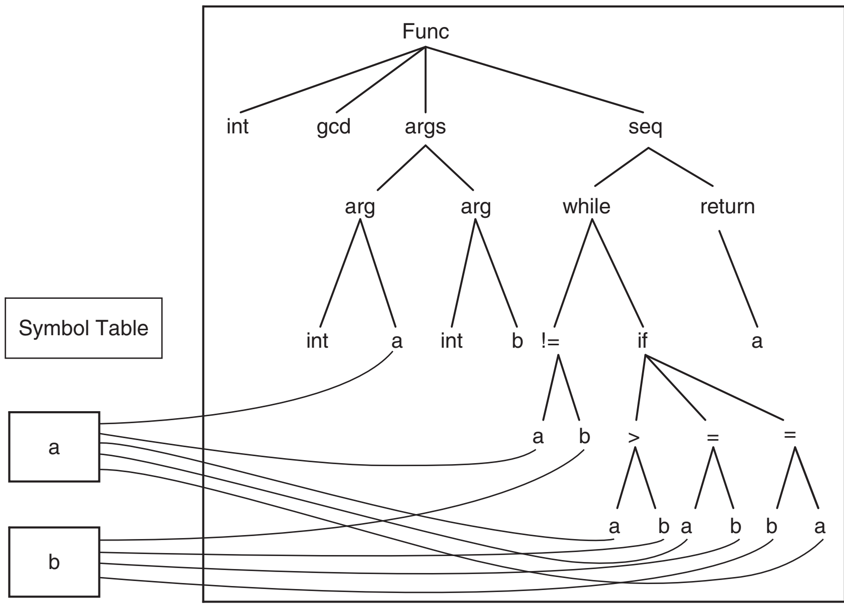
The second phase: Syntax Analysis Gives the Syntax Tree



```
int gcd(int a, int b)
   {
    while (a != b) {
    if (a > b) a = b;
        else b =a;
    }
        return a;
}
```

Abstract syntax tree built from parsing rules.

The third phase: Semantic Analysis Resolves Symbols



Translation into Three-Address Code

```
If (a !=b) go to L1
  go to Last
L1: if a > b goto L2
  goto L3
L2: a=b
  goto Last
L3: b=a
Last: goto calling program
```

Code optimization phase also will have the same output as there is no scope for any optimization. Now target code generator generates instructions as follows:

Translation into Target Code

```
mov a, R0              % Load a from stack
mov b, R1              % Load b from stack
cmp, !=, R0,R1
jmp .L1                % while (a != b)
jne .L5                % if (a < b)
subl %edx,%eax         % a =b
jmp .L8
.L5: subl %eax,%edx    % b =a
jmp .L8
.L3: leave             % Restore SP, BP
ret
```

Idealized assembly language with infinite registers

Summary

- A *compiler* is a program that translates a source program written in some high-level programming language (such as C++) into machine code.
- An interpreter reads source text line by line and executes without producing machine code.
- A preprocessor processes the source code before compilation and produces a code that can be more efficiently used by the compiler.
- An assembler is a translator that converts assembly code to machine code.
- A loader/linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.
- A phase is one which converts one form of representation to another form.
- The task of a compiler can be divided into six phases—lexical analysis, parsing, semantic analysis, intermediate code generation, code optimization, target code generation.

- The lexical analyzer or scanner groups the input stream (of characters) into a stream of tokens.
- The parser groups tokens into hierarchical structure called the parse tree.
- The semantic analyzer checks semantics, that is, whether the language constructs are meaningful.
- Intermediate Code Generator converts hierarchical representation of source text, that is, parse tree into a linear representation called **intermediate representation** (**IR**) of the program.
- **Intermediate Code Generator** can be represented in different ways (1) as a syntax tree, (2) DAG (directed acyclic graph), (3) postfix notation, and (4) Tree address code.
- **Code optimizer** tries to improve the efficiency of the code.
- **Target code generator** takes each IR statement and generates equivalent machine instruction.
- The **symbol table manager** is used to store record for each identifier and for every procedure encountered in the source program with relevant attributes.
- The **error handler** is used to report and recover from errors encountered in the source.
- A retargetable compiler is a compiler that can relatively easily be modified to generate code for different CPU architectures.
- A compiler that runs on one machine and produces code for another machine is called a **cross compiler**.

Fill in the Blanks

1. A compiler is a translator that converts source program to ________________.
2. ______________ reads source text line by line and executes the source text without producing machine code.
3. A preprocessor processes the ______________ and produces a code that can be more efficiently used by the compiler.
4. An assembler is a translator that converts assembly code to ______________.
5. ______________ is a program that takes one or more objects generated by a compiler and combines them into a single executable program.
6. A phase is one that converts______________.
7. The task of a compiler can be divided into ______________ phases.
8. The lexical analyzer or scanner groups the input stream into______________.
9. The parser groups tokens into a hierarchical structure called the ______________.
10. The semantic analyzer checks ______________.
11. Intermediate Code Generator can be represented in a tree form as ______________, ______________.
12. ______________ is an optional phase in a compiler.
13. Information in the symbol table is filled by ________________ phases.
14. The error handler is used to report and recover from errors encountered in the source. It is required in the ______________ phase.

15. A retargetable compiler is a compiler that can easily be modified to generate code for ______________.
16. A compiler that runs on one machine and produces code for another machine is called ______________.

Objective Question Bank

*1. Consider line number three of the following C program.

```
int main()
{ int I,N;
fro(I=0; I<N; I++);
}
```

Identify the compiler response about this line while creating the object module.
(a) No compilation error (b) only lexical error
(c) only syntactic error (d) both lexical and syntactic errors.

2. Output of an interpreter is
(a) HLL (b) Exe code (c) assembly code (d) machine code

3. Syntax analyzer verifies syntax and produces
(a) syntax tree (b) parse tree
(c) DAG (d) none

4. Intermediate Code Generator prepares IR as
(a) polish notation (b) dependency graph
(c) parse tree (d) all

5. Cross compiler uses which technique?
(a) backtracking (b) bootstrapping
(c) reverse engg (d) forward engg

6. Retargeting is useful for creating compilers for more and more
(a) CPU architectures (b) high-level languages
(c) assembly languages (d) none

7. Code optimization can be performed on
(a) source text (b) IR
(c) assembly code (d) all

8. Assembler can be designed in
(a) one pass (b) two pass
(c) many pass (d) all

*These questions have appeared in GATE examinations.

9. YACC is

   (a) scanner (b) parser generator
   (c) Intermediate Code Generator (d) none

10. Lex is a

   (a) scanner (b) parser generator
   (c) Intermediate Code Generator (d) none

Exercises

1. Explain the difference between a compiler and an interpreter.
2. What is the difference between pass and a phase? Explain the design of a compiler with phases.
3. Explain the different translators required for converting a high-level language to an executable code.
4. Give the output of each phase of compiler for the following source text
   for (i = 0; i < 10; i++) a = a + 10;
5. What is a cross compiler? What is the advantage of the bootstrapping technique with compilers?
6. What is retargeting? What is its use?
7. Give the output of each phase of compiler for the following source text:
   while (i < 10) a = a + 10;
8. What are the tools available for compiler construction?
9. Explain the analysis and synthesis model of a compiler.
10. What is the front end and the back end of a compiler? Explain them in detail.

Key for Fill in the Blanks

1. machine code
2. Interpreter
3. source code before the compilation
4. machine code
5. Link edit or
6. one form of representation to another form
7. 6
8. a stream of tokens
9. parse tree
10. semantics
11. DAG, Syntax tree
12. Code optimizer
13. all
14. all
15. different CPU architectures
16. cross compiler

Key for Objective Question Bank

1. c	2. b	3. b	4. a	5. b
6. a	7. a	8. d	9. b	10. a

CHAPTER 2

Lexical Analyzer

This is the first phase of a compiler. The compiler spends most of its time (20–30% of compile time) in this phase because reading character by character is done only in this phase. If a lexical analyzer is implemented efficiently, the overall efficiency of the compiler improves. Lexical analyzers are used in text processing, query processing, and pattern matching tools.

CHAPTER OUTLINE

2.1 Introduction
2.2 Advantages of Separating Lexical Analysis from Syntax Analysis
2.3 Secondary Tasks of a Lexical Analyzer
2.4 Error Recovery in Lexical Analysis
2.5 Tokens, Patterns, Lexemes
2.6 Strategies for Implementing a Lexical Analyzer
2.7 Input Buffering
2.8 Specification of Tokens
2.9 Recognition of Tokens
2.10 Finite State Machine
2.11 Lex Tool: Lexical Analyzer Generator

The scanner or lexical analyzer (LA) performs the task of reading a source text as a file of characters and dividing them up into tokens. Tokens are like words in natural language. This chapter deals with issues to be taken care of in the design of a lexical analyzer. Here we shall discuss how to specify tokens using regular expressions, how to design LA using Finite Automata (FA) or Lex tool. It is easy to design a pattern recognizer as Nondeterministic Finite Automata (NFA). But NFA is slower than Deterministic Finite Automata (DFA). Hence NFA, NFA with ε-transitions, inter conversions of NFA and DFA, and minimal DFA are also discussed. A software tool that automates the construction of a lexical analyzer, that is, LEX is also discussed. This tool allows people with different backgrounds to use pattern matching in their own application areas.

2.1 Introduction

Lexical analysis is the act of breaking down source text into a set of words called tokens. Each token is found by matching sequential characters to patterns. In general, programming languages are defined using Context-free grammars, and these include the regular languages. All the tokens are defined with regular grammar and the lexical analyzer identifies strings as tokens and sends them to a syntax analyzer for parsing. A typical lexical analyzer or scanner is shown in Figure 2.1.

Don't think that a lexical analyzer reads the complete text and sends stream of tokens. It is not so. The interaction of a lexical analyzer with a parser is shown in Figure 2.2. Only on getting a request from the parser, the lexical analyzer reads the next token and sends the next token to parser. If the token is an identifier or a procedure, then the lexical analyzer stores that token in the symbol table.

For example, if there is a source text A + B, LA does not read A + B at once and sends tokens: id, +, id. On getting the first request from the parser, the LA reads the first string A, recognizes that as the token id, stores it in the symbol table, and sends the token id to the parser. On the next request, it only reads + and sends the operator as it is as a token. On the third request, it reads string B, recognizes that as token id, stores it in the symbol table and sends the token id to the parser.

Out of all the phases, a compiler spends much time in lexical analysis. Hence, if this phase is implemented efficiently, this contributes to the overall efficiency of the compiler.

Lexical analysis and parsing can be combined in one phase but there are some advantages for doing it separately.

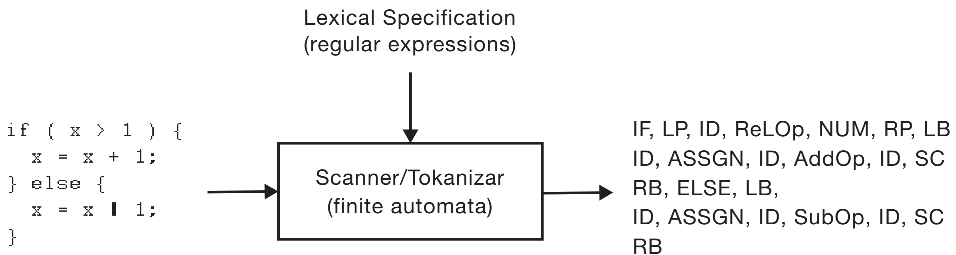


Figure 2.1 A Typical Lexical Analyzer

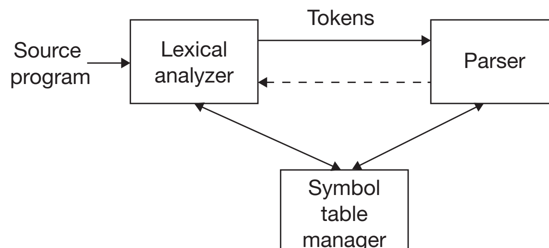


Figure 2.2 Interaction between a Lexical Analyzer and a Parser

2.2 Advantages of Separating Lexical Analysis from Syntax Analysis

There are several reasons for separating the analysis phase into lexical analysis and parsing.

1. **Simplicity**: By having LA as a separate phase, the compiler design is simplified. Techniques of DFA are sufficient for lexical analysis and techniques of PDA can be used for syntax analysis. Designing them together is a complex process. By dividing that complex task into simpler subtasks, design is simplified. Separation also simplifies the syntax analyzer and allows us to use independent tools.
2. **Efficiency:** Separating lexical analysis from syntax analysis increases design efficiency. Separation into different modules makes it easier to perform simplifications and optimizations unique to the different paradigms. For example, a compiler spends much time in LA. If this module is implemented efficiently, then this contributes to the overall efficiency of a compiler.
3. **Portability**: Portability is enhanced. Due to input/output and character set variations, lexical analyzers are not always machine independent. We can take care of input alphabet peculiarities at this level.

The most important advantage is that separation of these phases helps in designing special tools to automate the construction of lexical analyzers and parsers.

2.3 Secondary Tasks of a Lexical Analyzer

The main task of LA is to read the source text and break it to stream of tokens. In addition to this it even performs the following secondary tasks.

1. Filtering comment lines and white space characters.
   White space characters like tab, space, newline characters, and comment lines are not required during the translation of a source code. Hence, a lexical analyzer stripes them off the code.
2. Associating the error messages in other phases.
   A lexical analyzer reads the source code character by character. While reading it even remembers the line numbers. If an error is seen by any other phase, then the lexical analyzer helps other phases in giving error diagnostics properly.

2.4 Error Recovery in Lexical Analysis

Generally, errors often detected in a lexical analysis are as follows:

1. Numeric literals that are too long
2. Long identifiers (often a warning is given)
3. Ill-formed numeric literals
4. Input characters that are not in the source language

How to recover from errors?

There are four approaches to recover from lexical errors:

- **Delete**: This is the simplest and most important operation. If there is an unknown character, simply delete it. Deletion of character is also called panic-mode. This is used by many compilers. However, it has certain disadvantages:
  - The meaning of the program may get changed.
  - The whole input may get deleted in the process.

  Example: "charr" can be corrected as "char" by deleting "r"
- **Insert**: Insert an extra or missing character to group into a meaningful token.

  Example: "cha" can be corrected as "char" by inserting "r"
- **Transpose**: Based on certain rules, we can transpose two characters. Like "whiel" can be corrected to "while" by the transpose method.
- **Replace**: In some cases, it may require replacing one character by another.

  Example: "chrr" can be corrected as "char" by replacing "r" with "a".

Let us consider an example here.

Example:

```
        if a statements;
          else statements;
```

Here, to recover from the error, we need to insert a blank between "if" and "a." However, if we use the delete approach, then since 'ifa' is an identifier, it can't be followed just by an identifier, so *statements* is deleted. Next, else can't occur without an "if" preceding it, so this *else* is also deleted. So in this way we end up deleting a substantial portion of the program.

We can have different error handlers to handle different errors that occur at various stages. The error handler need not be totally generic. In lexical, the above four operations are based on characters, while in syntactical, these operations are based on tokens.

Certain desired goals for error recovery:

- Error recovery should be accurate.
- Location of error reported should be precise.
- Error recovery should be quick (the algorithm shouldn't look into the whole code for error recovery).
- Error recovery should not lead to error cascade (i.e., you recover from one error and get trapped into other errors and so on).

2.5 Tokens, Patterns, Lexemes

When talking about lexical analysis, most often we use the following terms: lexeme, token, and pattern. It is important to understand these terms.

Token: It is a group of characters with logical meaning. Token is a logical building block of the language.

Example: **id**, **keyword**, **Num** etc.

Pattern: It is a rule that describes the character that can be grouped into tokens. It is expressed as a regular expression. Input stream of characters are matched with patterns and tokens are identified.

Example: Pattern/rule for **id** is that it should start with a letter followed by any number of letters or digits. This is given by the regular expression: [A – Za – z][A – Za – z 0 – 9]*.

Using this pattern, the given input strings "xyz, abcd, a7b82" are recognized as token id.

Lexeme: It is the actual text/character stream that matches with the pattern and is recognized as a token.

For example, "int" is identified as token keyword. Here "int" is lexeme and keyword is token.

For example, consider the following statement.

float key = 1.2;

Lexeme	Token	Pattern
float	float	Float
key	id	A letter followed by any number of letters or digits
=	relop	< | > | <= | >= | = | != | ==
1.2	num	Any numeric constant
;	;	;

Design complexity of a lexical analyzer is mainly dependent on language conventions.

A popular example that illustrates the potential difficulty of LA design is that in many languages certain strings are reserved, that is, the meaning is predefined and cannot be changed by the user. If keywords are not reserved, then lexical analyzer must distinguish between a keyword and identifier. Example: Keywords are not reserved in PL/I. Thus, the rules for recognizing keywords from identifiers are quite complicated; it is understood by the following statement:

```
If then then then=else; else else=then;
```

Attributes for Tokens

When a token represents many lexemes, additional information must be provided by the scanner or lexical analyzer about the particular lexeme of a token that matched with the subsequent phases of the compiler.

For example, the token id matches with x or y or z. But it is essential for the code generator to know which string is actually matched.

The lexical analyzer accumulates information about tokens into their associated attributes. The tokens control parsing decisions, whereas the attributes show significance in translation of tokens. Generally, tokens will have only one attribute, that is, pointer to the symbol table entry in which information about the token is available. Suppose we want the type of id, token, lexeme, line number, all these information can be stored in the symbol table entry of that identifier.

Example 1:

Consider the statement:

```
distance = 0.5 * g * t * t;
```

When the lexical analyzer invokes **getnexttoken()** function for the next token, the sequence of tokens returned are as follows:

- **id ---------** (distance)
- =
- **Const ------** (0.5)
- *
- **id -----------** (g)
- *
- **id -------------**(t)
- *
- **Id -----------**(t)

Example 2:

Divide the following C code into tokens:

```
int  Square(a) int a {
    /* returns square of a, but is less than 100 */
    return(a<=-10||a>=10)?100:a*a;
}
```

Which of the following is NOT one of the tokens?
a. ?: **b.** /* **c.** never **d.** = **e.** /* returns……*/ **f.** <= **g.** 100 **h.**)

Solution:

a. ?:—is not a token. Although these symbols form one C operator, they are separated in the text, and need to be recognized separately by the lexical analyzer. Notice that, although they are fairly close in this example, there could be an arbitrarily long expression between them in the legal C code.

b. /* —is not a token. This symbol introduces the comment, but it is normal to strip out comments before tokenizing the source code.

c. never—is not a token. The word "never" looks like an identifier, but it is within a comment. Normally, comments are stripped out before tokenizing, but we would never separate a comment into parts during lexical analysis at any event.

d. = —is not a token. Although = might be a token by itself in some code, each use of = here is as part of a larger token.

e. /* returns x-squared but never more than 100*/– is not a token. The comment is normally stripped out before tokenizing the source code. It would not be returned as a token to the parser, because its presence does not affect the structure of the code; neither does it affect the object code to be produced.

f. <=—is a token.<= is the two-character "less-than-or-equals" operator, and is normally recognized as a single token.

g. 100—is a token. “100” is a constant. It would normally be returned as the token CONST, with a lexical value indicating that the actual value is 100.
h.)—is a token. Parentheses are always treated as tokens by themselves.

The list of tokens would be:

```
◆ float
  ID --------(limitedSquare)
  (
  ID --------(x)
  )
  float
  ID --------(x)
  {
  return
  (
  ID --------(x)
  <=
  CONST --------(-10.0)
  |
  ID --------(x)
  >=
  CONST --------(10.0)
  ?
  CONST --------(100)
  :
  ID --------(x)
  *
  ID --------(x)
  ;
  }
```

2.6 Strategies for Implementing a Lexical Analyzer

Lexical analyzer can be designed in the following ways:

1. Use a scanner generator tool like Lex/Flex to produce a lexical analyzer. The specification can be given using a regular expression, which on compilation generates a scanner capable of identifying tokens. The generator provides routines for reading and buffering the input. This is easy to implement but least efficient.
2. Design a lexical analyzer in a high-level programming language like C and use the input and buffering techniques provided by the language. This approach is intermediate in terms of efficiency.
3. Writing a lexical analyzer in assembly language is the most efficient method. It can explicitly manage input and buffering. This is a very complex approach to implement.

The above details provided for a lexical analyzer are in increasing order of complexity and efficiency.

Designing a lexical analyzer either by hand or automated tools mainly involves two steps:

1. Describe rules for tokens using regular expressions.
2. Design a recognizer for such rules, that is, for tokens. Designing a recognizer corresponds to converting regular expressions to Finite Automata. The processing can be speeded if the regular expression is represented in Deterministic Finite Automata. This involves the following steps:
   - Convert regular expression to NFA with ε
   - Convert NFA with ε to NFA without ε
   - Convert NFA to DFA

2.7 Input Buffering

In order to recognize a token, a scanner has to *look ahead* several characters from the current character many times. For example, "char" is a keyword in C, while the term "chap" may be a variable name. When the character "c" is encountered, the scanner cannot decide whether it is a variable, keyword, or function name until it reads three more characters. It takes a lot of time to read character by character from a file, and so specialized *buffering techniques* are developed. These buffering techniques, makes the reading process easy and also reduces the amount of overhead required to process. There are many buffering schemes that can be used; since these techniques are somewhat dependent on system parameters, we shall discuss one scheme here.

Look ahead with 2N buffering

We use a buffer divided into two N-Character halves as shown in Figure 2.3. Typically, N is the number of characters on one disk block, for example, 1024 or 4096. We read N input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character. If less than N characters remain in the input, then special characters **EOF** is read into the buffer after the input characters, as in Figure 2.3. **eof** marks end of source file and is different from any input character.

Two pointers "**lexeme**" and "**fwd**" to the input buffer are maintained. The string of characters enclosed between the two pointers is the current lexeme. To find the lexeme, first initialize both the pointers with the first character of the input. Keep incrementing the **fwd** pointer until a match for the pattern is found. Once a character not matching is found, stop incrementing the pointer and extract the string between the "**lexeme**" and "**fwd**" pointers. This string is required the lexeme and process it and set both the pointer to the next character to identify the next lexeme. With this scheme, comments and white space can be treated as patterns that yield no token.

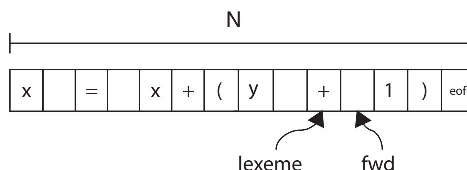


Figure 2.3 Interaction of Lexical Analyzer and Parser

While the **fwd** pointer is being incremented, if it is about to move past the halfway mark, the right half is filled with N new input characters. Else if the **fwd** pointer is about to move past the right end of buffer, not only the left half is filled with N new characters but also the **fwd** pointer wraps around the beginning of the buffer.

Most of the time this buffering scheme works quite well, but the amount of look ahead is limited with it. This limited look ahead may make it difficult to recognize tokens in cases where the distance that the **fwd** pointer must travel is more than the length of the buffer. For example, in PL/I program, consider the statement

DECLARE (ARG1, ARG2, ARG3……..ARGn)

Until we see what character follows the right parenthesis, we cannot determine whether DECLARE is a function name, keyword, or an array name. In all the cases, the lexeme ends at second E, but the amount of look ahead needed is proportional to the number of arguments, which in principle is unbounded.

Algorithm for moving the forward pointer "fwd"

```
If fwd is at end of first half
       reload second half;
       set fwd to point to beginning of second half;
Else if fwd is at end of second half
       load first half;
       set fwd to point to beginning of first half;
Else
       increment fwd pointer
```

This algorithm takes two tests each time to advance the fwd pointer. The performance can be improved by using other methods.

Buffer pairs

Sentinels

In order to optimize the number of tests to one for each advance of fwd pointer, sentinels are used with buffer. This is shown in Figure 2.4. The *idea* is to extend each buffer half to hold a *sentinel* at the end.

- One of the special characters that cannot occur in a program is Sentinel (e.g., EOF).
- It indicates the need for some special action (terminate processing or fill other buffer-half).

Algorithm for incrementing the forward pointer "fwd" that uses sentinels

```
increment fwd
If fwd is EOF
        If fwd is at end of first half
               reload second half
               set fwd to point to beginning of second half;
        Else if fwd is at end of second half
               reload first half;
               set fwd to point to beginning of first half;
        otherwise
               terminate the process.
```

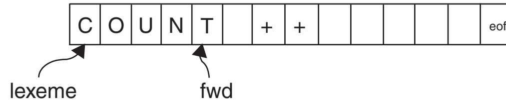


Figure 2.4 Sentinels in Input Buffering

This algorithm needs only one test per character.

2.8 Specification of Tokens

Patterns are specified using regular expressions. Each pattern matches a set of strings; so regular expressions will serve as names for sets of strings.

Strings and Languages

A **language** is a dynamic set of visual, auditory, or tactile symbols of communication and the elements used to manipulate them. *Language* can also refer to the use of such systems as a general phenomenon.

Symbol and Alphabet

A symbol is an abstract entity. It cannot be formerly defined as points in geometry.

Example: Letters, digits, or special symbols like $, @, # etc.

Alphabet: Finite collection of symbols denoted by Σ.

Example: English alphabet $\Sigma = \{a, b, \ldots\ldots z\}$
Binary alphabet $\Sigma = \{0, 1\}$

String/word: Set of symbols from the set of alphabets

Example: 1101, 1111, 01010101 strings from binary alphabet.
0a1 is not a string from binary alphabet.

A **word** over an alphabet can be any finite sequence, or string, or group of letters. The set of all words over an alphabet Σ is usually denoted by Σ^*. For any alphabet, there is only one word of length 0, the *empty word*, which is often denoted by e, ε, or Λ. An empty string can be denoted by €.

- Any string with any number of leading symbols of string is called **Prefix**.
- Any string with any number of trailing symbols of string is **Suffix**.
- Any substring except the string itself is **Proper** substring.

Example 1: String: "ramu"
Prefix: ε, r, ra, ram
Suffix: u, mu, amu
Substring: ε, r, ra, ram, ramu

2.8.1 Operations on Language

If L_1 and L_2 are two languages, then

i. **Union** of two languages is denoted as $L_1 + L_2$ or L_1 U L_2

By union we get the words from both languages.

ii. **Concatenation** of two languages is denoted as L_1L_2
Concatenation is a process of combining a word with other words to form a new word, whose length is the sum of the lengths of the original words. If we concatenate a word with an empty word, the result is the original word.

iii. **Kleene's closure** Σ^* is the language consisting of all words that are concatenations of 0 or more words in the original language (including null string €).

Example 2:

(i) $\Sigma = \{x\}$ $\Sigma^* = \{\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots\ldots\}$
$\Sigma^* = \{\varepsilon, x, xx, xxx \ldots\ldots\}$

(ii) $\Sigma = \{a, b\}$ $\Sigma^* = \{\varepsilon, a, b, aa, ab, bb, ba, aaa, aab, \ldots\ldots\}$

(iii) Positive closure $\Sigma^+ = \Sigma^* - \{\varepsilon\}$
$\Sigma^* = \Sigma^+ + \varepsilon$

L_1 = {smart, ugly} L_2 = {boy, girl}
$L_1 \cup L_2$ = {smart, ugly, boy, girl}
L_1L_2 {smart boy, smart girl, ugly boy, ugly girl}

Regular Expressions

1. Any atom (character) is a regular expression.
2. If r and s are regular expressions, then so are r. s, r + s, r* and (r).

The language described by a regular expression is called a *regular set*. The operators + and * are called *regular operators*.

Properties of Regular Expressions

If r and s are two regular expressions then

- r + s = s + r (union is commutative)
- r(s + t) = rs + rt, (s + t)r = sr + tr (concatenation distributes over union)
- r + (s + t) = (r + s) + t (union is associative)
- εr = r, rε = r (ε is the identity element for concatenation)
- r** = r* (closure is idempotent)

Regular Definitions: It is often desirable to name regular expression groupings for being able to recognize tokens at different levels.

Example: Identifiers in C are defined as a letter or underscore followed by zero or
More letters or underscores or digits.

Regular definition for id is as follows:

letter → A + B + C + : : : + Z + a + b : : : + z

digit → 0 + 1 + 2 + : : : + 9

id → (letter+)(letter+ +digit)_

Note: Be careful with recursive regular definitions! They may not be regular.

2.9 Recognition of Tokens

Lexical analysis can be performed with pattern matching through the use of regular expressions. Therefore, a lexical analyzer can be defined and represented as a DFA. Recognition of tokens implies implementing a regular expression recognizer. This entails implementation of a DFA.

Example of a lexical analyzer
Suppose that we want to build a lexical analyzer for the recognizing identifier, > =, >, integer const. The corresponding DFA that recognizes the above tokens is shown below in Figure 2.5.

How much should we match?
In general, find the longest match possible.
For example, given an input 123.45, should match with oken num_const(123.45) rather than num_const(123), "**.**", num_const(45).
The DFA for matching such numbers is shown below in Figure 2.6.

Implementing finite automata: We can hand code DFA as each state corresponds to a labeled code fragment and state transitions represented as control transfers as follows.

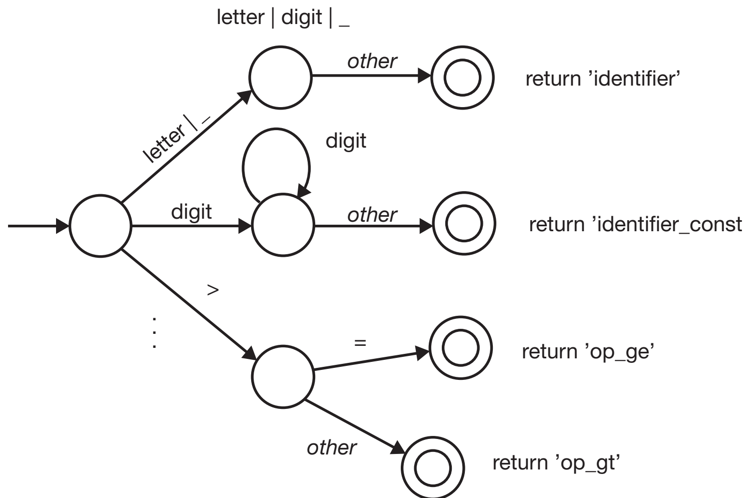


Figure 2.5 DFA That Recognizes Tokens id, integer_const, etc.

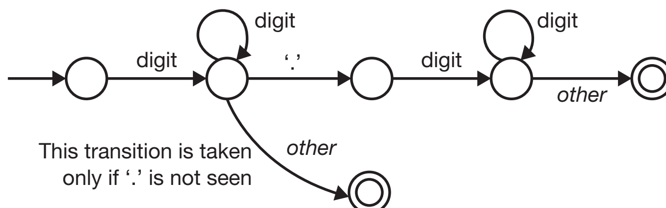
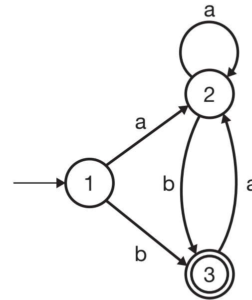


Figure 2.6 DFA That Recognizes Floating Point Numbers

```
int scanner ()
{ char c;
   while (TRUE){
   c = getchar();
   state1: switch (c) {    /*initial state*/
      case 'a': goto state2;
      case 'b' goto state3;
      default: Error();
      }
   state2: switch (c){
      case 'a': goto state2;
      case'b': goto state3;
      default: return SUCCESSFUL;
      }
   state3: switch (c){
      case 'a': goto state2;
      default: return SUCCESSFUL;
      }
}/*while*/
}
```



When the current state of automaton is a final state, then a match is found in DFA and no transition is enabled on the next input character.

Actions on finding a match:

- If the lexeme is valid, then copy it in an appropriate place where the parser can access it.
- Save any necessary scanner state so that scanning can subsequently resume at the right place.
- Return a value indicating the token found.

So the concept of finite automaton is essential for the design of a lexical analyzer. Hence, let us discuss about finite automata, its types, and inter conversions.

2.10 Finite State Machine

The finite state system is a mathematical model of a system with certain input and gives certain output finally. The input given to an FSM is processed by various states. These states are called intermediate states.

A good example of finite state systems is a control mechanism of elevator. This mechanism remembers only the current floor number pressed and it does not remember all the previously pressed numbers. The finite state systems are useful in design of text editors, lexical analyzers, and natural language processing. The word "automaton" is singular and "automata" is plural.

Definition: A finite automaton is formerly defined as a 5-tuple ($Q, \Sigma, \delta, q_0, F$) where

Q – is a finite set of states which is non empty

Σ – is input alphabet

q_0 – is initial state

F – is a set of final states and $F \subseteq Q$

δ – is a transition function or mapping function $Q x \Sigma \rightarrow Q$ using this the next state can be determined depending on the current input.

2.10.1 Finite Automaton Model

The finite automaton can be represented as in Figure 2.7.

i. **Input tape** is a linear tape having some cells that can hold an input symbol from ∑.
ii. **Finite control** is the finite control that indicates the current state and decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right and at any instance only one input symbol is read.

The reading head examines the read symbol and the head moves to the right side with or without changing the state. When the entire string is read and if finite control is in the final state then the string is accepted or else rejected. Finite automaton can be represented by a transition diagram in which the vertices represent the states and edges represent the transitions. We can model the real-world problems as finite automaton which helps in understanding the behavior and analyzing the behavior.

Example 3:
Lexical analyzer behavior can be shown as FA. Consider the lexical analyzer that matches words like “the”, “this”, “that”, and “to”. This is shown in Figure 2.8.

These systems are called Finite Automaton as the number of possible states and the number of letters in the alphabet are both finite. It is automaton because the change of state is totally governed by the input.

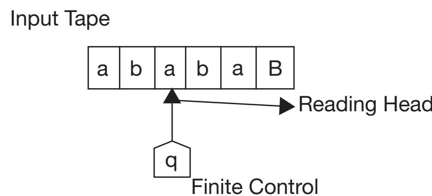


Figure 2.7 Finite Automaton

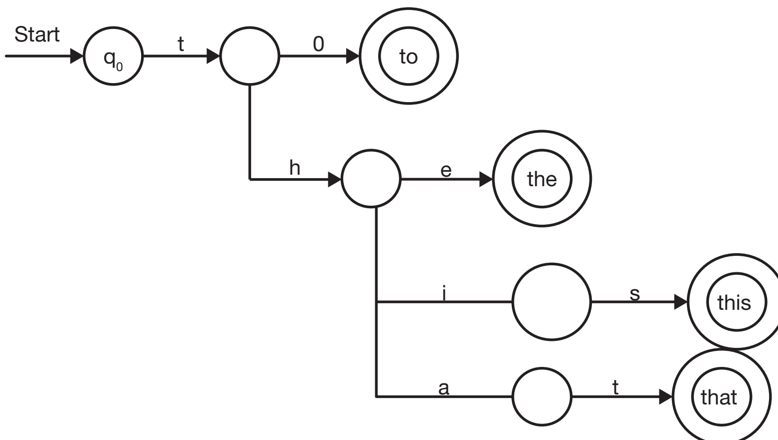


Figure 2.8 DFA that recognizes strings to, the, this, and that

2.10.2 Properties of the Transition Function "δ"

1. $\delta(q, \varepsilon) = q$. The states of FA are changed only by an input symbol.
2. For all strings **w** and i/p symbol a, $\delta(q, aw) = \delta(\delta(q, a), w)$

   An FA can be represented by a

   a. transition diagram

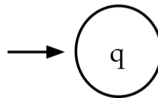
   b. transition table

2.10.3 Transition Diagram

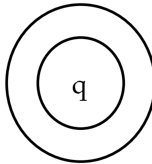
A transition graph contains

a. Set of states as circles

   Start state q_o with arrow



   Final state by double circle



b. A finite set of transitions (edges | labels) that show how to go from some state to other.

2.10.4 Transition Table

Following is a tabular representation where rows correspond to states and columns correspond to input. The start state is given by → and the final state by *

Example: $M=\{\{q_0, q_1, q_2\},\{a, b\}, \delta, q_0, \{q_2\}\}$

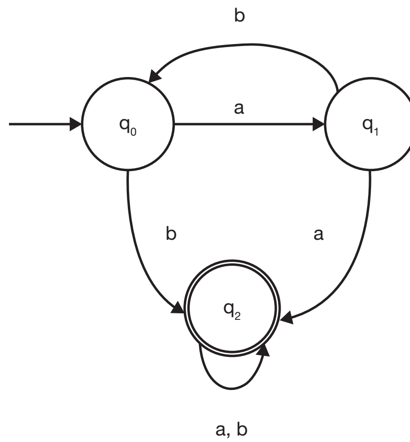
$\delta(q_0, a) = q_1 \;\; \delta(q_0, b) = q_2$

$\delta(q_1, a) = q_2 \;\; \delta(q_1, b) = q_0$

$\delta(q_2, a) = q_2 \;\; \delta(q_2, b) = q_2$

Δ/Σ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_2	q_0
$* q_2$	q_2	q_2

This able can be shown as a transition diagram as seen below.

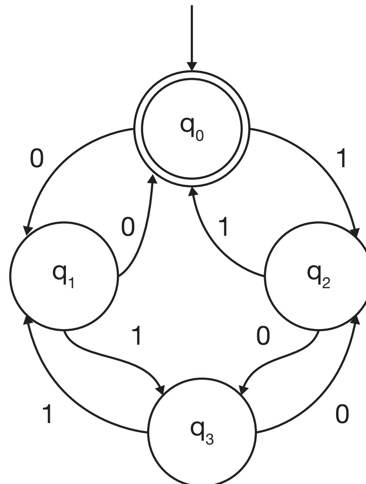


2.10.5 Language Acceptance

A string w is accepted by Finite Automaton U given as

U = {Q, Σ, δ, q_0, F} if δ (q_0, w) = P for some P in F. This concludes that the string is accepted when it enters into the final state on the last input element.

Example:



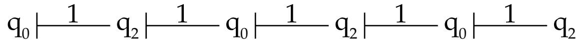
Let us check if input string 1010 is accepted or not

a. δ (q_0, 1010) = δ (q_2, 010) = δ (q_3, 10)= δ (q_1, 0) = q_0

$$q_0 \vdash\!\xrightarrow{1} q_2 \vdash\!\xrightarrow{0} q_3 \vdash\!\xrightarrow{1} q_1 \vdash\!\xrightarrow{0} q_0$$

Here q_0 is the final state. Hence, the string is accepted.

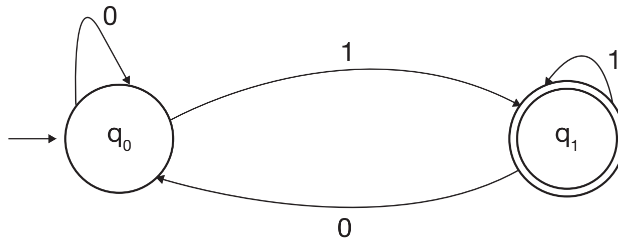
b. Check 11111



$q_2 \notin F$. Hence, the string is rejected.

Example 4:

Give the language defined by the following FA.



If we list different strings accepted by this automaton, we get

{1, 01, 0001, 10101, 011111………………}

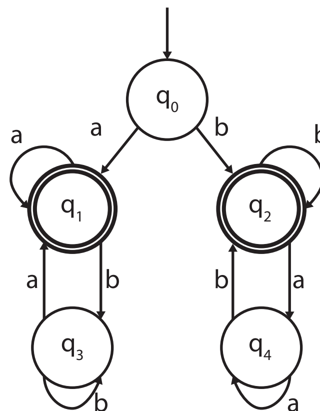
If we observe, all strings that are accepted always end with 1.

L(M) = {w /w ends with 1 on Σ = {0, 1} }.

Language accepted by machine M is L(M), which is the set of strings that are ending with 1.

Example 5:

Define language accepted by the following machine.



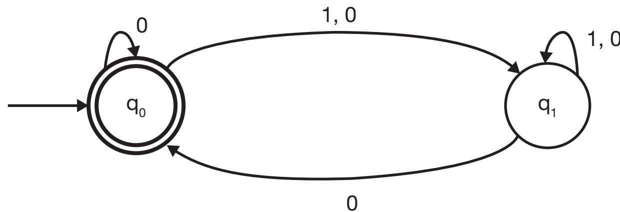
L(m) = {w/w contains all strings that start and end with the same symbol}

2.10.6 Finite Automation Is of Two Types

a. Deterministic finite automaton

b. Nondeterministic finite automaton

In DFA there will be unique transition in any state on an input symbol, whereas in NFA there can be more than one transition on an input symbol. Hence, DFA is faster than NFA. The above figure is an example of DFA. The following figure is an example of NFA.



In the above figure, in state q_0 on 0 it is either in state q_0 or in state q_1. Hence NFA.

2.10.7 Deterministic Finite Dutomaton (DFA)

Deterministic finite automation can be defined as quintuple

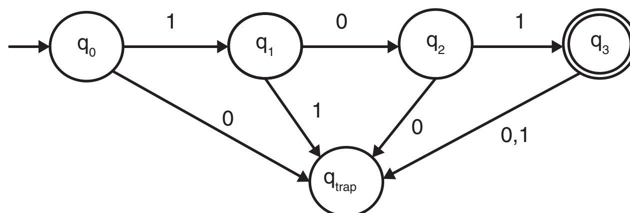
$M = (Q, \Sigma, \delta, q_0, F)$

Where Q = Non empty finite set of states
Σ = input alphabet
q_0 = initial start state
F = set of final states
δ = transition function that takes two arguments a state and input symbol and returns output as state i.e., $\delta: Q \times \Sigma \rightarrow Q$

Example: $\delta(q_1, a) = q_1$ DFA can be used as finite acceptor because its sole job is to accept certain input strings and reject other strings.

It is also called language recognizer because it merely recognizes whether the input strings are in the language or not.

Example 5:
Design a DFA that accepts only input 101 over the set {0, 1}.



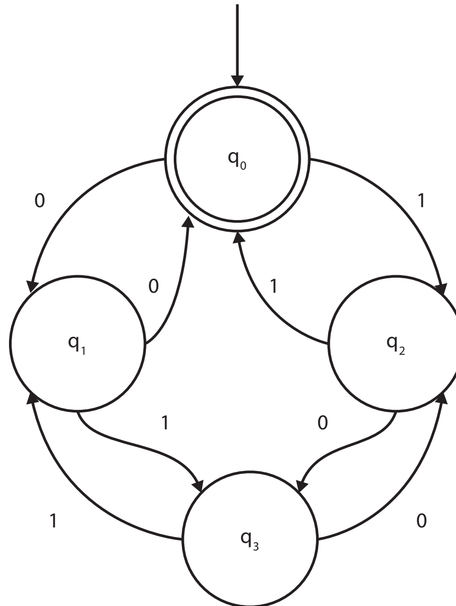
Here q_{trap} is called trap state/dummy state where unnecessary transitions are thrown away.

Example 6:
Design a DFA that accepts even number of 0's and even number of 1's.

Solution: This FA will have four different possibilities while reading 0's & 1's as input. The possibilities could be

Even number of 0's and even number of 1's–q_0
Odd number of 0's and even number of 1's–q_1
Even number of 0's and odd number of 1's–q_2
Odd number of 0's and odd number of 1's–q_3

where states are q_0, q_1, q_2 and q_3. Since the state q_0 indicates the condition of even number of 0's and even number of 1's this state is made as final state. The DFA is given by



2.10.8 Nondeterministic Finite Automaton (NFA)

For a given input symbol, there can be more than one transition from a state. Such automaton is called Nondeterministic Finite Automaton. NFA is mathematically described as quintuple. Nondeterministic finite automaton can be defined as quintuple

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

Where Q = Non empty finite set of states

Σ = input alphabet

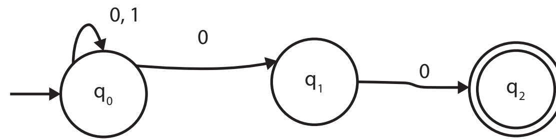
q_0 = initial start state

F = set of final states

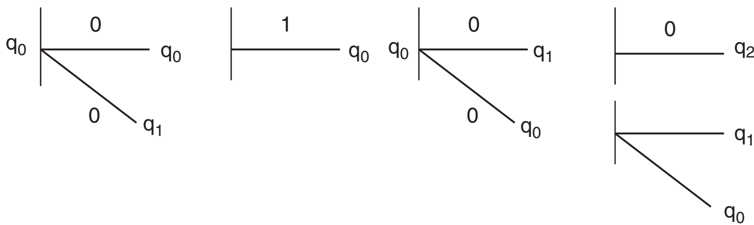
δ = transition function that takes two arguments a state and input symbol and returns output as state i.e., $\delta: Q \times \Sigma \rightarrow 2^Q$

Acceptance of NFA

Acceptance of a string is defined as reaching to the final states on processing the input string.



Check 0100 is accepted or not

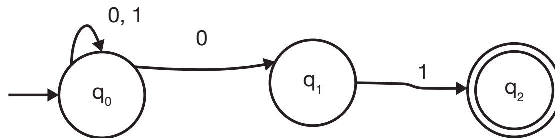


Since q_2 is in the final state, there is at least one path from the initial state to one of the final state. Hence the given string is accepted.

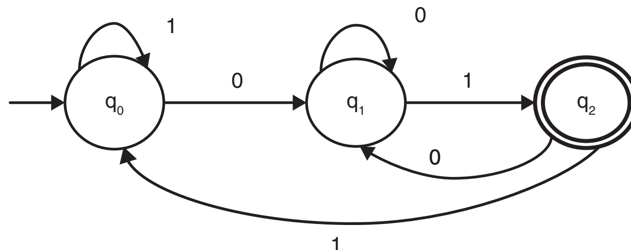
Note: It is easy to construct NFA than DFA, but the processing time of the string is more than DFA.

- Every language that can be described by NFA can be described by some DFA.
- DFA in practice has more states than NFA.
- Equivalent DFA can have at most 2^n states, whereas NFA has only 'n' states.

Example 7:
Design NFA accepting all string ending with 01 over $\Sigma = \{0, 1\}$.
We can have any string on 0 or 1 but should end with 0 followed by 1.



Corresponding DFA is



So drawing NFA is simple than DFA.

2.10.9 Equivalence of DFAs and NFAs

Let L be a set accepted by a NFA. Then there exists a DFA that accepts L.

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA accepting L.

Define DFA $M^1 = (Q^1, \Sigma^1, \delta^1, q_0^1, F^1)$ as follows.

The states of M^1 are all the subsets of the set of states of M. i.e., $Q^1 = 2^Q$

Example: If Q = {A, B} then Q^1 = {[ε], [A], [B], [AB]}

F^1 is the set of all states in Q^1 containing a final state of M.

Example: If F= {B} then F^1 = {[B], [AB]}

An element of Q^1 will be denoted by $[q_1, q_2, \ldots q_i]$ where $q_1, q_2, \ldots q_i$ are in Q.

Note: $[q_1, q_2, \ldots q_i]$ is a single state of DFA corresponding to set of states of NFA. $q_0^1 = [q_0]$

we define $\delta^1([q_1, q_2, \ldots q_i]a) = [P_1, P_2, \ldots P_i]$

iff $\delta(\{q_1, q_2, \ldots q_i\}a) = \{P_1, P_2, \ldots P_i\}$

that is, δ^1 applied to an element $[q_1, q_2, \ldots q_i]$ of Q^1 is computed by applying δ to each state of Q represented by $[q_1, q_2, \ldots q_i]$. It is easy to show by induction on length of input string x that

$\delta^1(q_0^1, x) = [q_1, q_2, \ldots q_i]$

iff $\delta(q_0, x) = \{q_1, q_2, \ldots q_i\}$

Basis: The result is trivial for $|x| = 0$ $\quad q_0^1 = [q_0]$

Induction: Suppose that the hypothesis is true for inputs of length m or less. Let "xa" be a string of length m+1 with a in Σ. Then

$\delta^1(q_0, xa) = \delta^1(\delta^1(q_0, x), a)$

By inductive hypothesis

$\delta^1(q_0, x) = [P_1, P_2, \ldots P_j]$

iff $\delta(q_0, x) = \{P_1, P_2, \ldots P_j\}$

But by def of δ^1

$\delta^1([P_1, P_2, \ldots P_j]a) = [r_1, r_2, \ldots r_k\}$

Thus

$\delta^1(q_0^1, xa) = [r_1, r_2, \ldots r_k]$

iff $\delta(q_0, xa) = \{r_1, r_2, \ldots r_k\}$

This establishes the inductive hypothesis.

To prove that $L(M) = L(M^1)$

The string x is accepted by NFA or DFA only if it is in one of the final state.

Let for a string x in NFA $\delta(q_0, x) = P$ where $P \in F$. Then $\delta^1(q_0, x) = [P]$ where $[P] \in F^1$. Hence, the string x is accepted iff it is accepted by the NFA.

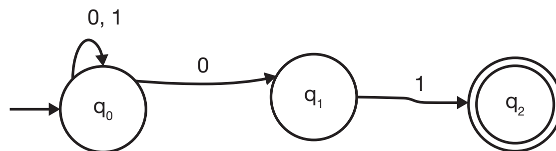
2.10.10 Converting NFA (MN) to DFA (MD)—Subset Construction

Let $M_N = (Q_N, \Sigma_N, \delta_N, q_{ON}, F_N)$ be the given NFA to construct the equivalent DFA M_D, $M_D = \{Q_D, Z_D, \delta_D, q_D, F_D\}$ where

i. $Q_D = 2^{Q_N}$. If NFA has n states, DFA at most can have 2^n states.
ii. $\Sigma_D = \Sigma_N$
iii. $[q_0] = \{q_o\}$
iv. F_D = Set of all states of Q_D that contains at least one final state of F_N.
v. $\delta_D((q_1, q_2, q_3), a) = \delta_n(q_1, a) \cup \delta_n(q_2, a) \cup \delta_n(q_3, a) = \{P_1, P_2, P_3\}$ say

Add state $[P_1, P_2, P_3]$ to Q_D if it is not there.

Example 8:
Convert the following NFA to DFA



Solution:

$Q = 2^3 = 8$ states = all subsets of q_0, q_1, q_2
$= \{\varnothing, [q_0], [q_1], [q_2], [q_0, q_1], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$

$\Sigma = 0, 1$

$q_0 = [q_0]$

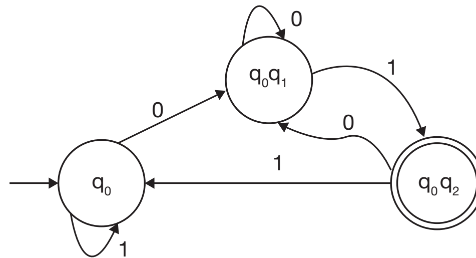
$F = \{[q_2], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$

δ = is given by $\delta_D([q_1\ q_2], a) = \delta_n(q_1, a)\ U\ \delta_n(q_2, a)$

when δ_n is transition function of NFA

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
→$[q_0]$	**$[q_0, q_1]$**	**$[q_0]$**
$[q_1]$	$\varnothing$	$[q_2]$
$[q_2]$	$\varnothing$	$\varnothing$
***$[q_0, q_1]$**	**$[q_0, q_1]$**	**$[q_0, q_2]$**
***$[q_0, q_2]$**	**$[q_0, q_1]$**	**$[q_0]$**
$[q_1, q_2]$	$\varnothing$	$[q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1]$	$[q_0, q_2]$

The states $[\varnothing]$, $[q_1]$, $[q_2]$, $[q_1, q_2]$ and $[q_0, q_1, q_2]$ are not reachable from start stated and hence cannot define any strings. So they can be ignored. Hence, the DFA can be simplified as follows:



To get this **Simplified DFA** construct the states of DFA as follows:

i. Start with the initial state. Do not add all subsets of states as there may be unnecessary states.

ii. After finding the transition on this initial state, include only the resultant states into the list until no new state is added to the list. For example, if $\delta(q_0, a) = \{q_0, q_1\}$, then add this as new state in DFA. Then find transition from this state on input symbol.

iii. Declare the states as final if it has at least one final state of NFA.

Example 9:

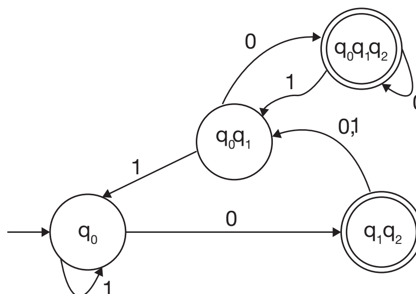
Convert the following NFA to DFA.

δ	0	1
$\rightarrow q_0$	$\{q_1\ q_2\}$	$\{q_0\}$
q_1	$\{q_0\ q_1\}$	Ø
$*q_2$	q_1	$\{q_0\ q_1\}$

DFA is

δ	0	1
$\rightarrow[q_0]$	$[q_1\ q_2]$	$[q_0]$
$*[q_1\ q_2]$	$[q_0\ q_1]$	$[q_0\ q_1]$
$[q_0\ q_1]$	$[q_0\ q_1\ q_2]$	$[q_0]$
$*[q_0\ q_1\ q_2]$	$[q_0\ q_1\ q_2]$	$[q_0\ q_1]$

The transition diagram of DFA is as shown below.



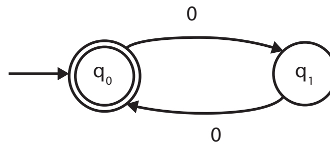
2.10.11 NFA with Epsilon (ε)-Transitions

We can extend an NFA by introducing a "ε-moves" that allows us to make a transition on the empty string. There would be an edge labeled ε between two states, which allows the transition from one state to another even without receiving an input symbol. This is another mechanism that allows NFA to be in multiple states at once. Constructing such NFA is easier, but is not that powerful. The NFA with ε-moves is given by $M = (Q, \Sigma, \delta, q_0, F)$ where δ is defined as $Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^Q$.

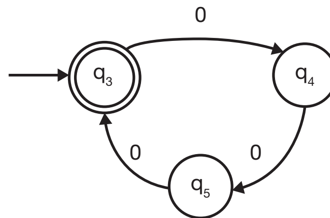
Example 10:

Design NFA for language $L = \{0^K \mid K \text{ is multiple of 2 or 3 }\}$

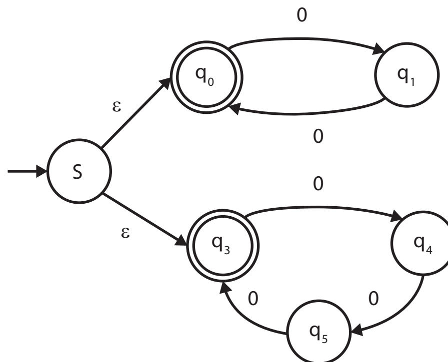
NFA for that set of strings that has a number of 0's, which are multiples of 2.



NFA for the set of strings that has a number of 0's, which are multiples of 3.



Combining these two NFAs,



2.10.12 Epsilon Closure (ε-closure)

Epsilon closure or ε-closure of a state is simply the set of all states reachable on ε. This can be expressed as either $\varepsilon(q)$ or ε-closure (q). In the above example,

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

Let us define the extended transition function for an NFA with ε-transitions. For a regular NFA we used the induction step as follows.

Let
$\hat{\delta}(q, w) = \{p_1, p_2, \ldots p_k\}$
$\hat{\delta}(p_i, a) = S_i$ for i=1, 2,...k

Then $\hat{\delta}(q, wa) = S_1 \cup S_2 \cup \ldots \cup S_k$
For an NFA with ε, we change for $\hat{\delta}(q, wa)$ as
$\hat{\delta}(q, wa) = \cup$ ε-closure$(S_1 \cup S_2 \cup \ldots \cup S_k)$
This includes the original set $S_1, S_2 \ldots S_k$ as well as any states we can reach via ε-transitions.

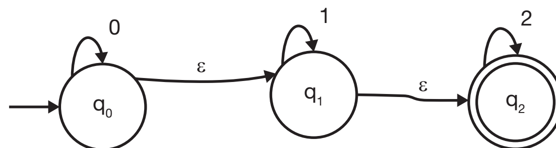
2.10.13 Eliminating ε-Transitions

ε-Transitions are used for convenience in some cases, but do not increase the power of the NFA. To eliminate them we can convert an NFA—with ε into an equivalent NFA—without ε by eliminating the ε edges and replacing them with the edged labeled with the symbol present in Σ. We can also convert NFA—with ε into an equivalent DFA, which is quite similar to the steps we took for converting a normal NFA to a DFA, except we must now follow all ε-transitions and add those to our set of states.

2.10.14 Converting NFA with ε-Transition to NFA Without ε-Transition

For each state, compute ε-closure(q) on each input symbol $a \in \Sigma$. If the ε-closure of a state contains a final state then make the state as final.

Let the following be NFA with ε-transitions.



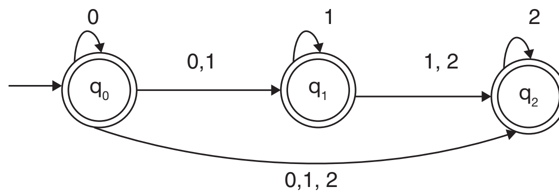
The transition table is

	a = 0	a = 1	a = 2	a = ε
q_0	q_0	Ø	Ø	q_1
q_1	Ø	q_1	Ø	q_2
$*q_2$	Ø	Ø	q_2	Ø

NFA without ε-transitions is

	a = 0	a = 1	a = 2
→*q_0	{q_0, q_1, q_2 }	{q_1, q_2 }	{q_2 }
*q_1	Ø	{q_1, q_2 }	{q_2 }
*q_2	Ø	Ø	{q_2 }

Transition diagram of NFA without ε-transitions



2.10.15 Converting NFA with ε-Transition to DFA

1. Compute ε* for the current state, resulting in a set of states S.
2. δ(S, a) is computed for all a ∈ Σ by
   a. Let S = {p1, p2, ... pk}
   b. Compute R = δ(S, a) as

   $$R = \bigcup_{p \in s} \delta(p_i, a) = \{r_1, r_2, r_3 ... r_m\}$$

   This set is achieved by following input a, not by following any ε-transitions.
   c. Add the ε-transitions by computing ε-closure(R).
3. Make a state an accepting state if it includes any final states in the NFA.

Note: The epsilon transition refers to moving from one state to another without reading an input symbol. These transitions can be inserted between any states.

Consider the NFA-epsilon move machine M = {Q, Σ, δ, q_0, F}
Q = { q_0, q_1, q_2}
Σ = {a, b, c} and ε moves

	a = 0	a = 1	a = 2	a = ε
q_0	q_0	Ø	Ø	q_1
q_1	Ø	q_1	Ø	q_2
*q_2	Ø	Ø	q_2	Ø

DFA construction

Step 1: Compute ε – closure of each state.

$\hat{\varepsilon}(q_0) = \{q_0, q_1, q_2\}$

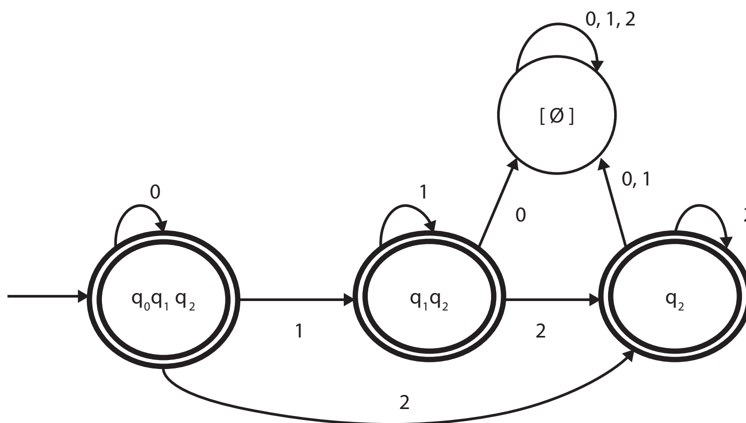
$\hat{\varepsilon}(q_1) = \{q_1, q_2\}$

$\hat{\varepsilon}(q_2) = \{q_2\}$

Step 2: Explore the states that are valid states in DFA using the above step 2 and step 3.

DFA transition table

	a = 0	a = 1	a = 2
$\rightarrow\varepsilon^*(q_0)$ =* $[q_0q_1q_2]$	$[q_0q_1q_2]$	$[q_1q_2]$	$[q_2]$
$^*[q_1q_2]$	[Ø]	$[q_1q_2]$	$[q_1q_2]$
$^*[q_2]$	[Ø]	[Ø]	$[q_2]$
[Ø]	[Ø]	[Ø]	[Ø]

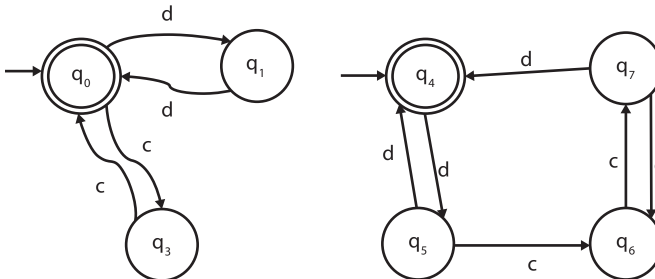


2.10.16 Comparison Method for Testing Equivalence of Two FAs

Let M and M^1 be two FAs over Σ. We construct a comparison table consisting of n + 1 columns where n is the number of i/p symbols.

1. The first column consisting of pair of nodes of form (q, q^1) where $q \in M$ and $q^1 \in M^1$.
2. If (q, q^1) appears in the same row of the first columns, then the corresponding entry in a column $(a \in \Sigma)$ is (qa, qa^1), where (qa, qa^1) are reachable from q and q^1 on a.

3. The table is constructed by starting with a pair of initial vertices q_{in}, q_n^1 of M and M^1. We complete construction by considering the pairs in second and subsequent columns, which are not in the first column.
   i. If we reach a pair (q, q^1) such that q is final states of M^1 and q^1 is nonfinal state of M^1 = > terminate construction and conclude that M and M^1 are not equivalent.
   ii. If construction is terminated, then no new element appears in second and subsequent columns which are not on the first column. Hence, M and M^1 are equivalent.



By looking at the number of states we cannot conclude

	c	d
$\rightarrow[q_0, q_4]$	$[q_3, \emptyset]$	$[q_1, q_5]$

Since we do not have a pair $(q_{M1}\ q_{M2})$ for input c, M_1 and M_2 are not equivalent.

2.10.17 Reduction of the Number of States in FA

- Any DFA defines a unique language but the converse is not true, that is, for any language there is unique DFA is not always true.
- For the same language there can exist many DFAs. So there can be considerable difference in the number of states. By using the comparison method we can test this.

Indistinguishable states:
Two states p and q of a DFA are said to be indistinguishable if

$$\delta^*(p, w) \in F \text{ implies } \delta^*(q, w) \in F$$

$$\text{and} \quad \delta^*(p, w) \notin F \text{ implies } \delta^*(q, w) \notin F$$

Or for some strong $w \in \Sigma^*$ if $\delta^*(p, w) \in F$ and $\delta^*(q, w) \in F$ or vice versa, then states p and q are said to be distinguishable by a string w.

Equivalent classes: The concept of equivalent class is used in minimizing the number of states. States that are equivalent can be combined as one class called equivalent class. Let us see the definition of equivalent states.

Definition 1
Two states q_1 and q_2 are equivalent ($q_1 \equiv q_2$) if both $\delta(q_1, a)$ and (q_2, a) are final states or both of them are non final states for all $a \in \Sigma$. These states are said to be 0-equivalent.

Definition 2

Two states q_1 and q_2 are K-equivalent ($K \geq O$) if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both non final states for all strong × of length K or less.

Therefore, any two final states are K-equivalent if they belong to same set in K-1 step otherwise they are not K-equivalent.

Properties:

P_1: If the relation (q_1 & q_2) is reflexive, symmetric and transitive, then it is said to be equivalent relation (i.e., K – Equivalence relation).

P_2: Every equivalence relation partition set. is K- Equivalence relation partition set Q.

P_3: If (q_1 and q_2) are K-equivalent for all, $K \geq O$, then they are equivalent.

P_4: If q_1 and q_2 are (K + 1) – Equivalent, then they are K-Equivalent.

2.10.18 Minimization of DFA

For any given Deterministic Automation with more number of states, we can construct its equivalent DFA with minimum number of states.

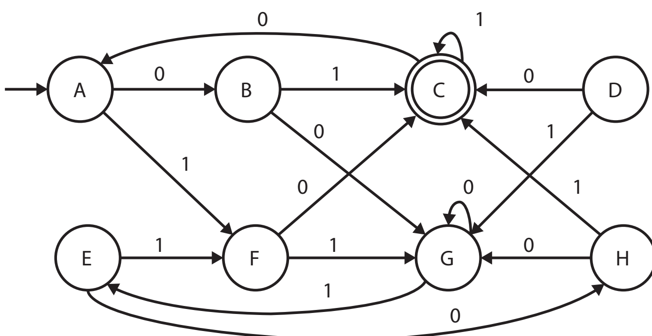
Construction of minimum automation

1. Initially construct 0 – equivalence class as $\prod_o = \{ Q_1^O, Q_2^O\}$ Where Q_1^O is set of final states and $Q_2^O = Q - Q_1^O$ is set of non final states.
2. Construct $\prod_{K+1}$ from $\prod_K$ further partitioning as follows:
   a. Let Q_1^K be any subset in $\prod_K$. if q_1 and q_2 are in Q_1^K they are (K + 1) equivalent provided $\delta(q_1, a)$ and $\delta(q_2, a)$ are K-equivalent.
   b. Find out whether $\delta(q_1, a)$ and $\delta(q_2, a)$ are in the same equivalence class in $\prod_K$ for every $a \in \Sigma$. If so, q_1 and q_2 are (k + 1) equivalence. This way Q_i^k is further divided into (K + 1) equivalence classes. Repeat this for every Q_i^K in $\prod_K$ to get all the elements of $\prod_{K+1}$.
3. Construct $\prod_n$ for n = 1, 2, 3,……………until $\prod_n = \prod_{n+1}$.
4. For required minimum state automation, states are equivalent classes obtained finally.

First approach

Example 11:

Find the minimum finite state automaton for the following DFA.



	0	1
→ a	b	f
b	g	c
c	a	c
d	c	g
e	h	f
f	c	g
g	g	e
h	g	c

Any two final states are 0 – equivalent and any two non final states are also 0 – equivalent.

$\prod_0$ (1, 2) = {{c},{a, b, d, e, f, g, h}}

	a	b	d	e	f	g	h
0	2	2	1	2	1	2	2
1	2	1	2	2	2	2	1

From the above table, we find a, e, and g are 1–equivalent and b and h are 1–equivalent and d, f are 1 – equivalent. Hence $\prod_1$ is as follows:

$\prod_1$ (1, 3, 4, 5) = {{a, e, g}{b, h} {d, f} {c}}

Using the new classes we find whether they are 2–equivalent.

	a	b	d	e	f	g	h
0	4	3	1	4	1	3	3
1	5	1	3	5	3	3	1

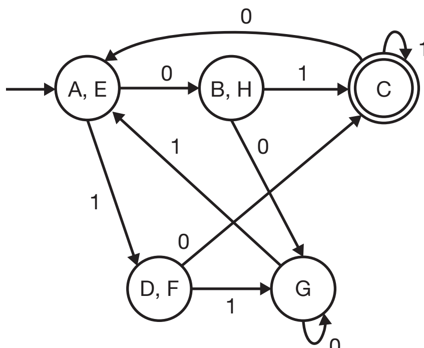
$\prod_2$ (1, 6, 7, 8, 9) = {{a, e},{b, h}, {d, f}, {g},{c}}

	a	b	d	e	f	g	h
0	7	9	1	7	1	9	9
1	8	1	9	8	9	6	1

$\prod_3$ (1, 6, 7, 8, 9) = {{a, e}, {b, h}, {d, f}, {g}, {c}}

	a	b	d	e	f	g	h
0	7	9	1	7	1	9	9
1	8	1	9	8	9	6	1

From the above two relations, ∏2 and ∏3 are the same. Hence, the final set of states are the sets 1, 6, 7, 8, 9 where {a, e}, {b, h}, {d, f}, {g},{c} are all 3 – equivalent. The minimized DFA is as follows:



	0	1
→ {A, E}	{B, H}	{D, F}
{B, H}	{G}	{C}
*{C}	{A, E}	{C}
{D, F}	{C}	{G}
{G}	{G}	{A, E}

Second approach

2.10.19 Minimization of DFA Using the Myhill Nerode Theorem

The Myhill Nerode theorem that is used to prove the given language is not regular and also to eliminate useless states in the given DFA. The theorem is stated as

- The language L accepted by a DFA is regular if and only if the number of equivalence classes of R_L is finite.
- The number of states in the smallest automaton accepting L is equal to the number of equivalence classes in R_L. Therefore, R_L is of finite index.

Let ≡ equivalence on the states in M such that

P ≡ q if and only if for each input string x

$\delta(p, x) = \delta(q, x) = q_a$ where q_a is accepting state.

⇒ p is equivalent to q.

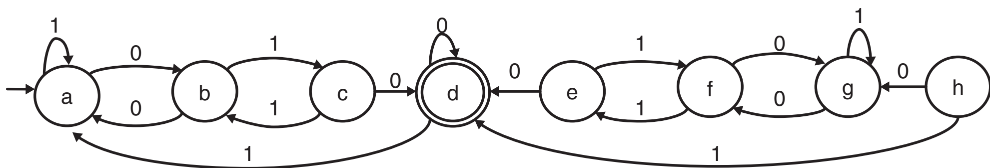
If $\delta(p, x) = q_a$ and $\delta(q, x) = q_n$ for some $q_a \in F$ and $q_n \notin F$.

⇒ p is distinguishable from q.

Algorithm for finding distinguishable states:

- For each pair [p, q] where p ∈ F and q ∈ {Q – F}, mark (p, q) = X
- For each pair of distinct states [p, q] in F × F or (Q – F) X (Q – F) do
  - If for some input symbol a δ([p, q], a) = [r, s], if [r, s] = X then
    - Mark [p, q]
    - Recursively mark all unmarked pairs which lead to [p, q] on input for all a ∈ Σ.
- Else
    - for all input symbols 'a' do
    - put [p, q] on the list for δ([p, q], a) unless δ([p, q], a)=[r, r].
- For each pair [p, q], which is unmarked are the states which are equivalent.

Example 12:
Find minimum-state automaton equivalent to the transition diagram given.



Solution: The distinguishable states are marked with symbol x. The relation of all states are represented as a matrix of size n × n. since if p is distinguishable to q, it implies that q is distinguishable to p. Therefore, it is sufficient to have a lower matrix to represent the relation of one state with all the other states.

Step 1: First mark for all states (p, q) where p is the final state and q is the non-final state.

(d, a) = x (d, b) = x (d, c) = x (d, e) = x (d, f) = x

(d, g) = x (d, h) = x

Step 2: Find the states that are distinguishable with a.

$\delta([a,b],0) = [b,a]$ $\delta([a,b],1) = [a,c]$

$\delta([a,c],0) = [b,d]$ $\delta([a,c],1) = [a,b]$ $\Rightarrow$ mark[a,c] = x as [b,d] = x

$\Rightarrow$ mark[a,b] = x as [a,c] = x

$\delta([a,e],0) = [b,d]$ $\delta([a,e],1) = [a,f]$ $\Rightarrow$ mark[a,e] = x as [b,d] = x

$\delta([a,f],0) = [b,g]$ $\delta([a,f],1) = [a,e]$ $\Rightarrow$ mark[a,f] = x as [a,e] = x

$\delta([a,g],0) = [b,f]$ $\delta([a,g],1) = [a,g]$

$\delta([a,h],0) = [b,g]$ $\delta([a,h],1) = [a,d]$ $\Rightarrow$ mark[a,h] = x as [a,d] = x

Find the states that are distinguishable with b

$\delta([b,c],0) = [a,d]$ $\delta([b,c],1) = [c,b]$ $\Rightarrow$ mark[b,c] = x as [a,d] = x

$\delta([b,e],0) = [a,d]$ $\delta([b,e],1) = [c,f]$ $\Rightarrow$ mark[b,e] = x as [a,d] = x

$\delta([b,f],0) = [a,g]$ $\delta([b,f],1) = [c,e]$

$\delta([b,g],0) = [a,f]$ $\delta([b,g],1) = [c,g]$ $\Rightarrow$ mark[b,g] = x as [a,f] = x

$\delta([b,h],0) = [a,g]$ $\delta([b,h],1) = [c,d]$ $\Rightarrow$ mark[b,h] = x as [c,d] = x

Find the states that are distinguishable with c

$\delta([c,e],0) = [d,d]$ $\delta([c,e],1) = [b,f]$

$\delta([c,f],0) = [d,g]$ $\delta([c,f],1) = [b,e]$ $\Rightarrow$ mark[c,f] = x as [d,g] = x

$\delta([c,g],0) = [d,f]$ $\delta([c,g],1) = [b,g]$ $\Rightarrow$ mark[c,g] = x as [d,f] = x

$\delta([c,h],0) = [d,g]$ $\delta([c,h],1) = [b,d]$ $\Rightarrow$ mark[c,h] = x as [d,g] = x

Find the states that are distinguishable with e

$\delta([e,f],0) = [d,g]$ $\delta([e,f],1) = [f,e]$ $\Rightarrow$ mark[e,f] = x as [d,g] = x

$\delta([e,g],0) = [d,f]$ $\delta([e,g],1) = [f,g]$ $\Rightarrow$ mark[e,g] = x as [d,f] = x

$\delta([e,h],0) = [d,g]$ $\delta([e,h],1) = [f,d]$ $\Rightarrow$ mark[e,h] = x as [d,g] = x

Find the states that are distinguishable with f

$\delta([f,g],0) = [g,f]$ $\delta([f,g],1) = [e,g]$ $\Rightarrow$ mark[f,g] = x as [e,g] = x

$\delta([f,h],0) = [g,g]$ $\delta([f,h],1) = [e,d]$ $\Rightarrow$ mark[f,h] = x as [e,d] = x

Find the states that are distinguishable with g

$\delta([g,h],0) = [f,g]$ $\delta([g,h],1) = [g,d]$ $\Rightarrow$ mark[g,h] = x as [g,d] = x

	a	b	c	d	e	f	g
b	x						
c	x	x					
d	x	x	x				
e	x	x		x			
f	x		x	x	x		
g		x	x	x	x	x	
h	x	x	x	x	x	x	x

From the lower triangle of the matrix it is clear that state [a,g], [b,f], and [c,e] belong to the same class. These states can be merged and the minimized DFA is

	a=0	a=1
→[a,g]	[b,f]	[a,g]
[b,f]	[c,e]	[a,g]
[c,e]	[d]	[b,f]
*[d]	[d]	[a,g]
[h]	[a,g]	[d]

2.11 Lex Tool: Lexical Analyzer Generator

2.11.1 Introduction

Lex is a language for specifying lexical analyzers.

Lex generates programs that can be used in simple lexical analysis of text. The input files contain regular expressions for recognizing tokens to be searched for and actions written in C to be executed when expressions are found.

Lex converts the regular expression into table-driven DFA. This deterministic finite automaton generated by Lex performs the recognition of the regular expressions. The order in which the program fragments written by the user are executed is the same as that in which the regular expressions are matched in the input stream.

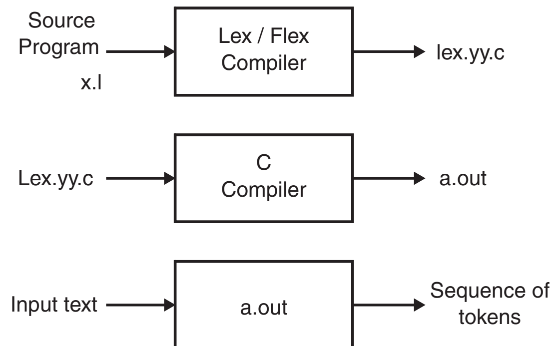
The general form of a Lex source file is:

```
Declarations
%%
Regular expressions {actions}
%%
Subroutines
```

Lex program contains three sections—declarations, regular expressions, and subroutines where the declaration and the user subroutines are often omitted. The second section, that is, the regular expressions part is compulsory. The declarations part contains 'c' declarations and lex declarations. 'c' declarations are embedded between % {and %}. Lex declarations contain token definitions. Token rules, that is, patterns are defined as regular expressions in the second part. When the given input matches with this pattern, action to be performed is described against that regular expression. Lex program is stored with an extension ".l" like x.l. Lex turns the user's expressions and actions into the host general-purpose language. The generated program is named yylex() in the lex.yy.c file. This lex.yy.c is the lexer in C. To run this c file, first compile lex.yy.c with cc then use the exe file a.out to test the output.

Expressions in a stream will be recognized by the yylex and it performs the specified actions for each expression whenever they are matched. See the following figure.

Assume that lex specification is prepared in file x.l. Run this input file x.l with lex, that gives lex.yy.c as output. Run lex.yy.c under the C compiler that gives a.out as output.



Regular expressions in Lex use the following operators:

a the character "a"
"a" an "a", even if a is an operator
\a an "a", even if a is an operator
[ab] the character a or b
[a-c] the characters a, b, or c
[^a] any character but a
. any character but newline
^a an a at the beginning of a line
<a>b a b when Lex is in start condition a
a$ an a at the end of a line
a? an optional a
a* 0, 1, 2, ... instances of a
a+ 1, 2, 3, ... instances of a
a|b an a or a b
(a) an a
a/b an a but only if followed by b
{aa} the translation of aa from the definitions section
a{m,n} m through n occurrences of a

Solved Problems

1. Write a lexical analyzer for input

```
void main()
{
   int a = 10;
}
```

Lex Code:

```
%{
#include<stdio.h>
%}
letter      [_a – z A – Z]
digit       [0-9]
id          {letter}({letter}|{digit})*
%%
{digit}*                printf("%d---- Number\n",yytext);
"int"|"void"|"main"|    printf("%s ---- keyword\n",yytext);
{id}                    printf("%s ---- identifier\n",yytext);
"="                     printf("%c ---- assignment operator\
                          n",yytext);
[()}({]                 printf("%c ---- Braces\n",yytext);
","|";"                 printf("%c ---Punctuation symbol
\n",yytext);
%%
int main(void)
{
 yylex();
}
 int yywrap()
{
 return 1;
}
```

Output:

```
void ----keyword
main---- identifier
(---- Braces
)---- Braces
{---- Braces
int---- keyword
a---- identifier
=---- assignment operator
10---- integer
}---- Braces
;---- Punctuation symbol
```

2. Here is a program in Lex to convert lowercase to uppercase. Ignore the blanks, and replace multiple blanks by a single blank.

```
     %%
      [a-z]  {putchar(yytext[0]+'A');}
```

```
    [ ]+  { }
    [ ]+  {putchar(' ');}
```

The following program copies an input file while adding 10 to every positive number divisible by 7.

```
%%
    int a;
[0-9]+ {
        a = atoi(yytext);
        if (a%7 == 0)
            printf("%d", a + 10);
        else
            printf("%d",a);
      }
```

In this program, the rule [0–9]+ recognizes strings of digits. The atoi( ) function converts the digits to binary and stores the result in "a." The modulus operator % is used to check whether "a" is divisible by 7. If it is divisible then "a" is incremented by 10, else written as it is to the output.

Summary

- A lexical analyzer reads the input text and divides it in to a stream of tokens.
- Token is a group of characters with logical meaning.
- Pattern is a rule that describes tokens. Usually it is a regular expression.
- Lexeme is actual text that matches the pattern and is recognized as token.
- A lexical analyzer removes comment lines and white space characters.
- Lex is an automated tool for generating scanners.
- There are three different sections of LEX program: definitions, translation rules, and subroutines.
- The output of the Lex compiler is in the lex.yy.c file.
- Finite automaton is used for recognizing tokens.
- Tokens are described by regular expressions.

Fill in the Blanks

1. The lexical analyzer reads source text and divides them into__________________.
2. Pattern is a rule that describes a __________________.
3. In "if a != b" Lexeme is for token id is__________________
4. __________________ and __________________ are the routines compulsory in the subroutine section.
5. The lex code that deletes all blanks or tabs at the ends of lines is __________________.
6. The lex code to count number of lines is __________________.

7. ________________ is the default 'c' file name Lex produces after compiling the Lex program to c program.
8. ________________ is an example of non token in C language.
9. Can we combine lexical analysis phase with parsing? Yes or No.________________
10. ________________ is the advantage of having lexical analysis as a separate phase.
11. ________________ is the regular expression that generates all and only the strings over alphabet {0, 1} that ends in 1.

Objective Question Bank

1. The example of lexical error is

   (a) Xyz (b) _xyz (c) x2yz (d) $xyz

2. Which of the following is an example of non tokens in C program?

   (a) if (b) #ifdef (c) rate (d) none

3. Which of the following is a regular expression for patterns, integer numbers, and real numbers with scientific notation? Let D is [0–9]*

   (a) D+(.D+)?(E(+|–)?D+) (b) (.D+)?(E(+|–)?D+)
   (c) D+(.D+)?(E(+|–)?D*) (d) D+(.D+)|(E(D+)

4. What are the tasks performed by a lexical analyzer?

   (a) Removes comment lines and white space characters

   (b) Correlates error messages. (c) dividing text into tokens (d) all

5. Find the number of tokens in the following code

   if (x > y) z=0;

   (a) 8 (b) 9 (c) 10 (d) 7

*6. Find the number of tokens in following **code**
   printf("&i = % x i = %d", & i, i);

   (a) 7 (b) 8 (c) 9 (d)10

7. Following is the function of a lexical analyzer:

   (a) Giving errors (b) removing blanks

   (c) correlating err messages (d) all

8. Following is a regular expression for the set of all strings over the alphabet {a} that has an even number of a's.

   (a) aa* (b) (aa)* (c) aa*a (d) a(aa)*

9. Find the number of tokens in the following C code

```
int max(int x, int y) {
     /* find max */
     return (x > y:? x : y); }
```

(a) 25 (b) 26 . (c) 22 (d) 21

10. Which of the following best describes the output of LEX?

(a) DFA (b) Tokens (c) parser (d) lexemes

*11. Which of the following strings can definitely be said to be tokens without looking at the next input characters while compiling a Pascal program?

(I) begin (II) Programs (III) < >

(a) I (b) II (c) III (d) all

*12. In a compiler, the module that checks every char of source text is called

(a) Code generation (b) Code optimization
(c) Lexical analysis (d) Syntax analysis.

*13. Match pairs

(a) Lexical analysis (p) DAGS
(b) Code optimization (q) Syntax Tree
(c) Code generation (r) PDA
(d) Abelian Group. (s) FA

(a) a – s, b – q, c – p, d – r (b) a – s, b – p, c – q, d – r
(c) a – s, b – p, c – r, d – q (d) none

14. Tables created during lexical analysis are

(a) terminal table (b) identifier table
(c) literal table (d) non uniform symbol table

15. Find the number of tokens in the following code:

if (x >= y) z = 0;

(a) 8 (b) 9 (c) 10 (d) 11

Exercises

1. Write regular definitions for specifying a floating point number in a programming language like C.
2. Write regular definitions for specifying an integer array declaration in a language like C.

3. Convert the following regular expressions into NFA with ε.
   (a) 0 + (1 + 0)*00
   (b) zero→0; one→1; bit→zero + one; bits→bit*
   (c) (011 + 101 + 110 + 111)*
   (d) (000 + 111)* + (101 + 010)+
4. Write a lex program to scan an input C source file and replace all "float" with "double."
5. Write a lex program that scans an input C source file and produces a table of all macros (#defines) and their values.
6. Write a lex program that reads an input file containing numbers (either integers or floating-point numbers) separated by one or more white spaces and adds all the numbers.
7. Write a lex program that scans an input C source file and recognizes identifiers and keywords. The scanner should output a list of pairs of the form (token; lexeme), where token is either "identifier" or "keyword" and lexeme is the actual string matched.
8. Run the lexical analyzer for the following C program and comment on tokens/output.

```
main()
{
 int a[3], t1t2;
 t1=2;
 a[0]=1; a[1]=2; a[t1]=3;
 t2 = -(a[2]+t1*6)/(a[2]-t1);
  if t2>5
         print(t2);
   else
   {
      int t3; t3=99;  t2=-25;
      print(-t1 +t2*t3); /*this is a comment on 2 lines */
   } endif
}
```

9. Write a program for the lexical analyzer to divide the following C program to tokens.

```
main()
{
 int x, y, z;
 x=10;
 y=20;
 z=30;
}
```

10. Run the lexical analyzer for the following C program and comment on tokens/output.

```
main()
{
int i, j;
```

```
while(i<100)
        {
        i=i+10;
        printf("%d\n",&i);
        }
}
```

Key for Fill in the Blanks

1. stream of tokens
2. Tokens
3. a or b
4. main(), yywrap()
5. [\t]+$;
6. [\n] {lno++;}
7. lex.yy.c
8. #include, #define, /*
9. yes
10. Design simplified.
11. (0+1)*1+

Key for Objective Question Bank

1. d 2. b 3. a 4. d 5. c 6. d 7. d
8. b 9. c 10. a 11. c 12. c 13. b 14. d 15. c

CODING:
Design a lexical analyzer generator that can handle the following sample input. The lexical analyzer should ignore redundant spaces, tabs, newlines, and comments.

```
Input:
                    main()
                    {
                      int x, y;
                      if (x > y)
                            printf("x is greater");
                      else
                            y=10; /* this is just a comment */
                    }
```

Logic of the program:

- Read the input text from the file input_text.dat.
- Read the text, line by line and divide it into different words.
- Match words with tokens like "opr" for operator, "num" for numeric constant, "id" for identifier, etc.

- To distinguish between keyword and id, compare the word with the keyword list first, if it does not match then treat it as an id.
- Print token along with lexeme and line number.

Lexical analyzer developed in C language

```
/* Lexical Analyzer for any C program as input */
#include<stdio.h>
#include<string.h>
#include<ctype.h>

int main()
{
 int l, i, j, k, m, sno=1;
 char *keys[5]={"main","int","printf","float","if","else"};
 char line[100]={0}, str[20];
 FILE *fptr;
 clrscr();
 fptr=fopen("input_txt.dat","r");          /* input file input_txt*/
 printf("S.no Lexeme Token Line no.\n");   /* output format */
 printf("__________________________________");
 for(l=1;!feof(fptr);l++)
 {
  fgets(line, 100, fptr);
  for(i=0; line [i]!='\0';)
  {
   if(line [i]==' ')                       /* skip blank spaces */
    i++;
   else if(line [i]=='\n')
    line [i]='\0';
   else if(isalpha(line [i]))
   {
    j=0;
    while(isalnum(line [i]))
    {
     str[j]= line [i];
     j++;
     i++;
    }                          /* check for arrays */
    if(line [i]=='[')
    {
      str[j]='\0';
      printf("%d. %s array %d\n", sno, str, l);sno++;
      i++; j=0;
      while(line [i]!=']')
```

```
      {
       str[j]= line [i];
       j++;
       i++;
      }
      str[j]='\0';
      printf((“%d. %s index %d\n”, sno,str, l);sno++;
      i++;
    }
    else                       /* check for  keyword or id */
    {
    str[j]='\0';
    for(k=0;k<5;k++)
    {
     if(!strcmp(keys[k],str))
      break;
    }
    if(k==5)
     printf((“%d. %s identifier %d\n”,sno,str,l);sno++;
    else
     printf((“%d. %s keyword %d\n”,sno,str,l);sno++;
    }
   }                           /* check for Number constants */
   else if(isdigit(line [i]))
   {
    j=0;
    while(isdigit(line [i]))
    {
     str[j]= line [i];
     j++;
     i++;
    }
    str[j]='\0';
    printf((“%d. %s const %d\n”,sno,str,l);sno++;
   }                                  /* check for comment lines */
   else if((line [i]=='/')&&(line [i+1]=='*'))
   {
    printf((“%d. /* start of comment %d “,sno,l);sno++;
    i=i+2;
    while(line [i]!='*' && line [i+1]!='/'&& line [i]!='\n')
    {
     i=i+1;
     if(line [i]=='\n')
     {
```

```
     fgets(line,100,fptr);
     l++;
    }
   }
    i=i+2;
    printf(" %d\n",l);
  }                                /* check for operators */
  else
  {
   j=0;
   str[j]= line [i];
   j++;
   switch(line [i])
   {
    case '+':
    case '-':
    case '*':
    case '/':
    case '=':
    case '>':
    case '<':
    case '<=':
    case '>=':
    case '%':{
              if(line[i]=='%'&&(line[i+1]=='d'||line[i+1]=='f'||
              line[i+1]=='c'))
            {
              str[j]=line[++i];
              i++;j++;
              str[j]='\0';
              printf(("%d. %s special symbol %d \n ",sno,str,l);sno++;
              j=0;
               break;
            }
            else
            {
              str[j]='\0';
              printf(("%d. %s operator %d\n",sno,str,l);sno++;
              j=0;
              i++; break;
            }
           }
    case '(':
    case ')':
```

```
    case '{':
    case '}':{
              str[j]='\0';
              printf(("%d. %s parenthesis %d\n",sno,str,l);sno++;
              j=0;
              i++;
              break;
             }
    case ';':{
              str[j]='\0';
              printf(("%d. %s psymbol %d\n",sno,str,l);sno++;
              j=0;
              i++;
              break;
             }
    default: {
              str[j]='\0';
              j=0;
              i++;
              printf(("%d. %s special symbol %d\n",sno,str,l);sno++;
              break;
             }
    }
   }
  }
 }
 getch();
return 1;
}
```

Output:

S.no	Lexeme	Token	Line no
1.	main	keyword	1
2.	(	parenthesis	1
3.	)	parenthesis	1
4.	{	parenthesis	2
5.	int	keyword	3
6.	x	identifier	3
7.	,	special symbol	3
8.	y	identifier	3

9.	;	psymbol	3
10.	if	keyword	4
11.	(	parenthesis	4
12.	x	identifier	4
13.	>	operator	4
14.	y	identifier	4
15.	)	parenthesis	4
16.	printf	keyword	5
17.	(	parenthesis	5
18.	“	psymbol	5
19.	x is greater	identifier	5
20.	”	psymbol	5
21.	)	parenthesis	5
22.	;	psymbol	5
23.	else	keyword	6
24.	y	identifier	7
25.	=	operator	7
26.	10	const	7
27.	;	p symbol	7
28.	/*this is just a comment*/	comment	7
29.	}	parenthesis	8

Lexical analyzer using the LEX tool

Program: Write the lexical analyzer using the ‘lex’ tool that can handle the following sample input.

```
Input:
       main()
       {
         int x,y;
         if (x > y)
               printf(“x is greater”);
         else
               y=10; /* this is just a comment */
       }
```

Logic

- Define regular expressions for all tokens-constants, keywords, ids, operators, etc.
- When input matches with that pattern, define the action to be performed against each regular expression.
- Include main () that calls yylex () function that is created by the lex tool.
- Include the yywrap () function that deals with EOF.

Lex Program

```
%{
#include<stdio.h>
#include<ctype.h>
int lno=1;
%}
letter       [a-zA-Z]
digit        [0-9]
id            {letter}({letter}|{digit})*
num           {digit}+
keywd        "main"|"while"|"float"|"printf"|"int"|"if"|"else"|"then"
arry         ({id}"["({num}|{id})"]")
commt        ("/*"({id}|"\n")*"*/")
sp           [.,;""""]
%%   /* Pattern matching rules */
[\n]         {lno++;}
{ws}         { }
{sp}         {printf("special symbol=%c lineno=%d\n",yytext, lno);}
">"|"<"|"<="|">="  {printf("relational opr=%s lineno=%d\n",yytext,
                    lno);}
{commt}            {printf("comment=%s lineno=%d\n",yytext, lno);}
"="                {printf("assignment opr=%s lineno=%d\n",yytext,
                    lno);}
"+"|"-"|"*"|"/"         {printf("Arithmetic opr=%s lineno=%d\
                         n",yytext, lno);}
"["|"]"|"{"|"}"|"("|")"  {printf("parenthesis=%s lineno=%d\
                          n",yytext, lno);}
{keywd}            {printf("keyword=%s lineno=%d\n",yytext, lno);}
{arry}             {printf("array=%s lineno=%d\n",yytext, lno);}
{id}               {printf("identifier=%s lineno=%d\n",yytext, lno);}
{num}              {printf("number constant=%s lineno=%d\n",yytext,
                    lno);}
%%                 /* subroutines Section*/
main(int argc,char **argv)
{
 if(argc > 1)
         yyin=fopen(argv[1],"r");
```

```
 else
        yyin= stdin;
 yylex();
}

yywrap()
{
 exit(0);
}
```

After executing the above program for the given input, the OUTPUT of lex is as follows:

Lexeme	**Token**	**Line no**
main	keyword	1
(	parenthesis	1
)	parenthesis	1
{	parenthesis	2
int	keyword	3
x	identifier	3
,	special symbol	3
y	identifier	3
;	special symbol	3
if	keyword	4
(	parenthesis	4
x	identifier	4
>	relational opr	4
y	identifier	4
)	parenthesis	4
printf	keyword	5
(	parenthesis	5
“	special symbol	5
x is greater	comment	5
”	special symbol	5
)	parenthesis	5
;	special symbol	5
else	keyword	6
y	identifier	7
=	Assignment opr	7

```
10                                number constant      7
;                                 special symbol       7
/* this is just a comment */      comment              7
}                                 parenthesis          8
```

CHAPTER 3

Syntax Definition — Grammars

The syntax of high-level language is defined with context free grammar. Hence, these are used as a powerful tool by parsers in verifying the syntax of any high-level language.

CHAPTER OUTLINE

3.1 Introduction
3.2 Types of Grammars—Chomsky Hierarchy
3.3 Grammar Representations
3.4 Context Free Grammars
3.5 Derivation of CFGs
3.6 Language Defined by Grammars
3.7 Left Recursion
3.8 Left-Factoring
3.9 Ambiguous Grammar
3.10 Removing Ambiguity
3.11 Inherent Ambiguity
3.12 Non-context Free Language Constructs
3.13 Simplification of Grammars
3.14 Applications of CFG

The structure of a sentence in any language is defined with grammars. Hence, we will discuss what grammar is, how many types of grammar there are, and what are the different representations of grammar. We mainly focus on Context Free Grammars (CFG). What is CFG? How is a language defined with CFG? What are derivation tree, left-most derivation, and right-most derivation? These are all explained in detail in this chapter. We shall also discuss the problems with CFG like left recursion, left factoring, ambiguous grammars, and how to simplify grammars.

3.1 Introduction

Grammar is a set of rules and examples dealing with the syntax and word structures of a language, usually intended as an aid to the learning of that language.

Grammar is basically defined as a set of 4-tuple (V, T, P, S),
where V is set of nonterminals (variables),
T is set of terminals (primitive symbols),
P is set of productions (rules), which govern the relationship between nonterminals and terminals,
And S is start symbol with which strings in grammar are derived.

These *productions* or *rules* define the strings belonging to the language. The motivation for these grammars was from the description of natural language. Let us examine the rules used to define a sentence in the English language.

```
<sentence> → <noun> <verb>
<noun> → <com-noun> | <prop-noun>
<verb> → ate | sat | ran
<com-noun > → Rama | Bhama
<prop-noun > → She | He
```

Using these set of rules many sentences can be derived by substituting for variables.

1. Rama ate.
2. Bhama sat.
3. She ran.
4. He sat.

Language defined by a grammar G is denoted by L(G). Here L(G) represents a set of strings w derived from G. Start symbol S in one or more steps derives the strings or sentences of grammar that is represented by $S \Rightarrow^{+} w$. The *sentential form* of grammar is a combination of terminals and nonterminals. It is derived from S using one or more derivations. It is represented by $S \Rightarrow^{+} \alpha$. Two grammars G1 and G2 are equivalent if L(G1) = L(G2).

Some of the ***notations used*** to represent the grammar are:

1. Terminals symbols: these are represented by
   - Lower case letters of alphabet like a, c, z, etc.
   - Operator symbols like +, –, etc.
   - Punctuation symbols like (, } , , etc.
   - Digits like 0…9, etc.
   - Bold face strings like **int**, **main**, **if**, **else** etc.
2. Nonterminal symbols: these are represented by
   - Upper case letters of alphabet like A, C, Z, etc.
   - Letter S is the start symbol.
   - Lower case strings like expr, stmt, etc.
3. Grammar symbols can be terminals or nonterminals. They can be represented by Greek letters α, β.
4. Lower case letters u, v, w, x, y, z are generally used to represent a string of terminals.

Language acceptance:
The language starts with the start symbol; at every step, replace the nonterminal by the right hand side of the rule. Continue this until a string of terminals is derived. The string of terminals gives the language accepted by grammar.

Consider the language represented by a^+, represented by a set {a, aa, aaa, ….}. To generate strings of this language we define grammar as $S \rightarrow a$ and $S \rightarrow aS$. Now we get strings as follows:

Strings starting with S:

$S \Rightarrow a$	{a}
$S \Rightarrow aS \Rightarrow aa$	{aa}
$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaa$	{aaa}

3.2 Types of Grammars—Chomsky Hierarchy

Linguist Noam Chomsky defined a hierarchy of languages, in terms of complexity. This four-level hierarchy, called the ***Chomsky hierarchy***, corresponds to four classes of machines. Each higher level in the hierarchy incorporates the lower levels, that is, anything that can be computed by a machine at the lowest level can also be computed by a machine at the next highest level.

The Chomsky hierarchy classifies grammars according to the form of their productions into the following levels:

a. **Type 0 Grammars—Unrestricted Grammars (URG):** These grammars include all formal grammars. In URG, all the productions are of the form $\alpha \rightarrow \beta$, where α and β may have any number of terminals and nonterminals, that is, no restrictions on either side of productions. Every grammar is included in it if it has at least one nonterminal on the left hand side.

   Example:
   The language specified by $L(G) = \{a^{2i} \mid I >= 1\}$ is an unrestricted grammar. This grammar is also called phrase structured grammar. The grammar is represented by the following productions:

   $$S \rightarrow A\,C\,a\,B$$
   $$C\,a \rightarrow a\,a\,C$$
   $$C\,B \rightarrow D\,B$$
   $$C\,B \rightarrow E$$
   $$a\,D \rightarrow D\,a$$
   $$A\,D \rightarrow A\,C$$
   $$a\,E \rightarrow E\,a$$
   $$A\,E \rightarrow \varepsilon$$

   The string "aa" can be derived from the above grammar as follows:

   $$S \Rightarrow ACaB \Rightarrow AaaCB \Rightarrow AaaE \Rightarrow AaEa \Rightarrow AEaa \Rightarrow aa$$

   They generate exactly all languages that can be recognized by a turing machine. The language that is recognized by a turing machine is defined as all the strings on which it halts. These languages are also known as the **recursively enumerable languages**.

b. **Type 1 Grammars—Context Sensitive Grammars (CSG):** These grammars define the context sensitive languages. In CSG, all the productions of the form $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$, α and β may have any number of terminals and nonterminals.

These grammars can have rules of the form $\alpha A \beta \rightarrow \alpha\gamma\beta$ with A as nonterminal and α, β and γ are strings of terminals and nonterminals. We can replace A by γ, where A lies between α and β. Hence, the name context sensitive grammar. The strings α and β may be empty, but γ must be nonempty. It cannot include the rule $S \rightarrow \varepsilon$. These languages are exactly all languages that can be recognized by a Linear Bound Automata.

Example:

The language specified by $L(G) = \{a^n b^n \mid n >= 1\}$ is a context sensitive grammar. The grammar is represented by the following productions.

$S \rightarrow S\,B\,C \mid a\,C$
$B \rightarrow a$
$C\,B \rightarrow B\,C$
$Ba \rightarrow aa$
$C \rightarrow b$

The string "aaabbb" can be derived from the above grammar as follows:

$S \Rightarrow SBC \Rightarrow SBCBC \Rightarrow aCBCBC \Rightarrow aBCCBC \Rightarrow aaCCBC \Rightarrow aaCBCC \Rightarrow aaBCC \Rightarrow aaaCCC \Rightarrow aaabbb$

c. **Type 2 Grammars—Context Free Grammars (CFG):** These grammars define the context free languages. These are defined by rules of the form $\alpha \rightarrow \beta$ with $|\alpha| \leq |\beta$ where $|\alpha| = 1$ and is a nonterminal and β is a string of terminals and nonterminals. We can replace α by β regardless of where it appears. Hence, the name context free grammars. These languages are exactly those languages that can be recognized by a nondeterministic pushdown automaton. Context free languages define the syntax of all programming languages.

Example:

The language specified by $L(G) = \{a^n b^n \mid n >= 1\}$ is a context free grammar. The grammar is represented by the following productions.

$S \rightarrow aSb \mid \varepsilon$

The string "aabb" can be derived from the above grammar as follows:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb \Rightarrow aabb$

d. **Type 3 Grammars—Regular Grammars (RG):** These grammars generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left hand side. The right hand side consists of either a single terminal or a string of terminals with a single nonterminal on the left or the right end. Here rules can be of the form $A \rightarrow a\,B \mid a$ or $A \rightarrow Ba \mid a$. The rule $S \rightarrow \varepsilon$ is also allowed here. These languages are exactly those languages that can be decided by a finite state automaton. This family of formal languages can be obtained even by regular expressions (RE). Regular languages are used to define search patterns and the lexical structure of programming languages.

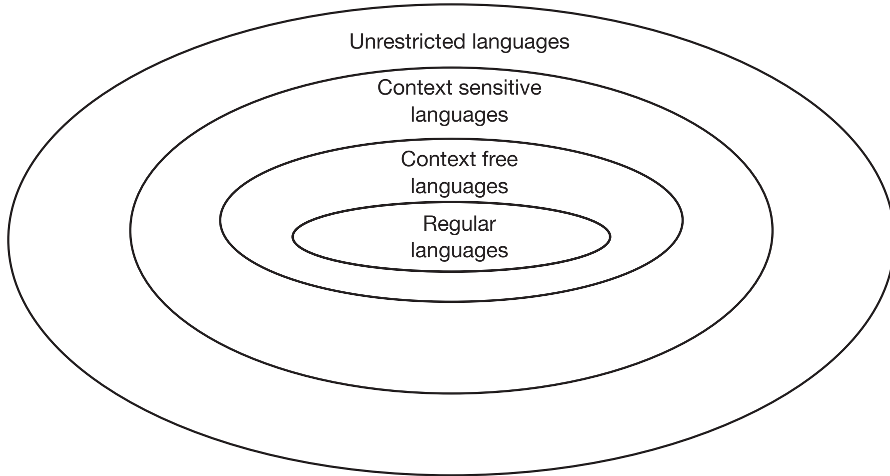
Example:

Right linear grammar: $A \rightarrow a\,A \mid a$
Left linear grammar: $A \rightarrow A\,a \mid a$
An example of a regular grammar G with $V = \{S, A\}$, $\Sigma = \{a, b, c\}$, P consists of the following rules:

$S \to aS,\ S \to bA, A \to \varepsilon, A \to cA$

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

Every regular language is context free since it is a subset of context free language; every context free language is context sensitive since it is a subset of context sensitive language and every context sensitive language is recursively enumerable. These are all proper



inclusions, meaning that there exist recursively enumerable languages that are not context sensitive; context sensitive languages are not context free and context free languages are not regular languages.

RG ⊆ Context Free ⊆ Context sensitive ⊆ Recursively enumerable

Table 3.1 summarizes each of Chomsky's four types of grammars, the class of languages it generates, the type of automaton that recognizes it, and the form of rules it must have. Here α and β represent a string of grammar symbols, that is, the string can be terminals or nonterminals.

Table 3.1 Chomsky's Hierarchy of Grammars

Type of grammar	Form	Language it defines	Corresponding automaton
Type 0 or Unrestricted Grammar	$\alpha \to \beta$ No restrictions	Recursively Enumerable	Turing Machine
Type 1 or Context Sensitive Grammar	$\alpha \to \beta$, $\lvert\alpha\rvert \le \lvert\beta\rvert$	Context Sensitive	Linear Bounded Automaton
Type 2 or Context Free Grammar	$\alpha \to \beta$, $\lvert\alpha\rvert \le \lvert\beta\rvert$, $\lvert\alpha\rvert =1$,	Context Free	Pushdown Automaton
Type 3 or Regular Grammar	$\alpha \to \beta$, $\alpha =\{V\}$ and $\beta=V\{T\}^*$ or $\{T\}^*V$ or T^*	Regular	Finite Automaton

Example 1:

Give a CSG but not CFG for (a^n | $n \geq 1$)

$S \rightarrow aS \mid X$
$aS \rightarrow aa$
$X \rightarrow a$

Give a CFG but not regular for (a^n | $n \geq 1$)

$S \rightarrow A\,S \mid a$
$A \rightarrow a$

Give a regular grammar for (a^n | $n \geq 1$)

$S \rightarrow aS \mid a$

Every regular grammar is context free, every CFG is context sensitive, and every CSG is unrestricted.

3.3 Grammar Representations

We need a way to represent grammars so that we can talk about them in general. For this, we have some standard representations.

1. Backus-Naur Form (BNF): A *metalanguage*, BNF is widely used as a notation for the grammars of computer programming languages, command sets and communication protocols, as well as a notation for representing parts of natural language grammars.

BNF is the most general representation scheme.

Example:

```
<syntax> ::= <rule>
```

We use symbols like **:=**, |, **<,>**, **(,)** etc.

The main operators here are:

1. Alteration "|"
2. Concatenation **.**"
3. Grouping "**( )**"
4. Production Sign " → " or "**::=**"

For example **A:=** $\alpha \mid \beta$ and **B**:= **a** γ | **b** γ is in BNF.

Some more examples are:

```
<simpl.e_expr> ::= <term> | <sign><term> | <simple_expr> <opr> <term>
<opr> ::= + | - | or
<unsign int> ::= <digit> | <unsign int><digit>
```

where:

- '::=' means "is defined as"
- '|' means "or"
- Terms enclosed in angle brackets, such as <term>, are *nonterminal symbols*.
- All other symbols (shown in boldface above [although it's not obvious for some characters, such as '-', in some browsers]) are *terminal symbols*.
- The concatenation of strings is represented by the juxtaposition of their descriptions. (In the example above, note that there is no space between <sign> and <term>.)

Note the recursive definition in the last example above. This is the standard way of specifying sequences of elements in BNF, and for describing nested syntax, as in:

```
<stmt> ::= <uncond_stmt>
       | if <expr> then < uncond_stmt >
       | if <expr> then < uncond_stmt > else < stmt >
```

BNF was first used in the *Algol-60 Report*.

2. Extended Backus-Naur Form (EBNF): This improves the readability and conciseness of BNF through extensions:

- *Kleene cross*—a sequence of **one or more** elements of the class marked.

  <unsign int> ::= <digit>$^+$

- *Kleene star*—a sequence of **zero or more** elements of the class marked.

  <id> : = <letter><alphanumeric>*

- Braces can be used for **grouping** elements, allowing us to avoid having to define intermediate classes such as <alphanumeric>.

  <id> : = <letter>{<letter> | <digit>}*

  Note: We generally use braces to mean zero or more repetitions of the enclosed elements (i.e., a Kleene star is assumed).

- Square brackets may be used to indicate **optional** elements (i.e., zero or one occurrence of the enclosed elements).

  <integer> : = [+ | –]<unsigned integer>

  Do not use a Kleene cross or star with square brackets.

  Stacking alternatives result in an even more pictorial representation of the syntax.

  <digit>

  <identifier> : = <letter> { <letter> }*

  <integer> : = [+ –]<unsigned integer>

EBNF is a simple extension of BNF, which tries to extend BNF and makes it easier to represent complex forms.

We add two new symbols to our original BNF symbol set viz.

1. **{}**—Repetition or Closure: Used when one pattern is repeated more than once.
2. **[]**—Optional: Used to represent optional patterns whose inclusion is not necessary.

   Example, to represent $\mathbf{A} \rightarrow \mathbf{A}\ \alpha \mid \beta$

   The corresponding RE is $\mathbf{A} : \beta\ \alpha^*$

   We can represent this in EBNF format as $\mathbf{A} \rightarrow \beta\ \{\ \alpha\ \}$

   Another example is:

   Consider the BNF form as follows:

   stmt → if (expr) stmt | if (expr) stmt else stmt

   The equivalent EBNF form is:

   stmt → if (expr) stmt [else stmt]

Note: An advantage of EBNF is that it is very easy to write the corresponding parser for it.

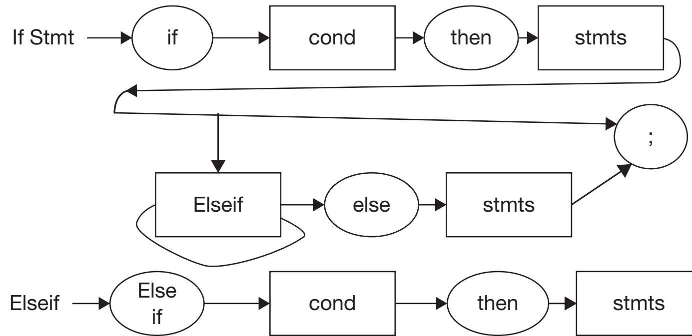


Figure 3.1 Syntax Diagrams

3. Syntax Diagrams: They are also called **railroad diagrams**. Syntax diagrams are a way to represent a context free grammar. They represent a graphical alternative to Backus-Naur Form BNF or EBNF. Figure 3.1 shows syntax diagrams of the conditional statement "if."

Syntax diagrams were first used in the *Pascal User Manual and Report* by Jensen and Wirth. The grammar can be represented as a set of syntax diagrams. Each diagram defines a variable or nonterminal. There is a main diagram that defines the language in the following way: to belong to the language, a word must describe a path in the main diagram.

Each diagram has an entry point and an end point. The diagram describes possible paths between these two points by going through other nonterminals and terminals. Terminals are represented by round boxes, while non terminals are represented by square boxes.

Example 2:

We use grammar that defines arithmetic expressions as an example. Following is the simplified BNF grammar for arithmetic expressions:

```
<Expr> ::= <Term> | <Term> "+" <Expr>
<Term> ::= <Fact> | <Fact> "*" <Term>
<Fact> ::= <num> | <id> | "(" <Expr> ")"
<id> ::= "x" | "y" | "z"
<num> ::= <digit> | <digit> <num>
<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

This grammar can also be expressed in EBNF:

```
Expr = Term, {"+", Term};
Term = Fact, {"*", Fact};
Fact = num | id | "(", Expr, ")";
id = "x" | "y" | "z";
num = digit, {digit};
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
```

The set of rail road diagrams or syntax diagrams for this grammar are shown in Figure 3.2.

Nonterminals are represented by rectangular boxes, whereas terminals are represented by circles or ellipses.

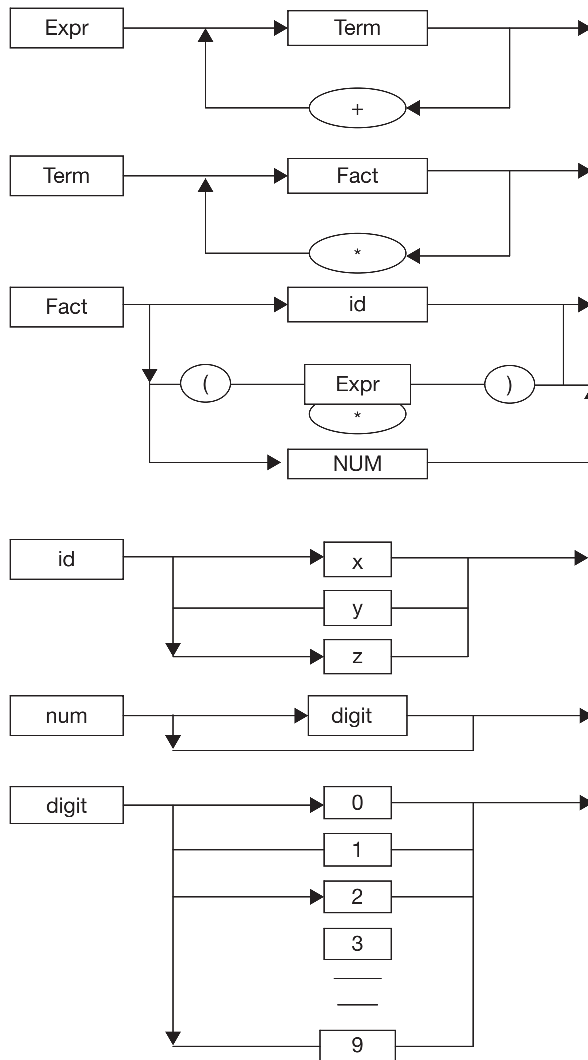


Figure 3.2 Syntax Diagrams for Expressions

3.4 Context Free Grammars

For a regular grammar, the productions are restricted in two ways: The left side must be a single variable and the right side can be any string of terminals and nonterminals. To create grammars that are more powerful, we must ease off some of the restrictions. By permitting anything on the right side, but retaining restrictions on the left side, we get context free grammars.

A grammar G = (V, T, P, S) is said to be context free if all production in P have the form $A \rightarrow x$ where $A \in V$ and $x \in (V \cup T)^*$.

V, T, P, S are the four important components in the grammatical description of a language.

V—the set of variables, also called ***nonterminals***. Each variable represents a set of strings, simply a language.

T—the set of terminals, which are a set of symbols that forms the strings of the language, also called terminal symbols.

P—the finite set of ***productions*** or rules that represent the recursive definition of language. Each production consists of a variable, production symbol →, and a string of terminals and nonterminals. The string is called ***body*** of production.

S—the start symbol. It is one of the variables that represent the language being defined.

The language generated (defined, derived, produced) by a CFG is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions. A language generated by a CFG is called a ***context free language*** (CFL).

Example 3:

terminal: a
nonterminal: S
productions: $S \rightarrow aS$
$S \rightarrow \varepsilon$

is a simple CFG that defines L(G) = a*.
where V = {S} T = {a}

Example 4:

The CFG for defining palindrome over {a or b}.
The productions P are:

$$S \rightarrow \varepsilon \mid a \mid b$$
$$S \rightarrow aSa$$
$$S \rightarrow bSb$$

And the grammar is G = ({S}, {a,b}, P, S)

Example 5:

The CFG for set of strings with equal no of a's and b's.
The productions P are:

$$S \rightarrow SaSbS \mid SbSaS \mid \varepsilon$$

And the grammar is G = ({S}, {a,b}, P, S)

Example 6:

The context free grammar for syntactically correct infix algebraic expressions in the variables x, y, and z:

And the grammar is G = ({S,T}, {+,*,(,),–,x,y,z}, P, S)

$$S \rightarrow T + S \mid T - S \mid T$$
$$T \rightarrow T * T \mid T \mid T$$
$$T \rightarrow (S)$$
$$T \rightarrow x \mid y \mid z$$

This grammar can generate the string (x + y) * x – z * y | (x + x).

Example 7:

A context free grammar for the language consisting of all strings over {a, b} which contain a different number of a's than b's is

$S \rightarrow U \mid V$
$U \rightarrow T\,a\,U \mid T\,a\,T$
$V \rightarrow T\,b\,V \mid T\,b\,T$
$T \rightarrow a\,T\,b\,T \mid b\,T\,a\,T \mid \varepsilon$

Here, 'T' can generate all strings with the same number of a's as b's, 'U' generates all strings with more a's than b's and 'V' generates all strings with less a's than b's.

Example 8:

a. Give the CFG for RE (011 +1)*(01)*

Solution:

CFG for (011 + 1)* is $A \rightarrow CA \mid \varepsilon$
$C \rightarrow 011 \mid 1$
CFG for (01)* is $B \rightarrow DB \mid \varepsilon$
$D \rightarrow 01$
Hence, the final CFG is $S \rightarrow AB$
$A \rightarrow CA \mid \varepsilon$
$C \rightarrow 011 \mid 1$
$B \rightarrow DB \mid \varepsilon$
$D \rightarrow 01$

b. Give the CFG for language $L(G) = a^n b^{2n}$ where $n \geq 1$.
Give the CFG for RE (011 + 1)*(01)*

Solution:

The given language is a*(bb)*.
Hence, it can be defined as
$S \rightarrow aSbb \mid abb$

c. Give the CFG for language containing all the strings of different first and last symbols over $\Sigma = \{0,1\}$

Solution:

The strings should start and end with different symbols 0, 1. But in between we can have any string on 0,1 i.e., (0 + 1)*. Hence, the language is

0(0 + 1)*1 | 1(0 + 1)*0. The grammar can be given by
$S \rightarrow 0A1 \mid 1A0$
$A \rightarrow 0A \mid 1A \mid \varepsilon$

3.5 Derivation of CFGs

It is a process of defining a string out of a grammar by application of the rules starting from the starting symbol. We can derive terminal strings, beginning with the start symbol, by repeatedly replacing a variable or nonterminal by the body of the production. The language of CFG is the set of terminal symbols we can derive; so it's called context free language.

Example 9:

Derive 'a^4' from the grammar given below
Terminal: a

Nonterminal: S
Productions: $S \rightarrow aS$
$S \rightarrow \varepsilon$

Solution:

The derivation for a^4 is:

$$
\begin{aligned}
S &\Rightarrow aS \\
&\Rightarrow aaS \\
&\Rightarrow aaaS \\
&\Rightarrow aaaaS \\
&\Rightarrow aaaa\varepsilon \Rightarrow aaaa
\end{aligned}
$$

The language has strings as {ε, a, aa, aaa, …….}

Example 10:

Derive "a^2" from grammar
Terminal: a
Nonterminal: S
Productions: $S \rightarrow SS$
$S \rightarrow a$
$S \rightarrow \varepsilon$

Solution:

Derivation of a^2 is as follows:

$$
\begin{aligned}
S &\Rightarrow SS \\
&\Rightarrow SSS \\
&\Rightarrow SSa \\
&\Rightarrow SSSa \\
&\Rightarrow SaSa \\
&\Rightarrow \varepsilon aSa \\
&\Rightarrow \varepsilon a\varepsilon a = aa
\end{aligned}
$$

The string can also be derived as

$$
\begin{aligned}
S &\Rightarrow SS \\
&\Rightarrow Sa \\
&\Rightarrow aa
\end{aligned}
$$

Example 11:

Find L(G) and derive "abbab."
Terminals: a, b
Nonterminals: S
Productions:
$S \rightarrow aS$
$S \rightarrow bS$
$S \rightarrow a$
$S \rightarrow b$

Solution:

More compact notation:

$$S \rightarrow aS \mid bS \mid a \mid b$$

Derive abbab as follows:

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow abS \\ &\Rightarrow abbS \\ &\Rightarrow abbaS \\ &\Rightarrow abbab \end{aligned}$$

L(G) is $(a + b)^+$

Example 12:

Find the language and derive 'abbaaba' from the following grammar:

Terminals: a, b
Nonterminals: S, X
Productions:

$$S \rightarrow XaaX$$
$$X \rightarrow aX \mid bX \mid \varepsilon$$

Solution:

CFL is $(a + b)^* aa(a + b)^*$

Derive 'abbaaba' as follows:

$$\begin{aligned} S &\Rightarrow XaaX \\ &\Rightarrow aXaaX \\ &\Rightarrow abXaaX \\ &\Rightarrow abbXaaX \\ &\Rightarrow abb\varepsilon aaX \Rightarrow abbaaX \\ &\Rightarrow abbaabX \\ &\Rightarrow abbaabaX \\ &\Rightarrow abbaaba\varepsilon \Rightarrow abbaaba \end{aligned}$$

3.6 Language Defined by Grammars

The only way to recognize the language is to try out various strings from the given production rules. Simply by observing the derived strings, one can find out the language generated from the given CFG.

Example 13:

Give the language defined by grammar G.

$G = \{\{S\}, \{a\}, \{S \rightarrow SS\}, S\}$

Solution: $L(G) = \Phi$

Example 14:

Give the language defined by grammar G.

$G = \{\{S,C\},\{a,b\}, P, S\}$ where P is given by

$$S \rightarrow aCa$$
$$C \rightarrow aCa \mid b$$

Solution:

$$\begin{aligned} S &\Rightarrow aCa \\ &\Rightarrow aaCaa \end{aligned}$$

$\Rightarrow$ aaaCaaa

L(G) = $a^n b a^n$ for $n \geq 1$

Example 15:

Give the language defined by grammar G.

G= {{S},{0,1},P,S} where P is given by

$S \rightarrow 0S1 \mid \varepsilon$

Solution:

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$

L(G) = $0^n 1^n$ for $n \geq 0$

3.6.1 Leftmost and Rightmost Derivation

The leftmost nonterminal in a working string is the first nonterminal that we encounter when we scan the string from left to right.

For example, in the string "bbabXbaY SbXbY," the leftmost nonterminal is X.

> If a word w is generated by a CFG by a certain derivation and at each step in the derivation, a rule of production is applied to the leftmost nonterminal in the working string, then this derivation is called a leftmost derivation (LMD).

Practically, whenever we replace the leftmost variable first in a string, then the resulting derivation is the leftmost derivation. Similarly, replacing rightmost variable first at every step gives rightmost derivation RMD.

Example 16:

Consider the CFG:({S,X},{a,b),P,S)

where productions are:

$S \rightarrow baXaS \mid ab$

$X \rightarrow Xab \mid aa$

Find LMD and RMD for string w = baaaababaab.

Solution:

The following is an LMD:

$S \Rightarrow baXaS$

$\Rightarrow baXabaS$

$\Rightarrow baXababaS$

$\Rightarrow baaaababaS$

$\Rightarrow baaaababaab$

The following is an RMD:

$S \Rightarrow baXaS$

$\Rightarrow baXaab$

$\Rightarrow baXabaab$

$\Rightarrow baXababaab$

$\Rightarrow baaaababaab$

Any word that can be generated by a given CFG can have LMD | RMD.

Example 17:

Consider the CFG:

$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$

Find LMD and RMD for string w = aabbabba.

Solution:

The following is an LMD:

$S \Rightarrow aB \Rightarrow aaBB$
$\Rightarrow aabSB$
$\Rightarrow aabbAB$
$\Rightarrow aabbaB$
$\Rightarrow aabbabS$
$\Rightarrow aabbabbA$
$\Rightarrow aabbabba$

The following is an RMD:

$S \Rightarrow aB \Rightarrow aaBB$
$\Rightarrow aaBbS$
$\Rightarrow aaBbA$
$\Rightarrow aaBbba$
$\Rightarrow aabSbba$
$\Rightarrow a\ abbAbba$
$\Rightarrow aaabbabba$

3.6.2 Derivation Tree

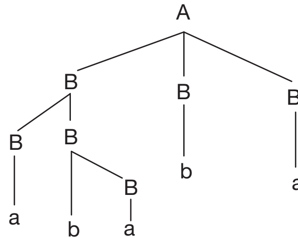
The process of derivation can be shown pictorially as a tree called derivation tree to illustrate how a word is derived from a CFG. These trees are called syntax trees, parse trees, derivation trees. These trees clearly depict how the symbols of terminal strings are grouped into sub-strings, each of which belongs to a language defined from one of the variables of grammar. For constructing a parse tree for a grammar G = (V, T, P, S)

- the start symbol S becomes root for the derivation tree
- variable or nonterminal in set V is marked as interior node
- Leaf node can be a terminal, or ε.
- For each production in P like $A \rightarrow Y_1, Y_2, \ldots, Y_k$, if an interior node is labeled A, and its children are labeled $Y_1, Y_2, \ldots, Y_k$ respectively, from the left to right.

Example 18:

CFG:
Terminals: a, b
Nonterminals: S, B
Production $A \rightarrow BBB \mid BB$
$B \rightarrow BB \mid bB \mid Ba \mid a \mid b$

String "ababa" has the following derivation tree:



By concatenating the leaves of the derivation tree from left to right, we get a string which is known as ***yield*** of the derivation tree. The yield is a string that is always derived from the root of the tree. There are derivation trees whose yields are in the language of underlying grammar. Such trees are important because of the following reasons:

- The root is labeled by the start symbol.
- The terminal string is yield. All leaves are labeled with a terminal "b" or ε.
- The strings we get in the intermediate step are called the sentential form.

3.6.3 Equivalence of Parse Trees and Derivations

A terminal string is in the language of a grammar iff it is the yield of at least one parse tree. The existence of leftmost derivation, rightmost derivation, and parse trees are equivalent conditions that define exactly the strings in the language of a CFG.

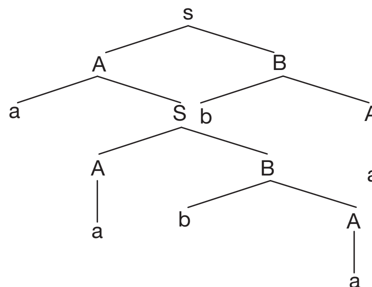
There are some CFGs for which it is possible to find a terminal string with more than one parse tree, or equivalently, more than one leftmost derivation and one rightmost derivation. Such a grammar is called ***ambiguous***.

High-level programming languages are generally the class of languages that can be parsed with a CFG.

Example 19:

The parse tree below represents a leftmost derivation according to the grammar

$S \rightarrow AB$, $A \rightarrow aS \mid a$, $B \rightarrow bA$.



Give the left-sentential forms in this derivation.

Solution:

To construct a leftmost derivation from a parse tree, we always replace the leftmost nonterminal in a left-sentential form. It helps to remember that each symbol of each left-sentential form corresponds to a node N of the parse tree.

When a nonterminal is replaced by a production body, the introduced symbols correspond to the children of node N, in order from the left.

Thus, we start with the left-sentential form S, where the symbol S corresponds to the root of the parse tree. At the first step, S is replaced by AB, the children of the root; A corresponds to the left child of the root, and B to the right child. Since A is the leftmost nonterminal of AB, it is replaced by the string "aS," formed by the labels of the two children of the left child of the root. The next left-sentential form is thus aSB. We proceed in this manner, next replacing the S. The complete leftmost derivation is:

$$S \Rightarrow AB \Rightarrow aSB \Rightarrow aABB \Rightarrow aaBB \Rightarrow aabAB \Rightarrow aabaB \Rightarrow aababA \Rightarrow aababa.$$

So S, AB, aSB, aABB, aaBB, aabAB, aabaB, aababA are left-sentential forms, whereas "aSbA" is not left-sentential form.

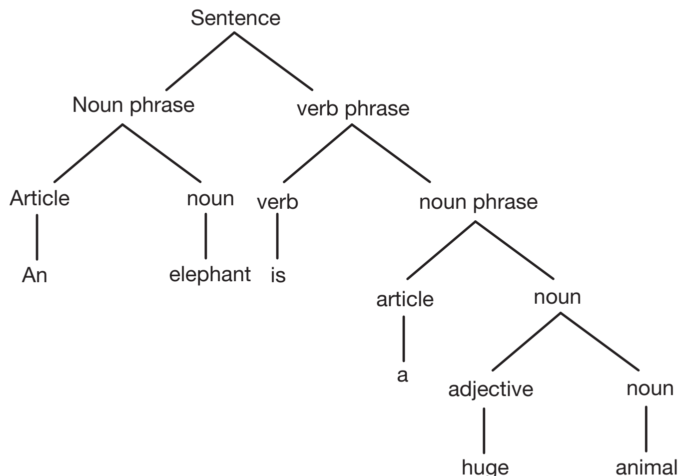
This is actually a sentential form, but neither rightmost nor leftmost. One derivation according to this parse tree is $S \Rightarrow AB \Rightarrow aSB \Rightarrow aSbA$. Remember, in a leftmost derivation, we must replace the leftmost nonterminal at every step.

Example 20:

Draw the derivation tree for the following natural language grammar.

```
<sentence> → <noun phrase> <verb phrase>
<noun phrase> → <article> <noun>
<verb phrase> → <verb> <noun phrase>
<article> → an | a
<noun> → <adjective> <noun>
<verb> → is | was
<noun> → Tiger | Cat | elephant | animal
<adjective> → small | huge | dangerous
```

The parse tree for the sentence "An elephant is a huge animal" is given by



Issues to resolve when writing a CFG for a programming language:

1. Left recursion
2. Indirect left recursion
3. Left factoring
4. Ambiguity

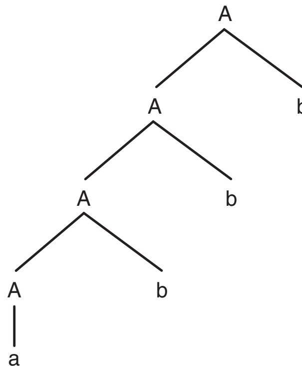
3.7 Left Recursion

A grammar is ***left recursive*** if it has a nonterminal A such that there is a derivation.

$A \Rightarrow^+ A\alpha$ for some string α. If this grammar is used in some parsers (top-down parser), parser may go into an infinite loop. Consider the following left recursive grammar

$A \rightarrow A\,b \mid a$

To derive a string "abbb," there is an ambiguity as to how many times the nonterminal "A" has to be expanded. As grammar is left recursive, the tree grows toward left.



Top-down parsing techniques **cannot** handle left-recursive grammars. So, we have to convert our left-recursive grammar into an equivalent grammar, which is not left-recursive. The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion:

A grammar G is left recursive if there is rule of the form

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A, that is, leftmost nonterminal on the right hand side is same as the nonterminal on the left hand side.

To eliminate immediate left recursion rewrite the grammar as

$A \rightarrow \beta A'$ $A' \rightarrow \alpha A' \mid \varepsilon$. This is an equivalent grammar G ′, which is free of left recursion.

L(G) is $\beta\,\alpha^*$ and L(G′) is also $\beta\ \alpha^*$ if α, β are assumed as nonterminals.

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ Eliminate immediate left recursion

$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion – Example

$E \to E{+}T \mid T$
$T \to T{*}F \mid F$
$F \to id \mid (E)$

After eliminating immediate left recursion, we get grammar as

$E \to T\,E'$
$E' \to +\,T\,E' \mid \varepsilon$
$T \to F\,T'$
$T' \to {*}F\,T' \mid \varepsilon$
$F \to id \mid (E)$

Left-Recursion – Problem

A grammar cannot be immediately left-recursive, but it still can be left-recursive. By just eliminating the immediate left-recursion, we may not get a grammar that is not left-recursive.

$S \to Aa \mid b$
$A \to Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.
$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion – Algorithm

Arrange nonterminals in some order: $A_1 \dots A_n$

```
for i from 1 to n do {
        for j from 1 to i-1 do {
              replace each production
                     Ai → Aj γ
                       by
                     Ai → α1 γ | ... | αk γ
                     where Aj → α1 | ... | αk
    }
   eliminate immediate left-recursions among Ai productions
}
```

Example 21:

Eliminate left-recursion in the following grammar:

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid f$

Order of nonterminals: S, A

for S:

We do not enter the inner loop.

There is no immediate left recursion in S.

For A:

Replace $A \to Sd$ with $A \to Aad \mid bd$

So, we will have $A \to Ac \mid Aad \mid bd \mid f$

Eliminate the immediate left-recursion in A

$A \to bdA' \mid fA'$

$A' \to cA' \mid adA' \mid \varepsilon$

So, the resulting equivalent grammar, which is not left-recursive is:

$S \to Aa \mid b$

$A \to bdA' \mid fA'$

$A' \to cA' \mid adA' \mid \varepsilon$

Example 22:

Eliminate left-recursion in the following grammar

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid \varepsilon$

Order of nonterminals: A, S

For A:

We do not enter the inner loop.

Eliminate the immediate left-recursion in A

$A \to SdA' \mid A'$

$A' \to cA' \mid \varepsilon$

For S:

Replace $S \to Aa$ with $S \to SdA'a \mid A'a$

So, we will have $S \to SdA'a \mid A'a \mid b$

Eliminate the immediate left-recursion in S

$S \to A'aS' \mid bS'$

$S' \to dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar, which is not left-recursive is:

$S \to A'aS' \mid bS'$

$S' \to dA'aS' \mid \varepsilon$

$A \to SdA' \mid A'$

$A' \to cA' \mid \varepsilon$

So by eliminating left recursion we can avoid top-down parser to go into infinite loop.

3.8 Left-Factoring

Sometimes we find common prefix in many productions like $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$, where α is common prefix. While processing α we cannot decide whether to expand A by $\alpha\beta_1$ or by $\alpha\beta_2$. So this needs back tracking. To avoid such problem, grammar can be left factoring.

Left factoring is a process of factoring the common prefixes of the alternatives of grammar rule. If the production of the form $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ has α as common prefix, by left factoring we get the equivalent grammar as

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3$

So, we can immediately expand A to $\alpha A'$.

One can also perform *left factoring* to avoid backtracking or for factoring the common prefixes. If the end result has no look ahead or backtracking needed, the resulting CFG can be solved by a "predictive parser" and coded easily in a conventional language. If backtracking is needed, a recursive descent parser takes more work to implement, but is still feasible. As a more concrete example:

$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S$

$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S_1\ \text{else}\ S_2$

can be factored to:

$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S\ S'$

$S' \rightarrow \text{else}\ S_2 \mid \varepsilon$

A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

Example:

Consider the following grammar

stmt → if expr then stmt else stmt
  | if expr then stmt

When we see "if," we cannot know which production rule to choose to rewrite *stmt* in the derivation. To left factor the above grammar, take out the common prefix and rewrite the grammar as follows:

stmt → if expr then stmt stmt′
stmt′ → else stmt | ε

Left-Factoring – Algorithm

- For each nonterminal A with two or more alternatives (production rules) with a common nonempty prefix, let us say

  $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$

  Convert it into

  $A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$

  $A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$

Example 23:

Left-factor the following grammar

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

Solution:

Here common prefixes are "a" and "cd." So first takeout 'a' and rewrite the grammar as

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

Now take out the common prefix 'cd' and rewrite the grammar as

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Example 24:

Left-factor the following grammar

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

Solution:

Here common prefixes are "a" and "ab." So first take out "a."

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

Now take out the common prefix "b" and rewrite the grammar as

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

The above problem can also be solved by taking the longest match "ab" first.

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow abA' \mid b \mid ad \mid a$

$A' \rightarrow \varepsilon \mid c$

Now there is a common prefix "a." So take that out now

$A \rightarrow a\,A'' \mid b$

$A' \rightarrow \varepsilon \mid c$

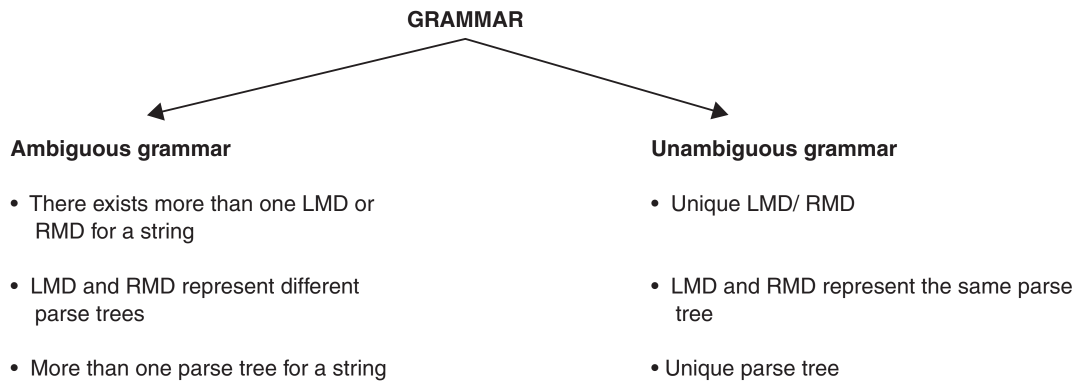
$A'' \rightarrow \varepsilon \mid bA' \mid d$

Finally, both ways we get the same solution.

So by left factoring we can avoid backtracking.

3.9 Ambiguous Grammar

A CFG is ambiguous if there exists more than one parse tree or equivalently, more than one leftmost derivation and one rightmost derivation for at least one word in its CFL.



Remember that there is no algorithm that automatically checks whether a grammar is ambiguous or not. The only way to check ambiguity is "to choose an appropriate input string and by trial and error find the number of parse trees." If more than one parse tree exists, the grammar is ambiguous.

There is no algorithm that converts an ambiguous grammar to its equivalent unambiguous grammar.

Example 25:

Show that the following grammar is ambiguous.

$$\mathbf{E \rightarrow id \mid E + E \mid E * E \mid E - E}$$

Solution:

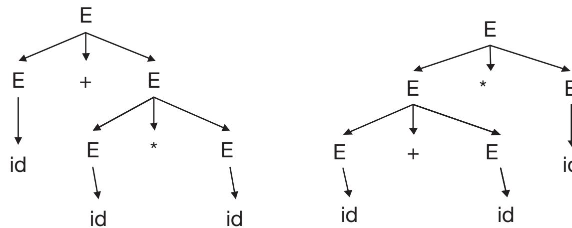
LMD: for string id + id * id is

$\mathbf{E \Rightarrow \underline{E} + E}$
$\mathbf{\Rightarrow id + \underline{E}}$
$\mathbf{\Rightarrow id + \underline{E} * E}$
$\mathbf{\Rightarrow id + id * \underline{E}}$
$\mathbf{\Rightarrow id + id * id}$

RMD: for string id + id * id is

$\mathbf{E \Rightarrow E * \underline{E}}$
$\mathbf{\Rightarrow \underline{E} * id}$
$\mathbf{\Rightarrow E + \underline{E} * id}$
$\mathbf{\Rightarrow \underline{E} + id * id}$
$\mathbf{\Rightarrow id + id * id}$

Parse trees represented by above LMD and RMD are as follows:



As there is more than one parse tree, the grammar is ambiguous.

Example 26:

The grammar G: S → SS | a | b is ambiguous. This means at least some of the strings in its language have more than one leftmost derivation. However, it may be that some strings in the language have only one derivation. Identify the string that has exactly two leftmost derivations in G.

Solution:

A string of length 1 has only one leftmost derivation, for example, S ⇒ a. A string of length 2 has only one derivation, for example, S ⇒ SS ⇒ aS ⇒ ab.

However, a string of length 3 has exactly two derivations, for example, S ⇒ SS ⇒ SSS ⇒ aSS ⇒ abS ⇒ aba and S ⇒ SS ⇒ aS ⇒ aSS ⇒ abS ⇒ aba. In general, we can decide whether the first S generates a single terminal or two S's.

On the other hand, strings of length four or more have more than two derivations. We can either start S ⇒ SS ⇒ SSS ⇒ SSSS or S ⇒ SS ⇒ SSS ⇒ aSS ⇒ aSSS or S ⇒ SS ⇒ aS ⇒ aSS, and there are other options as well.

Hence, strings like "aaa," "bbb," "aba," "bab," etc., has only two derivations.

Example 27:

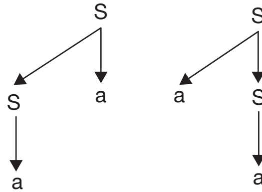
Consider the grammar 'G' with

terminals: a, b
nonterminals: S
productions: $S \rightarrow aS \mid Sa \mid a$.

Show that G is ambiguous.

Solution:

The word "aa" can be generated by two different trees:



Therefore, this grammar is ambiguous.

Example 28:

Consider the grammar "G" with

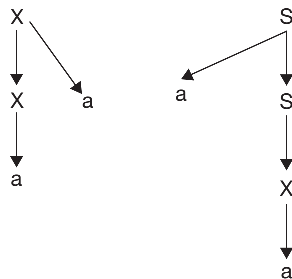
terminals: a, b
nonterminals: S, X
productions: $S \rightarrow aS \mid aSb \mid X$
$X \rightarrow Xa \mid a$

Show that G is ambiguous.

Solution:

The word "aa" has two different derivations that correspond to different syntax trees:

$S \Rightarrow X \Rightarrow Xa \Rightarrow aa$ $\qquad$ $S \Rightarrow aS \Rightarrow aX \Rightarrow aa$



Hence, the grammar is ambiguous.

Example 29:

The grammar 'G' for 'PALINDROMES' is

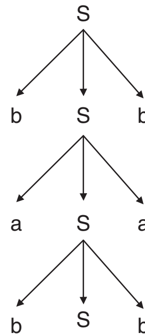
$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$. Check if G is ambiguous.

Solution:

The grammar can generate the word "babbab" as follows:

$$S \Rightarrow bSb$$
$$\Rightarrow baSab$$
$$\Rightarrow babSbab$$
$$\Rightarrow babbab$$

which has the following derivation tree:



Since there is only one parse tree, the grammar is unambiguous.

Example 30:

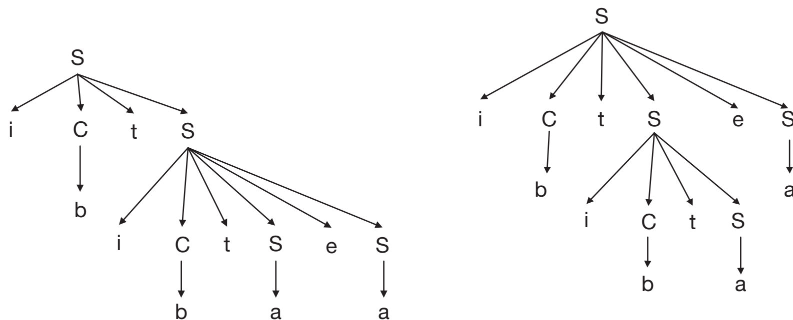
Check whether the given grammar is ambiguous or not.

$S \rightarrow i\,C\,t\,S \mid i\,C\,t\,S\,e\,S \mid a$

$C \rightarrow b$

Solution:

To check the ambiguities take an input string. Let us say the input string is "ibtibtaea." Let us draw the derivation trees.



Hence, grammar is ambiguous.

3.10 Removing Ambiguity

There is no algorithm that straightaway converts an ambiguous grammar to equivalent unambiguous grammar. But on analyzing the grammar, if we identify what is missing in the grammar and why it is unambiguous, then we can write equivalent unambiguous grammars.

For example, consider the expression grammar given below:

Expr → Expr + Expr | Expr * Expr | id

If we take a string id + id * id or id + id + id, we get two parse trees.

So if we analyze the grammar with the above two strings, we can understand the following.

1. Precedence of operators is not taken care of. Hence, you can derive the string replacing either Expr with Expr + Expr or with Expr * Expr.
2. Associative property is also not taken care of.
   If we take a string id + id + id, we can replace Expr Expr + Expr.
   Now we can replace either left Expr or right Expr.

So write equivalent unambiguous grammar by taking care of precedence and associativity.

To take care of **precedence** rewrite the grammar by defining rules starting with lowest precedence to highest precedence.

For example, in the given grammar 'id' has highest precedence, next is * and least is for +.

So do not define all of them at the same level but separate them into different levels by introducing extra nonterminals. Start defining the rules with +, then *, and finally with id as shown below.

E → E + T | T
T → T * F | F
F → id

For example, now we want to add operator unary. As unary has the highest precedence than * or +, add rule for – after * by introducing the new nonterminal P. So the resulting grammar is

E → E + T | T
T → T * F | F
F → -P | P
P → id

To ensure **associativity**, define the rule as left recursive if the operator is left associative. Define the rule as right recursive if the operator is right associative. In the given grammar, + and * are left associative. So the rules must be left recursive, that is, AAa | b. The equivalent unambiguous grammar is

E → E + T | T
T → T * F | F
F → id

This procedure can be used for any expression grammars.

Example:

Convert the following grammar to equivalent unambiguous grammar.

bExpr → bExpr and bExpr
| bExpr or bExpr
| not bExpr
| true
| false

Solution:

This is the grammar that defines Boolean expressions with basic operators 'and', 'or,' and 'not'. Here priority of operators is highest for 'not', then for 'and' and then the least for 'or.' So start defining 'or' then 'and, next 'not.' So the resulting grammar is

$E \rightarrow E$ or $T \mid T$
$T \rightarrow T$ and $F \mid F$
$F \rightarrow$ not $F \mid$ true $\mid$ false

Example:

Convert the following grammar to equivalent unambiguous grammar.

$R \rightarrow R + R \mid RR \mid R^* \mid a \mid b$

Solution:

This is the grammar that defines regular expressions with basic operations union, concatenation, and Kleenes closure. Here, priority of operators is the highest for closure, next highest for concatenation, and least or union.
So start defining least to highest. So the resulting grammar is

$E \rightarrow E + T \mid T$
$T \rightarrow T\ F \mid F$
$F \rightarrow F^* \mid a \mid b$

So given a grammar, by using the above procedure we can get precedence and associativity of operators. For example,

$A \rightarrow A\ \#\ B \mid B$
$B \rightarrow C\ \$\ B \mid C$
$C \rightarrow C\ @\ D \mid D$
$D \rightarrow a \mid b$

Here "#," "$," and "@" are "unknown" binary operators. From the grammar, we can get precedence and associativity of operators as follows:

"@" has the highest precedence and is left associative
"$" has the next highest precedence and is right associative
"#" has the least precedence and is left associative

3.11 Inherent Ambiguity

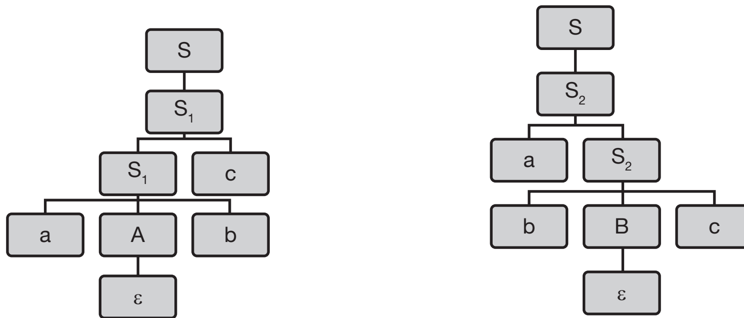
A CFL is said to be ***inherently ambiguous*** if it is defined only with ambiguous Grammar. It cannot be defined with unambiguous grammar. Following is a grammar for an inherently ambiguous language: $L = (a^i b^j c^k \mid i = j \text{ or } j = k\}$

Here $L = \{a^i b^i c^k \mid a^i b^k c^k\}$

Grammar for the above language is given by

$S \rightarrow S_1 \mid S_2$
$S_1 \rightarrow S_1 c \mid A$ $\qquad S_2 \rightarrow a S_2 \mid B$
$A \rightarrow aAb \mid \varepsilon$ $\qquad B \rightarrow b Bc \mid \varepsilon$

For example, consider a string "abc" from the language; it can be derived in two ways.



A CFG can be used to specify the **syntax** of programming languages.

3.12 Non-context Free Language Constructs

There are some language constructs found in many programming languages, which are not context free. This means that, we cannot write a context free grammar for these constructs. Following are a few examples:

- L1 = { ωcω | ω is in (a|b)*} is not context free
  → L1 abstracts the problem of declaring an identifier and checking whether it is declared or not before being used. We cannot do this with a context free language. We need semantic analyzer (which is not context free), which checks that identifiers are declared before use.
- L2 = $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context free
  → L2 is declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

  It is interesting to note that languages close to L1 and L2 are context free. For example,
- L1′ = $\{\omega c \omega^R \mid \omega \text{ is in } (a|b)^*\}$ is context free
  It is generated by grammar
  S → aSa | bSb | c.
- L2′ = $\{a^n b^m c^m d^n \mid n \geq 1 \text{ and } m \geq 1\}$ is context free.
  It is generated by grammar
  S → aSd | aAd
  A → bAc | bc
- Also L2″ = $\{a^n b^n c^m d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is context free.
  It is generated by grammar
  S → AB
  A → aAb | ab
  B → cBd | cd

3.13 Simplification of Grammars

As we have seen, various languages can be represented effectively by CFG. All the grammars are not always optimized. That means grammar may contain some extra symbols

(unnecessary symbols). These will increase the length of the grammar. Simplification of the grammar involves removing all these unnecessary symbols. For example, look at following grammar

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

Here C never defines any terminal.
E and C do not appear in any sentential form.
$E \rightarrow \varepsilon$ is a null production.
$B \rightarrow C$ simply replaces B by C.
Hence, if we simplify the grammar as follows

$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

Simplification of the grammar generally includes the following:

1. Elimination of useless symbols
2. Elimination of ε productions
3. Elimination of unit productions of the form $A \rightarrow B$

a. **Elimination of useless symbols**

A symbol is useless if it cannot derive a terminal or it is not reachable from the start symbol.

To check if the symbol is reachable from the start symbol, we can use the dependency graph or the following lemma.

Lemma 1: If the grammar G = (V,T,P,S) with $L(G) \neq \Phi$, we can effectively find an equivalent grammar G′ = (V′,T,P′,S) such that for each A in V′ there is some w in T* for which $A \stackrel{*}{\Rightarrow} w$.
If $A \rightarrow w$ is a production where $w \in T^*$, then A is in V′. If $A \rightarrow X_1X_2\ldots X_n$ is a production where each X_1 is in V′ then A is also in V′. The set V′ can be computed by the following algorithm:

1. Initialize $V1 = \Phi$.
2. Include A to V2 where $A \rightarrow w$ for some w in T*.
3. Repeat 4 and 5 while $V1 \neq V2$.
4. V1 = V2
5. $V2 = V1 \cup \{A\}$ where $A \rightarrow \alpha$ for some α in $(T \cup V1)^*$
6. V′ = V2

From the above algorithm, find all $A \in V'$; now include only those productions that include $V' \cup T$.

Lemma 2: Given a grammar G = (V,T,P,S) we can effectively find an equivalent grammar G′ = (V′,T,P′,S) such that for each X in $V' \cup T'$ there exist α and β in $(V' \cup T')^*$ for which $S \stackrel{*}{\Rightarrow} \alpha \times \beta$.

1. Initially place S in V′.
2. For all productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid .. \mid \alpha_n$ then add the variables $\alpha_1, \alpha_2, \ldots \alpha_n$ to V′ and all terminals to T′.
3. P′ is the set of productions, which includes symbols of $V' \cup T'$.

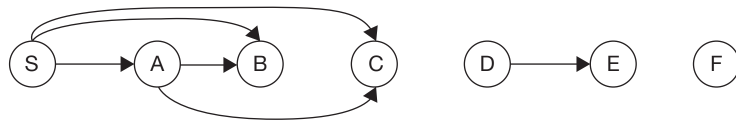
Example 31:

Eliminate useless symbols and productions from the following grammar.

$S \rightarrow ABa \mid BC, A \rightarrow aC \mid BCC, C \rightarrow a, B \rightarrow bcc,$
$D \rightarrow E, E \rightarrow d, F \rightarrow e$

Solution:

Step 1: Eliminate nongenerating symbols: All variables are generating.
Step 2: Elimination of nonreachable variables: Draw the dependency graph.



- D, E, and F are nonreachable from S.
- After removing useless symbols

$S \rightarrow ABa \mid BC$
$A \rightarrow aC \mid BCC$
$C \rightarrow a$
$B \rightarrow bcc$

Example 32:

Eliminate useless symbols in G.

$S \rightarrow AB \mid CA$
$S \rightarrow BC \mid AB$
$A \rightarrow a$
$C \rightarrow aB \mid b$

Solution:

Here B is not defined; hence B is a useless symbol.
C and A are reachable and are deriving terminals. Hence, useful.
So reduced grammar isone without useless symbols.

$S \rightarrow CA$
$A \rightarrow a$
$C \rightarrow b$

Example 33:

Eliminate useless symbols in G.

$S \rightarrow aAa$
$A \rightarrow bBB$
$B \rightarrow ab$
$C \rightarrow a\ b$

Solution:

Here C is useless as it is not reachable from the start symbol.

So reduced grammar is one without useless symbols.

$S \rightarrow aAa$
$A \rightarrow bBB$
$B \rightarrow ab$

Example 34:

Eliminate useless symbols in G.

$S \rightarrow aS \mid A \mid BC$
$A \rightarrow a$
$B \rightarrow aa$
$C \rightarrow aCb$

Solution:

Here C is useless as it is not deriving any string. B is not reachable.
So reduced grammar is the one without useless symbols.

$S \rightarrow aS \mid A$
$A \rightarrow a$

b. **Elimination of ε-Productions**

If some CFL contains the word ε, then the CFG must have a ε-production. However, if a CFG has a ε-production, then the CFL does not necessarily contain ε.

e.g.,

$S \rightarrow aX$
$X \rightarrow \varepsilon$

which defines the CFL {a}.

Nullable Variable: In a given CFG, a nonterminal X is nullable if

1. there is a production $X \rightarrow \varepsilon$
2. there is a derivation that starts at X and leads to ε:

$$X => \ldots => \varepsilon \qquad \text{i.e., } X => \varepsilon.$$

For any language L, define the language L_0 as follows:

1. If $L \Rightarrow \varepsilon$, then L_0 is the entire language L, i.e., $L_0 = L$.
2. If $\varepsilon \in L$, then L_0 is the language $L - \{\varepsilon\}$, so L_0 is all words in L except ε.

If L is a CFL generated by a CFG G_1 that includes ε-productions, then there is another CFG G_2 with no ε-productions that generates L_0.

Procedure for eliminating e productions:

a. Construct V_n set of all nullable variables.
b. For each production $B \rightarrow A$, if A is nullable variable, replace nullable variable by ε and *add* with all possible combinations on the RHS.
c. Do not add the production

$A \rightarrow \varepsilon$

Example 35:

Eliminate null production in the grammar

$S \rightarrow ABaC$
$A \rightarrow BC$
$B \rightarrow b \mid \varepsilon$
$C \rightarrow D \mid \varepsilon$
$D \rightarrow d$

Solution:

Nullable variables are $V_n = \{B,C,A\}$
Hence, equivalent grammar without null productions is

$S \rightarrow ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a$
$A \rightarrow BC \mid B \mid C$
$B \rightarrow b$
$C \rightarrow D$
$D \rightarrow d$

Example 36:

Eliminate null production in the grammar

$S \rightarrow aA$
$A \rightarrow BB$
$B \rightarrow aBb \mid \varepsilon$

Solution:

Nullable variables are $V_n = \{A,B\}$
Hence, equivalent grammar without null productions is

$S \rightarrow aA \mid a$
$A \rightarrow BB \mid B$
$B \rightarrow aBb \mid ab$

c. **Eliminating unit productions**

A unit production is a production of the form $A \rightarrow B$

If a language L is generated by a CFG G_1 that has no ε-productions, then there is also a CFG G_2 for L with no ε-productions and no unit productions.

Procedure for eliminating unit productions:

- For each pair of nonterminals A and B such that there is a production
  $A \rightarrow B$ and the non unit productions from B are
  $B \rightarrow s_1 \mid s_2 \mid \ldots \mid s_n$
  where the si ε $(\Sigma + N)^*$ are strings of terminals and nonterminals, then create the new productions as
  $A \rightarrow s_1 \mid s_2 \mid \ldots \mid s_n$
  –Do the same for all such pairs A and B simultaneously.
  –Remove all unit productions.

Example 37:

Eliminate unit productions in the grammar

$S \to A \mid bb$
$A \to B \mid b$
$B \to S \mid a$

Solution:

After eliminating unit productions $S \to A$, $A \to B$, $B \to S$, we get
$S \to a \mid b \mid bb$, $A \to a \mid b \mid bb$, $B \to a \mid b \mid bb$

Example 38:

Eliminate unit production from the grammar below:
$S \to Aa \mid B$, $B \to A \mid bb$, $A \to a \mid bc \mid B$

Solution:

Unit productions are $S \to B$, $B \to A$, and $A \to B$

- A, B, and S are derivable
- Eliminating B in the A production gives $A \to a \mid bc \mid bb$.
- Eliminating A in the B production gives $B \to a \mid bc \mid bb$.
- Eliminating B in the S production gives $S \to Aa \mid a \mid bc \mid bb$.

◆ The final set of productions after eliminating unit productions is given below:

$S \to Aa \mid a \mid bc \mid bb$
$B \to a \mid bc \mid bb$
$A \to a \mid bc \mid bb$

Example 39:

Simplify the following grammar.
$S \to aA \mid aBB$
$A \to aAA \mid \varepsilon$
$B \to bB \mid bbC$
$C \to B$

Solution:

Here it is better to eliminate null productions as this may introduce useless symbols and unit productions. Next, eliminate unit productions and at the end eliminate useless symbols.

Removing ε-productions gives resulting grammar as
$S \to aA \mid a \mid aBB$
$A \to aAA \mid aA \mid a$
$B \to bB \mid bbC$
$C \to B$

Eliminating unit productions we get the resulting grammar as
$S \to aA \mid a \mid aBB$
$A \to aAA \mid aA \mid a$
$B \to bB \mid bbC$
$C \to bB \mid bbC$

B and C are identified as useless symbols. Eliminating these we get
$S \to aA \mid a$
$A \to aAA \mid aA \mid a$

Finally, the reduced grammar is $S \to aA \mid a$, $A \to aAA \mid aA \mid a$, which defines any number of a’s.

3.14 Applications of CFG

- **Grammars** are useful in specifying syntax of programming languages. They are mainly used in the design of programming languages.
- They are also used in natural language processing.
- Tamil poetry called "Venpa" is described by context free grammar.
- Efficient parsers can be automatically constructed from a CFG.
- CFGs are also used in speech recognition in processing the spoken word.
- The expressive power of CFG is too limited to adequately capture all natural language phenomena. Therefore, extensions of CFG are of interest for computational linguistics.

Example 40:

As in example, CFG for Pascal statements are given below.

```
Stmt → begin optional_stmts end
optional_stmts → list_of_stmt | ε
list_of_stmt → list_of_stmt; Stmt | Stmt
Stmt → if Expr then Stmt
      |if Expr then Stmt else Stmt
      |while Expr do Stmt
      |id = Expr
Expr → Expr + Term | Term
Term → Term * Fctr | Fctr
Fctr → id | num
```

Solved Problems

1. Give the CFG, which generates all positive even integers up to 998.

Solution:

We need to generate positive integers with 1 digit, or 2 digits, or 3 digits.
One digit numbers are 0, 2, 4, 6, 8

Two-digit and three-digit positive integers can have any number in 10's or 100's place. Hence, we can define grammar as follows

$S \to A$ single-digit numbers
$\quad | BA$ ……double-digit numbers
$\quad | BBA$ ……Three-digit numbers
$B \to 0 \mid 1 \mid 2 \mid 3 \ldots\ldots 9$
$A \to 0 \mid 2 \mid 4 \mid 6 \mid 8$

2. Give CFG for RE (a +b)*cc(a+b)*

Solution:

The grammar for all the strings on a, b i.e., $(a + b)^*$ is $A \to aA \mid bA \mid \varepsilon$
In the middle of any string of a, b on either side, string "cc" is occurring.
So the grammar for the given regular expression is

$S \to AccA$
$A \to aA \mid bA \mid \varepsilon$

3. Give CFG for RE $\{0^m 1^n \; m, n \geq 0\}$

Solution:

The regular expression is $0^* 1^*$.
The grammar for strings a^* is $A \to aA \mid \varepsilon$
Hence the grammar for the given regular expression is:

$S \to AB$
$A \to 0A \mid \varepsilon$
$B \to 1B \mid \varepsilon$

4. Give CFG for RE $0^{2m}1^n \; m, n \geq 0$

Solution:

The given RE has zero or more strings of two zeros and one 1's.
The language is $(00)^*1^*$. So the final grammar is

$S \to AB$
$A \to 00A \mid \varepsilon$
$B \to 1B \mid \varepsilon$

5. Find the language defined by CFG.

$S \to aB \mid bA$
$A \to a \mid aS \mid bAA$
$B \to b \mid bS \mid aBB$

Solution:

To find the language defined by grammar, list the words defined by grammar.

$S \Rightarrow aB \Rightarrow ab$
$S \Rightarrow bA \Rightarrow ba$
$S \Rightarrow aB \Rightarrow abS \Rightarrow abaB \Rightarrow abab$
$S \Rightarrow bA \Rightarrow bbAA \Rightarrow bbaSA \Rightarrow bbabAa \Rightarrow bbabaa$

Hence, the language defined by the given grammar is *equal no of a's and b's*.

6. Find LMD and RMD for string 00101 in the grammar given below:

$S \to B \mid A$
$A \to 0\,A \mid \varepsilon$
$B \to 1B \mid 0B \mid \varepsilon$

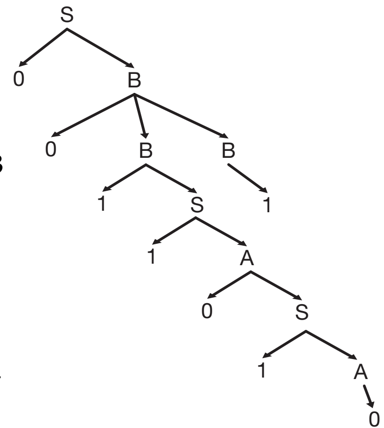
Solution:

LMD, RMD are the same.

$S \Rightarrow B \Rightarrow 0B \Rightarrow 00B \Rightarrow 001B \Rightarrow 0010B \Rightarrow 00101B$
$\Rightarrow 00101\varepsilon$
$\Rightarrow 00101$

7. Find LMD, RMD, and derivation tree for string 00110101 in the grammar given below.

$S \to 0B \mid 1A$
$A \to 0 \mid 0S \mid 1AA$
$B \to 1 \mid 1S \mid 0BB$



Solution:

LMD

$S \Rightarrow 0\underline{B} \Rightarrow 00\underline{B}B \Rightarrow 001\underline{S}B \Rightarrow 0011\underline{A}B$
$\Rightarrow 00110\underline{S}B$
$\Rightarrow 001101\underline{A}B$
$\Rightarrow 0011010B$
$\Rightarrow 00110101$

RMD

8. Check whether the following grammar is ambiguous.

$S \to 0S1 \mid SS \mid \varepsilon$

Solution:

Let us consider the string 01. This string can be generated in two ways.

i. $S \Rightarrow 0S1 \Rightarrow 01$
ii. $S \Rightarrow SS \Rightarrow 0S1S \Rightarrow 01S \Rightarrow 01$

Hence, the given grammar is ambiguous grammar.

9. Check whether the following grammar is ambiguous for w = ab

$S \to aB \mid ab$
$A \to aAB \mid a$
$B \to AB\,b \mid b$

Solution:

The string "ab" can be derived using LMD as follows:

1. $S \Rightarrow ab$
2. $S \Rightarrow a\underline{B} \Rightarrow ab$

Since there are two possible ways to derive the string, it is ambiguous for w.

10. Check whether the following grammar is ambiguous for w = aab.

$S \to AB \mid aaB$
$A \to a \mid Aa$
$B \to b$

Solution:

The string "aab" can be derived using LMD as follows:

1. $S \Rightarrow AB \Rightarrow AaB \Rightarrow aaB \Rightarrow aab$
2. $S \Rightarrow aa\underline{B} \Rightarrow aab$

Since there are two possible ways to derive the string, it is ambiguous for w.

11. Check whether the following grammar is ambiguous for w = abababa.

$S \to SbS \mid a$

Solution:

The string "aab" can be derived using LMD as follows:

1. $S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow SbSbSbS$
$\Rightarrow abSbSbS \Rightarrow ababSbS \Rightarrow abababS \Rightarrow abababa$

2. $S \Rightarrow SbS \Rightarrow abS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababSbS$
$\Rightarrow abababS \Rightarrow abababa$

Since there are two possible ways to derive the string, it is ambiguous for w.

12. Eliminate useless symbols in G.

$S \rightarrow a\ Aa$
$A \rightarrow Sb \mid bCc \mid DaA$
$C \rightarrow abb \mid DD$
$E \rightarrow a\ C$
$D \rightarrow a\ DA$

Solution:

Here D is useless as it is not deriving any string. E is not reachable.
So reduced grammar is the one without useless symbols.

$S \rightarrow aAa$
$A \rightarrow Sb \mid bCc$
$C \rightarrow abb$

13. Eliminate useless symbols in G.

$S \rightarrow a\ A \mid bB$
$A \rightarrow aA \mid a$
$B \rightarrow bB$
$D \rightarrow ab \mid Ea$
$E \rightarrow aC \mid d$

Solution:

Here B is useless as it is in loop, not deriving any string. There is no rule for C. So C is also useless. E and D are not reachable from S.
Hence, the reduced grammar is the one without useless symbols.

$S \rightarrow a\ A$
$A \rightarrow aA \mid a$

14. Left factor of the grammar $S \rightarrow abc \mid abd \mid ae \mid f$

Solution:

Here common prefix are "ab" and "a." So first take out the longest match "ab" from S.

$S \rightarrow abS' \mid ae \mid f$
$S' \rightarrow c \mid d$

Now take out the common prefix "a" and rewrite the grammar as

$\Downarrow$

$S \rightarrow aS'' \mid \mathbf{f}$
$S'' \rightarrow bS' \mid e$
$S' \rightarrow c \mid d$

15. Left factor of the grammar $S \rightarrow AaS \mid AaA,\ A \rightarrow a \mid ab$

Solution:

Here common prefix are "Aa" and "a." So first take out "Aa" from S.

$S \rightarrow AaA'$
$A' \rightarrow S \mid A$

Now take out the common prefix "a" from A and rewrite the grammar as

$\Downarrow$

$A \rightarrow aA''$
$A'' \rightarrow \varepsilon \mid b$

16. Eliminate left recursion and left factor of the grammar

$E \rightarrow aba \mid abba \mid Ea \mid EbE$

Solution:

Here first eliminate left recursion; then we get equivalent grammar as

$E \rightarrow abaE' \mid abbaE'$
$E' \rightarrow bE' \mid bEE' \mid \varepsilon$

Now left factor the grammar and rewrite the grammar as

$\Downarrow$

$E \rightarrow abA$
$A \rightarrow aE' \mid \mathbf{b}\, aE'$
$E' \rightarrow bB$
$B \rightarrow E' \mid \mathbf{E}E' \mid \varepsilon$

Summary

- Context free grammar mainly defines the syntax of the programming language and push-down automata is used to recognize the language.
- A context free grammar can be simplified by eliminating useless symbols or null productions or unit productions.
- An equivalent grammar can be constructed for any language without null by eliminating null productions.
- Context free grammars can be represented in standard form using Chomsky or Greibach normal forms.

Fill in the Blanks

1. _________verifies if the tokens are properly sequenced in accordance with the grammar of the language.
2. The language defined by $S \rightarrow SS$ is _________.
3. A nonterminal is useless if it is _________.
4. A variable that derives ε is called the _________ variable.
5. Left linear grammars are _________ of CFG.
6. CFGs are _________ of CSG.
7. If there is a unique LMD, then the grammar is _________.
8. The grammar is CFG. (True | False)
9. $S \rightarrow bbba \mid \varepsilon$

10. $A \rightarrow \varepsilon$
11. Context free languages are described by type _________ grammars.
12. In Type 1 grammars, if $\alpha \rightarrow \beta$, then relation between α and β is _________.
13. To simplify the grammar $S \rightarrow ST \mid \varepsilon, T \rightarrow abT \mid ab$ we apply elimination of _________.
14. _________ is a context free grammar to generate expression with balanced parenthesis.
15. Grammar $S \rightarrow \alpha S \beta \mid SS \mid \varepsilon$ is _________ grammar.
16. For every ambiguous grammar there is equivalent grammar for the same language which is unambiguous. (True | False)
17. _________ languages are the subset of context free languages.
18. Every context free language is a context sensitive language. (True | False)
19. Elimination of null productions results in simplified grammar with no unit productions and useless symbols. (True | False)
20. If a grammar has different LMD or RMD, then it is ambiguous. (True | False)

Objective Question Bank

1. A context free grammar is ambiguous if
   (a) The grammar contains useless nonterminals.
   (b) It produces more than one parse tree for some sentence.
   (c) Some production has two nonterminals side by side on the right-hand side.
   (d) None of the above

2. Find the number of parse trees for an input string "aaa" in $S \rightarrow Sa \mid aS \mid a$
   (a) 2 (b) 3 (c) 4 (d) infinite

3. Find the number of parse trees for an input string "abab" in $S \rightarrow aSbS \mid \mathbf{bS}aS \mid \varepsilon$
   (a) 2 (b) 3 (c) 4 (d) infinite

4. What is the language generated by the CFG?
   $S \rightarrow aSb$
   $S \rightarrow aAb$
   $S \rightarrow aBb$
   $A \rightarrow aA \mid a$
   $B \rightarrow Bb \mid b$

   Here V = { S, A, B) and T = {a, b}
   (a) $\{a^n b^m, m > 0, |n - m| > 2\}$
   (b) $\{a^n b^m, m > 1, |n - m| > 1\}$
   (c) $\{a^n b^m, m > 0, |n - m| > 1\}$
   (d) $\{a^n b^m, m > 0, |n - m| > 0\}$

*5. Consider the grammar
   Stmt → if id then stmt
   | if id then stmt else stmt
   | id = id

Which of the following is not true?
(a) The string "if a then if b then c=d" has two parse trees.
(b) The LMD, RMD of "if a then if b then c=d" represent different parse trees.
(c) The string "if a then if b then c = d else e = f" has more than two parse trees.
(d) The string "if a then if b then c=d else e=f" has two parse trees.

6. Let L denote the language generated by the grammar $S \rightarrow 0S0 \mid 00$. Which of the following is true?
(a) L = 0+ (b) L is regular but not 0+
(c) L is context free but not regular (d) L is not context free

7. Aliasing in the context free programming languages refers to
(a) multiple variables having the same memory location
(b) multiple variables having the same values
(c) multiple variables having the same memory identifier
(d) multiple uses of the same variable

8. Let G = ({s}), {a,b}, R, s) be a context free grammar where the rule set R
$S \rightarrow a\,S\,b \mid S\,S \mid \varepsilon$
Which of the following is true?
(a) G is not ambiguous
(b) There exist $x, y \in L(G)$ such that $xy \notin L(G)$.
(c) There is a deterministic PDA that accepts L(G).
(d) We can find a DFA that accepts L(G).

9. The grammar $S \rightarrow Sa \mid aS \mid a$ is
(a) left recursive (b) right recursive
(c) unambiguous grammar (d) ambiguous

10. The following CFG $S \rightarrow aS \mid bS \mid a \mid b$ is equivalent to the regular expression
(a) (a * + b)* (b) (a+b)* (c) (a + b)(a+b)* (d) (a+b)*(a+b)

11. Any string of terminals that can be generated by the following CFG
$S \rightarrow xy$, $X \rightarrow ax \mid bx \mid a$, $Y \rightarrow ya \mid yb \mid a$
(a) has at least one b (b) should end in a 'a'
(c) has no consecutive a's or b's (d) has at least two a's

12. The following CFG
$S \rightarrow aB \mid bA$, $A \rightarrow a \mid aS \mid bAA$, $B \rightarrow b \mid bS \mid aBB$
Generates strings of terminals that have
(a) Equal number of a's and b's
(b) odd number of a's and odd number of b's
(c) even number of a's and even number of b's
(d) odd number of a's and even number of b's

13. The set $\{a^n b^n \mid n = 1,2,3\ldots\}$ can be generated by the CFG
(a) $S \rightarrow ab \mid aSb$ (b) $S \rightarrow aaSbb \mid ab$
(c) $S \rightarrow ab \mid aSb \mid \varepsilon$ (c) $S \rightarrow aaSbb \mid ab \mid aabb$

14. Choose the correct statements.
    (a) All languages can be generated by CFG.
    (b) Any regular language has an equivalent CFG.
    (c) Some non-regular languages can't be generated by any CFG.
    (d) Some regular languages can't be generated by any CFG.

15. Here is a context free grammar G:

    S → AB
    A → 0A1 | 2
    B → 1B | 3A

    Which of the following strings is in L(G)?
    (a) 021300211 (b) 00213021 (c) 002111300211 (d) 0211300021

16. The grammar

    A → B ↑ A / A * B | B
    B → B - B / C / B | C
    C → i

    (a) reflects that - has high precedence than - (b) reflects that - has high precedence than /
    (c)reflects that - has high precedence than ↑ (d) none of the above

17. Consider the following four grammars:
    1. S → abS | ab
    2. S → SS | ab
    3. S → aB; B → bS | b
    4. S → aB | ab; B → bS

    The initial symbol is S in all cases. Determine the language of each of these grammars. Then, find, in the list below, the pair of grammars that define the same language.
    (a) G1: S → aB, B → bS, B → ab G2: S → SS, S → ab
    (b) G1: S → SS, S → ab G2: S → aB, B → bS, B → b
    (c) G1: S → abS, S → ab G2: S → aB, B → bS, S → a
    (d) G1: S → aB, B → bS, B → ab G2: S → aB, B → bS, S → ab

18. Consider the grammar S → a S b | b S a | ⇒
    The number of parse trees the grammar generates for an input string "abab" is
    (a) 1 (b) 2 (c) 3 (d) 4

19. Consider the grammar R → RR | **R + R** | **R*** | **a** | **b is**
    (a) ambiguous (b) unambiguous
    (c) inherently ambiguous (d) None of the above

20. Consider the following CFG. G is defined by productions

    S → aSbS | bSaS | ε

    The language generated by this CFG is
    (a) the set of all strings which contains even number of a's and even number of b's
    (b) the set of all strings which contains odd number of a's and even number of b's

(c) the set of all strings which contains odd number of a's and odd number of b's
(d) the set of all strings with an equal number of a's and equal number of b's

21. Consider the following grammar productions

    $S \rightarrow AB$,
    $A \rightarrow BB \mid a$,
    $B \rightarrow AB \mid b$,

    Choose the incorrect statement.
    (a) aabbb can be derived from the above grammar.
    (b) aabb can be derived from the above grammar.
    (c) ababab can be derived from the above grammar.
    (d) aaabb can be derived from the above grammar.

22. Consider the following context free grammars
    1. $S \rightarrow aSbb \mid a$
    2. $S \rightarrow aSA \mid a$, $A \rightarrow bB$, $B \rightarrow b$

    Which of the following is correct?
    (a) The language generated by '1' is subset of '2'.
    (b) The language generated by '2' is subset of '1'.
    (c) The language generated by both the grammars '1' and '2' is one and the same.
    (d) None of the above

23. The grammar

    $S \rightarrow a\,S\,b \,/\, SS \mid \in$ is ambiguous as "ab" has

    (a) one LMD and one RMD (b) two LMD
    (c) No RMD (d) all true

24. A context free grammar is said to be ambiguous if it has
    (a) $w \in L(G)$ which has at least two distinct derivative trees
    (b) $w \in L(G)$ which has at least two left-most derivations
    (c) $w \in L(G)$ which has at least two right-most derivations
    (d) Any one of the above

25. Consider the grammar G with the start symbol S:

    $S \rightarrow bS \mid aA \mid b$
    $A \rightarrow bA \mid aB$
    $B \rightarrow bB \mid aS \mid a$

    Which of the following is a word in L(G)?
    (a) ababba (b) bbaabb (c) babbbabaaaa (d) aabb

Exercises

1. Define leftmost and rightmost derivations. Give example.
2. Consider the grammar G. $S \rightarrow S+S \mid S*S \mid (S) \mid a$. Show that the string "a + a * a" has
    a. Parse trees
    b. Left most derivations

3. Find the unambiguous grammar G′ equivalent to G and show that L(G) = L(G′) and G′ is unambiguous.
4. Convert the following CFG to equivalent unambiguous grammar.

   $E \to E + E$
   $E \to E * E$
   $E \to a \mid b \mid (E)$
5. Check whether the grammar is ambiguous or not.
   a. $S \to 0S1 \mid SS \mid \varepsilon$ w='0011'
   b. $S \to AB \mid aaB,\ A \to a \mid AA,\ B \to b$ w = "aab"
   c. $S \to SbS \mid a$ w = "abababa"
   d. $S \to aSb \mid ab$
   e. $R \to R + R \mid RR \mid R^* \mid a \mid b \mid c$ w = a + b * c
6. Construct the reduced grammar from regular grammar given below.

   $S \to Aa \mid bs \mid \varepsilon$
   $A \to aA \mid bB \mid \varepsilon$
   $B \to aA \mid bc \mid \varepsilon$
   $C \to aC \mid bc$
7. Find a CFG, without ε productions, unit production, and useless productions equivalent to the grammar defined by

   $S \to ABaC$
   $A \to BC$
   $B \to b \mid \varepsilon$
   $C \to D \mid \varepsilon$
   $D \to d.$
8. Left factor the given grammar

   $S \to aSSbS \mid aSaSb \mid abb \mid b$
9. Left factor the given grammar

   $S \to 0SS1S \mid 0SS0S \mid 01 \mid 1$
10. Remove ε productions from the following grammar:

    $S \to XYX$
    $X \to 0X \mid \varepsilon$
    $Y \to 1Y \mid \varepsilon$
11. Eliminate useless symbols and productions from the following grammar:

    G = (V, T, P, S) where V = {S, A, B, C}, T = {a,b} and productions P given below:
    $S \to ab \mid A \mid C,\ A \to a,\ B \to ac,\ C \to aCb$
12. Remove ε productions from the following grammar:

    $S \to A0B \mid BB \mid 0C \mid CBA$
    $A \to C0 \mid 1C \mid CC \mid CBA$
    $B \to B0 \mid AB \mid \varepsilon$
    $C \to 0A \mid BBB$

13. Remove unit productions from the grammar
    $S \to 0A \mid 1B \mid C$
    $A \to 0S \mid 00$
    $B \to 1 \mid A$
    $C \to 01$
14. Remove unit productions from the grammar
    $S \to AB$
    $A \to a$
    $B \to C \mid b$
    $C \to D$
    $D \to E \mid bC$
    $E \to d \mid Ab$
15. Optimize the following grammar.
    $S \to A \mid 0C1$
    $A \to B \mid 01 \mid 10$
    $C \to CD \mid \varepsilon$
16. Optimize the following grammar.
    $S \to aAa$
    $A \to Sb \mid bCc \mid DaA$
    $C \to abb \mid DD$
    $E \to aC$
    $D \to aDA$
17. Write equivalent unambiguous for the following CFG.
    $R \to R + R \mid RR \mid R^* \mid a \mid b$
18. Write equivalent unambiguous for the following CFG.
    bExpr → bExpr and bExpr | bExpr or bExpr | not R bExpr | a | b
19. Eliminate left recursion in the grammar.
    $S \to Aa / b$
    $A \to Ac / Sd \mid \varepsilon$
20. Check whether the grammar.
    $S \to AaAb / BaBb$
    $A \to \in$
    $B \to \in$ is ambiguous or unambiguous
21. Eliminate left recursion from the following grammar:
    $A \to Bd$
    $\mid Aff$
    $\mid CB$
    $B \to Bd$
    $\mid Bf$
    $\mid df$
    $C \to h$
    $\mid g$

Key for Fill in the Blanks

1. Syntax anlyzer
2. Ø
3. not deriving any terminal
4. nullable
5. Subset
6. Subset
7. unambiguous
8. True
9. Type 2
10. $|\alpha| <= |\beta|$ & $|\alpha| = 1$.
11. Null rule
12. False
13. $S \to C_a A \mid b, A \to C_a A \mid C_a C_b$
    $C_a \to a, \quad C_b \to b$
14. $S \to (S) \mid \varepsilon$
15. Ambiguous
16. False
17. RG
18. True
19. False
20. True

Key for Objective Question Bank

1. b 2. a 3. c 4. b 5. c 6. b 7. a 8. c 9. d 10. b
11. d 12. a 13. a 14. b,c 15. a 16. c 17. b 18. a 19. a 20. d
21. d 22. a 23. b 24. d 25. a

This page is intentionally left blank.

CHAPTER 4

Syntax Analysis — Top-Down Parsers

Syntax analysis or parsing recognizes the syntactic structure of a programming language and transforms a string of tokens into a tree of tokens. Top-down parsers are simple to construct. Parsers are also used in natural language applications.

CHAPTER OUTLINE

4.1 Introduction
4.2 Error Handling in Parsing
4.3 Types of Parsers
4.4 Types of Top-Down Parsers
4.5 Predictive Parsers
4.6 Construction of Predictive Parsing Tables
4.7 LL(1) Grammar
4.8 Error Recovery in Predictive Parsing

The second phase in the process of compilation of a source program is syntax analysis. This phase comes after the lexical analysis phase. While lexical analyzer reads an input source program and produces an output—a sequence of tokens—the syntax analyzer verifies whether the tokens are properly sequenced in accordance with the grammar of the language. If not, the syntax analyzer detects the errors and produces proper sequence of error messages to the user and if possible recovers from the errors. The output of syntax analyzer is a parse tree, which is used in the subsequent phases of compilation. This process of analyzing the syntax of the language is done by a module in the compiler called **parser**. For performing syntax analysis, the grammar of the language has to be specified. Context Free Grammars (CFGs) are used to define standard syntax rules for the language. This process of verifying whether an input string matches the grammar of the language is called **parsing**. This chapter describes the process of parsing, error handling, types of parsers, and top-down parsing in detail.

4.1 Introduction

The syntax analyzer obtains a string of tokens from the lexical analyzer. It then verifies the syntax of the input string by verifying whether the input string can be derived from the grammar or not. If the input string is derivable from the grammar, then the syntax is correct; if the input string is not derivable, then the syntax is wrong. This is shown in Figure 4.1. The parser should also report syntactical errors in a manner that is easily understood by the user. It should also have procedures to recover from these errors and to continue parsing action.

The output of a parser is a parse tree for the set of tokens produced by the lexical analyzer. In practice, there are a number of tasks that are carried out during syntax analysis, such as collecting information about various tokens and storing into symbol table, performing type checking and semantic checking, and generating intermediate code. These tasks will be discussed in detail in syntax-directed translations.

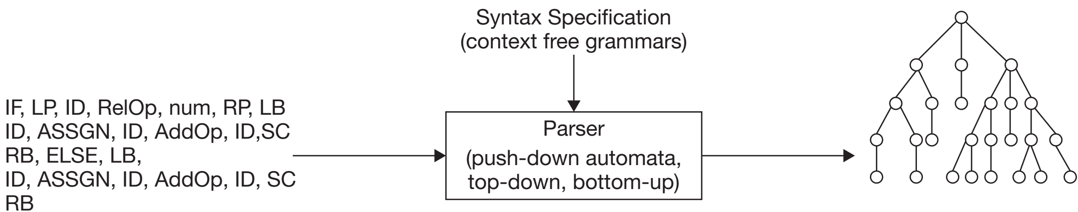


Figure 4.1 Syntax Analysis

4.2 Error Handling in Parsing

Compiler construction would be very simple if it has to process only correct programs. But most of the programmers write incorrect programs; hence, a good compiler should always assist a programmer in identifying and detecting errors. Generally, programming languages do not specify how a compiler should respond to errors. The response is left to the compiler designer.

Generally, errors in programs are detected at different levels.

- **At lexical analysis**: Unrecognized group of characters like &abc, $abc, etc., which cannot be a keyword or identifier
- **At syntax analysis**: Missing operator/operands in expression
- **At semantic analysis**: Incompatible types of operands to an operator
- **Logical errors:** Infinite loop. Detecting logical errors at compile time is a tedious task.

In the compilation process, 90% of errors are captured during the syntax and semantic analysis phase.

That's why error detection and recovery in compiler are centered on parsing. The main reason for this is that many errors are syntactic in nature or are captured when a stream

of tokens from the lexer does not match with the grammar of the programming language. Another reason is the precision of modern parsing methods. They can detect the presence of syntax errors very efficiently. In this section, we shall discuss a few techniques for syntax error recovery.

Error-Recovery Strategies
There are four different error-recovery strategies generally used by parsers.

- Panic Mode
- Phrase Level
- Error Production
- Global Correction

4.2.1 Panic Mode Error Recovery

This is a simple method used by most of the parsers. On an error, the recovery strategy used by the parser is to skip input symbols one at a time until one of the designated synchronizing tokens is found. The synchronizing tokens can be 'end','}',')', ';' whose role in the source program is well defined. The compiler designer must select the appropriate set of synchronizing tokens from the source language. For example, in "C" language, on detecting an error, it simply skips all characters till ";" since ";" is at the end of every statement.

4.2.2 Phrase Level Recovery

On detecting an error, the parser may perform some local corrections on the remaining input. It could be replacing an incorrect character with a correct one or swapping two adjacent characters such that the parser can continue with the process. The local correction is a choice left to the compiler designer. A token can be replaced, deleted, or inserted as a prefix to the input; this will enable the parser to continue with the process. We must be very careful while doing replacements. Sometimes, replacements may lead to infinite loops.

4.2.3 Error Productions

If the compiler designer has a good idea about possible errors, then he can add rules with the grammar of the programming language to handle erroneous constructs. A parser is constructed for this new grammar such that it handles errors. The error productions are used by the parser to issue appropriate error diagnostics on erroneous constructs in the input. Automatic parser generator "YACC"—yet another compiler compiler—uses this strategy.

4.2.4 Global Correction

Generally, it is preferred to have minimum changes, that is, corrections in the input string. There are algorithms to obtain a globally least cost corrections that help in choosing a minimal sequence of changes.

4.3 Types of Parsers

Parsers can broadly be classified into three types as shown in Figure 4.2.

4.3.1 Universal Parsers

Universal parsers perform parsing with any grammar. That's why they are called universal parsers. They use parsing algorithms like Cocke-, Younger-, Kasami-algorithm or Earley's algorithm. It uses the Chomsky Normal form of the CFG. This method is inefficient. So it is not used in commercial compilers.

Parsers that are commonly used in compilers are top-down parsers or bottom-up parsers.

4.3.2 Top-Down Parsers (TDP)

Top-down parsing involves generating the string starting from the start symbol repeatedly applying production rules for each nonterminal until we get a set of terminals. The output of any parser is a parse tree. A top-down parser constructs the parse tree starting with root and proceeds toward the children. Here it uses Left Most Derivation (LMD) in deriving the string. The task of a top-down parser at any time is to replace a nonterminal by the right hand side of the rule.

Consider the following example:

Example 1:

Grammar and input string

$S \rightarrow a\ A\ B\ e$
$A \rightarrow b \mid b\ c$
$B \rightarrow d$

w = "abcde"

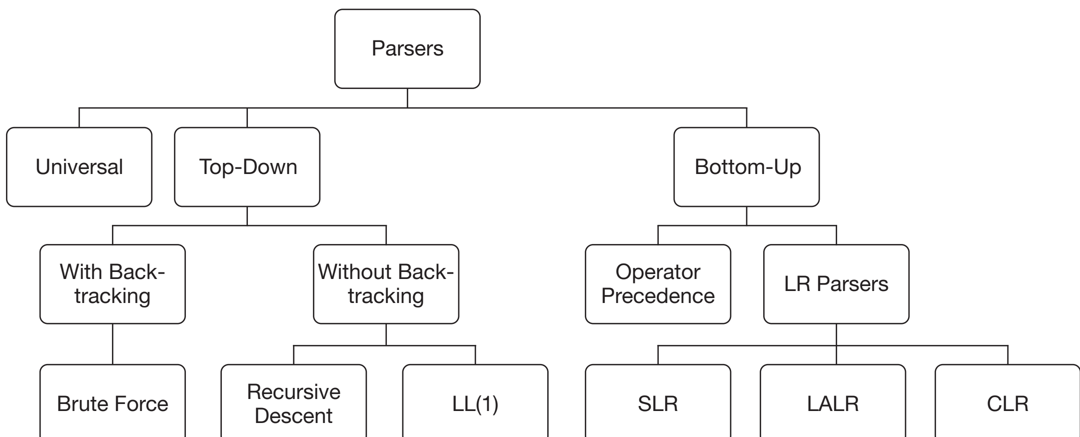


Figure 4.2 Types of Parsers

The construction of the top-down parser is shown in Figures 4.3 and 4.4. A top-down parser starts the parse tree with the start symbol, S.

Replace S by the right hand side, that is, "aABe".

Now, in the expanded string there are two nonterminals, A and B. Which one do we expand first? This is resolved by the parser as follows. A TDP uses LMD in recognizing a sentence. Hence, the parser first selects "A" for expansion. Now nonterminal A has two alternative rules. Which one do we choose fist? This is the primary problem. Here, if we expand A by first alternative, that is, "b," we cannot get the required string. The right choice here is the second alternative, that is, "bc." So a top-down parser is constructed as follows:

Though the task of the top-down parser is very simple—replacing nonterminal by right hand side of rule—the primary problem in design is if a nonterminal has more than one alternative, then there should be some mechanism to decide the right choice for expansion. How can this problem be handled by the different types of top-down parsers? This is what we will discuss in the remaining topics.

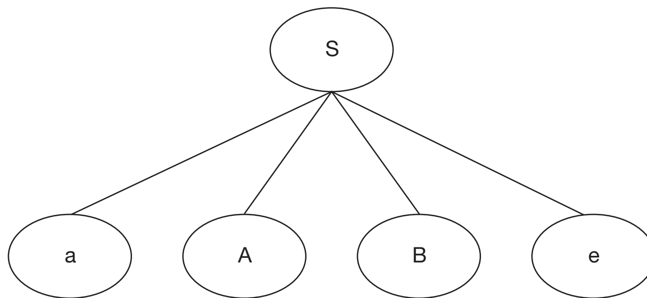


Figure 4.3 Top-Down Parser Construction

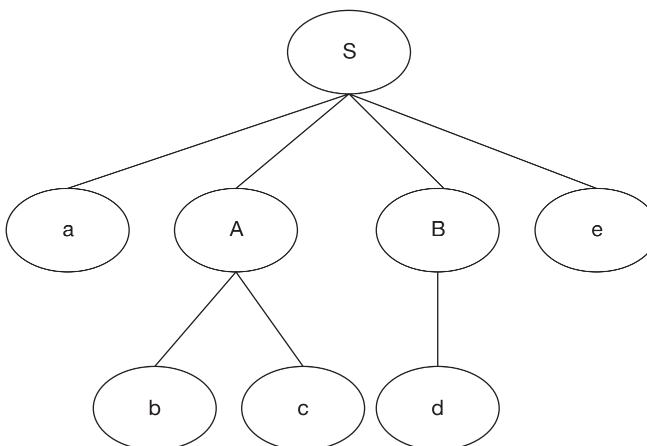


Figure 4.4 Top-Down Parser Construction

4.3.3 Bottom-Up Parsers

Bottom-up parsing is not simple like top-down parsing. Bottom-up parsing involves repeatedly reducing the input string until it ends up in the first nonerminal, that is, the start symbol of the grammar.

Let us look at the complexity of reducing the input string to start symbol. The task of a bottom up parser at any time is to identify a substring that matches with the right hand side.

Replace that string by the left hand side nonterminal.

Consider the following grammar and input string.

Example 2:

$S \rightarrow a\ A\ B\ e$
$A \rightarrow b \mid b\ c$
$B \rightarrow d$

w = "abcde"

The construction of bottom-up parser is shown in Figure 4.5. A bottom-up parser looks at reducing the string "abcde" to start the symbol S. Now let us understand the complexity of reducing the string to the start symbol. If at all the string "abcde" is reduced to "a A B e," then we can replace "aABe" with S. To reduce "abcde" to "aABe," after the first "a," "A" is required. "A" can be either "b" or "bc." Since the task of a bottom-up parser at any time is to identify a substring that matches with right hand side, replace that string by the left hand side nonterminal. Hence, we can take string "b" then replace it by "A." But if we do so, it cannot be reduced to the start symbol "S". So this is not a valid reduction. How do we determine valid reductions? This is resolved as follows.

A bottom-up parser uses the reverse of right most derivation in reducing the input string to the start symbol. So whenever a substring is identified and reduced, it should give one step along the reverse of the right most derivation. So a bottom-up parser works as follows:

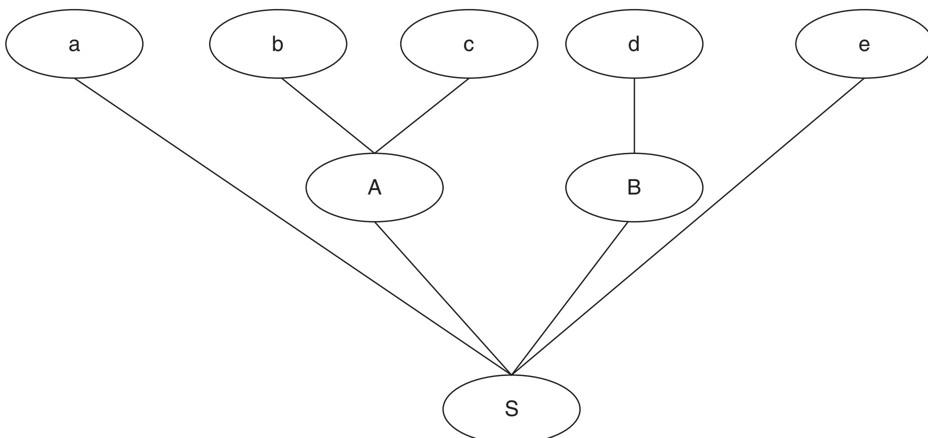


Figure 4.5 Bottom-up parser construction

So the task of a bottom-up parser at any time is to identify a substring that matches with the right hand side; if that substring is replaced by the left hand side nonterminal, it should give one step along the reverse of the right most derivation.

The main difficulty in bottom-up parsing is identifying the substring that is called handle.

Bottom-up parsing can be described as "detect the handle" and "reduce the handle." The main difficulty is detecting the handle.

While discussing different top-down parsers and bottom-up parsers, we will see how this problem is handled in the design of such parsers. Let us consider top-down parsers first.

4.4 Types of Top-Down Parsers

Top-down parsing can be viewed as an attempt to construct a parse tree in a top-down manner for the given input. The tree is created by starting from the root and creating the nodes of the parse tree in preorder. It uses leftmost derivation.

In the above section, the process of top-down parsing and bottom-up parsing, the difficulties in design are introduced. Recall that top-down parsing is characterized as a parsing method, which begins with the start symbol, attempts to produce a string of terminals that is identical to a given source string. This matching process is executed by successively applying the productions of the grammar to produce substrings from nonterminals. Here we discuss several ways of performing top-down parsing.

The type of grammar that is used in parser construction is context free grammar. Empty productions (ε - rules) are also allowed with grammar.

Top-down parsers are broadly classified into two categories based on the design methodology used—with full backtracking and without backtracking. With full backtracking we have the Brute Force Technique.

4.4.1 Brute Force Technique

Top-down parsing with full backtracking is known as Brute Force Technique. Since it is top-down parser, it attempts to create the parse tree starting with root and proceeds to children. Whenever a nonterminal has more than one alternative, it follows the procedure given below in choosing the production.

1. Whenever a nonterminal is to be expanded for the first time, always substitute with the first alternative only, that is, first time, first rule.
2. Even in the newly expanded string the same procedure is repeated, i.e., first time first alternative only for expansion.
3. This process continues until the nonterminal gets a string of terminals.
4. Once the nonterminal gets the string of terminals, it compares that with the input string; if both of them match, it announces successful completion. Otherwise, it realizes that that is not the desired string. So it backtracks and now tries with the second alternative at the next level. Once all possibilities are exhausted at the lower level, and if too do not match, then it backtracks to the next level and repeats the same procedure until all combinations are verified.

As an example of the Brute Force Technique, consider the following grammar and input string w.

Example 3:

$S \rightarrow$ **r Ad | r B**
$A \rightarrow$ **y | z**
$B \rightarrow$ **zzd | ddz**
String w is: "rddz"

Let us see how the brute force parser works. This is shown in Figure 4.6. Parser starts with the root symbol S.

The parser replaces S by the first alternative, that is, "rAd." In the newly expanded string "rAd," A is to be replaced for the first time; hence the parser replaces by the first alternative only, that is, y. Now we get a string "ryd," which is not the expected string.

So at the lower level, for A we have already tried the first rule; now it goes with the second rule, that is, z. Now we get a string "rzd," which is not the expected string.

Since all the possibilities with "A" are exhausted, the parser now backtracks and goes with the first level, that is, S. With "S" all the possibilities with the first alternative are verified; it then backtracks and goes with second alternative rule, that is, "rB." In the expanded string, "B" is to be expanded for the first time; hence, it goes with the first alternative only, that is, "zzd." It gets a string "rzzd," which is not the expected string. So it backtracks and goes with the second alternative, that is, "ddz.". Then it gets the desired string "rddz."

The main disadvantages of the Brute Force Technique are:

1. Too much of backtracking is involved; hence, such parsers are dead slow.
2. Error recovery is also difficult. Unless all the possibilities are verified, we cannot say anything about any error that could have occurred. Even if there is an error, we cannot say when exactly the error has occurred.
3. Backtracking is costly.

That is why such parsers are not preferred for practical applications. Practically used parsers are under the second category, that is, without backtracking. These parsers are even called predictive parsers.

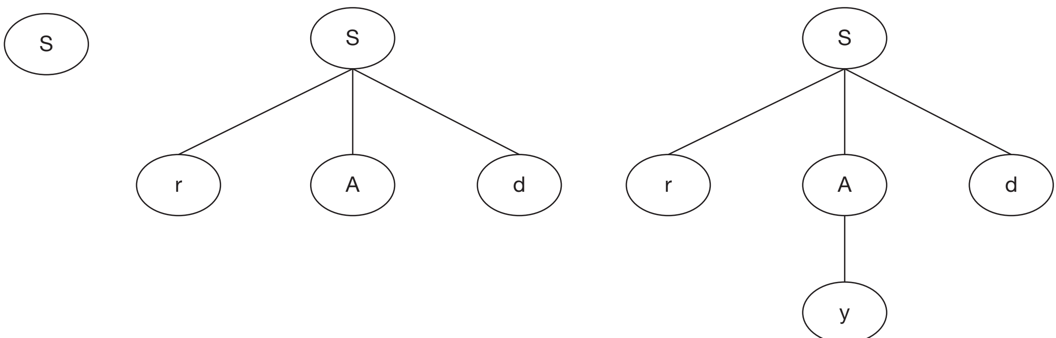


Figure 4.6(a) Brute Force Parser

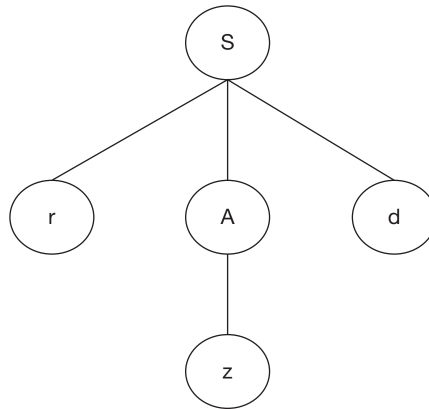


Figure 4.6(b) Brute Force Parser

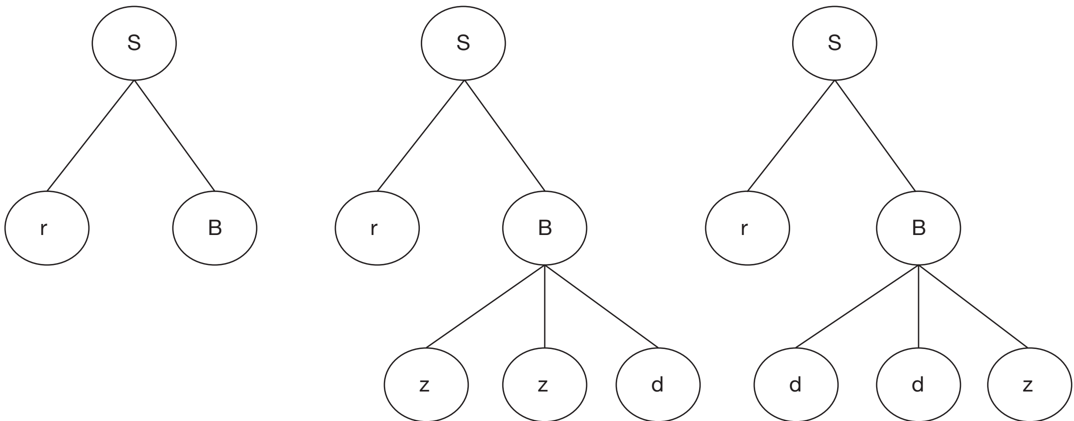


Figure 4.6(c) Brute Force Parser

4.5 Predictive Parsers

A predictive parser is capable of predicting the right choice for expanding a nonterminal. Given a grammar and input string, to design a predictive parser, there is a restriction on the grammar. The restriction is: The grammar should be free of left recursion and should be left factored. A *predictive parser* tries to predict which production produces the least chances of a backtracking and infinite looping. Predictive parsing relies on information about what first symbols can be generated from a production. If the first symbols of a production can be a nonterminal, then the nonterminal has to be expanded till we get a set of terminals.

There are two types of predictive parsers.

1. Recursive descent parser
2. Nonrecursive descent parser

The simplest is the recursive descent parser.

4.5.1 Recursive Descent Parser

The top-down parsing method given in the previous section is very general but can be very time consuming. A more efficient (but less general) method is recursive descent parsing. One of the most straightforward forms of top-down parsing is recursive descent parsing.

Recursive descent parsing is writing recursive procedures for each nonterminal. This is a top-down process in which the parser attempts to verify whether the syntax of the input string is correct as it is read from left to right. A top-down parser always expands a nonterminal by the right hand side of the rule at any time until it gets a string of terminals. The parser actually reads input characters from the input stream, symbol by symbol from left to right, and matching them with terminals from the grammar that describes the syntax of the input. Our recursive descent parsers will read one character at a time and advance the input pointer when the proper match occurs. The routine presented in the following figures accomplishes this matching and reading process.

Recursive descent parser actually performs a depth-first search of the derivation tree for the string being parsed; hence, the name "descent." It uses a collection of recursive procedures; hence, the name "recursive descent."

As our first example, consider the following simple grammar:

Example 4:

$$\mathbf{S \rightarrow a \; + A}$$
$$\mathbf{A \rightarrow (S)}$$
$$\mathbf{A \rightarrow a}$$

and the derivation tree for the expression a + (a + a) is shown in Figure 4.7.

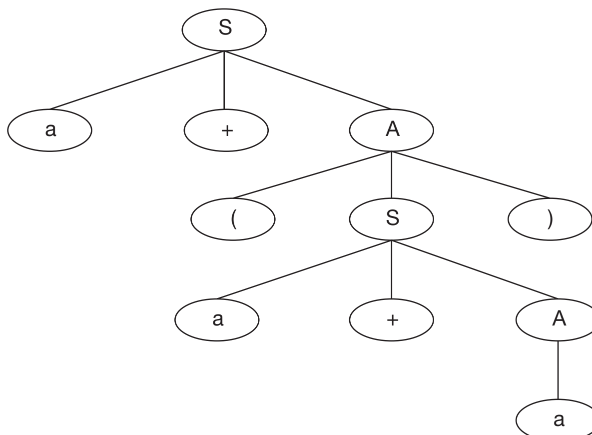


Figure 4.7 Derivation Tree for a + (a + a)

A recursive descent parser is a top-down parser and the parser works starting with the start symbol. Hence, it traverses the tree by first calling a procedure "S." This procedure "S" reads the symbol "a" and "+" and then calls a procedure "A."

Let the token to be read in the input stream be next_token. The function match() is matching the next token in the parsing with the current terminal derived in the grammar and advancing the input pointer, such that next_token points to the next token. match() is effectively a call to the lexical analyzer to get the next token. This would look like the following routine:

```
char next_token;
next_token = getchar();
int match(char terminal)
      {if next_token == terminal then next_token = getchar();
       else error();
     }
```

Note that "error" is a procedure that notifies the user that a syntax error has occurred and then possibly terminates execution.

S → a +A

For this rule, procedure for "S" is defined as follows:

```
Procedure S()
  {if next_token == 'a' {
               match('a');
               match('+');
               A();
              }
     else error();
 }
```

A → (S) | a

In order to recognize "A," the parser must decide which of the productions to execute. This is not difficult and is done based on the next input symbol to be recognized. The procedure can be defined as follows:

```
Procedure A()
{if next_token == '(' {
              match('(');
              S();
              match(')');
               }
 else
 if next == 'a' {
          match('a');
           }
 else error();
}
```

In the above routine, the parser must decide whether "A" can be (S) or a. If it does not match these two then the error routine is called; otherwise, the respective symbols are recognized.

Here both the procedures S() and A() are nonrecursive as the rules defined for that nonterminal are nonrecursive. If the rules are recursive, procedures also become recursive.

Generally, the grammar is a set of recursive rules. Only because of recursion, with finite rules, we define infinite sentences. That's why we say "recursive descent parsing is writing recursive procedures for each nonterminal." So, all one needs to write a recursive descent parser is a nice grammar that is free of left recursion and is left factored.

Note that given a grammar, we can write a recursive descent parser without left factoring the grammar. Then that becomes a recursive descent parser without backtracking. So a recursive descent parser can be with backtracking or without backtracking. It depends on the grammar. Remember though it is simple, this is also of restricted use. If the language is simple, that is, if the grammar is simple, we can choose recursive descent parsing.

Out of all top-down parsers the most widely used parser is the nonrecursive descent parser, which is also called the LL(1) parser. This does not use any procedures but uses a table and a stack.

4.5.2 Nonrecursive Descent Parser—LL(1) Parser

It is possible to build a nonrecursive descent parser, also called LL (1) parser, by using a stack explicitly, rather than implicitly via recursive calls. The key problem in the design of predictive parser is that of determining the right production to be applied for a nonterminal. The nonrecursive parser looks up the production to be applied in parsing table. We shall see how the table can be constructed directly from the grammars. This is shown in Figure 4.8.

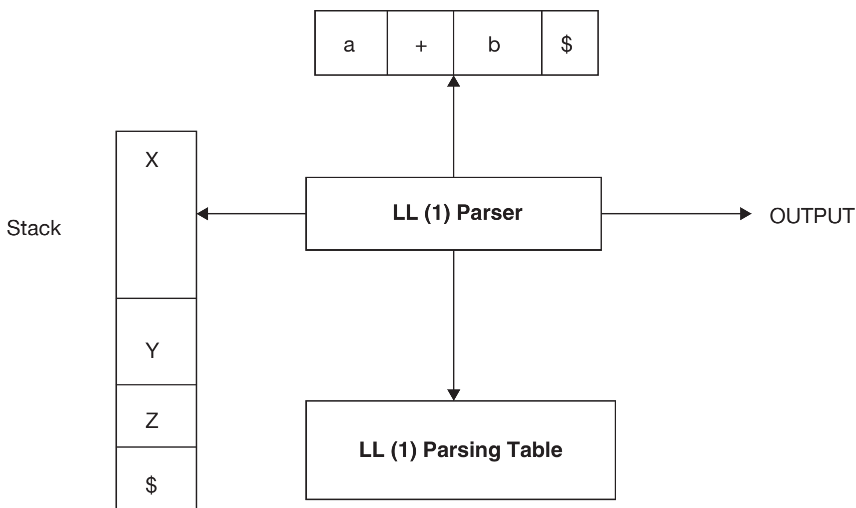


Figure 4.8 Nonrecursive Descent Parser/LL(1) Parser

A table-driven nonrecursive predictive parser uses an input buffer, a stack, a parsing table, and an output stream. The string to be parsed is read into the input buffer and "$" is appended at end. "$" is a symbol used as a right-end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols at any time. Initially "$" is pushed on to the stack, indicating the bottom of the stack. A top-down parser starts working by pushing the start symbol of the grammar on to the stack. The parsing table is a two-dimensional array M [S,a] where "S" is a nonterminal and "a" is a terminal or the input right end marker "$."

The parser works as follows: Suppose if "X" is the symbol on the top of the stack and "a," the current input symbol. The current symbol pointer by input pointer at any time is called the look-ahead symbol. Parser always works by comparing the top of the stack 'X' with the look-ahead symbol "a." These two symbols determine the action to be performed by the parser.
Depending on the top of the stack (grammar symbol) there are three possibilities.

1. If X = = a = = $, that is, when the entire input string is read and by the time the stack becomes empty, the parser halts and announces successful completion of parsing.
2. If ((X = = a) ! = $), that is, if the top of the stack is a terminal matching with the look ahead symbol but it is not end of input, then the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the parser consults parsing table entry M[X,a]. This parsing table entry will be either a production of the grammar or blank entry. If, for example, M[X,a] = {X → UVW}, the parser replaces nonterminal X on top of the stack by WVU (with U on the top). If entry is blank then it is an error.

As output, we shall assume that the parser just prints the production used.

So the parser repeatedly uses step 2 or 3 until it reaches step 1. Coming to the error detection, in step 2, if the top of the stack is a terminal but does not match with the look ahead, then it is an error.

Similarly in step 3, when the top of the stack is a nonterminal, the parser refers to the parsing table for replacing X. If the respective entry in table is a blank entry, it announces error. Remember blank entries in parsing table indicate errors. If M[X,a] = error, the parser calls an error recovery routine.

The behavior of the parser can be well understood if we consider input with stack configurations that give the stack contents. Let us understand the LL(1) parsing algorithm with an example.

4.5.3 Algorithm for LL(1) Parsing

Input: An input string "w" and a parsing table M for grammar G.

Output: If w is in L(G), parse tree for string w; otherwise, an error.

Method: Initially, the parser has $S on the stack with start symbol "S" of Grammar "G" on top, and w$ in the input buffer.

The program that uses the predictive parsing table M to parse the given input is as follows:

```
set the input pointer "iptr" to point to the first look ahead symbol
of w$.
repeat
   let A be the top stack symbol and a the symbol pointed to by iptr.
   if A is a terminal of $ then
        if A == a then
           pop X from the stack and advance iptr
        else error()
     else
        if M[A,a] = A → Y1Y2...Yk then begin
           pop A from the stack;
           push Yk,Yk-1...Y1 onto the stack, with Y1 on top;
           output the production A → Y1Y2...Yk
        end
        else error()
until A = $
```

Example 5:

Given a grammar and an input string,

E → E + T | T
T → T * F | F
F → (E) | id and string w = 'id+id*id'

Let us understand how an LL(1) parser parses the input using the grammar.

Solution: Given a grammar, to construct LL(1) parser, there is a restriction on grammar. The grammar should be free of left recursion and should be left factored. So if we eliminate left recursion resulting grammar is as follows:

E → TE′
E′ → +TE′ | e
T → FT′
T′ → *FT′ | e
F → (E) | id

Use this grammar and construct the LL(1) parsing table. As we have not discussed the procedure of LL(1) table construction, let us take the table that is required for understanding the parsing algorithm. Later we will discuss how to construct such a table.

LL(1) Table for the above grammar is:

Non-Terminal	(	id	+	*	)	$
E	E → T E′	E → T E′				
E′			E′ → +T E′		E′ → ε	E′ → ε
T	T → F T′	T → F T′				
T′			T′ → ε	T′ → *F T′	T′ → ε	T′ → ε
F	**F → (E)**	**F → id**				

With input string **'i d + id * id'**, the LL(1) parser makes a sequence of moves shown in Figure 4.9. Initially, the input pointer points to the leftmost symbol.

The output of a parser is a parse tree. The last step in parsing algorithm, that is, step 4 helps in constructing the parse tree. The order in which different nonterminals are used in parsing is listed in the above table. If we follow that order, that gives us a parse tree.

Given a grammar G to construct nonrecursive predictive parser, that is, LL(1), we use two functions, **First and Follow.** These functions allow us to fill in the entries of a predictive parsing table for G whenever possible. During panic-mode error recovery, set of tokens given by the "Follow" set can be used as synchronizing tokens. Let us understand these two functions now.

4.5.4 First(α), Where α Is Any String of Grammar Symbols

First(α) gives the set of terminals that begin the strings derived from α. If $\alpha \overset{*}{\Rightarrow} \varepsilon$ or $\alpha \rightarrow \varepsilon$ then ε is also in First(α).

To compute First(α) for all grammar symbols α, apply the following rules until no more terminals or e can be added to the First set. 'α' is a grammar symbol. Hence, two possibilities exist. "α" can be a terminal or nonterminal.

1. If α is a terminal, then First(α) is {α}.

   Example:

   First(+) = {+}, First(*) = {*}, First(id) = {id}

Stack	Input	Output
$E	id+id*id $	
$E′ T	id+id*id $	E → TE′
$E′ T′ F	id+id*id $	T → FT′
$E′ T′ id	id+id*id $	F → id
$E′ T′	+id*id $	T′ → ε
$E′	+id*id $	E′ → +TE′
$E′ T +	+id*id $	
$E′ T	id*id $	T → FT′
$E′ T′ F	id*id $	F → id
$E′ T′ id	id*id $	
$E′ T′	*id $	T′ → *FT′
$E′ T′ F*	*id $	
$E′ T′ F	id $	F → id
$E′ T′ id	id $	
$E′ T′	$	T′ → ε
$E′	$	E′ → ε
$	$	

Figure 4.9 Sequence of Moves by the LL(1) Parser while Parsing "id + id * id"

2. If "α" is a nonterminal, two possibilities exist: α may be defined with null production or non null production.

   If "α" is a nonterminal

   a. & is defined with the null rule, that is, $\alpha \to \varepsilon$ is a production, then add 'ε' to First(α).

   b. If α is a nonterminal and is defined with non null production like $X \to Y_1Y_2...Y_k$ is a production, then

      $\text{First}(\mathbf{X}) = \text{First}(Y_1Y_2...Y_k) = \text{First}(Y_1)$ if $Y_1 \Rightarrow \varepsilon$ else

      $\text{First}(\mathbf{X}) = \text{First}(Y_1) \ U \ \text{First}(Y_2...Y_k)$ if $Y_1 \Rightarrow \varepsilon$

Consider the example

Example 6:

$S \to a\,A\,b$
$A \to cd \mid ef$
Find First(S) and First(A).

Solution: First(S) = First(aAd) = First(a) = {a}
First(A) = First(cd) ∪ First(ef) =
First(c) ∪ First(e) = {c,e}

Example 7:

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$
Find first of each nonterminal.

Solution: Apply the first function for each nonterminal on the L.H.S. of rule.
First(E) = First(T) = First(F) = First('(') ∪ First(id) = {(,id}
First(E′) = First(+) ∪ First(ε) = {+, ε}
First(T) = First(E) = {(, id}
First(T′) = First(*) ∪ First(ε) = {*, ε}
First(F) = First(E) = {(, id}

Hence, first of each nonterminal is
First(E) = {(, id}
First(E′) = {+, ε}
First(T) = {(, id}
First(T′) = {*, ε}
First(F) = {(, id}

Example 8:

$S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \varepsilon$
$C \to h \mid \varepsilon$

Find first of each nonterminal.

Solution:

First(S) = First(ACB) ∪ First(CbB) ∪ First(Ba)
First(ACB) = First(A) = {d} ∪ First(BC)
= {d} ∪ First(B) = {d} ∪{g}∪First(C) ……as first(B) contains ε
= {d,g,h, ε} ….here we add ε as First(A) is ε because of 'BC'&First(C)&First(B) also contains ε. Hence add ε in First(ACB) at end.
First(CbB) = First(C) = First(C) is {h, ε}.so add only 'h' and apply First on next symbol 'b' i.e First(b) = {b}. Hence
First(CbB) = {h,b}

Similarly,

First(Ba) = First(B) which is {b,ε}. Because of ε continues with a .Hence
First(Ba) = {g,a}
So First(S) = {d,g,h,ε,b,a,}
First(A) = {d,g,h,ε}
First(B) = {g,ε}
First(C) = {h,ε}

	First()
S	{a,b,d,g,h,ε}
A	{d,g,h,ε}
B	{g,ε}
C	{h,ε}

4.5.5 Follow(A) Where 'A' is a Nonterminal

Follow (A), where A is a nonterminal, gives a set of terminals as output. This output set of terminals may appear immediately to the right of A in some sentential form. If A is the right-most symbol in some sentential form, then $ is in Follow (A).

To compute the Follow(A) for all nonterminals A, apply the following rules until nothing can be added to any Follow set.

1. If "A" is the start symbol and $ is the input right end marker, then place $ in Follow (A).
2. If there is a production S → a A β where β is the string of grammar symbols, then First(β) except ε is placed in Follow(A).
3. If there is production S → aA or a production A → aAβ where First (β) contains ε, then everything in Follow(S) is in Follow(A).

To find the first of nonterminal, we consider the rule that is defined for nonterminal. But for finding the follow of nonterminal, search for nonterminal on the right hand side of a rule.

If any rule contains that nonterminal, use that rule for evaluating Follow ().

Rule 2 says if there are some grammar symbols next to nonterminal A, then first of that string of grammar symbols is Follow (A).

If follow of string of grammar symbols ends with terminal set, evaluation ends here. If this first set contains an ε, then rule 2 as well as rule 3 must be applied.

Rule 3 says if there are no symbols next to nonterminal A, that is, if "A" happens to be rightmost on the right hand side, then the follow of the left hand side nonterminal, that is, "S" is follow(A).

Consider the following example to understand First and Follow functions.

Example 9:

Find the first and follow of all nonterminals in the Grammar-

E → TE′
E′ → +TE′ | ε
T → FT′
T′ → *FT′ | ε
F → (E) | id

Solution:

First(E) = {(, id}
First(E′) = {+, ε}
First(T) = {(, id}
First(T′) = {*, ε}
First(F) = {(, id}

Follow(E) = {$} by rule 1
= {)} by rule 2 on production F → (E) | id
= {$,)}

Follow(E′) = Follow(E) by rule 3 on production E → T E′
= {$,)}

Follow(T) = First(E′) by rule 2
= {+, ε}
= Follow(E) by rule 3 on production E → T E′
= {$,)}
= {+,), $}

Follow(T′) = Follow(T) by rule 3 on production T → F T′
= {+,), $}

Follow(F) = First(T′) by rule 2
= {*}
= Follow(T) by rule 3 on production T → F T′
= {+,), $}
= Follow(T′) by rule 3 on production T′ → F T′
= {+,), $}
= {+,), *, $}

Follow(E) = Follow(E′) = {),$}
Follow(T) = Follow(T′) = {+,),$}
Follow(F) = {+,*,),$}

Follow()

E	{),$}
E′	{),$}
T	{+,),$}
T′	{+,),$}
F	{+,*,),$}

For example, by rule 3 First(E) = First(T) and First(T) = First(F); But First(F) is given by { id, (} because First(id) = (id) and First('(') = {(} by rule 1.

To compute Follow, we add $ in Follow (E) by rule 1 as E is start symbol.By rule 2 applied to production F → (E), right parenthesis is also in Follow (E). By rule 3 applied to production E → TE′, $ and right parenthesis are in Follow (E′).

Example 10:

S → aABb
A → c | ε
B → d | ε

Find the Follow of each nonterminal.

Solution:

Follow(S) = {$} by rule 1.
Follow(A) = First(Bb) by rule 2
= {d, b} as First(B) contains ε continues with b in 'Bb'
= {d,b}
Follow(B) = {b} by rule 2

Follow()

S	{$}
A	{d,b}
B	{b}

Example 11:

S → aBD h
B → c C
C → b C | ε
D → EF
E → g | ε
F → f | ε
Find Follow of each nonterminal.

Solution:

Follow(S) = {$} by rule 1.
Follow(B) = First(Dh)
First(D) = First(EF) since D → EF
First(E) = {g,ε}
since First(E) contains ε so continue with F for First(EF).

First(EF) = {g,f,ε} since this set contains ε
First(Dh) = {g,f,h}

Follow(C) = Follow(B) = {g,f,h}
Follow(D) = First(h) = {h}
Follow(E) = First(F) = {f, ε} since there is ε, apply rule 3
= {f} ∪ Follow(D) = {f,h}
Follow(F) = Follow(D) by rule 3
= {h}
Hence

Follow()

S	{$}
B	{g,f,h}
C	{g,f,h}
D	{h}
E	{f,h}
F	{h}

Now let us see how to make use of these two functions in constructing the parsing table.

4.6 Construction of Predictive Parsing Tables

Given a grammar to construct LL(1) parsing table, use the following procedure:

The rows in the table are given by nonterminals and columns in the table are given by terminals. For each nonterminal "A" there is a row "A" defined in the parsing table. So the number of rows is equal to the number of nonterminals. Coming to the columns, for each terminals, there a column defined in the parsing table and one extra column for $, which is the input right-end marker. So the number of columns is equal to the number of terminals +1. Construction of table mainly deals with determining the position of each rule in the table. For each rule S → a, place the rule in the row given by the left hand side nonterminal that is, S. So here without applying any procedure, we can determine in which row a rule is to be placed. The only information that needs to be determined is column information. For getting that information, we use first and follow function as follows.

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is:

Input: Grammar G

Output: Parsing table M

Method

1. For each production S → a of the grammar, perform steps 2 and 3.
2. For each terminal a in First (a), add S → a, to M[S,a].

3. If 'ε' is in First (a), add S → a to M[S,b] for each terminal b in Follow(S). If 'ε' is in First (a) and $ is in Follow(S), add S → a to M[S,$].

The above algorithm can be applied to any grammar G to produce a parsing table M. If G is left recursive or ambiguous, then there will be at least one multiply-defined entry in the parsing table M. All undefined entries in symbol table are error entries.

4.7 LL(1) Grammar

When a predictive parsing table is constructed, if it doesn't contain multiple entries, then such grammars are said to be LL(1) grammars. Any grammar that is LL(1), is an unambiguous grammar and does not have left recursion. Suppose the grammar has multiple-defined entries; then to convert it to LL(1), first eliminate left recursion and left factor it. This may not convert all CFGs to LL(1) as there are some grammars that will not give an LL(1) grammar after any kind of alteration. In general, there is no universal rule to convert multiple-defined entries into single-valued entries without affecting the language recognized by the parser.

Example 12:

Construct LL(1) parsing table for the following grammar.
S → a | (T)
T → T, S | S

Solution: First ensure that the grammar is free of left recursion and left factored. As there is left recursion, first eliminate left recursion.

S → a | (T)
T → S T′
T′ → , S T′ | ε

Find First and Follow sets

First(S) = {a,(}
First (T) = First(S) = {a, (}
First (T′) = {, , ε}
Follow(S) = {$}
Follow (T) = {)}
Follow (T′) = {)}

The LL(1) table is given by

Place S → a in row S and column first(a) = a
Place S → (T) in row S and column first((T)) = (
Place T → S T′ in row T and column first**(S) =** {a,(}
Place T′ → ,S T′ in row T′ and column first (,S T′**) = {,}**
Place T′ → ε in row T′ and column Follow(T′) = {)}

	A	,	(	)	$
S	S → a		S → (T)		
T	T → S T′		T → S T′		
T′		T′ → ,S T′		T′ → ε	

Example 13:

Consider the grammar

$S \rightarrow$ **a** $B\ C$ **d** | **d** $C\ B$ **e**
$B \rightarrow$ **b** B | ε
$C \rightarrow$ **c a** | **a c** | ε

a. Compute the First sets and Follow sets for each of the nonterminals in the grammar.
b. Construct a recursive decent parser in C for the grammar. You may assume that functions **match**() and **error**() are already available and global variable **token** is used to store the current token read by function match().
c. Construct an LL(1) parsing table for the grammar.

Solution: a. FIRST(S) = {**a**, **d**}, FIRST(B) = {**b**, **ε**}, FIRST(C) = {**a**, **c**, **ε**}. FOLLOW(S) = {$}, FOLLOW($B$) = {**a**, **c**, **d**, **e**}, FOLLOW(C) = {**b**, **d**, **e**}.

b.

```
void S() {
      switch(token) {
            case a: match(a); B(); C(); match(d); break;
             case d: match(d); C(); B(); match(e); break;
      default: error();
        }
      }
       void B() {
            switch(token) {
            case b: match(b); B(); break;
            case a: case c: case d: case e: break;
            default: error();
            }
      }
      void C() {
            switch(token) {
                  case c: match(c); match(a); break;
                  case a: match(a); match(c); break;
                  case b: case d: case e: break;
                    default: error();
}
}
```

c. The parsing table is as follows:

	a	**b**	**c**	**d**	**e**	$
S	$S \rightarrow \mathbf{a}\ B\ C\mathbf{d}$			$S \rightarrow \mathbf{d}\ C\ B\mathbf{e}$		
B	$B \rightarrow \varepsilon$	$B \rightarrow \mathbf{b}\ B$	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	
C	$C \rightarrow \mathbf{a\ c}$	$C \rightarrow \varepsilon$	$C \rightarrow \mathbf{ca}$	$C \rightarrow \varepsilon$	$C \rightarrow \varepsilon$	

Example 14:

Construct LL(1) parsing table.

E → TE′
E′ → +TE′ | ε
T → FT′
T′ → *F T′ | ε
F → (E) | id

Solution: We have discussed first and follow sets for this grammar in Sections 4.5.4 and 4.5.5.

	First()	Follow()
E	{(, id}	{),$}
E′	{+, ε}	{),$}
T	{(, id}	{+,),$}
T′	{*, ε}	{+,),$}
F	{(, id}	{+,*,),$}

To construct LL(1) table, there has to be five rows since there are five nonterminals.

In the row E, place E → TE′ in the column given by First(TE′) = {(, id}
In the row E′, place E′ → +TE′ in the column given by First(+TE′) = {+}
In the row E′, place E′ → ε in the column given by Follow(E′) = {),$}
In the row T, place T → FT′ in the column given by First (FT′) = {(, id}
In the row T′, place T′ → *F T′ in the column given by First (*F T′) = {*}
In the row T′, place T′ → ε in the column given by Follow(T′) = {+,),$}
In the row F, place F → (E) in the column given by First ((E)) = {(}
In the row F, place F → id in the column given by First (id) = {id}
LL(1) Table for the above grammar is:

Non-Terminal	**(**	**id**	**+**	*****	**)**	**$**
E	E → T E′	E → T E′				
E′			E′ → +T E′		E′ → ε	E′ → ε
T	T → F T′	T → F T′				
T′			T′ → ε	T′ → *F T′	T′ → ε	T′ → ε
F	**F → (E)**	**F → id**				

Example 15:

Check whether the following grammar is LL(1) or not.

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Solution: First construct the LL(1) table, then check for multiple entries.

To construct LL(1) table, there has to be only one row since there is only one nonterminal.

In the row S, place $S \rightarrow aSbS$ in the column given by First(aSbS) = {a}

In the row S, place $S \rightarrow bSaS$ in the column given by First (bSaS) = {b}

In the row S, place $S \rightarrow \varepsilon$ in the column given by Follow(S) = {a,b,$}

So table is

	a	b	$
S	$S \rightarrow aSbS$ $S \rightarrow \varepsilon$	$S \rightarrow bSaS$ $S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

As there are multiple entries in table at a and b, the grammar is not LL(1). In fact the grammar is ambiguous grammar; hence, is not LL(1).

Example 16:

Check whether the following grammar is LL(1) or not.

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Solution: First construct the LL (1) table, then check for multiple entries.

To construct LL (1) table, there are three rows as there are three nonterminals.

In the row S, place $S \rightarrow AaAb$ in the column given by First(AaAb) = {a} and place $S \rightarrow BbBa$ in column given by First (BbBa) = {b}.

In the row A, place $A \rightarrow \varepsilon$ in the column given by Follow (A) = {a,b}

In the row B, place $B \rightarrow \varepsilon$ in the column given by Follow (B) = {b,a}

So the table is

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

As there are no multiple entries in table, the grammar is LL(1).

This method is an efficient one and eliminates backtracking but it has some limitations like

- Framing an unambiguous grammar for all possible constructs in the source language.
- Left recursion elimination and left factoring are simple to apply but they make the resulting grammar complex.

It is always better to use a predictive parser for control structures and use operator precedence for expression. However, if an LR parser generator is available, it is better choice as it would provide the benefits of predictive parsing and operator precedence automatically.

An LL(1) grammar indicates that grammar is suitable for LL(1) parser construction, that is, if LL(1) table is constructed there will not be any multiple entries. So there is no confusion for the parser. But given a grammar to check whether the grammar is LL(1) or not do we have to construct the complete table?

No not required. Without constructing the table also we can check the possibility of multiple entries as follows.

If every nonterminal in grammar is defined with one rule then there is no problem of multiple entries because the rule is one, in worst case placed under every column. But the possibility of multiple entry exists if each nonterminal is defined with more than one rule like $A \rightarrow a_1 \mid a_2$ as both rules are to be placed in the same row A. So this can be stated as a rule as follows.

Rule 1:

A grammar without ε rules is LL(1) if for each production of the form $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \alpha_4 \ldots\ldots\ldots\ldots \mid \alpha_n$, First($\alpha_1$), First($\alpha_2$), First($\alpha_3$), First($\alpha_4$),..are pair-wise mutually disjoint.

i.e. $\text{First}(\alpha_i) \cap \text{First}(\alpha_k) = \varnothing$ for $i \neq k$

Rule 2:

A grammar with ε rules is LL(1) if for each production of the form $A \rightarrow \alpha \mid \varepsilon$, First(α) and Follow(A) must be mutually disjoint.

i.e. $\text{First}(\alpha) \cap \text{Follow}(A) = \varnothing$

Let us look at why left recursion is not preferred by LL(1) parsers.

For example, let us check $A \rightarrow A\alpha \mid \beta$ is LL(1). Assume α, β to be terminals.

If it is LL (1), condition to be satisfied is

$\text{First}(A\alpha) \cap \text{First}(\beta) = \varnothing$

But First (Aα) contains β. So this condition can never be true.

Grammar is not LL(1).This can be generalized.

Rule 3:

Left recursive grammar cannot be LL(1).

Example 17:

Check whether the following grammar is LL(1) or not.

$S \rightarrow A \mid a, A \rightarrow a$

Solution: If this is LL(1), condition to be satisfied is, $\text{First}(A) \cap \text{First}(a) = \varnothing$.

But First(A) = {a} so condition is not satisfied. In fact, the given grammar is ambiguous grammar. For string "a," we get two parse trees. If grammar is ambiguous, condition will not be satisfied.

Rule 4:

Ambiguous grammar cannot be LL(1).

Example 18:

Check whether the following grammar is LL(1) or not.

$S \rightarrow A\#;\ A \rightarrow Bb \mid Cd;\ B \rightarrow aB \mid \varepsilon;\ C \rightarrow cC \mid \varepsilon$

Solution:

As "S" is defined with one rule, there is no danger of multiple entries in row S.

For row A, First(Bb) $\cap$ First(Cd) = Ø

{a,b} $\cap$ {c,d} = Ø Condition satisfied so no problem.

For row B, First(aB) $\cap$ Follow(B) = Ø

{a} $\cap$ {b} = Ø Condition satisfied so no problem.

For row C, First(cC) $\cap$ Follow(C) = Ø

{c} $\cap$ {d} = Ø Condition satisfied so no problem.

Since condition is satisfied in each row, there are no multiple entries. Hence, grammar is LL(1).

Example 19:

Check whether the following grammar is LL(1) or not.

$S \rightarrow aSA \mid \in$

$A \rightarrow c \mid \in$

Solution: For row S condition to be satisfied is First(aSA) $\cap$ Follow(S) = Ø

i.e. {a} $\cap$ {$,First(A)} = Ø

{a} $\cap$ {$,c} = Ø … satisfied.

For row A condition to be satisfied is First(c) $\cap$ Follow(A) = Ø

i.e. {c} $\cap$ {Follow(S)} = Ø

{c} $\cap$ {$,c} ≠ Ø … is not satisfied.

A grammar in which every alternative production for a rule starts with a different terminal is called simple grammar or S-Grammar. S-Grammar is suitable for LL(1).

For example,

$S \rightarrow aA \mid bB \mid cC\ \ A \rightarrow dD \mid eE\ \ B \rightarrow fF \mid gG\ C \rightarrow hH \mid kK$ is a S-grammar.

4.8 Error Recovery in Predictive Parsing

An error is detected during predictive parsing in two ways. One is when the terminal on top of the stack does not match with the next input symbol. Second is when nonterminal A is on top of the stack, "a" is the next input symbol, and the parsing table entry M[A,a] is empty. There are various techniques that can be applied when such an error occurs. One simple method that is discussed is panic mode error recovery.

Panic Mode Error Recovery

In this technique, either skip symbols on the input or ignore the top symbol from the stack until a synchronizing set of tokens appear on both input and stack top. The set should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. The following are some heuristics that can be applied.

1. We can place all the symbols in Follow(A) into the synchronizing set for nonterminal A. If an error occurs, then skip the tokens until an element of Follow(A) is seen; then pop the top element from the stack to continue the parsing.

2. Sometimes it may not be sufficient to add the Follow(A) as the synchronizing set for A. For example, if ";" terminate statements, as in C, then keywords that begin statements may not appear in the Follow set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being discarded. In general, any programming language specifies the hierarchical structure on constructs; where expressions appear within statements, statements appear within blocks and so on. We can add the symbols that begin the higher constructs to the synchronizing set of a lower construct; Where expressions may occur within statements, statements may occur within blocks and so on. We can add to the synchronizing set of a lower construct symbols that begin the higher constructs.
3. Parsing can be resumed if symbols of First(A) are added to synchronizing set for A based on A is a symbol in First(A) appears in the input.
4. If an empty string can be derived from a nonterminal, then use this as the default option. By using this option, the error may be missed or may be postponed to some other point. This approach reduces the number of nonterminals that have to be considered during error recovery.
5. If the terminal on top of the stack cannot be matched, a simple solution is to pop the terminal and issue a message saying extra terminal and continue parsing.

Let us take an example.

Example 20:

Consider error recovery with input string "+id*+id$".
To parse this we need LL(1) table for arithmetic expressions given by grammar

E → TE′
E′ → +TE′ | e
T → FT′
T′ → *FT′ | e
F → (E) | id

Solution:
Take the LL(1) table constructed in Example 13.
Let us use the above heuristics in a simple way with parsing algorithm as follows:

1. Add "synch" (as synchronizing set of symbol) to Follow (E) except for those whose first(E) has ε.
2. When parser looks up entry in table M, if entry is blank then input symbol is skipped.
3. If entry is synch, then the terminal on top of the stack is popped.
4. If terminal on top of the stack does not match, pop the stack.

Non-Terminal	(	Id	+	*	)	$
E	E → T E′	E → T E′	e1	e1	synch	synch
E′			E′ → +T E′		E′ → ε	E′ → ε
T	T → F T′	T → F T′	synch		synch	synch
T′			T′ → ε	T′ → *F T′	T′ → ε	T′ → ε
F	**F → (E)**	**F → id**	synch	synch	synch	synch

Stack	Input	Output	Action taken
$E	**+id*+id** $	E → TE′	e1- skip input, issue error msg
$E	**id*+id** $	T → FT′	push TE′
$ E′ T	**id*+id** $	F → id	push FT′
$E′ T′ F	**id*+id** $	T′ → *FT′	push id
$E′ T′ id	***+id** $	T′ → ε	pop id
$E′ T′	***+id** $	E′ → +T E′	push *FT′
$E′ T′ F*	***+id** $	T → FT′	pop *
$E′ T′ F	**+id** $	F → id	pop non terminal F
$E′ T′	**+id** $	T′ → ε	erase T′
$E′	**+id** $	E′ → ε	Push +T E′
$E′ T +	+ **id** $		Pop +
$E′ T	**id** $		push FT′
$E′ T′ F	**id** $		push id
$E′ T′ id	id $		pop id
$E′ T′	$		Erase T′
$E′	$		Erase E′
$	$		Successful Completion

Figure 4.10 Sequence of moves by LL(1) Parser while parsing "**+ id * + id**"

Synch is added in Follow (E), Follow(T) and Follow(F) as first of this nonterminals do not have ε.

With input string "**+ id * + id**," LL(1) parser makes sequence of moves shown in Figure 4.10. Initially input pointer points to the leftmost symbol.

Solved Problems

1. Find first () and follow () for each terminal.

   S → ABCDE; A → a | ε; B → b | ε: C → c | ε; D → d | ε; E → e | ε

   Solution: First(S) = First(ABCDE) = First(A) = {a,ε}
   Since first(A) has ε, First(ABCDE) = {a} ∪ First(BCDE)
   First(BCDE) = First(B) = {b,ε}
   Since first(B) has ε, First(ABCDE) = {a,b} ∪ First(CDE)
   First(CDE) = First(C) = {c,ε}
   Since first(C) has ε, First(ABCDE) = {a,b,c} ∪ First(DE)
   First(DE) = First(D) = {d,ε}
   Since first(D) has ε, First(ABCDE) = {a,b,c,d} ∪ First(E)
   First(E) = {e,ε}
   Since first(E) has ε, First(ABCDE) = {a,b,c,d,e,ε}

So First()

	First()
S	{a,b,c,d,e,ε}
A	{a,ε}
B	{b,ε}
C	{c,ε}
D	{d,ε}
E	{e,ε}

Follow(S) = {$} by rule 1.
Follow(A) = First(BCDE) by rule 2
= {b,c,d,e,ε} .
As First set contains ε, apply rule 3 also. Add all symbols except ε to Follow(A).
So Follow(A) = {b,c,d,e} ∪ Follow(S) = {b,c,d,e,$}

Follow(B) = First(CDE) by rule 2
= {c,d,e,ε} .
As First set contains ε, apply rule 3 also. Add all symbols except ε to Follow(B).
So Follow(B) = {c,d,e} ∪ Follow(S) = {c,d,e,$}

Follow(C) = First(DE) by rule 2
= {d,e,ε} .
As First set contains ε, apply rule 3 also. Add all symbols except ε to Follow(C).
So Follow(C) = {d,e} ∪ Follow(S) = {d,e,$}

Follow(D) = First(E) by rule 2
= {e,ε} .
As First set contains ε, apply rule 3 also. Add all symbols except ε to Follow(D).
So Follow(D) = {e} ∪ Follow(S) = {e,$}

Follow(E) = Follow(S) = {$]

So	Follow()
S	{$}
A	{b,c,d,e,$}
B	{c,d,e,$}
C	{d,e,$}
D	{e,$}
E	{$}

*2. Compute First and Follow for each nonterminal and LL(1) table.
$E \rightarrow a\ A \mid (E)$
$A \rightarrow +\ E \mid *\ E \mid \varepsilon$

Solution: First (E) = {a, (}
First(A) = {+, *,ε}
Follow(E) = {$,)} by rule 1 & rule 2.
Follow(A) = Follow(E) = {$,)}

To construct LL(1) table, rows are two as there are two nonterminals.
In the row E, place E → a A in the column given by First(aA) = {a}

In the row E, place E → (E) in the column given by First((E)) = {(}
In the row A, place A → + E in the column given by First(+ E) = {+}
In the row A, place A → * E in the column given by First(* E) = {*}
In the row A, place A → ε in the column given by Follow(A) = {),$}

Now LL(1) table is

	a	**+**	*****	**(**	**)**	**$**
E	E → a A			E → (E)		
A		A → + E	A → * E		A → ε	A → ε

∗3. Build an LL(1) parser for the following grammar
1. Program → begin d semi X end
2. X → d semi X | S Y
3. Y → semi S Y | ε

Solution: Place rule(1) in row ‘Program’ and column First(begin) = begin
Place rule(2) in row ‘X’ and column First(d) = d
Place rule(3) in row ‘Y’ and column First(semi) = semi
In parsing table let us place production numbers.

	begin	**d**	**semi**	**S**	**end**	**$**
Program	1					
X		2		3		
Y			4		5	

4. Construct LL(1) parsing table.
S → qABC
A → a | bbD
B → a | ε
C → b | ε
D → c | ε

Solution: First find first and follow sets of each nonterminal.

	First()	Follow()
S	{q}	{$}
A	{a,b}	{a,b,$}
B	{a,ε}	{b,$}
C	{b,ε}	{$}
D	{c,ε}	{a,b,$}

The LL(1) table is

	q	**a**	**b**	**c**	**$**
S	S → qABC				
A		A → a	A → bbD		

B		B → a	B → ε		B → ε
C			C → b		C → ε
D		D → ε	D → ε	D → c	D → ε

As there are no multiple entries, the grammar is LL(1);

5. Construct LL(1) parsing table.
   S → A, A → a B | Ad, B → bBC | f, C → g.

   Solution: The given grammar is left recursive. So first eliminate left recursion. The resulting grammar is as follows:

   S → A, A → a B A′, A′ → d A′ | ε, B → bBC | f, C → g.

   Now find first and follow functions.

	First()	Follow()
S	{a}	{$}
A	{a}	{$}
A′	{d,ε}	{$}
B	{b,f}	{d,g,$}
C	{g}	{d,g,$}

   The LL(1) table is

	a	b	d	f	g	$
S	S → A					
A	A → a B A′					
A′			A′ → d A′			A′ → ε
B		B → bBC		B → f		
C					C → g	

6. Check whether the following grammar is LL(1) or not. If it is not LL(1) where do you find multiple entries in parsing table?

   S → AB | C, A → Bb | Cd, B → aB, C → ad

   Solution: For grammar to be LL(1)
   For row S, condition to be satisfied is ---- First(AB) ∩ First(C) = Ø
   {a} ∩ {a} ≠ Ø ..condition is not satisfied
   The multiple entries would be S → AB, S → C.
   For row A, condition to be satisfied is ---- First(Bb) ∩ First(Cd) = Ø
   {a} ∩ {a} ≠ Ø condition is not satisfied. Hence grammar is not LL(1). We get multiple entries in the row A under the column "a";
   The multiple entries would be A → Bb, A → Cd.
   Anyway nonterminal C is defined with one rule. Hence, no multiple entries in row C.

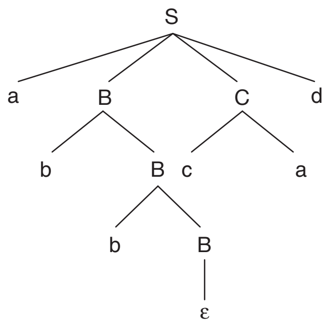
7. Consider the grammar

   $S \rightarrow$ **a** $B\ C$ **d** | **d** $C\ B$ **e**
   $B \rightarrow$ **b** B | **ε**
   $C \rightarrow$ **c a** | **a c** | **ε**

   a. What are the terminals, nonterminals, and the start symbol for the grammar?
   b. Give the parse tree for the input string **abbcad**.
   c. Compute the First sets and Follow sets for each of the nonterminals in the grammar.

   Solution:

   a. terminals: {a, b, c, d, e}, nonterminals: {S, B, C}, start symbol: S.
   b.



   c. FIRST(S) = {**a**, **d**}, FIRST(B) = {**b**, ε}, FIRST(C) = {**a**, **c**, ε}.
   FOLLOW(S) = {\$}, FOLLOW(B) = {**a**, **c**, **d**, **e**}, FOLLOW(C) = {**b**, **d**, **e**}.

8. Construct a recursive decent parser in C for the grammar in problem 7.

   Solution: Assume that functions **match**() and **error**() are already available and the global variable **token** is used to store the current token read by function match().

```
    void  S() {
             switch(token) {
              case a: match(a); B(); C(); match(d); break;
            case d: match(d); C(); B(); match(e); break;
            default: error();
            }
      }
    void C() {
    switch(token) {
      case c: match(c); match(a); break;
      case a: match(a); match(c); break;
      case b: case d: case e: break;
      default: error();
   }
}
```

9. Construct an LL(1) parsing table for the grammar in problem 7.

Solution:

	a	b	c	d	e	$
S	$S \to$ **a** B C **d**			$S \to$ **d** C B **e**		
B	$B \to \varepsilon$	$B \to$ **b** B	$B \to \varepsilon$	$B \to \varepsilon$	$B \to \varepsilon$	
C	$C \to$ **a c**	$C \to \varepsilon$	$C \to$ **c a**	$C \to \varepsilon$	$C \to \varepsilon$	

10. Consider the following grammar:

$S \to ScB \mid B$
$B \to e \mid efg \mid efCg$
$C \to SdC \mid S$

Give an LL(1) grammar that generates the same lanuage.

First, eliminate left-recrsion:

$S \to BS'$
$S' \to cBS \mid \varepsilon$
$B \to e \mid efg \mid efCg$
$C \to SdC \mid S$

Then, eliminate common prefixes by left-recrsion:

$S \to BS'$
$S' \to cBS' \mid \varepsilon$
$B \to eB'$
$B' \to \varepsilon \mid f\, B''$
$B'' \to g \mid C\, g$
$C \to SC'$
$C' \to dC \mid \varepsilon$

Summary

- The process of verifying whether an input string matches the grammar of the language is called **parsing**.
- The output of a parser is a parse tree for the set of tokens produced by the lexical analyzer.
- Parsers are broadly classified into two categories—Top-down parser or bottom-up parser based on the way the parse tree is constructed.
- Top-down parsers are simple to construct.
- **Top-down** parsing involves generating the string starting from the start symbol repeatedly applying production rules for each nonterminal until we get a set of terminals.
- Top-down parsers are broadly classified into two categories based on the design methodology used, with full backtracking and without backtracking. With full backtracking, we have Brute Force Technique.
- Brute Force Technique is not preferred practically because it is very slow.
- A *predictive parser* tries to predict which production produces the least chances of a backtracking and infinite looping.

- The advantage of eliminating left recursion is it avoids parser going into infinite loop.
- The advantage of left factoring is it avoids backtracking.
- Recursive descent parsing is writing recursive procedures for each nonterminal.
- Out of all top-down parsers the most widely used parser is nonrecursive descent parser, which is also called LL(1) parser. This does not use any procedures but uses a table and a stack.
- First (α) gives the set of terminals that begin the strings derived from α.
- Follow (A), for nonterminals A, gives a set of terminals that can appear immediately to the right of A in some sentential form.
- A grammar whose parsing table has no multiply defined entries is said to be LL(1) grammar.

Fill in the Blanks

1. In LL(1), First L indicates _______.
2. In LL(1), Second L indicates _______.
3. In LL(1), "1" indicates _______.
4. The requirements for LL(1) parser is _______.
5. Is LL(1) parser a top-down parser or bottom-up parser? _______
6. Can every unambiguous grammar be parsed by LL(1)? Yes/No
7. The number of procedures to be defined in recursive descent parser is _______.
8. The advantage of eliminating left recursion is _______.
9. The advantage of left factoring is _______.
10. For $A \rightarrow A\alpha \mid \beta$, equivalent grammar without left recursion is _______.
11. Left factor the grammar $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$
12. Is every LL(1) unambiguous? Yes/No._______
13. Is left factoring compulsory in designing recursive descent parser? Yes/No_______
14. _______is the advantage of recursive descent parser.
15. _______is the disadvantage of recursive descent parser.
16. Are the procedures in recursive descent parser recursive or nonrecursive?_______
17. Can we design a recursive descent parser with ambiguous grammar? Yes/No _______

Objective Question Bank

*1. The Grammar $A \rightarrow AA \mid (A) \mid \varepsilon$ is not suitable for predictive parsing because, the grammar is
(a) ambiguous (b) left recursive (c) right recursive (d) None

*2. Which of the following derivation does a top-down parser use while parsing an input string? The input is assumed to be in LR order.
(a) LMD (b) Leftmost traced out in reverse
(c) RMD (d) Rightmost traced out in reverse

*3. Consider the grammar shown below

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b$

In LL(1) table M of this grammar, the entries M[S′,e] and M[S′, $] are
(a) $S' \rightarrow eS$ & $S' \rightarrow \varepsilon$
(b) $S' \rightarrow eS$ &{}
(c) $S' \rightarrow \varepsilon$ & $S' \rightarrow \varepsilon$
(d) {$S' \rightarrow eS$ & $S' \rightarrow \varepsilon$} & $S' \rightarrow \varepsilon$

*4. Consider the grammar shown below
$S \rightarrow FR,\ R \rightarrow *S \mid \varepsilon,\ F \rightarrow id$
In LL (1) table M of this grammar, the entries M[S, id] and M[R, $] are
(a) $S \rightarrow FR, R \rightarrow \varepsilon$ (b) $S \rightarrow FR$ & {}
(c) $S \rightarrow FR$ & $R \rightarrow *S$ (d) $F \rightarrow id, R \rightarrow \varepsilon$

*5. Which of the following suffices to convert an arbitrary CFG to an LL(1) Grammar?
(a) Removing left recursion alone
(b) Factoring grammar alone
(c) Removing left recursion and factoring the grammar
(d) None

6. Consider the following grammar over the alphabet $\{b, g, h, i\}$:
$A \rightarrow B\ C\ D$
$B \rightarrow b\ B \mid \varepsilon$
$C \rightarrow C\ g \mid g \mid C\ h \mid i$
$D \rightarrow A\ B \mid \varepsilon$

Find follow(C)
(a) {$,g,b,i}
(b) {$,g,b}
(c) {$,g,b,i,h}
(d) None

7. Consider the following grammar:
$S \rightarrow ScB \mid B$
$B \rightarrow e \mid efg \mid efCg$
$C \rightarrow SdC \mid S$

Give an LL(1) grammar that generates the same language.

(a) $S \rightarrow B\ S'$
$S' \rightarrow c\ B\ S' \mid \varepsilon$
$B \rightarrow e\ B'$
$B' \rightarrow \varepsilon \mid f\ B''$
$B'' \rightarrow g \mid C\ g$
$C \rightarrow S\ C'$
$C' \rightarrow d\ C \mid \varepsilon$

(b) $S \rightarrow B\ S'$
$S' \rightarrow c\ B\ S' \mid \varepsilon$
$B \rightarrow e\ B'$
$B' \rightarrow \varepsilon \mid f\ B''$
$B'' \rightarrow g \mid C\ g$
$C \rightarrow S\ C'$

(c) $S \rightarrow B\ S'$
$S' \rightarrow c\ B\ S' \mid \varepsilon$
$B \rightarrow e\ B''$
$B' \rightarrow \varepsilon \mid f\ B''$
$B'' \rightarrow g \mid C\ g$
$C \rightarrow d\ C \mid \varepsilon$

(d) none

8. The following grammar is

$A \rightarrow \mathbf{A\ x}\ B$
$\quad\quad |\ \mathbf{x}$
$B \rightarrow \mathbf{x}\ B$
$\quad\quad |\ \mathbf{x}$

(a) unambiguous (b) left recursive (c) right recursive (d) ambiguous

9. Eliminate left recursion from the following grammar:

$A \rightarrow Bd$
$\quad\quad |\ Aff$
$\quad\quad |\ CB$
$B \rightarrow Bd$
$\quad\quad |\ Bf$
$\quad\quad |\ df$
$C \rightarrow h$
$\quad\quad |\ g$

(a) $A \rightarrow Bd\ A^I\ |\ CB\ A^I$
$A^1 \rightarrow ff\ A^I\ |\ \varepsilon$
$B \rightarrow df\ B^I$
$B^1 \rightarrow d\ B^I\ |\ fB^I\ |\ \varepsilon$
$C \rightarrow h$
$\quad |\ g$

(b) $A \rightarrow Bd\ A^I\ |\ CB\ A^I$
$A \rightarrow ff\ A^I\ |\ \varepsilon$
$B \rightarrow df\ B^I$
$B^I \rightarrow d\ B^I\ |\ fB^I\ |\ \varepsilon$
$C \rightarrow h$
$\quad |\ g$

(c) $A \rightarrow Bd\ A^I\ |\ CB\ A^I$
$A^I \rightarrow ff\ A^I$
$B \rightarrow df\ B^I$
$B^I \rightarrow d\ B^I\ |\ fB^I$
$C \rightarrow h$
$\quad |\ g$

(d) $A \rightarrow Bd\ A^I\ |\ CB$
$A^I \rightarrow ff\ A^I\ |\ \varepsilon$
$B \rightarrow df\ B^I$
$B^I \rightarrow d\ B^I\ |\ fB^I\ |\ \varepsilon$
$C \rightarrow h$
$\quad |\ g$

10. Give the Follow set for the nonterminal E in the grammar.

$E^I \rightarrow E$
$E \rightarrow \mathbf{num}$
$E \rightarrow (+\ A)$
$A \rightarrow A\ A$
$A \rightarrow E$

(a) {num, (,)} (b) {(,), num, $}
(c) {(, num, $} (d) {), num, $}

11. Give the Follow of B in the following grammar:

$A \rightarrow B\ C\ D$
$B \rightarrow w$
$\quad\quad |\ B\ x$
$C \rightarrow y\ C\ z$
$\quad\quad |\ m$
$D \rightarrow D\ B$
$\quad\quad |\ a$

(a) {x, y, m, $} (b) {x, y, w, $}
(c) {x, w, m, $} (d) {x, y, m, w, $}

Exercises

1. Construct a recursive descent parser to parse the string "w" for the given grammar "G"

   a. w = id*id+id & grammar is $E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow id \mid (E)$
   b. w = id+ id*id & grammar is $E \rightarrow E + T \mid T \quad T \rightarrow T\ F \mid F \quad F \rightarrow F * \mid a \mid b$
   c. w = bdc & grammar is $S \rightarrow Aa \mid bAc \mid dc \mid bda \quad A \rightarrow d$

2. Eliminate left recursion in the following grammar.

   $S \rightarrow aAb;\ S \rightarrow Ac \mid Sd \mid \varepsilon$

3. Eliminate left recursion in G.

   $S \rightarrow aAcBe;\ A \rightarrow Ab \mid b;\ B \rightarrow d$

4. Eliminate left recursion in G.

   $S \rightarrow L; L \mid L;\ L \rightarrow LB \mid B;\ B \rightarrow 0 \mid 1$

5. Eliminate left recursion in

   $A \rightarrow b \mid Bd; \quad B \rightarrow Bc \mid Ac$

6. Left factor the grammar

   $S \rightarrow iE\ tS \mid a \mid iE\ SeS.\ E \rightarrow b$

7. Left factor the grammar

   $S \rightarrow aAd \mid aB$
   $A \rightarrow b \mid c;\ B \rightarrow ccd \mid ddc$

8. Consider the following grammar, compute first () and follow () for each terminal. Check whether the grammar is LL (1).
   $S \rightarrow ACB \mid CbB \mid Ba;$
   $A \rightarrow da \mid BC; \qquad B \rightarrow g \mid e; \qquad C \rightarrow h \mid \varepsilon$

9. Consider the following grammar, computer first () and follow () for each terminal. Check whether the grammar is LL (1).

   $S \rightarrow ABCDE;\ A \rightarrow a \mid \varepsilon;\ B \rightarrow b \mid \varepsilon;\ C \rightarrow c \mid \varepsilon;\ D \rightarrow d \mid \varepsilon;\ E \rightarrow e \mid \varepsilon$

10. Construct LL(1) parsing table and verify whether it is LL (1).

    $S \rightarrow ABDh$
    $B \rightarrow cC$
    $C \rightarrow bC \mid \varepsilon$
    $D \rightarrow EF$
    $E \rightarrow g \mid e$
    $F \rightarrow f \mid \varepsilon$

11. Construct LL(1) parsing table and verify whether it is LL (1).

    $S \rightarrow AaAb \mid BaBb$ is it LL (1)
    $A \rightarrow \varepsilon, B \rightarrow \varepsilon$

12. Construct LL (1) parsing table and verify whether it is LL (1).

    $S \rightarrow a\ AB \mid \varepsilon$
    $A \rightarrow 1AC \mid 0C$
    $B \rightarrow 0S$
    $C \rightarrow 1$

13. Construct an LL(1) parser to parse the string "w" for the given grammar "G."

    id+ id*id & grammar is $E \rightarrow E + T \mid T \quad T \rightarrow T\ F \mid F \quad F \rightarrow F * \mid a \mid b$

14. Check whether the following grammar is LL (1).

    $S \rightarrow A, A \rightarrow a\ B \mid Ad, B \rightarrow bBC \mid f, C \rightarrow g.$

15. Check whether the following grammar is LL (1).

    $S \rightarrow a\ SbS \mid bSaS \mid \varepsilon$

16. Convert the following grammar to unambiguous grammar.
    Check whether the resulting grammar is LL (1).

    $R \rightarrow R + R \mid RR \mid R * \mid (R) \mid a \mid b$

17. Check whether the following grammar is LL (1).

    bExpr → bExpr or bterm | bterm
    bterm → bterm and bfactor | bfactor
    bfactor → not bfactor | (bexpr) true | false.

18. Check whether the above grammar is LL (1).

    $S \rightarrow A \mid a, A \rightarrow a$

19. Check whether the following grammar is LL (1).

    $S \rightarrow iE\ tS\ S1 \mid a$
    $S1 \rightarrow \varepsilon \mid eS$
    $E \rightarrow b$

20. Check whether the following grammar is LL (1).

    $E \rightarrow aA \mid (E)$
    $A \rightarrow +E \mid *E \mid \varepsilon$

21. Check whether the following grammar is LL (1).

    $S1 \rightarrow S\#$
    $S \rightarrow Aa \mid b \mid cB \mid d$
    $A \rightarrow a\ A \mid b$
    $B \rightarrow cB \mid d$

22. Check whether the following grammar is LL (1).

    $S1 \rightarrow S\#$
    $S \rightarrow AB$
    $A \rightarrow a \mid \varepsilon$
    $B \rightarrow b \mid \varepsilon$

23. Check whether the following grammar S-Grammar is LL (1).

    $S \rightarrow aS \mid bA;$ $A \rightarrow ccA \mid d$

24. Check whether the following grammar is LL (1).

    $S \rightarrow aB \mid \varepsilon$
    $B \rightarrow bC \mid \varepsilon$
    $C \rightarrow cS \mid \varepsilon$

25. Check whether the following grammar is LL (1).

    $S1 \rightarrow S\#$
    $S \rightarrow aAa \mid \varepsilon$
    $A \rightarrow abS \mid c$

26. Check whether the following grammar is LL (1).

    $S1 \rightarrow A\#$
    $A \rightarrow Bb \mid Cd$
    $B \rightarrow aB \mid \varepsilon$
    $C \rightarrow cC \mid \varepsilon$

27. Construct the LL (1) parsing table for following grammar.

    Program $\rightarrow$ begin d semi X end
    $X \rightarrow$ d semi X | s Y
    $Y \rightarrow$ semi s Y | ε

Key for Fill in the Blanks

1. scanning of input from left to right
2. parser uses LMD in deriving the string
3. number of look ahead symbols used in taking the parsing decisions.
4. Grammar should be free of left recursion and should be left factored.
5. Top-down parser
6. No
7. Generally depends on number of nonterminals. For every nonterminal there should be a procedure.
8. Avoids parser going into infinite loop
9. Avoids backtracking
10. $A \rightarrow \beta A', \quad A' \rightarrow \alpha A' \mid \varepsilon$
11. $A \rightarrow a\,A', \quad A' \rightarrow \beta 1 \mid \beta 2 \mid \beta 3$
12. Yes
13. No
14. Simple to construct
15. Can be designed for small languages
16. It depends on the production that is defined for the nonterminal. If production is recursive then procedure also will be recursive.
17. No

Key for Objective Question Bank

1. a	2. a	3.d	4. a	5. c	6. c
7. a	8. d	9. a	10. b	11. d	

Programming on Parsers

Parsers Code:

1. Write a C program for recursive descent parser for following grammar.

$E \rightarrow T E'$
$E' \rightarrow + TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow i$

/* Recursive descent parser in C*/
/* match()- is to match terminal derived from grammar with lookahead */

```
#include<stdio.h>
#include<conio.h>
void E();
void E'();
void T();
void T'();
void F();
void match(char);
 int flag = 1;
char l,t;

main()
{
Printf("enter input string\n");
scanf("%c",&l);
E();
}

/* Procedure for matching terminal with token */

void match(char t)
{
if(l == t)
scanf("%c",&l);
 else
{
 flag = 0;
```

```
        }
      }
/* Procedure for nonterminal E */

void E()
   {
      T();
      E' ();
      if((l == '$')&&(flag ! = 0))
printf("successful\n");
else
printf("unsuccessful\n");
}

/* Procedure for nonterminal E´ */

void E' ()
{
     if(l == '+')
     {match('+');
     T();
     E'();
     }
     else return;
}

/* Procedure for nonterminal T */

void T()
{
     F();
     T' ();
}

/* Procedure for nonterminal T´ */

void T´ ()
{
if(l == '*')
     {match('*');
     F();
     T' ();
     }
```

```
      else return;
}

/* Procedure for nonterminal F */


void F()
{
      match('i');
}
```

output:
Enter the input string:

i+i*i$

successful

i**i$

unsuccessful.

2. Design LL(1) parser

   Program logic
   - Read LL(1) table and input string to be parsed.
   - Read terminals in array 'ter'. This is used as columns of LL(1) table.
   - Read non terminals in array 'nter' This is used as rows of LL(1) table.
   - Now read table in two dimensional array of strings 'table'.
   - Use LL(1) parsing algorithm to parse the input string.
   - Whenever top of the stack is a nonterminal, to replace that by corresponding production—use two functions get_nt(), get_t() to get row and column in table.
   - Take the production from that row and column, reverse it and push symbol by symbol.

   Coding:
   /* LL(1) parser in C */

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
char table[10][10][10], nter[10],ter[10],inp[20],stack[20];
int nut,nun,i = 0,top = 0;
int get_nter(char);
int get_ter(char);
```

```
void replace(char,char);

void main()
{
  int i,j;
  clrscr();

       /* read terminals and nonterminals */

  printf("Enter number of terminals\n");
  scanf("%d",&nut);
  printf("Enter number of nonterminals\n");
  scanf("%d",&nun);
  printf("Enter all non terminals\n");
  scanf("%s",nter);
  printf("Enter all terminals\n");
  scanf("%s",ter);
  for(i = 0;i<nut;i++) printf("%c\t",nter[i]);
  printf("\n");
  for(j = 0;j<nun;j++) printf("%c\t",t[j]);
  printf("\n");

     /* read table */
  for(i = 0;i<nun;i++)
    for(j = 0; j<nut;j++)
    {
      printf("Enter for %c and %c",nter[i],ter[j]);
      scanf("%s",table[i][j]);
   }

     / * print table */

  for(j = 0;j<nut;j++)
    printf("\t %c",ter[j]);
  printf("\n");
  for(i = 0;i<nun;i++)
  {
    printf("%c \t",nter[i]);
    for(j = 0;j<nut;j++)
    {
      printf("%s \t",table[i][j]);
   }
    printf("\n");
 }
```

```
     /* read input string to parse */

  printf("Enter the string to parse\n");
  scanf("%s",inp);

     /* use LL(1) algoritm */

  stack[top++] = '$';
  stack[top++] = nter[0];
  i = 0;
  while(1)
  {
    if((stack[top-1] == '$')&&(inp[i] == '$'))
    {
      printf("String accepted\n");
      return;
   }
    else if(!isupper(stack[top-1]))
    {
     if(stack[top-1] == inp[i])
     {
     i++; top--;
     }
      else
      {
      printf("error not accepted\n");
      return;
     }
   }
    else
    {
      replace(stack[top-1],inp[i]);
   }
 }

}

    /* get index of nonterminals in nonterminals array */

int get_nter(char x)
{
  int a;
  for(a = 0;a<nun;a++)
    if(x == nter[a]) return a;
```

```
  return 100;
}

/* get index of terminal in terminal array */

int get_ter(char x)
{
  int a;
  for(a = 0;a<nut;a++)
    if(x == ter[a]) return a;
  return 100;
}
void replace(char NT, char T)
{
  int in1,it1,len;
  char str[10];
  in1 = get_nter(NT);
  it1 = get_ter(T);
  if((in1! = 100)&&(it1! = 100))
  {
    strcpy(str,table[in1][it1]);
    if(strcmp(str,"#") == 0)
    {
      printf("Error\n");
      exit();
    }
    if(strcmp(str,"@") == 0)
      top--;
    else
    {
      top--;
      len = strlen(str);
      len--;
      do
      {
      stack[top++] = str[len--];
     }while(len> = 0);
   }
  }
  else
  {
   printf("Not valid\n");
 }
}
```

Results

INPUT:

Enter number of terminals: 6

Enter number of non terminals: 5

Enter all non terminals: E E′ T T′ F

Enter all terminals: id + * () $

Enter table- # for blank, @ for ε:

```
T E′  #    #   TE′  #  #
#   +TE′  #    #    @   @
FT′   #    #   FT′   #    #
#    @   *FT′   #    @   @
id     #    #      (E)   #    #
```

Enter the string to parse

id+id*id$

output:

String accepted

CHAPTER 5

Bottom-Up Parsers

Bottom-up parsing is a more general parsing technique when compared with top-down parsing. The widely used method in practice is bottom-up parsing. Examples of bottom-up parsers are YACC and Bison, which are automatic parser generators.

CHAPTER OUTLINE

5.1 Bottom-Up Parsing
5.2 Handle
5.3 Why the Name SR Parser
5.4 Types of Bottom-Up Parsers
5.5 Operator Precedence Parsing
5.6 LR Grammar
5.7 LR Parsers
5.8 LR Parsing Algorithm
5.9 Construction of the LR Parsing Table
5.10 LR(0) Parser
5.11 SLR(1) Parser
5.12 Canonical LR(1) Parsers CLR(1)/LR(1)
5.13 LALR(1) Parser
5.14 Comparison of Parsers: Top-Down Parser vs. Bottom-Up Parser
5.15 Error Recovery in LR Parsing
5.16 Parser Construction with Ambiguous Grammars

Bottom-up parsing is the most widely used parsing technique used in most of the automatic parser generators. In this chapter, we discuss what is shift reduce (SR) parsing and the types of different SR parsers. We also discuss in detail parsing algorithms of different bottom-up parsers. Given a grammar, how to test whether it is suitable for any type of SR parser is also discussed. Finally, comparison of all the parsers is described.

5.1 Bottom-Up Parsing

Bottom-up parsing is an attempt to construct a parse tree for the given input string in the bottom-up manner. It starts construction of the tree starting at the leaves (the bottom) and working up toward the root (the top). Bottom-up parsing is defined as an attempt to reduce the input string *w* to the start symbol of the grammar by tracing out the rightmost derivation of *w* in reverse. This is analogous to constructing a parse tree for a given input string "w" in bottom-up manner by starting with the leaves and proceeding toward the root.

Given a grammar "G" and an input string *w*, bottom-up parser starts construction of parse tree with input string *w*. At any time, it identifies a substring that matches with the right hand side of a production of the grammar. This substring is replaced by the left-hand side nonterminal of the grammar. If this replacement gives the generation of sentential form that is one step along the reverse of the rightmost derivation. This process of detecting the substring that matches with the right hand side of production and replacing that substring by the left hand side nonterminal is continued until the given string is reduced to the start symbol S. Let us understand the process with the following example.

Consider the grammar:

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

The sentence "abbcde" can be reduced to "S" as follows shown in Figure 5.1.

a<u>b</u>bcde
a<u>Abc</u>de (replace b by A using $A \rightarrow b$)
aA<u>d</u>e (replace Abc by A using $A \rightarrow Abc$)
<u>aABe</u> (replace d by B using $B \rightarrow d$)
S (replace aABe by S using $S \rightarrow aABe$)

The process of replacing a substring by a nonterminal in bottom-up parsing is called *reduction*.

Bottom-up parsing can be viewed as tracing out the rightmost derivation in reverse. Look at the reductions in the above example:

$$abbcde \Leftarrow aAbcde \Leftarrow aAde \Leftarrow aABe \Leftarrow S$$

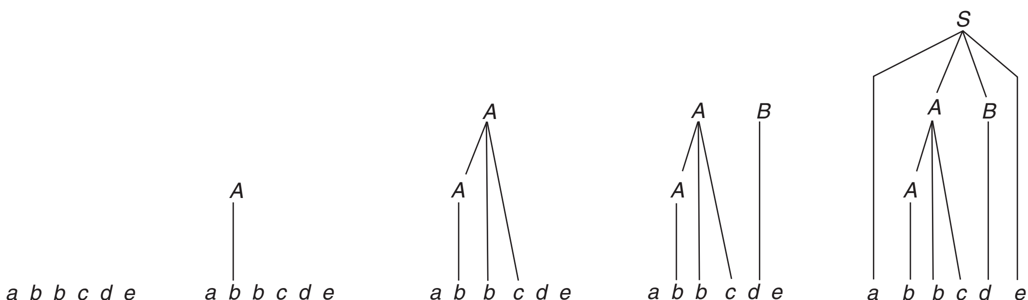


Figure 5.1 Reducing the String "abbcde"

This is nothing but the reverse of the rightmost derivation. The reason why bottom-up parser traces out the rightmost derivation in reverse but not leftmost is because the parser scans the input string *w* from left to right, one symbol at a time. To trace out *in reverse* by *scanning from left to right*, it is possible to reduce only with rightmost derivation.

So the task of a bottom-up parser at any time is to identify a substring that matches with the right hand side of the production, if that substring is replaced by left hand side nonterminal that should give one step along the reverse of rightmost derivation.

In bottom-up parsing, finding a substring that matches with the right hand side of a rule as well as its position in the current sentential form is very important. In order to take care of both factors into account, let us define handle.

5.2 Handle

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 γ. Hence, handle can be defined as: A *handle* of a right sentential form γ is a production $A \rightarrow \beta$ and the position of β in γ. The string β will be found and replaced by A to produce the previous right sentential form in the rightmost derivation of γ. Handle is nothing but a substring that matches the right hand side of a production, which when reduced to nonterminal gives one step along the reverse of the rightmost derivation.

Consider the following example:

$$\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} * \text{Expr} \mid \text{id}$$

The rightmost derivation for the string "id + id * id" is

$$\begin{aligned} \text{Expr} &\Rightarrow \text{Expr} + \text{Ex\underline{p}r} \\ &\Rightarrow \text{Expr} + \text{Expr} * \text{Ex\underline{p}r} \\ &\Rightarrow \text{Expr} + \text{E\underline{x}pr} * \text{id} \\ &\Rightarrow \text{E\underline{x}pr} + \text{id} * \text{id} \\ &\Rightarrow \text{id} + \text{id} * \text{id} \end{aligned}$$

The handles of the sentential forms occurring in the above derivation are shown below in Table 5.1.

Table 5.1 Handles of the Right Sentential Form

Sentential form	Handle
id + id * id	Expr → id at the first id
Expr + id * id	Expr → id at the position following +
Expr + Expr * id	Expr → id at the position following *
Expr + Expr * Expr	Expr → Expr * Expr at the position following +
Expr + Expr	Expr → Expr * Expr at the position preceding the end marker

Bottom-up parsing can be described as an attempt to detect handle and reduce the handle. Reducing the handle is even called *handle pruning*. The difficulty in bottom-up parsing is detecting the handle at any time in parsing.

5.3 Why the Name SR Parser

Bottom-up parsers are also called SR parsers. To understand why they are called SR parser, let us look at the way a bottom-up parser works.

Given a grammar Expr → Expr + Expr | Expr * Expr | id & input string w = "id + id * id," let us look at bottom-up parsing. Parser takes the string w and reads, symbol by symbol from left to right. Whatever it reads it should remember till a handle is recognized. So whatever is read will be stored on an auxiliary memory called "stack." It reads symbol by symbol and stores/pushes it on to the stack until a handle is detected. This process of reading the symbol and pushing on to the stack is called "shift action." Once a handle is detected, the parser reduces the handle by nonterminal of the production. This is called "reduce action." So at any time, the task performed by a bottom-up parser is to read the symbol and shift the symbol until a handle is detected, when a handle is detected it performs reduce action. So the main actions performed by a parser are shift or reduce. That is why it is called SR parser. This shift/reduce action is performed until the parser halts. Halting may occur in two situations—either on successful completion or on the occurrence of an error.

A simple bottom-up parsing technique using a stack based implementation is as follows:

1. Initially stack contains only the sentinel $, and input buffer contains the input string w$.
2. While stack not equal to $S do
   a. While there is no handle at the top of stack, do shift input buffer(read symbol from L-R) and push the symbol onto stack
   b. if there is a handle on top of stack, then pop the handle and reduce the handle with its nonterminal and push it onto stack
3. Halt.

Consider the grammar:

S → aABe
A → Abc | b
B → d

The action performed by parser while parsing the input string is shown in Table 5.2.

Table 5.2 Shift Reduce Parsing

Stack	Input	Action
$	abbcde$	shift a
$a	bbcde$	shift b
$ab	bcde$	reduce by A → b
$aA	bcde$	shift b
$aAb	cde$	shift c

\$aAbc	de\$	reduce by A → Abc
\$aA	de\$	shift d
\$aAd	e\$	reduce by B → d
\$aAB	e\$	shift e
\$aABe	\$	reduce by S → aABe
\$S	\$	**accept**

5.4 Types of Bottom-Up Parsers

Bottom-up parsers are broadly classified into two categories as shown in Figure 5.2—operator precedence parser and LR parser. Operator precedence parser is a simple parser, which is preferred for parsing expressions. The most powerful parsers are under the LR category. There are four types of LR parsers: LR(0), simple LR-SLR(1), look ahead LR-LALR(1), and

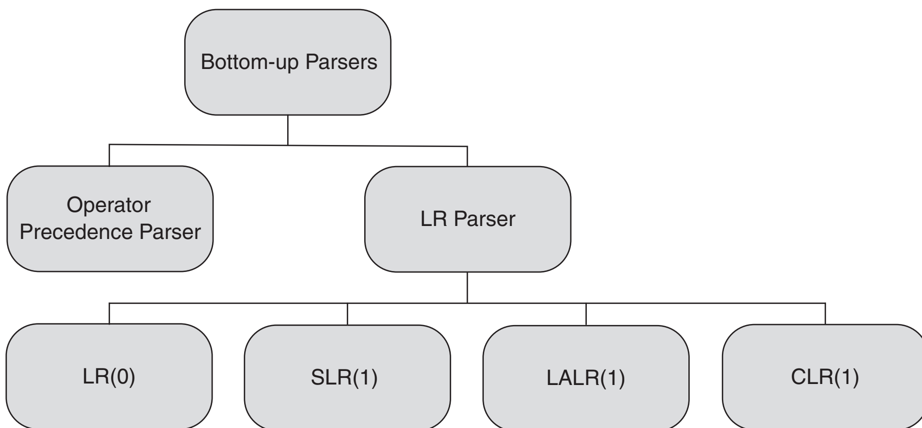


Figure 5.2 Types of Bottom-Up Parsers

canonical LR- CLR(1). Out of all LR parsers CLR(1)/LR(1) is the most powerful parser; it can be used to parse almost all grammars.

Out of all bottom-up parsers, LR(0) is less powerful. It can be used to parse only a small class of grammars; hence, it is practically not used. The preferred parsers are SLR(1), LALR(1), and CLR(1). SLR(1) is simple to construct but once again least powerful. Canonical LR(1) is the most powerful; hence its implementation is costly. The power and cost to construct LALR(1) is intermediate between those of SLR(1) and LALR(1). Though LR(0) is not used, it is the basic machine on which all the other three are built. Hence, we also discuss the design of LR(0) parser. Let us look at the design of a simple SR parser, that is, operator precedence parser.

5.5 Operator Precedence Parsing

This is a technique for constructing shift-reduce parsers by hand for a small class of grammars. To construct LL(1) parser, there is a restriction on the grammar; grammar should be free of left recursion and should be left factored. Similarly, to construct operator precedence parser, there is a restriction on the grammar, that is, the grammar must be operator grammar. Let us look at what is operator grammar.

***Operator grammar*:** The grammar that does not contain null production and two adjacent nonterminals on the right hand side of any production is called operator grammar.

For example:

$E \rightarrow E\ B\ E \mid id$
$B \rightarrow + \mid - \mid =$

This is not operator grammar. There are no null productions but adjacent nonterminals are not allowed. To construct operator grammar, expand the nonterminal B. Then the resulting grammar is operator grammar.

$E \rightarrow E + E \mid E * E \mid E = E \mid id$

The resulting grammar is ambiguous grammar since it is operator grammar; hence, it can be used for parser construction. Operator precedence parser does not bother about whether the grammar is ambiguous or not but it must be operator grammar. Operator precedence parser is the only parser that can be constructed even if the given grammar is ambiguous. All other parsers (LL, LR) are constructed only with unambiguous grammars. Now let us see given an ambiguous grammar how to construct operator precedence grammar that parses the string unambiguously.

Example 1: Convert the following grammar to operator grammar.

$S \rightarrow S\ A\ S \mid a$
$A \rightarrow b\ S\ b \mid b$ assume S is the start symbol.

Solution: There are no null productions but adjacent nonterminals SAS is not allowed.
Hence expand the nonterminal A. By substituting for nonterminal A, we get the grammar as follows

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

This is the resulting operator grammar. Here A is the useless symbol. So it can be ignored.

Operator precedence parser mainly works by considering the precedence among the operators. That is why it is called operator precedence parser. Though the given grammar is ambiguous, the parser preserves the actual precedence among the operators in a table called precedence relation table. Like LL(1) table used by LL(1) parser, operator precedence parser uses precedence relation table for parsing the input string. In this table, the actual precedence among two operators is preserved as precedence relation $\lessdot, \gtrdot$ or $\doteq$. So let us understand precedence relations.

5.5.1 Precedence Relations

The following precedence relations are defined across terminals of an operator grammar.

a ⋗ b (a “takes precedence over” b) So a is reduced before b. Ex: * ⋗ +
a ⋖ b (a “yields precedence to” b). So b is reduced before a. Ex: + ⋖ *
a ≐ b (a “has the same precedence as” b). So both are reduced at the same time.
Ex: (=).

For the sentinel symbol $ and any terminal b we define $ < b and b < $.

The precedence relations looks similar to logical relations >, < and ==. But they are not the same. With logical relations whenever a > b then b > a never exits. But with precedence relations both may become true at the same time, that is, a > b and b > a are true at the same time. Later we will discuss the situation where such a condition is satisfied.

5.5.2 Recognizing Handles

The main difficulty in bottom-up parsing is identifying the handle. The precedence relations help the parser in recognizing the handle. Once the precedence information in stored in the precedence table with precedence relations, let us see how to recognize the handle. Let us understand with an example. Consider the grammar E → E + E | E * E | id and input string

“ id + id * id.”

Parser constructs the parsing table first. Assume that parsing table is available as shown in Table 5.3.

Table 5.3 Precedence Relation Table

	id	**+**	*****	**$**
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

To parse the given string, insert the string between two end markers $. Then read symbol by symbol from left to right and insert the precedence relation between two terminals by using the table. Then we get the string as follows:

$ ⋖ id ⋗ + ⋖ id⋗ * ⋖id ⋗ $.

Once precedence relations are inserted, recognition of handle is as follows:

1. Scan the string from the left end until the first ⋗ is encountered.
2. Then scan backwards to the left over any ≐ until a ⋖ is encountered.

3. Everything to the left of the first > and to the right of the encountered <, including any intervening or surrounding nonterminals is the handle.

We apply the above procedure on string $ < id > + < id > * < id > $.

Here handle is whatever is included between < and > at anytime. This is the main criteria used for recognizing the handle. So in first step of parsing, handles are id, id, id. Once handle is recognized, it is reduced. So after reducing we get the string

$ < id > + < id > * < id > $ ⇒ $E + E * E$.

The resulting string $E + E * E$ is viewed by the parser as $ + * $. That is, nonterminals are ignored by the parser as they are not defined with any precedence. Once again the parser repeats the above procedure on the string until it reduces to the start symbol. This is shown below.

$ < id > + < id > * < id > $ (reduces id by E; left out string is E + E * E)
$ + * $.
$ < + < * > $ (reduces E * E by E, then E+E by E; left out string is E)
$ $ successful completion. (nonterminals are ignored)

The above procedure can be stated as an algorithm. The operator precedence parser uses a stack. It reads the input string into a buffer and appends $ as right end marker. It then pushes $ onto the stack. This is to identify the bottom of the stack. It mainly works by comparing the top of the stack with the look ahead symbol at anytime. Here comparison is in terms of precedence relation and based on this the parsing decision is made by using the following algorithm.

An SR parser mainly performs two actions at any time: shift or reduce until it halts. Halting can be either on successful completion or on an error. So to define parsing algorithm for *any SR parser* we need to define only two steps—(1) when it prefers a shift action and how it completes a shift action (2) when it prefers a reduce action and how it completes a reduce action. The parsing algorithm of an operator precedence parser is given below.

5.5.3 Parsing Algorithm for Operator Precedence Parser

1. Initially, stack contains only $ and input buffer contains *w*$ where *w* is input string
2. Repeat forever
   a. Let "a" be the top element on stack and "b" is the current element pointed by the input pointer, that is, the look ahead symbol
   b. if a < b or a ≐ b, push b onto the stack and increment input pointer. *(Shift action)*
   c. if a > b then *(Reduce action)* Repeat Pop the stack until the top of the stack is < to the terminal most recently popped
   d. If a = b = $, announces successful completion

Example 2: Parse the input string a + b * c by using the above algorithm.

Solution: The result of each step is described in the following Table 5.4.

Table 5.4 Parsing Actions of the Operator Precedence Parser

Stack	Input	Action taken
$	a + b * c$	
$a	+ b * c$	shift a because $ ⋖ a
$	+ b * c$	pop a because a ⋗ +
$+	b * c$	shift + because $ ⋖ +
$+b	* c$	shift b because + ⋖ b
$+	* c$	pop b because b ⋖ *
$+*	c$	shift * because + ⋗ *
$+*c	$	Shift c because * ⋖ c
$+*	$	pop c because c ⋖ $
$+	$	pop * because * ⋖ $
$	$	pop + because + ⋖ $
$	$	Accept

5.5.4 Construction of the Precedence Relation Table

The actual precedence among the operators is used in constructing the precedence relation table.

If θ_1 and θ_2 are two operators, the following heuristics are used in constructing the table.

1. If θ_1 has higher precedence than θ_2 then make the. entries as $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$. For example, * has higher precedence than + so define the relations between them as * ⋗ + as well as + ⋖ *.
2. If θ_1 and θ_2 are of equal precedence, then consider associativity.
   i. If they are left associative, define relation as $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$.
   For example, take expression a + b – c + d. Both + & – are left associative. So here first + is to be reduced. Whichever is to be reduced first, insert that between ⋖ and ⋗. Here to ensure left associativity, the relation must be $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$, i.e. + ⋗ – ⋗ + ⋗. Here θ_1 is +ve and θ_2 is –ve, i.e. $\theta_1 \gtrdot \theta_2 \gtrdot \theta_1 \gtrdot$. Unless we define the relation as $\theta_1 \gtrdot \theta_1$ and $\theta_2 \gtrdot \theta_1$, left associativity cannot be obtained.
   ii. If they are right associative, define relation as $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$.
   For example, consider the expression a ↑ b ↑ c ↑ d. Assume ↑ is right associative. So here right ↑ is to be reduced then second ↑. Whichever is to be reduced first, insert that between ⋖ and ⋗. Here to ensure right associativity, the relation must be $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$ i.e.,⋖ ↑ ⋖ ↑ ⋖ ↑ ⋗. Here θ_1 is ↑ and θ_2 is ↑. Hence $\lessdot \theta_1 \lessdot \theta_2 \lessdot \theta_1 \gtrdot$. Unless we define the relation as $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$ right associativity cannot be obtained.
3. In addition to operators, we find other symbols in an expression like id, (,), $. The precedence relation is defined based on actual precedence among the terminals. Some of the relations are as follows.
   i. θ has less precedence than id. $\Rightarrow \theta \lessdot id$ and $id \gtrdot \theta$.
   ii. θ has high precedence than \$. $\Rightarrow \theta \gtrdot \$$ and $\$ \lessdot \theta$.

iii. id has high precedence than \$. $\Rightarrow$ id $\gtrdot$ \$ and \$ $\lessdot$ id
iv. θ and (are right associative. $\Rightarrow \theta \lessdot ($ and $(\lessdot \theta$.
v. θ and) are left associative. $\Rightarrow \theta \gtrdot)$ and $) \gtrdot \theta$.

Other relations are \$ $\lessdot$ (   ) $\lessdot$ \$ (=)
) $\gtrdot$)   ($\lessdot$ id   id $\gtrdot$), ($\lessdot$ (.

Example 3: Prepare precedence relation table for parsing the input string id + id * id.

Solution: The rows and columns of the table correspond to terminals. Here terminals are id, +, * and \$(input end marker). So there will be four rows and four columns for each terminal. To define relations in table, we need to know the actual relations among operators. Here we assume that * has higher precedence than + and both are left associative. By using above procedure we prepare Table 5.5 as shown below.

Table 5.5 Precedence Relation Table

	id	**+**	*****	**\$**
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
\$	$\lessdot$	$\lessdot$	$\lessdot$	

∗**Example 4:** Consider the following grammar

Para → Sentence Rp | Sentence

Rp → b̸ Sentence Rp | Sentence

Sentence → word b̸ Sentence | word

word → letter * word | letter

letter → id

Here "b̸" is blank space

a. Convert the following grammar into operator form.
b. Define precedence relations among the terminals and show how to use a stack algorithm to parse the string "id * id b̸ id * id."

Solution: (a) In the given grammar, para and Rp are not in the operator form because they contain adjacent nonterminals. The remaining productions are in the required form. So to convert para and Rp into the operator form, we need to eliminate adjacent nonterminals "Sentence Rp." To do this, substitute for Rp in para; then we get the grammar as follows.

Para → Sentence b̸ Sentence Rp | Sentence b̸ Sentence | Sentence

Here once again we are getting adjacent nonterminals. So mere substitution may not work. If you look at the grammar "para" is defined as "Sentence Rp" (Para → Sentence Rp). So in the above grammar, replace "Sentence Rp" by "para." The resulting grammar is

Para → Sentence b̸ para | Sentence b̸ Sentence | Sentence

Sentence → word b̸ Sentence | word

word → letter * word | letter

letter → id

Now this is operator grammar.

(b) To prepare the precedence relation table, we need to have the actual precedence among operators. Look at the above grammar; here b̸,* and id are terminals. id has highest precedence. * has higher precedence than b̸ as * is defined at a lower level. * and b̸ are right associative as the productions defining * and b̸ are right recursive.

The precedence relation table will have 4 rows and 4 columns for id, *, +, and $. It is shown in Table 5.6.

Table 5.6 Precedence Relation Table

	id	*	b̸	$
id		>	>	>
*	<	<	>	>
b	<	<	–	>
$	<	<	<	

No two ids appear in any expression. It is an error. So there is no relation between id on id. The same is applicable for b on b also.

5.5.5 Mechanical Method of Constructing Operator Precedence Table

AQ 1

There is another mechanical way of constructing the operator precedence table. This way uses two functions Leading () and Trailing(). Let us look at the functions.

Leading(A) gives set of terminals "a" such that "a" is the leftmost terminal in some string derived from A. It is evaluated by using following rule.

If "a" is in Leading(A) if there is a production of the form $A \rightarrow \alpha a \beta$ where α is ε or single nonterminal.

Leading(A) $=\{a \mid A \Rightarrow \alpha a \beta$ where α is ε or single nonterminal$\}$.

Tailing(A) is set of terminals "a" such that "a" is rightmost in a string derived from A.

It is evaluated by using following rule.

If "a" is in Trailing(A) if there is a production of the form $A \rightarrow \alpha a \beta$ where β is ε or single nonterminal.

Trailing(A) $=\{a \mid A \Rightarrow \alpha a \beta$ where β is ε or single nonterminal$\}$.

For example, consider the following grammar:

S → aABe
A → Ab | d
B → d

Leading(S) = {a} Trailing(S) = {e}
Leading(A) = {b, d} Trailing(A) = {b,d}
Leading(B) = {d} Trailing(B) = {d}

5.5.6 Calculating Operator Precedence Relation < > =

Once leading and trailing of each nonterminal are computed, then the operator precedence table is filled with the relations based on the following rules.

1. If there is a production A → αaBbβ, where α and β are some strings then set the relation for a and b as **a ≐ b.**
2. If there is a production A → αaBβ, where α and β are some strings then set the relation for a with the elements in leading (B) as **a ⋖ leading(B).**
3. If there is a production A → αBaβ, where α and β are some strings then set the relation for elements in trailing (B) with a as **trailing(B) ⋗ a.**
4. If S is the starting nonterminal then set the relation for $ with elements of leading(S) **$ ⋖ leading(S).**
5. If S is the starting nonterminal then set the relation for elements of leading(S) with $ as **trailing(S) ⋗ $.**

The algorithm for setting the relation based on the production is given below. After applying the algorithm apply rule 4 and 5 to fill the relation with respect to $.

For each A → $X_1 X_2 \ldots X_n$ do
For i = 1 to n−1 do

1. If X_i and X_{i-1} are both terminals then set
   $X_i \doteq X_{i+1}$, ex: S→ ab a≐b
   Or if X_i and X_{i+2} are terminals, X_{i+1} is nonterminal then set
   $X_i \doteq X_{i+2}$, ex: S→ aAb a≐b
2. If X_i is terminal and X_{i+1} is nonterminal then set
   Xi ⋖ Leading(Xi+1), ex: S → aA a⋖ Leading(A)
3. If X_i is nonterminal and X_{i+1} is terminal then set
   Trailing(X_i) ⋗ X_{i+1}, ex: S → Aa Trailing(A) ⋗ a

Example 5: Construct precedence relation table for the following grammar.
S → aAcBe
A → Ab | b
B → d

Solution: First calculate leading and trailing of nonterminals.

Leading(S) = {a} Trailing(S) = {e}
Leading(A) = {b,d} Trailing(A) = {b,d}
Leading(B) = {d} Trailing(B) = {d}

Now construct the table with the above algorithm.

Consider the first production $S \rightarrow aAcBe$,

Here using rule 1,

Hence $a \doteq c$ and $c \doteq e$

According to rule 2, $a \lessdot$ Leading(A), $c \lessdot$ Leading(B)

Hence $a \lessdot \{b,d\}$, $c \lessdot d$.

According to rule 3, Trailing(A) $\gtrdot c$, Trailing(B) $\gtrdot e$

Hence $\{b,d\} \gtrdot c$, $d \gtrdot e$

According to rule 4, $\$ \lessdot$ leading(S).

Hence $\$ \lessdot a$.

According to rule 5 trailing(S) $\gtrdot \$$

Hence $e \gtrdot \$$.

Consider the second production $A \rightarrow Ab$

According to rule 3 trailing(A) $\gtrdot b$

Hence $\{b,d\} \gtrdot b$

The table is prepared as shown in Table 5.7.

Table 5.7 Precedence Relation Table

	a	**b**	**c**	**d**	**e**	**$**
a		$\lessdot$	$\doteq$	$\lessdot$		
b		$\gtrdot$	$\gtrdot$			
c				$\lessdot$	$\doteq$	
d		$\gtrdot$	$\gtrdot$		$\gtrdot$	
e						$\gtrdot$
$	$\lessdot$					

Example 6: Construct the precedence relation table for the following grammar.

$S \rightarrow baXaS \mid ab$

$X \rightarrow Xab \mid aa$

Solution:

leading(S) = {b,a} trailing(S) = {a,b}

leading(X) = {a} trailing(X) = {b,a}

Considering production $S \rightarrow baXaS$ we get the relations as

using rule 1 we get $a \doteq a$

$a \lessdot$ leading(X) $\Rightarrow a \lessdot \{a\}$

$a \lessdot$ leading(S) $\Rightarrow a \lessdot \{a,b\}$

trailing(X) $\gtrdot a$ $\Rightarrow \{a,b\} \gtrdot a$

Considering production $X \rightarrow Xab$ we get the relations as

trailing(X) $\gtrdot a$ $\Rightarrow \{a,b\} \lessdot a$

Applying rule 4 and 5 we get

$ ⋖ leading(S) ⇒ $ ⋖ {a,b}

Trailing(S) ⋗$ ⇒ {a,b} ⋗ $

Operator precedence table for the given grammar is shown in Table 5.8.

Table 5.8 Precedence Relation Table

	a	**b**	**$**
a	≐,⋖,⋗	⋖	⋗
b	⋗,⋖		⋗
$	⋖	⋖	

Note: This grammar is ambiguous grammar as the parser has options to both shift and to reduce.

5.5.7 Error Recovery in Operator Precedence Parser

Blank entries in any parsing table refer to errors. So for each blank entry, have a pointer to subroutine. So that whenever a blank entry is referred, the corresponding subroutine is called. For writing error recovery routines, the compiler designer should have good knowledge about all possible errors. For example, look at the following Table 5.9. e1, e2, e3, and e4 are error recovery routines. They specify how to recover from the error.

Table 5.9 Error Recovery Routines in Precedence Relation Table

	id	**(**	**)**	**$**
id	e1	e1	⋗	⋗
(	⋖	⋖	⋗	e4
)	e1	e1	-	⋗
$	⋖	⋖	e2	e3

Here e1( ) tells that operator is missing between the terminals. So error recovery here can be to insert an operator and issue error message.

e2( ) tells that expression is starting with). So error recovery here can be to delete input symbol and issue an error message.

e3( ) tells that input expression is missing. So error recovery here can be to issue an error message.

e4( ) tells that expression is ending with (. So error recovery here can be to pop the input symbol and issue an error message.

The advantage of operator precedence parser is that it is simple to construct and can be used for parsing expression grammar. It works by considering precedence among operators.

Hence, if there is an operator that has different precedence (e.g., unary –), it cannot be handled by this parser. To solve the problem at lexical analysis itself, separate them into two different tokens instead of a single token of two precedence.

Another difficulty with operator precedence parser is that, if operators supported in grammar are more, more space is required for parsing table (as it is n * n table where n is the number of operators/terminals). So to save space, a precedence relation table is converted into another table called the precedence function table.

For example, consider the precedence relation Table 5.10.

Table 5.10 Precedence Relation Table

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

It can be reduced to a precedence function table as shown below in Table 5.11.

Table 5.11 Precedence Function Table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Here two integer mapping functions "f" and "g" are used for reducing each precedence relation into a numerical value. The actual relation is given by numerical comparison between the two entries.

For example,

If we want precedence relation between * and +.

"*" is a row element, function "f" is used for mapping row values, so take f(*) = 4.

Now "+" is a column element, function "g" is used for mapping column values, so take g(+) = 1.

f(*) > g(+) hence relation is * >+.

If we want precedence relation between + and id.

"+" is a row element, function "f" is used for mapping row values, so take f(+) = 2.

Now "id" is a column element, function "g" is used for mapping column values, so take g(id) = 5.

f(+) < g(id); hence relation is + < id.

5.5.8 Procedure for Converting Precedence Relation Table to Precedence Function Table

Input is precedence relational table.

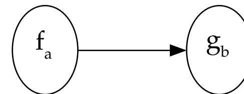
Step 1: For every terminal, create two symbols f_a and g_a.
Step 2: Divide the symbols into groups by using ≐ relation.

For example, a ≐ b then f_a and g_b are combined into one group.

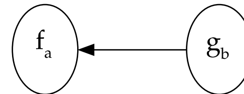
In addition to the above relation, a ≐ c, then f_a, g_b, g_c are combined into one group.

Step 3: Create a digraph with groups in Step 2 as nodes and edges given by < or >.

For example, a > b then add edge as



For example, a < b then add edge as



Add an edge for each < or > relation in the table.

Step 4: Check the digraph for cycles. If there are any cycles, stop the procedure and conclude that the table cannot be converted to a function table.

Step 5: If there are no cycles, that is, if digraph is acyclic, then length of the longest path from f_a gives f(a) for each terminal a.

Example 7: Convert the following relation Table 5.12 to function table.

Table 5.12 Precedence Relation Table

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

Solution: There are four terminals id,*,+, and $; hence, create 8 symbols f_{id}, g_{id}, f_+, g_+, f_*, g_*, $f_\$$ and $g_\$$. As there are no ≐ relations, each symbol is treated as a separate group. Now construct digraph as shown in Figure 5.3 with each symbol as a node, that is, 8 nodes as follows.

To get the precedence function f(id), start from node f_{id}, traverse all possible paths from f_{id}. The path must start with f_{id} and can end anywhere (need not be f$ or g$).

For example, from node f_{id} there are four possible paths as follows:

$f_{id} - g_\$$ where path length is 2
$f_{id} - g_* - f_+ - g_\$$ where path length is 3
fid – g* –f+ – g+ –f$ where path length is 4
fid – g+ –f$ where path length is 3

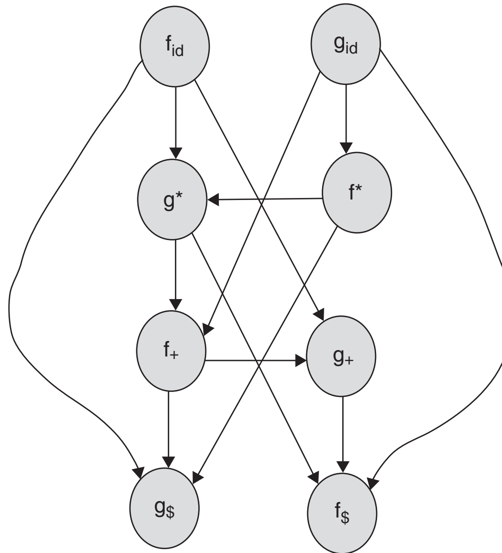


Figure 5.3 Digraph for the Precedence Relation Table

Longest is 4 hence f(id) is 4.
Hence the resulting table is shown below in Table 5.13.

Table 5.13 Precedence Function Table

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

5.6 LR Grammar

A grammar for which we can construct LR parser is called *LR grammar*. A grammar that can be parsed by a parser after examining k input symbols on each move is called *LR(k) grammar*. There is a difference between LL and LR grammar. For a grammar to be LR(k) we must be able to recognize the right side of the production having seen from what is derived with k look ahead symbols. In LL(k), the task is simple. It should recognize the use of a production by seeing only the first k input symbol. LR parsers can describe more languages than LL parsers.

5.7 LR Parsers

LR parsers are the most efficient bottom-up parsers and can be implemented for almost any programming language. The class of grammars that can be parsed using LR parsers

is a proper superset of the class of grammars that can be parsed with predictive parsers. It is difficult to write/trace LR parsers by hand. Usually a parser generator like yacc (bison) is required to write LR parsers. With such tools, one can write context free grammar and have the generator automatically produce a parser for the grammar. In LR(k) parsing, the first L stands for left-to-right scan of the input buffer, the second R stands for a right-most derivation in reverse, and k stands for the maximum of lookaheads used for taking parsing decisions. If k is omitted, it is assumed to be 1. *To construct an LR parser, there is no restriction on the grammar. Unlike LL(1), we do not have to worry about left recursion and left factoring, etc.* Even if it is left recursive, you still can start construction without eliminating it.

The LR parser is a bottom-up parser that makes use of DFA. The DFA is used to recognize the set of all viable prefixes by reading the stack from bottom to top. It determines what handle is available. A viable prefix of a right sentential form is a prefix that contains a handle, but no symbol to the right of the handle. Therefore, if a DFA that recognizes viable prefixes is constructed, it can be used to guide the handle selection in the bottom-up parser. Let us understand how an LR parser works.

The LR parser uses a DFA that recognizes viable prefixes to guide the selection of handles. Hence, it must keep track of the states of the DFA. That is why the LR parser stack contains two types of symbols: state symbols used to identify the states of the DFA and grammar symbols. The state symbol on top of stack contains all the information it needs. The parser starts with the initial state of the DFA, I_0, on the stack. The parser operates by looking at the next input symbol "a" and state symbol I_i on top of the stack. If there is a transition from the state I_i on "a" in the DFA going to state I_j, then it shifts the symbol "a" followed by the state symbol "j" onto the stack. If there is no transition from the state Ii on input symbol "a" in the DFA and if the state Ii on the top of stack recognizes a viable prefix that contains the handle $S \rightarrow \alpha$, then the parser carries out the reduce action by popping off α and pushing S onto the stack. This is equivalent to making a backward transition from state Ii on α in the finite automata and then making a forward transition on A. Every shift action of the parser corresponds to a transition on terminal symbol in the DFA. Therefore, the current state of the DFA and the next input symbol determine whether the parser performs a shifts action or reduce action. Now let us define the LR parsing algorithm. An LR parser or an SR parser mainly performs two actions at any time: shift or reduce until it halts. Halting can be either on successful completion or on an error.

5.8 LR Parsing Algorithm

5.8.1 Task of LR Parser: Detect Handle and Reduce Handle

An LR parser shown in Figure 5.4 uses a stack and input buffer and parsing table. The parsing table is divided into two parts—action and goto. The rows in parsing table correspond to states of DFA that is used in recognizing the handle. What action an LR parser is supposed to take at any state is defined under the action part. The entries in action part would be shift given by S_i or reduce r_i, or accept. The Goto part contains state number under the nonterminals. Try to understand the structure of the LR parsing table. We shall discuss the procedure for construction later. Look at the example of LR parsing Table 5.14 shown below.

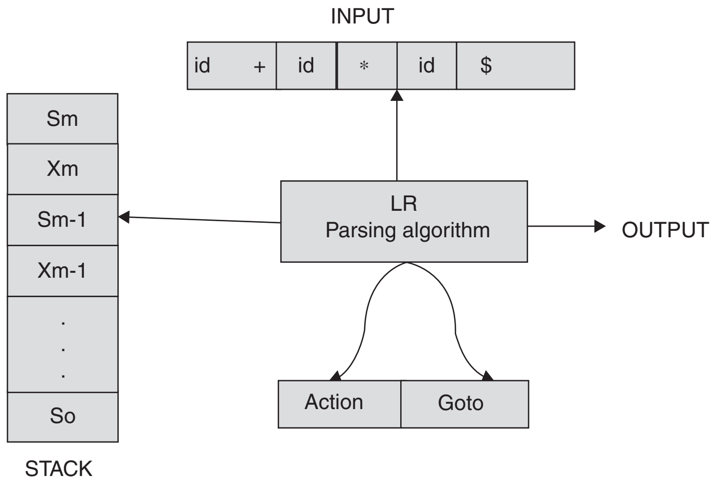


Figure 5.4 Model of LR Parser

Table 5.14 LR Parsing Table

State	Action			Goto	
	a	b	$	A	B
0	S_3			1	2
1		r_2	accept		
2	S_2			3	
3		S_1			
4	r_1				4

Stack is used to store grammar symbols surrounded by state symbols like $s_m a_m s_{m-1} X_{m-1} s_{m-2} a_{m-2} \ldots s_0$ where $s_m\ s_{m-1} s_{m-2} \ldots s_0$ are state symbols and $a_m..X_{m-1}..a_{m-2}$ are grammar symbols. To take the parsing decisions, always the top of the stack should be a state because rows in parsing table are defined with states. So to start with, the LR parser pushes the start state on to the stack.

Given a grammar and input string to be parsed, the first parser constructs a parsing table. The input string to be parsed is read into the input buffer. An end marker "$" is appended at the end. Initially "$" is pushed on to the stack; this is to identify the bottom of the stack. It then pushes the start state of DFA, that is, 0 onto the stack. The input pointer "ip" points to the first symbol in the input buffer.

Parsing algorithm:

Let "X" be the state on top of the stack and "a" be the symbol pointed by "ip."

Action part of parsing table is used to take parsing decisions as follows.

Repeat for ever

1. If action[X,a] = S_i, then *(Shift action)*
   Push "a" then state I onto stack. Advance the input pointer to the next symbol.
2. If action[X,a] = r_i, then *(Reduce action)*
   {
   Take r_i, that is, the ith production from the grammar. Let r_i is $A \rightarrow \beta$.
   Pop 2 * $|\beta|$ symbols from stack and replace them by nonterminal A.
   If Xi is the state below the nonterminal A, then push
   goto[Xi,A] onto the stack. Output the production $A \rightarrow \beta$
   }
   Now continue parsing.
3. If action[X,a] = accept, then *(Successful completion)*

All the LR parsers use the same parsing algorithm. Let us understand the algorithm with an example.

Example 8: Parse the input string "aabb" using the grammar

$S \rightarrow AA, A \rightarrow aA \mid b$

Solution: The LR parsing is shown in Table 5.15 using the given grammar.

Table 5.15 SLR(1) Parsing Table

State	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3	8	
5			r_1		
6	r_2	r_2	r_2		

Parser compares the top stack state "s" with the look ahead symbol "a." It refers to action part of parsing table, and performs action defined under terminal "a" and state "s."

The set of actions performed by the parser is shown in Table 5.16.

All the four LR parsers—LR(0), SLR(1), LALR(1), and CLR(1)—use the algorithm mentioned above for parsing. They differ only in construction of the parsing table. Let us look at the parsing table construction. Most of the procedure for the construction of the parsing table is common for all the four parsers. Hence, we first discuss the common procedure, then we take up specific parser design separately. The general procedure for all four LR parsers parsing table construction is as follows:

Table 5.16 LR Parsing Actions

Stack	Input	Action performed
$	aabb$	Push state 0
$0	aabb$	S_3: shift "a" then 3, increment ip.
$0a3	abb$	S_3: shift "a" then 3, increment ip.
$0a3a3	bb$	S_4: shift "b" then 4, increment ip.
$0a3a3b4	b$	reduce by r_3, i.e., $A \rightarrow b$.pop 2 symbol, replace by A, push goto[3,A] = 6
$0a3a3A6	b$	reduce by r_2, i.e., $A \rightarrow aA$.pop 4 symbol, replace by A, push goto[3,A] = 6
$0a3A6	$	reduce by r_2, i.e., $A \rightarrow aA$.pop 4 symbol, replace by A, push goto[0,A] = 2
$0A2	$	S_4: shift "b" then 4, increment ip.
$0A2b4	$	reduce by r_3, i.e., $A \rightarrow b$.pop 2 symbol, replace by A, push goto[2,A] = 5
$0A2A5	$	reduce by r_1, i.e., $S \rightarrow AA$.pop 4 symbol, replace by A, push goto[0,S] = 1
$0S1		accept

5.9 Construction of the LR Parsing Table

1. Given grammar, take augmented grammar.
2. Create canonical collection of LR items.
3. Draw DFA and prepare table.

Here the term LR items are different for different types of the LR parser.

LR(0), SLR(1) } **LR(0)items**

LALR(1), CLR(1) } **LR(1)items**

If you want to define the construction of the SLR(1)/LR(0) parsing table, in the above procedure, change the word "LR items" by LR(0) items.

1. Given grammar, take augmented grammar.
2. Create canonical collection of LR(0)items.
3. Draw DFA with the set of items and the prepare table.

If it is CLR(1) /LALR(1), change the word "LR items" by "LR(1) items."

1. Given grammar, take augmented grammar.
2. Create canonical collection of LR(1)items.
3. Draw DFA and the prepare table.

Since most of the procedure is common, let us look at the common procedure step by step.

5.9.1 Augmented Grammar

Given a grammar "G" with "S" as the start symbol, if we want to change start symbol "S" to "**S**," then add a rule to the grammar as **S″ → S.** The grammar with the newly augmented production is called augmented grammar. So writing augmented grammar is easy. But a question may rise—what is the use for augmented production. This rule helps the parser to understand when to stop parsing and announce the successful completion of the process. LR parsing is a bottom-up parsing where it starts with the input string, performs a series of reductions until the string is reduced to start symbol. So ultimate reduction is reducing by start symbol. If we take reducing by the start symbol as final reduction, sometimes there is confusion for the parser because of two reasons. (1) The start symbol may appear on the right hand side of any rule also because of which reduction by S can be used in between not only at the end. (2) There is no restriction that the start symbol should be defined with only one string. It can have more than one alternative definition; in such a case, there is a dilemma in which one to consider final reduction. That's why whether the start symbol has one or more rules or is occurring on the right hand side of any rule, there is no loss in appending augmented production like S″ → S, which unambiguously determines when to stop parsing and announce successful completion.

Hence, given a grammar "G" with "E" as start symbol, write the augmented grammar as G together with E″ → E**.**

Here, we first discuss the construction of LR(0) and SLR(1). The next step in parsing table construction is creating canonical collection of LR(0) items. First let us look at what is LR(0) item or item in simple.

5.9.2 LR(0) Item

A production with "dot" at any point on the right hand side of a rule is called LR(0) item. So with "dot," a production is called LR(0) item and without dot as production. "dot" makes the difference between the two. So let us understand the significance of "dot."

A bottom-up parser, starts with the input string and reads symbol by symbol from left to right until a handle is detected; once handle is found, it reduces. Handle is nothing but the right hand side of any rule. Until complete handle is read, reduction cannot be performed. So "dot" tells us how far the right hand side is seen by the parser at any time.

For example, if there is a production X → abc. Here handle is "abc." Parser will not read "abc" at once. It first reads "a," which will be indicated by item X → a•bc. Once it reads b then it is indicated by item X → a•bc. So the sequence of items generated while recognizing handle "abc" is as follows:

X → •abc ………………not yet read anything
X → a•bc ……………..read "a" yet to read "bc"
X → ab•c ……………..read "b" yet to read "c"
X → abc• ……………..read "c" ready for reducing.

An item with "•" at the end is called complete item/final item. Similarly, a production S → ABC yields items as follows:

$S \to \bullet ABC$
$S \to A \bullet BC$
$S \to AB \bullet C$
$S \to ABC \bullet$

The production $A \to \varepsilon$ generates final item $A \to \bullet$. An item can be represented by a pair of integers (a,b) where a is the number of the production and b is the position of the dot.

We have so far studied what is item or a set of items; now, let us see how to construct the LR parser. For constructing the LR(0) parser or SLR(1) parser, we have to create canonical collection of LR(0) items. To create canonical collection of LR(0) items, we use two functions—Closure(0) and "Goto()." Let us understand these functions.

5.9.3 Closure(I)

I is set of LR(0) items like $A \to \alpha \bullet B\beta$, $C \to a\bullet$. The function closure takes input as a set of items and produces set of LR(0) as output. The function is evaluated using the following two rules.

1. Initially add every item from input to output.
2. If $A \to \alpha \bullet B\beta$ is in I where B is nonterminal & $B \to \gamma$ is the rule for B, then add $B \to \bullet\gamma$ to Closure (I). Repeat this for every newly added item.

To evaluate Closure (I), we first add all items from input I to output. With this we get the initial items. In these initial items, verify the grammar symbol that follows "•." If it is a terminal, there is no need to add any new items. But if it is a nonterminal, then take the production for nonterminal and add it to Closure(I) with dot at the beginning.

Example 9: Find Closure ($E'' \to \bullet E$) for the following grammar:

$E'' \to E$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to id \mid (E)$

Solution: First add $E'' \to \bullet E$. In the newly added item, the grammar symbol that follows dot is a nonterminal, ie., E; so take the rules for E and add dot at the beginning.

We get

$E'' \to \bullet E$
$E \to \bullet E + T \mid \bullet T$

Now repeat rule 2 for the newly added item. "T" is a nonterminal, hence add rule for T with dot at the beginning.

$E'' \to \bullet E$
$E \to \bullet E + T \mid \bullet T$
$T \to \bullet T * F \mid \bullet F$

Now repeat rule 2 for the newly added item. "F" is a nonterminal, hence add rule for F with dot at the beginning.

$E'' \rightarrow \bullet E$
$E \rightarrow \bullet E + T \mid \bullet T$
$T \rightarrow \bullet T * F \mid \bullet F$
$F \rightarrow \bullet id \mid \bullet (E)$ This is closure($E'' \rightarrow \bullet E$).

5.9.4 Goto(I,X)

Where I is a set of item and X is the grammar symbol.

The Goto function basically defines what is the transition/change in the set of items I on seeing the grammar symbol X. This is formally defined as

goto(I,X) = Closure({A → aX•b | A → a•Xb is in I}),

that is, closure of all the productions of the form $A \rightarrow aX\bullet b$ such that $A \rightarrow a\bullet Xb$ is in I.

If $I = A \rightarrow a\bullet Xb$, then goto($A \rightarrow a\bullet Xb$,X) is, first find transition in I on X. The grammar symbol X matches with the symbol that follows dot in I, hence change is new item where dot is moved next to X, that is, $A \rightarrow aX\bullet b$. Now find the closure of such resulting items. Suppose I is **A → aX•A then goto(A → aX•A,X) =** ϕ, that is, no new item is generated. Because the grammar symbol that follows dot in item is A and argument is X, both do not match. Hence, no new item is produced. The evaluation of goto function involves two steps:

1. Find transition
2. Apply Closure() on resulting items.

Consider the following example:

Example 10: If I = {$E'' \rightarrow \bullet E$, $E \rightarrow E \bullet + T$} and G is
$E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow id \mid (E)$
Find goto (I, +).

Solution: First find transition on seeing the symbol "+" in each item. In the first item "$E'' \rightarrow \bullet E$," there is no match with "+." In second item, that is, "$E \rightarrow E \bullet + T$" there is a match with "+"; so the result is $E \rightarrow E + \bullet T$. Now apply closure on this. "T" is a nonterminal; so add rules for "T" with dot at the beginning. The result therefore is as follows:

I_1: $E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F \mid \bullet F$
$F \rightarrow \bullet id \mid \bullet (E)$

Let us see one more example. Find goto(I_1,T).

First find transition on seeing "T" in each item. In the first item "$E \rightarrow E + \bullet T$," there is a match so result is $E \rightarrow E + T\bullet$. In second the item, that is, "$T \rightarrow \bullet T * F$" there is a match; so result is $T \rightarrow T\bullet * F$. So after finding the transition, we get items as follows:

$E \rightarrow E + T\bullet$
$T \rightarrow T \bullet * F$

Now apply closure on this. "*" is a terminal and so there is no need to add rules. The result therefore is as follows

$E \rightarrow E + T\bullet$
$T \rightarrow T \bullet * F$

As we have understood closure and goto functions, let us discuss how to create canonical collection of LR(0) items by using these two functions.

5.9.5 Creating Canonical Collection "C" of LR(0) Items

$C = \{I_0, I_1, I_2, I_3 \ldots . I_n\}$

1. The initial item in C is I_0 = closure(augmented production with dot at the beginning). For example, in the above grammar, $I_0 = \text{closure}(E'' \rightarrow \bullet E)$
2. For each I_i in C and each grammar symbol X in G
   Repeat

      while goto(I,X) is not in set C and not a empty set

      add goto(I,X) to C

      until no more new set of items can be added to set C.

The initial item in C is obtained using the closure function. The remaining elements in set C are obtained by the goto function.

Example 11: Construct canonical collection of LR(0) items for grammar.

$S'' \rightarrow S,\ S \rightarrow AA,\ A \rightarrow aA \mid b$

Solution: $I_0 = \text{closure}(S'' \rightarrow \bullet\ S,) = S'' \rightarrow \bullet\ S$

$\mathbf{S \rightarrow \bullet\ AA}$

$\mathbf{A \rightarrow \bullet\ aA \mid \bullet\ B}$

$I_1 = \text{Goto}(I_0,S) = S'' \rightarrow S\bullet$

$I_2 = \text{Goto}(I_0,A) = \mathbf{S \rightarrow A \bullet A}$

$\mathbf{A \rightarrow \bullet\ aA \mid \bullet b}$

$I_3 = \text{Goto}(I_0,a) = \mathbf{A \rightarrow a \bullet A}$

$\mathbf{A \rightarrow \bullet\ aA \mid \bullet\ b}$

$I_4 = \text{Goto}(I_0,b) = \mathbf{A \rightarrow b\ \bullet}$

That's all the transitions from I_0. Now apply the goto function on I_1. Since I_1 has only one final item, there is no need to apply the goto function. Now consider I_2

$I_5 = \text{Goto}(I_2,A) = \mathbf{S \rightarrow AA\ \bullet}$

$\text{Goto}(I_2,a) = I_3,\ \text{Goto}(I_2,b) = I_4$

$I_6 = \text{Goto}(I_3,A) = \mathbf{A \rightarrow aA\ \bullet}$

$\text{Goto}(I_3,a) = I_3,\ \text{Goto}(I_3,b) = I_4$

As I_4, I_5 and I_6 have only final items, this completes canonical collection.

5.9.6 Construction of DFA with a Set of Items

Now after creating the canonical collection, we draw DFA with a set of items as final states—I_0 as the initial state and edges as goto transitions. This deterministic finite automaton recognizes the viable prefixes of G. The state symbol stored on top of the stack is the state the handle recognizing finite automaton would be in if it had read the grammar symbols of the stack from bottom to top. From the canonical collection in the above Example 11, we can draw the DFA as shown in Figure 5.5.

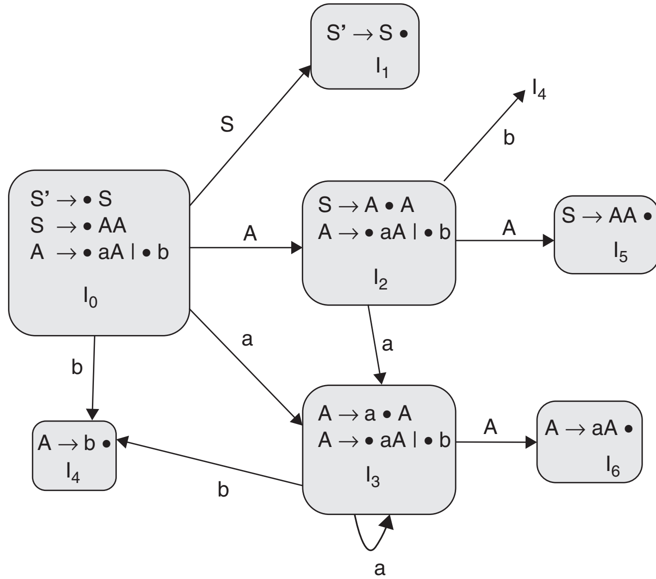
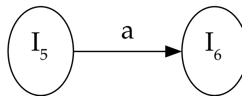


Figure 5.5 DFA for Grammar in Example 11

If each state, I_i of DFA is a final state and I_0 is the initial state, then DFA recognizes exactly the viable prefixes of grammar.

The final step in the construction of the LR parsing table is preparing the table from DFA. To prepare the LR parsing table, we first need to know the number of rows and columns. For each state "I_i" in DFA, we define a row "i" in the parsing table. For example, consider the above DFA, for that the number of rows in LR table, there are 7 as there are 7 states I_0 to I_6. Columns are given by terminal in the action part and nonterminals in the goto part. One additional column is defined for "$" in the action part. For the above example, the LR table will have 7 rows, that is, 0 to 6 and 5 columns, that is, a, b, $, S, A. Coming to the table preparation, we need to know how to enter shift entries, goto states, and reduce entries. All the parsers basically differ only in placing reduce entries. The remaining procedure is the same, that is, placing shift entries and goto states are also common for all the parsers. Now let us see how to prepare shift entries and goto information using DFA. In the DFA, transition in any state on terminals yields to shift entries and nonterminal yields to the goto state.

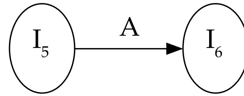
Consider the following transition on terminal 'a',



The corresponding shift entry is

	a
5	S_6

Consider the following transition on nonterminal A,



The corresponding goto state entry is

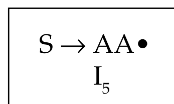
	A
5	6

The procedure we have discussed thus far is the common procedure used for all LR parsers except item type. To complete the table the left out part is reduce entries, this is where all the four parsers differ. Hence, let us consider each parser separately for discussing the preparation of reduce entries. Let us start with the simplest of all, that is, LR(0) parser.

5.10 LR(0) Parser

The main characteristic of this parser is that it does not use any look ahead in making the parsing decisions. If you recollect, in LR parsing, parsing decisions (when to go for shift action/reduce action) are made by looking at the top stack state and the look ahead symbol. In LR(0) parser, the top stack state unambiguously determines the action to be performed at anytime without looking at the look ahead symbol.

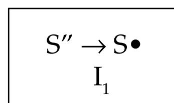
To enter the reduce entries, once again DFA is used by the parser. In the DFA, it checks for states that contain the final item (an item with dot at the right end). The final item indicates that the entire right hand side is seen by the parser; hence, it is ready for reduce action. Hence, it checks for states that contains final items. If a state I_i contains the final item, then place the reduce entries in the corresponding row "i." If state Ii has a final item $A \rightarrow \alpha\bullet$ the productions in given grammar is numbered. If the rule $A \rightarrow \alpha$ is j^{th} rule, then in the row "i" place the reduce entry as $\mathbf{r}_j$ under all the columns because the columns in the table are look ahead symbols. LR(0) makes parsing decisions independent of look ahead. Hence, place the reduce entry under every column in the action part. For example, consider the following state of DFA:



The corresponding reduce entry in table is

	a	**b**	**$**
5	$\mathbf{r}_1$	$\mathbf{r}_1$	$\mathbf{r}_1$

If the final item is $S'' \rightarrow S\bullet$, then place the entry as "acc" instead of reduce. Because reducing by $S'' \rightarrow S\bullet$ is the accepting state. For example, consider the following state of DFA:



The corresponding reduce entry in table is

	a	b	$
1	acc	acc	acc

or

	a	b	$
1	acc		

When the top of the stack is state 1, without looking at the input symbol it announces successful completion.

Example 12: Consider the grammar:

1. S → (L)
2. S → a
3. L → S
4. L → L, S

Prepare the LR(0) parsing table for the above grammar.

Solution: Step 1: Take the augmented grammar as follows:

S″ → S, S → (L),S → a
L → S, L → L, S

Step 2: Create canonical collection and draw DFA. DFA is shown in Figure 5.6.
Step 3: Table is prepared and is shown in Table 5.17.

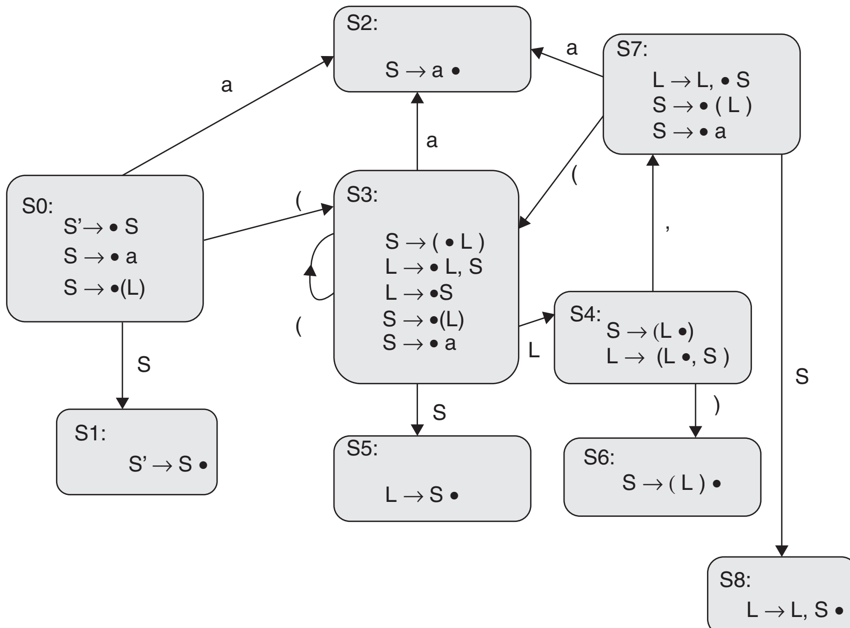


Figure 5.6 DFA for Grammar in Example 12

Table 5.17 LR Parsing Table for Example 12

States	Action					Goto	
	(	)	a	,	$	S	L
0	s_3		s_2			1	
1	Accepting state						
2	r_2	r_2	r_2	r_2	r_2		
3	s_3		s_2			5	4
4		s_6		s_7			
5	r_3	r_3	r_3	r_3	r_3		
6	r_1	r_1	r_1	r_1	r_1		
7	s_3		s_2			8	
8	r_4	r_4	r_4	r_4	r_4		

5.10.1 Advantages of the LR(0) Parser

- It is very simple to construct the LR(0) Parser compared to other LR parsers.
- Each row in the table defines unique action, that is, either shift action or reduce action or accept.

5.10.2 Disadvantages of the LR(0) Parser

- The LR(0) Parser can be used to parse a small class of grammars.
- Look ahead is not used in making parsing decisions.

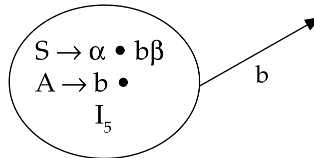
5.10.3 LR(0) Grammar

The grammar for which there are no multiple entries in the LR(0) parsing table is called LR(0) Grammar. An LR(0) grammar indicates that this grammar is suitable for constructing the LR(0) parser.

Given a grammar, let us see how to check whether it is LR(0) or not. The straightforward method is to construct the LR(0) parsing table for the grammar and check whether there are any multiple entries in the table. No multiple entry in the table indicates that the grammar is LR(0). For example, if grammar has 100 nonterminals and 100 terminals, then to check whether it is LR(0) or not, we need to construct a table with 100 × 100, which is a tedious task. Hence, let us see a simple way of checking whether the grammar is LR(0) or not. The basic criterion used is checking the multiple entries in the table. Let us see how to check the possibility of multiple entries without constructing the table.

5.10.4 Conflicts in Shift-Reduce Parsing

After constructing the DFA, check if any state contains a combination of final and nonfinal item. For example, assume that state I_5 has items as follows.



If we prepare the corresponding parsing table entries, we get a table as shown below:

	a	B	$
5	r	s/r	r

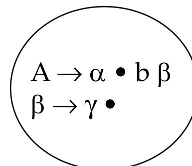
On seeing the input symbol "b," there is a conflict. The nonfinal item in state I_5 indicates that on seeing "b" it may enter to some other state, which may lead to shift action. At the same time, the final item in the same state indicates reduce action. So the same state is indicating two different actions—shift/reduce, that confuses the parser. This conflict is called **shift/reduce conflict**. Here, the parser is not able to decide whether to shift or to reduce. Hence, by checking out shift/reduce conflicts, we can check for multiple entries in the table.

For example, the following LR(0) items

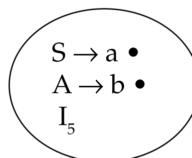
$A \to \alpha \bullet A\beta$

$B \to \gamma \bullet$ do not have an SR conflict because there is no shift action. The symbol next to dot in nonfinal item is "A" which is a nonterminal. On seeing "A," it enters to another state that corresponds to the goto state but not shift action. There is only reduce action. Hence, no SR conflict at all. So to test SR conflict we test each state of DFA as follows:

If there is a combination of final and nonfinal items where grammar symbol next to dot in nonfinal item is a terminal such as



Another way of checking the possibility of multiple entries without constructing the table is as follows. After constructing the DFA, check if any state contains a combination of more than one final item. For example assume that state I_5 has items as follows:



If we prepare the corresponding parsing table entries, we get a table as shown below:

Let r_1: $S \to a$ and r_2: $A \to b$

	a	b	$
5	r_1/r_2	r_1/r_2	r_1/r_2

There are two final items in the state I_5. A final item indicates reduce action. So same state is indicating two different reduce actions, that confuses the parser. This conflict is called reduce/reduce conflict. Here, the parser is not able to decide whether to reduce by r_1 or r_2. Hence, by checking out reduce/reduce conflicts, we can check for multiple entries in the table. To test RR conflict we test each state of DFA as: *If there is a combination of more than one final item in any state.*

There are two kinds of conflicts:

Shift/reduce conflict: Here, the parser is not able to decide whether to shift or to reduce.
Reduce/reduce conflict: Here, the parser cannot decide which sentential form to use for reduction.

To verify whether the given grammar is LR(0) or not, check each state of DFA for shift/reduce or reduce/reduce conflicts. The necessary condition for conflicts is there should be at least one final item. In any state of DFA, with the final item if there is one more nonfinal item (where grammar symbol next to dot must be terminal), then it is shift/reduce conflict. In any state of DFA, if there is more than one nonfinal item then it is reduce/reduce conflict.

Example 13: Check whether the following grammar is LR(0)

S → AA, A → aA | b

Solution: Look at the DFA constructed for the above grammar in the previous example. Check each state for conflicts. None of the states has a final item along with final item or nonfinal item; hence, no conflicts. So the grammar is LR(0).

Example 14: Check whether the following grammar is LR(0)

S→ **a | A, A → a**

Solution: Take augmented grammar G″ as follows:

S″→S, S→ **a | A, A → a**

Draw the DFA with LR(0) items by using the procedure described in 5.9.6.

State I_2 has reduce-reduce conflict as shown in Figure 5.7. On reading input symbol "a," it may not be able to decide whether to reduce by first rule or second rule. Hence, it is not LR(0).

In fact, the grammar is ambiguous grammar. ***No ambiguous grammar can be LR(0).***

Example 15: Check whether the following grammar is LR(0)

E → **E + T | T, T → a**

Solution: Take augmented grammar G″ as follows:

E″ → E, E→ **E + T | T, T → a**

Draw the DFA with LR(0) items by using the procedure described in 5.9.6.

Look at Figure 5.8. Does state I_2 has shift-reduce conflict? No, this is not shift reduce conflict. A shift-reduce conflict means same state specifying different actions. In state I_2, the

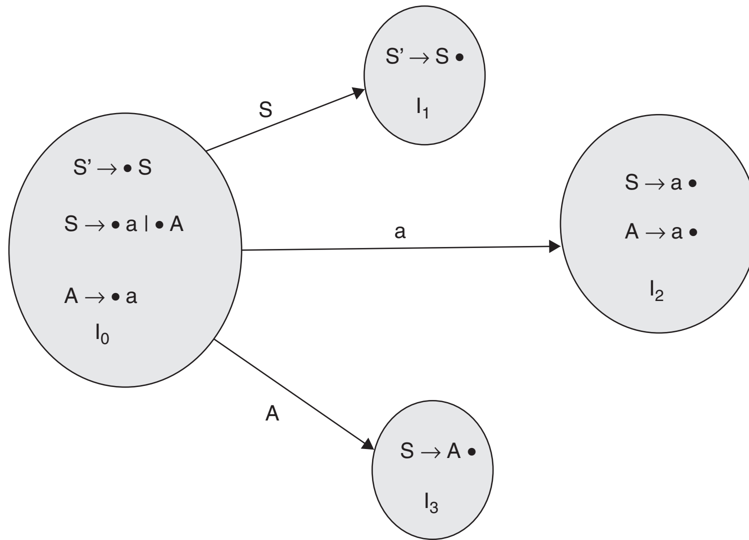


Figure 5.7 RR Conflicts in DFA

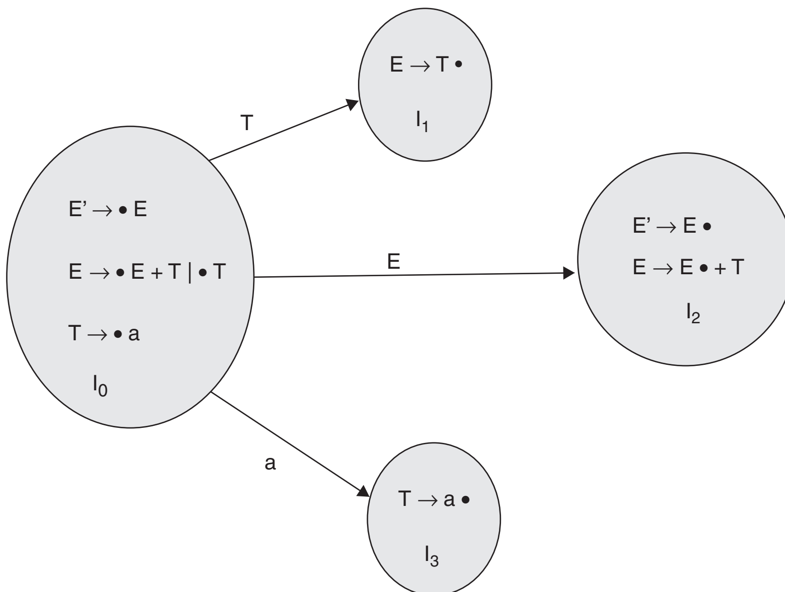


Figure 5.8 Confusion in State I_2 for the LR Parser

nonfinal item is specifying the shift action. But the final item is not for reduce action. It indicates accept action. There are only shift-reduce or reduce–reduce conflict; accepting state itself should not be a confusion. Remember augmented production is not present in original grammar. This is what we have added. So to avoid such confusion with augmented production, we can append a special symbol at the right end of augmented production like **E″ → E#**. Now let us construct DFA with this. It is shown in Figure 5.9.

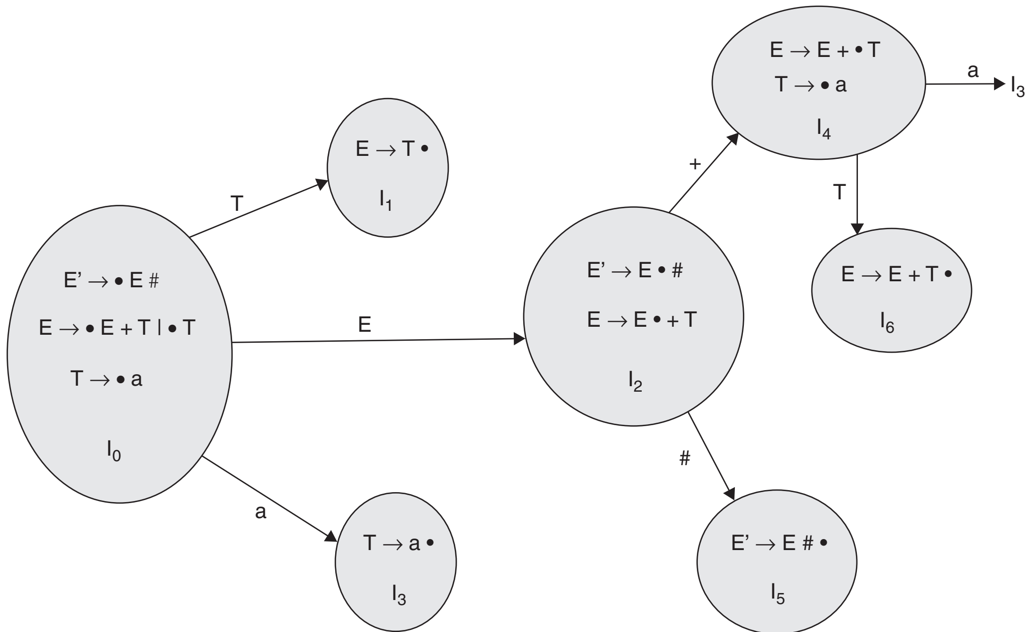


Figure 5.9 DFA with LR(0) Items for Grammar in Example 15

Now there are no conflicts. Hence, it is LR(0).

Now let us understand the power of the LR(0) parser. Consider a simple unambiguous grammar given below. Here the difference between previous example and this grammar is, the operator "+" is right associative.

Example 16: Check whether the following grammar is LR(0) or not

E → **T + E | T, T → a**

Solution: Take augmented grammar G″ as follows:

E″ → E, E→ **T + E | T, T → a**

Draw the DFA with LR(0) items by using the procedure described in 5.9.

Look at Figure 5.10. State S_2 has shift-reduce conflict. In this state it may not be able to decide whether to shift or reduce. Hence, it is not LR(0).

Remember, to avoid such conflict, if you add "#" in augmented production as added in previous example, the conflict will not be resolved. Here confusion is not because of augmented production. It is in the original grammar. The above example demonstrates the

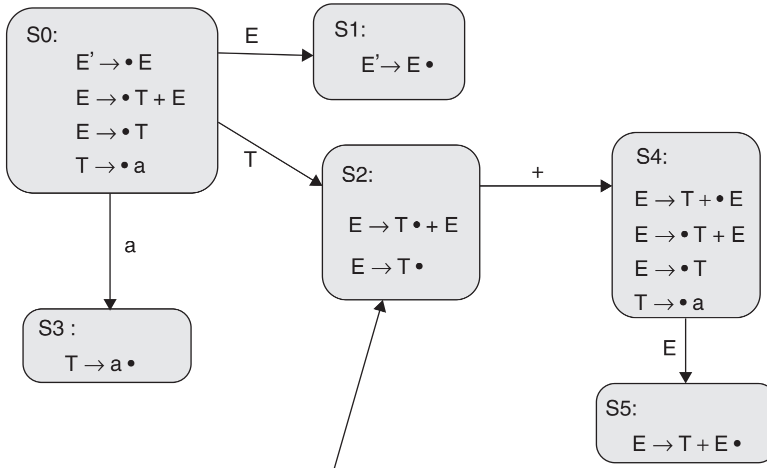


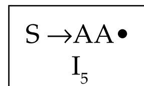
Figure 5.10 SR Conflicts in DFA of Grammar in Example 16

power of the LR(0) parser. Such a simple unambiguous grammar cannot be parsed by the parser. LR(0) can parse only a very small class of grammars.

Now let us examine why LR(0) is failing here. The main drawback of LR(0) is "0" that is, it does not use any lookaheads (0 look aheads) in making parsing decisions. Hence, while constructing the parsing table, if a state has a final item, we place reduce action under every column. One entry is already placed in each column because of which the possibility of multiple entries will be increased. Hence, the LR(0) parser can parse only few grammars. To overcome this disadvantage, the better parser, that is, simple LR parser, SLR(1) is constructed with one look ahead symbol.

5.11 SLR(1) Parser

The SLR(1) parser is a simple LR parser, which is easy to construct. This is better than LR(0) as it uses a look ahead symbol. The "1" in SLR(1) indicates the number of lookaheads used by the parser. It uses a look ahead given by the follow set. The procedure for SLR(1) parsing table is the same as LR(0); the only difference is in reduce entries. To place reduce entries once again, SLR(1) uses the DFA. It checks if a state has the final item. The state that contains a final item indicates in which row the reduce entries are to be placed. This procedure is the same as LR(0). For example, if state I_i has a final item, we place reduce entries in row "i." But in row "i," finding out columns is different for SLR(1). If it is ***LR(0), we place under every column, but for SLR(1) it is under the columns given by the follow set.*** For example, consider the following state of DFA:



Assume follow(S) is {$}

The corresponding reduce entry in table is

	State	a	+	$
LR(0) parser	5	r_1	r_1	r_1
SLR(1) parser	5			r_1

SLR(1) determines the columns by follow of the left hand side nonterminal. To understand how SLR(1) is better than LR(0), consider the above example where LR(0) is failing. Let us construct the SLR(1) parsing table for the same grammar. E′→ E, E → **T + E | T, T → a** (Figure 5.11).

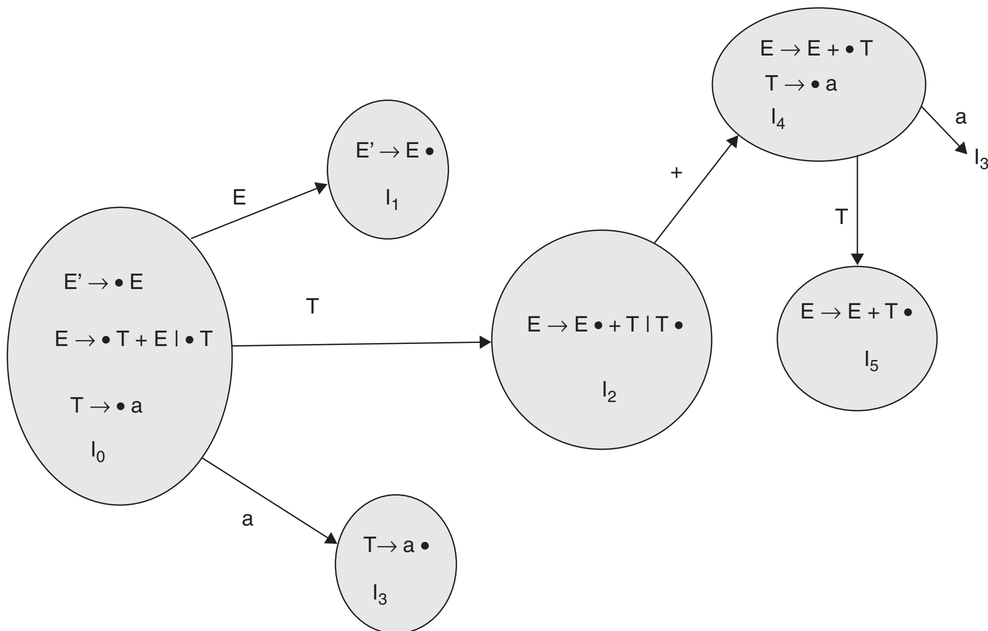


Figure 5.11 DFA for Grammar in Example 16

Consider state 2, the entry in table is as follows

	State	a	+	$
LR(0) parser	2	r_2	s_4/r_2	r_2
SLR(1) parser	2		s_4	r_2

There is a Shift-Reduce (SR) conflict in state I_2. If this conflict is resolved by SLR(1) parser, then the grammar is SLR(1). To verify if the conflict is resolved or not, let us construct the SLR(1) parsing table.

Follow(**E"**) = Follow (E) = {$}, Follow(T) = {+,$}

Look at the Table 5.18, row 2; though there is an SR conflict in state I_2, there are no multiple entries in row 2. That means conflicts are resolved by the SLR(1) parser. Hence, the given grammar cannot be parsed by LR(0) but *can be parsed by SLR(1)*. That is the power of SLR(1).

Table 5.18 SLR(1) Parsing Table

	a	+	$	E	T
0	S_3			1	2
1			acc		
2		S_4	r_2		
3		r_3	r_3		
4	S_3				5
5			r_1		

Example 17: Check whether the following grammar is SLR(1) or not.

E→ **E + T | T, T → TF | F, F → F* | a| b**

Solution: Take augmented grammar G" as follows:

E" → E, E → **E + T | T, T → TF | F, F → F* | a| b**

Draw the DFA with LR(0) items by using the procedure described in 5.9.6. Look at Figure 5.12.

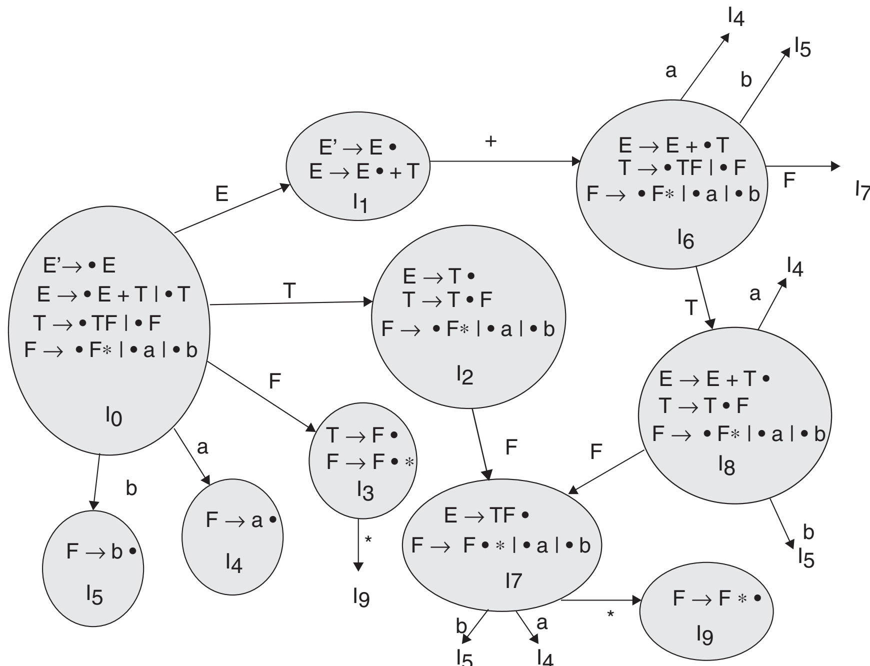


Figure 5.12 DFA for Grammar in Example 17

Observe that there are conflicts in states 2, 3, 7, and 8. The states of LR parser with conflicts are called ***inadequate states***. Because of the conflicts, the grammar is not LR(0). To test whether it is SLR(1) or not, we need to test whether the conflicts are resolved by SLR(1) or not. For this we need not have to construct the complete SLR(1) table. Construct the rows for inadequate states. If there are no multiple entries in that rows, then we can say that the grammar is SLR(1). So let us construct rows for inadequate states, that is, 2, 3, 7, and 8. It is shown in Table 5.19.

Table 5.19 SLR(1) Parsing Table for Inadequate States

State	Action				
	a	**b**	**+**	*	**$**
2	S_4	S_5	r_2		r_2
3	r_4	r_4	r_4	S_9	r_4
7	r_3	r_3	r_3	S_9	r_3
8	S_4	S_5	r_1		r_1

Follow(E) = {$, +}, Follow(T)={$, +, a, b}, Follow(F) = {$, +, a, b, *},

As there are no multiple entries in any row, the grammar is SLR(1).

Example 18: Check whether the following grammar is SLR(1) or not.

$S \to A\ \mathbf{a}\ A\ \mathbf{b} \mid \mathrm{B}\ \mathbf{b}\ B\ \mathbf{a}$

$A \to \varepsilon$

$B \to \varepsilon$

Solution: Take augmented grammar G″ as follows:

S″ → S, S → **AaAb** | **BbBa**, **A** → ε, **B** → ε

Create canonical collection of LR(0) items

I_0 = {S″ → • S, S → • A a A b, S → • B b B a, A → •, B → • },
I_1 = goto(I_0, S) = {S″ → S •},
I_2 = goto(I_0, A) = {S → A • a A b }
I_3 = goto(I_0, B) = {S → B • b B a }
I_4 = goto(I_2, **a**) = {S → A a • A b, A → • }
I_5 = goto(I_3, **b**) = {S → B b • B a, B → • }
I_6 = goto(I_4, A) = {S → A a A • b }
I_7 = goto(I_5, B) = {S → B b B • a }
I_8 = goto(I_6, **b**) = {S → A a A b • }
I_9 = goto(I_7, **a**) = {S → B b B a • }

Draw the DFA with LR(0) items by using the procedure described in 5.9.6.

This is a simple unambiguous grammar, which is LL(1) but not LR(0). Look at Figure 5.13. State I_0 has RR conflict. There are no conflicts in states I_2 or I_6. Let us check for SLR(1) by constructing table.

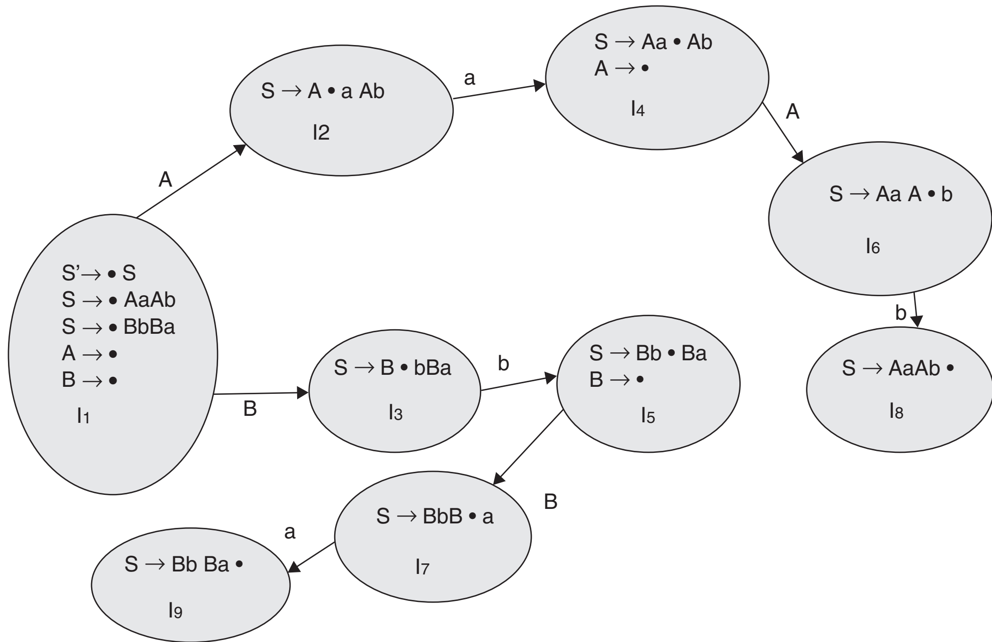


Figure 5.13 DFA for Grammar in Example 18

Follow(S) = {\$}, Follow(A) = {**a**, **b**}, Follow(B) = {**a**, **b**};

Look at the Table 5.20. Since there are reduce–reduce conflicts in [I_1, **a**] and [I_1, **b**], this grammar is not an SLR(1) grammar. Hence, this grammar is LL(1) but not LR(0) and not SLR(1).

Let us examine why SLR(1) is weak. There are two reasons for the weakness of SLR(1).

- Though SLR(1) uses look ahead given by the follow set, actually follow sets are larger than actual lookaheads. That leads to more entries than required in parsing table. Hence, SLR(1) parses only a small class of grammars.
- Look ahead information is not available in any state of parser, it is used independent of any state by the follow set.

To overcome the above disadvantages, the better parsers, that is, LR(1) and LALR(1) are constructed using LR(1) items.

LR(1) item = LR(0) item + look ahead terminal
Ex: A → a•, a where the input symbol "a" is called the "lookahead"
(which is of length 1). That's why it is called LR(1) item.

When look ahead is present with the final item, it tells when to perform reduce action. In the above example, LR(1) item indicates that reduce a to A only if the next symbol is "a."

Table 5.20 Conflicts in SLR(1) Parsing Table

States	Action			Goto		
	a	b	$	S	A	B
I_0	r_3/r_4	r_3/r_4		1	2	3
I_1			accept			
I_2	S_4					
I_3		S_5				
I_4	r_3	r_3			6	
I_5	r_4	r_4				7
I_6		S_8				
I_7	S_9					
I_8			r_1			
I_9			r_2			

5.12 Canonical LR(1) Parsers CLR(1)/LR(1)

This technique of parsing is the most powerful among all LR parsers. Let us see how to construct the LR(1) parser. Procedure is the same as SLR(1). Take augmented grammar, create canonical collection of LR(1) items, draw DFA, and prepare table. So to create canonical collection of LR(1) items, we use closure and goto functions. Earlier, we have defined them for LR(0) items; now let us redefine closure and goto functions for LR(1) items.

5.12.1 Closure(I) Where I Is a Set of LR(1) Items

I is a set of LR(1) items like [A→α•Bβ, $], [C→a•,a]. The function closure takes input as a set of LR(1) items and produces set of LR(1) as output. The function is evaluated using the following two rules:

1. Initially add every item from input to output.
2. If A→α•Bβ,$ is in I where B is nonterminal & B → γ is the rule for B then add **B** → γ, First(β**$)** to Closure (I). Repeat this for every newly added item.

Example 19: Consider the grammar

S → CC
C → cC
C → d
Find closure(S″ → • S, $)

Solution: First add every item from input to output.

$S'' \rightarrow \bullet S, \$$

Now because of "S" add rules for S where look ahead is First($) = {$}

$S \rightarrow \bullet \mathbf{CC}, \$$

Now because of "C," add rules for C where look ahead is First(C$) = {c,d}

$C \rightarrow \bullet \mathbf{cC}, c \mid d$

$\mathbf{C} \rightarrow \bullet \mathbf{d}, c \mid d$

In the newly added items there are no nonterminals next to dot. So that completes closure().

Result is

$S'' \rightarrow \bullet S,\$$
$S \rightarrow \bullet \mathbf{CC},\$$
$C \rightarrow \bullet \mathbf{cC}, c \mid d$
$\mathbf{C} \rightarrow \bullet \mathbf{d}, c \mid d$

5.12.2 Goto(I,X)

Given the set of all items of the form $[A \rightarrow \alpha \bullet X\beta, a]$ in I, $goto[I, X] = closure([A \rightarrow \alpha X \bullet \beta; a]$

The Goto function involves two steps—find transition and apply closure. Earlier we have defined the *goto()* function for LR(0) items. The same definition still holds good for LR(1) also. But we need to extend the definition for look ahead. The change in look ahead in goto function is as follows:

- While finding transition look ahead remains the same
- While finding closure, look ahead may change.

Example 20: Find goto(I_0,C) if I_0 is as follows

I_0: $S'' \rightarrow \bullet S,\$$
$S \rightarrow \bullet \mathbf{CC},\$$
$C \rightarrow \bullet \mathbf{cC}, c \mid d$
$\mathbf{C} \rightarrow \bullet \mathbf{d}, c \mid d$

Solution: First find transition in each item on "C." Result is

$S \rightarrow \mathbf{C} \bullet \mathbf{C},\$$

Now apply closure on the above item. Here evaluate look ahead *as first of grammar symbols next to nonterminal including look ahead.*

Because of "C" add rules for C where look ahead is First($)={$}

$C \rightarrow \bullet \mathbf{cC}, \$$
$\mathbf{C} \rightarrow \bullet \mathbf{d}, \$$

So result of goto() is

I_1: $S \rightarrow \mathbf{C} \bullet \mathbf{C},\$$
$C \rightarrow \bullet \mathbf{cC}, \$$
$\mathbf{C} \rightarrow \bullet \mathbf{d}, \$.$

Example 21: Find **goto(I_1,c)**

Solution: First find transition in each item on "C." Result is

$C \rightarrow c \bullet \mathbf{C}, \$$

Now apply closure on the above item.
Because of "C" add rules for C where look ahead is First($) = {$}

$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

So result of goto() is

$I_2: C \rightarrow c \bullet C, \$$
$C \rightarrow \bullet cC, \$$
$C \rightarrow \bullet d, \$$

Creating the canonical collection of LR(1) items and drawing the DFA is the same as LR(0).

5.12.3 Creating Canonical Collection "C" of LR(1) Items

$C = \{I_0, I_1, I_2, I_3 \ldots I_n\}$

1. The initial item in C is I_0 = closure (augmented production with dot at the beginning, $).
   For example, in the above grammar, I_0 = closure($E'' \rightarrow \bullet E, \$$)
2. For each I_i in C and each grammar symbol X in G
   Repeat

   While goto(I,X) is not empty and not in C
   Add goto(I,X) to C

   Until no more set of items can be added to C.

Let us understand the procedure with an example.

Example 22: Draw DFA with LR(1) items for the grammar

$S \rightarrow AA$
$A \rightarrow aA \mid b$

Solution: Take augmented grammar

$S'' \rightarrow S$, $S \rightarrow AA$, $A \rightarrow aA \mid b$

I_0 = closure($S'' \rightarrow \bullet S, \$$) = $S'' \rightarrow \bullet S, \$$
$S \rightarrow \bullet AA, \$$
$A \rightarrow \bullet aA \mid \bullet b, a \mid b$

I_1 = Goto(I_0,S) = $S'' \rightarrow S, \$$

I_2 = Goto(I_0,A) = $S \rightarrow A \bullet A, \$$
$A \rightarrow \bullet aA \mid \bullet b, \$$

I_3 = Goto(I_0,a) = $A \rightarrow a \bullet A, a \mid b$
$A \rightarrow \bullet aA \mid \bullet b, a \mid b$

I_4 = Goto(I_0,b) = $A \rightarrow b \bullet, a \mid b$

That's all the transitions from I_0. Now apply the goto function on I_1. Since I_1 has only the final item, there is no need to apply the goto function. Now consider I_2

I_5 = Goto(I_2,A) = $S \rightarrow AA \bullet, \$$

I_6 = Goto(I_2,a) = $A \rightarrow a \bullet A, \$$

$$A \rightarrow \bullet aA \mid . B, \$$$

I_7= Goto(I_2,b) = $\mathbf{A \rightarrow b\bullet, \$}$

I_8= Goto(I_3,A) = $\mathbf{A \rightarrow aA\bullet, a \mid b}$

I_9= Goto(I_6,A) = $\mathbf{A \rightarrow aA\bullet,\$}$

Goto(I_3,a) = I_3, Goto(I_3,b) = I_4

Goto(I_6,a) = I_6, Goto(I_6,b) = I_7

The DFA for the above set of LR(1) items is as shown in Figure 5.14.

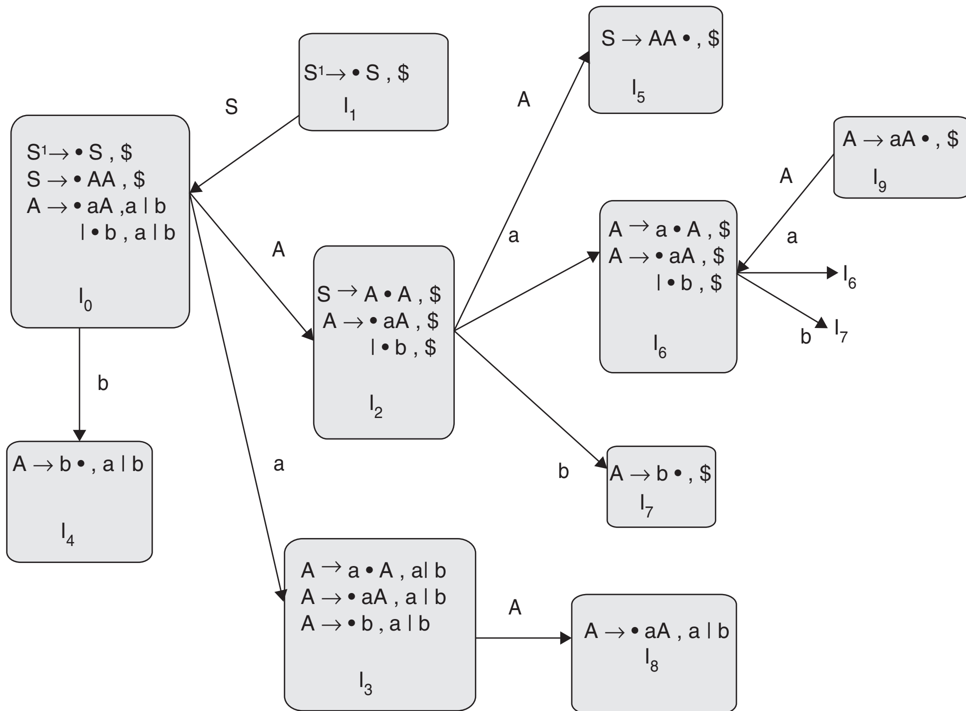


Figure 5.14 DFA for Grammar in Example 22

5.12.4 Constructing CLR(1) Parsing Table

Once DFA is drawn, by using the DFA we can construct the parsing table. The procedure is similar to SLR(1) the only difference being reduce entries. So let us discuss how CLR(1) prepares reduce entries in the parsing table. SLR(1) uses the follow set but there are no complications with LR(1). Look aheads are already available in item itself. So CLR(1) checks the DFA for final item, if final item is available in state I_2 as A → abc, a | $, then it places reduce entry in the row 2 under columns given by look ahead, that is, a, $ as shown in Table 5.21.

Table 5.21 Reduce Entries in the CLR(1) Parsing Table

	a	b	$
2	R		R

This is how invalid reduction of SLR(1) are avoided by CLR(1). SLR(1) places reduce entry at more places as follow sets are larger than actual look aheads but CLR(1) puts only under actual look aheads. That's why CLR(1) is the most powerful and can parse almost all CFG.

Example 23: Construct the CLR(1) parsing table for the above DFA.

Solution: There are 10 states in DFA I_0 to I_9. So 10 rows. Columns are a, b, $, in action part and S, A in goto part.

Table 5.22 is the parsing table of the most powerful parser, that is, the CLR(1) Parser.

Table 5.22 CLR(1) Parsing Table

States	Action			Goto	
	a	b	$	S	A
0	s_3	s_4		1	2
1			acc		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

5.12.5 CLR(1) Grammar

If the CLR(1) parser does not contain any multiple entries then grammar is called ***CLR(1) or LR(1) Grammar***. Given a grammar to check whether it is CLR(1) or not, construct DFA with LR(1) items. If there are any conflicts in DFA then it is not CLR(1). So checking for SR or RR conflicts in DFA is very important for verifying whether the grammar is LR(1) or not. Let us review checking SR or RR conflicts with LR(0) and LR(1) items. An example for each case is shown in Table 5.23.

Table 5.23 Conflicts with LR Items

	With LR(0) Items	**With LR(1) Items**
Shift Reduce conflict	$A \to \alpha \bullet a\beta$ $B \to \gamma \bullet$	$A \to \alpha \bullet a\beta, \$$ $B \to \gamma \bullet, a$
Reduce Reduce Conflict	$A \to \alpha \bullet$ $B \to \gamma \bullet$	$A \to \alpha \bullet, a$ $B \to \gamma \bullet, a$

For example the following LR(1) items

$A \to \alpha \bullet a\beta, \$$
$B \to \gamma \bullet, b$

	a	**b**	**$**
i	s	r	

There is no SR conflict because the shift action is on "a" and reduce action is on "b." There is no conflict at all.

For example, the following LR(1) items

$A \to \alpha \bullet A\beta, \$$
$B \to \gamma \bullet, b$

	a	**b**	**$**
i		r	

There is no SR conflict because there is no shift action as the symbol next to dot in nonfinal item is "A" nonterminal. On seeing "A" if it enters to another state that corresponds to the goto state but not shift action. There is only reduce action on "b." There is no conflict at all.

For example, the following LR(1) items

$A \to \alpha \bullet, a$
$B \to \gamma \bullet, b.$

	a	**b**	**$**
i	r_1	r_2	

There is no RR conflict because first reduce action is on "a" and next reduce action is on "b." There is therefore no conflict at all.

So we can test for conflicts in LR(1) items as follows:

SR conflict*: If there is final item with nonfinal item where look ahead in final item is same as the terminal next to dot in nonfinal item, then it is an SR conflict.*

RR conflict*: If there is more than one final item with at least one common look ahead, then it is a RR conflict.*

Example 24: Check if the following grammar is LR(1) or not.

$S \to AA$
$A \to aA \mid b$

Solution: The DFA for above grammar is shown in Figure 5.14.

As there are no conflicts in any state of DFA, the grammar is LR(1).

The same can be verified with the LR(1) parsing table constructed in Example 23.

No multiple entries in Table 5.19. So it is LR(1).

Compare the SLR(1) and CLR(1) tables for the grammar $S \rightarrow AA, A \rightarrow aA \mid b$. The SLR(1) table is in Example 8 and LR(1) table is in Example 23. Note the difference. In SLR parser table there are 7 rows, whereas in LR(1) there are 10 rows. So we observed that for a given grammar if we construct SLR(1) and CLR(1) tables, table size of LR(1) is more as number of rows in LR(1) is more. This is the main disadvantage of LR(1). Though it is most powerful, it requires more table space; that's why this is not a preferred parser. LALR(1) overcomes this disadvantage; that's why LALR(1) is preferred rather than LR(1). Even most of automatic parser generators use the LALR technique rather than the CLR technique. Let us see how to construct the LALR(1) parser.

5.13 LALR(1) Parser

The procedure to construct the LALR(1) parser is the same as that of CLR(1). Construct DFA with LR(1) items. Now check if any two states are differing by look aheads, then merge them into a single state. For example, consider the DFA with LR(1) items for the grammar

$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

Look at the DFA shown in Figure 5.15.

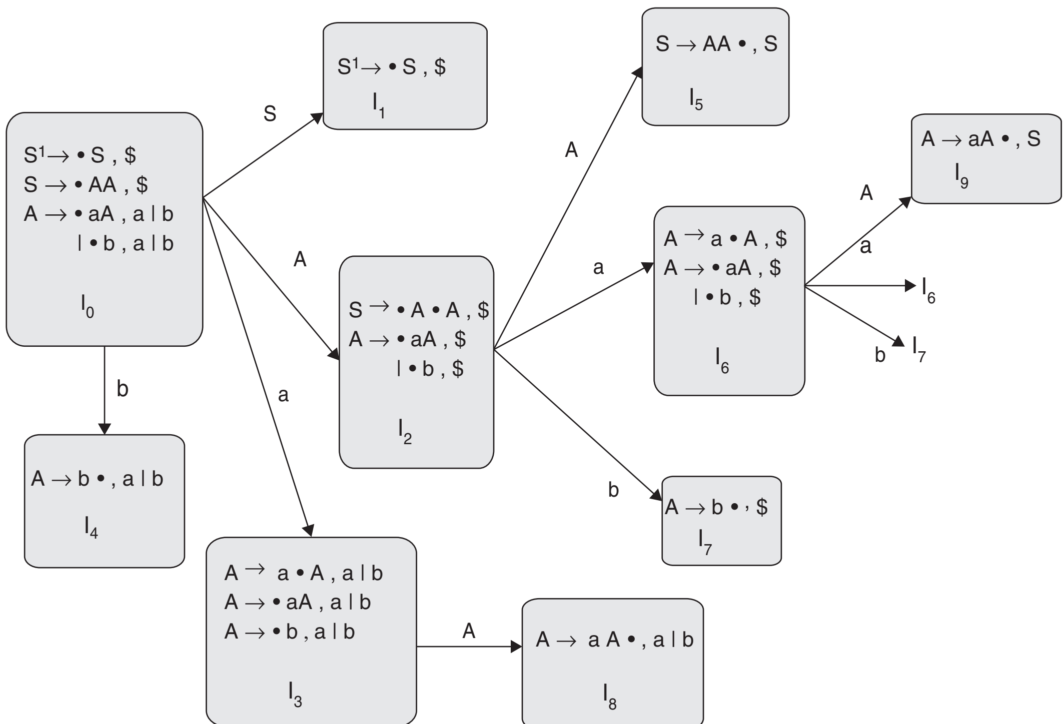


Figure 5.15 DFA for a Given Grammar

Check for the states that are differing by look ahead. Here I_3I_6 and I_4I_7 and I_8I_9 are such states.

Now in LALR(1) we merge them to a single state like I_3 I_6 to new state I_{36}, I_4 I_7 to new state I_{47}, I_8I_9 to new state I_{89}. To merge the states we take items as they are but add lookaheads as union of two states. For example,

$$I_{36} = A \to a \bullet A,\ a \mid b \mid \$$$
$$A \to \bullet\, aA \ \mid \bullet\, b,\ a \mid b \mid \$$$
$$I_{47} = A \to b\bullet,\ a \mid b \mid \$$$
$$I_{89} = \text{Goto}(I_3, A) = A \to aA\bullet,\ a \mid b$$

Now the table is constructed with new states with same procedure. It is shown in Table 5.24.

Table 5.24 LALR(1) Parsing Table

State	Action			Goto	
	a	**b**	**$**	**S**	**A**
0	s_{36}	s_{47}		1	2
1			acc		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

This is the LALR(1) parsing table. Compare this with the SLR parsing Table 5.15. What is the difference? Are they same? Yes. Number of states in SLR is the same as LALR. As LALR is merging the states where items are same but look aheads are different into same state. The number of states is reduced compared to CLR(1). CLR(1) treats such states as separate. That's why more states and more number of rows are present in a parsing table. Hence, LR(1) requires more space. As LALR(1) is combining such states together, the number of states are reduced. Hence it requires less table space than CLR(1). But *for a given grammar, table size of SLR and LALR is the same but CLR will be large.*

Now to test the power of CLR(1) and LALR(1), we can take all the previous examples that failed to be SLR(1). Let us start with the grammar in Example 18.

Example 25: The following grammar is LL(1), not LR(0), not SLR(1)

$$S \to A\,\mathbf{a}\,A\,\mathbf{b} \mid B\,\mathbf{b}\,B\,\mathbf{a}$$
$$A \to \varepsilon$$
$B \to \varepsilon$ check if it is LR(1) and LALR(1).

Solution: Take augmented grammar G″ as follows:

$$S'' \rightarrow S,\ S \rightarrow \mathbf{AaAb} \mid \mathbf{BbBa},\ \mathbf{A} \rightarrow \varepsilon,\ \mathbf{B} \rightarrow \varepsilon$$

Create canonical collection of LR(1) items

I_0 = S″ → • S, \$, S → • A **a** A **b**, \$, S → • B **b** B **a**, \$ A → •, a, B → •, b
I_1 = goto(I_0, S) = {S″ → S •, \$},
I_2 = goto(I_0, A) = {S → A • **a** A **b, \$** }
I_3 = goto(I_0, B) = {S → B • **b** B **a**, \$}
I_4 = goto(I_2, **a**) = {S → A **a** • A **b**, \$ A → •, b }
I_5 = goto(I_3, **b**) = {S → B **b** • B **a**, \$, B → •, a}
I_6 = goto(I_4, A) = {S → A **a** A • **b, \$** }
I_7 = goto(I_5, B) = {S → B **b** B • **a, \$** }
I_8 = goto(I_6, **b**) = {S → A **a** A **b** •, \$ }
I_9 = goto(I_7, **a**) = {S → B **b** B **a** •, &}

Draw the DFA with LR(0) items by using the procedure described in 5.9.6.

Look at Figure 5.16. State I_0 has no conflict. Though there are two final items [A → •, a] and [B→•, a]. Two look aheads are different. Hence, there is no conflict. The grammar is CLR(1) as there are no conflicts in DFA.

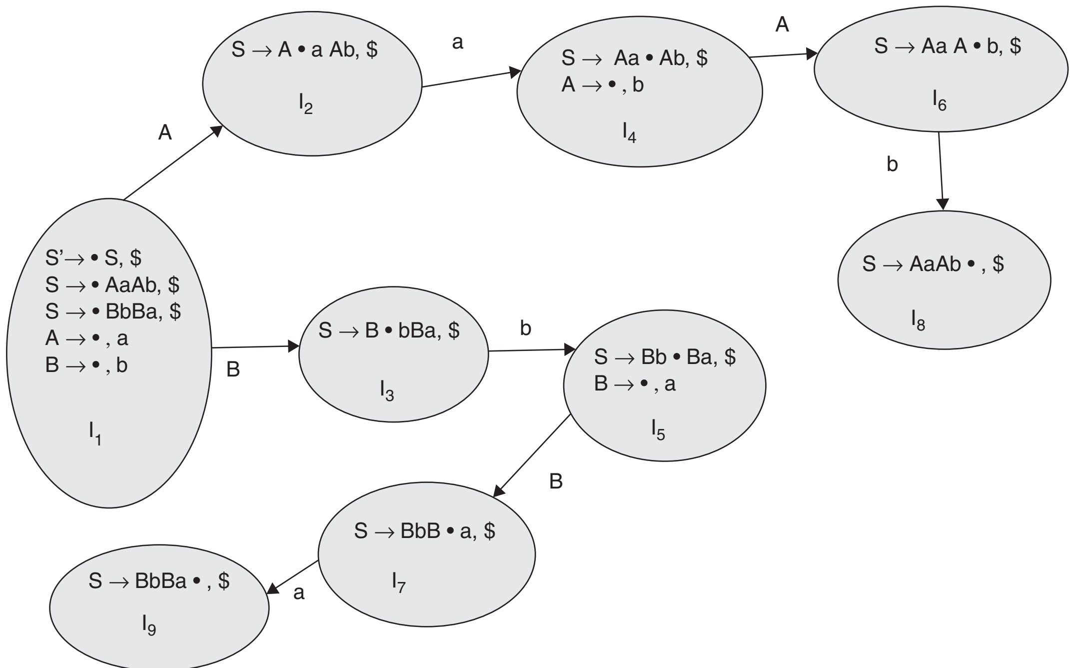


Figure 5.16 DFA for Grammar in Example 25

For checking LALR(1), we need to check if any two states are differing by look aheads.

If such states exist, merge them into single states and then check for conflicts. In this example, there are no such states. Hence, this is also LALR(1). The above grammar cannot be parsed by LR(0) or SLR(1) but can be parsed by CLR(1) or LALR(1). That is the power of LR(1) and LALR(1). Let us understand this power with one more example.

Example 26: The following grammar is not LL(1), not LR(0), and not SLR(1). Check if it is CLR(1) and LALR(1). Also generate the LR(1) parsing table for the following grammar:

$S \rightarrow A\,\mathbf{a} \mid \mathbf{b}\,A\,\mathbf{c} \mid \mathbf{d}\,\mathbf{c} \mid \mathbf{b}\,\mathbf{d}\,\mathbf{a}$
$A \rightarrow \mathbf{d}$

Solution: Take augmented grammar

$S'' \rightarrow S$

1. $S \rightarrow A\,\mathbf{a}$
2. $S \rightarrow \mathbf{b}\,A\,\mathbf{c}$
3. $S \rightarrow \mathbf{d}\,\mathbf{c}$
4. $S \rightarrow \mathbf{b}\,\mathbf{d}\,\mathbf{a}$
5. $A \rightarrow \mathbf{d}$

Create canonical collection of LR(1) items.

$I_1 = \{(S'' \rightarrow \bullet\, S, \$), (S \rightarrow \bullet\, A\,\mathbf{a}, \$), (S \rightarrow \bullet\, \mathbf{b}\,A\,\mathbf{c}, \$), (S \rightarrow \bullet\, \mathbf{d}\,\mathbf{c}, \$), (S \rightarrow \bullet\, \mathbf{b}\,\mathbf{d}\,\mathbf{a}, \$), (A \rightarrow \bullet\, \mathbf{d}, \mathbf{a})\}$,

$I_2 = \text{goto}(I_1, S) = \{(S'' \rightarrow S\, \bullet, \$)\}$,

$I_3 = \text{goto}(I_1, A) = \{(S \rightarrow A\, \bullet\, \mathbf{a}, \$)\}$,

$I_4 = \text{goto}(I_1, \mathbf{b}) = \{(S \rightarrow \mathbf{b}\, \bullet\, A\,\mathbf{c}, \$), (S \rightarrow \mathbf{b}\, \bullet\, \mathbf{d}\,\mathbf{a}, \$), (A \rightarrow \bullet\, \mathbf{d}, \mathbf{c})\}$,

$I_5 = \text{goto}(I_1, \mathbf{d}) = \{(S \rightarrow \mathbf{d}\, \bullet\, \mathbf{c}, \$), (A \rightarrow \mathbf{d}\, \bullet, \mathbf{a})\}$,

$I_6 = \text{goto}(I_3, \mathbf{a}) = \{(S \rightarrow A\,\mathbf{a}\, \bullet, \$)\}$,

$I_7 = \text{goto}(I_4, A) = \{(S \rightarrow \mathbf{b}\,A\, \bullet\, \mathbf{c}, \$)\}$,

$I_8 = \text{goto}(I_4, \mathbf{d}) = \{(S \rightarrow \mathbf{b}\,\mathbf{d}\, \bullet\, \mathbf{a}, \$), (A \rightarrow \mathbf{d}\, \bullet, \mathbf{c})\}$,

$I_9 = \text{goto}(I_5, \mathbf{c}) = \{(S \rightarrow \mathbf{d}\,\mathbf{c}\, \bullet, \$)\}$,

$I_{10} = \text{goto}(I_7, \mathbf{c}) = \{(S \rightarrow \mathbf{b}\,A\,\mathbf{c}\, \bullet, \$)\}$,

$I_{11} = \text{goto}(I_8, \mathbf{a}) = \{(S \rightarrow \mathbf{b}\,\mathbf{d}\,\mathbf{a}\, \bullet, \$)\}$.

Look at Figure 5.17. The states I_5 and I_8 contain final and nonfinal items but there is no SR or RR conflict as look ahead in the final item is not the same as terminal next to dot in the nonfinal item. So no conflicts. This grammar is an LR(1) grammar. There are no states differing by look aheads; hence, it is also LALR(1). The parsing table for CLR(1) and LALR(1) is as shown in Table 5.25.

This grammar is not LL(1), not LR(0), not SLR(1) but is CLR(1) and LALR(1) grammar.

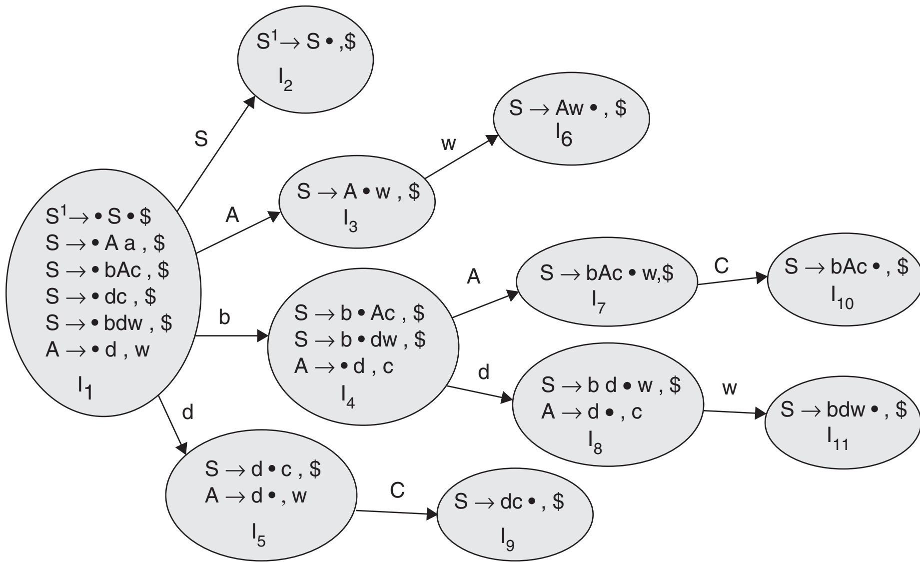


Figure 5.17 DFA for Grammar in Example 26

Table 5.25 CLR(1)/LALR(1) Parsing Table

States	Action					Goto	
	A	b	c	d	$	S	A
1		s_4		s_5		2	3
2					accept		
3	s_6						
4				s_8			7
5	r_5		s_9				
6					R_1		
7			s_{10}				
8	s_{11}		r_5				
9				r_3			
10				r_2			
11				r_4			

Example 27: The following grammar is not LL(1), not LR(0) and not SLR(1). Check if it is CLR(1) and LALR(1). Also generate the LR(1) parsing table for the following grammar

$S \rightarrow A\,\mathbf{a} \mid \mathbf{b}\,A\,\mathbf{c} \mid B\,\mathbf{c} \mid \mathbf{b}\,B\,\mathbf{a}$
$A \rightarrow \mathbf{d}$
$B \rightarrow \mathbf{d}$

Solution: Take augmented grammar

$S'' \rightarrow S$

1. $S \rightarrow A\,\mathbf{a}$
2. $S \rightarrow \mathbf{b}\,A\,\mathbf{c}$
3. $S \rightarrow B\,\mathbf{c}$
4. $S \rightarrow \mathbf{b}\,B\,\mathbf{a}$
5. $A \rightarrow \mathbf{d}$
6. $B \rightarrow \mathbf{d}$

Create canonical collection of LR(1) items.

$I_1 = \{(S'' \rightarrow \bullet\, S, \$), (S \rightarrow \bullet\, A\,\mathbf{a}, \$), (S \rightarrow \bullet\, \mathbf{b}\,A\,\mathbf{c}, \$), (S \rightarrow \bullet\, B\,\mathbf{c}, \$),$
$(S \rightarrow \bullet\, \mathbf{b}\,B\,\mathbf{a}, \$), (A \rightarrow \bullet\, \mathbf{d}, \mathbf{a}), (B \rightarrow \bullet\, \mathbf{d}, \mathbf{c})\}$
$I_2 = \text{goto}(I_1, S) = \{(S'' \rightarrow S\,\bullet, \$)\},$
$I_3 = \text{goto}(I_1, A) = \{(S \rightarrow A\,\bullet\,\mathbf{a}, \$)\},$
$I_4 = \text{goto}(I_1, \mathbf{b}) = \{(S \rightarrow \mathbf{b}\,\bullet\,A\,\mathbf{c}, \$), (S \rightarrow \mathbf{b}\,\bullet\,B\,\mathbf{a}, \$), (A \rightarrow \bullet\,\mathbf{d}, \mathbf{c}),$
$(B \rightarrow \bullet\,\mathbf{d}, \mathbf{a})$
$I_5 = \text{goto}(I_1, B) = \{(S \rightarrow B\,\bullet\,\mathbf{c}, \$)\},$
$I_6 = \text{goto}(I_1, \mathbf{d}) = \{(A \rightarrow \mathbf{d}\,\bullet, \mathbf{a}), (B \rightarrow \mathbf{d}\,\bullet, \mathbf{c})\},$
$I_7 = \text{goto}(I_3, \mathbf{a}) = \{(S \rightarrow A\,\mathbf{a}\,\bullet, \$)\},$
$I_8 = \text{goto}(I_4, A) = \{(S \rightarrow \mathbf{b}\,A\,\bullet\,\mathbf{c}, \$)\},$
$I_9 = \text{goto}(I_4, B) = \{(S \rightarrow \mathbf{b}\,B\,\bullet\,\mathbf{a}, \$)\},$
$I_{10} = \text{goto}(I_4, \mathbf{d}) = \{(A \rightarrow \mathbf{d}\,\bullet, \mathbf{c}), (B \rightarrow \mathbf{d}\,\bullet, \mathbf{a})\},$
$I_{11} = \text{goto}(I_5, \mathbf{c}) = \{(S \rightarrow B\,\mathbf{c}\,\bullet, \$)\},$
$I_{12} = \text{goto}(I_8, \mathbf{c}) = \{(S \rightarrow \mathbf{b}\,A\,\mathbf{c}\,\bullet, \$)\},$
$I_{13} = \text{goto}(I_9, \mathbf{a}) = \{(S \rightarrow \mathbf{b}\,B\,\mathbf{a}\,\bullet, \$)\},$

This grammar is an LR(1) grammar as there are no conflicts in the above DFA as shown in Figure 5.18. To check for LALR(1), verify whether there are any two states differing by look aheads. Look at states I_6 and I_{10}. Though there are no conflicts in the two states, later for LALR(1), when we combine the two, it results in RR conflict.

$I_{610} = A \rightarrow \mathbf{d}\,\bullet, \mathbf{a} \mid \mathrm{c}$
$B \rightarrow \mathbf{d}\,\bullet, \mathbf{c} \mid \mathbf{a},$

This is a conflict in LALR(1). Hence, this grammar cannot be parsed by LALR(1) also. It can be parsed only by CLR(1—the most powerful parser.

The CLR(1) parsing table is as shown in Table 5.26.

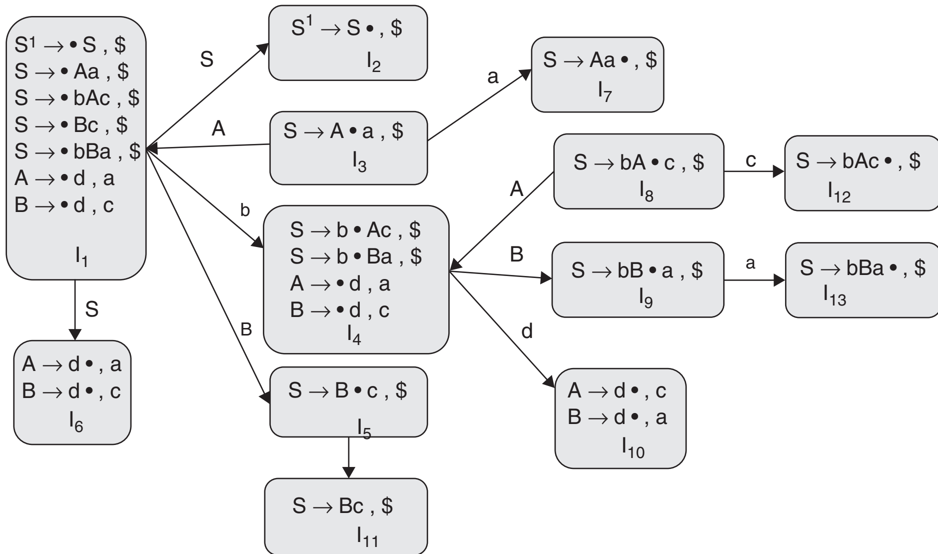


Figure 5.18 DFA for Grammar in Example 27

Table 5.26 CLR(1) Parsing Table

States	Action					Goto		
	A	b	c	d	$	S	A	B
1		s_4		s_6		2	3	5
2					accept			
3	s_7							
4				s_{10}			8	9
5			s_{11}					
6	r_5		r_6					
7					r_1			
8			s_{12}					
9	s_{13}							
10	r_6		r_5					
11				r_3				
12				r_2				
13				r_4				

Example 28: Check whether the following grammer is CLR(1) or LALR(1).

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

Solution: First construct the initial item.

$I_0 = \{(S'' \rightarrow \bullet A, \$), (A \rightarrow \bullet A B, \$), (A \rightarrow \bullet, \$).$

This is what we get when we apply closure on first production. Now we need to apply closure to newly added item, which results in

$(A \rightarrow \bullet A B, a \mid b), (A \rightarrow \bullet, a \mid b).$

So, finally we get

$I_0 = \{(S'' \rightarrow \bullet A, \$), (A \rightarrow \bullet A B, \$), (A \rightarrow \bullet, \$),(A \rightarrow \bullet A B, a \mid b), (A \rightarrow \bullet, a \mid b),\}$

As all these are in the same state, we can combine similar items together. Hence, we get

$I_0 = \{(S'' \rightarrow \bullet A, \$), (A \rightarrow \bullet A B, a \mid b \mid \$), (A \rightarrow \bullet, a \mid b \mid \$)\}$

DFA with LR(1) items is shown in Figure 5.19.

This grammar is CLR(1) and LR(1). Also as there are no conflicts in DFA and no two states differing by look ahead.

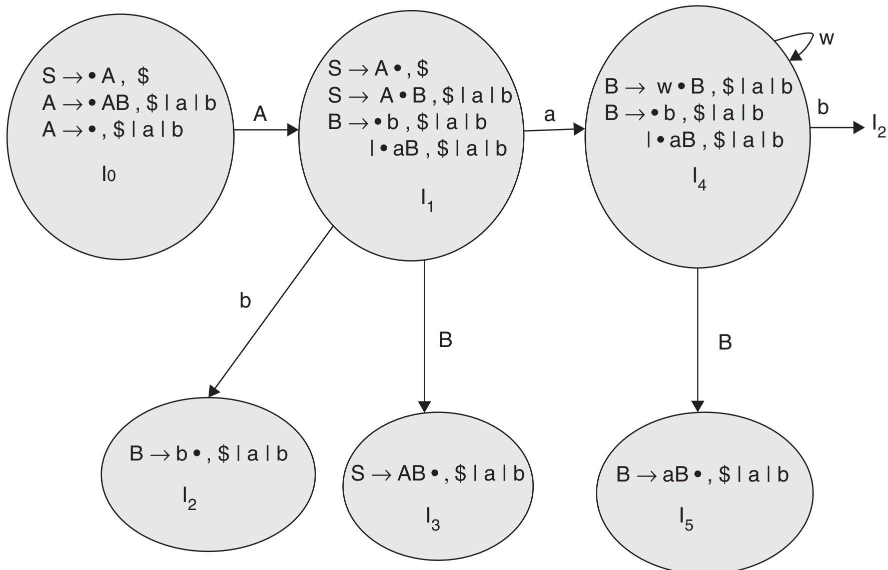


Figure 5.19 DFA for Grammar in Example 28

5.14 Comparison of Parsers: Top-Down Parser vs. Bottom-Up Parser

Out of all top-down parsers widely used, one is the LL(1) parser. Out of all bottom-up parsers widely used, one is the LALR(1) parser. Let us compare them with the following criteria.

***Design*:** Top down is simple to construct than bottom up.

***Table size*:** LALR(1) parser table size is exponential to the size of grammar. This is not the case with LL(1) where the table size is bounded by the square of the size of grammar. The size of the bottom-up parser is roughly double the size of top-down parser.

***Power*:** $LL(k) \subset LR(k)$

***Error Detection*:** Predictive nature of TDP allows errors to be detected at the earliest possible time. The TDP parser stack can be used to repair as it contains information on what is expected to be seen in contrast with BUP where the stack contains what has already been encountered in parsing. So far we have discussed many examples of parsers. Let us summarize them in Table 5.27.

The LL(1) parser is a simple top-down parser that can parse only a small class of grammars as there is a restriction on the grammar that it should be free of left recursion and should be left factored. LR(0) is the simplest of all LR parsers but not used practically as it is less powerful. It does not use any look ahead in making parsing decisions; hence, it can be used to parse only a small class of grammars. SLR(1) is better than LR(0) as it uses look ahead.

Table 5.27 Grammars That Can be Parsed by Different Parsers

Grammar		LL(1)	LR(0)	SLR(1)	LALR(1)	CLR(1)
1.	$S \to a$	√	√	√	√	√
2.	$E \to E + T \mid T$ $T \to a$	X	√	√	√	√
3.	$E \to E + T \mid T$ $T \to T * F \mid F$ $F \to id$	X	X	√	√	√
4.	$S \to Aa \mid bAc \mid dc \mid bda$ $A \to d$	X	X	X	√	√
5.	$S \to Aa \mid bAc \mid Bc \mid bBa$ $A \to d, B \to d$	X	X	X	X	√
6.	$S \to a \mid A, A \to a$	X	X	X	X	X

But once again it is the least powerful as it cannot avoid invalid reduction using the follow set. CLR(1) is the most powerful parser among all but requires more table space. LALR(1) is better than SLR(1) as it uses LR(1) items; it is preferred than CLR(1) as it requires less table space. The class of grammars that can be parsed by different parsers is shown in Figure 5.20.

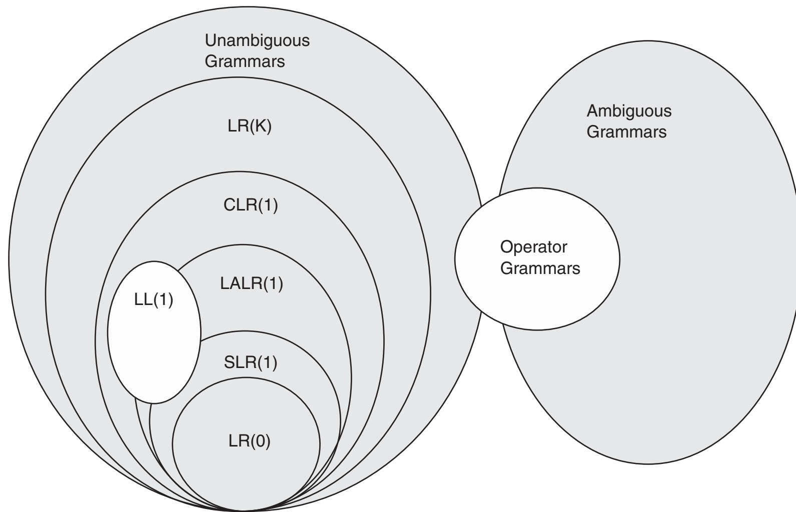


Figure 5.20 Comparison of All Parsers

If a grammar is LR(0), it is SLR(1), LALR(1), and CLR(1). If a grammar is LALR(1), it can be CLR(1) but may or may not be SLR(1) and it may or may not be LR(0). But every LL(1) grammar is LALR(1).

5.15 Error Recovery in LR Parsing

An LR parser will detect error when it consults the parsing action table and find a blank entry. An LR parser implements panic mode error recovery as follows: for each blank entry, have a pointer to error recovery subroutine. So that whenever the blank entry is referred, the corresponding subroutine is called. For example, look at Table 5.28, e_1, e_2, e_3, and e_4 are error recovery routines. They specify how to recover from an error.

Table 5.28 Error Recovery in LR Parsing Table

	id	+	*	(	)	$
0	S_3	e_1	e_2	S_2	e_2	e_1

Here e_1( ) tells that an operand is missing at the beginning of the expression. So push id and issue error message “missing operand.”

e_2( ) tells that expression is starting with). So error recovery here can be to delete input symbol and issue error message.

5.16 Parser Construction with Ambiguous Grammars

So far whatever the parsing techniques discusses are applicable provided the grammar is unambiguous. If it is ambiguous, if we follow the above procedures, we get multiple entries in the parsing table—whether it is LL(1) or LR(1). So let us see whether there is any way of constructing a parser with ambiguous grammars. Before looking at parser design, let us first understand what the advantages of ambiguous grammars are.

- *Ambiguous grammars are shorter and natural.* For example, look at the following ambiguous and its equivalent unambiguous grammar.

$$\begin{aligned} E &\rightarrow E + E \\ &\mid E * E \\ &\mid id \end{aligned} \qquad\qquad \begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

- *If the parser is constructed with ambiguous grammar, we can change precedence associativity of the parser at a later time,* as precedence and associativity are not fixed in ambiguous grammar.
- *No wastage of time.* Unambiguous grammar wastes time with reduction like $E \rightarrow T$, $T \rightarrow F$, $F \rightarrow id$. No such wastage of time.

To take advantage of all the above, most importantly, it is easy to write ambiguous than unambiguous. Hence, it is better to construct parser with ambiguous grammar. For this we are not going to modify any previous procedure. Follow the same procedure and construct the table (anything LL(1) or LR(1)). Now the table will have multiple entries. Use disambiguating rules and resolve multiple entries to single entries. So the change in the procedure here is resolving multiple entries to single entries. To understand how to resolve multiple entries to single entries, let us take a SLR(1) parsing table for ambiguous grammar **$E \rightarrow E + E \mid E * E \mid id$.** By following the procedure already discussed, we get SLR(1) Table 5.29.

Table 5.29 Error Recovery in LR Parsing Table

States	Action				Goto
	id	+	*	$	E
0	S_2				1
1		S_3	S_4	Acc	
2		r_3	r_3	r_3	
3	S_2				5
4	S_2				6
5		**S_3/r_1**	**S_4/r_1**	r_1	
6		**S_3/r_2**	**S_4/r_2**	r_2	

There are four SR conflicts. So this table as such cannot be used. Let us see how to resolve this. We use disambiguating rules for resolving multiple entries. The disambiguating rules for this grammar are precedence and associativity of operators. "*" has higher precedence than "+" and both are left associative. With this information we resolve conflicts in the following ways:

Take a simple string "id + id * id $." Parse it.

id + id * id $

	~~2~~	5
	~~id~~	E
1	3	3
E	+	+
~~2~~	1	1
~~id~~	E	E
0	0	0

State 5 on "*" is $\mathbf{S}_4/\mathbf{r}_1$—an S/R conflict. Let us understand why SR conflict. The string that is already read is id + id. There is no conflict till this is read. But on reading the next operator, that is, "*," if you look at the stack, there is a handle E + E; so it can go for reduce action as handle is available. Otherwise, it can read further for the longest match, which is nothing but shift action. That's why we have an SR conflict. Now let us resolve what right action is. "*" has higher precedence than "+." So if parser goes with reduce action, + is given higher precedence, wrong decision. It should be resolved in *favour of shift*. that is, $\mathbf{S}_4$. This means, read further till id * id is read. Reduce "*" first and then "+."

Similarly, take a string "id + id + id$." In state 5 on "+," the parser will refer to $\mathbf{S}_3/\mathbf{r}_1$—an S/R conflict. The string that is already read is "id+id." So on the next token "+," now parser may not be able to decide whether to go for shift/reduce as handle E + E is already available. Here associativity of "+" should be considered. As it is left associative, resolve conflict in *favour of reduce*, that is, $\mathbf{r}_1$.

Similarly, take a string "id * id + id$." In state 6 on "+," the parser will refer to $\mathbf{S}_3/\mathbf{r}_2$—an S/R conflict. The string that is already read is "id * id." So on the next token "+," now parser may not be able to decide whether to go for shift/reduce as handle E * E is already available. "*" has higher precedence than "+." So the parser goes with reduce action. Resolve conflict in *favour of reduce*, that is, $\mathbf{r}_2$.

Similarly, take a string "id * id * id$." In state 6 on "*," parser will refer $\mathbf{S}_4/\mathbf{r}_2$—an S/R conflict. The string that is already read is "id * id." So on the next token "*," now parser may not be able to decide whether to go for shift or reduce as handle E * E is already available. Here associativity of "+" should be considered. As it is left associative, resolve conflict in *favour of reduce*, that is, $\mathbf{r}_2$. So the resulting parsing table is shown in Table 5.30.

As there are no multiple entries, it can be used by the parser to parse any string defined by the grammar.

Table 5.30 Error Recovery in the LR Parsing Table

States	Action				Goto
	id	**+**	*****	**$**	**E**
0	S_2				1
1		S_3	S_4	acc	
2		r_3	r_3	r_3	
3	S_2				5
4	S_2				6
5		$\mathbf{S_3}$	$\mathbf{r_1}$	r_1	
6		$\mathbf{r_2}$	$\mathbf{r_2}$	r_2	

Solved Problems

1. What is the equivalent operator grammar for the following grammar?

    $S \rightarrow AB$, $A \rightarrow c \mid d$, $B \rightarrow aAB \mid b$

    Solution: Here replace nonterminal B by its production. Then we get

    $S \rightarrow AaAB \mid Ab$, $A \rightarrow c \mid d$, $B \rightarrow aAB \mid b$

    but AB is S. So equivalent operator grammar is

    $S \rightarrow AaS \mid Ab$, $A \rightarrow c \mid d$, $B \rightarrow aS \mid b$

2. Convert the following precedence relation table to function table

	a	**(**	**)**	**,**	**$**
a			>	>	>
(	<	<	≐	<	
)			>	>	>
,	<	<	>	>	
$	<	<			

Solution: There are five terminals; so create 10 symbols. Here there is ≐ relation between "(" and ")". So take f(and g) into one group, remaining symbols can be taken as separate nodes. Now add edges for each < and > relation. We get the following digraph shown in Figure 5.21.

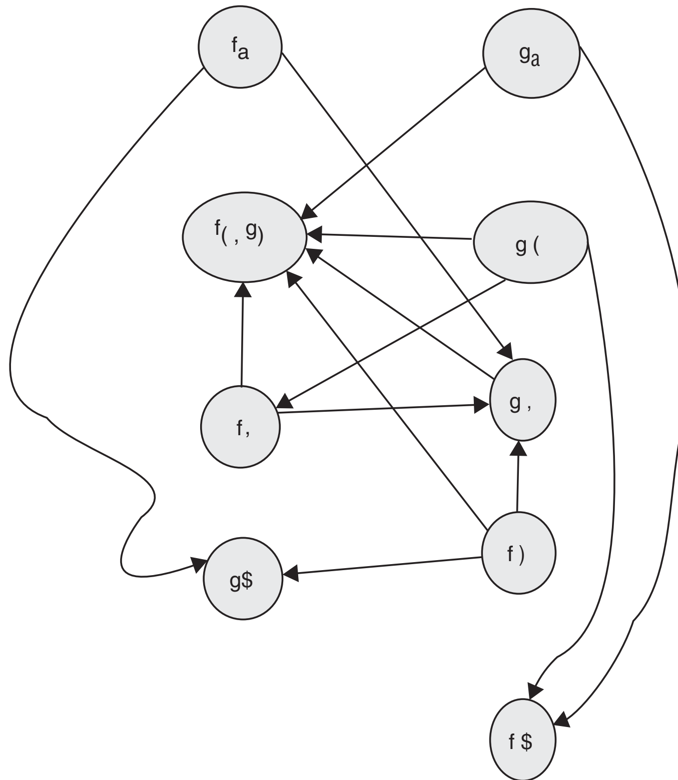


Figure 5.21 Digraph

The precedence function table is as follows:

	a	(	)	,	$
f	2	0	2	2	0
g	3	3	0	1	0

3. Consider the following grammar:

   E → E + T | T
   T → T * F | F
   F → (E) | id.

   Construct SLR(1) parsing table.

   Solution: The DFA with LR(0) items is shown in Figure 5.22.

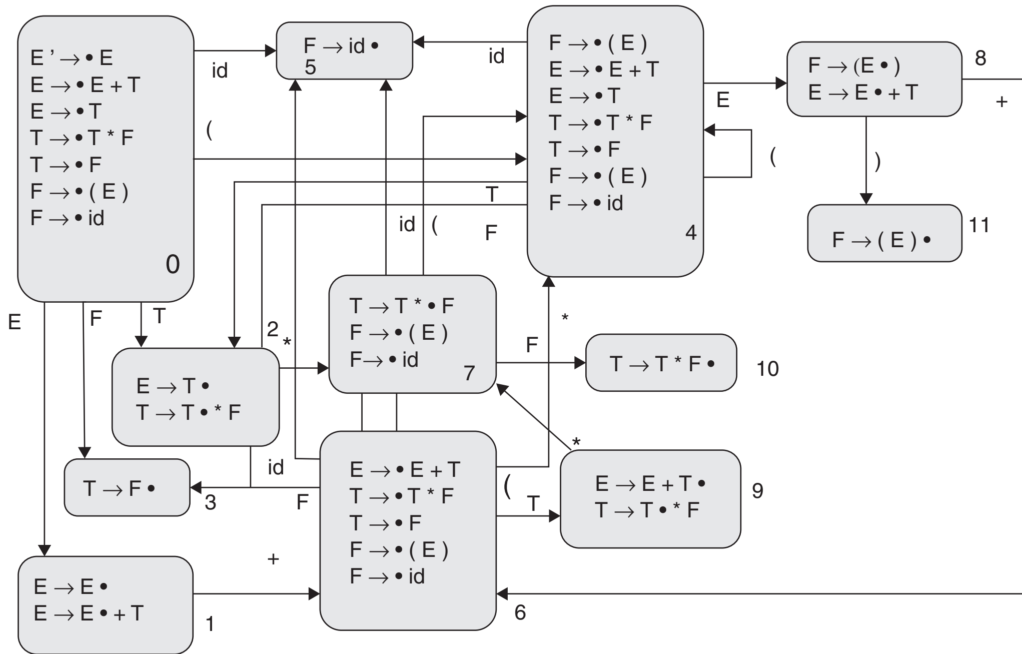


Figure 5.22 DFA with LR(0) items

The SLR(1) parsing table for the above grammar is given in Table 5.31.

Table 5.31 SLR(1) Parsing Table for Example 3

States	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

4. Construct DFA with LR(1) items for the following grammar:

S → CC
C → cC
C → d

Solution:

I_0: closure({(S″ → • S, $)}) = { (S″ → • S, $),
(S → • C C, $), (C → • cC, c/d), (C → • d, c/d)}
I_1: goto(I_0, S) = (S″ → S •, $)
I_2: goto(I_0, C) = {(S → C • C, $), (C → • cC, $), (C → • d, $)}
I_3: goto(I_0, c) = {(C → c • C, c/d), (C → • cC, c/d), (C → • d, c/d) }
I_4: goto(I_0, d) = (C → d •, c/d)
I_5: goto(I_2, C) = (S → C C •, $)
I_6: goto(I_2, c) = {(C → c • C, $), (C → • cC, $), (C → • d, $) }
I_7: goto(I_2, d) = (C → d •, $)
I_8: goto(I_3, C) = (C → cC •, c/d)
: goto(I_3, c) = I_3
: goto(I_3, d) = I_4
I_9: goto(I_6, C) = (C → cC •, $)
: goto(I_6, c) = I_6
: goto(I_6, d) = I_7
DFA with LR(1) items is shown in Figure 5.23.

5. Consider the following grammar:

S → a A d | a c e | b A e
A → c

a. Construct the SLR(1) parsing table for this grammar. Is this grammar SLR(1)?
b. Construct the LR(1) parsing table for this grammar. Is this grammar LR(1)?
c. Construct the LALR(1) parsing table for this grammar. Is this grammar LALR(1)?

Solution:

a. 0 S″ → S
1 S → a A d
2 S → a c e
3 S → b A e
4 A → c

Follow(S) = {$}, Follow(A) = {d, e};

I_1 = {S″ → • S, S → • a A d, S → • a c e, S → • b A e},
I_2 = goto(I_1, S) = {S″ → S •},
I_3 = goto(I_1, a) = {S → a • A d, S → a • c e, A → • c}
I_4 = goto(I_1, b) = {S → b • A e, A → • c}
I_5 = goto(I_3, A) = {S → a A • d}

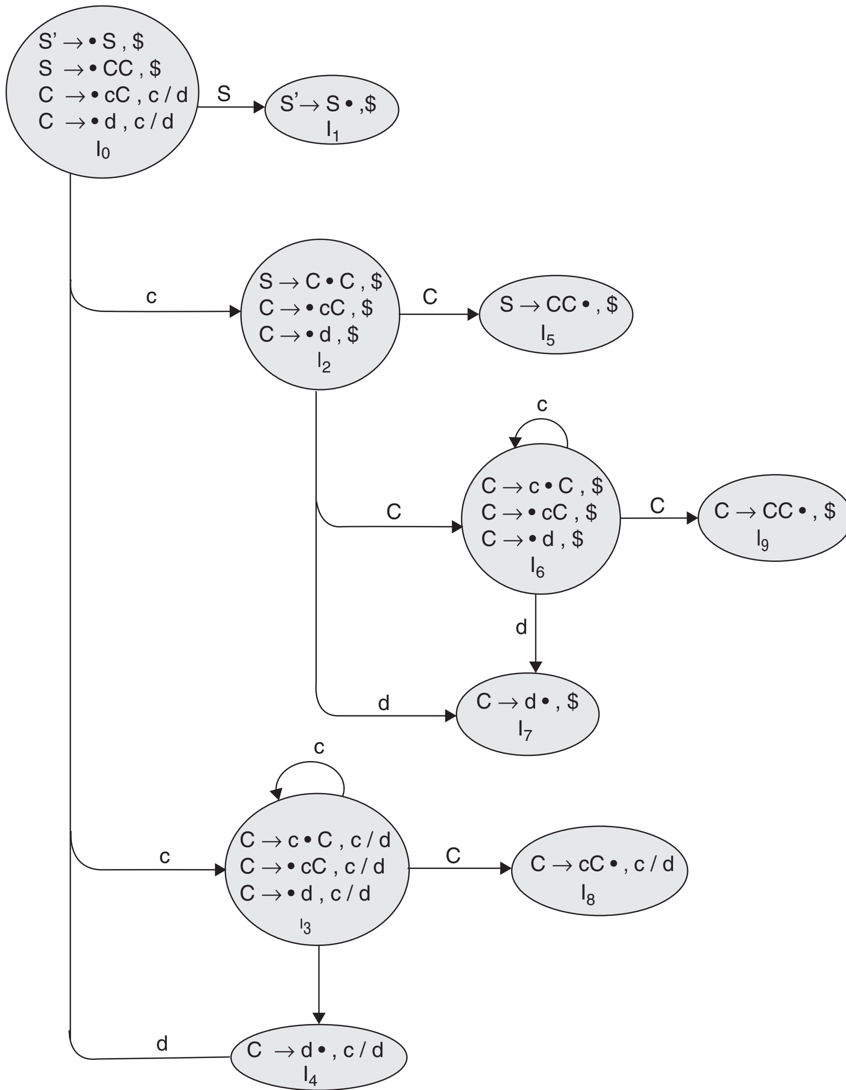


Figure 5.23

$I_6 = \text{goto}(I_3, c) = \{S \to a\ c \bullet e, A \to c \bullet\}$

$I_7 = \text{goto}(I_4, A) = \{S \to b\ A \bullet e\}$

$I_8 = \text{goto}(I_4, c) = \{A \to c \bullet\}$

$I_9 = \text{goto}(I_5, d) = \{S \to a\ A\ d \bullet\}$

$I_{10} = \text{goto}(I_6, e) = \{S \to a\ c\ e \bullet\}$

$I_{11} = \text{goto}(I_7, e) = \{S \to b\ A\ e \bullet\}$

States	Action					Goto		
	a	b	c	d	e	$	S	A
1	s_3	s_4					2	
2						accept		
3			s_6					5
4			s_8					7
5				s_9				
6				r_4	s_{10}/r_4			
7					s_{11}			
8				r_4	r_4			
9						r_1		
10						r_2		
11						r_3		

Since there is a shift/reduce conflict at state I_6, this grammar is not an SLR(1) grammar.

b. Set of LR(1) items are as follows:

I_1 = {[S″ → • S, $], [S → • a A d, $], [S → • a c e, $], [S → • b A e, $]},

I_2 = goto(I_1, S) = {[S″ → S •, $]},

I_3 = goto(I_1, a) = {[S → a • A d, $], [A → • c, d], [S → a • c e, $]}

I_4 = goto(I_1, b) = {[S → b • A e, $], [A → • c, e]}

I_5 = goto(I_3, A) = {[S → a A • d, $]}

I_6 = goto(I_3, c) = {[A → c •, d], [S → a c • e, $]}

I_7 = goto(I_4, A) = {[S → b A • e, $]}

I_8 = goto(I_4, c) = {[A → c •, e]}

I_9 = goto(I_5, d) = {[S → a A d •, $]}

I_{10} = goto(I_6, e) = {[S → a c e •, $]}

I_{11} = goto(I_7, e) = {[S → b A e •, $]}

States	Action					Goto		
	a	b	c	d	e	$	S	A
1	S_3	S_4					2	
2						accept		
3			S_6					5
4			S_8					7
5				S_9				
6				r_4	S_{10}			
7					S_{11}			
8					r_4			
9						r_1		
10						r_2		
11						r_3		

Since there is no conflict in the parsing table, this grammar is an LR(1) grammar.

c. Since there are no two states with the same core in the LR(1) parsing table, the LALR(1) table is the same as the LR(1) table, and the grammar is an LALR(1) grammar.

*6. Consider the following grammar.

$$S \rightarrow SS \mid a \mid \varepsilon$$

a. Construct collection of sets of LR(0) items for this grammar and draw its goto graph.
b. Indicate SR and RR conflicts in various states of LR(0) parser.

Solution: Draw DFA with LR(0) items as shown in Figure 5.24.
Here states I_0, I_1, and I_3 have SR conflicts. State I_3 has RR conflict also.

Summary

- Bottom-up parsing is the most efficient non-backtracking technique.
- There is no restriction on grammar for constructing LR parsers.
- Operator precedence parser is a simple bottom-up parser mainly used for parsing expression grammars.

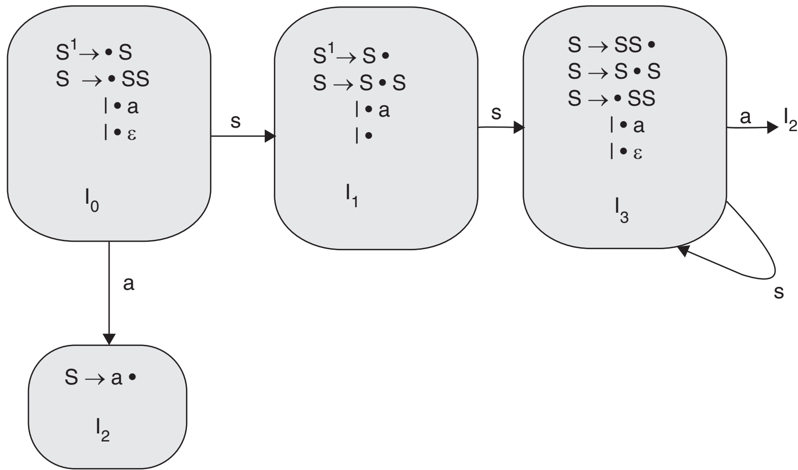


Figure 5.24 DFA with LR(0) items

- LR(0) parser is the simplest of all LR parsers but is least powerful. Hence practically it is not used.
- SLR(1) is simple to construct, better than LR(0), but can parse only a small class of grammars.
- LALR(1) is the most widely used LR parser. Yacc also uses this technique. It requires less table space compared to LR(1).
- LR(1) is most powerful bottom-up parsing technique. But requires less table space.
- SR conflict confuses the parser whether to take shift action/reduce action.
- RR conflict confuses the parser whether to reduce by first rule or second rule.
- If there are any SR/RR conflicts, we cannot construct LR parser.
- Every LL(1) grammar is LALR(1).
- LR(0) grammars are subset of SLR(1), LALR(1) and LR(1).
- SLR(1) grammars are subset of LALR(1) and LR(1).
- LALR10) grammars are subset of LR(1).
- Operator grammar can be ambiguous or unambiguous.
- We can construct parsers with ambiguous grammars also but need to resolve multiple entries.

Fill in the Blanks

1. Operator precedence parser is ______________ type of parser.
2. The parsing technique that is used in parsers generators is ______________.
3. If $A \rightarrow \alpha \bullet Bb$, a is in I then closure (I, a) is ______________.

4. If $A \rightarrow B$, a is a production in LR(1) items, then reduce operation is entered under ____________.
5. ____________ items are used for LALR(1) parsers.
6. The type of item used in $[A \rightarrow \bullet\, Aa, a \mid b \mid c]$ ____________.
7. Relation between grammars LL(1) ____________ LL(k).
8. Relation between grammars LL(1) ____________ LR(k).
9. If there are no RR conflicts in a LR(1) parser then it ____________ in LALR(1).
10. If there are no SR conflicts in a LR(1) parser then it ____________ in LALR(1).
11. Can we reduce every precedence relation table to function table? ____________.
12. The other name for bottom-up parser is ____________.
13 YACC is ____________.
14. Operator precedence parser ____________ handle operator with different precedence.
15. ″0″ or "1" in LR items represent ____________.

Objective Question Bank

1. Consider the grammar given below

$$A \rightarrow Ba \mid Aa \mid c$$
$$B \rightarrow B\,b \mid A\,b \mid d$$

Convert equivalent operator grammar

(a) $A \rightarrow Ba\,D \mid c\,D$
$D \rightarrow a\,D \mid \in$
$B \rightarrow B\,b \mid A\,b \mid d$

(b) $A \rightarrow Ba\,D \mid c\,D$
$D \rightarrow a\,D \mid \in$
$B \rightarrow c\,D\,b\,E \mid d\,E$
$E \rightarrow b\,E \mid a\,D\,b\,E \mid \in$

(c) $A \rightarrow Ba\,D \mid c\,D$
$D \rightarrow a\,D \mid \in$
$B \rightarrow A\,b\,E \mid d\,E$
$E \rightarrow b\,E \mid \in$

(d) none

2. For a given grammar if SLR(1) has n1 states, LALR(1) has n2 states, LR(1) has n3 states which of the following is true?

(a) n2 = n3 (b) n2 <= n3 (c) n2 < n3 (d) none

3. For a given grammar if SLR(1) has n1 states, LALR(1) has n2 states, LR(1) has n3 states which of the following is true?

(a) n3 > n1 (b) n3 >= n1 (c) n3 = n1 (d) none

4. Consider the grammar given below

$A \rightarrow SB \mid S$
$B \rightarrow ;\,S\,B \mid ;\,S$
$S \rightarrow a$

Convert the equivalent operator grammar

(a) $A \to S; A \mid S; S \mid S$
$S \to a$

(b) $A \to S; B \mid S; S \mid S$
$B \to ; SB \mid ; S$
$S \to a$

(c) $A \to S; A \mid S; S \mid S$
$B \to ; S B \mid ; S$
$S \to a$

(d) does not exist

5. What is the precedence relation between**;** & a?

(a) less (b) greater (c) equal (d) none

*6. Consider SLR(1) and LALR(1) tables for CFG. Which of the following is false?
(a) Goto of both tables may be different
(b) Shift entries are identical in both tables
(c) Reduce entries in tables may be different
(d) Error entries in tables may be different

*7. For a given grammar if SLR(1) has n1 states, LALR(1) has n2 states, which of the following is true?

(a) n2 < n2 (b) n1 = n2 (c) n1 > n2 (d) none

*8. Consider the grammar $S \to CC$, $C \to cC \mid d$ is

(a) LL(1) (b) SLR(1) but not LL(1)
(c) LALR(1) but not SLR(1) (d) LR(1) but not LALR(1)

*9. Which of the following is the most powerful parsing method?

(a) LL(1) (b) CLR(1) (c) SLR(1) (d) LALR(1)

10. Consider the grammar $S \to SS \mid d \mid \varepsilon$ is

(a) LL(1) (b) LR(1) (c) LR(0) (d) none

*11. Consider the grammar $E \to E + n \mid E \times n \mid n$. For a sentence "n + n × n," the handles in the right sentential form of reduction are

(a) n, E + n and E + n × n (b) n, E + n and E + E × n
(c) n, n + n and n+ n × n (d) n, E + n and E × n

*12. Consider the grammar $S \to (S) \mid a$. If SLR(1) has n1 states, LR(1) has n2 states, LALR(1) has n3 states, which of the following is true?

(a) n1<n2<n3 (b) n1=n3<n2 (c) n1=n2=n3 (d) n1>n3>n2

*13. An LALR(1) parser for "G" can have SR conflicts if and only if

(a) The SLR(1) has SR conflicts (b) The LR(1) has SR conflicts
(c) The LR(0) has SR conflicts (d) The LALR(1) has RR conflicts

14. Which of the following is true?
    (a) every SLR(1) is LR(0)
    (b) every LL(1) is LR(1)
    (c) every LL(1) is SLR(1)
    (d) every LL(1) is LR(0)
15. The LL(1) and LR(0) techniques
    (a) are both same in power
    (c) are not same in power
    (b) both simulate RMD
    (d) none
16. If a grammar is SLR(1) then [ ]
    (a) it may have S/R conflict
    (c) It may have R/R conflict
    (b) It will not have any conflict
    (d) it will not have S/R but may have R/R conflict
17. Using LR(0) items we can construct. [ ]
    (a) SLR parsing table
    (c) CALR parsing table
    (b) LALR parsing table
    (d) none.
18. I_0 : S→ • aA, $ A→ • ε, a the state I0 has [ ]
    (a) S/R conflict
    (c) S/S conflict
    (b) R/R conflict
    (d) None
19. The following grammar is [ ]

    S →ABC
    A → 0A1 | ε
    B → 1B | ε
    C → 1C0 | ε

    (a) LL(1) (b) LR(0) (c) not LL(1) (d) not LL(1) and not LR(0)
20. Consider the grammar [ ]

    E → A | B
    A → a | c
    B → b | c

    (a) LR(0) (b) LR(0) & LR(1) (c) LR(1),SLR(1) (d) none

Exercises

1. Convert the following grammar to operator grammar.

    P → S R | S
    R → θ S R | θ S
    S → W θ s | W
    W → L ↑ W | L
    L → a.

   Prepare a precedence relation table and parse the i/p string using the above grammar.

    a ↑ a θ a ↑ a $

2. Check whether the following grammar is LR(0).

$$S \rightarrow S\#$$
$$S \rightarrow dA \mid aB$$
$$A \rightarrow bA \mid c$$
$$B \rightarrow bB \mid c$$

3. Check whether the following grammar is LR(0).

$$S \rightarrow (L)$$
$$S \rightarrow x$$
$$L \rightarrow S$$
$$L \rightarrow L, S$$

4. Check whether the following grammar is LR(0), SLR(1).

$$S \rightarrow A \mid B$$
$$A \rightarrow aAb \mid ab$$
$$B \rightarrow aBbb \mid abb$$

5. Check whether the following grammar is LR(0), SLR(1).

$$E \rightarrow bEa \mid aEb \mid ba$$

6. Check whether the following grammar is LR(1).

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

7. Check whether the following grammar is SLR(1).

$$S \rightarrow X\,Y\,a\,\#,$$
$$X \rightarrow a \mid Yb,$$
$$Y \rightarrow \in \mid c$$

8. Design the SLR parser with the following ambiguous grammar.

$$S \rightarrow i\,S \mid iSeS \mid a$$

9. Check if the following grammar is SLR(1), LR(1).

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid id$$
$$R \rightarrow L$$

10. Check $L(G) = \{ww^R \mid w \in [a+b]^+\}$ is LR(1)

$$S \rightarrow aSa \mid bSb \mid a \mid b$$

11. How many conflicts occur in DFA with LR(1) items for the following grammar?

$$S \rightarrow SS \mid a \mid c$$

12. Find closure of ($E^1 \rightarrow \bullet E, \$$) in G.

$E \rightarrow E + T \mid T;$ $T \rightarrow T * F \mid F;$ $F \rightarrow id$

13. Find the number of conflicts, if any, in DFA with LR(1) item.

$S \rightarrow A;$ $A \rightarrow AB \mid \varepsilon;$ $B \rightarrow aB \mid b$

14. Check if G is SLR(1).

$S \rightarrow AB \mid \varepsilon;$ $A \rightarrow aASb \mid a;$ $B \rightarrow bS$

15. Check if G is LR(1).

$S \rightarrow aB \mid ab;$ $A \rightarrow aAB \mid a;$ $B \rightarrow ABb \mid b$

Key for Fill in the Blanks

1. Bottom up
2. LALR(1)
3. ϕ
4. a
5. LR(1)
6. LR(1)
7. $\subset$
8. $\subset$
9. may occur
10. Will not occur
11. No. if digraph has cycle
12. SR parser
13. Yet another compiler compiler.
14. cannot
15. presence or absence of lookahead.

Key for Objective Question Bank

1. b	2. b	3. b	4. a	5. a
6. a	7. b	8. a	9. b	10. d
11. d	12. b	13. b	14. b	15. c
16. b	17. a	18. a	19. d	20. d

CHAPTER 6

Syntax-Directed Translation

Syntax-directed definition is a generalization of a context free grammar (CFG), but effectively is an attribute grammar. Syntax-directed translation describes the translation of language constructs guided by CFGs. They are used in automatic tools like YACC.

CHAPTER OUTLINE

6.1 Introduction
6.2 Attributes for Grammar Symbols
6.3 Writing Syntax-Directed Translation
6.4 Bottom-Up Evaluation of SDT
6.5 Creation of the Syntax Tree
6.6 Directed Acyclic Graph (DAG)
6.7 Types of SDTs
6.8 S-Attributed Definition
6.9 Top-Down Evaluation of S-Attributed Grammar
6.10 L-Attributed Definition
6.11 Converting L-Attributed to S-Attributed Definition
6.12 YACC

Syntax-directed translation is an extension of context free grammars (CFGs). This helps the compiler designer to translate the language constructs directly by attaching semantic actions or subroutines. In this chapter, we discuss what is syntax-directed translation (SDT), how to write SDT″s, and how to evaluate semantic actions. We also discuss the different types of SDT″s and how to write S-attributed definition and L-attributed definition in detail. Converting L-attributed to simple attributed is also discussed.

6.1 Introduction

So far we have discussed parsing. Given a grammar, how a parser checks the syntax of programming language construct. Now let us look at another technique called syntax-directed translation with which we can combine many tasks along with parsing. Along with each production of the grammar, we attach a semantic action. The grammar together with semantic action is called syntax-directed translation (SDT).

$$A \rightarrow \alpha + \{action\} = SDT$$

Semantic action or translation rule or semantic rule or action is the same. We can attach semantic actions for grammar rules for performing different tasks. Examples are:

1. To store/retrieve type information in symbol table
2. To perform consistency checks like type checking, parameter checking, etc.
3. To issue error messages
4. To build syntax trees
5. To generate intermediate or target code

There are two ways of defining semantic rules. One is syntax-directed definition (SDD), which is a high-level language specification for translation. This hides many implementation details and frees the user to explicitly specify the order of evaluation. The other scheme syntax-directed translation (SDT) specifies the order in which semantic rules are to be evaluated. So they allow some implementation details to be shown. Here we use the names SDD or SDT interchangeably but the basic difference between the two is in specifying the evaluation order. With SDD/SDT, we parse the input, create parse tree, and traverse the parse tree as needed to evaluate semantic actions at the parse tree nodes. Evaluation of semantic rules may result in storing information in symbol table, issue error diagnostics or other activities.

Here we discuss syntax-directed translations. They are also called attribute grammars, where

- ◆ Each grammar symbol is assigned certain attributes.
- ◆ Each production is augmented with semantic rules which are used to define attribute values

6.2 Attributes for Grammar Symbols

Syntax-directed translation is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes. If we consider parse tree node as a record for holding information, then an attribute is the name of each field in record. A grammar symbol can have "n" attributes. The attributes for grammar symbol can be a type, value, address, pointer, or a string. The attributes that can be associated with a grammar symbol can be classified into two types as defined below.

Attribute types:

1. **Synthesized attribute—**The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

   EX: $A \rightarrow XYZ$

   Here in Figure 6.1 we assume that there is an attribute ."s" with each grammar symbol. Then if it is evaluated as $A \bullet s = f(X \bullet s \mid Y \bullet s \mid Z \bullet s)$ that is,

   $A \rightarrow XYZ \quad \{A \bullet s = X \bullet s + Y \bullet s + Z \bullet s\}$

   Then ."s" is called the synthesized attribute.

2. **Inherited attribute—**The value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

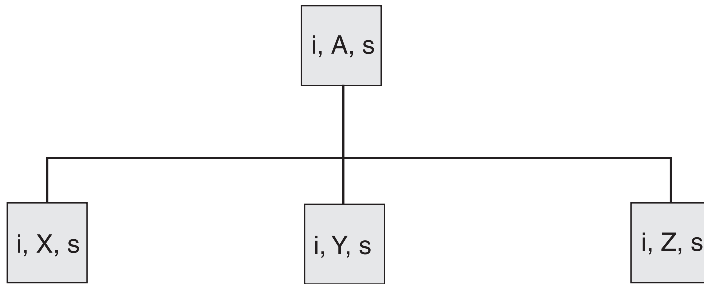


Figure 6.1 Parse Tree for Rule A → XYZ

In Figure 6.1, we assume that there is an attribute "**i**" with each grammar symbol. Then if it is evaluated as Y • i = f(X • i | Z • i | A • i| A • s) that is,

A → XYZ {Y • i = A • i + X • i + Z • i}
Then "•i" is called the inherited attribute.

Once we consider an attribute as a synthesized attribute, wherever it is used, it should preserve the basic property of the synthesized attribute, that is, the value at parent is evaluated with attribute values of its children. We cannot use the same attribute as synthesized for some time and inherited for some time. Generally, we don't specify explicitly, whether an attribute is synthesized or inherited, by looking at semantic actions, that is, dependency of attribute, we need to understand whether it is synthesized or inherited.

6.3 Writing Syntax-Directed Translation

Let us see how to write SDT. Let us try to understand with an example. Consider parsing the input string "**1 + 2 * 3.**" Recollect, how a bottom-up parser parses this string. It uses the following grammar:

E → E + T | T
T → T * F | F
F → num

Parses the string. The result is a parse tree as shown in Figure 6.2.

Now along with parsing, that is, checking the syntax of a string, if we want to perform the evaluation of an expression, let us see how to write SDT. So here the additional task that is combined with parsing is evaluation of expression. For this we need to attach semantic rules. Parse tree gives us the *order* in which reductions are carried out by the parser. If we follow the tree from bottom to top we get the order. For example in Figure 6.2, the order of reductions is number to F, F to T, T to E, next number to F, F to T, then number to F, T * F to T, E + T to E. This is very useful to think about semantic actions. Whenever parser reduces by rule, we must think what should be the corresponding action to get the desired output. In SDT, initially we assume that whenever a bottom-up parser reduces by production, the corresponding action is automatically carried out.

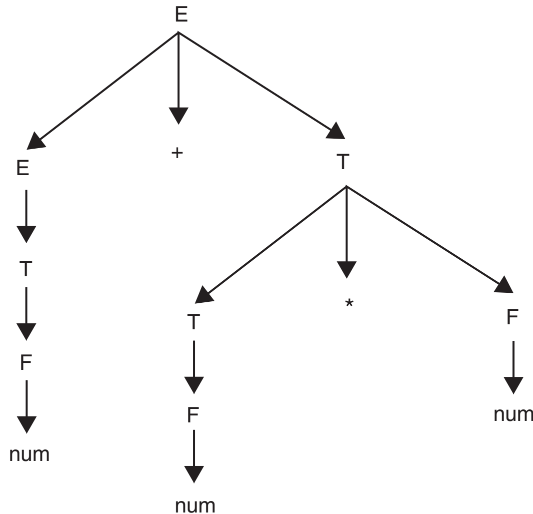


Figure 6.2 Parse Tree for Expression Grammar

How do we define semantic rules? Here first we need to find out what additional information is required. When a bottom-up parser sees the input string "1 + 2 * 3," on reading the first symbol "1," it gets a token from the lexical analyzer as "num." Now the first step performed by the parser is token "num" reduced to F by using production F → num. If it is only checking the syntax, this reduction is enough. But along with this, if we want to take care of the evaluation of expression, reducing first token to nonterminal alone may not be sufficient. To take care of the evaluation of expression at any time, we have to propagate additional information along with nodes of the parse tree. That additional information here is lexeme value of token "num." So to store the additional information, first we need to assume an attribute with grammar symbol. We attach attribute to grammar symbol with"." We assume that there is an attribute ".val" with each grammar symbol. Once attribute is present, it will be for all grammar symbols like E.val, T.val, F.val. Now semantic actions define how to evaluate this attribute values at any node in the parse tree.

For example, when the parser reduces "num" to F by using production F → num, we need to store the attribute value at node F. Hence, we attach semantic action as {F**.**val= num**.** lval;} where num.lval is lexeme value of token num. It can be any value at run time like "1" or "2" or "3" or "1000." So the result is as shown in Figure 6.3.

The next step in parsing is reducing F to T. So here semantic action is forwarding lexeme value from F to T by attaching semantic action {T•val = F•val;}. The same thing happens with T to E also. The result in shown below in Figure 6.4.

Next the parser reads the next token "+," and then "2." Now it gets a token num and reduces that to F. As translation for T → F is already defined, value "2" is propagated to node F and then to node T also. Next the parser reads the next token "*," then "3." Now it gets a token num and reduces that to F. Now the parser reduces T * F; here the semantic action is to evaluate the expression and store the result. So SDT is defined as follows:

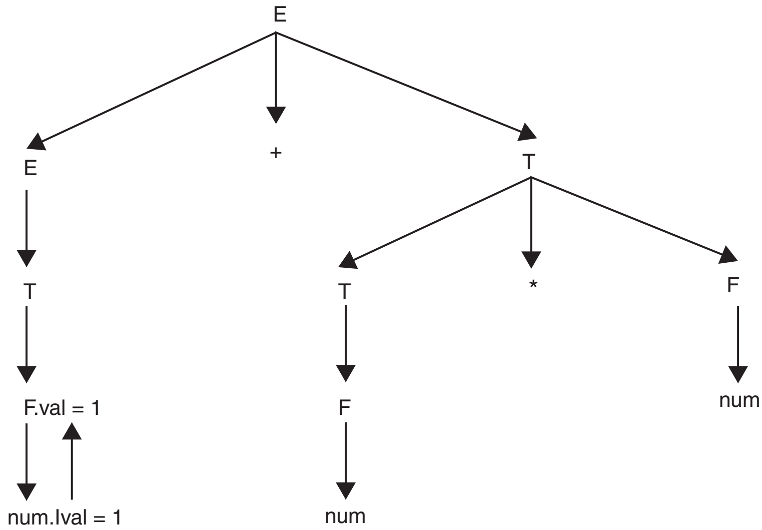


Figure 6.3 Parse Tree with Attribute Values

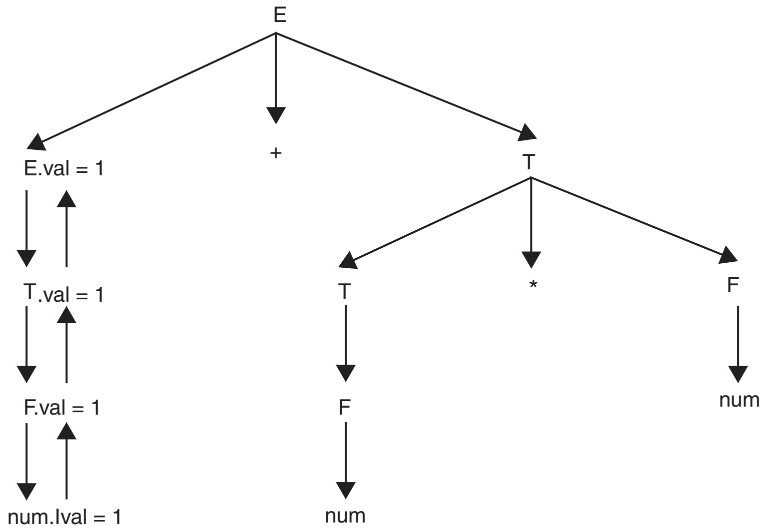


Figure 6.4 Parse Tree with Attribute Values

Example 1:

SDT for evaluation of expressions/Desk calculator

$E \rightarrow E_1 + T$	{E•val = E_1•val + T•val}
$E \rightarrow T$	{E•val = T•val}
$T \rightarrow T_1 * F$	{T•val = T_1•val * F•val}
$T \rightarrow F$	{T•val = F•val}
$F \rightarrow id$	{F•val = num•lval}

Here E or E_1 is the same. Just to understand the semantic action, this notation is used. Here num.lval is the lexeme value of token num. The result of carrying out actions is shown in Figure 6.5.

The parse tree that shows attribute values at each node is called annotated or decorated parse tree as shown in Figure 6.5.

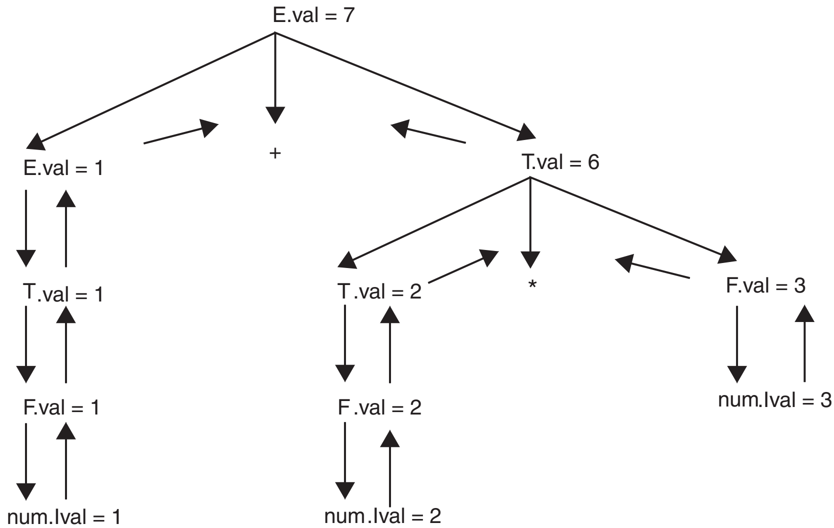


Figure 6.5 Annotated Parse Tree

New semantic actions can be added without disturbing the existing translations.

As we have seen in the first example, let us understand the procedure for writing SDTs. It involves the following three steps:

1. Define grammar for input string.
2. Take a simple string and draw a parse tree.
3. Attach semantic actions by looking at the expected output.

Parser always starts with grammar. Here the second step is constructing the parse tree. Drawing the parse tree has two advantages. One is to check whether defined grammar is right or wrong, another is to get order in which reductions are carried out. This order is important to define semantic action. Follow the same order, and think about semantic action. Let us understand the procedure with one more example.

Example 2:

Write an SDT for converting infix expressions to post fix form, that is, given an input string "a + b * c" should give "abc * +" as output.

Let us see how to write the SDT. The first step is define the grammar. As input is the same as the previous example, the same grammar can be used. Semantic actions are not the same because the expected output is different. Now take a simple string "a + b" and draw a parse tree as shown Figure 6.6.

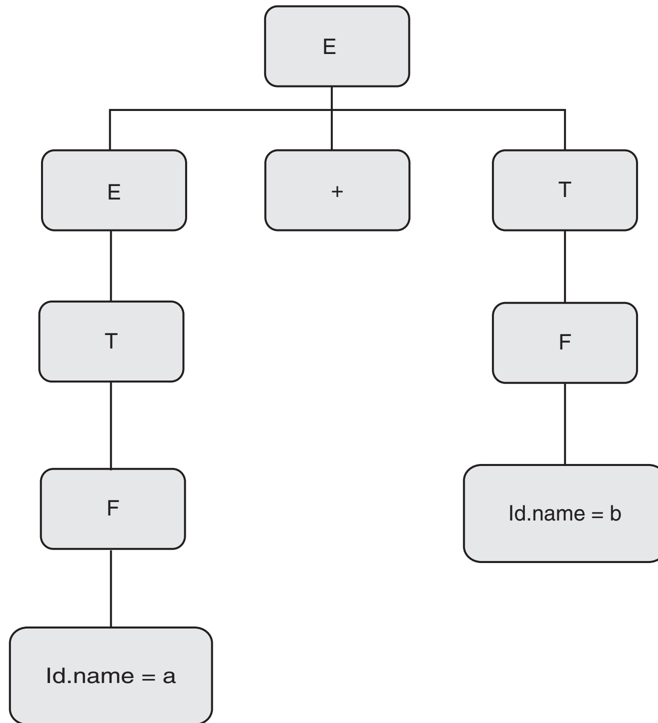


Figure 6.6 Parse Tree for String "a+b"

Look at Figure 6.6.

1. The parser first reads "a"; it will be matched with the token "id." Now look at output, "ab +"; whatever is read should appear in output. So let the semantic action with reduce action F → id be {print("id•name");}, where "id.name" is the name of the identifier, whether it is "a" or "b "or "c."
2. Now look at the next step performed by the bottom-up parser T → F. Here during this reduction, do we have to propagate any additional information? No. As there is no additional information to be propagated, there is no need of assuming attributes to grammar symbols. So with reduce action T → F, there is no need of any semantic action. Hence, we define the translation as T → F {}. Same thing happens with E → T also.
3. Next parser reads "+." Just with E+, it cannot perform any reduce action; Hence reads "b."

   Now "b" is reduced to F, here "b" is printed out as F→ id {print("id•name");}. Then it reduces, F to T, and there is no action.
4. The next reduction is "E+T" by E. Here already "ab" is printed and the only left out character to be printed is the operator "+." So whenever an expression is reduced, print operator. Add this as semantic action with the rule. The resulting SDT is shown below

SDT for converting infix to postfix expression

```
E → E1 + T      {print("+");}
E → T                     { }
T → T1 * F      {print("*");}
T → F                     { }
F → id   {print("id•name");}
```

Let us understand how to define semantic action by taking one more example. The next phase in parsing is semantic analysis. The semantic analyzer's main function is type checking. Let us write a simple type checker.

Example 3:
Write an SDT for a simple type checker. Here we assume that our type checker is very simple and recognizes only three types—int, bool, and err. Here "int" type is for integers, "bool" type is for Boolean values True/false and "err" type is for error. Here we would like to have a type checker that checks for expressions of the form: (8 + 8) = = 8. Given such expression, it should check the type compatibility of operands and the final result is true/false, that is, bool.

1. Define grammar. Grammar is for expressions. Why do we have to consider unambiguous grammars? Let us take ambiguous grammar. We have already discussed how to construct parsers with ambiguous grammars also. So define grammar as follows.

```
E → E+ E
E → E and E
E → E = = E
E → true
E → false
E → num
E → (E)
```

2. Take the input string and draw a parse tree for string "(8 + 8)==8" as in Figure 6.7.
3. Now let us attach semantic actions. First the parser reads "(" and then "8." Here we are not interested with value; what we are writing is a type checker. A type checker first should collect type information, and then should verify type compatibility of operands. Here when "8" is read, we need to collect and store type information. For storing type information, we assume that there is an attribute ."type" with each grammar symbol. So when token num is reduced to E after reading "8," we define semantic rule as { E•type=int;} because "8" is integer type. Similarly for true/false, assign the type as bool.
4. Now the parser reads "+," then "8," and reduces "8" to E. Now the next reduction is E + E to E. Here type checker should verify types. To distinguish between left hand side nonterminal E and operands on right hand side Es,"we take them as E_1, E_2, and E_3 respectively. When expression with arithmetic addition is reduced, type check could be checking if both operands are integer type. If they are integers, it returns integers, or else it returns an error type. The above semantics can be implemented by attaching the semantic action as follows:

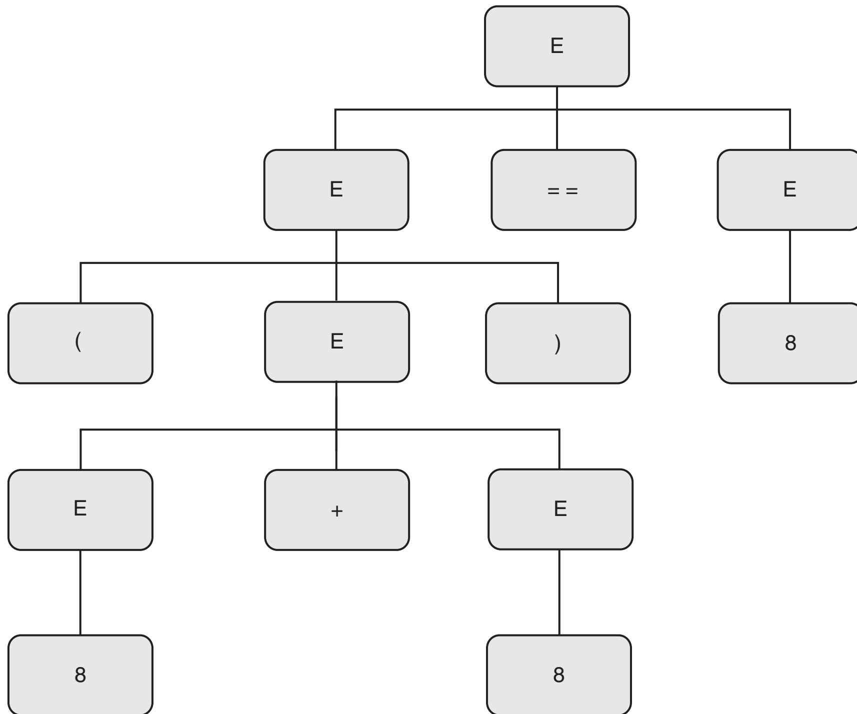


Figure 6.7 Parse Tree for String "(8 + 8) = = 8"

```
E1 → E2 + E3 { if( E2•type = =int and E3•type = =int)
           then E1•type = int else E1.type=error;}
```

5. Similarly, if an expression with Boolean operator is reduced, type check could be checking if both operands are integer type or Boolean type. If they are int/bool, return bool, else return error type. The above semantics can be implemented by attaching the semantic action as follows:

```
E1 → E2  and E3 { if( E2•type = = E3•type ) && (E2•type = =int/bool))
              then E1•type = bool else E1.type=error;}}
```

6. Similarly, if an expression with relational operator is reduced, type check could be checking if both operands are integer type or Boolean type. If they are int/bool, return bool, else return error type. The above semantics can be implemented by attaching the semantic action as follows:

```
E1 → E2  ==  E3 { if( E2•type = = E3•type )
              then E1•type = bool else E1.type = error;}
```

The SDT can be defined as follows:

SDT for simple type checking

```
E1 → E2 + E3    {if( E2•type = =int and E3•type = =int)
                then E1•type = int else E1.type=error;}
E1 → E2 and E3  {if( E2•type = = E3•type ) && (E2•type = =int/bool))
                then E1•type = bool else E1.type=error;}
E1 → E2 = = E3  {if( E2•type = = E3•type )
                then E1•type = bool else E1.type = error;}
E → true        {E•type = bool}
E → false       {E•type = bool}
E → num         {E•type = int}
E1 → (E2)       {E1•type = E2•type}
```

The annotated parse tree for the string (8 + 8) == 8 is as follows in Figure 6.8.
Now let us see how to understand a given SDT.

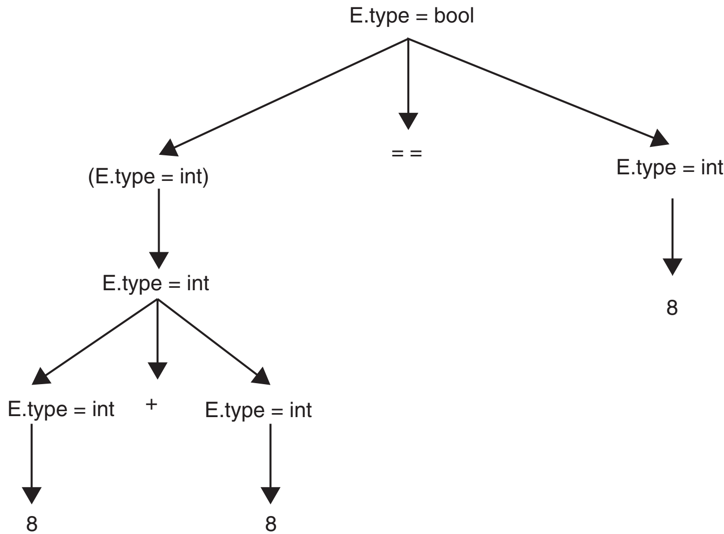


Figure 6.8 Annotated Parse Tree for String "(8 + 8) = = 8"

6.4 Bottom-Up Evaluation of SDT

Given an SDT, we can evaluate attributes even during bottom-up parsing. To carry out the semantic actions, parser stack is extended with semantic stack. The set of actions performed on semantic stack are mirror reflections of parser stack. Maintaining semantic stack is very easy.

During shift action, the parser pushes grammar symbols on the parser stack, whereas attributes are pushed on to semantic stack.

During reduce action, parser reduces handle, whereas in semantic stack, attributes are evaluated by the corresponding semantic action and are replaced by the result.

For example, consider the SDT

$A \rightarrow X\,Y\,Z \qquad \{A \cdot a := f(X \cdot x, Y \cdot y, Z \cdot z);\}$

The stack contents would be as shown in Figure 6.9.

symbol	val	
...	...	
X	X.x	val [top – 2]
Y	Y.y	val [top – 1]
Z (← top)	Z.z	val [top]

Figure 6.9 Stack Content While Parsing

Strictly speaking, attributes are evaluated as follows

$A \rightarrow X\,Y\,Z \qquad$ {val[ntop] := f(val[top - 2], val[top - 1], val[top]);}

Evaluation of Synthesized Attributes

- Whenever a token is shifted onto the stack, then it is shifted along with its attribute value placed in val[top].
- Just before a reduction takes place the semantic rules are executed.
- If there is a synthesized attribute with the left-hand side nonterminal, then carrying out semantic rules will place the value of the synthesized attribute in val[ntop].

Let us understand this with an example:

```
E → E1 "+" T       {val[ntop] := val[top-2] + val[top];}
E → T              {val[top] := val[top];}
T → T1 "*" F       {val[ntop] := val[top-2] * val[top];}
T → F              {val[top] := val[top];}
F → "(" E ")"      {val[ntop] := val[top-1];}
F → num            {val[top] := num.lvalue;}
```

input string 7 * 7

Figure 6.10 shows the result of shift action. Now after performing reduce action by $E \rightarrow E * T$ resulting stack is as shown in Figure 6.11.

Along with bottom-up parsing, this is how attributes can be evaluated using shift action/reduce action.

Parser stack	Semantic stack
T	T.val=7
+	
E	E.val=7
$	$

Figure 6.10 Stack Content While Parsing

Parser stack	Semantic stack
E	E.val = 12
$	$

Figure 6.11 Stack Content After Reduction

∗**Example 4:**

Consider the following SDT.

```
S → xxW          { print("1");}
      | y        { print("2");}
W → Sz           { print("3");}
```

If an SR parser carries out the translations specified, immediately after reducing with rules of grammar, what is the result of carrying out the above translations on an input string "x^4 y z^2"?

Solution: Given an SDT, to trace out SDT on an input string, take the string and draw a parse tree. Now look at the order in which reductions are performed by the bottom-up parser. Whenever the parser reduces by rule, carry out the attached semantic action, that gives you the result. For the input string "xxxxyzz," the parse tree is as shown in Figure 6.12

So here the first reduction is y to S, result is print (2). Next reduction is Sz to W, result of semantic action is print (3); next reduction is xxW to S, result is print(1); Next reduction is Sz to W, result of semantic action is print (3); next reduction is xxW to S, result is print(1);

So the final output is: **23131.**

∗**Example 5:**

Consider the SDT given below.

```
E → E * T        { E•val = E•val * T•val;}
E → T            { E•val = T•val;}
T → F - T        { T•val = F•val - T•val;}
T → F            { T•val = F•val;}
F → 2            { F•val = 2;}
F → 4            { F•val = 4;}
```

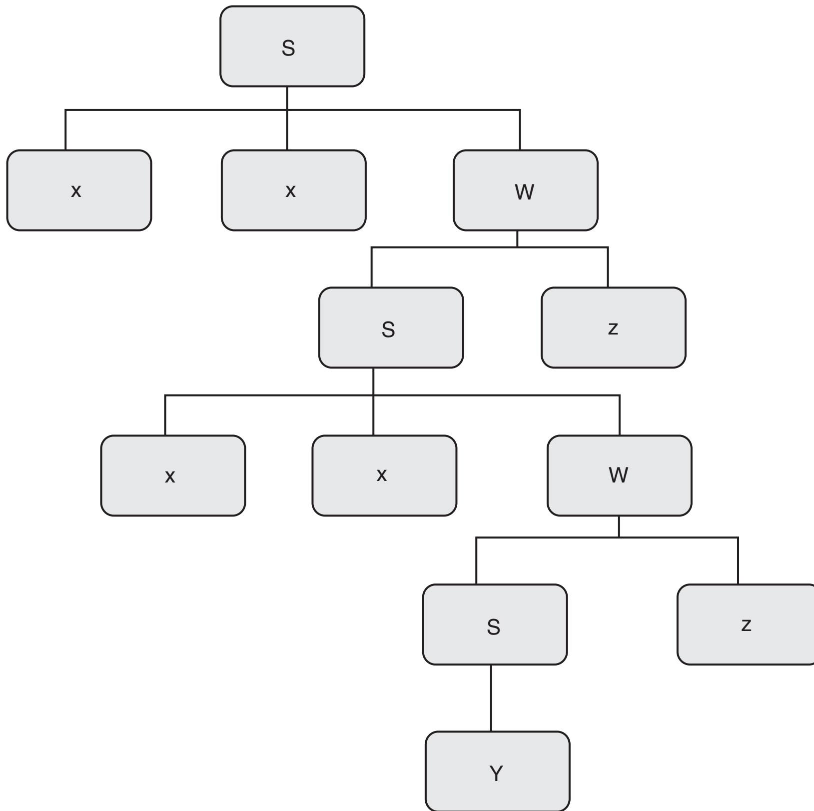


Figure 6.12 "xxxxyzz" Parse Tree

a. Using the SDT construct parse tree and evaluate string 4 – 2 – 4 * 2
b. It is also required to compute the total number of reductions performed to parse the given input string. Modify the SDT to find the number of reductions.

Solution: a. To evaluate the expression, there are two ways. The simple method is to take the precedence and associativity of operators defined in the grammar. "–" has higher precedence than "*." "–" is right associative and "*" is left associative. By considering this it can be evaluated without having the parse tree as follows:

$$(4 - (2 - 4)) * 2 = 4 - (-2) * 2 = 12.$$

The second method is the blind method, that is, using the parse tree. Construct a parse tree for the expression that takes care of precedence and associativity. Evaluate the expression from the parse tree as shown in Figure 6.13.

First evaluate inner sub tree (2 – 4) = –2. The next sub tree to be evaluated is 4 – (–2) = 6. The final sub tree to be evaluated is 6 * 2 = 12.

b. Modifying SDT is similar to writing SDT. To find the number of reductions, one simple way is to use a global variable "count" and increment for every reduction. But this is not an advisable solution.

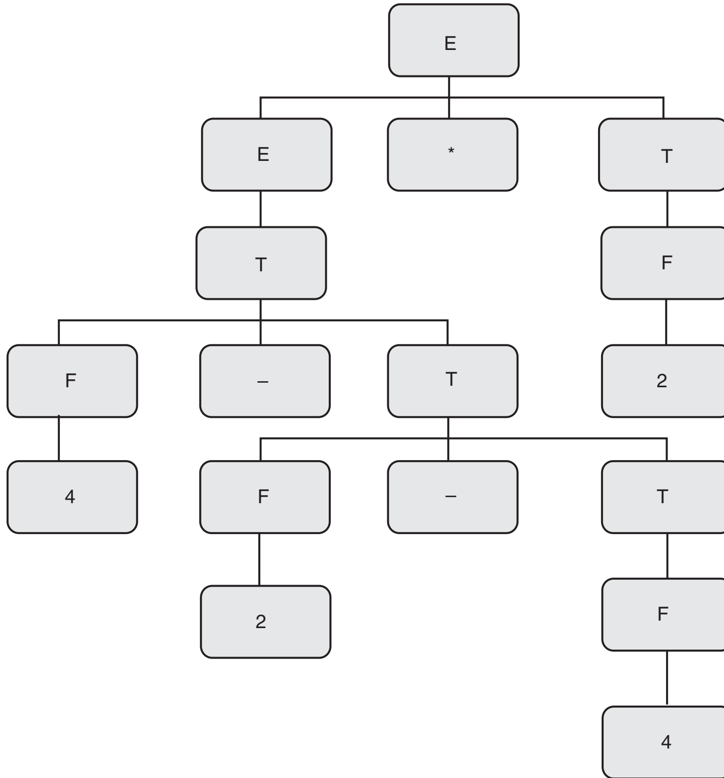


Figure 6.13 Parse Tree for “4 - 2 - 4 * 2”

Other alternative is to use an attribute ″red,″ to keep track of the number of reductions. The advantage of taking the number of reductions as attribute is that we can store the number of reductions at any node in the parse tree with the attribute. Now let us see how to define semantic actions for evaluation of attribute .″red.″

Use the parse tree shown in Figure 6.13, traverse the parse tree, whenever any reduction is performed, define rule for finding number of reductions.

For example, the first reduction is reducing terminal “4” to F. So here the number of reductions performed is one. So whether “2” is reduced or “4” is reduced, the number of reduction is one. Hence, define semantic actions as follows:

$F \rightarrow 2$ { F•val = 2; F•red = 1;}
$F \rightarrow 4$ { F•val = 4; F•red = 1;}

The next reduction is F to T; when this is reduced, the number of reductions is the number of reductions at child node plus one. Hence, define semantic actions as follows:

$T \rightarrow F$ { T•val = F•val; T•red = F•red + 1;}

Similarly, for T to E, the same thing is applicable.

$E \rightarrow T \quad \{ E{\bullet}val = T{\bullet}val;\ E{\bullet}red = T{\bullet}red + 1; \}$

The next reduction is F-T to T; when this is reduced, the number of reductions is the number of reductions at the left child plus the number of reductions at the right child plus one. Hence, define semantic actions as follows:

$T \rightarrow F - T \quad \{ T{\bullet}val = F{\bullet}val - T{\bullet}val;\ T{\bullet}red = F{\bullet}red + T{\bullet}red + 1; \}$

Similarly, for E * T to E, the same thing is applicable.

$E \rightarrow E * T \quad \{ E{\bullet}val = E{\bullet}val * T{\bullet}val;\ E{\bullet}red = E{\bullet}red + T{\bullet}red + 1; \}$

Hence, the final SDT after modifications is as follows:

$E \rightarrow E * T \quad \{ E{\bullet}val = E{\bullet}val * T{\bullet}val;\ E{\bullet}red = E{\bullet}red + T{\bullet}red + 1; \}$
$E \rightarrow T \quad \{ E{\bullet}val = T{\bullet}val;\ E{\bullet}red = T{\bullet}red + 1; \}$
$T \rightarrow F - T \quad \{ T{\bullet}val = F{\bullet}val - T{\bullet}val;\ T{\bullet}red = F{\bullet}red + T{\bullet}red + 1; \}$
$T \rightarrow F \quad \{ T{\bullet}val = F{\bullet}val;\ T{\bullet}red = F{\bullet}red + 1; \}$
$F \rightarrow 2 \quad \{ F{\bullet}val = 2;\ F{\bullet}red = 1; \}$
$F \rightarrow 4 \quad \{ F{\bullet}val = 4;\ F{\bullet}red = 1; \}$

The annotated parse tree for the input string "4 – 2 – 4 * 2" is shown in Figure 6.14.

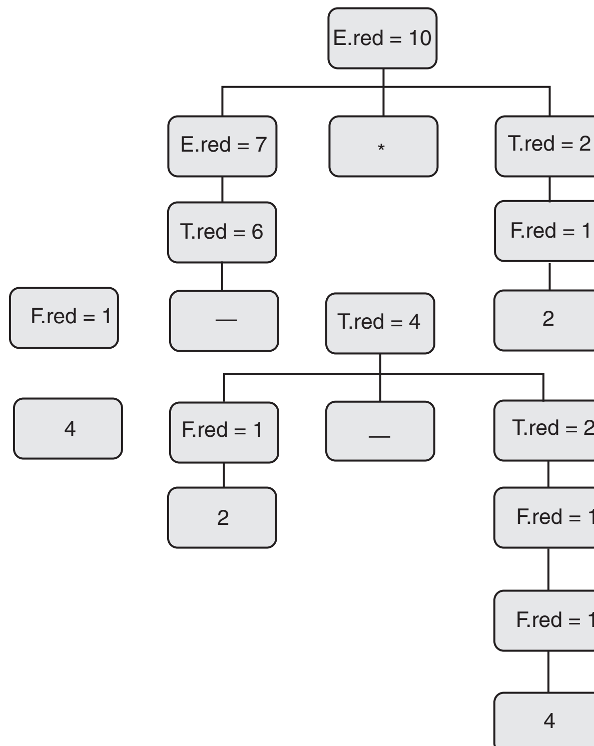


Figure 6.14 Decorated Parse Tree for "4 - 2 - 4 * 2"

Example 6:

Write an SDT to count the number of binary digits. For example, "1000" is 4.

Solution:

First define the grammar for a binary number. Binary digit "B" can be either "1" or "0." The number can be a list of binary digits. The list can recursively define bits "B." So grammar is as follows:

$N \rightarrow L$
$L \rightarrow LB$
$L \rightarrow B$
$B \rightarrow 0$
$B \rightarrow 1$

The next step is to take a small string and draw a parse tree. So take 1000 and draw a parse tree as shown in Figure 6.15.

The third step in SDT is to define semantic actions. Assume "count" as an attribute to count digits.

For example, the first reduction is the reducing terminal "1" to B. So here the number of digits, that is, the digit count is one. So whether "0" is reduced or "1" is reduced, the count is one. Hence, define semantic actions as follows:

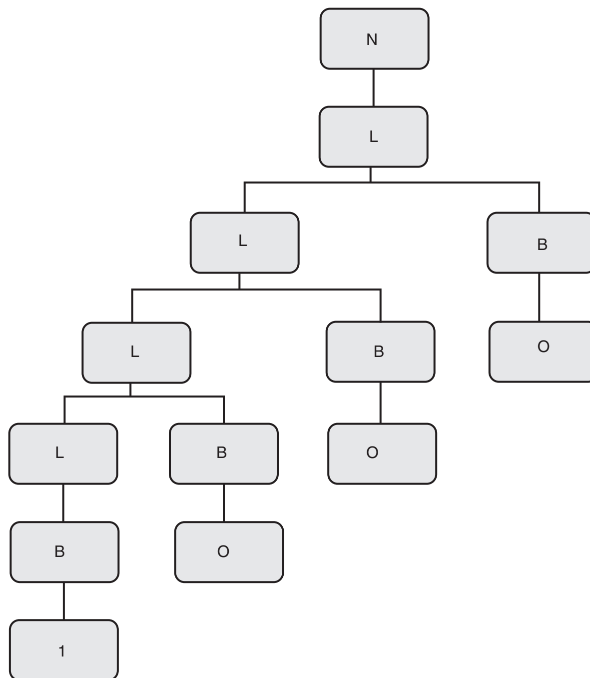


Figure 6.15 Parse Tree for "1000"

$B \rightarrow 0$ { B•count = 1;}
$B \rightarrow 1$ { B•count = 1;}

The next reduction is B to L; when this is reduced, the number of digits is the number of digits at the child node. Hence, define semantic actions as follows:

$L \rightarrow B$ { L•count = B•count;}

The next reduction is LB to L; when this is reduced, the count is the number of digits at the left child plus the count at the right child. Hence, define semantic actions as follows:

$L \rightarrow LB$ { L•count = L•count + B•count;}

Final SDT for counting the number of digits in a binary number is as follows:

$N \rightarrow L$ { N•count = L•count;}
$L \rightarrow LB$ { L•count = L•count + B•count;}
$L \rightarrow B$ { L•count = B•count;}
$B \rightarrow 0$ { B•count = 0;}
$B \rightarrow 1$ { B•count = 1;}

Example 7:

Write an SDT to convert binary to decimal. For example, the binary number 101.101 denotes the decimal number 5.625.

Solution:

First define the grammar for a binary number. The binary digit "B" can be either "1" or "0." The number can be a list of binary digits. The list can recursively define bits "B." A binary number can be with decimal or without decimal. So the grammar is as follows:

$N \rightarrow L_1 \bullet L_2$
$N \rightarrow L$
$L \rightarrow LB$
$L \rightarrow B$
$B \rightarrow 0$
$B \rightarrow 1$

The next step is to take a small string and draw a parse tree. First, we consider a binary without decimal. So take 101.101 and consider only the left sub tree for 101. Draw a parse tree as shown in Figure 6.16.

The third step in SDT is to define semantic actions. Assume "dval" as an attribute to store the decimal equivalent of binary.

For example, the first reduction is the reducing terminal "1" to B. So here "dval" is one. If it is "0" decimal equivalent "dval" is 0. Hence, define semantic actions as follows:

$B \rightarrow 0$ { B•dval = 0;}
$B \rightarrow 1$ { B•dval = 1;}

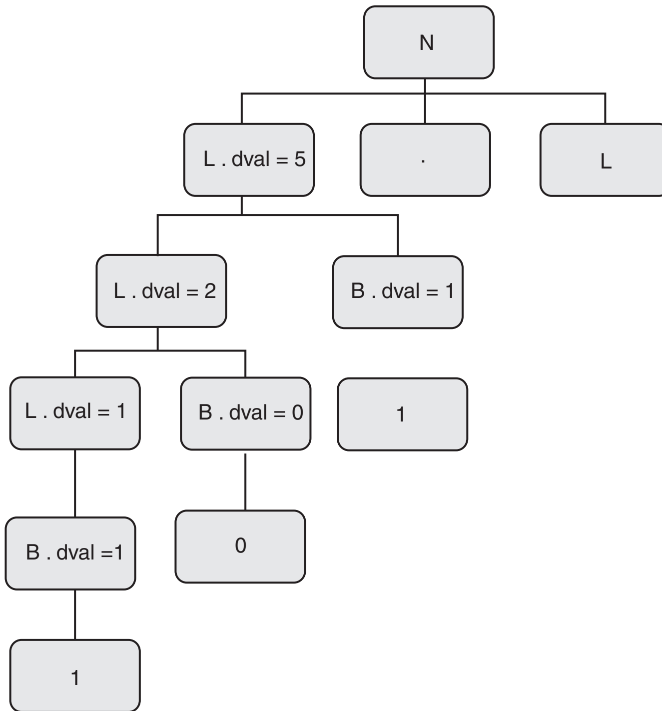


Figure 6.16 Parse Tree for “4 - 2 - 4 * 2”

The next reduction is B to L; when this is reduced, decimal equivalent “•dval” is whatever value the child node has. Hence, define semantic actions as follows:

$L \rightarrow B$ { L•dval = B•dval;}

The next reduction is LB to L; when this is reduced, the decimal equivalent “dval” is “dval” at the left child *2 plus “dval” at the right child. Hence, define semantic actions as follows:

$L \rightarrow LB$ { L•dval = L•dval * 2 + B•dval;}

The final SDT for converting a binary number without decimal to decimal equivalent is as follows:

$N \rightarrow L_1 \bullet L_2$	{}
$N \rightarrow L$	{ N•dval = L•dval;}
$L \rightarrow LB$	{ L•dval = L•dval * 2 + B•dval;}
$L \rightarrow B$	{ L•dval = B•dval;}
$B \rightarrow 0$	{ B•dval = 0;}
$B \rightarrow 1$	{ B•dval = 1;}

The annotated parse tree for string 101 is as shown in Figure 6.17.

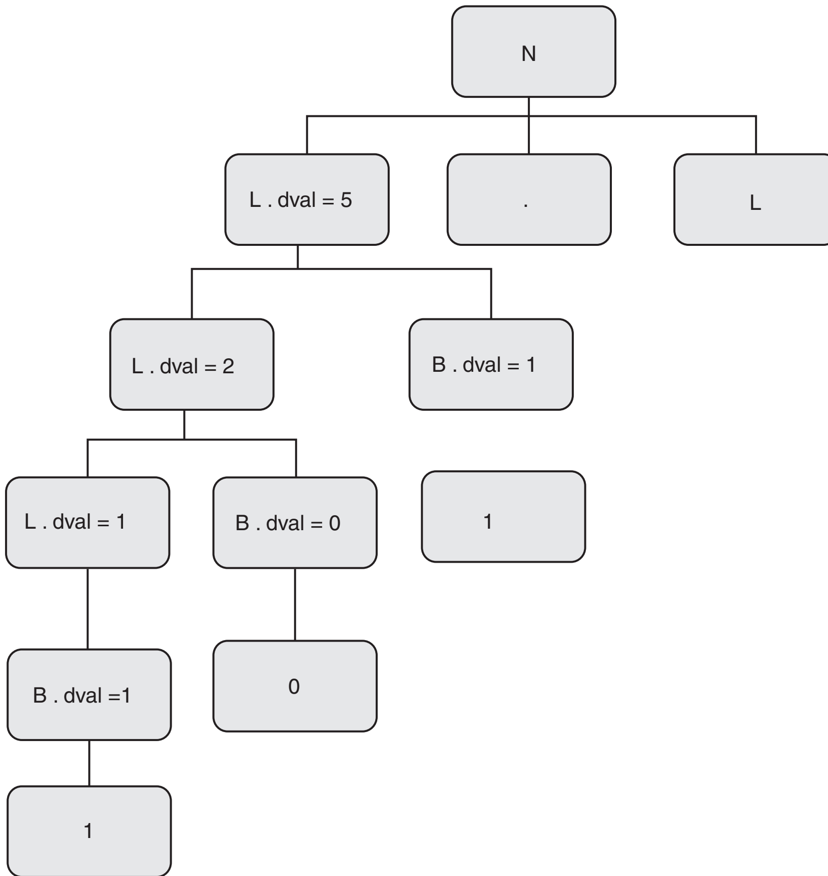


Figure 6.17 Annotated Parse Tree for "101"

Now let us extend the SDT for the decimal part. As we have already defined rules for the nonterminal L, given an input string 101•101, the defined semantic actions are carried out even for the right sub tree. This results in an equivalent for the decimal part, that is, 101 as 5. Now as this is already evaluated, we can get the decimal equivalent for the decimal part as follows:

$L.dval/2^{L.nd}$ where "dval" is the decimal equivalent of the number after the decimal and "nd" is the number of digits after the decimal. For example, 101, dval = 5 and nd = 3.

So equivalent = $5/2^3 = 5/8 = 0.625$.

Let us now extend SDT for evaluating the number of digits "nd." This is similar to the previous example of counting the number of digits in binary. So the final SDT is shown below:

$N \rightarrow L_1 \bullet L_2$	{ $N \bullet dval = L_1 \bullet dval + L_2 \bullet dval / 2^{L2 \bullet nd}$;}
$N \rightarrow L$	{ N•dval = L•dval;}
$L \rightarrow LB$	{ L•dval = L•dval * 2 + B•dval; L•nd = L•nd + B•nd;}

$L \to B$ { L•dval = B•dval; L•nd = B•nd;}
$B \to 0$ { B•dval = 0; B•nd = 1;}
$B \to 1$ { B•dval = 1; B•nd = 1;}

6.5 Creation of the Syntax Tree

The parser produces a tree while checking the syntax of the programming language construct. This is called the parse tree. A parse tree gives the complete syntax of a string. It tells you what the syntax behind the string is. Hence it is called ***concrete syntax tree,*** whereas a condensed form of a parse tree is called ***syntax tree,*** which abstracts away unnecessary terminals and nonterminals. Hence, it is even called ***abstract syntax tree(AST).***

For example, consider the string 3 * (4 + 2). The parse tree and syntax tree for the string are shown in Figure 6.18.

Now let us see how to write an SDT for creating a syntax tree.

Example 8:

SDT for creating a syntax tree

Solution:

Assume that an input string is of the form "a + b * c."

The first step is to define grammar. Take unambiguous grammar that defines all arithmetic expressions with "+" and "*."

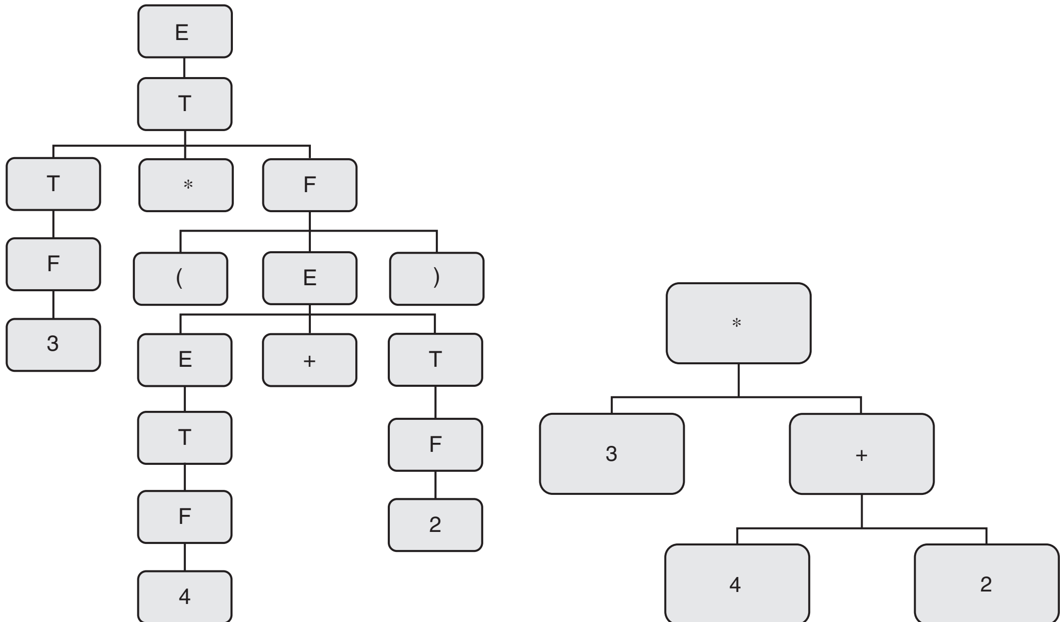


Figure 6.18 Parse Tree/Concrete Syntax Tree Syntax Tree/Abstract Syntax Tree

E→E+T | T
T→T*F | F
F→id

The second step is to take the input string and draw a parse tree. It is shown in Figure 6.19.

The third step is defining semantic actions. Here we are supposed to construct a node in the parse tree at any time. For creating a node, assume that there is a procedure mknode(l, d, r), where l is the left pointer, d is the data element, and r is the right pointer. Assume that when mknode() is called with three arguments, creates a node, and returns the pointer to the newly created node. Now let us see how the tree is created by using mknode().

The first reduction is identifier "a" is reduced to F. Here we need to create a leaf node. For creating leaf nodes we use the mknode() function with the left pointer and the right pointer as NULL. The return type of function is the pointer to the newly created node. So to store this node pointer, assume that there is an attribute "nptr." The semantic action is as follows:

F → id { F•nptr = mknode (NULL, id•lvalue, NULL);}

The next reduction is F to T. Here simply extend the pointer further to T by adding semantic action as

T → F { T•nptr = F•nptr;}.

Similarly, even for the next reduction T to E, extend the pointer further.

The next reduction is "T*F" to T. Here create a node using the mknode() function with "*" as data element and T•nptr as the left pointer and F•nptr as the right pointer as shown below:

T → T * F { T•nptr = mknode (T•nptr, "*", F•nptr);}

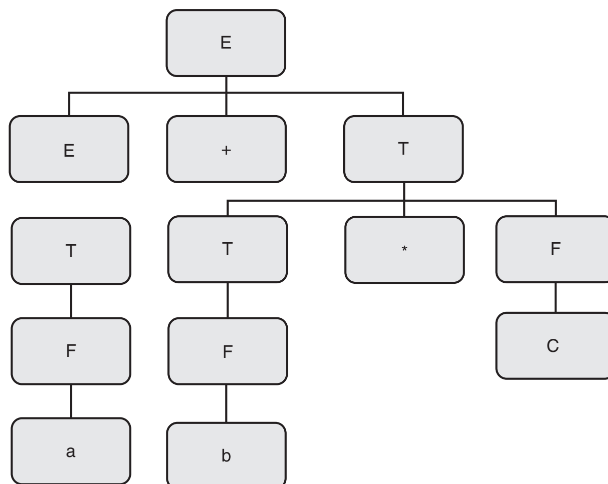


Figure 6.19 Parse Tree for "a + b * c"

The same thing can be carried out for E+T to E as

$$E \rightarrow E + T \qquad \{ E\bullet nptr = mknode\ (E\bullet nptr, \text{“+”}, T\bullet nptr);\}$$

So the final SDT for creating syntax tree is as follows:

$$\begin{array}{ll} E \rightarrow E + T & \{ E\bullet nptr = mknode\ (E\bullet nptr, \text{“+”}, T\bullet nptr);\} \\ E \rightarrow T & \{ E\bullet nptr = T\bullet nptr;\} \\ T \rightarrow T * F & \{ T\bullet nptr = mknode\ (T\bullet nptr, \text{“*”}, F\bullet nptr);\} \\ T \rightarrow F & \{ T\bullet nptr = F\bullet nptr;\} \\ F \rightarrow id & \{ F\bullet nptr = mknode\ (NULL, id\bullet lvalue, NULL);\} \end{array}$$

where mknode (l, d, r) is a function that creates a node with “l” as the left pointer and “d” as the data element, and “r” as the right pointer. It returns pointer to a newly created node.

The resulting syntax tree for input string “a + b * c” is as follows shown in Figures 6.19 and 6.20.

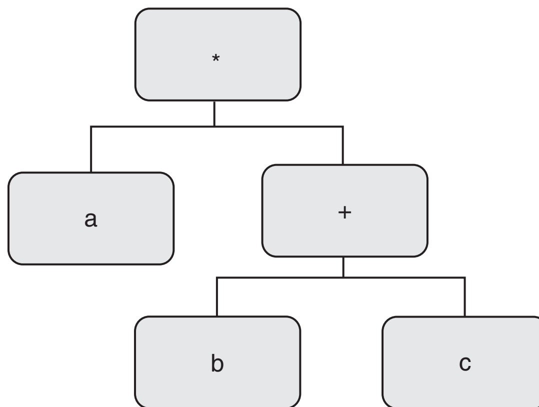


Figure 6.20 Syntax Tree for “a + b * c”

6.6 Directed Acyclic Graph (DAG)

DAG is used to identify common sub expressions. It helps us in eliminating redundant code.

For example, let there be an expression a + (b * c) + (b * c). This expression can be represented as tree or DAG as shown in Figure 6.21.

So here, for a common sub expression, a node is created twice in a tree, whereas in DAG, it is created only once and reused later. Now let us see how to write SDT for creating DAG, that is, for the above SDT, given an input string a+a*a, if we carry out the above translations, we get the syntax tree as found in Figure 6.22.

But we want DAG as shown in Figure 6.23.

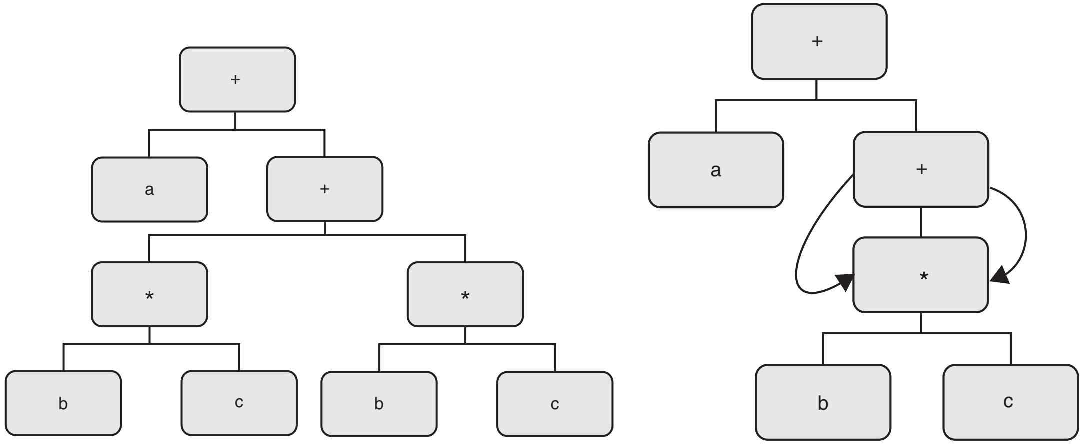


Figure 6.21(a) Syntax Tree (b) DAG

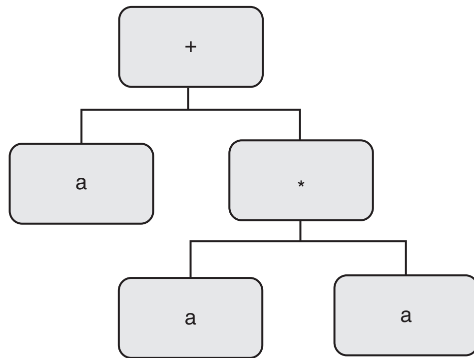


Figure 6.22 Syntax Tree for "a + a * a"

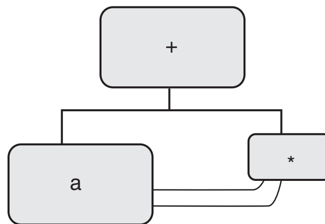


Figure 6.23 DAG for "a + a * a"

So to get a DAG, do we have to rewrite the SDT in Example 8 or do we have to modify the SDT? What changes are required, if any? No single change is required in SDT—Example 8. As it is, the same SDT can be used even for creating DAG but the difference is in function mknode(). Earlier, we have defined mknode() as function that when called blindly creates a node and returns a node. Now the modification required is in mknode(). Here, it should maintain a list of pointers already created.

Whenever mknode() is called, it first verifies whether the node is already created or not. If it is already created, return the same pointer. If it is not there in the already created node list, then it creates one and adds to list.

Example 9:

SDT for creating DAG

Solution:

E → E + T	{ E•nptr = mknode (E•nptr, "+", T•nptr);}
E → T	{ E•nptr = T•nptr;}
T → T * F	{ T•nptr = mknode (T•nptr, "*", F•nptr);}
T → F	{ T•nptr = F•nptr;}
F → id	{ F•nptr = mknode (NULL, id•lvalue, NULL);}

where mknode (l,d,r) function maintains a list of pointers already created. Whenever it is called, first it verifies whether the node is already created or not. If it is already created, return the same pointer. If it is not there in already created node list, then creates one and adds to list.

6.7 Types of SDTs

There is one more type called the L-attribute definition. The difference between S-attributed and L-attributed is as follows:

S-attributed definition	L-attributed definition
1. It uses only synthesized attributes.	1. It allows both types. But if an inherited attribute is present, there is a restriction. The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.
2. Semantic actions can be placed only at the end of the right hand side of a production.	2. Semantic actions can be placed anywhere on the right hand side.
3. Attributes are generally evaluated during bottom-up parsing.	3. Attributes are generally evaluated by traversing the parse tree depth first and left to right.

6.8 S-Attributed Definition

So far whatever the examples we have used is S-attributed definition only. Look at the examples; in all the examples, the type of attribute used, that is, "val" or "type" or "red" or "count" or "nptr" is only synthesized. A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition, where "S" stands for simple. S-attributed definition is also called *postfix SDT* as semantic actions are always placed only at the right end of productions.

6.9 Top-Down Evaluation of S-Attributed Grammar

So far we have seen bottom-up evaluation of S-attributed grammar, that is, given an SR/LR parser. If the translations are carried out, what would be the output? Suppose we have a top-down parser with SDT; can't we carry out the same translations? Yes. We can carry out even by using the top-down parser. The difference is if it is bottom-up parser, we can easily specify when exactly the translation is carried out. For example, so far the assumption is that whenever the parser reduces by any production, the corresponding semantic action is carried out automatically, whereas with top-down parser, no doubt translations can be carried out. But we cannot tell when exactly the translations are carried out. So far we have seen enough examples for bottom-up evaluation. Now let us see some examples for top-down evaluation.

Given an SDT, a top-down parser carries out translations as follows:

1. Assumes semantic actions as dummy nonterminals, which become part of the right hand side of production.
2. Translations are pushed onto the stack along with grammar symbols.
3. When a dummy nonterminal appears on top of the stack, the terminal is popped and corresponding action is carried out.

To understand the above procedure, let us take a simple SDT

Example 10:

Write an SDT for counting the number of balanced parenthesis. For example, given ((())) should give 3 as output.

Solution: The grammar for balanced parentheses is as follows:

$$\mathbf{S \rightarrow (S)} \\ \mathbf{\mid \varepsilon}$$

The second step is to draw the parse tree for an input string "((()))." It is shown in Figures 6.24 and 6.25.

The last step is to define the rules.

$$S \rightarrow (S_1) \quad \{ S\bullet cnt=S_1\bullet cnt+1;\}$$
$$\mid \varepsilon \quad \{ S\bullet cnt=0;\}$$

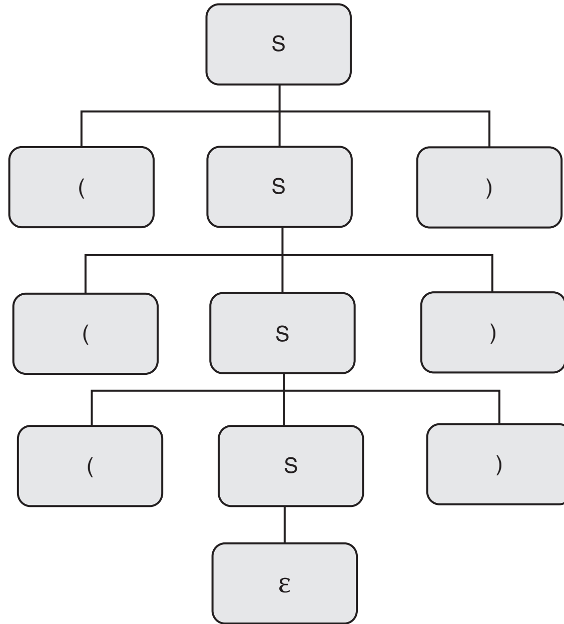


Figure 6.24 Parse Tree for Balanced Parentheses

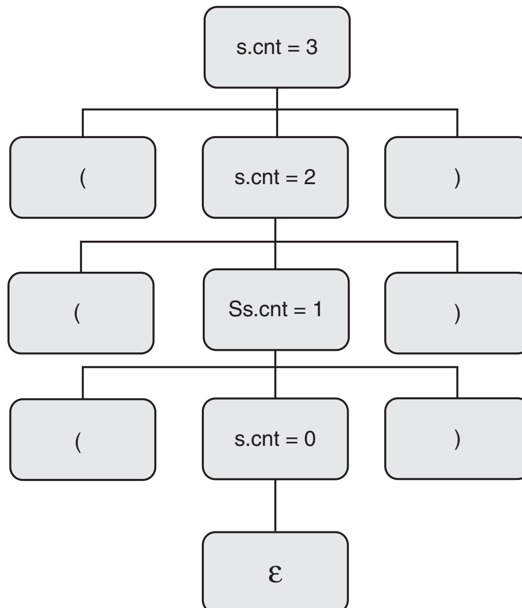


Figure 6.25 Annotated Parse Tree for "((())."

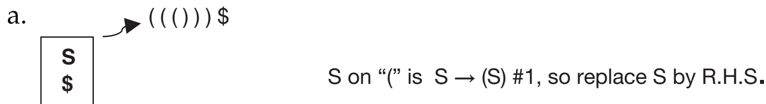
The annotated parse tree is as follows:

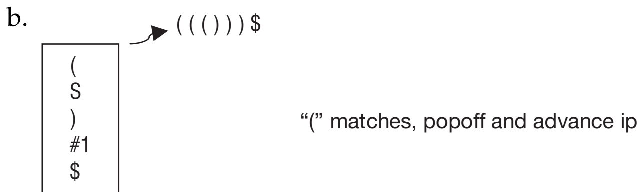
Top-down evaluation is as follows:

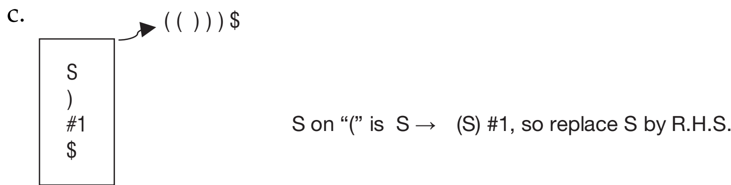
1. Assume semantic actions as dummy nonterminals.

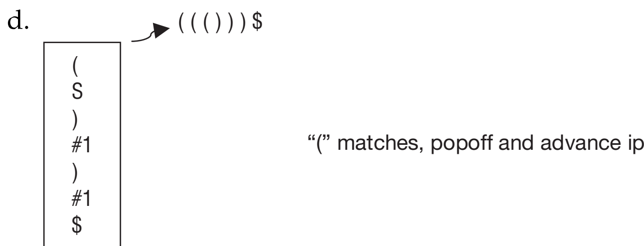
$\mathbf{S \rightarrow (S_1)}$	**#1**
$\mathbf{\mid \varepsilon}$	**#2**
where #1	is $S \bullet cnt = S_1 \bullet cnt + 1$;
and #2	is $S \bullet cnt = 0$;

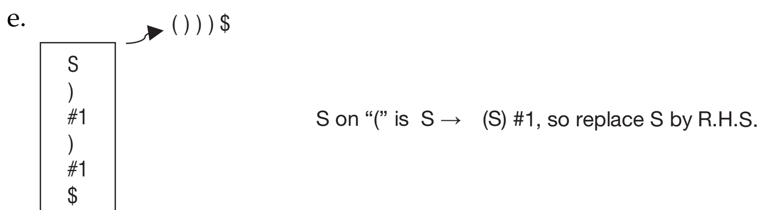
2. Use LL(1) parsing algorithm to parse the input string.
3. During parsing if #1 or #2 appears on top of the stack, then take out the symbol and carry out the translations as shown in Figure 6.26.

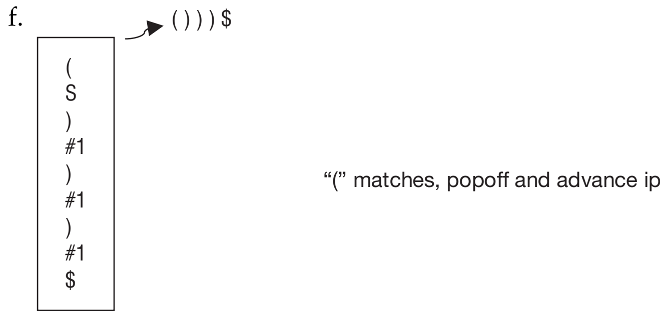


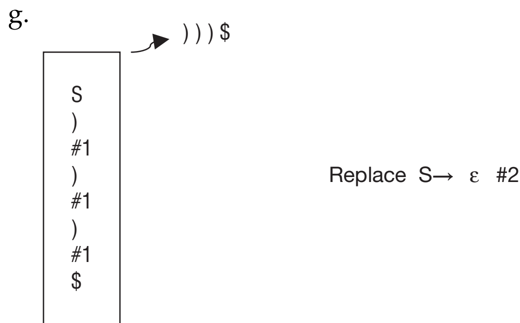


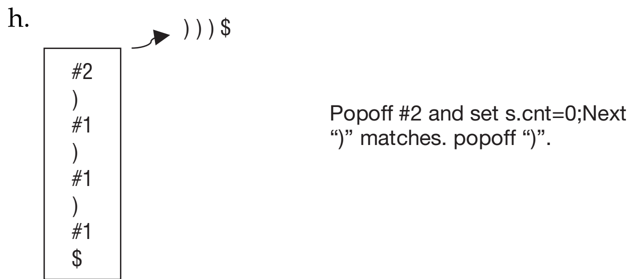












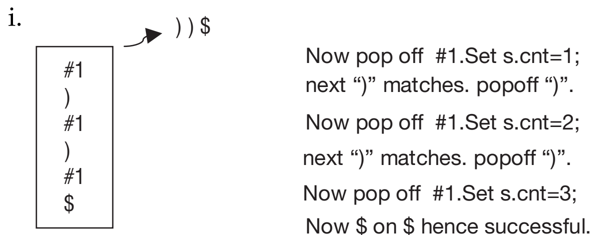


Figure 6.26 Parser Actions in Top-Down Evaluation

This is how top-down evaluation of S-attributed definition can be carried out. Even observe that given an SDT whether it is top-down evaluation or bottom-up evaluation, the result is the same for the given input string.

6.10 L-Attributed Definition

1. It allows both types, that is, synthesized as well as inherited. But if at all an inherited attribute is present, there is a restriction. The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.

   For example, consider the rule

   A → XYZPQ assume that there is an inherited attribute. "i" is present with each nonterminal. Then,

   Z•I = f(A•i | X•i | Y•i) but Z•i = f(P•i | Q•i) is wrong as they are right siblings.

2. Semantic actions can be placed anywhere on the right hand side.

   Ex: A → a {}
   B → C {} D
   C → EF {}

   They can be placed either in the beginning or in the middle of grammar symbols or at the end.

3. Attributes are generally evaluated by traversing the parse tree depth first and left to right. It is possible to rewrite any L-attributes to S-attributed definition.

Example 11:

Consider the following SDT:

```
A → LM      { L•i = f(A•i);
              M•i = f(L•s)
              A•s = f(M•s);}
A → QR      { R•i = f(A•i);
              Q•i = f(R•s)
              A•s = f(Q•s);}
```

Is it L-Attributed or S-Attributed?

Solution: Look at semantic actions. Here by looking at semantic actions we understand that "s" is synthesized and "i" is inherited. So actions in first rule are correct since "i" is dependent on parent or left siblings only. But look at the second action in A → QR, Q.i = f(R.s); this violates the basic property of L-attributed definition. Inherited attribute can inherit either from parent or from left sibling only. So this is a wrong SDT—not S-attributed or not L-attributed.

So far we have discussed enough examples for S-attributed definition. Now let us look at some examples for L-attributed definition.

Let us start with a simple example .Consider the S-attributed definition for converting infix to the post fix form.

```
E → E1 + T        { print("+");}
E → T             {}
T → T1 * F        { print("*");}
T → F             {}
F → id            { print("id.name");}
```

Consider actions as dummy nonterminals #1,#2,#3.

$$
\begin{array}{ll}
E \rightarrow E_1 + T & \#1 \\
E \rightarrow T & \\
T \rightarrow T_1 * F & \#2 \\
T \rightarrow F & \\
F \rightarrow id & \#3
\end{array}
$$

Eliminate left recursion. Now we get simple L-attributed definition as follows:

Example 12:

L-attributed definition for converting infix to post fix form.

$$
\begin{array}{ll}
E \rightarrow TE'' & \\
E'' \rightarrow +T \ \#1 \ E'' \mid \varepsilon & \\
T \rightarrow F \ T'' & \\
T'' \rightarrow \ * F \ \#2 \ T'' \mid \varepsilon & \\
F \rightarrow id & \#3
\end{array}
$$

where #1 corresponds to printing "+" operator, #2 corresponds to printing "*," and # 3 corresponds to printing id.val.

Look at the above SDT; there are no attributes, it is L-attributed definition as the semantic actions are in between grammar symbols. This is a simple example of L-attributed definition. Let us analyze this L-attributed definition and understand how to evaluate attributes with depth first left to right traversal. Take the parse tree for the input string "a + b*c" and perform *Depth first left to right traversal, i.e. at each node traverse the left sub tree depth wise completely then right sub tree completely.*

Follow the traversal in Figure 6.27. During the traversal whenever any dummy nonterminal is seen, carry out the translation.

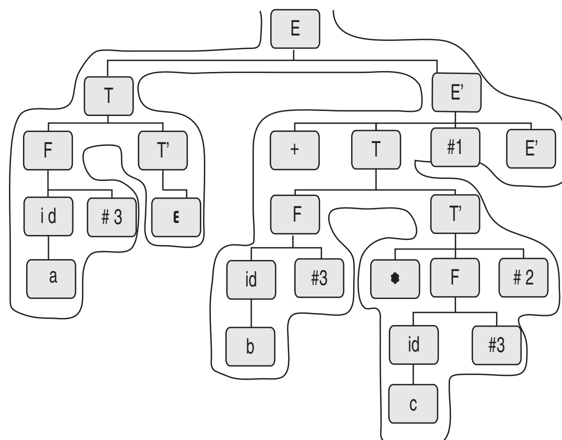


Figure 6.27 Depth First Traversal of Tree

- So if we follow the path, the first dummy nonterminal that is seen is #3 after seeing identifier "a." So carryout #3, which results in printing out "a."
- Next dummy nonterminal seen is #3 after seeing identifier "b." So carryout #3, which results in printing out "b."
- Next dummy nonterminal seen is seen is #3 after seeing identifier "c." So carryout #3, which results in printing out "c."
- Next dummy nonterminal seen is #2, which results in printing out "*."
- The last dummy nonterminal seen is #1, which results in printing out "+."
- So the final result is "abc*+."

Let us take another example of L-attributed with attributes.

Example 13:

Write an SDT for storing type information in a symbol table.

Solution:

Given an input string int a,b,c, the SDT should give an output as

A	int
B	int
C	int

The first step is to define grammar for declaration statements of the form int a,b,c

$D \rightarrow T\ L$
$T \rightarrow$ int
$T \rightarrow$ char
$L \rightarrow L$, id
$L \rightarrow$ id

The second step is to take the input string and draw a parse tree as shown in Figure 6.28.

The third step is to define semantic actions. Here, the parser first reads "int" and reduces to T. So to store the type information, assume an attribute "type" with a grammar symbol and store the type by defining semantic actions as follows:

$T \rightarrow$ int { T•type = int}
$T \rightarrow$ char { T•type = char}

The resulting parse tree is as shown in Figure 6.29.

Now look at the parse tree; type information is available at node T, whereas node L is deriving identifiers a, b, c. So if the type information at node T is made available with node L then whenever an identifier is seen, it stores type together with identifier. Now the question is how to propagate type information from node T to node L. Can we write semantic action as follows?

$D \rightarrow TL$ {L•type = T•type;}

Is this correct? No. "type" is a synthesized attribute; so using synthesized attribute we cannot propagate information to sibling. So the only way here is to use an inherited attribute.

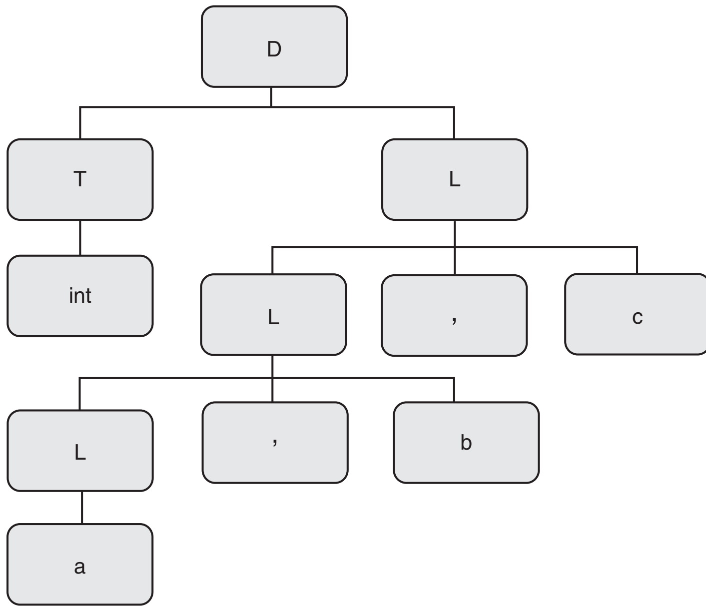


Figure 6.28 Parse Tree for "int a, b, c"

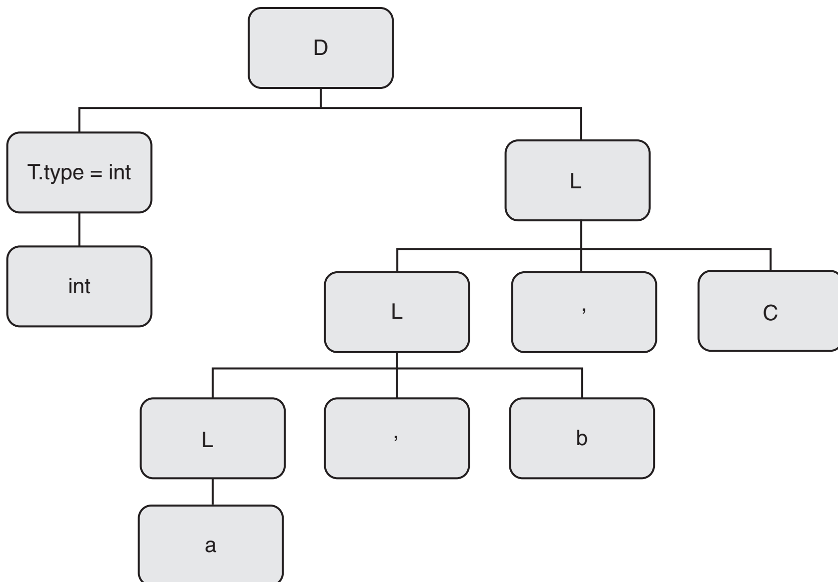


Figure 6.29 Propagating Type Info in Parse Tree

Assume that there is an inherited attribute "in." Now we can define the rule as follows:

$D \rightarrow TL$ {L•in= T•type;}

As "in" is an inherited attribute, it can take type information from node T.

Now node L takes type information from T. Now child L takes type info from parent using inherited attribute by having action as

$L \rightarrow L_1, id$ { L_1•in = L•in;}.

Now type is available with L and id is also available. Assume add_type(type,id) as a function that stores id together with type into symbol table. So add the action as follows:

$L \rightarrow L_1, id$ { L_1•in = L•in; add_type(L_1•in, id);}

Finally, when id is reduced to L, we have type information together with id, so store it in symbol table by defining rule as

$L \rightarrow id$ { add_type(L•in, id•name);}

So the final SDT is as follows:

$D \rightarrow TL$	{L•type = T•type;}
$T \rightarrow int$	{ T•type = int}
$T \rightarrow char$	{ T•type = char}
$L \rightarrow L_1, id$	{ L_1•in = L•in; add_type(L_1•in, id);}
$L \rightarrow id$	{ add_type(L•in, id•name);}

The annotated parse tree is as shown in Figure 6.30.

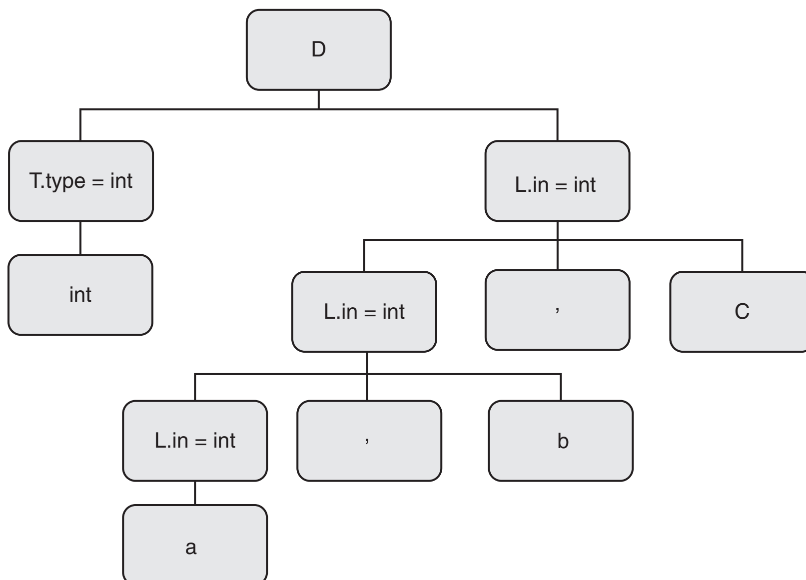


Figure 6.30 Depth First Traversal of Tree

This is an L-attributed definition that uses both synthesized and inherited attributes.

General evaluation of semantic actions in any SDT is as follows:

1. Take input string, parse it, and it gives parse tree.
2. Traverse the parse tree and draw the dependency graph.
3. Graph gives evaluation order.

Let us understand the procedure with the above example.

Take the parse tree and dependency graph. *Dependency graph is a graph that shows inter dependencies among the attributes*. The dependency graph is shown below in Figure 6.31.

We follow the dependency graph that gives you the evaluation order.

The evaluation order is as follows:

a1 = int
a2 = a1
a3 = a2
add_type(int,a)
a4 = a3
add_type(int,b)
add_type(int,c)

The general procedure is complex. Hence, we have discussed specific procedures. The simple procedures are if it is S-attributed, evaluate during bottom-down parsing. If it is L-attributed, traverse the parse tree depth first left to right.

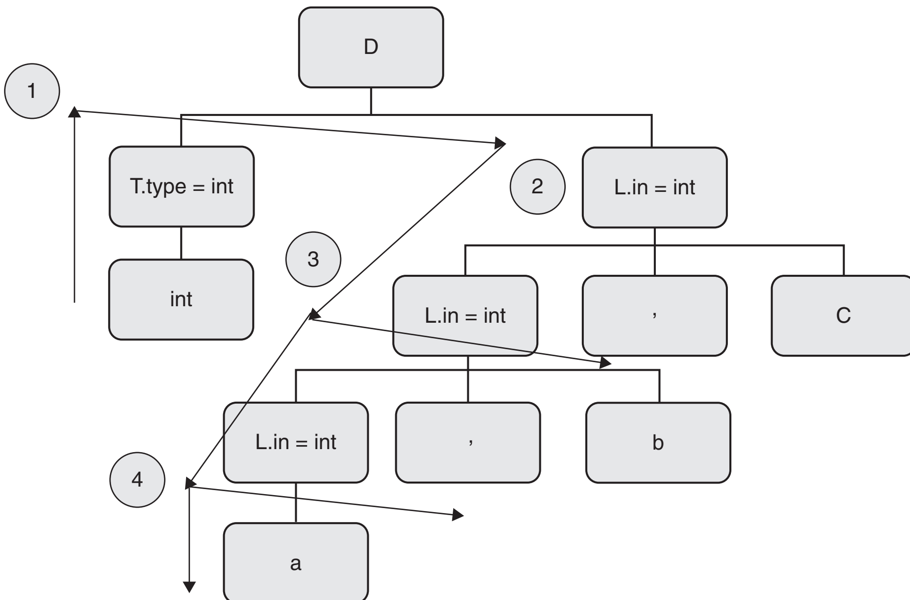


Figure 6.31 Dependency Graph

Let us see how to evaluate attributes by traversing parse tree depth first left to right. Depth first left to right traversal is at each node traverse the left sub tree depth wise then right sub tree completely. During this traversal evaluate attributes as follows:

- Evaluate inherited attributes if the node is visited for the first time
- Evaluate synthesized attributes if the node is visited for the last time

Look at Figure 6.32. Start at point 1; here node D has no attributes defined. So continue till 2. Here node T is visited for the first time; so evaluate inherited attributes. Node T is not defined with any inherited attributes, so continue till point 3. At 3, it is the last visit for node T, so evaluate synthesized attributes T → int {T.type = int}. So node T will have type information. Now continue till point 4; at 4, the first visit for node L, evaluate inherited attribute {L.in=T.type;}. Node L, at point 5 is the first visit, so evaluate inherited.

L → L_1, id { L_1.in = L.in; add_type(L_1.in, id);} Here add_type() has two arguments—L.in, which is inherited attribute and 'id' is synthesized attribute. As function has both types of attributes, it can be evaluated only at the last visit. Similarly, at point 6, the inherited attribute is evaluated. Now at point 7, the last visit, so add_type() is carried out with the result, "a" with type "int" is stored in symbol table. Now at point 8, the last visit, add_type() is carried out with the result, "b" with type "int" is stored in symbol table. Now at point 9, the last visit, add_type() is carried out with the result, "c" with type "int" is stored in the symbol

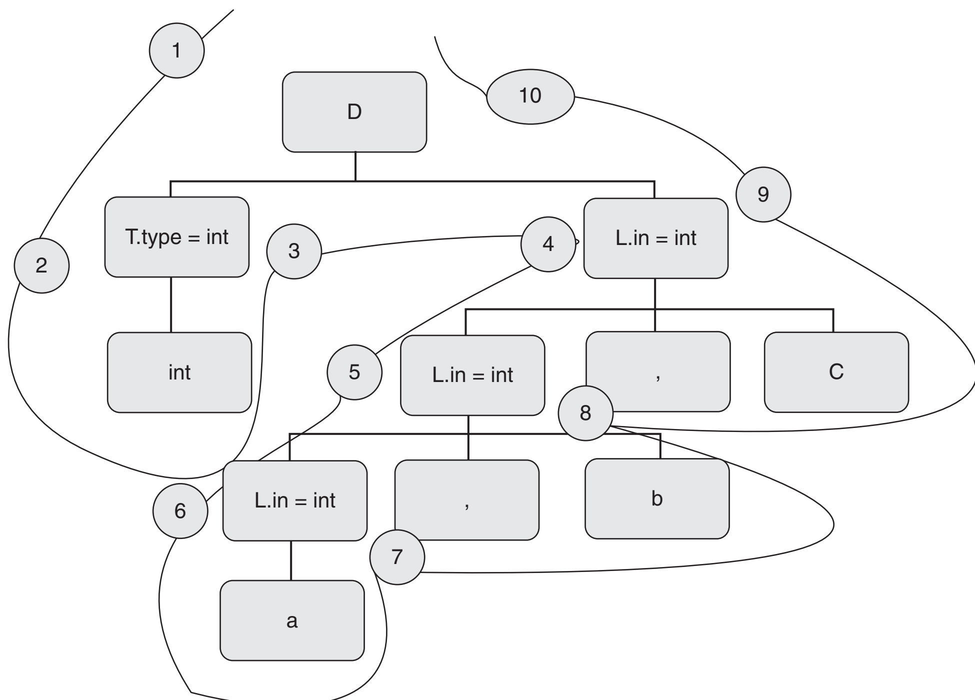


Figure 6.32 Depth First Left to Right Traversal of Tree

table. At point 10, it completes traversal. By this time a,b,c, along with type is stored in the symbol table.

6.11 Converting L-Attributed to S-Attributed Definition

Now that we understand that S-attributed is simple compared to L-attributed definition, let us see how to convert an L-attributed to an equivalent S-attributed.

Consider an L-attributed with semantic actions in between the grammar symbols. Suppose we have an L-attributed as follows:

S → A {} B

How to convert it to an equivalent S-attributed definition? It is very simple!!

Replace actions by nonterminal as follows:

S → A M B
M → ε {}

Example 14:

Convert the following L-attributed definition to equivalent S-attributed definition.

E → TE″
E″ → +T #1 E″ | ε
T → F T″
T″ → *F #2 T″ |ε
F → id #3

Solution:

Replace dummy nonterminals that is, actions by nonterminals.

```
E → TE″
E″ → +T A E″ | ε
A →                     { print("+");}
T → F T″
T″ →  *F B T″ |ε
B →                     { print("*");}
F → id                  { print("id");}
```

Example 15:

Consider Example 13; let us see how to convert L-attributed to equivalent S-attributed definition.

Solution:

To get the equivalent S-attributed definition, we need to eliminate inherited attributes in Example 13. How to avoid inherited attributes in Example 13? Recollect Example 13, there as type information is to be taken from node L which is a sibling, the only way is using

inherited attribute. So as long as we use that grammar, L will be a sibling to T. As long as L is sibling to T, inherited attribute is needed. So to avoid inherited attribute the only way is to rewrite the grammar as follows:

$D \rightarrow D_1$, id
$D \rightarrow T$, id
$T \rightarrow$ int | char

The parse tree for the string int a,b,c is as shown in Figure 6.33.

Now define the semantic actions as follows:

First read "int" then reduce "int" to T. So to store type, assume attribute type and define rule as

$T \rightarrow$ int { T•type = int}

Next it reads "a," then reduces 'T id' to D. So here we propagate type information to parent by using the same synthesized attribute and also store type with id into symbol table by using add_type() by defining semantic action as follows:

$D \rightarrow T$, id {L•type = T•type; add_type(T•type, id•name)}

Next it reads "b," then reduces 'D, id' to D. So here we propagate type information to parent by using the same synthesized attribute and also store type with id into symbol table by using add_type() by defining semantic action as follows:

$D \rightarrow D_1$, id {D•type = D_1•type; add_type(D_1•type, id•name);}

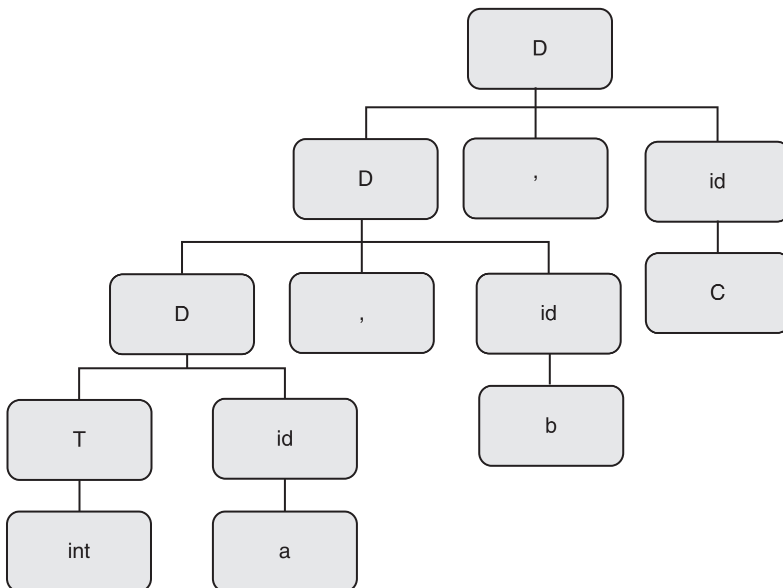


Figure 6.33 Parse Tree for "int a, b, c"

So the final equivalent L-attributed definition is as follows:

D → D_1, id	{D•type = D_1•type; add_type(D_1•type, id•name);}
D → T, id	{L•type = T•type; add_type(T•type, id•name)}
T → int	{ T•type = int}
T → char	{ T•type = char}

So to convert L-attributed to equivalent S-attributed here, grammar is changed. But rewriting grammars is not a simple solution. Consider the following example SDT:

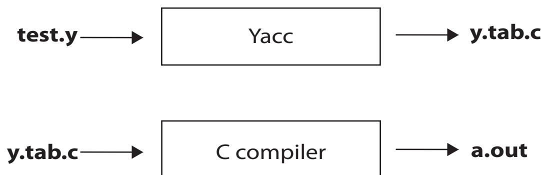
A → BC {A•i=B•i + C•i;} is S-attributed and also L-attributed since there are no inherited attributes. So every S-attributed is even L-attributed. However, the reverse is not true.

6.12 YACC

YACC—Yet Another Compiler Compiler—is a tool for construction of automatic LALR parser generator.

Using Yacc

Yacc specifications are prepared in a file with extension ".y" For example, "test.y." Then run this file with the Yacc command as "$yacc test.y." This translates yacc specifications into C-specifications under the default file mane "y.tab.c," where all the translations are under a function name called yyparse(); Now compile "y.tab.c" with C-compiler and test the program. The steps to be performed are given below:



Commands to execute

$yacc test.y

This gives an output "y.tab.c," which is a parser in c under a function name yyparse().

With –v option ($yacc –v test.y), produces file y.output, which gives complete information about the LALR parser like DFA states, conflicts, number of terminals used, etc.

```
$cc y.tab.c
$./a.out
```

Preparing the Yacc specification file

Every yacc specification file consists of three sections: the *declarations*, *grammar rules*, and *supporting subroutines*. The sections are separated by double percent "%%" marks.

```
  declarations
%%
  Translation rules
%%
  supporting subroutines
```

The declaration section is optional. In case if there are no supporting subroutines, then the second %% can also be skipped; thus, the smallest legal Yacc specification is

```
%%
  Translation rules
```

Declarations section

Declaration part contains two types of declarations—Yacc declarations or C-declarations. To distinguish between the two, C-declarations are enclosed within %{ and %}. Here we can have C-declarations like global variable declarations (int x=1;), header files (#include….), and macro definitions(#define…). This may be used for defining subroutines in the last section or action part in grammar rules.

Yacc declarations are nothing but tokens or terminals. We can define tokens by %token in the declaration part. For example, "num" is a terminal in grammar, then we define

% token num in the declaration part. In grammar rules, symbols within single quotes are also taken as terminals.

We can define the precedence and associativity of the tokens in the declarations section. This is done using %left, %right, followed by a list of tokens. The tokens defined on the same line will have the same precedence and associativity; the lines are listed in the order of increasing precedence. Thus,

```
%left '+' '–'
%left '*' '/'
```

are used to define the associativity and precedence of the four basic arithmetic operators '+','–','/','*'. Operators '*' and '/' have higher precedence than '+' and '–' and both are left associative. The keyword %left is used to define left associativity and %right is used to define right associativity.

Translation rules section

This section is the heart of the yacc specification file. Here we write grammar. With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values when a token is matched. An action is defined with a set of C statements. Action can do input and output, call subprograms, and alter external vectors and variables. The action normally sets the pseudo variable "$$" to some value to return a value. For example, an action that does nothing but return the value 1 is { $$ = 1;}

To obtain the values returned by previous actions, we use the pseudo-variables $1, $2, . . ., which refer to the values returned by the grammar symbol on the right side of a rule, reading from left to right. Thus, if the rule is

A: B C D

for example, then $1 has the value returned by B, and $2 the value returned by C. As an example, consider the rule

E: '(' E ')';

The value returned by this rule is usually the value of the E in parentheses. This can be indicated by

E : '(' E ')' { $$ = $2;}

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with lexeme values, if desired) to the parser. This can be done in two ways—either the lex tool can be used or a user can write his own hand-coded lexical analyzer in C. The lexical analyzer is an integer valued function with a default name "yylex()." The yylex() function returns an integer, which is the token number, representing the kind of token that is read. If there is a value associated with that token, it should be assigned to the yacc-defined global variable "yylval." The parser and the lexer must agree upon these token numbers so that communication can take place between them. The numbers may be chosen by Yacc or by the user. In either case, These numbers are symbolically returned by the lexical analyzer by using the "# define" macro mechanisms of C.

For example, suppose that the token name NUM has been defined in the declarations section of the Yacc specification file.

```
yylex(){
extern int yylval;
int c;
. . .
c = getchar();
. . .
switch(c) {
. . .
        case '0':
        case '1':
. . .
        case '9':
        yylval = c-'0';
        return( NUM );
. . .
}
. . .
```

The above code is to return a token NUM along with its value on seeing a digit. The value of token is equal to the numerical value.

If the lexical analyzer is supplied with the lex tool, first prepare the lex file like test.l. Run under lex. That gives you the "lex.yy.c" file. Add this as the first statement in supporting routines part as follows:

```
%%

%%
#include "lex.yy.c"
---------
```

When Yacc is invoked with the –v option (verbose), a file called y.output is created with complete description of the parser. The y.output file corresponds to the above grammar with a full description of the LALR parser.

Yacc invokes two disambiguating rules by default:

1. In case of a shift/reduce conflict, the default action is to go with the shift action.
2. In case of a reduce/reduce conflict, the default action is to reduce action by the earlier grammar rule.

Supporting routines section

When a user prepares input a specification file to Yacc with "y" extension , it creates an output file called y.tab.c. An integer valued function "yyparse()" is produced by Yacc; when yyparse() is called, in turn it calls yylex(), which is the lexical analyzer supplied by the user to obtain input tokens. yyparse() returns the value 1 on error and returns 0 on EOF. The user must provide some supporting routines for this parser. For example, as with every C program, a program called "main" must be defined, that eventually calls parser function yyparse(). In addition to this, a routine called yyerror()is also required as it gives error diagnostics when a syntax error is detected. These two routines are compulsory for any yacc program. Consider the following example:

```
main(){
 yyparse();
}
# include <stdio.h>
yyerror(str) char *str; {
fprintf(stderr, "%s\n," str);
}
```

The argument to yyerror() is a string "str" containing an error message. The default stack type in YACC is integer type. If we want to change the stack type, we can overload stack definition using %union as follows:

```
%union {
body of union ...
}
```

This declares the Yacc value stack, and the global variables *yylval*. If Yacc was invoked with the -**d** option, the union declaration is copied onto the *y.tab.h* file.

Using ambiguous grammars

When ambiguous grammar is used in translation rules, we get conflicts. So to resolve conflicts, we need to supply precedence and associativity of operators using %left…. %right… For example, when ambiguous grammar is given,then specifying following:

```
%left "+" "-"
%left "*" "/"
```

Indicates that *,/ has higher precedence than +, – and are left associative.

Error recovery with YACC

In YACC, error recovery is performed using error production. We add extra productions for taking care of errors as follows:

```
Line  : Line Expr  "\n" {printf("%d\n,"$2);}
| error   "\n"    {yyerror("Enter last line once again");
yyerrok;}
```

Errors (sent by lexical analyzer as error tokens) are parsed using the second rule shown in the above grammar. So when an error is captured, the error message "Enter last line once again" will appear. "yyerrok" is a yacc-defined macro that resets parser to its normal mode of operation. Let us have an example:

Example 16:

Write a YACC program for evaluation of expression like "8*8+8."

Solution:

If we want to use *hand coded yylex()* for lexical analyzer with yacc, first prepare a YACC file "test.y" as follows:

Yacc program

```
%{
#include<stdio.h>
%}
%token   NUM
%%
Line:Line Expr "\n"
{printf("value of expr=%d\n,"$2);}
    |;
Expr:Expr "+" Term {$$=$1+$3;}
     | Term
   ;
Term:Term "*" Factor {$$=$1*$3;}
   |Factor
   ;
```

```
Factor:NUM {$$=$1;}
    ;
%%
#include "lex.yy.c"
main()
{
yyparse();
}
void yyerror(char *s)
{printf("%s,"s);
}
yylex()
{
char c;
while((c=getchar())!="\n")
{
       if(isdigit(c))
       {
       yylval=c-"0";
       return NUM;
       }
       else return c;
}
return c;
}
```

To run the above program, execute the commands as follows:

```
$yacc test.y
$cc y.tab.c
$./a.out
8+8*8
72
$
```

If we want to use the "lex tool" instead of hand-coded yylex(), then first prepare the lex file "test.l" as follows:

Lex program

```
Digit     [0-9]*
%%
{Digit}   {sscanf(yytext,"%d,"&yylval); return NUM;}
[*-/+]    { return yytext[0];}
"\n"      { return yytext[0];}
```

Let us use the lex tool with ambiguous grammar. The yacc program "test.y is as follows:

```
%{
#include<stdio.h>
%}
%token   NUM
%left "+" "-"
%left "*" "/"
%%
Line:Line Expr "\n"  {printf("value of expr=%d\n,"$2);}
    |;
Expr:Expr "+" Expr                    {$$=$1+$3;}
   |Expr "*" Expr                     {$$=$1*$3;}
   |Expr "-" Expr                     {$$=$1-$3;}
   |Expr "/" Expr                     {$$=$1/$3;}
   |NUM                               {$$=$1;}
   ;
%%
```

```
#include "lex.yy.c"
main()
{
yyparse();
}
void yyerror(char *s)
{printf("%s,"s);
}
}
```

To run the above program, execute the commands as follows:

```
$lex test..l
$yacc test.y
$cc y.tab.c
$./a.out
8*8+8
72
$
```

Solved Problems

1. Consider the following SDT. If an LR parser carries out the translations on an input string "aabbccdb," what is the output?

S → aaS	{ print("a");}
∣ bA	{ print("b");}
∣ b	{ print("c");}
A → bcA	{ print("d");}
∣ cdS	{ print("e");}

Solution: Take the input string; draw the parse tree

The first reduction is "b to S." When this is reduced, the corresponding semantic action is print("c");

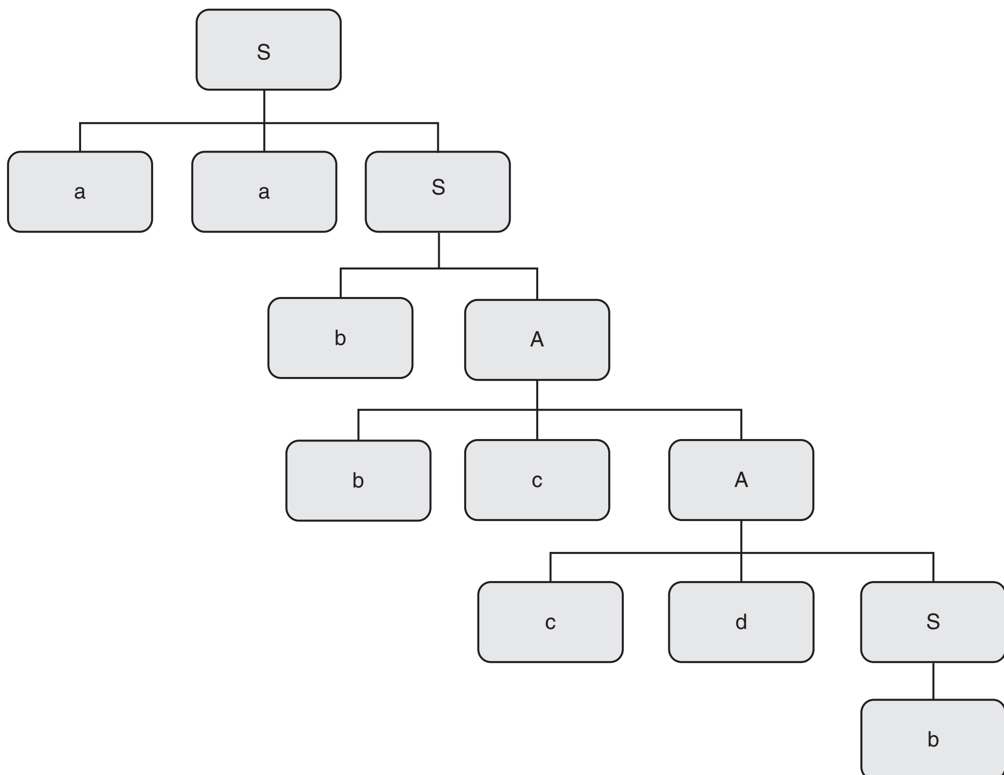
The next reduction is "cdS" to A. When this is reduced, the corresponding semantic action is print("e");

The next reduction is "bcA" to A. When this is reduced, the corresponding semantic action is print("d");

The next reduction is "bA" to S. When this is reduced, the corresponding semantic action is print("b");

The final reduction is "aaS" to S. When this is reduced, the corresponding semantic action is print("a");

Hence the final result is the string **"cedba"** is printed.



*2. Consider the SDT given below.

E → E # T	{ E•val = E•val * T•val;}
E → T	{ E•val = T•val;}
T → T & F	{ T•val = T•val + F•val;}
T → F	{ T•val = F•val;}
F → num	{ F•val = num•val;}

What is the output for an input "2#3&5#6&4"**?**

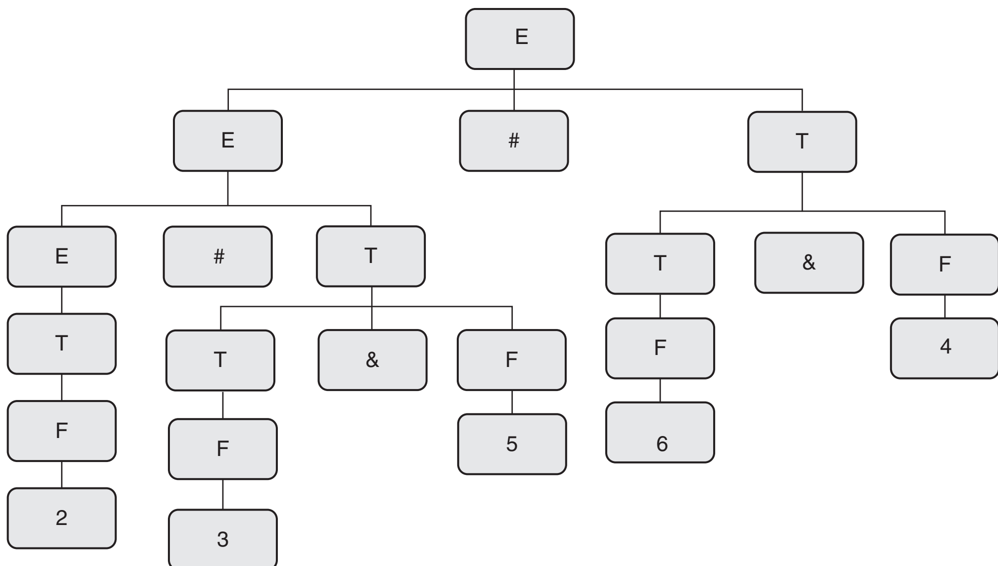
Solution: Draw the parse tree for the string "2#3&5#6&4" as follows:
Here, look at semantic actions, "&" is used for addition and "#" is used for multiplication.
So the given string can be reduced as follows after replacing "&" by "+" and "#" by "*."

2 * 3 + 5 * 6 + 4.

If you look at the grammar, "+" has higher precedence than "*" and both are left associative.

Hence, the given string is evaluated as follows:

2 * (3 + 5) * (6 + 4) = 2 * 8 * 10 = **160.**



*3. Consider the SDT shown below

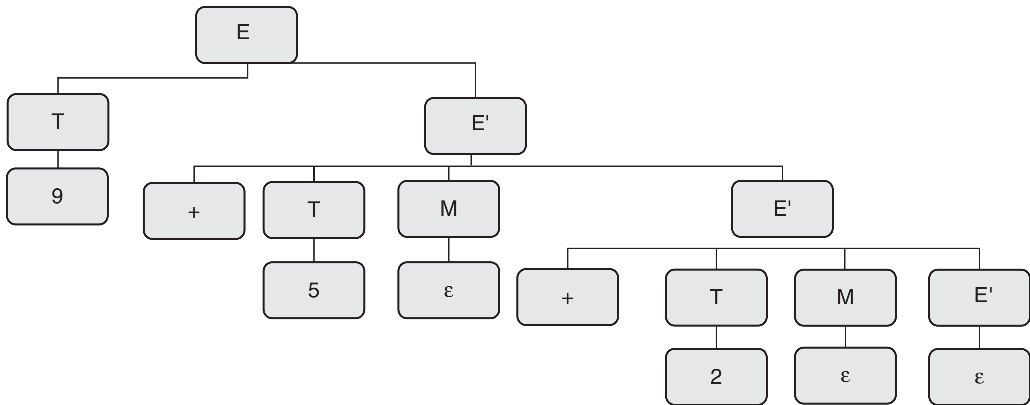
```
E  → TE"
E" → + T  {print("+");}  E" | ε
T  → num
```

Convert it to S-attributed; then check if an LR parser carries out translation on an input "9+5+2"; what is the output?

Solution:

Equivalent S-attributed definition is

```
E  → TE″
E″ → + T M E″ | ε
M  →  ε  {print("+");}
T  → num
```



Draw a parse tree

Now evaluate from the parse tree. The first reduction is 9 to T. So prints 9.
Next is 5 to T, so prints 5. Next ε to M, prints +. Next reduction is 2 to T, prints 2.
Next ε to M, prints +. Next ε to E″, no action.
Final result is 95+2+.

4. Convert the following SDT

$$\begin{aligned} A &\rightarrow A\ \{a\}\ B \\ &\mid B\ \{b\} \\ B &\rightarrow 0\ \{c\} \end{aligned}$$

to an SDT that is postfix SDT and has no left recursion in underlying grammar.

Solution:

First eliminate left recursion. The resulting grammar is as follows:

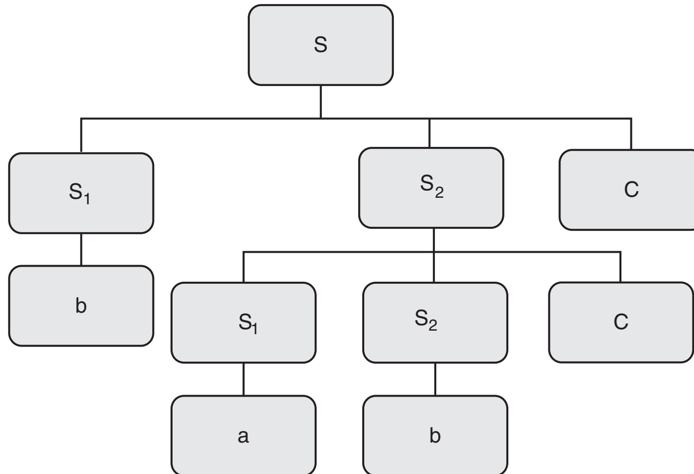
$$\begin{aligned} A &\rightarrow B\ \{b\}\ A'' \\ A'' &\rightarrow \{a\}\ B\ A'' \\ &\mid \varepsilon \\ B &\rightarrow 0\ \{c\} \end{aligned}$$

To get postfix SDT, move actions at right end as follows:

$$A \rightarrow B\ M\ A''$$
$$M \rightarrow \{b\}$$
$$A'' \rightarrow R\ B\ A''$$
$$\mid \varepsilon$$
$$R \rightarrow \{a\}$$
$$B \rightarrow 0\ \{c\}$$

5. Consider the following SDT. If an LR parser carries out the translations on an input string “babcc.” What is the output?

$S \rightarrow S_1S_2c$	$\{S.t = S_1.t - S_2.t;\}$
$\mid a$	$\{S.t=5;\}$
$\mid b$	$\{S.t=2;\}$



Solution: Draw a parse tree and carry out the translations.

When “b” is reduced to S, result is s.t = 2.

Next when “a” is reduced to S, result is s.t = 5.

Next when “b” is reduced to S, result is s.t = 2.

Next when “SSc” is reduced to S, result is 0.3

Last when “SSc” is reduced to S, result is –1.

Summary

- Grammar together with semantic actions is called syntax-directed translation.
- If the value of an attribute at a node is computed from the values of attributes at the children of that node in the parse tree then it is called synthesized attribute.

- If the value of an inherited attribute is computed from the values of attributes at the siblings or parent of that node.
- The parse tree that shows attribute values at each node is called annotated or decorated parse tree
- Parse tree is even called concrete syntax tree
- An SDT that uses only synthesized attributes is called S-attributed definition.
- An L-attributed definition allows both types. But if an inherited attribute is present there is a restriction. The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.
- Every S-attributed is even L-attributed.
- We can convert an L-attributed definition to equivalent S-attributed definition.
- S-attributed definition is also called postfix SDT.

Fill in the Blanks

1. Consider the SDT given below.
   S → aSb { S.count = S.count + 2;}
   | bSa { S.count = S.count + 2;}
   | ε { S.count = 0;}
   This SDT checks______________.
2. The other name of S-attributed definition is ______________.
3. In SDT, A → BC {C.s = B.s;} attribute ."s" is ______________.
4. Every L-attributed is S-attributed (Yes/No)______________.
5. An attribute that is evaluated in terms of attributes of its parent is called______________.
6. In______________SDD, semantic actions are placed anywhere on the right hand side.
7. Every S-attributed is L-attributed (Yes/No)______________.
8. We cannot convert an L-attribute definition to S-attributed definition. (Yes/No) ______________
9. In SDT, A → BC {A.i = B.i;} attribute "i" is ______________.
10. The interdependencies among attributes are shown by the ______________ graph.

Objective Question Bank

1. Consider the SDT given below.

   E → E + T | T { count = count + 5; print (count);}
   T → T * F | F { count = count * 5;}
   F → i { count = 10;}

   What is the count after parsing the string "i + i * i"?
   (a) 110 (b) 55 (c) 550 (d) 11

2. The SDT, A → BC {C.s = B.s;} is ______________.
   (a) S-attributed (b) L-attributed (c) both (d) none

3. The SDT, $A \rightarrow BC$ {A.i = B.i;} is ______________.
   (a) S-attributed (b) L-attributed (c) both (d) none
4. The SDT, $A \rightarrow BC$ {B.s = C.s;} is ______________.
   (a) S-attributed (b) L-attributed (c) both (d) none
5. An S-attributed definition can be evaluated ______________.
   (a) Top down (b) bottom up (c) both (d) none

Key for Fill in the Blanks

1. total number of a's and b's
2. postfix SDT
3. Inherited
4. N
5. inherited
6. L-attributed definition
7. Y
8. N
9. synthesized
10. dependency

Key for Objective Question Bank

1. b 2. b 3. c 4. d 5. c

CHAPTER 7

Semantic Analysis

A semantic analyzer checks the semantics of a program, that is, whether the language constructs are meaningful or not. A semantic analyzer mainly performs static type checking.

CHAPTER OUTLINE

7.1 Introduction
7.2 Type Systems
7.3 Type Expressions
7.4 Design of Simple Type Checker
7.5 Type Checking of Expressions
7.6 Type Checking of Statements
7.7 Type Checking of Functions
7.8 Equivalence of Type Expressions
7.9 Type Conversion
7.10 Overloading of Functions and Operators
7.11 Polymorphic Functions

A compiler must ensure that the source program follows the syntax and semantic conventions of the source language. Once the syntax is verified, the next task to be performed by a compiler is to check the semantics of the language. A semantic analyzer shown in Figure 7.1 mainly verifies whether the language constructs are meaningful (semantics) or not. This is called even static type checking, which ensures that certain kinds of programming errors will be detected and reported.

7.1 Introduction

Parsing cannot detect some errors. Some errors are captured during compile time called static checking (e.g., type compatibility). Languages like C, C++, C#, Java, and Haskell uses static checking. Static checking is even called *early binding*. During static checking programming errors are caught early. This causes program execution to be efficient. Static checking not only increases the efficiency and reliability of the compiled program, but also makes execution faster.

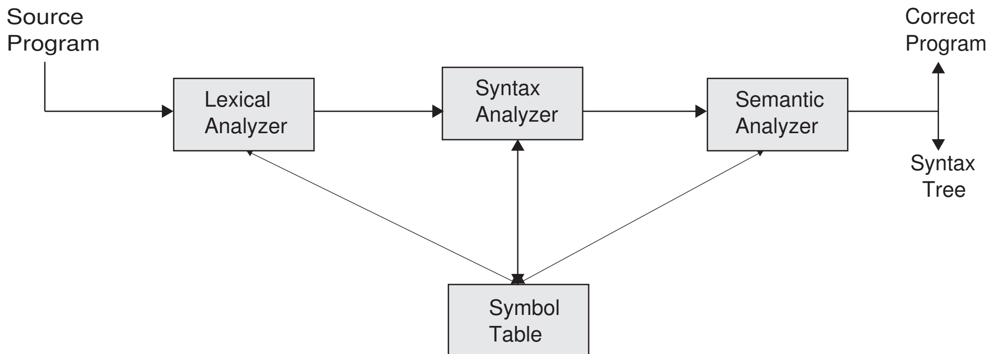


Figure 7.1 Role of a Semantic Analyzer in Compilation

Type checking is not only limited to compile time, it is even performed at execution time. This is done with the help of information gathered by a compiler; the information is gathered during compilation of the source program.

Errors that are captured during run time are called dynamic checks (e.g., array bounds check or null pointers dereference check). Languages like Perl, python, and Lisp use dynamic checking. Dynamic checking is also called late *binding*. Dynamic checking allows some constructs that are rejected during static checking. A *sound* type system eliminates run-time type checking for type errors. A programming language is *strongly-typed*, if every program its compiler accepts will execute without type errors. In practice, some of the type checking operations is done at run-time (so, most of the programming languages are not strongly typed).

For example, int x[100]; … x[i] → most of the compilers cannot guarantee that i will be between 0 and 99

A semantic analyzer mainly performs static checking. Static checks can be any one of the following type of checks:

Uniqueness checks: This ensures uniqueness of variables/objects in situations where it is required. For example, in most of the languages no identifier can be used for two different definitions in the same scope.

Flow of control checks: Statements that cause flow of control to leave a construct should have a place to transfer flow of control. If this place is missing, it is confusion. For example, in C language, "break" causes flow of control to exit from the innermost loop. If it is used without a loop, it confuses where to leave the flow of control.

Type checks: A compiler should report an error if an operator is applied to incompatible operands. For example, for binary addition, operands are array and a function is incompatible. In a function, the number of arguments should match with the number of formals and the corresponding types.

Name-related checks: Sometimes, the same name must appear two or more times. For example, in ADA, a loop or a block may have a name that appears at the beginning and end of the construct. The compiler must check whether the same name is used at both places.

What does semantic analysis do? It performs checks of many kinds which may include

- All identifiers are declared before being used.
- Type compatibility.
- Inheritance relationships.
- Classes defined only once.
- Methods in a class defined only once.
- Reserved words are not misused.

In this chapter we focus on type checking. The above examples indicate that most of the other static checks are routine and can be implemented using the techniques of SDT discussed in the previous chapter. Some of them can be combined with other activities. For example, for uniqueness check, while entering the identifier into the symbol table, we can ensure that it is entered only once. Now let us see how to design a type checker.

A type checker verifies that the type of a construct matches with that expected by its context. For example, in C language, the type checker must verify that the operator "%" should have two integer operands dereferencing is applied through a pointer, indexing is done only on an array, a user-defined function is applied with correct number and type of arguments. The goal of a type checker is to ensure that operations are applied to the correct type of operands. Type information collected by a type checker is used later by code generator.

7.2 Type Systems

Consider the assembly language program fragment. Add R1, R2, R3. What are the types of operands R1, R2, R3? Based on the possible type of operands and its values, operations are legal. It doesn't make sense to add a character and a function pointer in C language. It does make sense to add two float or int values. Irrespective of the type, the assembly language implementation remains the same for add. A language's type system specifies which operations are valid for which types. A type system is a collection of rules for assigning types to the various parts of a program. A type checker implements a type system. Types are represented by type expressions. Type system has a set of rules defined that take care of extracting the data types of each variables and check for the compatibility during the operation.

7.3 Type Expressions

The type expressions are used to represent the type of a programming language construct. Type expression can be a basic type or formed by recursively applying an operator called a type constructor to other type expressions. The basic types and constructors depend on the source language to be verified. Let us define type expression as follows:

- A basic type is a type expression
  - Boolean, char, integer, real, void, type_error
- A type constructor applied to type expressions is a type expression
  - Array: array(I, T)

Array(I,T) is a type expression denoting the type of an array with elements of type T and index set I, where T is a type expression. Index set I often represents a range of integers. For example, the Pascal declaration

var C: array[1..20] of integer;

associates the type expression array(1..20, integer) with C.

- Product: T1 × T2
- If T1 and T2 are two type expressions, then their Cartesian product T1 × T2 is a type expression. We assume that × associates to the left.
- Record: record((N1 × T1) × (N2 × T2))

  A record differs from a product. The fields of a record have names. The record type constructor will be applied to a tuple formed from field types and field names. For example, the Pascal program fragment

```
type node = record
                  address : integer ;
                  data : array [1..15] of char
            end;
var node_table : array [1..10] of node ;
```

declares the type name "node" representing the type expression

record((address×integer) × (data × array(1..15,char)))

and the variable "node_table" to be an array of records of this type.

- Pointer: pointer(T)

  Pointer(T) is a type expression denoting the type "pointer to an object of type T where T is a type expression. For example, in Pascal, the declaration

  *var ptr: *row*

  declares variable "ptr" to have type pointer(row).

- Function: D → R

  Mathematically, a function is a mapping from elements of one set called domain to another set called range. We may treat functions in programming languages as mapping a domain type "Dom" to a range type "Rg.". The type of such a function will be denoted by the type expression Dom → Rg. For example, the built-in function *mod*, i.e. modulus of Pascal has type expression int × int → int.

  As another example, the Pascal declaration

  *function fun(a, b: char) * integer;*

  says that the domain type of function "fun" is denoted by "char × char" and range type by "pointer(integer)." The type expression of function "fun" is thus denoted as follows:

  char × char → pointer(integer)

However, there are some languages like Lisp that allow functions to return objects of arbitrary types. For example, we can define a function "g" of type (integer → integer) → (integer → integer).

That is, function "g" takes as input a function that maps an integer to an integer and "g" produces another function of the same type as output.

Example 1: Write type expression for a pointer to array of real, where the array index ranges from 1 to 100.

Solution: The type expression is ***pointer(array[1..100, real])***

Example 2: Write a type expression for a function whose domains are functions from integers to character and whose ranges are pointer to integer.

Solution:

Type expression is
Domain type expression is ***integer → character***
Range type expression is pointer ***(integer)***
The final type expression is ***(integer → character) → (pointer(integer))***

7.4 Design of Simple Type Checker

Different type systems are designed for different languages. The type checking can be done in two ways. The checking done at compile time is called static checking and the checking done at run time is called dynamic checking. A system is said to be a *Sound System* if it completely eliminates the dynamic check. In such systems, if the type checker assigns any type other than type error for some fragment of code, then there is no need to check for errors when it is run. Practically this is not always true; for example, if an array X is declared to hold 100 elements. Usually the index would be from 0 to 99 or from 1 to 100 depending on the language support. And there is a statement in the program referred to as X[i]; during compilation this would not guarantee error free at runtime as there is possibility that if the value of i is 120 at run time then there will be an error. Therefore, it is essential that there is a need even for the dynamic check to be done.

Let us consider a simple language that has declaration statements followed by statements, where these statements are simple arithmetic statements, conditional statements, iterative statements, and functional statements. The program block of code can be generated by defining the rules as follows:

Type Declarations

P → D ";" E	
D → D ";" D	
| id ":" T	{add_type(id.entry, T.type) }
T → char	{T.type := char }
T → integer	{T.type := int }
:.....	:.....
T → "*" T_1	{T.type := pointer(T_1.type) }
T → array "["num "]" of T_1	{T.type := array(num.value, T_1.type) }

These rules are defined to write the declaration statements followed by expression statements. The semantic rule { add_type(id.entry, T.type) } indicates to associate type in T with the identifier and add this type info into the symbol table during parsing. A semantic rule of the form {T.type := int } associates the type of T to integer. So the above SDT collects type information and stores in symbol table.

7.5 Type Checking of Expressions

Let us see how to type check expressions. The expressions like 3 mod 5, A[10], *p can be generated by the following rules. The semantic rules are defined as follows to extract the type information and to check for compatibility.

$E \rightarrow$ literal	{E.type := *char*}
$E \rightarrow$ num	{E.type := *int*}
$E \rightarrow$ id	{E.type := *lookup(id.entry)*}
$E \rightarrow E_1$ mod E_2	{E.type := if E_1.type = *int* and E_2.type = *int* then *int* else *type_error*}
$E \rightarrow E_1$ "[" E_2 "]"	{E.type := if E_1.type = *array(s, t)* and E_2.type = *int* Th en *t* else *type_error*}
$E \rightarrow$ "*" E_1	{E.type := if E_1.type = *pointer(t)* then *t* else *type_error*}

When we write a statement as **i mod 10**, then while parsing the element i, it uses the rule as $E \rightarrow$ id and performs the action of getting the data type for the id from the symbol table using the lookup method. When it parses the lexeme 10, it uses the rule $E \rightarrow$ num and assigns the type as int. While parsing the complete statement **i mod 10,** it uses the rule $E \rightarrow E_1$ mod E_2, which checks the data types in both E_1 and E_2 and if they are the same it returns int otherwise type_error.

7.6 Type Checking of Statements

The statements are simple of the form "a = b + c" or "a = b." It can be a combination of statements followed by another statement or a conditional statement or iterative. To generate either a simple or a complex group of statements, the rules can be framed as follows: To validate the statement a special data type ***void*** is defined, which is assigned to a statement only when it is valid at expression level, otherwise *type_error* is assigned to indicate that it is invalid. If there is an error at expression level, then it is propagated to the statement, from the statement it is propagated to a set of statements and then to the entire block of program.

```
P → D  ";"  S
S → id  ":="  E            {S.type := if lookup(id.entry)= E.type
                                        then void
                                        else type_error}
S → S1 ";"  S2             {S.type := if S1.type = void and S2.type
                              = void
                                        then void
                                        else type_error}
S → if  E  then  S1        {S.type := if E.type = boolean
                                        then S1.type
                                        else type_error}
S → while  E  do  S1       {S.type := if E.type = boolean
                                        then S1.type
                                        else type_error}
```

7.7 Type Checking of Functions

The rules for writing a function and for associating type expression with non terminal T are written as follows:

```
T → T1 "→"  T2
    {T.type := T1.type  →  T2.type}
E → E1  "("  E2  ")"
    {E.type := if E1.type = s → t and E2.type = s
           then t else type_error}
```

These rules specify that an expression formed by applying E_1 to E_2, the type of E_1 must be a function of the form s → t from the type s of E_2 to some range type t, makes the type of $E_1(E_2)$ is t.

7.8 Equivalence of Type Expressions

The two expressions are equal if both are of the same basic type; otherwise, it is type_error. To check for type equivalence of two expressions constructed using basic types, there must be a well-defined procedure to check for the equivalence. Such equivalence between two type expressions is structural equivalence.

7.8.1 Structural Equivalence

A type expression can be a basic type or the one obtained by applying a constructor to the basic type. Two type expressions T_1 and T_2 are said to be structurally equivalent if they are of the same basic type or obtained by applying same constructor to the same basic type.

Example 3:

An integer is structurally equivalent to another integer
Pointer(char) is structurally equivalent to Pointer(char)
Integer is not structurally equivalent to char

A recursive algorithm for testing the structural equivalence is designed, which considers the basic types and the type constructors for arrays, products, pointers, and functions.

Algorithm: To check structural equivalence of two type expressions.

Input: Expressions s and t
Output: true if equivalent otherwise false.

If s and t are basic types, then return true
else If $s = array(s_1, s_2)$ and $t = array(t_1, t_2)$ then
    return true if equal(s_1, t_1) and equal(s_2, t_2)
else If $s = s_1 \times s_2$ and $t = t_1 \times t_2$ then
    return true if equal(s_1, t_1) and equal(s_2, t_2)
else If $s = pointer(s_1)$ and $t = pointer(t_1)$ then
    return true if equal(s_1, t_1)
else If $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$ then
    return true if equal(s_1, t_1) and equal(s_2, t_2)
else return false.

7.8.2 Encoding of Type Expressions

To check equivalence of type expression we can encode type expression using bit representation. For instance, we can use four bits to represent different basic data types as follows:

```
Basic Type      Encoding
boolean         0000
char            0001
integer         0010
real            0011
```

To represent the constructed types expressions, we can use the corresponding number of integers. For instance, let us assume that there are arrays, pointers, and functions where pointer(t) represents as a pointer of type t, freturns(t) represents a function of some argument that returns an object of type t, and array(t) denotes as array of elements of type t. These can be represented by using two bits as follows:

```
Type constructor     encoding
pointer              01
array                10
freturns             11
```

Each of the above constructors is a unary operator. Type expressions formed by applying these constructors to a basic type have a uniform structure. Each constructor can be

expressed by a sequence of bits using a simple encoding scheme. For instance, the possible constructed type expressions are integer, freturns(integer), pointer(freturns(integer)), array(pointer(freturns(integer))). If we assume that the maximum number of constructors is three then we require 6 bits to encode the complete type expression. For the above type expressions, the encoding is as follows:

```
Type expression                          encoding
integer                                  000000 0010
freturns(integer)                        000011 0010
pointer(freturns(integer))               000111 0010
array(pointer(freturns(integer)))        100111 0010
```

To check for the equivalence, we can compare the bit sequence because two different type expressions cannot be represented with the same sequence if they are formed with different basic types of type constructors.

Note: Here we considered only three constructed types and four basic types. The bit representation can be modified depending on the number of basic types, number of constructors that may be included, and the maximum number of constructors that can be included.

Merits of structural equivalence

i. No type names are needed to check the equivalence.
ii. Gives exact internal representation of types.

Demerits of structural equivalence

i. Very tedious, as substitution has to be done for every type name.
ii. While checking structural equivalence, infinite looping or cycles may be encountered.

7.8.3 Name Equivalence

In some languages, the type expressions are given names. To handle such situations the type expressions are allowed to be named; hence, even names appear in the type expressions. For example, in Pascal program fragment

```
type link = ↑ node;
var   next : link;
     last :  link;
     a    : ↑ node;
     b,c  : ↑ node;
```

the link is declared as a type node. To check whether next, last, a, b, c are identical, we compare for equivalence in terms of the names in the type expression. Name equivalence views each type name as a distinct type, so two type expressions are name equivalent if and only of they are identical.

Note: link, next, last, a, b, c are structural equivalent
a, b, c – name equivalent
next, last – name equivalent

Merits of Name Equivalence

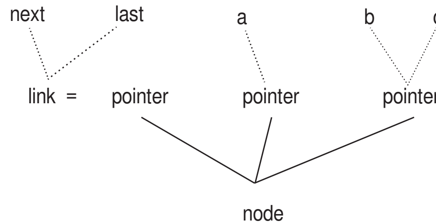
i. Easy to implement
ii. Types with different names are treated differently
iii. Faster in checking type equivalence since no substitution is done.

Demerits of Name equivalence

i. Not systematic
ii. Needs a type name for every type expression.

7.8.4 Type Graph

To check for the type equivalence, type graph can be used. To construct a type graph, a node is created for every constructed type or basic type that is encountered. For every type name, a leaf node is created. Two type expressions are said to be equivalent if they are represented by the same node in the type graph. The following graph shows the equivalence for next, last, a, b, and c in the above example.

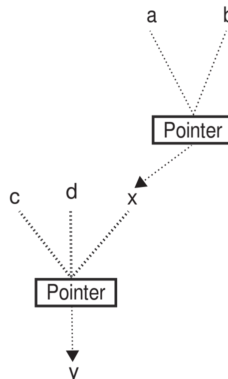


Example 5:

Draw the type graph for the following

```
type    x = ↑ y;
var     a : ↑ x;
        b : ↑ x;
        c : ↑ y;
        d : ↑ y;
```

Solution:



A type graph can also have cycles in it. For example, consider the following example in Pascal.

```
type link = ↑ cell;
type cell = record
        x : int,
     next : link
   end;
```

Note that type name link is defined in terms of a cell and that cell is defined in terms of a link. So the definitions are recursive. Recursively defined type names can be substituted out by introducing cycles in type graph as shown in Figure 7.2. If pointer(cell) is substituted for link, the type expression shown in Figure 7.3 is obtained for a cell. We cannot use structural equivalence if there are cycles in type expressions.

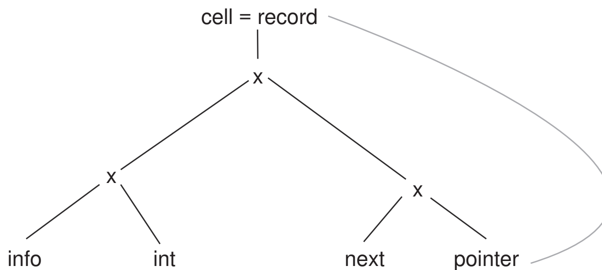


Figure 7.2 Type Graph with Cycles

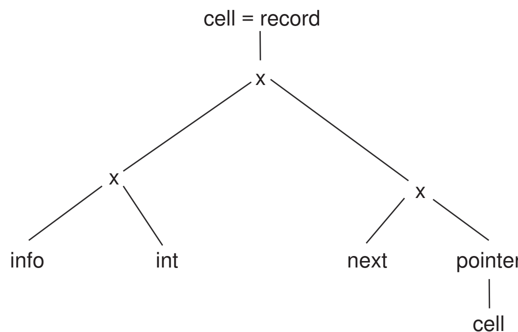


Figure 7.3 Type Graph without Cycles

C uses structural equivalence to avoid cycles like in Figure 7.3. In C the declaration for cell would look like

```
struct cell {
        int info;
        struct cell * next;
};
```

7.9 Type Conversion

In an expression, if there are two operands of different type, then it may be required to convert one type to another in order to perform the operation. For example, the expression "a + b," if *a* is of integer and *b* is real, then to perform the addition *a* may be converted to real. The type checker can be designed to do this conversion. The conversion done automatically by the compiler is implicit conversion and this process is known as *coercion*. If the compiler insists the programmer to specify this conversion, then it is said to be explicit. For instance, all conversions in Ada are said to be explicit. The semantic rules for type conversion are listed below.

$E \rightarrow$ num	{E.type := *int*}
$E \rightarrow$ num.num	{E.type := *real*}
$E \rightarrow$ id	{E.type := *lookup(id.entry)*}
$E \rightarrow E_1$ op E_2	{E.type := if E_1.type = *int* and E_2.type = *int* then *int* else if E_1.type = *int* and E_2.type = *real* then *real* else if E_1.type = *real* and E_2.type = *int* then *real* else if E_1.type = *real* and E_2.type = *real* then *real* else *type_error*}

7.10 Overloading of Functions and Operators

An operator is overloaded if the same operator performs different operations. For example, in arithmetic expression a + b, the addition operator "+" is overloaded because it performs different operations, when a and b are of different types like integer, real, complex, and so on. Another example of operator overloading is overloaded parenthesis in ada, that i, the expression A(i) has different meanings. It can be the ith element of an array, or a call to function A with argument I, and so on. Operator overloading is resolved when the unique definition for an overloaded operator is determined. The process of resolving overloading is called operator identification because it specifies what operation an operator performs. The overloading of arithmetic operators can be easily resolved by processing only the operands of an operator.

Like operator overloading, the function can also be overloaded. In function overloading, the functions have the same name but different numbers and arguments of different types. In Ada, the operator "*" has the standard meaning that it takes a pair of integers and returns an integer. The function of "*" can be overloaded by adding the following declarations:

Function "*"(a,b: integer) return integer.
Function "*"(a,b: complex) return integer.
Function "*"(a,b: complex) return complex.

By addition of the above declarations, now the operator "*" can take the following possible types:

1. It takes a pair of integers and returns an integer
2. It takes a pair of integers and returns a complex number
3. It takes a pair of complex numbers and returns a complex number

Function overloading can be resolved by the type checker based on the number and types of arguments.

The type checking rule for function by assuming that each expression has a unique type is given as

```
E → E1(E2)        { E.type : = t
                    E2.type : = t → u then
                    E.type : = u
                    else E.type : = type_error
                  }
```

If the syntax-directed definition for expression has more than one possible type, then type checking can be performed with the following semantic rules:

```
E' → E          {E'.type := E.type}
E → id          {E.type := lookup(id.entry)}
E → E1(E2)      {E.type := { u | there exists an s in E2.type
                        Such that s → u is in E1.type }
```

An overloaded identifier may have several types saved in the symbol table. The function lookup() returns this set. The type of expression $E_1(E_2)$ is u if s is one of the types of E_2 and the function maps s to u.

7.11 Polymorphic Functions

A piece of code is said to be *polymorphic* if the statements in the body can be executed with different types. A function that takes the arguments of different types and executes the same code is a polymorphic function. The type checker designed for a language like Ada that supports polymorphic functions, the type expressions are extended to include the expressions that vary with type variables. The same operation performed on different types is called overloading and are often found in object-oriented programming. For example, let us consider the function that takes two arguments and returns the result.

int add(int, int)
int add(real, real)
real add(real, real)

The type expression for the function add is given as

$\text{int} \times \text{int} \rightarrow \text{int}$
$\text{real} \times \text{real} \rightarrow \text{int}$
$\text{real} \times \text{real} \rightarrow \text{real}$

Solved Problems

1. Write type expression for an array of pointer to real, where the array index ranges from 1 to 100.

Solution: The type expression is ***array[1..100,pointer(real)]***

2. Write a type expression for a two-dimensional array of integers (that is, an array of arrays) whose rows are indexed from 0 to 9 and whose columns are indexed from –10 to 10.

Solution: Type expression is ***array[0..9, array[-10..10,integer]]***

3. Write a type expression for a function whose domains are functions from integers to pointers to integers and whose ranges are records consisting of an integer and a character.

Solution: Type expression is

Domain type expression is ***integer → pointer(integer)***

Let range has two fields a and b of type integer and character respectively.

Range type expression is ***record((a × integer)(b × character))***

The final type expression is ***(integer → pointer(integer)) → record ((a × integer) (b × character))***

4. Consider the following program in C and the write the type expression for abc.

```
typedef struct {
        int a,b;
        } NODE;
        NODE abc[100];
```

Solution: The type expression for NODE is ***record((a × integer) × (b × integer))***

abc is an array of NODE; hence, its type expression is

array[0..99, record((a × integer) × (b × integer))]

5. Consider the following declarations.

```
type cell=record
        info: integer;
        next: pointer(cell)
type    link = ↑ cell;
var     next = link;
        last = link;
        p = ↑ cell;
        q,r = ↑ cell;
```

Among the following, which expressions are structurally equivalent? Which are name equivalent? Justify your answer.

   i. link
   ii. Pointer(cell)

iii. Pointer(link)
iv. Pointer (record ((info × integer) × (next × pointer (cell))).

Solution: Let A = link
B = pointer (cell)
C = pointer (link)
D = Pointer (record ((info × integer) × (next × pointer(cell))).

To get structural equivalence we need to substitute each type name by its type expression.

We know that, link is a type name. If we substitute pointer (cell) for each appearance of link we get,

A = pointer (cell)
B = pointer (cell)
C = pointer (pointer (cell))
D = Pointer (record ((info × integer) × (next × pointer (cell))).

We know that, cell is also type name given by

type cell=record
    info: integer;
    next: pointer(cell)

Substituting type expression for cell in A and B, we get

A = pointer (record ((info × integer) × (next × pointer (cell)))
B = pointer (record ((info × integer) × (next × pointer (cell)))
C = pointer(pointer(cell))
D = Pointer (record ((info × integer) × (next × pointer (cell))).

We have not substituted for the type expression of cell in "C" as it is anyway different from A, B, and D. That is, even if we substitute in C, the type expression will not be the same for A, B, C, and D.

We can say that A, B, and D are structurally equivalent.

For name equivalence, we will not do any substitutions. Rather we look at type expressions directly. If they are the same then we say they are name equivalent.

None of A, B, C, D are name equivalent.

Summary

- Static type checking is done at compile time.
- Dynamic checking is done at run time.
- Semantic rules support the type checker for verifying the type of variables.
- A system is said to be a *Sound System* if it completely eliminates the dynamic check.
- type_error is a special type used for expressing that the type expression is invalid.
- Variables that are name equivalent are said to be structural equivalent.
- Structural equivalent elements need not be name equivalent.

Fill in the Blanks

1. ________ is type expression for the char a[10].
2. ________ avoids dynamic type checking.
3. ________ is the type expression used to express type for statements.
4. Two variables that are name equivalent are also ________.
5. ________ is a special type used for expressing invalid type.
6. The conversion done automatically by the compiler implicitly is known as ________.
7. Error checking performed at compile time is ________.
8. Error checking performed at run time is ________.
9. Static checking is done in languages ________.
10. Late binding is also called ________.

Objective Question Bank

1. Give type expression for "x" in var x : array[2..20] of tables
   (a) array[0..18, table] (b) array[2..20, table]
   (c) array[table, 0..18] (d) array[table, 2..20]
2. Which of the following is an example of semantic checking?
   (a) $E \rightarrow E_1 + E_2$ {E.val := E_1.val + E_2.val }
   (b) $E \rightarrow E_1 + E_2$ {E.type := if E_1.type = *int* and E_2.type = *int*
   then *int*
   else *type_error*}
   (c) Both
   (d) None
3. Which of the following semantic rule is valid for type checker of $E \rightarrow E + E$?
   (a) {E.type := if E_1.type = *int* && E_2.type = *int* then *int* else *type_error*}
   (b) {E.type := if E_1.type = *int* | | E_2.type = *int* then *int* else *type_error*}
   (c) {E.type := if E_1.type = *int* &! E_2.type = *int* then *int* else *type_error*}
   (d) None
4. ________ are constructed data types.
   (a) arrays (b) structures (c) pointer (d) all
5. While performing type checking ________ special basic types are needed.
   (a) 1 (b) 2 (c) 3 (d) 4
6. The semantic way of expressing type of a language construct is called ________.
   (a) semantic expression (b) type expression
   (c) type system (d) all

7. The constructed data types are build using ________ data types.
   (a) Enumerated (b) Boolean
   (c) Basic (d) Builtin

8. Semantic errors can be detected ________.
   (a) only at compile time (b) only at run time
   (c) both at compile time and at run time (d) none

9. Type checking done by the compiler is called ________.
   (a) static checking (b) dynamic checking
   (c) semantic checking (d) none

10. Two type expressions are ________ if two expressions are either the same basic type or are formed by applying the same constructor to structurally equivalent.
    (a) name equivalent (b) structural equivalent
    (c) valid (d) invalid

Exercises

1. Which of the following type expressions are equivalent?
   e_1 = integer → e_1
   e_2 = integer → (integer → e_2)
   e_3 = integer → (integer → e_1)
2. Among the following expressions, which expressions are structurally equivalent? Which are name equivalent?
   i. node.
   ii. pointer(n)
   iii. pointer(node)
   iv. pointer(record(info × integer) × (next × pointer(n)))
3. Express, using type variables, the type of the following functions.
   (a) The function ref that takes as argument an object of any type and returns a pointer to that object.
   (b) A function that takes as arguments an array indexed by integer, with elements of any type, and returns an array whose elements are the objects pointed to by the elements of the given array.
4. Compute the type expressions for the following program fragments.
   (a) c: char; i : integer; c mod i mod 3
   (b) p: ↑ integer; a : array[10] of integer; a[p↑]
   (c) f : integer → boolean;
       i : integer; j : integer; k : integer;
       k : = i
       i : = j mod i;
       j : = k

Key for Fill in the Blanks

1. array[0..9,char]
2. sound system
3. void
4. structural equivalent
5. type_error
6. coercions
7. static checking
8. dynamic checking
9. C,C++
10. dynamic checking

Key for Objective Question Bank

1. b 2. b 3. a 4. d 5. d 6. b 7. c 8. c 9. a 10. b

CHAPTER 8

Intermediate Code Generation

Intermediate code generator makes target code generation easier. It takes the hierarchical representation of the source program as parse tree and prepares linear representation that is simple and easy to analyze.

CHAPTER OUTLINE

8.1 Introduction
8.2 Intermediate Languages
8.3 Types of Three Address Statements
8.4 Representation of Three Address Code
8.5 Syntax-Directed Translation into Three Address Code

8.1 Introduction

The intermediate code is useful representation when compilers are designed as two pass system, i.e. as front end and back end. The source program is made source language independent by representing it in intermediate form, so that the back end is filtered from source language dependence. The intermediate code can be generated by modifying the syntax-directed translation rules to represent the program in intermediate form. This phase of intermediate code generation comes after semantic analysis and before code optimization as shown in Figure 8.1.

Benefits of intermediate code

- Intermediate code makes target code generation easier
- It helps in retargeting, that is, creating more and more compilers for the same source language but for different machines.
- As intermediate code is machine independent, it helps in machine-independent code optimization.

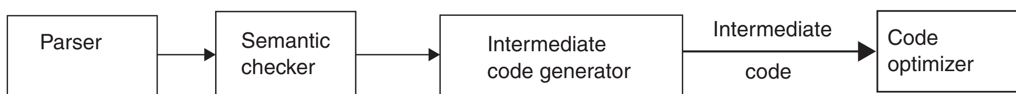


Figure 8.1 Role of Intermediate Code Generator

8.2 Intermediate Languages

Intermediate code can be represented in the following four ways.

1. Syntax trees
2. Directed acyclic graph(DAG)
3. Postfix notation
4. Three address code

8.2.1 Syntax Trees

A syntax tree is a graphical representation of the source program. Here the node represents an operator and children of the node represent operands. It is a hierarchical structure that can be constructed by syntax rules. The target code can be generated by traversing the tree in post order form. For instance, consider an assignment statement a = b* – (c – d) + b* – (c – d) when represented using the syntax tree it appears as follows shown in Figure 8.2.

The rules for constructing the syntax tree for assignment statements are produced by the syntax-directed definition as shown in Figure 8.3

The tree for the statement a = b* – (c – d) + b* – (c – d) is constructed by creating the nodes in the following order.

p_1 = mkleaf(id, c)
p_2 = mkleaf(id, d)
p_3 = mknode('–', p_1, p_2)
p_4 = mknode('U', p_3, NULL)
p_5 = mkleaf(id, b)

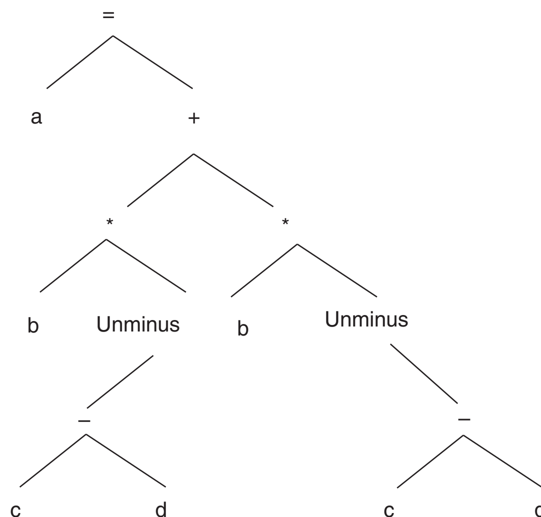


Figure 8.2 Syntax tree for a = b* – (c – d) + b* – (c – d)

Production	Semantic Rule
S → id := E	S.nptr := mknode(‘assign’, mkleaf(id, id.place), E.nptr)
E → E1 + E2	E.nptr := mknode(‘+’, E1.nptr ,E2.nptr)
E → E1 * E2	E.nptr := mknode(‘* ’, E1.nptr ,E2.nptr)
E → – E1	E.nptr := mkunode(‘uminus’, E1.nptr)
E → (E1)	E.nptr := E1.nptr
E → id	E.nptr := mkleaf(id, id.place)

Figure 8.3 SDT for Creating the Syntax tree

p_6 = mknode(‘*’, p_5, p_4)
p_7 = mkleaf(id, c)
p_8 = mkleaf(id, d)
p_9 = mknode(‘–’, p_7, p_8)
p_{10} = mknode(‘U’, p_9, NULL)
p_{11} = mkleaf(id, b)
p_{12} = mknode(‘*’, p_{11}, p_{10})
p_{13} = mknode(‘+’, p_6, p_{12})
p_{14} = mkleaf(id, a)
p_{15} = mknode(‘=’, p_{14}, p_{13})

The tree that is constructed using the above procedures is shown in Figure 8.4.

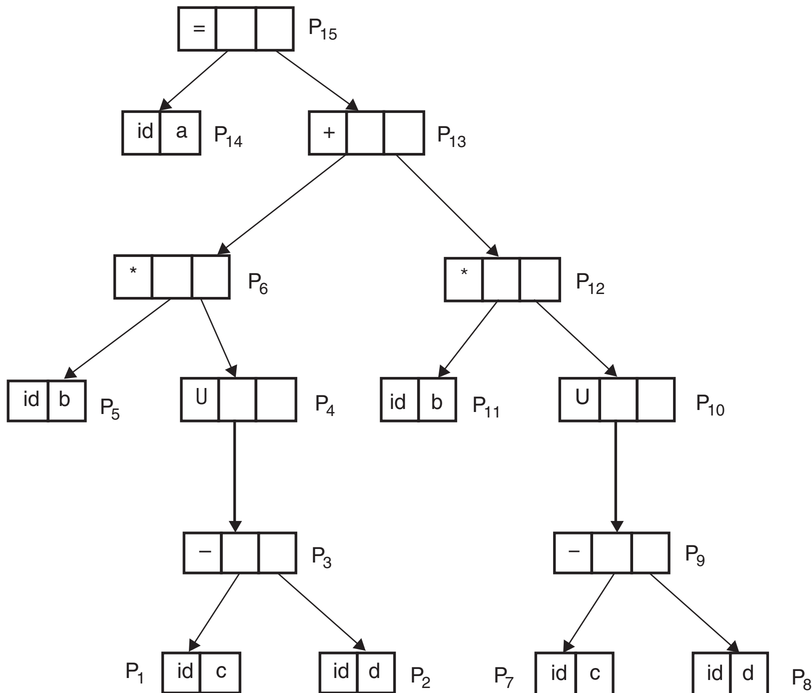


Figure 8.4 Tree Construction for a = b* – (c – d) + b* – (c – d)

8.2.2 Directed Acyclic Graph (DAG)

The tree that shows the same information with identified common sub-expression is called Directed Acyclic Graph (DAG). On examining the above example, it is observed that there are some nodes that are unnecessarily created. To avoid extra nodes these functions can be modified to check the existence of similar node before creating it. If a node exists then the pointer to it is returned instead of creating a new node. This creates a DAG, which reduces the space and time requirement. We can use the same SDT (Figure 8.3) to create a DAG without any modification but mknode() is redefined as above. The list of nodes created in DAG is as follows:

p_1 = mkleaf(id, c)
p_2 = mkleaf(id, d)
p_3 = mknode('–', p_1, p_2)
p_4 = mknode('U', p_3, NULL)
p_5 = mkleaf(id, b)
p_6 = mknode('*', p_5, p_4)
p_7 = mkleaf(id, c) = p_1
p_8 = mkleaf(id, d) = p_2
p_9 = mknode('–', p_1, p_2) = p_3
p_{10} = mknode('U', p_3, NULL) = p_4
p_{11} = mkleaf(id, b) = p_5
p_{12} = mknode('*', p_5, p_4) = p_6
p_{13} = mknode('+', p_6, p_6)
p_{14} = mkleaf(id, a)
p_{15} = mknode('=', p_{14}, p_{13})

The Figure 8.5 shows the constructed DAG for the given expression.

In this tree the number of nodes is reduced and also helps in identifying redundant sub expression. All the nodes in the syntax tree can be visited by following the pointers starting from the root and the instructions can be generated by traversing the tree in post order form.

8.2.3 Postfix Notation

Postfix notation is a linear representation of a syntax tree. This can be written by traversing the tree in the post order form. The edges in a syntax tree do not appear explicitly in postfix notation; only the nodes are listed. The order is followed by listing the parent node immediately after listing its left sub tree and its right sub tree. In postfix notation, the operators are placed after the operands. The postfix notation for the statement is as follows:

a = b* – (c – d) + b* – (c – d) is *a b c d – U * b c d – U * + =*

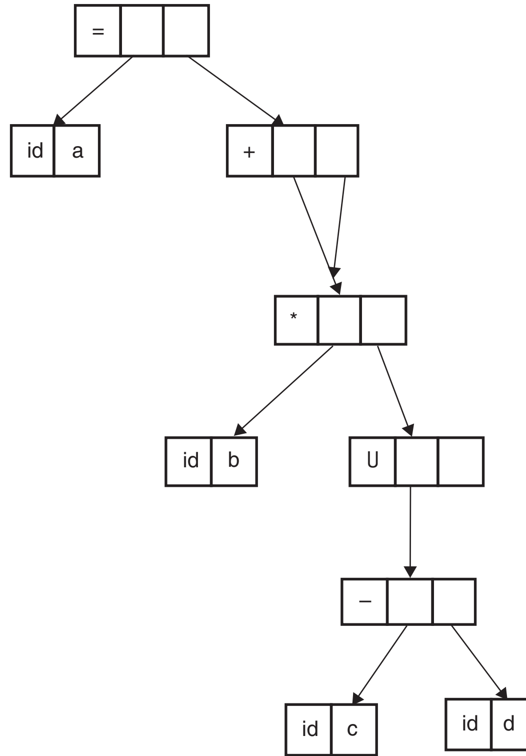


Figure 8.5 DAG Construction for a = b* - (c - d) + b* - (c - d)

8.2.4 Three Address Code

Three address code is a linear representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. Three address code is a sequence of statements of the form A = B ***OP*** C where A, B and C are the names of variables, constants or the temporary variables generated by the compiler. ***OP*** is any arithmetic operation or logical operation applied on the operands B and C. The name reflects that there are at most three variables where two are operands and one is for the result. In three address statement, only one operator is permitted; if the expression is large, then break it into a sequence of sub expressions using the BODMAS rules of arithmetic and store the intermediate results in newly created temporary variables. For example, consider the expression a + b * c; this expression is expressed as follows:

$$T_1 = b * c$$
$$T_2 = a + T_1$$

Here T_1 and T_2 are compiler-generated temporary names. This simple representation of a complex expression in three address code makes the task of optimizer and code generator

simple. It is also easy to rearrange the sequence for efficient code generation. Three address code for the statement a = b* – (c – d) + b* – (c – d) is as follows:

$$T_1 = c - d$$
$$T_2 = -T_1$$
$$T_3 = b * T_2$$
$$T_4 = c - d$$
$$T_5 = -T_4$$
$$T_6 = b * T_5$$
$$T_7 = T_3 + T_6$$
$$a = T_7$$

The code can also be written for DAG as follows:

$$T_1 = c - d$$
$$T_2 = -T_1$$
$$T_3 = b * T_2$$
$$T_4 = T_3 + T_3$$
$$a = T_4$$

Example 1: Consider the assignment a: = b* – c + b* – c. Draw the syntax tree and the DAG.

Solution: The tree and DAG for the expression a: =b* – c + b* – c is shown in Figure 8.6.

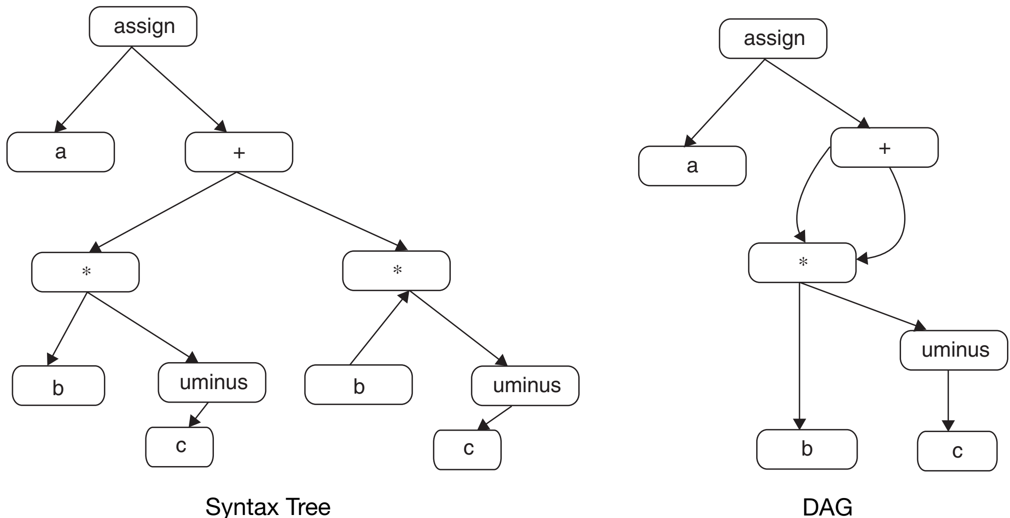


Figure 8.6 Syntax tree and DAG for a = b * – c + b * – c

8.3 Types of Three Address Statements

For expressing the different programming constructs, the three address statements can be written in different standard formats and these formats are used based on the expression. Some of them are as follows:

- *Assignment statements with binary operator*. They are of the form **A := B op C** where op is a binary arithmetic or logical operation.
- *Assignment statements with unary operator*. They are of the form **A: = op B** where op is a unary operation like unary plus, unary minus, shift, etc.
- *Copy statements*. They are of the form **A: = B** where the value of B is assigned to variable A.
- *Unconditional Jumps* such as **goto L**: The label L with three address statement is the next statement number to be executed.
- *Conditional Jumps* such as **if X relop Y goto L**. If the condition is satisfied, then this instruction applies a relational operator (<=,>=,<,>) to X and Y and executes the statement with label L else the statement following if X relop Y goto L is executed.
- *Functional calls:* The functional calls are written as a sequence of **param A**, **call fun,n,** and **return B** statements, where A indicates one of the input argument in *n* arguments to be passed to the function fun that returns B. The return statement is optional. For example, if the statement is
- B fun(A1, A2, A3, ... An), then three address statements for it are as follows:

```
param A1
param A2
param A3
  .
  .
param An
call fun, n
return B
```

- *Indexed assignments* The statements of the form A: = B[i] and A [i]: = B are indexed assignments.
  - In the A: = B[i] statement, A is set to the value in the location i memory units beyond location B.
  - A[i]:=B sets the contents of the location i units beyond A to the value of B. In both these instructions, A, B, and i refer to data objects.
- *Address assignments* Statement of the form A:= &B, which sets A to the location B.
- *Pointer assignment* Statements of the form A:= *B and *A: = B are included. For instance,
  - A:= *B sets the value of A to the value pointed to by B.
  - *A:=B changes the location of the value in A to the address pointed by B.

For any intermediate code generator, the choice of allowable operator is an important issue as this will simplify the optimization and code generation task.

8.4 Representation of Three Address Code

Three address codes can be represented in special structures known as quadruple, triple and indirect triple.

8.4.1 Quadruple

A quadruple is a record structure with four fields. The first field is to store the operator, the second and third fields are for the operands used in the operation, and the fourth field is for the result. For example, the three address statement A=B op C is written by placing op in the first field, that is, operator, B, and C are placed in the second and third fields, that is, operand 1 and operand 2 respectively, and A in fourth field, that is, result. If the operator is unary, then the third field is not used. In case of conditional and unconditional jump statements, the target label is placed in the result field. The quadruple for the statement a = b* – (c – d) + b* – (c – d) is shown below.

We use U for unary minus and – as it is for binary minus.

Operator	Opr1	Opr2	Result
–	c	d	T_1
U	T_1		T_2
*	b	T_2	T_3
–	c	d	T_4
U	T_4		T_5
*	b	T_5	T_6
+	T_3	T_6	T_7
=	T_7		A

8.4.2 Triple

In quadruples there is an overhead for managing the temporary variables created. This problem can be reduced by referring the position of the statement that computes the value of the sub expression. In triples there are only three fields, one for the operation and the remaining two for operands that may have a variable or a constant or the statement position number that computes the value of the operand. Since there are only three fields, it is called “triples.”

Parenthesized numbers represent the position number of the triple structure, while symbol-table pointers are represented by the names themselves. Since the operands represent two different entries, the fields can be encoded in a proper manner. For example, in case of copy statement a:= t1, where t1 is computed at some position (i) then the triplet entries are “=” is placed in the first field, a in the second field and (i) in the third field. For a ternary operation like a[i]:=b, it requires two entries in the structure. The first entry finds the position of i with reference to a and then the actual copy statement as given below.

Statement No	Operator	Arg 1	Arg 2
(0)	[]=	a	i
(1)	=	(0)	b

The triple for the statement a = b* − (c − d) + b* − (c − d) is

Statement No	Operator	Arg 1	Arg 2
(0)	−	c	d
(1)	U	(0)	
(2)	*	b	(1)
(3)	−	c	d
(4)	U	(3)	
(5)	*	b	(4)
(6)	+	(2)	(5)
(7)	=	a	(6)

8.4.3 Indirect Triples

Indirect triples is another implementation of three address code where the listing of pointer to triples is given separately as shown in the following example:

Statement No	Operator	Arg 1	Arg 2
(20)	−	c	d
(21)	U	(20)	
(22)	*	b	(21)
(23)	−	c	d
(24)	U	(23)	
(25)	*	b	(24)
(26)	+	(22)	(25)
(27)	=	a	(26)

Sequence	Statement No
(0)	(20)
(1)	(21)
(2)	(22)
(3)	(23)
(4)	(24)
(5)	(25)
(6)	(26)
(7)	(27)

8.4.4 Comparison of Representations

The main difference between the three forms of structures is that in case of quadruples, temporary variables are created, which need entries in the symbol table. This leads to waste of space. This problem is eliminated in triples and indirect triples. The second difference is regarding the extent of indirection that is present in the representation. For instance, when the target code is produced, each variable or compiler generated temporary variable is assigned some run-time memory location. There would be an entry regarding these addresses in the symbol table. In case of quadruples, the three address statement that uses these temporary variables can be accessed immediately from the symbol table. The advantage of these entries in the symbol table is when the optimizer rearranges the statements for generating efficient code, it requires no changes for the variables. In case of triples moving a single statement that defines a temporary value, many changes have to be made in all the statements that refer to this computation either in arg1 or in arg2. Hence, it is difficult for the optimizer to optimize the code.

In case of indirect triples, this problem does not exist. A statement can be moved by reordering the statement list. Since pointers to temporary values refer to the statements in the actual list of statements, which are not modified, the reference pointers can be reordered easily. The indirect triples are almost similar to quadruples in terms of space and utility. However, indirect triples can save some space compared with quadruples if the same temporary value is used more than once. This is because two or more entries in the statement array can point to the same line in the actual triple structure.

For example, lines (20), (21), and (22) could be combined with (23), (24), and (25).

8.5 Syntax-Directed Translation into Three Address Code

Syntax-directed translation rules can be defined to generate the three address code while parsing the input. It may be required to generate temporary names for interior nodes which are assigned to nonterminal E on the left side of the production E→E1 op E2. While processing, the variable names and code generated so far are tracked till they reach the starting nonterminal. For this purpose, we associate two attributes place and code associated with each nonterminal.

- E.place, the name that will hold the value of E.
- E.code, the sequence of three-address statements evaluating E.

8.5.1 Assignment Statement

To generate intermediate code for assignment statement, first searching is applied to get the information of the identifier from the symbol table. These identifiers are simple or multidimensional array or a constant value that is stored in a literal table. After searching, the three address code is generated for the program statement. Function lookup will search the symbol table for the lexeme and store it in id.place. Function ***newtemp*** is defined to return a new temporary variable when invoked and ***gen*** function generates the

three address statement in one of the above standard forms depending on the arguments passed to it.

Let us consider the example arithmetic statement a = – (b – c). When it is represented as three address code, the statements are

$$T_1 = b - c$$
$$T_2 = - T_1$$
$$a = T_2$$

Let us consider the order in which the statements are generated. First the statement relating to c-d is generated by using a new temporary variable. While parsing the possible rules used for this expression are

$$S \rightarrow id = E$$
$$E \rightarrow - E_1$$
$$E \rightarrow E_1 - E_2$$
$$E \rightarrow id$$

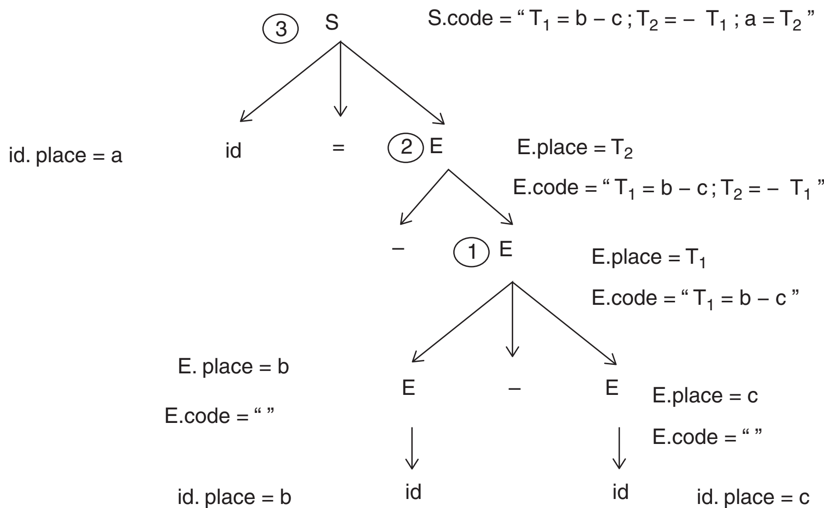


Figure 8.7 Syntax tree for a = –(b – c)

Consider the syntax tree for the expression a = –(b – c) shown in Figure 8.7.

1. Using the production $E \rightarrow id$, that is, $E \rightarrow b$ and $E \rightarrow c$. Check for entries b and c in the symbol table; if these entries are not present an error message is displayed. If these entries are present then E_1.place = b and E_2.place = c (pointer to symbol table for the entry b and c).
2. At node 1 in the figure using the production $E \rightarrow E_1 - E_2$ it creates a temporary variable T_1 using newtemp(). Three address code E.place = E_1.place + E_2.place is generated.
3. At node 2 in the figure using the production $E \rightarrow - E_1$, it creates a temporary variable T_2 using newtemp(). Three address code E.place = – E_1.place is generated.

4. At node 3 using the production S → id = E searches for a in the symbol table, assuming it the code produced is a = E.place

The syntax-directed translation for arithmetic statements can thus be written as follows:

$S \rightarrow id = E$	{ id.place = lookup(id.name); if id.place ≠ null then S.code = E.code ∣∣ gen(id.place ":=" E.place) else S.code=type_error}
$E \rightarrow - E_1$	{E.place = newtemp(); E.code = E_1.code ∣∣ gen(E.place ":=" "–" E_1.place)}
$E \rightarrow E_1 + E_2$	{E.place = newtemp(); E.code = E_1.code ∣∣ E_2.code ∣∣ gen(E.place ":=" E_1.place "+" E_2.place)}
$E \rightarrow E_1 - E_2$	{E.place = newtemp(); E.code = E_1.code ∣∣ E_2.code ∣∣ gen(E.place ":=" E_1.place "–" E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place = newtemp(); E.code = E_1.code ∣∣ E_2.code ∣∣ gen(E.place ":=" E_1.place "*" E_2.place)}
$E \rightarrow E_1 / E_2$	{E.place = newtemp(); E.code = E_1.code ∣∣ E_2.code ∣∣ gen(E.place ":=" E_1.place "/" E_2.place)}
$E \rightarrow id$	{E.place = lookup(id.name), E.code = " "}

8.5.2 Addressing Array Elements

Elements of array are stored in consecutive memory location. If the array is of size n and the size of each element is s, then the ith element of the array can be accessed at the base address + (i – low) * s when the array is single dimension.

Here base is the base address of the array or the address of the first element of array and low is the lower bound of the array or the index of the first element of the array.

Example 2: Let A[5] be an array of 5 elements. Let the size of each element be 2, that is, s = 2 and the array is stored from memory location 100, that is, base address = 100.

A[1]
A[2]
A[3]
A[4]
A[5]

To refer to the third element of the array, the address is calculated as 100 + (3 – 1) * 2 = 104.

In case of the two-dimensional array, the expression is written as $i * s$ + base – low * s where the first sub expression is ($i * s$) and second sub expression is (base – low * s). In the second expression, all the components are known before compilation; hence, they can be pre-computed and stored. This reduces the time taken to generate the address of the ith element. In case of multi-dimension arrays like matrix, elements are either stored as row major or column major order.

Example 3: Consider Array A[3,3] with elements stored in row major order is shown below in Figure 8.8.

A[1,1]
A[1,2]
A[1,3]
A[2,1]
A[2,2]
A[2,3]
A[3,1]
A[3,2]
A[3,3]

Figure 8.8 Array A[3,3]

The address of element A[i,j] in row major order is computed with the expression **base+((i – low_i)* n_2 + j – low_j) * s,** where base is the starting address of the array, low_i is the lower bound of i, low_j is the lower bound of j, n_2 is the number of columns, and s is the size of each element.

This expression can be written as **((i * n_2) + j) * s + (base – ((low_i * n_2) + low_j) * s** The second part of the expression can be pre-computed by knowing the value of base. low_i, low_j, and s. This helps in faster generation of address of A[i,j]. The generalized rules may be as follows:

S → A = E
E → E + E
E → E * E
E → A
A → Alist]
A → id
Alist → Alist, E | id [E

A can be a simple name (has only one base address and no offset) or an indexed name (has base address and object) assignment to location.

S → A = E { If A.offset = null then gen(A.place '=' E.place
else
gen(A.place '[' A.offset']"='E.place)}

E → E_1 + E_2 {E.place = newtemp();
E.code =E_1.code | | E_2.code | | gen(E.place ":=" E_1.place"+"E_2.place)}

```
E → E1 + E2          {E.place = newtemp();
                      E.code= E1.code | | E2.code | | gen(E.place ":=" E1.place "*"E2.place)}
E → A                { If A.offset = null then E.place = A.place
                      else begin
                      E.place=newtemp()
                      gen(E.place '='A.place '[' A.offset']' )
                      end}
A → Alist            { A.place =newtemp()
                      A.offset =newtemp()
                      gen(A.place '=' c(Alist.array)
                      gen(A.offset '=' Alist.place '*' width(Alist.array))}
A → id               { A.place =id.place
                      A.offset =null}
Alist → Alist1, E    { t=newtemp()
                      m= Alist1.dim +1
                      gen(t '=' Alist.place '*' lmt(Alist.array,m))
                      gen(t '=' t '+' E.place )
                      Alist.array=Alist1.array
                      Alist.place = t
                      Alist.ndim=m}
Alist → id [ E       { Alist.array =id.place
                      Alist.place = E.place
                      Alist.ndim = 1}
```

If A A.offset and A.value are the two attributes associated with A as L-values. If A is an array, A.offset is a temporary variable to store the first part of the expression and A.value stores the second part of the expression. If A is a simple variable, then A.value points to the symbol table and A.offset is set to null. Here we define other functions like ***lmt()*** for maximum number of elements present in the jth dimension, ***width()*** for the size of the array, ***m*** to denote the dimension of the array, and ***c*** for the second component of the expression.

8.5.3 Logical Expression

An expression that contains operators like +, –, *, / are simple arithmetic expressions, whereas the expression that contains relational operators like <, >, ≥, ≤, or , and, not, etc., are logical expressions. The logical expressions are mainly used for a set of statements that are to be executed based on the condition that is satisfied, that is, to control the flow of path. The use of logical expression always results in either true or false, which is considered 0/1. 0 indicates false and 1 or a positive number indicates true. The rules for writing the logical expressions are as follows:

$E \rightarrow E_1 \text{ or } E_2$
$E \rightarrow E_1 \text{ and } E_2$
$E \rightarrow \text{not } E_1$
$E \rightarrow id_1 \text{ relop } id_2$
$E \rightarrow (E_1)$

$E \rightarrow$ true
$E \rightarrow$ false

To convert the expression with logical operator requires two things to be added after the evaluation of the expression to true or false.

1. Where the control should go when the condition is true and
2. Where to go when the condition is false.

This is done by first evaluating the expression and then adding the "if" statement that checks the result and sets the value to 0 or 1. While generating three address statement, we use variable ***next*** that keeps track of current statement number that is used to generate the next required statement number for true or false condition. The translation rules to convert to three address code are as follows:

$E \rightarrow E_1$ or E_2	{E.value = newtemp(); gen(E.value "=" E_1.value "or" E_2.value)}
$E \rightarrow E_1$ and E_2	{E.value = newtemp(); gen(E.value "=" E_1.value "and" E_2.value)}
$E \rightarrow$ not E_1	{E.value = newtemp(); gen(E.value "=" "not" E_1.value)}
$E \rightarrow (E_1)$	{E.value = E_1.value}
$E \rightarrow id_1$ **relop** id_2	{E.value = newtemp(); gen("if" id_1.value relop.op id_2.value "goto" nextstat + 3) gen(E.value "=" "0") gen("goto" nextstat + 2) gen(E.value "=" "1")}
$E \rightarrow$ true	{E.value = newtemp(); gen(E.value "=" "1")}
$E \rightarrow$ false	{E.value = newtemp(); gen(E.value "=" "0")}

Example 4: Write three address statement for the ***x or y*** and ***not z***

Solution: Three address code for the above expression is

t_1 = not z
t_2 = y and t_1
t_3 = x or t_2

Example 5: Write three address statement for the ***if x < y then 1 else 0.***

Solution: The address code for the above expression is

10. if x < y go to 13
11. t_1 = 0
12. go to 14
13. t_1 = 1
14.

8.5.4 Control Statements

Flow of control statements can be shown pictorially as in the Figures 8.9 and 8.10 below.

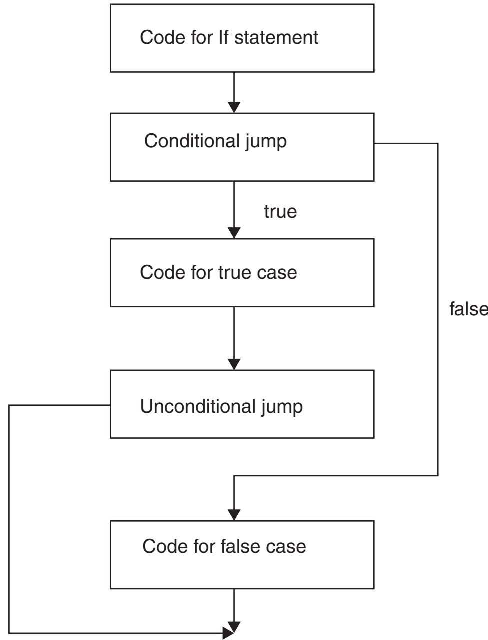


Figure 8.9 Control low for if-then-else statement

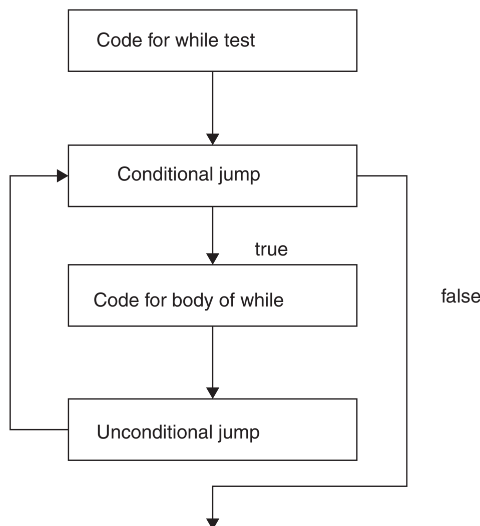


Figure 8.10 Control low for while statement

While generating the code we need to generate the label that will execute the segment of code depending on the condition whether it is true or false. The rules for writing different constructs are as follows:

$S \rightarrow$. if E then S_1
$S \rightarrow$ if E then S_1 else S_2
$S \rightarrow$ while E do S_1

While writing the translation rules for the first rule, we need to generate a new label for the true segment and the next statement is the statement that follows S. Hence, the translation rules are as follows:

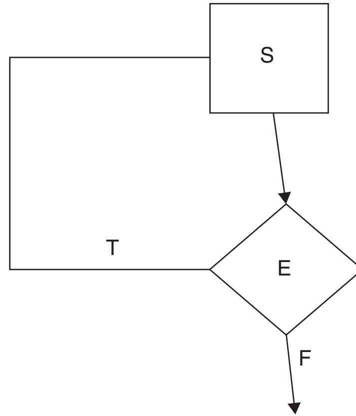
```
S → if E then S1            {E.true = newlabel();
                             E.false = S.next;
                             S1.next = S.next;
                             S.code = E.code || gen(E.true , " :") || S1.code}

S → if E then S1 else S2    {E.true = newlabel();
                             E.false = newlabel();
                             S1.next = S.next;
                             S2.next = S.next;
                             S.code = E.code || gen(E.true , " :") || S1.code ||
                             gen(" GOTO ", S.next) || gen(E.false , " :") || S2.code}

S →  while E do S1          {S.begin = newlabel();
                             E.true = newlabel();
                             E.false = S.next;
                             S1.next = S.next;
                             S.code = gen(S.begin ":") || E.code || gen(E.true , " :") ||
                                         S1.code || gen("GOTO" , S.begin)}
```

Example 6: Give three address code for the following:
While (a < 5) do a: = b + 2

Solution:

```
L1:   If a < 5 goto L2
      goto last
L2:   t1 = b + 2
      a = t1
      goto L1
last:
```

Similarly, we can translate for the "do…while" and the "for" loop.
Translate "do S while (E)."
The flow of control for the "do while" loop is as shown in Figure 8.11.



L1: S
if E goto L1
goto…

Figure 8.11 Control flow of repeat... statement do... while

Example 7: Translate do x = y + z while $a < 10$

Solution: L1:

$t_1 = y + z$
$x = t_1$
If $a < l_0$ goto L_1

Flow of control for the "for" loop is shown in Figure 8.12 for(E1;E2;E3) S;

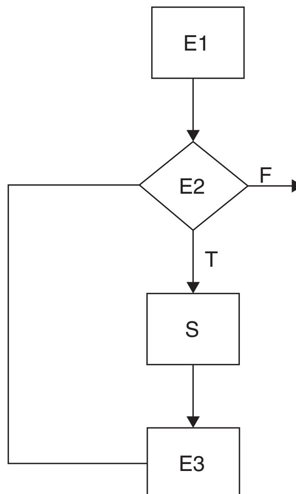


Figure 8.12 Control flow of for loop

Translate the following switch statement.

```
switch(E)
{ case v1: s1;
case v2:s2;
        :
        :
case vn:sn
}
```

The translation is based on the way the flow of control executes the statement and is given as follows:

```
                t1=E
                goto test
    L1:         s1
                goto last
    L2:         s2
                goto last
    Ln:         sn
                goto last
    test: if (t1==v1) goto L1
          if (t2==v2) goto L2
                :
          if (tn==vn) goto Ln
    last:
```

Solved Problems

1. Translate the following IR forms

$$-(a + b) * (c + d) + (a + b + c)$$

Solution:

Postfix form = ab+ – cd+ * ab + c + +

Three address code:

$t_1 = a + b$
$t_2 = -t_1$
$t_3 = c + d$
$t_4 = t_2 * t_3$
$t_5 = a + b$
$t_6 = t_5 + c$
$t_7 = t_4 + t_6$

Syntax tree: The syntax tree for the given expression is shown in Figure 8.13.

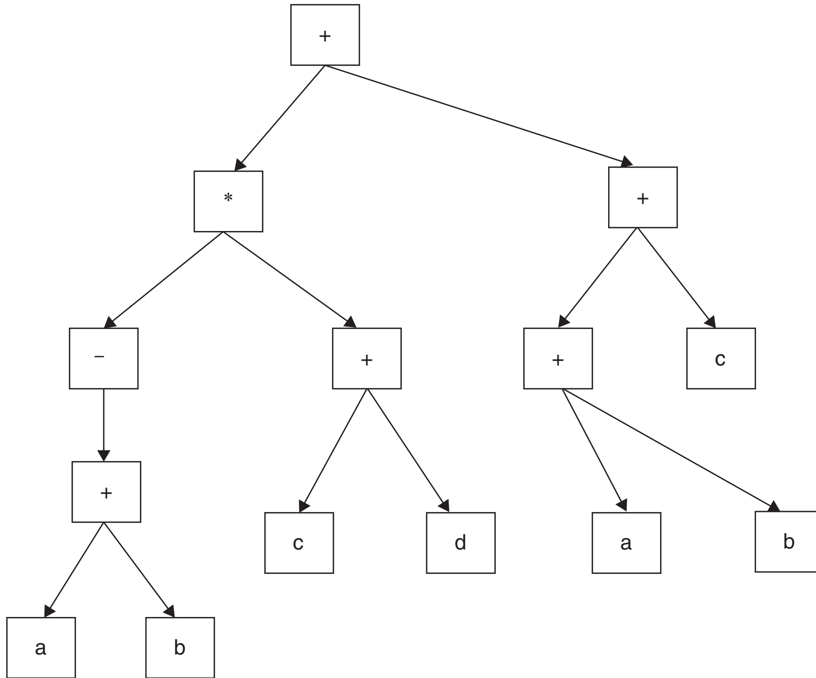


Figure 8.13 Syntax tree

DAG: The DAG for the given expression is shown in Figure 8.14.

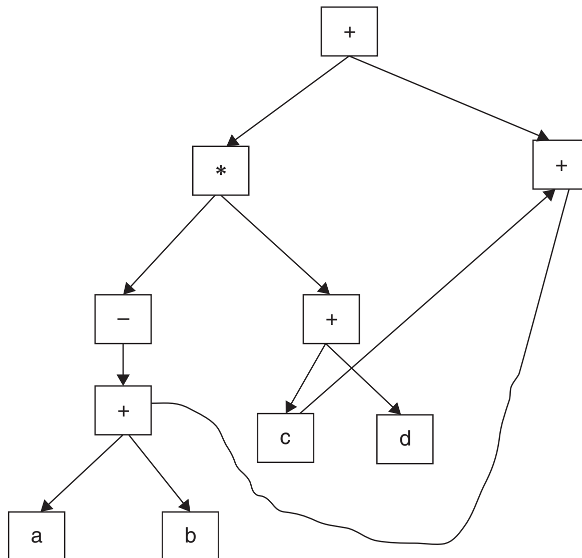


Figure 8.14 DAG

2. Convert into postfix notation and write the three address code.

a * (b + c) / (d + e)

Solution: The postfix form step by step is given by

a* (bc+) /(de+)
(abc+*)/(de+)
abc+*de+/

Three address code is

$t_1 = b + c$
$t_2 = a * t_1$
$t_3 = d + e$
$t_4 = t_2/t_3$

3. Convert into postfix notation and write the three address code.

−a + b / c ↑ d ↑ e*f/g

Solution: postfix form step by step is given by

(a−)+ b / c ↑ d ↑ e*f/g
(a−)+ b / c ↑ (de ↑)*f/g
(a−)+ b / (cde↑↑)*f/g
(a−)+ (bcde↑↑/)*f/g
(a−)+ (bcde↑↑/f*g/)
a−bcde↑↑/f*g/+

Postfix form is: a−bcde↑↑/f*g/+

Three address code:

$t_1 = -a$
$t_2 = d \uparrow e$
$t_3 = c \uparrow t_2$
$t_4 = b / t_3$
$t_5 = t_4 * f$
$t_6 = t_5 / g$
$t_7 = t_1 + t_6$

4. Let A be the two-dimension matrix with the size 10 * 10; generate the address of x=A[y,z].

Solution: The lower bound of both is equal to 1, that is, low1 = low2 = 1 and higher bound is equal to 10, that is, n1 = n2 = 10 and let the size of each element be 4, that is, s = 4. The annotated parse tree generates the address of x = A[y, z] as shown in Figure 8.15.

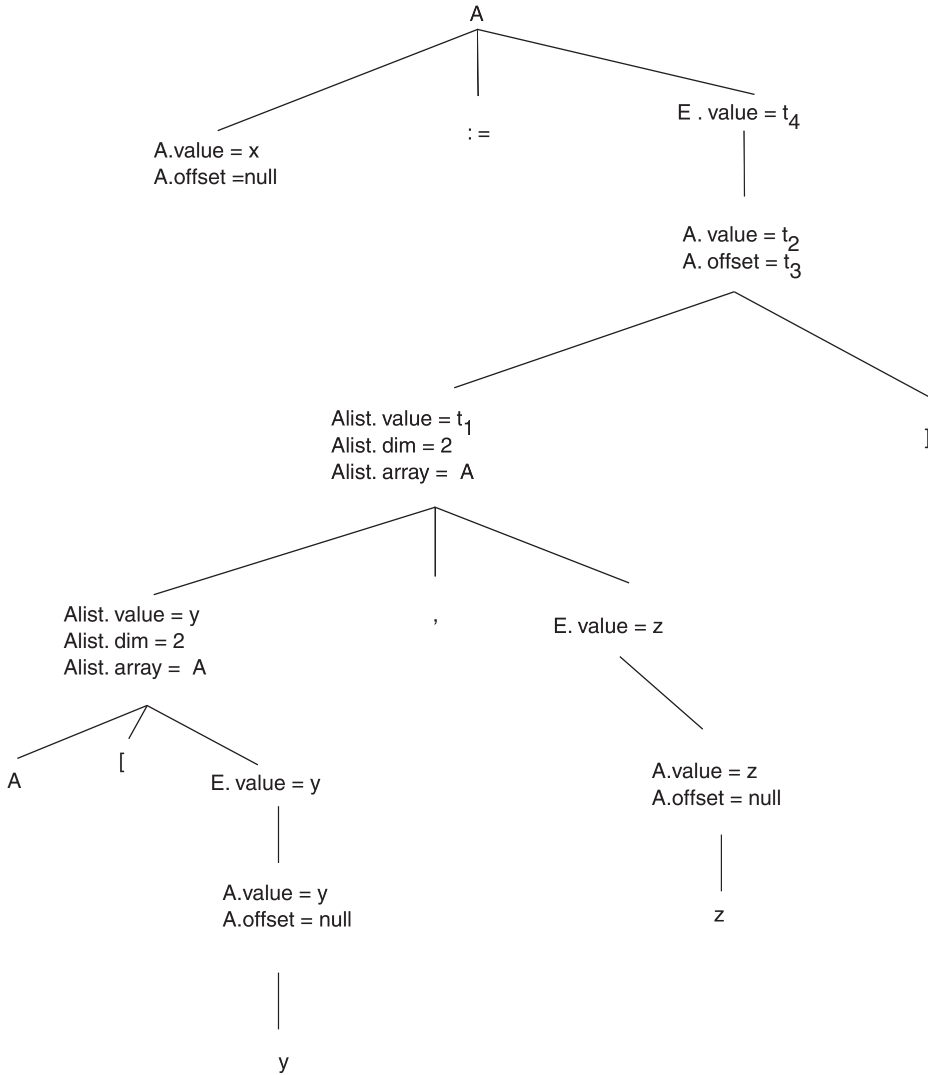


Figure 8.15 Parse tree for x = A[y, z]

5. Write the three-address statement for ***x < y or z < x1 and y1 < z1.***

Solution: Three address code for the above expression will be as follows:

10. if x < y go to 13
11. t_1= 0
12. go to 15
13. t_1 = 1

14. go to 22
15. if $z < x_1$ go to 18
16. $t_2 = 0$
17. go to 23
18. if $y_1 < z_1$ go to 21
19. $t_3 = 0$
20. go to 23
21. $t_3 = 1$
22. go to true
23. false cond

6. Generate the three address code for the following statement:

```
while a<b
do
  if c < d then
     x = y + z
else
     x = y - z
done
```

Solution: The three address code will be as follows:

```
L1. if a<b then GOTO L2
    GOTO LNEXT
L2. if c<d then GOTO L3
    GOTO L4
L3. t1 = y + z
    x = t1
   GOTO L1
L4. t1= y - z
    x = t1
    GOTO L1
LNEXT.
```

7. Generate the three address code for the following C program:

```
main(0
{
int i=1;
int a[10];
while(i<=10)
       A[i] = 10;
}
```

Solution: Three address code is as follows:

```
    i=1
L:  if i<=10 goto L1
    goto last
L1: t1 = i * w
    t2 = addr(A)
    t2[t1]= 10
    Goto L
Last:
```

where w is the width of each array location.

8. Translate the following switch statement.

```
switch(i+j)
{ case 1: x = y + z;
case 2: u = v + w;
default: p = q + r
}
```

Solution:
The translation as follows:

```
         t=i + j;
         goto test
L1:      t1 = y + z;
         x = t1;
         goto last
L2:      t2 = v + w
         u = t2;
         goto last
L3:      t3 = q + r;
         p = t3;
         goto last
test:    if (t==1) goto L1
         if (t==2) goto L2
         goto L3
last:
```

Summary

- The intermediate code is useful representation when the compilers are designed as front end and back end.
- A syntax tree is graphical representation of the source program.
- DAG representation is useful for common sub expression elimination.

- Postfix notation is a linear representation of a syntax tree.
- Three address code is the most commonly used intermediate representation.
- Quadruples use temporary variables to store intermediate results.
- Syntax-directed translation rules can be defined for translating the source program to intermediate code.

Fill in the Blanks

1. __________ rules are defined to generate the three address code.
2. __________ representation makes the program source language independent.
3. __________ is the simplest form of intermediate representation.
4. The postfix notation for the expression x+-y*c is __________.
5. The three address code for the expression x+-y*c is __________.
6. __________ is a special graph that eliminates the nodes for common expression.
7. Three address code for the statement *a= add(int x, int y)*__________.
8. __________ representation of three address code uses temporary variables.
9. __________ representation uses references within the table that has three address code.
10. __________ is a graphical representation of the source program.

Objective Question Bank

1. Which of the following is the postfix representation for the expression x + - y * (− y + z)?

   (a) xy + −y − z* +
   (b) xy − y −z+ * +
   (c) xy − +y −z +*
   (d) xy −y −z +* +

2. _________ is an invalid three address code.

   (a) Quadruple.
   (b) Threaded code
   (c) Triple
   (d) Indirect Triple

3. DAG stands for_________.

   (a) directed adjacent graph
   (b) double adjacent graph
   (c) directed acyclic graph
   (d) double acyclic graph

4. Program is independent of the source language and the target language when it is represented as _______.

(a) Machine code
(b) Intermediate code
(c) Assembly code
(d) Relocatable code

5. Machine-independent optimization is applied on the code when it is in ______.

(a) Intermediate form
(b) Assembly code
(c) Target code
(d) Machine Code

6. The three address code for the statement x + − y * (− y + z) is

(a) $t_1 = -y, t_2 = t_1 + z, t_3 = t_1 * t_2, t_4 = x + t_3$
(b) $t_1 = -y, t_2 = x + t_1, t_3 = t_1 + z, t_4 = t_2 * t_3$
(c) Both a and b are valid
(d) None

7. Temporary variables are generated in ____________.

(a) quadruple
(b) threaded code
(c) triple
(d) indirect triple

8. Statements can be moved in and around in ____________.

(a) quadruple
(b) threaded code
(c) triple
(d) indirect triple

9. Intermediate code can be represented as ____________.

(a) graph
(b) prefix expression
(c) tree
(d) DAG

10. Postfix expression can be evaluated using the ____________data structure.

(a) tree
(b) stack
(c) graph
(d) queue

Exercises

1. Translate the arithmetic expression x*-(y-z) into a syntax tree, postfix notation and three address code.

2. Translate the executable statement of the following C program into a syntax tree, three address code.

```
main()
{
 int i;
 int a[10];
 i = 1;
 while(i ≤ 10)
 {
   a[i] = 0.0; i = i + 1;
 }
}
```

3. Write a program to implement the syntax-directed definition for translating Boolean expression into three address code.
4. Write a program to implement the syntax-directed definition for translating flow- of-control statements into three address code.
5. Translate the following assignment statement into three address code using the translation scheme.
   A[i , j] : = B[i , j] + C[A[k , l]] +D[i + j]

Key for Fill in the Blanks

1. Syntax direction translation
2. Intermediate code
3. Postfix
4. xy – c*+
5. $t_1 = -y$, $t_2 = t_1 * c$, $t_3 = x + t_2$.
6. Directed Acyclic Graph
7. param x, param y, call add 2, return a
8. Quadruples
9. Indirect triples
10. Syntax trees

Key for Objective Question Bank

1. d. 2. b. 3. c. 4. b. 5. a. 6. a. 7. a. 8. a.
9. b. 10. b.

This page is intentionally left blank.

CHAPTER 9

Symbol Table

Symbol table organization is important for improving the efficiency of the compiler. It is important to understand the different forms of symbol table and how it effects the performance while retrieving the variable information

CHAPTER OUTLINE

9.1 Introduction
9.2 Symbol Table Entries
9.3 Operations on the Symbol Table
9.4 Symbol Table Organization
9.5 Non-block Structured Language
9.6 Block Structured Language

Symbol table is an important data structure which stores the information of all the variables and procedures in the program. This table is created during first phase and is used by all the phases for inserting the information or retrieving the information. Organizing the elements in the symbol table shows the impact on the performance of the compiler. This chapter focuses on what type of details are stored in the symbol table, how they are organized and how the arrangement of these elements effect the retrieval time. The focus in mainly on simple array structure, linked list, trees and hash tables for both block and non-block structured languages.

9.1 Introduction

So far we have discussed the different phases of the compiler, each performing some specific task. The main objective of any compiler is to generate a target code that corresponds to a given source code in terms of task execution, correctness, and meaningfulness. These objectives are achieved with the support of a special data structure called the symbol table. It is a data structure that stores information about the name, type, and scope of the variables for performing the tasks defined in all the phases of a compiler.

The symbol table is created during the lexical analysis phase and is maintained till the last phase is completed (Table 9.1). It is referred to in every phase of the compiler for some purpose or the other.

Table 9.1 Symbol table

S.No.	Phase	Usage
1.	Lexical	Creates new entries for each new identifiers
2.	Syntax	Adds information regarding attributes like type, scope, and dimension, line of reference, and line of use.
3.	Semantic	Uses the available information to check for semantics
4.	Intermediate	Helps to add temporary variables information.
5.	Optimization	Helps in machine-dependent optimization by considering address and aliased variables information.
6.	Code Generation	Generates the code by using the address information of the identifiers.

It is clear that every time the compiler finds a new identifier in the source code during lexical analysis phase, it needs to check if this identifier is already in the table, and if not it needs to store it there. Every insertion operation is always preceded with search operation during lexical phase. During other phases it searches in the symbol table to access the attribute information of the entries.

The basic two operations that are often performed with the symbol table are insert—to add new entries—and lookup—to find existing entries. The performance of the table depends on how these elements are arranged in the table. The different mechanisms that govern the performance of the table are linear list, hierarchical list, and hash-based lists. These mechanisms are evaluated based on the number of insertions(n) and the number of lookup(e) operations. A linear list is the simplest to implement, but its performance is poor when *n* and *e* are large. Hierarchical list gives performance proportional to n(log(n))+e(log(n)), where n is the number of insertions and e is the number of lookup operations. Hashing schemes provide better performance with greater programming effort and space overhead.

Apart from organizing the elements, the size of the table is also an important factor. The size can be fixed when the compiler is written. The fixed size has a limitation—if chosen small, it cannot store more variables; if chosen large, a lot of space is wasted. It is important for the symbol table to be dynamic to allow the increase in size at compile time.

The entries in the symbol table are made during the lexical phase as soon as the name is seen in the declaration statement of the input program. The information on the attributes is added when available. In some cases, the attribute information is added along with entry, on the first appearance of the variable.

For example, the C declarations

```
char NAME[20];
int AGE;
char ADDRESS[30];
int PHONENO[10];
```

For block structured languages, the entries are made when the syntactic role played by the name is discovered. The attributes are entered as action corresponding to identifying a LEXEME for a token, which is an identifier in declaration statements. This action is performed for every colon encountered in the sequence of the variable list.

9.2 Symbol Table Entries

Each entry in the symbol table is associated with attributes that support the compiler in different phases. These attributes may not be important for all compilers, but each should be considered for a particular implementation.

- Name
- Size
- Dimension
- Type
- Line of declaration
- Line of usage
- Address

There is a distinction between the token id for an identifier or name, the lexeme consisting of the character string forming the name, and the attributes of the name. Lexeme is used mainly to distinguish the attributes of one variable from the other categorized as the same token type. During the lexical analysis, when the lexeme that corresponds to an identifier is found, it first looks up in the symbol table and if it does not appear then the entry is made. A common representation of a name is a pointer to a symbol table entry for it.

All these attributes are not of a fixed size. Their space requirement in symbol table is not always constant. For instance, the size of the variable's name is language dependent. If there is an upper limit on the length, then the characters in the name can be stored in the symbol table entry as shown in Figure 9.1.

Attributes

Name	Type	Size	Dim	Line of decle	Line of usage	address
N A M E	Char	20	1	…	….	….
A G E	Int	1	0	…	….	….
A D D R E S S	Char	30	1	…	….	….
P H O N E N O	Int	20	1	…	….	….
………	…	….	….	…	….	….

Figure 9.1 Symbol table

In fixed size space within a record.

In language like BASIC and FORTRAN, the name is of a limited size of two or six. In such languages it is better if the name is stored in the symbol table.

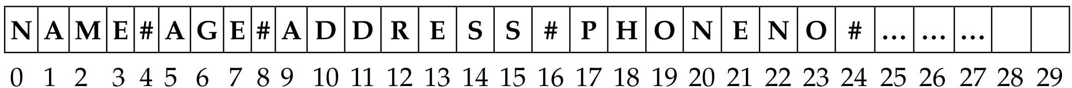
If the limit on the length is not fixed, then it is not feasible to fix the size in the symbol table. The solution in such a case is to use the indirect scheme. Instead of storing the variable name, it is good to store the address of the location where the variable is stored. The indirect scheme permits the size of the name field of the symbol table entry itself to remain a constant.

The complete lexeme constituting a name must be stored in a separate location to ensure that all uses of the same name can be associated with the symbol table record. The following table shows the entries in the symbol table for the above example, which is with fixed size space requirement.

In languages like C, C++, Java, etc., the variable name can vary from one character to 31 characters. In such cases the indirect scheme provides a better way of storing the information.

Name		Type	Size	Dim	Line of decle	Line of usage	address
0	4	Char	20	1	…	….	….
5	3	Int	1	0	…	….	….
9	8	Char	30	1	…	….	….
17	7	Int	20	1	…	….	….
		…	….	….	…	….	….

Symbol Table



N	A	M	E	#	A	G	E	#	A	D	D	R	E	S	S	#	P	H	O	N	E	N	O	#	…	…	…		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Figure 9.2 Array to store all name attributes

The name attribute in the symbol table has two values, the starting address of the name and the size of the name. In this approach, good space is saved but the disadvantage is when the compiler has to look for the attribute information. It first looks in the symbol table for the first name location; then it checks if that is the required variable; if that is not the required variable, it has to again make the symbol table reference until it gets the required variable information. A lot of time is wasted in searching for the required information.

9.3 Operations on the Symbol Table

The operations performed on the symbol table are dependent on whether the language is block structured or non-block structured. In case of non-block structured languages, there is only one instance of the variable declaration and its scope is throughout the program. In block structured languages, the variables may be re-declared and its scope is within the block. To handle the variable information, some more operations are required to keep track of the scope of the variable.

For non-block structured languages the operations are:

- Insert
- Lookup

For block-structured languages the operations are:

- Insert
- Lookup
- Set
- Reset

Insert (s,t) performs the insertion of string s and token t and returns the index of this new entry.

Lookup(s) finds for string S, if found, it returns the index, to get attributes; otherwise, it returns 0.

Set operation is performed on every block entry to mark the beginning of a new scope of variables declared in that block. Every insert and look up operation depends on this information entered.

Reset is performed at every block exit, to remove all declarations inside this scope and the details of the block.

Set and reset operations are performed in block structured languages to keep the scope information of the variables. Since, scope behave like stacks the best way to implement these functions is to use a stack. When we begin a new scope, we push a special marker to the stack to indicate the beginning of the block. The list of new declarations is inserted after this special marker is verified. At the exit of block the scope the element ends and hence all the elements inserted are popped off the stack.

9.4 Symbol Table Organization

We can use any data structure for the symbol table. The performance of any compiler depends on the organization of the symbol table and the type of declaration supported. Some programming languages support implicit declaration. In such languages, the variable can be declared or can be directly used without declaration. For example, in FORTRAN, the variables can be used without declaration. It considers the variable as int if it starts with I, J, K, L, M, or N; otherwise, it is real. In some languages, all declarations are done at the start of the program and are used later. All insert operations are performed first followed by lookup operations in case of explicit languages. In implicit languages, the insert and lookup operations may be performed at any time. The various ways a symbol table can be stored are as follows and each has its own advantages and disadvantages.

- Linear list
  - Array
    - Ordered
    - Unordered
  - Linked list
    - Ordered
    - Unordered

  - Hierarchical
    - Binary search tree
    - Height balanced tree
- Hash table.

9.5 Non-block Structured Language

9.5.1 Linear List in Non-block Structured Language

It is the simplest form of organizing the symbol table to add or retrieve attributes. The list can be ordered or unordered. Consider the following example in Ada language shown in Figure 9.3.

```
void Number(int Mike, int Alice, int John_Smith, float
  F=1.0)
{
printf( "Enter value of Mike \n" );
scanf("%d",&Mike);
printf( "Enter value of Alice\n");
scanf("%d",&Alice);
John_Smith = 3*Mike+2*Alice+2;
printf("%d\n",3*Mike+2*Alice+11);
printf("%d\n",John_Smith);
John_Smith=Mike + Alice+1000000;
printf("A million more than Mike and Alice %d\n",
John_Smith);
F=F+float(Mike)+3.1415265;
printf("F as an Integer %d\n".F);
}
```

Figure 9.3 Program in Ada

In the above example, the variables are Mike, Alice, and John_Smith declared as integers and F declared as float, and value assigned as 1.0.

9.5.1.1 Ordered List

In an ordered symbol table, the entries in the table are lexically ordered on the variable names. Every insertion operation must be preceded with a lookup operation to find the position of the new entry. Two search techniques can be applied, that is, linear or binary search. The following Figure 9.4 shows the order in which these variables are inserted.

Name	Type	Size	Value	....
Mike	Integer	1	…	….

(a) After inserting Mike

Name	Type	Size	Value	....
Alice	Integer	1	…	….
Mike	Integer	1	…	….

(b) After inserting Alice

Name	Type	Size	Value	....
Alice	Integer	1	…	….
John_ Smith	Integer	1	…	….
Mike	Integer	1	…	….

(c) After inserting John_smith

Name	Type	Size	Value	....
Alice	Integer	1	…	….
F	Float	1	1.0	….
John_ Smith	Integer	1	…	….
Mike	Integer	1	…	….

(d) After inserting F

Figure 9.4 Insert operation in ordered list

Performance:

Insert operation has the overhead of moving the elements to find the place to insert the new element. On the average, the lookup time would be (n+1)/2 if linear search is used. This can be improved if binary search is used and it would be proportional to (log n).

9.5.1.2 Unordered List

As shown in Figure 9.5, it is easy to insert the elements in an unordered list, since it inserts at the end. The look up time increases as it has to search the entire list to fetch the information of variables.

Name	Type	Size	Value	....
Mike	Integer	1	…	….

(a) After inserting Mike

Name	Type	Size	Value	....
Mike	Integer	1	…	….
Alice	Integer	1	…	….

(b) After inserting Alice

Name	Type	Size	Value	....
Mike	Integer	1	…	….
Alice	Integer	1	…	….
John_ Smith	Integer	1	…	….

(c) After inserting John_smith

Name	Type	Size	Value	....
Mike	Integer	1	…	….
Alice	Integer	1	…	….
John_ Smith	Integer	1	…	….
F	Float	1	1.0	….

(d) After inserting F

Figure 9.5 unordered list – Insert

Performance:

If the language supports explicit declaration then there is no need to perform a lookup operation for every insertion. But to avoid duplication, complete table checkup may be needed after all insertions. The lookup operation requires, on an average, a search length of $(n + 1)/2$ assuming there are n records in the table. This value is derived based on the position of the element while comparisons are made as follows:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}(n + 1) * (n)/2) = (n + 1)/2$$

If the element is not found, it indicates an error condition and the error handler should handle it.

If variables are declared implicitly, attributes are added according to the order in which they are encountered during compilation. Every insert operation must be preceded by a look up operation. If the lookup operation returns 0, it indicates the absence of a variable and it must be inserted. In worst case scenario, it requires n elements to be checked and then inserted.

If only lookup operation is performed, then on an average it requires $(n + 1)/2$ comparisons. Let the ratio of first variable reference to total variable references be denoted by x; then the lookup and insertion process requires on an average

$$x * n + (1 - x) * (n + 1) / 2$$

This unordered table organization should be used only if the number of variables are less or the table size is small, since the average time for lookup and insertion is directly proportional to the table size.

These tables are not suitable where the insert and look up operations are always applied. They are good to be used to store the reserved words in a language. The main drawback with arrays is the overhead and fixed size. This is overcome by the linked list.

9.5.2 Linked List or Self-organizing Tables

9.5.2.1 Ordered list

The variables information is inserted as shown in Figure 9.6 in the sequence encountered. The node requires an extra field to store the pointer to the next node and the last node in the list has NULL.

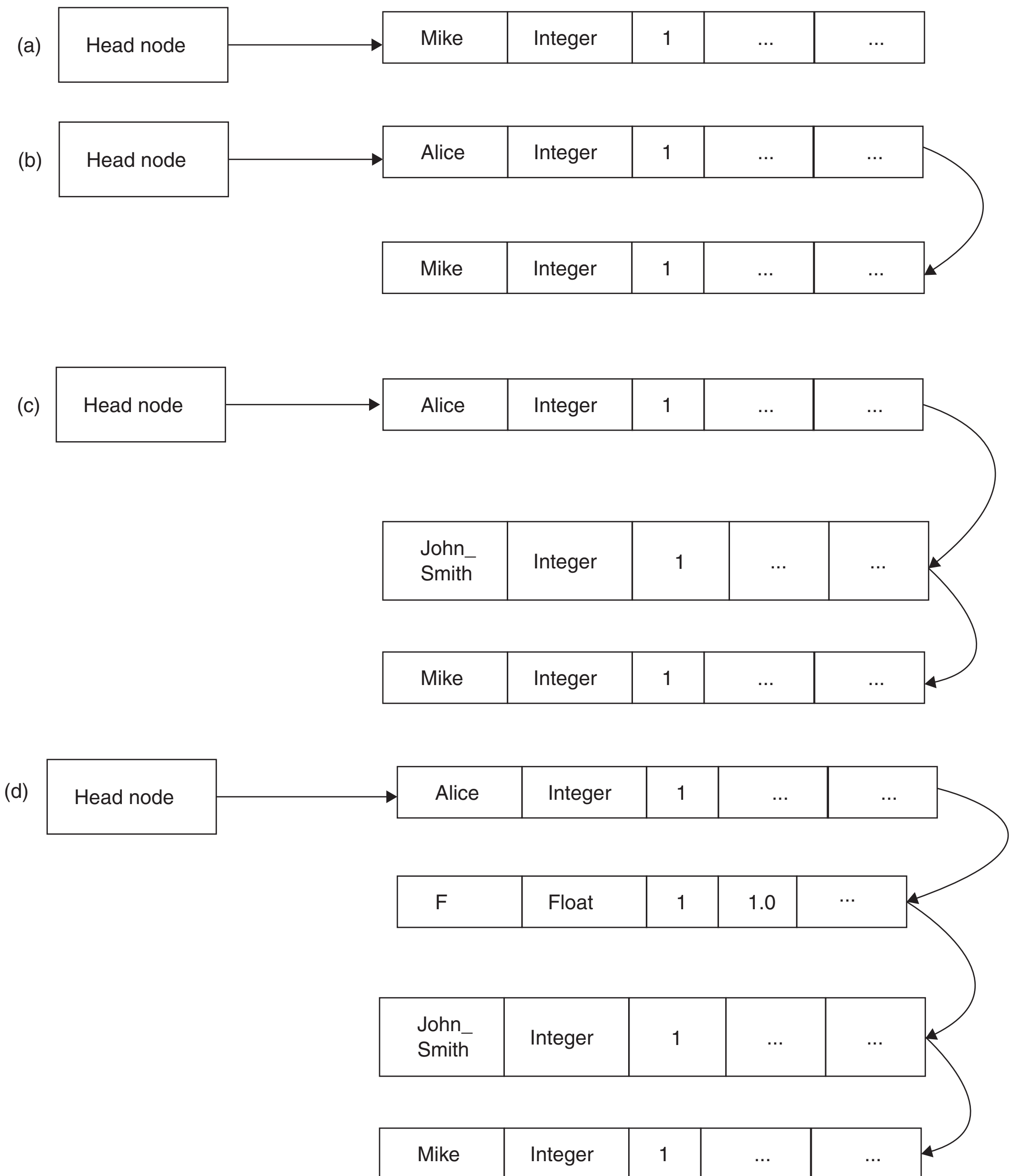


Figure 9.6 Insertion sequence

For each insertion operation, it should check if the element is there in the list, if not present, then it creates a new node and places in the order changing only two link pointers. This makes the insertion to overcome the overhead of moving the elements. The time to insert the element is O(n) + O(1). The lookup operation is always sequential with worst case O(n).

9.5.2.2 Unordered Linked List

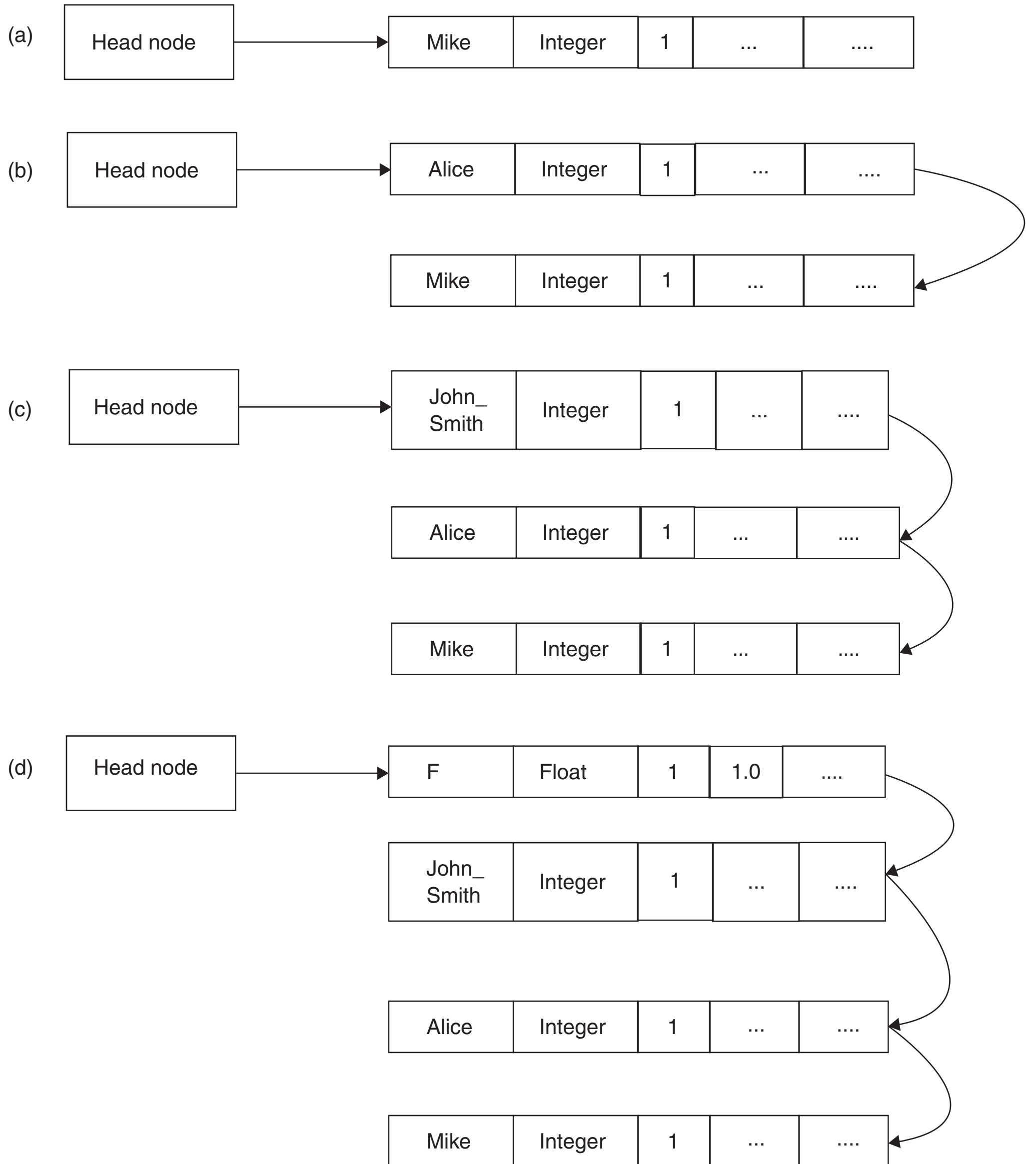


Figure 9.7 Insertion in unordered list

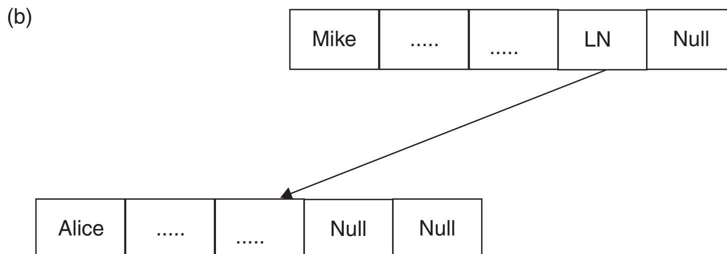
In case of an unordered list, the time for insertion is 0(1) as the insertion is done at head node as shown in Figure 9.7. Each lookup operation always has the worst case time since it has to search the entire list.

9.5.3 Hierarchical List

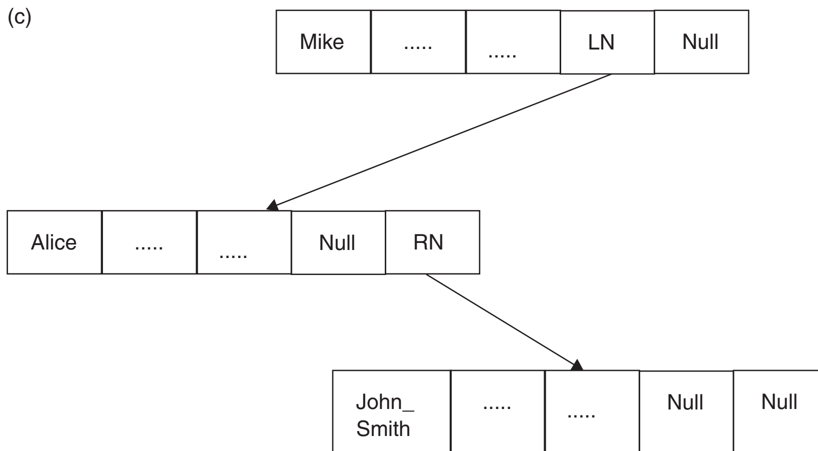
Binary search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record, and these links point to the record in the search tree. Whenever a new name is found decision is made either to add it or ignore it. First the name is searched in the tree if it exists, then it is ignored. If it does not exist, then a new record is created for the new name and is inserted as a leaf node. This organization follows a lexicographical order, that is, all the names accessible from $name_i$ with value less than $name_i$ are found by following a left link. Similarly, all the names accessible from $name_i$ that follow $name_i$ in alphabetical order are found by following the right link. The expected time needed to enter n names and to make m queries is proportional to $(m + n) \log_2 n$. So for a greater number of records (higher n), this method has advantages over linear list organization.
For the example program the tree structure is shown below in Figure 9.8.

1. On inserting first variable Mike.

2. On inserting the second variable Alice.



3. On inserting third element John_smith.



4. On inserting the last element F the tree is as shown below.

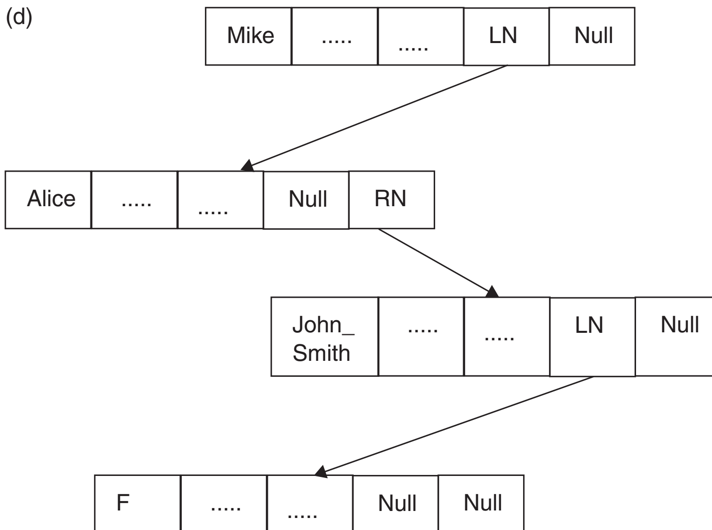


Figure 9.8 Insertion in hierarchical list

From the above example, it is clear that the tree structure may not always give better performance. If the tree formed is a skewed tree (each level has only one node), then the time complexity is again dependent on the number of elements in the program. This can be overcome by using height balanced trees like AVL trees or 2 – 3 trees.

9.5.3.1 AVL Trees

An AVL tree is similar to a binary search tree but involves balancing operations after insertions and deletions when it leaves the tree unbalanced. These operations are called rotations, which help to restore the height balance of the sub-trees.

Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. The only difference lies in the time taken to execute the lookup operation. It is proportional to the height of the tree which is O(*log n*). The time for search is maintained as O(log *n*). The efficiency can be further increased by adding index number to each node and the elements are retrieved based on index.

The parent or child node can be explored in amortized constant time after a node is searched in the tree. The traversing requires maintaining at most $2 \times \log(n)$ links. If it is required to explore n nodes it may require at most approximately $2 \times (n - 1)/n$ links, which is just 2.

Insertion

Every node in an AVL tree is associated with a balance factor *bf*. This *bf* is the difference of height of left and right sub-trees.

$$bf = H_l - H_r$$

The permissible value can be −1, 0, or +1 if the value is o then the node is balanced. If the value is 1, then we say the node is left heavy as the left sub-tree has height one more than the right sub-tree. If it is –1, then we say it is right heavy as the height of the right sub-tree is one more than the left sub-tree. However, if the balance factor becomes ±2, then the sub-tree rooted at this node is unbalanced and rotation is applied to balance the sub-tree.

After inserting a node, the balance factor is adjusted for all the nodes that lie in the path from the point of insertion and the root. If there is any node whose value is ±2, then depending on the condition, the balancing rules are applied to balance it and are shown in Figure 9.9 to 9.12.

The following figures explain how rotations can rebalance the tree, proceeding toward the root and updating the balance factor of the nodes that lie in the path. There are four types of rotations where two are symmetric to the other two in opposite directions.

Right–Right (RR)

This case occurs when X is the root sub-tree with Y as the right child of X. Let Z be the right child of Y. Let the height if X_L, Y_L, Z_L, Z_R be H. Then the *bf* Z is 0, Y is –1 and for X is –2. Hence, node X is critical. Since the node is critical to the right and the right child is right heavy, we apply Right–Right rotation (single). The resultant tree has the node Y as root of the tree and X as left child and Z as right child. The sub-trees left to Y are adjusted as shown in the Figure 9.9.

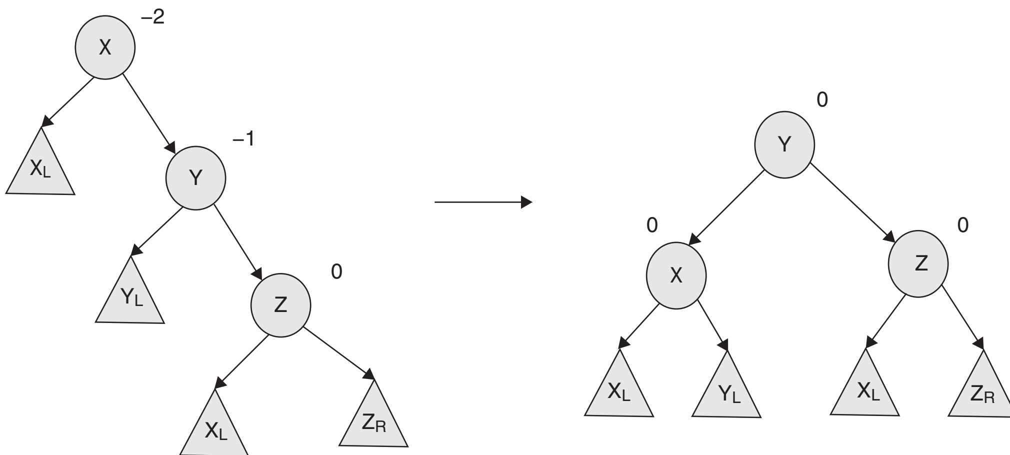


Figure 9.9

Right–Left (RL)

This case occurs when X is the root sub-tree with Z as the right child of X. Let Y is the left child of Z. Let the height if X_L, Y_L, Y_R, Z_R be H. then the *bf* Y is 0, Z is 1 and for X is -2. Hence node X is critical. Since the node is critical to the right and the right child is left heavy, we apply Right–Left rotation (double). The resultant tree has the node Y as the root of the tree and X as the left child and Z as the right child. The sub-trees left to Y are adjusted as shown in the Figure 9.10.

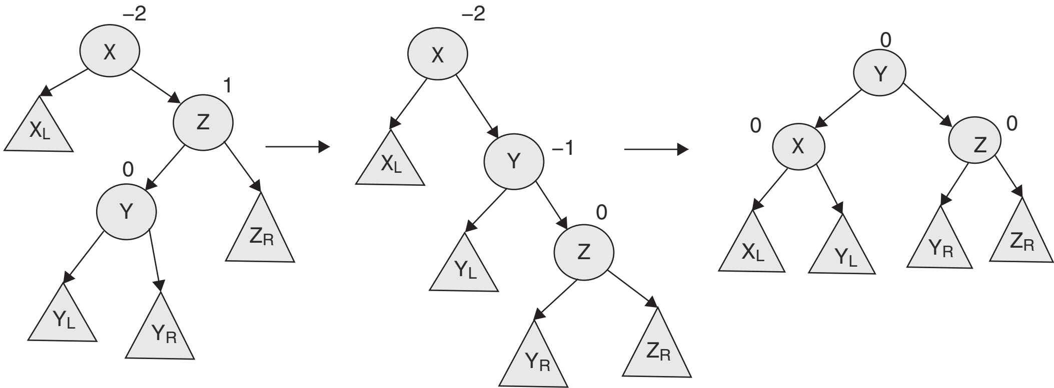


Figure 9.10

Left–Left (LL)

This case occurs when Z is the root sub-tree with Y as the left child of Z. Let X be the left child of Y. Let the height of X_L, X_R, Y_R, Z_R be H. Then the *bf* of X is 0, Y is 1, and for Z is 2. Hence, node Z is critical. Since the node is critical to the left and the left child is left heavy, we apply Left–Left rotation(single). The resultant tree has the node Y as the root of the tree and X as left child and Z as right child. The sub-trees right to Y are adjusted as shown in the Figure 9.11.

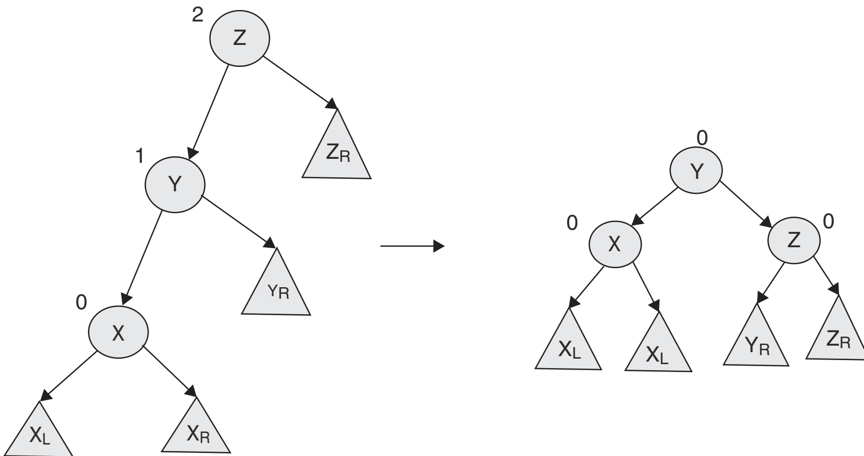


Figure 9.11

Left–Right (LR)

This case occurs when Z is the root sub-tree with X as the left child of Z. Let Y be the right child of X. Let the height of X_L, Y_L, Y_R, Z_R be H. Then the *bf* of Y is 0, X is –1 and for Z is 2. Hence node Z is critical. Since the node is critical to the left and the left child is right heavy, we apply Left–Right rotation (double). The resultant tree has the node Y as the root of the tree and X as the left child and Z as the right child. The sub-trees left to Y are adjusted as shown in the Figure 9.12

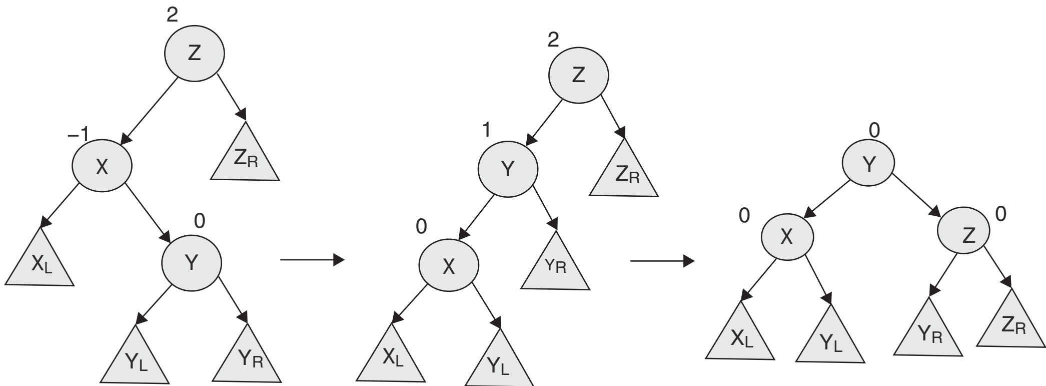


Figure 9.12

Deletion

Deletion of the node is similar to the binary tree. After deletion, the balance factor of the nodes is adjusted till it encounters the root node. If the balance factor for the tree is +2 / –2 and that of right/left sub-tree is 0, then right/left rotation is performed at the root of that sub-tree.

While retracing, if the balance factor of any node has a value between –2 and +2, based upon the balance factor, do any one of the following.

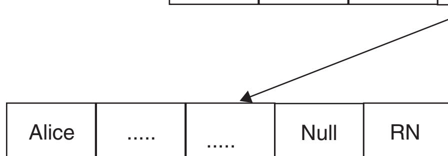
- If the balance factor is either –1 or +1, then the tree remains unchanged so stop adjusting.
- If the balance factor is 0, it indicates that the height of sub-tree is decreased by one, hence, it continues to retrace towards root.
- If balance factor is either –2 or +2 it indicates that the node is critical; hence, rotation is applied. If the balance factor of any node is 0, then retrace toward the root.

The time required for the deletion operation is O(log n) as the time required for lookup and adjusting nodes backwards id O(log n) + O(log n).

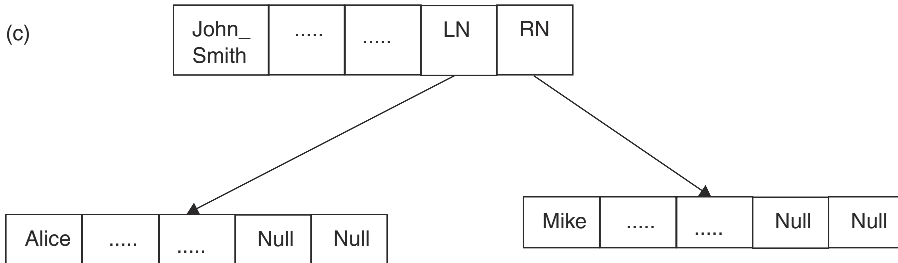
For the previous example, the AVL tree would be constructed as follows.

1. On inserting first variable Mike.

2. On inserting the second variable Alice.



3. On inserting third element John_smith it requires to apply rotation to balance the tree structure.



4. On inserting the last element F the tree is as shown below in Figure 9.13.

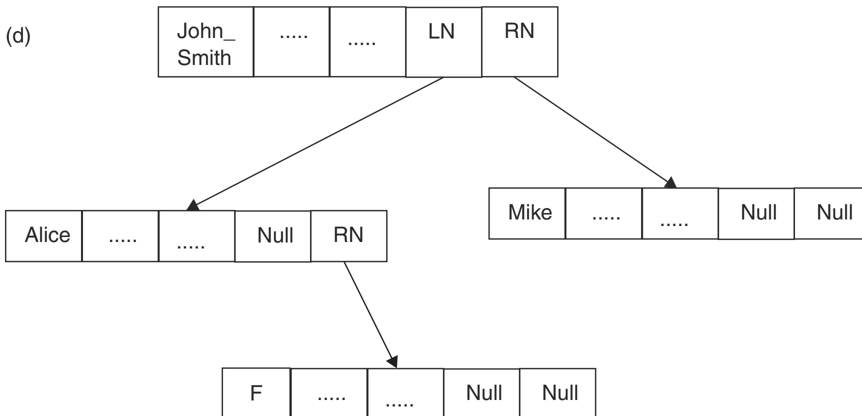


Figure 9.13

9.5.4 Hash Tables

The data structures discussed so far has the time complexity that is expressed as a function of *n*. As the value of n becomes large, the time requirement also increases. There is a special structure where the operation time is independent of *n*.

Let there be m locations and n elements whose keys are unique. If m ≥ n, then each element is stored in the table T[m], so that the hash function applied on the key K results in the location T_i. If the location T_i is empty then the element is inserted; otherwise, if it contains an element it applies the second strategy to insert element. When searching for a key element K, in location T_i it would return the element if found, otherwise returns NULL. Sample table is shown in Figure 9.14.

To use the hash table technique, it requires the keys to be unique. Also the range of keys must be bounded with the addresses of the locations.

Note: If keys are not unique, then there are various mechanisms that can be adopted. A simple method is to construct a set of m lists that store the heads of these lists in the direct address table. If the elements in the collection have at least one distinguishing feature other than the key, then the search for the specific element in that collection depends on the maximum number of duplicates.

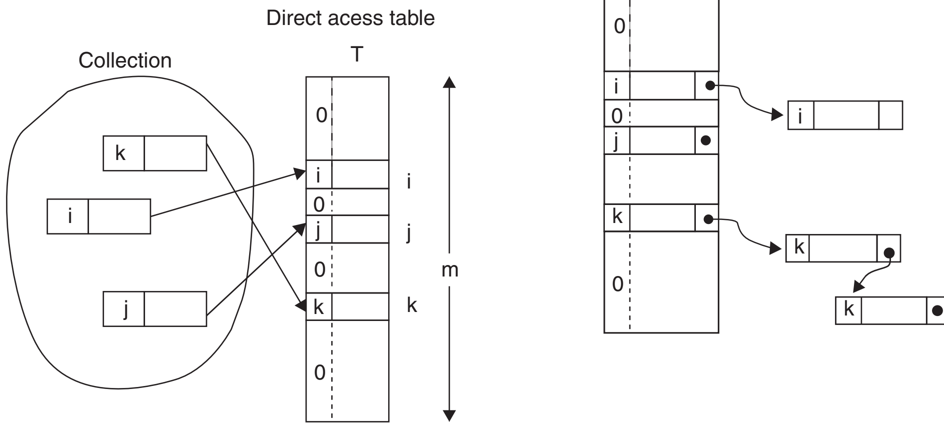


Figure 9.14

Mapping Function

The direct address approach requires a hash function, h(K) to map the key K to one of the location that is in the range (1,m) where 1,m indicates the range of memory address. It is said to be a perfect mapping function if it is one-to-one as it results in search time that is O(1).

It is easy to define such a perfect hash function theoretically but practically it is always not possible. For example, consider (1,m) to be (0,100). To map elements with keys 12, 112, 512, if the hash function is K%100, then all the keys are mapped to the 12th location. When more than one key is mapped to the same location, we call the condition, as collusion. When there are collusions, then more than one element has to be stored in the same location; this is not possible. In such a condition, we apply collusion-resolving techniques.

Handling the Collisions

Techniques have to be applied for resolving collusions for easy insertion and search. The following is the list of collusion-resolving techniques.

1. chaining,
2. re-hashing,
   a. linear probing(using neighboring slots),
   b. quadratic probing,
   c. random probing.
3. overflow areas

Chaining

It is the simplest technique to chain all the collusions in the list attached to the appropriate slot. This method doesn't require a priori knowledge of how many elements are contained

in the collection. This would exhibit poor performance if all the elements are mapped to the same location as it would create a linked list whose time is again proportional to O(n).

Re-Hashing

This technique uses a second hash function when there is a collusion. This is repeated until it finds an empty space in the table. In general, the second function could be the same function with varying parameters constituting a new function. This technique requires applying the same hash function in the same order for searching of insertion. It also involves overhead in finding the elements and requires more hash functions.

a. Linear probing

Simple rehash function that can be chosen is +1 of –1, that is, looking in the consecutive locations until a free slot is found as shown in Figure 9.15. It is easy to implement.

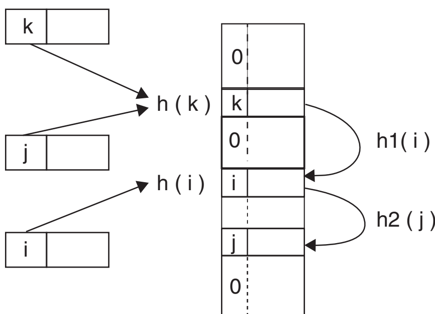


Figure 9.15

b. Random probing

This technique uses a random function to find the next location to insert the element. If the location has an element, then the random function is applied until a free slot is found. This ensures the proper utilization of the complete table as the random function generates the index which ranges uniformly. The problem is, the time taken by the random function.

c. Quadratic probing

In quadratic probing when a collusion occurs, secondary hash function is used to map the key to address. The advantage of this technique is that the address obtained by secondary function is distributed quadratically.

Address = h(key) + c i^2 on i^{th} re-hash.

(A more complex function of *i* may also be used to have better performance.)

This re-hashing scheme uses the space within the allocated table avoiding the overhead of maintaining the linked list. To apply the approach it is required to know the items that are to be stored in advance. At the same time it adds up a drawback of creating collusions for other valid keys as the table space is pre-occupied by the elements that caused the collusion earlier.

Overflow area

The table is divided into two sections in this technique. One is the primary area to which the keys are mapped and overflow area to take care of collusions as shown in Figure 9.16.

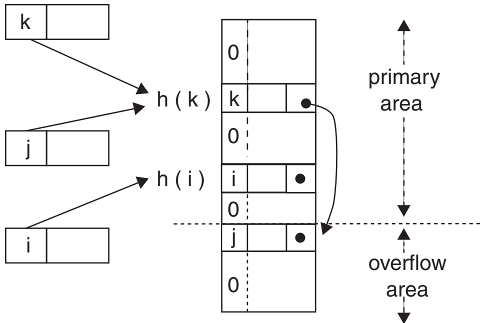


Figure 9.16

Whenever collusion occurs, the space in overflow area is used for the new entry. This location is linked to the primary table space. This appears to be similar to chaining but there is slight difference in the allocation of table space. Here the extra space is allocated along the actual table; hence, it provides faster access. It is essential to know the size of elements before allocation of space for table

It is possible to design the system with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area which provides flexibility without losing the advantage of the overflow scheme.

The following table gives the summary of the hash table organization:

Organization	Advantages	Disadvantages
Chaining	Unlimited number of elements Unlimited number of collisions	Overhead of multiple linked lists
Re-hashing	Fast re-hashing Fast access through use of main table space	Maximum number of elements must be known Multiple collisions may become probable
Overflow area	Fast access Collisions don't use primary table space	Two parameters which govern performance need to be estimated

Example: Let the variables names be

Mike, Alice: Integer;
John_Smith: Integer;
F: Float := 1.0;

Let the hash function be chosen as the sum of ASCII representation of each alphabet and let the table size be 10. We use linear probing to overcome the collusion.

Solution:
The result of hash function on each variable is shown in the table below.

Variable Name	Sum of ASCII values	Total value(TV)	Hash function (TV)/10	Mapping location
Mike	77+105+107+101	390	390/10	0
Alice	65+108+105+99+101	478	478/10	8
John_Smith	74+111+104+110+95+83 +109+105+116+104	1011	1011/10	1
F	70	70	70/10	0

The insertions are done as shown in Figure 9.17.

Insert Mike

0	Mike
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert Alice

0	Mike
1	
2	
3	
4	
5	
6	
7	
8	Alice
9	

(a) Insert John_Smith

0	Mike
1	John_Smith
2	
3	
4	
5	
6	
7	
8	Alice
9	

(b)

(c) Insert F

0	Mike
1	John_Smith
2	F
3	
4	
5	
6	
7	
8	Alice
9	

(d)

Figure 9.17

9.6 Block Structured Language

Block structured languages comprise a class of high-level languages in which a program is made up of *blocks*, which may include *nested blocks* as components, such nesting being

repeated to any depth. A block is a group of statements that are preceded by declarations of variables that are visible throughout the block and the nested blocks. These declarations are invisible if the inner blocks have the same variables declared. Once the scope of the inner block is completed, the variables of the outer block become effective. Variables are said to have *nested scopes*. The concept of block structure was first introduced in the Algol family of languages. The symbol table organization in block structured languages is complex, compared to the structured languages. It requires the additional information to be stored at every block entry and exit. Let us consider the following example shown in Figure 9.18.

This program contains four blocks. *B1* is main that has two inner blocks ***B2*** and ***B3***. Block ***B3*** has another inner block ***B4***. On execution first the ***main*** is called, which invokes the function ***fun1***; ***fun1*** in turn calls ***fun2,*** which includes ***B4***. On completion of ***B4,*** it executes the remaining part of ***fun2()***. On completion ***fun2()*** it returns to next statement of ***Call fun2()*** in ***fun1()***. On completion of ***fun1()*** it executes the next statement of ***call fun1()*** in ***main***.

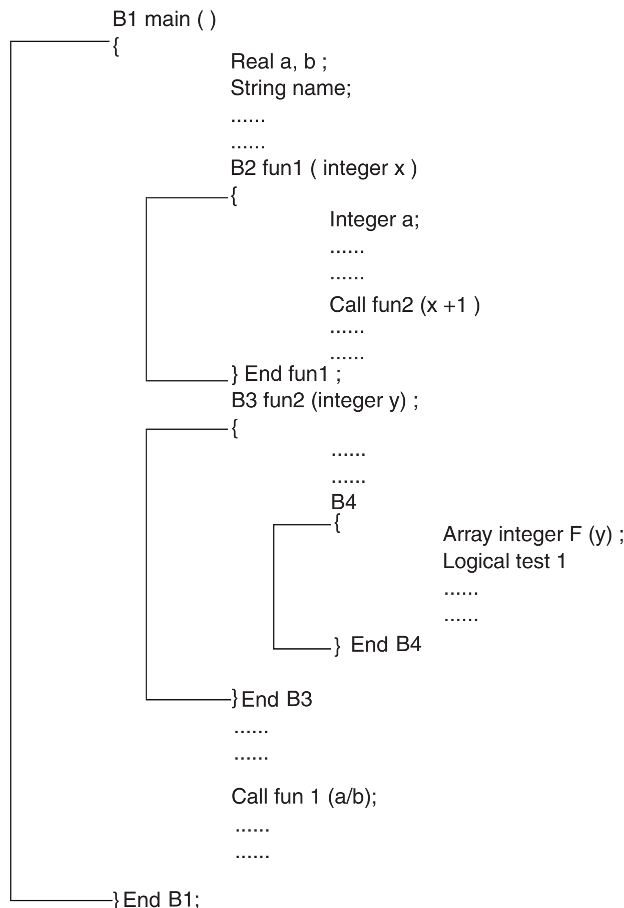


Figure 9.18

During execution, these two blocks behave differently, but during compilation both types of blocks require similar types of processing. During the compilation, at the block entry, a sub-table should be created for new variables using the set operation. For the variables declared with the same name in both inner block and outer block, care should be taken, so that the variables of the outer block are inactive and the variables in the inner block are active. At the block exit, these entries should be deleted using the reset operation. The following is the trace of compilation with set and reset operations at every block.

B1 entry: Set operation is performed and no variable is either active or inactive.
B2 entry: Set operation is performed and at this moment variables *a*, *b*, *name* and *fun1* are active. No variable is inactive.
B2 exit: Reset operation is performed. ***a***, *b*, *name*, *fun1*, *x* and ***a*** are active. No variable is inactive.
B3 entry: Set operation is performed. *a*, *b*, *name*, *fun1*and *fun2* are active. *x* and *a* are inactive.
B4 entry: Set operation is performed. *a*, *b*, *name*, *fun1*, *fun2* and *y* are active. *x* and *a* are inactive.
B4 exit: Reset operation is performed. *a*, *b*, *name*, *fun1*, *fun2*, *y*, *F* and *test1* are active. *x* and *a* are inactive.
B3 exit: Reset operation is performed. *a*, *b*, *name*, *fun1*, *fun2* and *y* are active. *x*, *a*, *F* and *test1* are inactive.
B1 exit: Reset operation is performed. *a*, *b*, *name*, *fun1* and *fun2* are active. *x*, *a*, *F*, *test1* and *y* are inactive.
End of compilation: All variables *a*, *b*, *name*, *fun1*, *fun2*, *x*, *a*, *F*, *test1* and *y* are inactive.

Note: In block B2 both the instances of ***a*** are active but the lookup operation should return the attributes of the recent instance of ***a***. Hence, the best suitable data structure that can be used is a stack.

9.6.1 Stack Symbol Tables

In this organization, records containing the attributes of the variables are stacked as they are encountered. At the block exit these records are deleted since they are not required outside the block. This organization contains two stacks—one that holds the records and the other that holds the block index details. The four operations performed on the table are explained below.

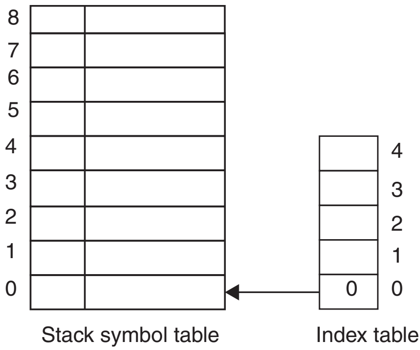
Set: This operation generates a new block index entry at the top of block index table, which corresponds to the top of the symbol stack. This entry marks the start of the variable in the new block.

Reset: This operation removes all the records corresponding to the current completed block. This corresponds to setting the top in symbol stack to the value pointed by the block index and popping the current top in block index.

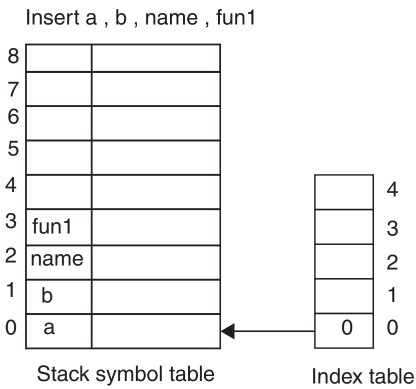
Insert: This operation is simple and involves in adding the new record on top of symbol stack. This operation requires examining that no duplicates exist in the same block.

Look up: This operation is similar to the linear search of the table from the top to bottom. It searches for the variable that is the latest declaration.

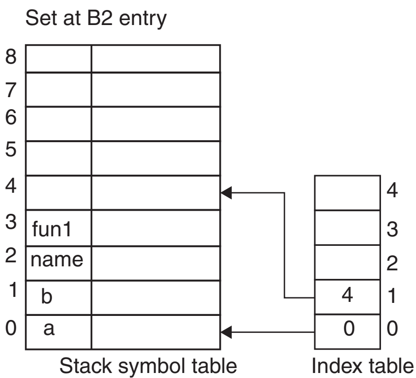
For the above example, the following Figure 9.19 shows how the variable information is added or removed after the entry/exit of every block.



(a) Set at B1 entry

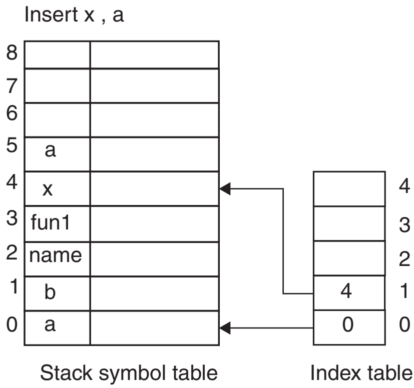


(b) Insert a, b, name ,fun1

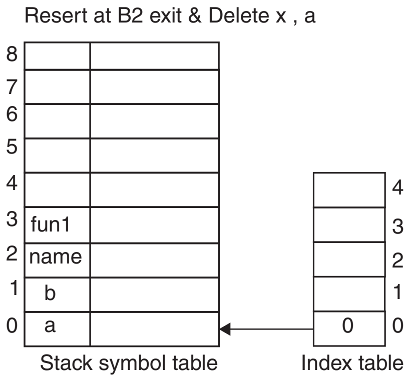


(c) Set at B2 entry

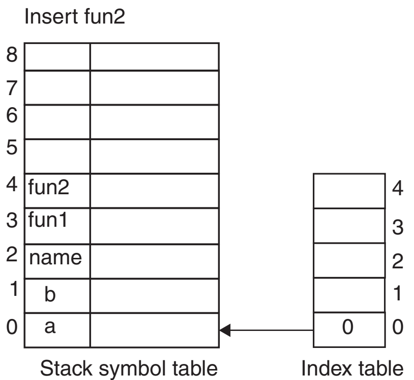
Figure 9.19



(d) Insert x , a

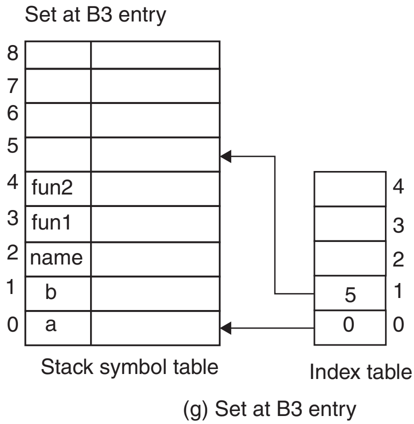


(e) Delete ex,a & Resert at B2 exi

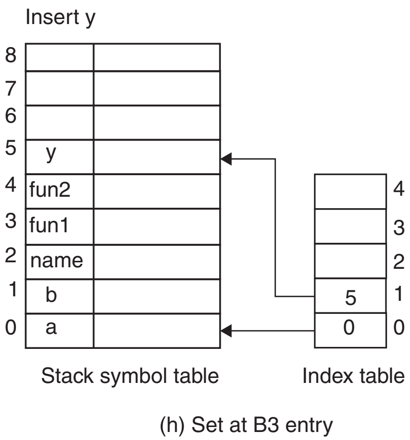


(f) Insert fun2

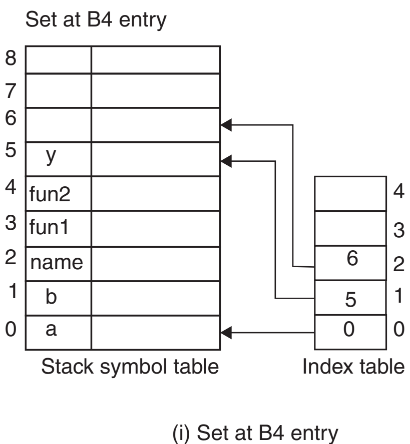
Figure 9.19



(g) Set at B3 entry

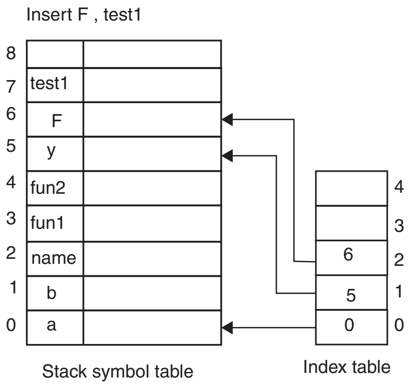


(h) Set at B3 entry

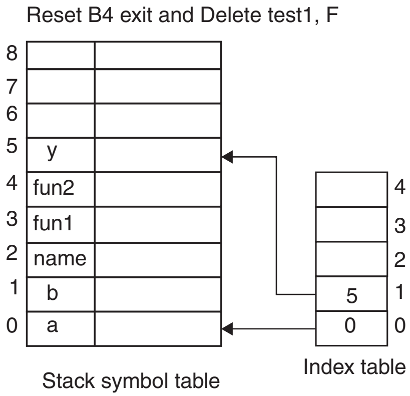


(i) Set at B4 entry

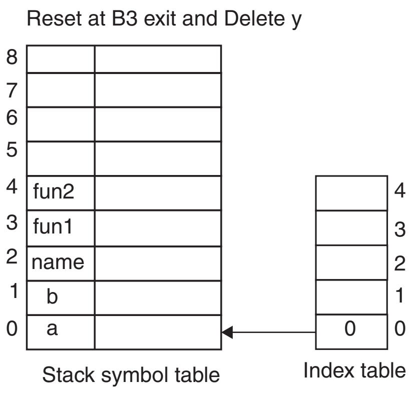
Figure 9.19



(j) Insert F and test1

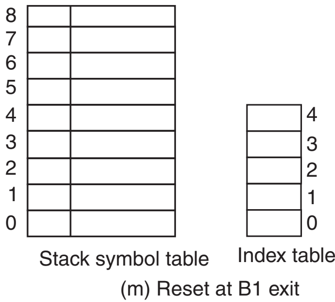


(k) Reset at B4 exit



(l) Reset at B3 exit

Figure 9.19



(m) Reset at B1 exit

Figure 9.19

9.6.2 Stack-Implemented Tree-structured Symbol Tables

Tree structured symbol table organization can be done in two different forms for block structured languages. The first approach is using a single tree for all the variables. This involves removal of records for a block, when the compilation of the block is completed. Since the records of all the blocks are merged as one tree, it requires a complex procedure to address the required records while applying the operations on the tree.

Insertions are always done as the leaf node; care should be taken while performing a lookup operation to ensure that the reference is for the current block. Every deletion operation involves the following steps.

1. Locate the position of the record in the tree.
2. Remove the record from the tree and adjust the links to bypass the node.
3. Rebalance the tree if the deletion causes the tree unbalanced.

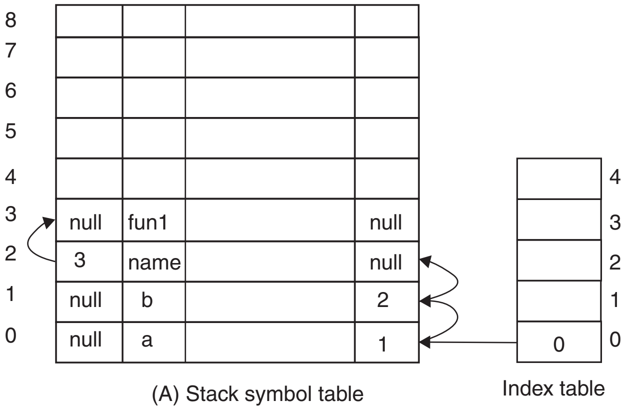
Note: Single tree structure may not be suited to compile the nested languages.

The second approach is to construct a forest of trees where each block has allocated its own tree—structured table. When the compilation of the block is complete, the entire tree for that block is deleted.

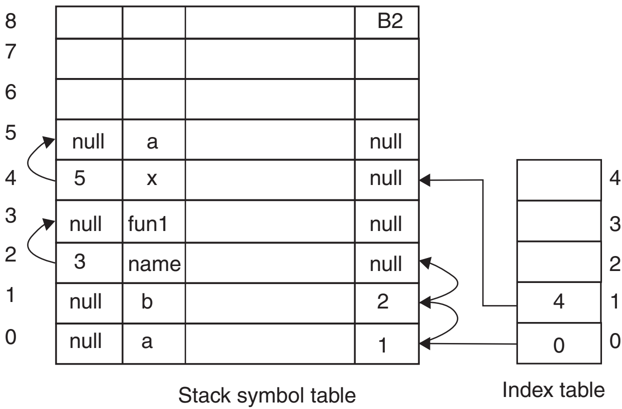
In this organization, the node for each record is associated with two special pointers along with all the attributes—one left pointer to point the left node and right pointer to point to right node. The symbol table is maintained as a stack. When the block is entered during compilation, the value of top of stack table is stored at the top of the block index table. As decelerations are encountered, records are inserted at the top of the symbol table and rebalanced if the insertions make it unbalanced.

A lookup operation must ensure that the latest occurrence is located. The search must begin at the tree structure for the last block to be entered and proceed down the block index table till it points the root of the tree for the first block entered. For example, to lookup for variable "a" in Figure 9.20B, it first starts the search at the root pointed by the top of the block index. The top points to location 4, compares with it, and searches in the left sub-tree until it is found and returns the index as 5. When the search is for b, the sub-tree pointed by the top of block index returns null; hence a search is made in the sub-tree pointed by the (top-1) and returns the location is at 1.

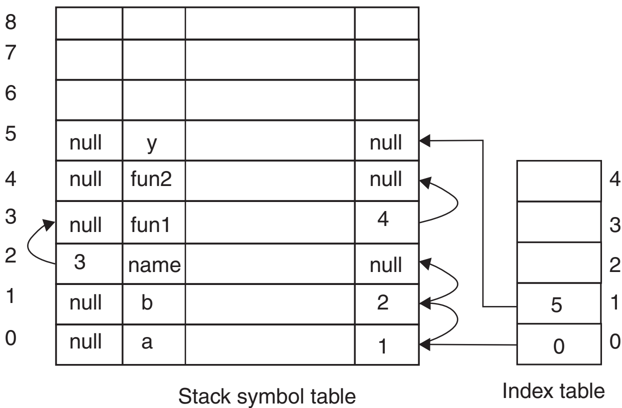
The following figure shows the operations on every entry and exit of the block.



(A) Stack symbol table

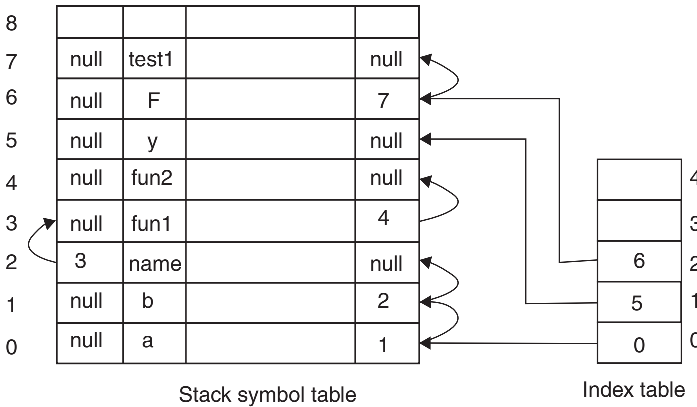


(B) Before call to fun2 in B2



(C) Just before entry at B4

Figure 9.20



(D) In the Block B4

Figure 9.20

9.6.3 Stack-Implemented Hash-Structured Symbol Table

Applying hashing technique is complex for block structured languages, as it requires some techniques to preserve the information regarding the variables of same block. This is achieved by using an intermediate hash table. The hash table stores the link to the location in the stack symbol table where the variable is stored. The stack symbol table stores the information of the variable along with the link, to the location of the variable which maps to the same location in hash table. The block index table stores the starting location of the variables on the current block. The operations performed are also complex.

Set: On every block entry, the current top of the stack symbol table is stored in block index table.

Insert: First the hash function is applied on the key

a. If it maps to a location with no collusion, then the variable information is stored in the current top of stack symbol table and that index is stored in hash table.
b. If it maps to a location with collusion, then the index stored in the hash table is stored along with the variable information in the current top of the stack symbol table and this index is updated in the hash table. This enables to store the variable information without losing the information of the variable previously inserted.

Lookup: First the hash function is applied on the key,

a. If the hash table has null, then return null.
b. If it points to the some location in the stack symbol table,
    a. If it is the required variable, return the index if the index is greater than the current top of the block index.
    b. Otherwise, use the link pointer to go to next location and perform step a.

Reset: Delete all the variables from the stack symbol table whose index is greater than or equal to the current top in block index. While deleting the variable, it requires the following modifications.

a. If the link field of the variable is null, then delete the variable and set the index in hash table to null which points it.
b. If the link field is not null, then store this index in the hash table and delete the variable information.

Example: The symbol table organization for block structured languages using hashing technique is shown below.

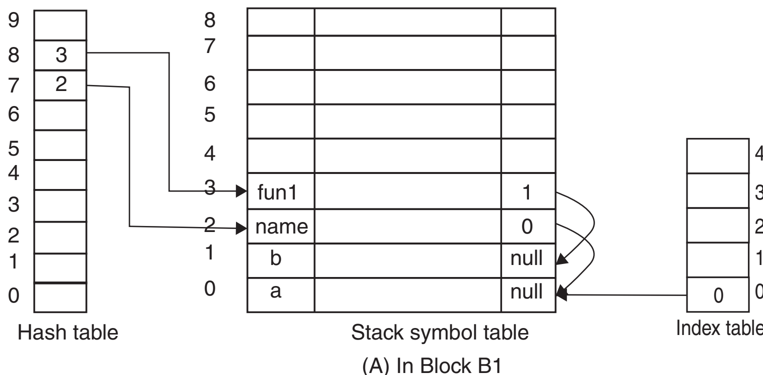
Let the hash function be chosen as the sum of ASCII representation of each alphabet and let the table size be 10. We use linear probing to overcome the collusion.

Solution:

The result of hash function on each variable is shown in the table below. The Figure 9.21 shows the content of hash table after set and insert operation in each block entry.

Variable Name	Sum of ASCII values	Total value (TV)	Mapping location (TV/10)
A	97	97	7
B	98	98	8
Name	110+97+109+101	417	7
fun1	102+117+110+49	378	8
X	120	120	0
fun2	102+117+110+50	379	9
Y	121	121	1
F	70	70	0
test1	116+101+115+116+49	497	7

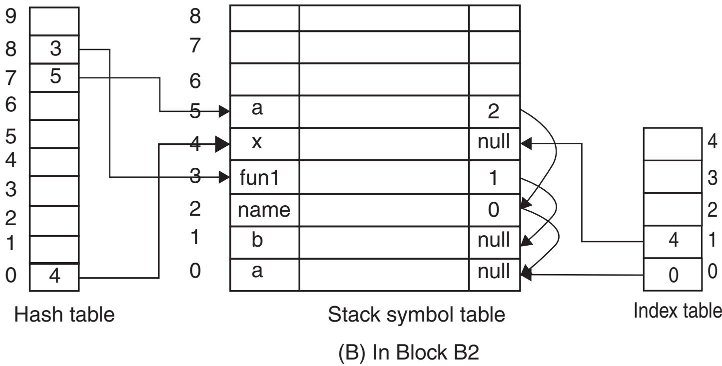
Block B1: Four variables are inserted *a*, *b*, *name*, and *fun1* where *a* and *name* map to the 7th index and *b* and *fun1* map to the 8th index in the hash table.



(A) In Block B1

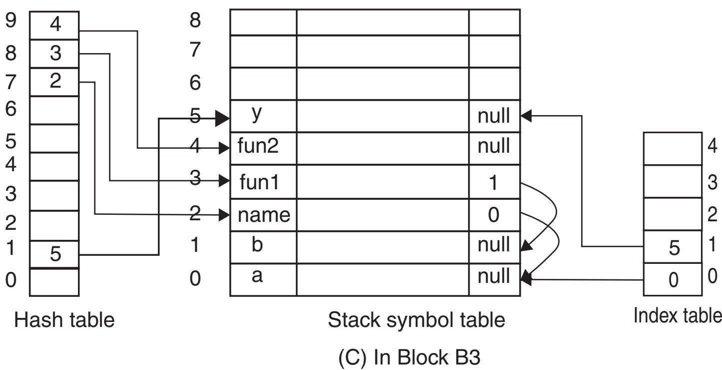
Figure 9.21

Block B2: Two variables *x* and *a* are inserted where *a* is mapped to the 7th index and *x* is mapped to the 0th index in the hash table.



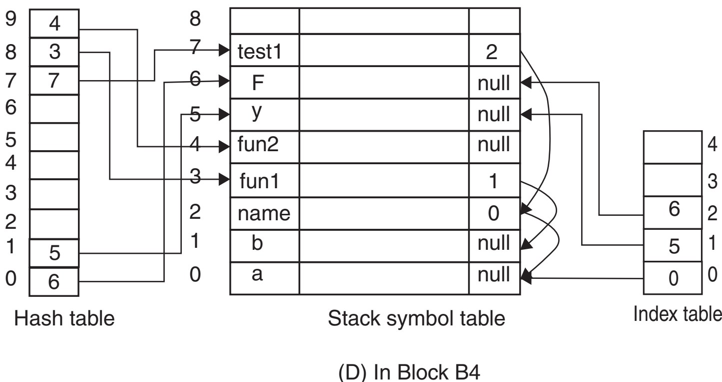
(B) In Block B2

Block B3: Two variables *fun2* and *y* are inserted where *fun2* is mapped to the 9th index and *y* is mapped to the 1st index in the hash table.



(C) In Block B3

Block B4: Two variables F and test are inserted where F is mapped to the 9th index and test1 is mapped to the 1st index in the hash table.



(D) In Block B4

Figure 9.21

Summary

- A symbol table is created during the lexical phase.
- The indirect scheme permits the size of the name field of the symbol table entry itself to remain a constant.
- In non-block structured languages the operations performed on symbol table are insert and lookup.
- In block structured languages the operations performed on symbol table are set, reset, insert, and lookup.
- Linear structures like arrays and linked list are simple to implement but performance is poor.
- A search tree is a more efficient approach to symbol table organization.
- The expected time needed to enter n names and to make m queries is proportional to $(m + n) \log_2 n$.
- Balanced trees are more suitable than binary search trees.
- Performance of hash-based technique is independent of number of elements in the list.
- Insert (s,t) performs the insertion of string s and token t and returns the index of this new entry.
- Lookup(s) finds for string S, if found returns the index, 0 otherwise.
- Set operation is performed at every block entry and it stores the top of the variable index table in the stack table.
- Reset operation is performed at every block exit. It removes the variable list information of the block it exited and the top of the stack table.

Fill in the Blanks

1. The symbol table is created during ________________ phase.
2. Hierarchical list gives performance proportional to ________________.
3. For ________________ the entries are made when the syntactic role played by the name is discovered.
4. ________________ is used mainly to distinguish the attributes of one variable with the other categorized as same token type.
5. ________________ performs the insertion of string s and token t and returns the index of this new entry.
6. ________________ finds for string S, if found returns the index, 0 otherwise.
7. ________________ is performed on every block entry to mark the beginning of new scope of variables declared in that block.
8. ________________ is performed at every block exit, to remove all declarations inside this scope and the details of the block.
9. Set and reset operations are performed in Block structured languages to keep ________________ of the variables.
10. In ________________ the variables can be used without declaration.
11. ________________ operation has the overhead of moving the elements to find the place to insert new element.

12. Balance factor *bf* of every node in an AVL tree is associated with a value _______________.
13. If the node is critical to the right and the right child is right heavy we apply _______________.
14. If the node is critical to the right and the right child is left heavy we apply _______________.
15. If the node is critical to the left and the left child is left heavy we apply _______________.
16. If the node is critical to the left and the left child is right heavy we apply _______________.
17. If the balance factor becomes _______________ then the sub-tree is said to be unbalanced.
18. The range of the key determines the size of the _______________ and may be too large to be practical.
19. When more than one key is mapped to the same location we say _______________ has occurred.
20. The concept of block structure was first introduced in the _______________ family of languages.

Objective Question Bank

1. The basic two operations that are often performed with the symbol table are ______.

   (a) set and reset
   (b) insert and lookup
   (c) set and insert
   (d) reset and lookup

2. __________ schemes provide better performance for greater programming effort and space overhead.

   (a) Arrays
   (b) Linked list
   (c) Trees
   (d) Hashing

3. In case of __________ there in only one instance of the variable declaration and its scope is throughout the program.

   (a) non-block structured languages
   (b) block structured languages
   (c) object oriented languages
   (d) All of the above

4. In __________the variables may be re-declared and its scope is within the block.

   (a) non-block structured languages.
   (b) block structured languages.
   (c) object oriented languages.
   (d) All of the above

5. __________ performs the insertion of string s and token t and returns the index of this new entry.

   (a) Lookup
   (b) Set
   (c) Insert
   (d) Reset

6. __________ finds for attributes of the variable in symbol table.

   (a) Lookup
   (b) Set
   (c) Insert
   (d) Reset

7. __________ operation marks the scope of variables in the block.

   (a) Insert
   (b) Lookup
   (c) Set
   (d) Reset

8. __________ operation removes all variable declarations inside the scope of a block on exit.

   (a) Insert
   (b) Lookup
   (c) Set
   (d) Reset

9. __________ operations are performed in block structured languages to keep scope information of the variables.

   (a) Set and reset
   (b) Insert and lookup
   (c) Set and insert
   (d) Reset and lookup

10. In ordered symbol table, the entries in the table are lexically ordered on the __________.

    (a) variable names
    (b) variable size
    (c) variable type
    (d) All.

11. __________ is applied if the node is critical to the right and the right child is right heavy.

    (a) Right - Right rotation.
    (b) Right - Left rotation
    (c) Left - Left rotation
    (d) Left - Right rotation

12. __________ is applied if the node is critical to the right and the right child is left heavy.

    (a) Right - Right rotation
    (b) Right - Left rotation
    (c) Left - Left rotation
    (d) Left - Right rotation

13. __________ is applied if the node is critical to the left and the left child is left heavy.

    (a) Right - Right rotation
    (b) Right - Left rotation
    (c) Left - Left rotation
    (d) Left - Right rotation

14. __________ is applied if the node is critical to the left and the left child is right heavy.

    (a) Right - Right rotation
    (b) Right - Left rotation
    (c) Left - Left rotation
    (d) Left - Right rotation

15. The __________ requires that the function, h(k) to map each key k to one address that is in the range (l,m).

    (a) direct address approach.
    (b) indirect address approach.
    (c) Both.
    (d) None.

16. The ________ technique uses a second hashing operation when there is a collision.

    (a) quadratic hashing.
    (b) linear hashing.
    (c) re-hashing.
    (d) double hashing.

17. The concept of block structure was first introduced in the __________ family of languages.

    (a) Fortran.
    (b) Pascal
    (c) Lisp.
    (d) Algol.

Exercises

1. a. What is the use of the symbol table in the compilation process? List out various attributes stored in the symbol table.
   b. Explain the different schemes of storing the name attribute in symbol table.

2. a. Write about the importance of symbol table.
   b. Explain the importance of each attribute stored in symbol table.

3. Explain the importance and format of storing the following attributes in symbol table.

   a. Variable name.
   b. Variable type.
   c. Size.
   d. Address of variable.

4. List the various data structures that can be used to organize a symbol table. Compare the performance.

5. a. Explain symbol table organization as arrays and linked list.
   b. Construct ordered array list for the variable in the following program.

```
int main()
{
 int a1, a2, c1, c2;
 char b1;
 float d1, d2;
    ----
    ----
}
```

6. a. Explain the hash table organization of symbol table.
   b. List the advantage of the hash table over other data structures.

7. a. Explain about block and non-block structured languages with example.
   b. List the operations in block structured language.

8. a. What is the use of the symbol table in compilation process? List out the various attributes stored in the symbol table.
   b. Explain dynamic storage allocation.

9. Explain symbol table organization using hash tables. Construct hash-based structure for symbol table for the variable in the following program.

```
int main()
{
 int a1, a2, c1, c2;
 char b1;
 float d1, d2;
    ----
    ----
}
```

10. Explain symbol table organization in trees. Construct the tree structure for symbol table for the variable in the following program.

```
int main()
{
 int a1, a2, c1, c2;
 char b1;
 float d1, d2;
   ----
   ----
}
```

11. Explain symbol table organization using hash tables. With an example show the symbol table organization for block structured language.

12. What is a symbol table? What is its use? What are the different ways of organizing data in symbol table?

13. a. Explain tree-structured symbol tables with example.
    b. What is the main criterion used in comparing different symbol table organizations?

14. What is the use of the symbol table in compilation process? List out various attributes stored in the symbol table.

15. a. Explain the importance of each attribute stored in symbol table.
    b. Compare the performance of different symbol table organization.

16. a. Construct ordered array list for the variable in the following program.

```
int main()
{
 int a1, a2, c1, c2;
 char b1;
 float d1, d2;
   ----
   ----
}
```

    b. Explain about block and non-block structured languages with example.

17. Explain the operations in block structured languages with examples.
    a. Set
    b. Reset
    c. Lookup
    d. Insert

Key for Fill in the Blanks

1. Lexical analysis.
2. n(log (n)) +e (log (n)).
3. Block structured languages.
4. Lexeme.
5. Insert.
6. Lookup.
7. Set operation.
8. Reset.
9. Scope information.
10. FORTRAN.
11. Insert.
12. −1, 0, or +1.
13. Right – Right rotation
14. Right – Left rotation.
15. Left – Left rotation.
16. Left – Right rotation.
17. −2 or +2
18. direct address table.
19. collusion
20. Algol.

Key for Objective Question Bank

1. b. 2. d. 3. a. 4. b. 5. c. 6. a. 7. c. 8. d. 9. a. 10. a.
11. a. 12. b. 13. c. 14. d. 15. a. 16. c. 17. d

CHAPTER 10

Code Optimization

Code optimization is an important phase to improve the time and space requirement of the generated target code. Given a code in intermediate form it applies optimizing techniques and reduces the code size and returns the code in intermediate form.

CHAPTER OUTLINE

10.1 Introduction
10.2 Where and How to Optimize
10.3 Procedure to Identify the Basic Blocks
10.4 Flow Graph
10.5 DAG Representation of Basic Block
10.6 Construction of DAG
10.7 Principle Source of Optimization
10.8 Function-Preserving Transformations
10.9 Loop Optimization
10.10 Global Flow Analysis
10.11 Machine-Dependent Optimization

Code optimization phase is an optional phase in the phases of a compiler, which is either before the code generation phase or after the code generation phase. This chapter focuses on the types of optimizer and the techniques available for optimizing. It also includes global flow analysis, a technique to gather the information of how data and control flows, to apply global optimization.

10.1 Introduction

It is possible for a programmer to outperform an optimizing compiler by using his or her knowledge in choosing a better algorithm and data items. This is far from practical conditions. In such an environment, there is need for an optimizing compiler. Optimization is the process of transformation of code to an efficient code. Efficiency is in terms of space requirement and time for its execution without changing the meaning of the given code. The optimization is considered an important phase because of the following practical reasons:

- Inefficient programming (which forces many invisible instructions to be performed for actual computation.) e. × **a = a + 0**
- The programming constructs for easy programming. e. × **Iterative loops**.
- Compiler generated temporary variables or instructions.

The following constraints are to be considered while applying the techniques for code improvement:

- The transformation must preserve the meaning of the program, that is, the target code should ensure semantic equivalence with source program.
- Program efficiency must be improved by a measurable amount without changing the algorithm used in the program.
- When the technique is applied on a special format, then it is worth transforming to that format only when it is beneficial enough.
- Some transformations are applied only after detailed analysis, which is time consuming. Such analysis may not be worthy if the program is run very few number of times.

The optimization can be classified depending on

- Level of code.
  - Design level—efficiency of code can be improved by making the best use of available resources and selection of suitable algorithm.
  - Source code level—the user can modify the program and change the algorithm to enhance the performance of the object code.
  - Compile level—the compiler can enhance the program by improving the loops, optimizing on the procedure calls and address calculations. This is possible when the representation is in three address code.
  - Assembly level—the compiler optimizes the code based on the machine architecture and is based on the available registers and suitable addressing modes.
- Programming language
  - Machine Independent—the code-improvement techniques that do not consider the features of the target machine are machine-independent techniques. Constant folding, dead code elimination, and constant propagation are examples of machine-independent techniques. These are applied on either high-level language or intermediate representation.
  - Machine dependent—these techniques require specific information relating to target machine. Register allocation, strength reduction, and use of machine idioms are examples of machine-dependent techniques.
- Scope
  - Local—Optimizations performed within a single basic block are termed as local optimizations. These techniques are simple to implement and does not require any: analysis since we do not require any information relating to how data and control flows.

- Global—optimization performed across basic blocks is called global optimizations. These techniques are complex as it requires additional analysis to be performed across basic blocks. This analysis is called data-flow analysis.

Optimization is the field where most compiler research is done today. High-quality optimization is more of an art than a science. Compilers for mature languages are judged based upon the quality of the object code generated and not based on how well they parse or analyze the code.

Example 1:

Consider the following example, which sets every element of an array to 1.

```
a) int array_ele[10000];
   void Binky() {
   int i;
   for (i=0; i < 10000; i++)
   array_ele[i] = 1;
   }

b) int array_ele[10000];
   void Winky() {
   register int *p;
   for (p = array_ele; p < array_ele + 10000; p++)
   *p = 1;
   }
```

In the above two examples, one may think the second one is faster than the first. It may be true if they use a compiler without optimization. Many modern compilers emit the same object code by using clever techniques like "loop-induction variable elimination."

10.2 Where and How to Optimize

Optimization techniques can be applied to intermediate code or on the final target code. It is a complex and a time-consuming task that involves multiple sub phases, sometimes applied more than once. Most compilers allow the optimization to be turned off to speed up compilation process. For example, in ***gcc*** there are specific flags that are turned on/off for individual optimization. To apply optimization it is important to do control flow analysis and data flow analysis followed by transformations as shown in Figure 10.1.

Control Flow Analysis: It determines the control structure of a program and builds a control flow graph.

Data Flow Analysis: It determines the flow of scalar values and builds data flow graphs. The solution to flow analysis propagates data flow information along a flow graph.

Transformation: Transformations help in improving the code without changing the meaning or functionality.

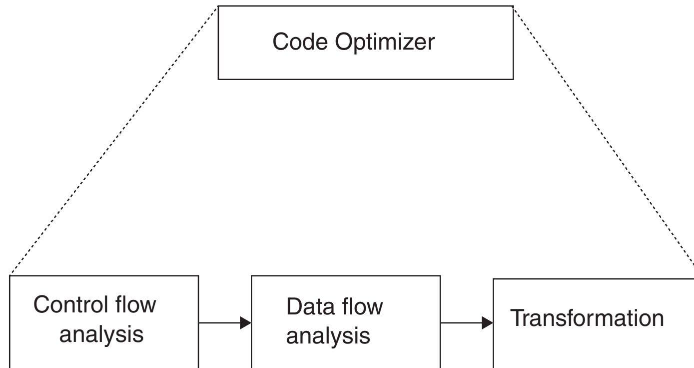


Figure 10.1 Code Optimization Model

Flow Graph

A graphical representation of three address code is called flow graph. The nodes in the flow graph represent a single basic block and the edges represent the flow of control. These flow graphs are useful in performing the control flow and data flow analysis and to apply local or global optimization and code generation.

Basic Block

A basic block is a set of consecutive statements that are executed sequentially. Once the control enters into the block then every statement in the basic block is executed one after the other before leaving the block.

Example 2: For the statement a = b + c * d/e the corresponding set of three address code is

$t_1 = c * d$
$t_2 = t_1 / e$
$t_3 = b + t_2$
$a = t_3$

All these statements correspond to a single basic block.

10.3 Procedure to Identify the Basic Blocks

Given a three address code, first identify the leader statements and group the leader statement with the statements up to the next leader. To identify the leader use the following rules:

1. First statement in the program is a leader.
2. Any statement that is the target of a conditional or unconditional statement is a leader statement.
3. Any statement that immediately follows a conditional/unconditional statement is a leader statement.

Example 3: Identify the basic blocks for the following code fragment.

```
main( )
{
 int i = 0, n = 10;
 int a[n];
 while ( i <=(n-1))
 {
a[i] = i * i;
i=i+1;
 }
 return;
}
```

The three address code for the initialize function is as follows:

(1). i: = 0
(2). n: = 10
(3). t_1: = n – 1
(4). If i > t_1 goto (12)
(5). t_2: = i * i
(6). t_3: = 4 * i
(7). t_4: = a[t_3]
(8). t_4: = t_2
(9). t_5: = i + 1
(10). i:= t_5
(11). goto (3)
(12). return

Identifying leader statements in the above three address code

Statement (1) is leader using rule 1
Statement (3) and (12) are leader using rule 2
Statement (4) and (12) are leaders using rule 3

1. i: = 0 → Leader 1
2. n: =10
3. t_1: = n – 1 → Leader 2
4. If i > t_1 go to (12)
5. t_2: = i * i → Leader 3
6. t_3: = 4 * i
7. t_4: = a[t_3]
8. t_4: = t_2
9. t_5: = i + 1
10. i: = t_5
11. go to (3)
12. Return → Leader 4

Basic block 1 includes statements (1) and (2)
Basic block 2 includes statements (3) and (4)
Basic block 3 includes statements (5)–(11)
Basic block 4 includes statement (12)
Basic blocks are shown in Figure 10.2.

10.4 Flow Graph

Flow graph shows the relation between the basic block and its preceding and its successor blocks. The block with the first statement is B1. An edge is placed from block B1 to B2, if block B2 could immediately follow B1 during execution or satisfies the following conditions.

- The last statement in B1 is either conditional or unconditional jump statement that is followed by the first statement in B2 or
- the first statement in B2 follows the last statement in B1 and is not an unconditional/conditional jump statement.

Flow graph for Figure 10.2 is shown in Figure 10.3.

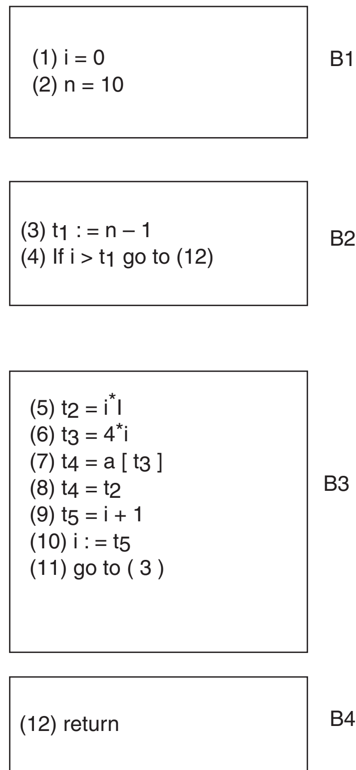


Figure 10.2 Basic Blocks

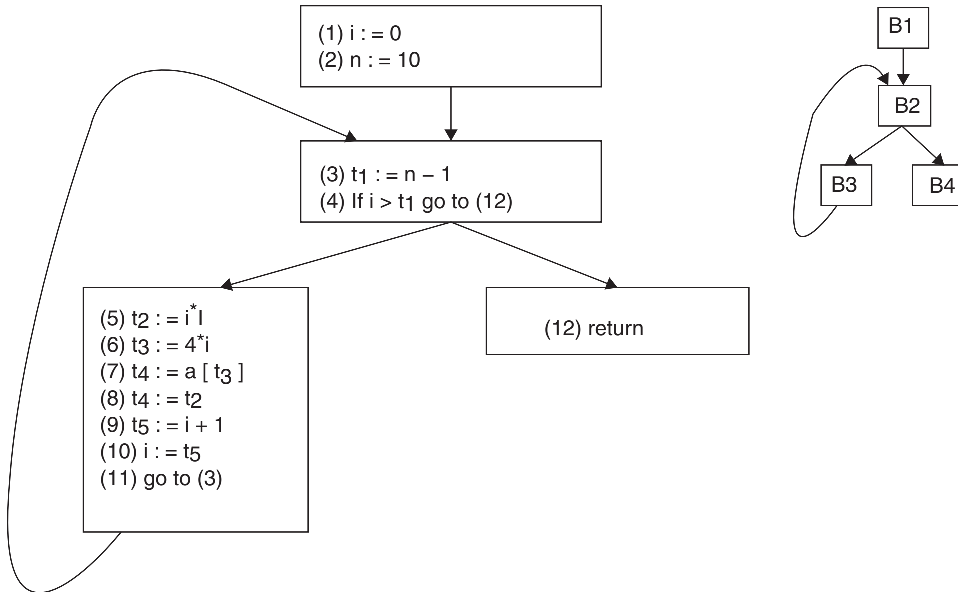


Figure 10.3 Flow Graph for Example 3

10.5 DAG Representation of Basic Block

A DAG is a useful data structure for implementing transformations within a basic block. It gives the pictorial representation of how values computed at one statement are useful in computing the values of other variables. It is useful in identifying common sub-expressions within a basic block. A DAG has nodes, which are labeled as follows:

- The leaf nodes are labeled by either identifiers or constants. If the operators are arithmetic then it always requires the r- value.
- The labels of interator nodes correspond to the operator symbol.
- Some nodes are sometimes referred to by the sequence of identifiers for labels. The interior nodes represent computed values.

Note: The DAG is not the same as a flow graph. Each node in a flow graph is a basic block, which has a set of statements that can be represented using DAG.

10.6 Construction of DAG

To construct DAG, we process each statement of the block.

If the statement is a copy statement, that is, a statement of the form $a = b$, then we do not create a new node, but append label a to the node with label b.

If the statement is of the form $a = b\ op\ c$, then we first check whether there is a node with same values as $b\ op\ c$, if so we append the label a to that node. If such node does not exist then we first check whether there exists nodes for b and c which may be leaf nodes or an internal nodes if recently computed, then create a new node for *op* and add to it the left child b and the right child as c. Label this new node with a. This would become the value of a to be used for next statements; hence, we mark the previously marked nodes with a as a_0.

DAG creation would be easy if we maintain the information of the nodes created for all the identifiers and facility to create a linked list of attached identifiers for each node.

We also define a function that returns the most recently created node associated with identifier.

10.6.1 Algorithm for Construction of DAG

Given a basic block we need to construct a DAG, which has the following information:

- A label (identifier/operator) for each node.
- For each node list of identifiers attached to it.

In the construction process, we process each statement and categorize it to one of the following cases.

i. $a = b\ op\ c$
ii. $a = op\ b$
iii. $a = b$
iv. Relational operators are treated as case (i) for example if $i \leq 20$ go to l_1 considered similar to case (i) with a undefined.

Initially we assume there are no nodes, and a node is undefined for all arguments and does the following steps:

1. For case (i)
   a. If node(b) and node(c) are undefined, create leafs labeled b and c respectively; let node(b) and node(c) be these nodes.
   b. Find if there is a node labeled op, with left child as node(b) and right child as node(c); if found return this node; otherwise, create a node and let this be n.
   c. Delete a from the list of attached identifiers for node(a). Append a to the list of attached identifiers for the node n found in (b) and set node(a) to n.
2. For case (ii)
   a. If node(b) is undefined, create a leaf labeled b; let node(b) be the node.
   b. Find if there is a node labeled op, whose lone child is node(b); if so, return this node otherwise create a node and let this be n.
   c. Delete a from the list of attached identifiers for node(a). Append a to the list of attached identifiers for the node n found in b) and set node(a) to n.
3. For case (iii)
   a. node(b) is an existing node and let it be n.

b. Delete *a* from the list of attached identifiers for node(*a*). Append *a* to the list of attached identifiers for the node *n* found in (a) and set node(*a*) to *n*.

Example 4: Let us consider Figure 10.4 and construct DAG for each block step by step.

For block B1

For the statement which satisfies case iii) we first create a leaf labeled 4 and attach identifier i to it as shown in Figure 10.5.

For block B2

For first statement, which satisfies case i) we first create nodes for n and 1, then create node for operator(–) and label it as t1. Figure 10.6(a) shows for single statement in B2.

For second statement, which is a conditional statement, we create a node for operator > and label it as 12 as when this condition is satisfied it should go to statement 12. Figure 10.6(b) shows for all statements in B2 block.

Block B3

For the first statement, which satisfies case (i), we create nodes for *i* and use as right child, then create node for operator(*) and label it as t2 as shown in Figure 10.7(a).

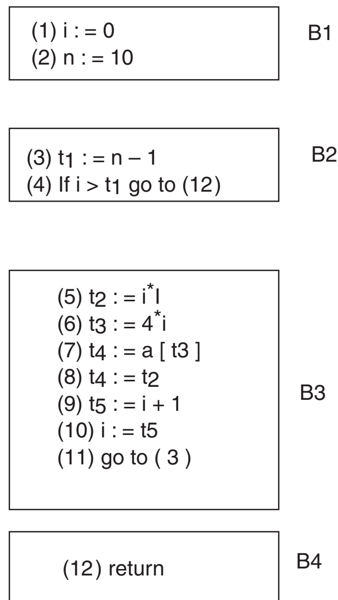


Figure 10.4 Example 3 Program in Basic Blocks

Figure 10.5 DAG for Block B1

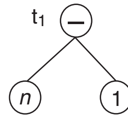


Figure 10.6(a) DAG for First Statement in Block B2

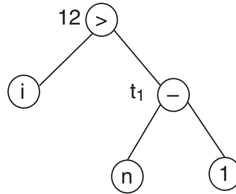


Figure 10.6(b) DAG for Block B2

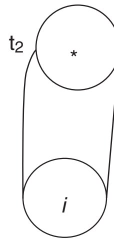


Figure 10.7(a) DAG for One Statement Block B3

For second statement, we first create nodes for 4 and use node(i), then create node for operator(*) and label it as t_3 as in Figure 10.7(b).

For third statement, we first create nodes for a and use node(t_3), then create node for operator([]) and label it as t4 as in Figure 10.7(c).

For fourth statement, since it satisfies the copy statement, we delete label t_4; mark it as t_{40} and attach t_4 label to node with label t_2 as in Figure 10.7(d).

After construction for the entire block, the DAG would be as shown in the Figure 10.7(e).

10.6.2 Application of DAG

- By looking at DAG we can determine
  - which ids have their values used in the block.
  - which statement computes values which could be used outside the block.
- DAGs are useful for redundancy elimination

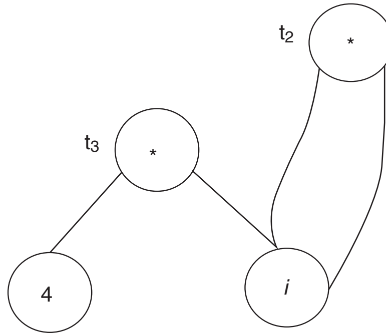


Figure 10.7(b) DAG for Two Statements in Block B3

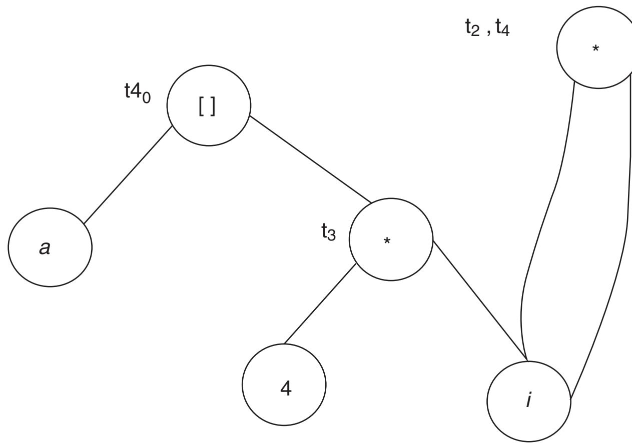


Figure 10.7(c) DAG for Three Statements in Block B3

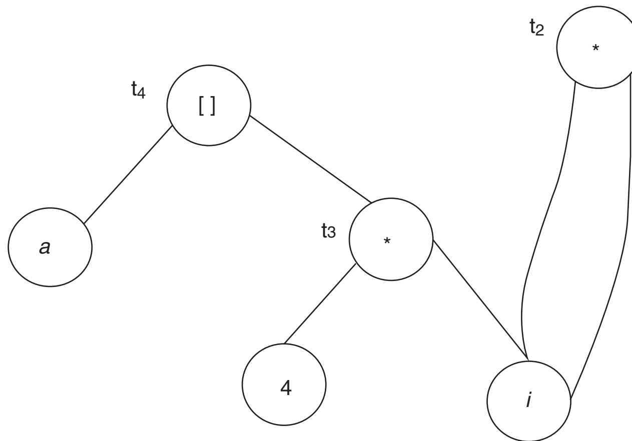


Figure 10.7(d) DAG for Four Statements in Block B3

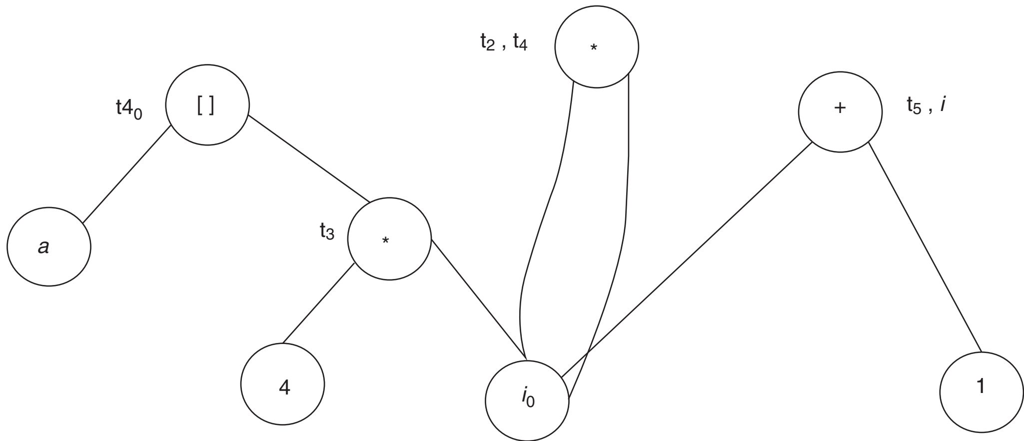


Figure 10.7(e) DAG for Complete Block B3

10.7 Principle Source of Optimization

A transformation of a program is called local if it is applied within a basic block and global if applied across basic blocks. There are different types of transformations to improve the code and these transformations depend on the kind of optimization required.

- Function-preserving transformations are those transformations that are performed without changing the function it computes. These are primarily used when global optimizations are performed.
- Structure-preserving transformations are those that are performed without changing the set of expressions computed by the block. Many of these transformations are applied locally.
- Algebraic transformations are used to simplify the computation of expression set using algebraic identities. These can replace expensive operations by cheaper ones, for instance, multiplication by 2 can be replaced by left shift.

10.8 Function-Preserving Transformations

The following techniques are function-preserving transformations, where common sub expression elimination can be applied locally or globally. The other techniques are applied globally.

10.8.1 Common Sub-expression Elimination

An expression E is said to be common sub expression if E is computed before and the variables in the expression are not modified since its computation. If such expression is present, then the re-computation can be avoided by using the result of the previous computation. This technique can be applied both locally and globally. We need to maintain a table to store

the details of expressions evaluated so far and use this information to identify and eliminate the re-computation of the same expression. The common sub-expression elimination can be done within basic block by analyzing and storing the information of expression in the table until the operands in the expression are redefined. If any operand in the expression is redefined, then remove the expression from the table. The algorithmic approach is given below.

```
Function subexpr_elimination (Block B)
{
   For each statement that evaluates an expression within basic block
   {
      Maintain in the table the set of expressions evaluated so far
      If any operand in the expression if found as redefined, then
        remove it from the table
      Modify the instructions accordingly as you go
      Generate temporary variable and store the expression in it and
        Use the latest variable definition next time the expression
        is encountered.
   }
}
```

Figure 10.8 is an example that shows the optimized code on applying this technique locally and globally.

Global common sub expression elimination is applied with the extra information collected in flow graph analysis. At every block entry and exit the collected information should be maintained to identify the evaluated expression.

Example 5: The following example in Figure 10.9 shows the common sub expression elimination globally by replacing the evaluated expression with a new temporary variable t_1 and t_2 and these variables are used wherever the same expression is used. This generates copy statements. If these copy statements are unnecessary, then they are eliminated by the technique explained in the next section.

Example 6: The bubble sort program is given below with its corresponding three address code represented in flow graph.

```
void quicksort(int LI, int HI)
{
   int i, j;
   int pivot, z ;
   if( HI ≤LI) return;
   i = LI - 1;
   j = HI
   pivot =a [HI];
   while(1)
   {
         do i = i + 1; while (a[i] < pivot);
```

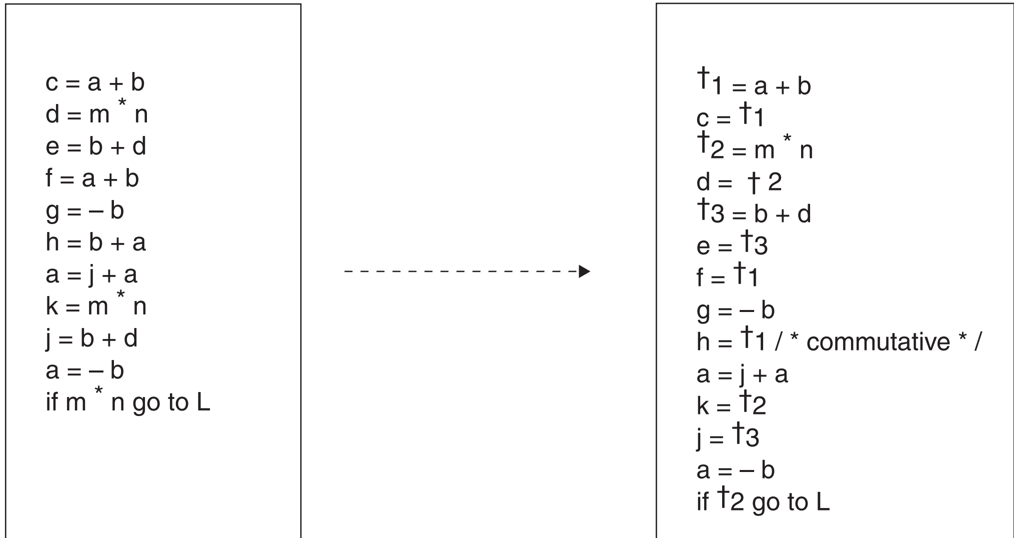


Figure 10.8 Program Code Before and After Common Sub Expression Elimination

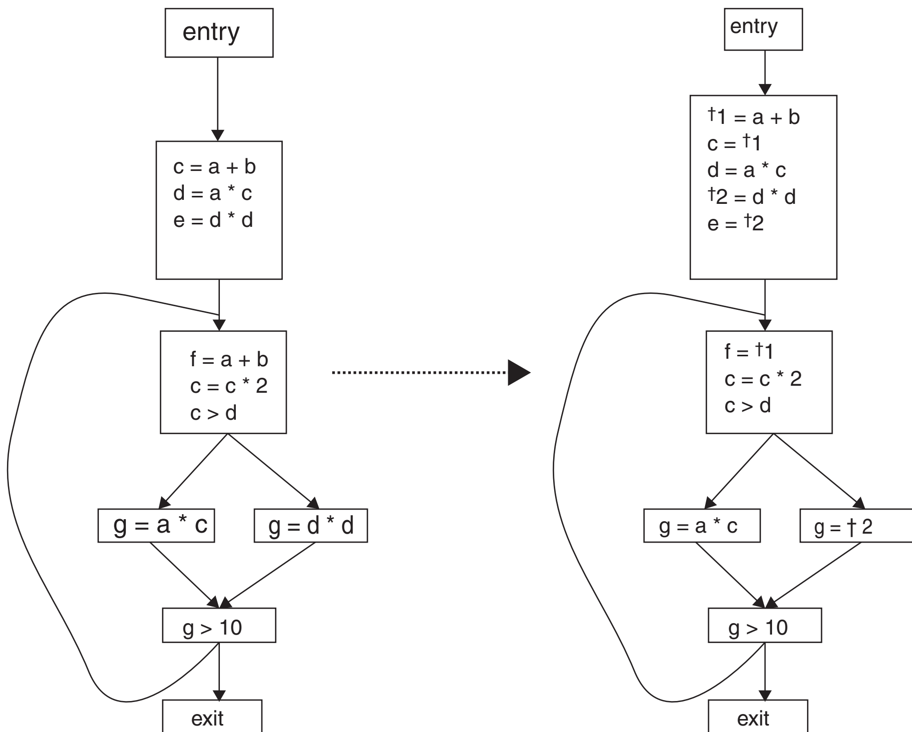


Figure 10.9 Example for Global Common Sub Expression Elimination

```
                do j = j - 1; while (a[j] > pivot);
                if( i³j) break;
                z = a[i];
                a[i] = a[j];
                a[j] = z;
           }
        z = a[i];
        a[i] = a[HI];
        a[HI] = z;
        quicksort(LI,j);
        quicksort(j+1,HI);
    }
```

Its corresponding three address code is given below and corresponding flow graph is shown in Figure 10.10.

```
 1. i: = LI - 1
 2. j: = HI
 3. temp1: = 4 * HI
 4. pivot: = a[ temp1 ]
 5. i: = i + 1
 6. temp2: = 4 * i
 7. temp3: = a[ temp2 ]
 8. if temp3 < pivot go to 5
 9. j: = j - 1
10. temp4: =4 * j
11. temp5: = a[ temp4 ]
12. if temp5 > pivot go to 9
13. if i ³ j go to 23
14. temp6: = 4 * i
15. z: = a[ temp6 ]
16. temp7: = 4 * i
17. temp8: = 4 * j
18. temp9: = a[ temp8 ]
19. a[ temp7 ]: = temp9
20. temp10: =4 * j
21. a[ temp10 ]: = z
22. go to 5
23. temp11: = 4 * i
24. z: =a[ temp11 ]
25. temp12: = 4 * i
26. temp13: = 4 * HI
27. temp14: = a[ temp13 ]
28. a[ temp12 ]: = temp14
29. temp15: = 4 * HI
30. a[ temp15 ]: = z
```

On applying local common sub expression elimination, we have block B5 and block B6 with improved code as the expressions 4 * *i* is computed more than once without change of value. Hence, on optimizing locally, we get resultant code as shown in Figure 10.11.

On applying this technique globally, the code is further improved and the resultant code is shown below in Figure 10.12. It is clear that the computation of a[i] in block B2 can be used in B5 and B6. Similarly, a[*j*] computed in B3 can be used in B5 and B6.

10.8.2 Copy Propagation

A copy statement is a statement that is in the form $a = b$. Copy propagation technique is applied to replace the later use of variable *a* with the use of *b* if original values of *a* do not

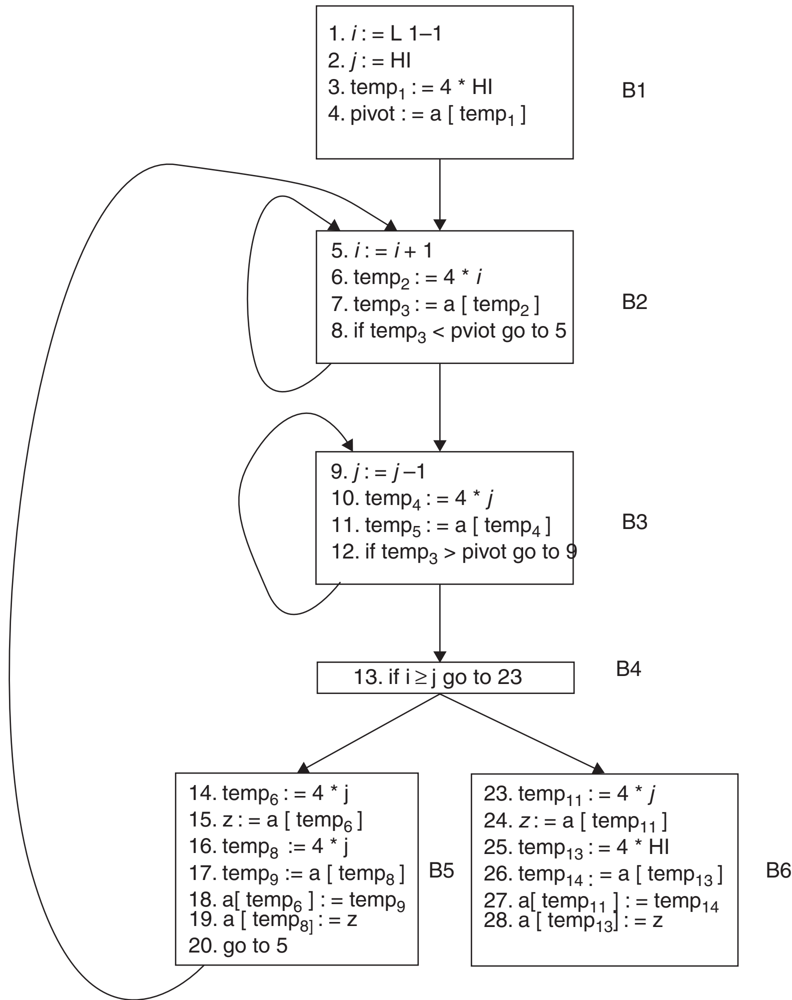


Figure 10.10 Flow Graph for Quick Sort Program

change after this assignment. This technique can be applied locally and globally. This optimization reduces runtime stack size and improves execution speed. To implement this, we need to maintain a separate table called copy table, which holds all copy statements. While traversing the basic block, if any copy statement is found, add this information into the copy table. While processing any statement, check the copy table for variable *a*, which can be replaced variable *b*. If there is an assignment statement that computes the value for *a* then remove the copy statement from the copy table. Copy propagation helps in identifying the dead code.

Let us assume that there are a set of statements $s_1, s_2, s_3 \ldots . s_n$, where the statements would be either a copy statement of the statement which uses the right hand side of copy statement or computes the value for a variable.

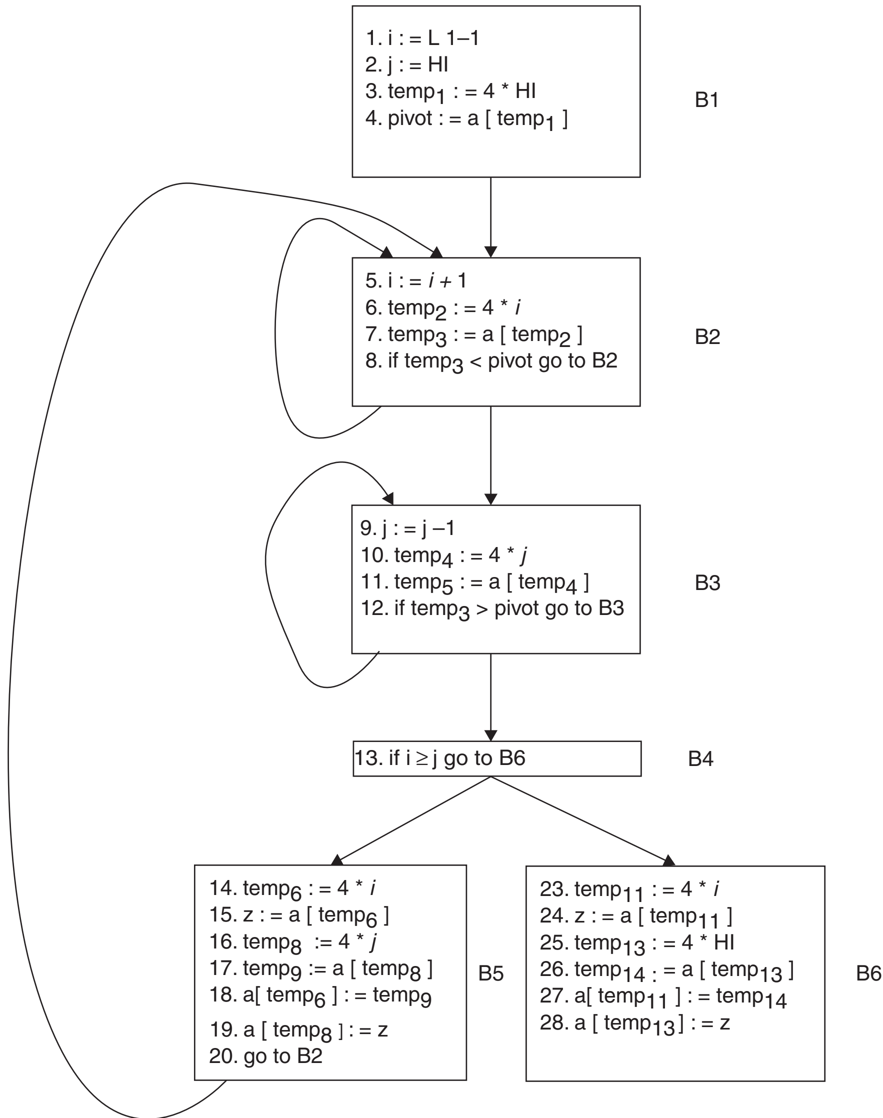


Figure 10.11 After Applying Common Sub Expression Elimination for Figure 10.10

Algorithm for copy propagation

Let INSERT (a, b) be a function that inserts {(a, b)} in a copy table, for a copy statement *a* = *b*, which indicates that *a* can be replaced by *b*.

GET (a, table) be a function that checks in the table for the variable *a*. if found returns *b*; otherwise, returns *a*.

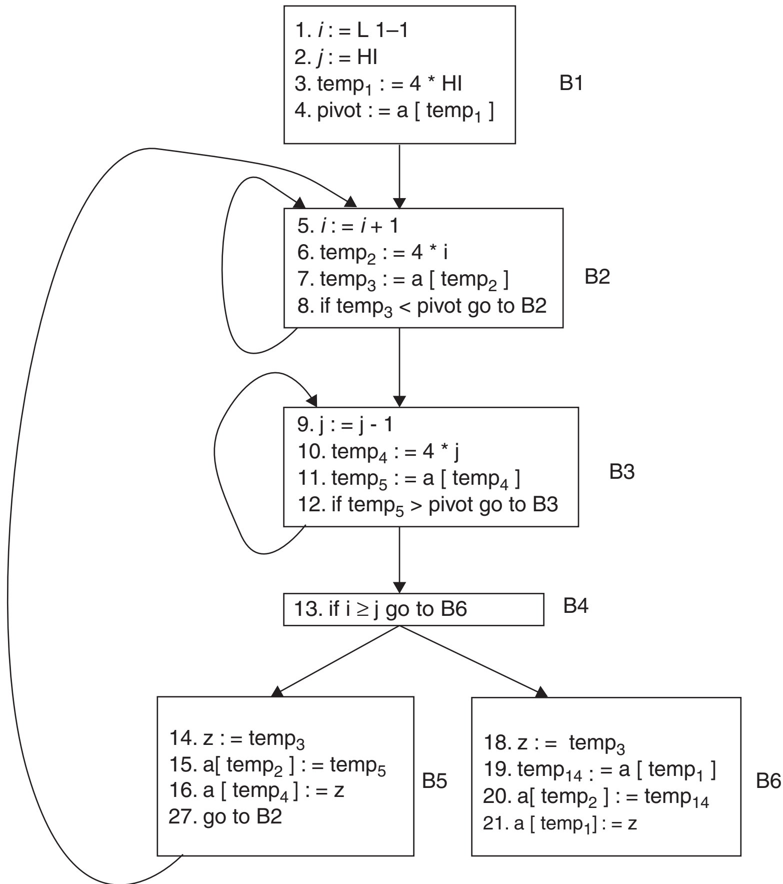


Figure 10.12 After Applying Global Common Sub Expression Elimination for Figure 10.11

```
GET(a, table)
{
   if you find (a, b) in table
         return b
   else return a
}
GET(a, table)
{
   if you find (a, b) in table
        return b
   else return a
}
```

```
OPT_CP_PROP(B)
{
 for each statement from s1 to sn
   if statement is of the form "a = b op c"
     b = GET(b, table)
     c = GET(c, table)
   else if statement is of the form "a = x"
     x = GET(x, table)
   if statement has a left hand side a,
     REMOVE from table all pairs involving a.
   if statement is of the form "a = x"
          insert {(a, GET(x, table))} in the table
 endfor
}
```

Example 7: Let the basic block contain the following set of statements:

Y = X
Z = Y + 1
W = Y
Y = W + Z
Y = W

Statement No	Instruction	Updated instruction	Copytable content
1.	Y=X	Y=X	{(Y,X)}
2.	Z=Y+1	Z=X+1	{(Y,X)}
3	W=Y	W=X	{(Y,X),(W,X)}
4	Y=W+Z	Y=X+Z	{(W,X)}
5	Y=W	Y=W	{(W,X),(Y,X)}

On the first statement 1, since it is a copy statement, the information is added in to the copy table. Statement 2 uses the value of X indirectly from Y. hence GET(Y,table) would return X and the statement is modified replacing Y with X. The third statement is a copy statement and since Y is to replace X, we insert into the copy table W to be replaced by X. When statement 4 is processed, Y value is defined; hence, details for Y are removed from the copy table. The fifth statement is a copy statement and is inserted into the copy table.

To apply the technique globally, we perform flow analysis and given a copy statement **X = Y** and a statement as **W = X** in successive blocks, we can replace **W = X** with **W = Y** only if the following conditions are met:

- **X = Y** must be the only definition of **X** reaching **W = X.** This can be determined through ud-chains.

◆ The value of Y is not redefined in any path from the statement **X** = **Y** to **W** = **X**. Use iterative data flow analysis to gather this information.

Example 8: The following example in Figure 10.13 shows the copy propagation technique applied globally across different blocks. After the copy propagation we can see that the copy statements $d = c$ and $g = e$ is a dead code as the values assigned to d and g are not used; hence they can be eliminated.

Example 9: For the resultant quicksort code in Figure 10.12, obtained after common sub expression elimination, copy propagation is applied the resultant code as is shown in Figure 10.14.

10.8.3 Dead Code Elimination

Code that is unreachable or that does not affect the program is said to be dead code. Such code requires unnecessary CPU time, which can be identified and eliminated using this technique. The following program gives an example in high-level language where, the value assigned to i is never used, and the value assigned to *var*1 variable in statement 6 is dead store and the statement 9 is unreachable. These statements can be eliminated as they do not affect the program execution.

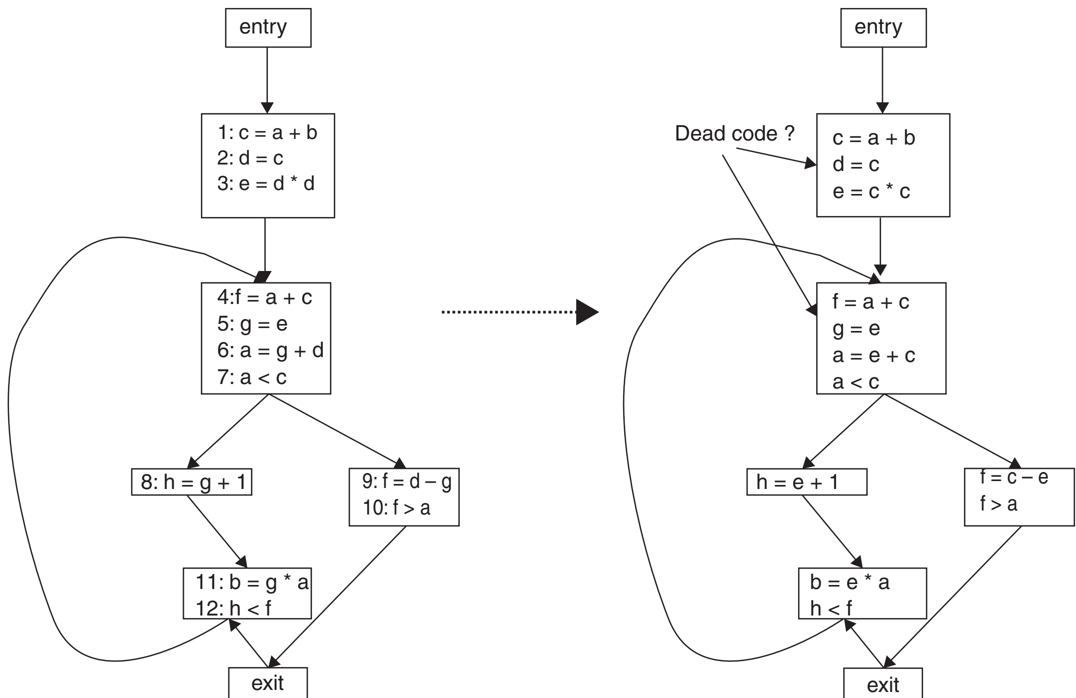


Figure 10.13 Example for Global Copy Propagation Technique

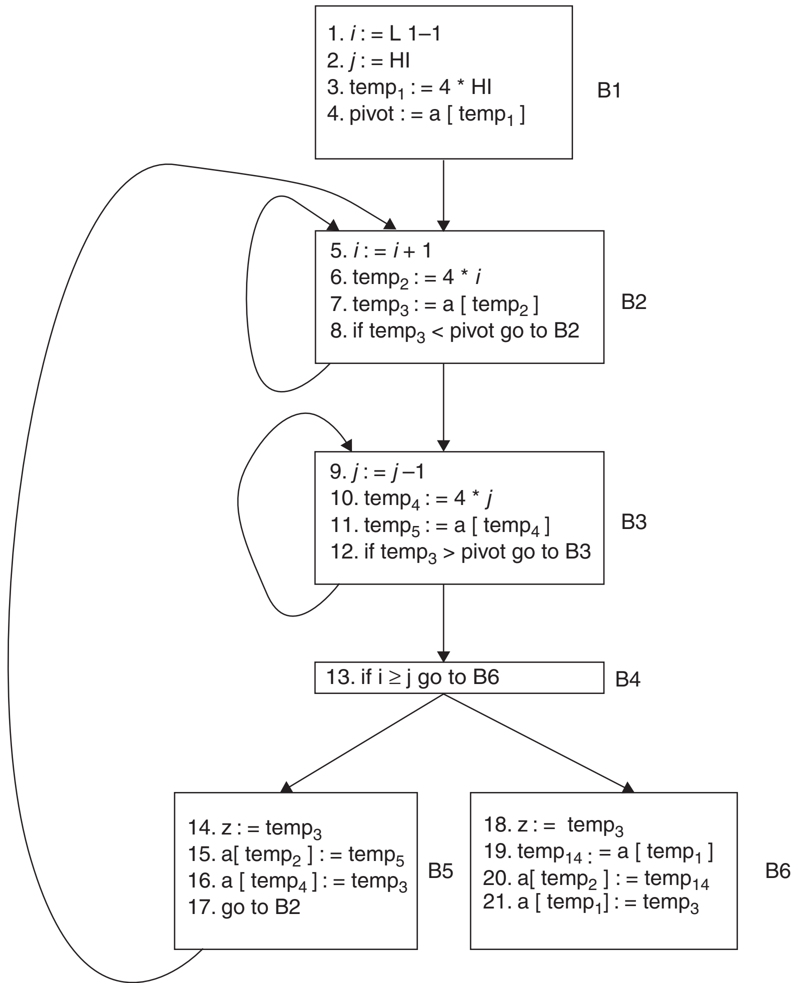


Figure 10.14 Quick Sort Program After Applying Copy Propagation

```
 1. int var1;
 2. void sample()
 3. {
 4. int i;
 5. i = 1; /* dead store */
 6. var1 = 1; /* dead store */
 7. var1 = 2;
 8. return;
 9. var1 = 3; /* unreachable */
10. }
```

Below is the code fragment after dead code elimination.

```
1. int var1;
2. void sample()
3. {
4. var1 = 2;
5. return;
6. }
```

Example 10: Blocks B5 and B6 of flowchart corresponding to quick sort there is dead store to variable Z which can be eliminated by dead code elimination. The resultant code is shown in Figure 10.15.

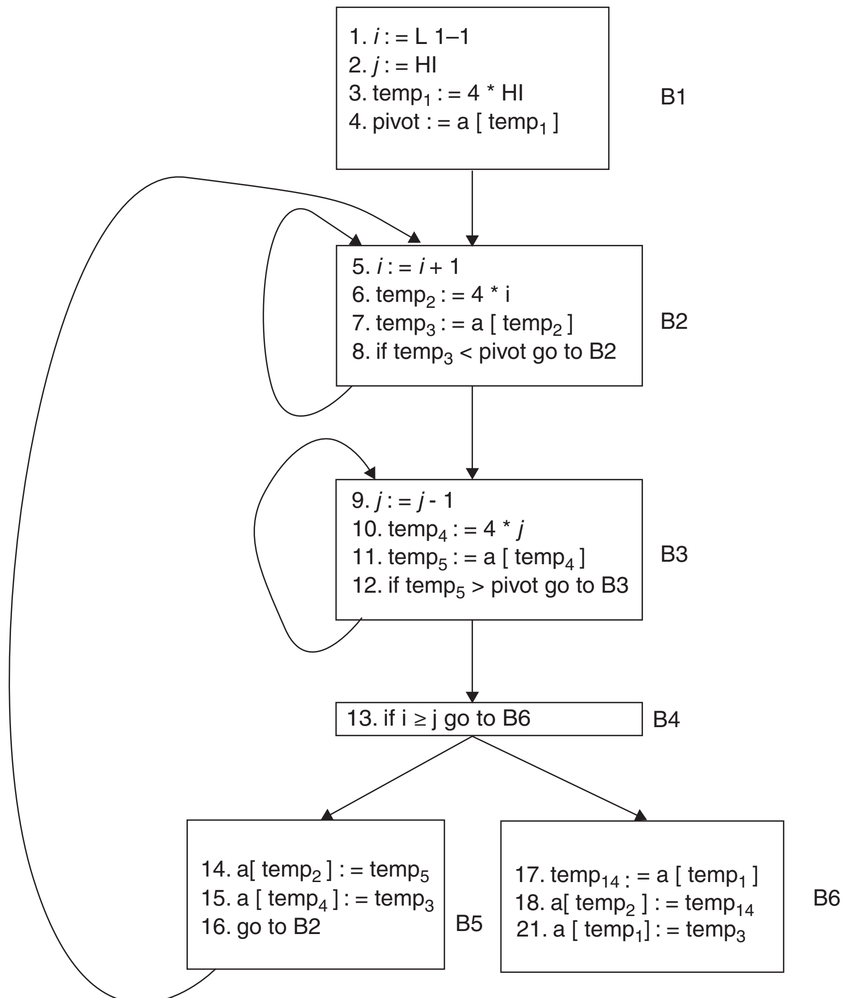


Figure 10.15 Quick Sort Program After Dead Code Elimination

10.8.4 Constant Propagation

Constant propagation is an approach that propagates the constant values assigned to a variable at the place of its use. For example, given an assignment statement $x = c$, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x. This approach is similar to copy propagation and is applied at first stage of optimization. This method can analyze by propagating constant value in conditional statement, to determine whether a branch should be executed or not, that is, identifies the dead code.

Example 11: Let us consider the following example:

```
1. pi = 22/7
2. void area_per(int r)
3. {
4.    float area, perimeter;
5.    area = pi * r * r;
6.    perimeter = 2 * pi * r;
7.    print area, perimeter;
8. }
```

In this short example, we can notice some simple constant propagation results, which are as follows:

- In line 1 the variable *pi* is constant and this value is the result of 22/7, which can be computed at compile time and has the value of 3.413.
- In line 5 the variable *pi* can be replaced with the constant value 3.413.
- In line 6 since the value of *pi* is constant, the partial result of the statement can be computed and the statement can be modified as

perimeter = 6.285 * r;

Note: In common sub expression elimination or constant propagation, often it may require to rearrange the expressions by using the associative properties of the algebraic expression.

10.9 Loop Optimization

In most of the programs, 90% of execution time is inside the loops; hence, loop optimization is the most valuable machine-independent optimization and are good candidates for improvement. Loops are for convenience of program writing, but these programs has to be transformed using loop transformation techniques. Let us consider the example of a program that has a loop statement.

```
for (int i=0; i<1000; i++)
   x = x + y/z;
print x;
```

The optimized code of this program is represented in flow graph as shown in Figure 10.16.

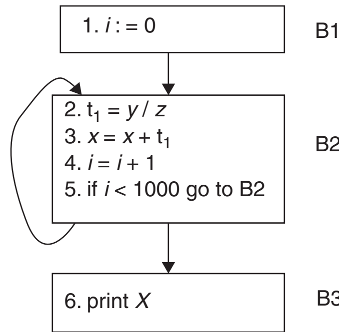


Figure 10.16 Flow Graph After Code Optimization

It is clear that for the code the total number of statements that are executed are as follows:

Statement Number	Frequency of execution	Total number of statements
1	1	1
2 – 5	1000	4000
6	1	1

On the whole, the number of statements that are executed are 4002. Instead, if the code is written as follows, then the total number of statements that are executed are only 1002.

1. $t_1 = y / z$
2. $x = x + t_1$
3. $x = x + t_1$
4. $x = x + t_1$
5. …

   ….

1001. $x = x + t_1$
1002. print x

The execution is three times better for later than the first form, but the space requirement is more. Inefficient code is generated in the loops for various reasons like

- Induction variable usage to keep track of iteration
- Unnecessary computations made inside the iterative loops that are not effective
- Use of high strength operators inside the loop

The important loop optimizations are elimination of loop invariant computations, strength reduction, code motion and elimination of induction variables.

10.9.1 A Loop Invariant Computation

A loop invariant computation is one that evaluates the same value every time the statements in loop are executed. Moving such a computational statements outside the loop leads to a reduction in the execution time.

Algorithm for elimination of loop invariant code

Step 1: Identify loop-invariant code.

Step 2: An instruction is loop-invariant if, for each operand:
- The operand is constant, OR
- All definitions for all the operands that reach the instruction inside the loop are made outside the loop, OR
- There is exactly one definition made using loop invariant variables inside the loop for an operand that reaches the instruction.

Step 3: Move it outside the loop.
- For each loop-invariant definition statement *i*: $x = y + z$ in basic block B, verify that it satisfies the conditions:
  a. B is the first block that dominates all exits of the loop
  b. *x* is not redefined anywhere else in the loop, that is, it is defined once in block
  c. block B dominates all uses of *x with* in the loop
     (i.e., all uses of *x* within the loop can be reached only by statement *i*.)
- Move each instruction i to newly created preheader if and only if it satisfies these requirements of the loop, making certain that any non-constant operands (like *y*, *z*) have already had their definitions moved to the pre-header.

Note: It is important that while applying loop-invariant code motion in nested loops, start at innermost loop and work outwards.

Example 12:

```
for (int i=0; i<1000; i++)
     x = x + y/z;
print x;
```

In the example, the value of *y* and *z* remains unchanged as there is no definition for these variables in the loop. Hence, the computation of *y*/*z* remains unchanged, which can be moved above the loop and the same code can be rewritten as follows:

```
t1= y/z
for (int i=0; i<1000; i++)
     x = x + t1;
print x;
```

10.9.2 Induction Variables

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. There are two types of induction variables—basic and derived. Basic induction variables are those that are of the form

$$I = I \pm C$$

where I is loop variable and C is some constant.

Derived induction variables are those that are defined only once in the loop and their value is a linear function of the basic induction variable.

For example,

$$J = A * I + B$$

Here J is a variable that is dependent on the basic induction variable *I* and the constants A and B. This is represented as a triplet (I, A, B)
Induction variable elimination involves three steps.

1. Detecting induction variable
2. Reducing the strength of induction variable
3. Eliminating induction variable

10.9.2.1 Detecting Induction Variable

Given a loop L with reaching definition information and the families of induction variable this algorithm generates the output that indicates the set of induction variables. For each induction variable J that belongs to family of I if, there is as associated triplet (I, A, B) where I is the basic induction variable, A and B are constants such that, value of J is computed as A * I + B.

Algorithm for detecting the induction variable

For all basic blocks

i. Scan the statements of loop L, if there is a loop—invariant computation associated with each basic induction variable, the triplet as (I, 1, 0).
ii. Search for a variable J within the loop L is defined as single assignment and is in the family of I where *I* is basic induction variable.

    J = I + B (then the triplet is (I, 1, B)
    J = I – B (then the triplet is (I, 1, –B)
    J = A * I (then the triplet is (I, A, 0)
    J = I / A (then the triplet is (I, 1/A, 0)

iii. If the induction variable J is in the family of some K, where K is not basic induction variable but is in family if I, then the other requirements are
    a. There is no assignment to I between the actual point of assignment to K in L and the assignment to J and
    b. There is no definition of K outside L reaches J.

Suppose the induction variable J = C * K where the triplet for K is (I, A, B), then triplet for J is given as (I, A * C, B * C). Once the family of induction variable is found, we modify the instructions, computing induction variable to use additions or subtractions rather than multiplications. This can be done by the strength reduction process, which is shown in the next algorithm.

Example 13: Let us consider the block B2, optimized code of quick sort program as in Figure 10.17. In this block, variable *i* is a basic induction variable as there is a lone assignment to *i* in the loop and increments by 1. $temp_2$ is in the family of *i* and its triplet is (*i*, 4, 0). Similarly *J* is the only basic induction variable in block B3 with $temp_4$ in the family of *J* with triplet (j, 4, 0)

```
5. i : = i + 1
6. temp2 : = 4 * i
7. temp3 : = a [ temp2 ]
8. if temp3 < pivot go to B2
```

Figure 10.17 Block B2 from Quick Sort Program

10.9.2.2 Strength Reduction Applied to the Induction Variable

For a loop L with reaching definition information and families of induction variables, we revise the loop where the high strength operations are replaced by low strength operations.

Algorithm

Consider each basic induction variable I, consider a variable J in the family of I depending on the variable I with triplet (I, a, b):

1. Create a new variable *K* for the two variables J_1 and J_2 with same triplet.
2. Replace the assignment to J by J = S
3. Add a new statement S = S + A*n immediately after every assignment I = I + n in L, and place S in family of I.
4. Ensure that S is initialized to A * I + b on entry to the loop. This initialization may be placed at the end of preheader which consists of

   S = A * I
   S = S + B

Example 14: On applying the previous algorithm, we got that in block B3 the basic induction variable is j and $temp_4$ is in family of J with triplet (j, 4, 0). On applying the above algorithm, the statement $temp_4$ = 4 * j is replaced by $temp_4 = s_4$ and inserts the assignment statement $s_4 = s_4 - 4$ after the assignment of j = j – 1 where –4 is obtained by multiplying the –1 in the assignment to j and the 4 in the triplet (j, 4, 0) for $temp_4$. Initialization of s_4 is placed at the end of block B1, which contains the definition of J. After strength reduction, we find that only use of some induction variables is in tests, we can replace a test of such an induction variable by that of another as shown in Figure 10.18.

10.9.2.3 Elimination of Induction Variable

The induction variable can be eliminated if the reaching definition information, loop invariant computation information, and live variable information is available.

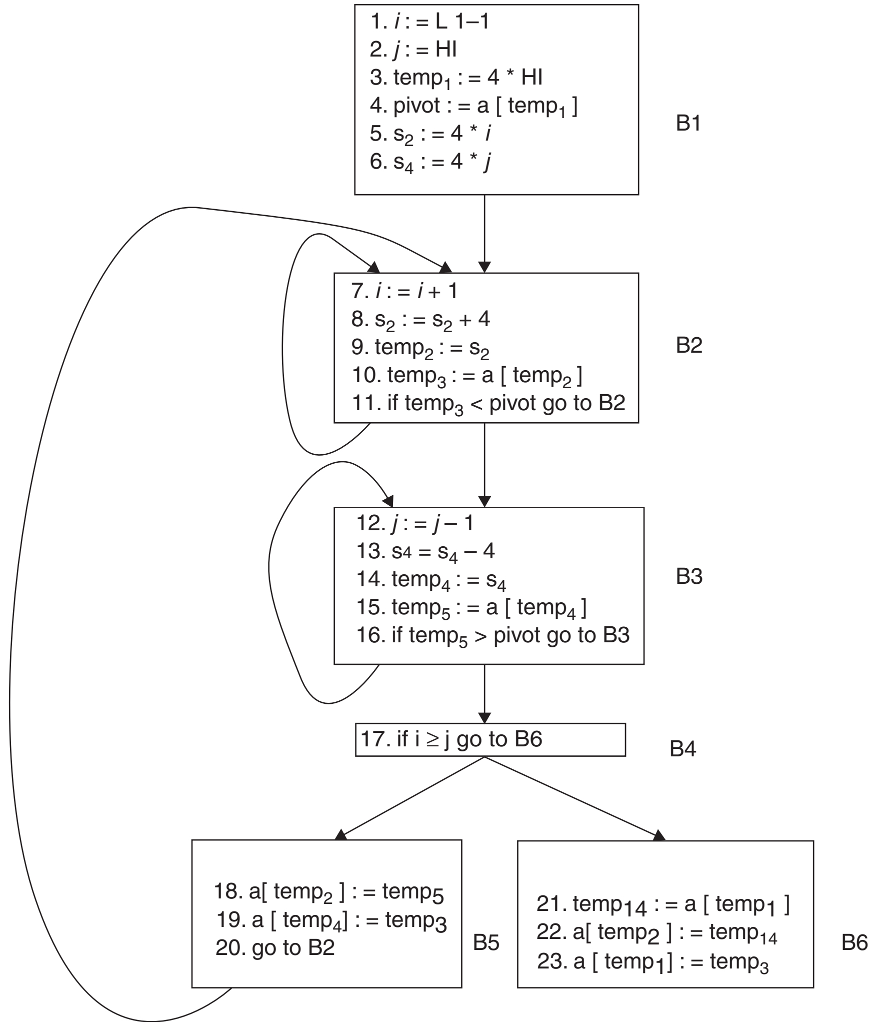


Figure 10.18 Quick Sort Program After Applying Strength Reduction

1. Let there be some basic induction variable that is used only to compute other induction variable in its family and in conditional branches.

   For example, let J be in the family of I with triplet (I, A , B) where A is positive.

   $\Rightarrow J = A * I + B.$

   Let there be a test of the form

   if I relop X goto B

   where X is not an induction variable; then this statement can be replaced as follows:

   $C = A * X$

C = C + B
If J relop C goto B

where C is a new temporary variable. This modification makes the code independent of induction variable I since comparing I with X is the same as comparing J with A*X+B
If there are two variables I_1 and I_2 then the test statement is

if I_1 relop I_2 goto B,

then we check for the variables J_1 and J_2 in the families of I_1 and I_2 with the triplets as (I_1, A_1, B_1) and (I_2, A_2, B_2) respectively with $A_1 = A_2$ and $B_1 = B_2$ then the statement

if I_1 relop I_2 goto B, can be replaced as

if J_1 relop J_2 goto B.

Once these variables are replaced, delete all the induction variables as these are useless.

On applying induction variable elimination to the quick sort program, the resultant code is shown in Figure 10.19.

10.10 Global Flow Analysis

To apply the optimization globally and get a good optimized code, it is required to collect information about variables or expressions in the program as a whole and distribute this information to each block. Data flow information can be collected by setting up equations and solving these equations at various points. These equations are framed in terms of four variables that are listed below.

IN(S)	it indicates the set of definitions that are reaching the statement S.
OUT(S)	it indicates the set of definitions that are leaving the statement S.
GEN(S)	the definition that is generated at this statement.
KILL(S)	the definition that is killed at this statement.

A typical equation has a form

OUT(S) = GEN(S) ∪ (IN(S) – KILL(S))

This indicates the information at the end of a statement is either generated within the statement, or enters at the beginning of the statement and is not killed in as the control flows through the statement.

The factors that influence the framing of the equations and solving them depends on

1. The notions of generating and killing depend on the desired information. Sometimes it may require analyzing the flow backwards rather than normal flow.
2. Since the data flows along the control path, the data flow analysis is affected by the control structures in the program.
3. There may be cases that may control the analysis. For example, procedure calls, assignment through pointers or array variables.

Before understanding the method for framing and solving the equations, let us try to understand reaching definition, use definition chains, definition use chains, live variable analysis.

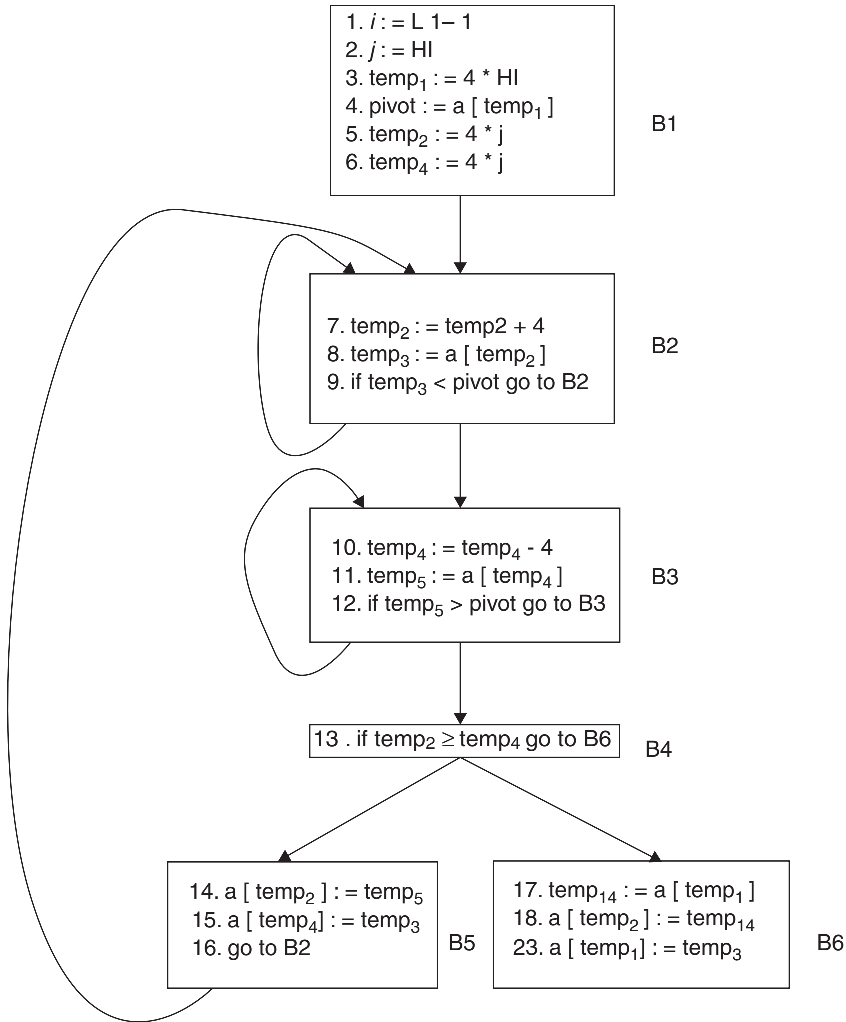


Figure 10.19 Code After Induction Variable Elimination

10.10.1 Points and Paths

A point is between two adjacent statements, as well as the point before and after a statement. If a block has four statements, then there would be five points in the block as shown in Figure 10.20.

A path is a sequence of points p_1, p_2, ……p_n, which give the global view of how the control flows and satisfies the condition for each i between 1 to n; either

1. p_i is a point immediately preceding a statement and p_{i+1} is the point immediately following the statement in the same block or

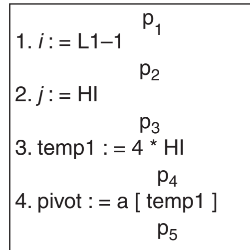


Figure 10.20 Basic Block Indicating Different Points

2. p_i is a point at the end of some block and p_{i+1} is a point which is at the beginning of the succeeding block.

10.10.2 Reaching Definition

A definition of a variable *a* is a statement that assigns, or may assign, a value to *a*. The common forms of definition to assign value to *a* is by an input statement or by an assignment statement. These forms are said to be unambiguous. The ambiguous definitions are

1. A call to a procedure with variable *a* as a parameter using call by reference.
2. An assignment through a pointer that could refer to variable *a*.

A definition *d* that reaches a point *p*, if and only if there exists a path from the point of its definition to the point immediately following *d*, such that *d* is not killed in any path that reaches *d*.

10.10.3 Use Definition Chains

Use definition chains are used to store the reaching definition information. It is the list of each use of a variable out of all definitions that reach that use.

- If a variable *a* has unambiguous definition then ud-chain for that use is a set of definitions in IN[B]. If there is an unambiguous definition within the block, then the use of *a* is the one that is last defined in the same block.
- If a variable *a* has an ambiguous definition, then all of these for which no unambiguous definition of *a* lies between it and the use of *a* are on the ud-chain for this use of *a*.

10.10.4 Live Variable Analysis

Some code optimization techniques depend on the information computed in the direction opposite to the flow of control. For example, in dead code elimination, we may eliminate the statements that assign values to variables that are never used. This requires the analysis in the opposite direction, which identified what definitions are used at any statement. In live variable analysis, we wish to know for a variable *x* and a point *p* whether the value *x* at *p* could be used along some path in the flow graph starting at *p*. if we say so, *x* is live at *p*, otherwise *x* is dead at *p*.

10.10.5 Definition Use Chains

A variable is used at a statement *s* if the r-value of the variable is required in the computation in statement *s*. The du-chain represents for a point *p* the set of uses s of a variable *x*, such that there is a path from *p* to *s* that does not redefine *x*.

10.10.6 Data Flow Analysis of Structured Programs

The program includes different control flow constructs such as if statements, iterative statements, which would control the framing of data flow equations. This section gives different ways of framing equations that are structure dependent.

Figure 10.21 gives the list of rules for framing equations where Da indicates the set of all definitions corresponding to variable *a*.

10.10.7 Representation of Sets

The set of definitions for GEN and KILL can be represented using bit representation, which is compact and easy to compute. We assign a number for each definition and then use a bit vector that has the bits depending on the number of definitions. To compute union, we can

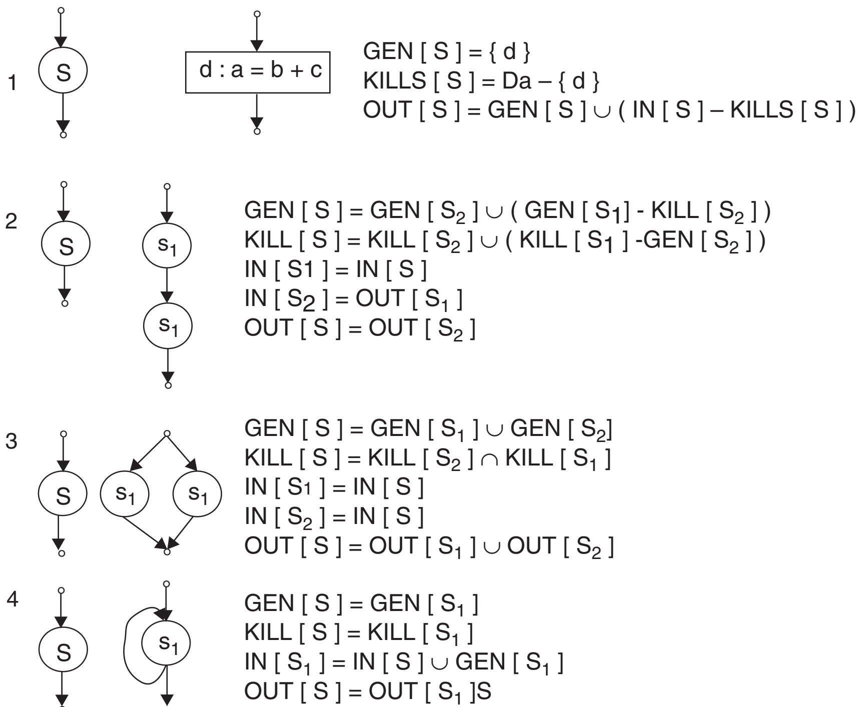


Figure 10.21 Rules for Framing Equations

perform logical operation, for intersection perform a logical operation and for difference, that is, A – B perform logical A and (not B).

Example 15: Consider the program that has eight definitions as shown below

```
d1  I = n - 1
d2  J = m
d3  a = x 1
    do
d4  I = I + 1
d5  J = J - 1
    if E1 then
d6        a = x2
    else
d7        I = x2
d8        a = x3
    while E2
```

In the above partial program, there are eight definitions from d1 through d8. Eight bits are required for bit representation. At any given point, the KILL, GEN, IN, OUT are represented as 8 bits. The bits that are set to 1 are said to be those definitions that are killed, generated, reaching, or leaving the point respectively. The set for KILL and GEN are computed by applying the data flow equations to the statements. These statements can be represented as a syntax tree using the following rules.

S → id: = E | S ; S | if E then S else S |
do S while E | if E then S
E → id + id | id – id | id * id | id / id | id

We can represent the entire program in the form of syntax tree using the rules. For each node in the parse tree, we can compute the values for GEN and KILL as explained below.

GEN(d1) = 1000 0000 / as only one definition is done here

KILL(d1)= 0001 0010 / the definition here is for I and this kills the definition of I in definitions of d4 and d7

Similarly, for all definitions, d2, d3 …..d8 the calculations are done. At a point above d1 and d2 to compute GEN and KILL we apply the rule of statements in sequence

GEN[S] = GEN[S2] ∪ (GEN[S1] – KILL[S2])
⇒ 0100 0000 ∪ (1000 0000– 0000 1000)
⇒ 0100 0000 ∪ (1000 0000 ∩ 11110111)
⇒ 0100 0000 ∪ (1000 0000)
⇒ 1100 0000

KILL[S] = KILL[S2] ∪ (KILL[S1] – GEN[S2])
⇒ 0000 1000 ∪ (0001 0010 – 0100 0000)
⇒ 0000 1000 ∪ (0001 0010 ∩ 1011 1111)
⇒ 0000 1000 ∪ (0001 0010)
⇒ 0001 1010

Similarly, calculations are done for each node and the values are shown in the annotated parse tree of Figure 10.22.

10.10.8 Iterative Algorithm for Reaching Definition

Once the KILL and GEN for each block is computed, we can compute IN and OUT for each block using the formulas

$$N[B] = \bigcup_{p \in \text{predecessor}(B)} OUT[p]$$

$$OUT[B] = GEN[B] \cup (N[B] - KILL[B])$$

We use iterative approach, starting with an estimation for IN[b] = ∅ for all blocks B and converging to the desired value of IN and OUT. The algorithm sketch is given below.

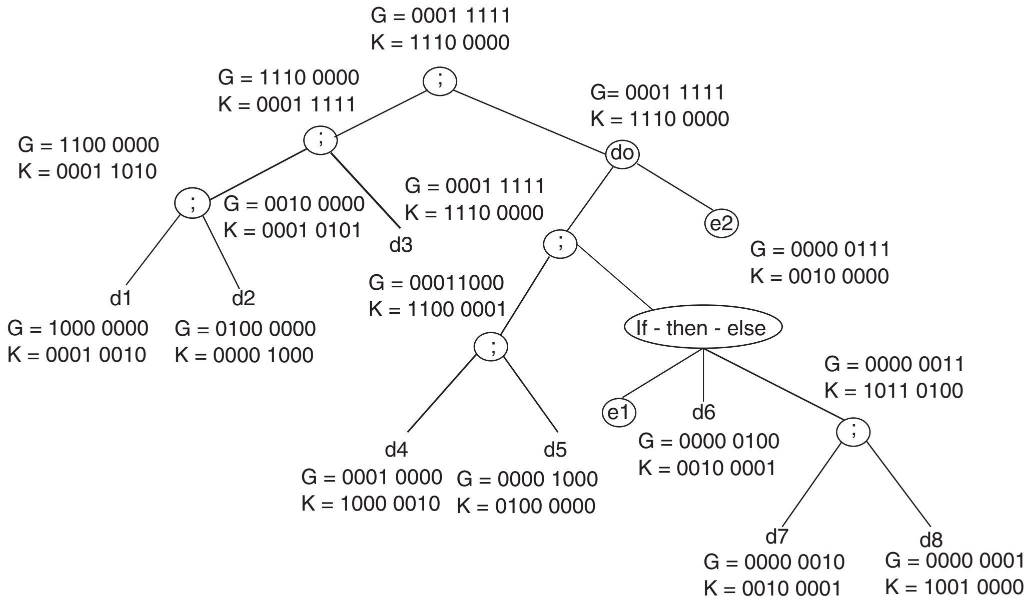


Figure 10.22 Annotated Parse Tree with GEN and KILL Values

1. initialize IN[B]= ∅ and OUT[B] based on GEN[B] values
2. for each block B do until OUT[B] = GEN[B]
3. change = true
4. while change do begin
5. change = false
6. for each block B do begin
7. $N[B] = \bigcup_{p \in \text{predecessor}(B)} OUT[p]$

8. OLDOUT= OUT[B]
9. OUT[B]=GEN[B] ∪ (IN[B] – KILL[B])
10. if OUT[B] [1] OLDOUT then change = true
11. end
12. end

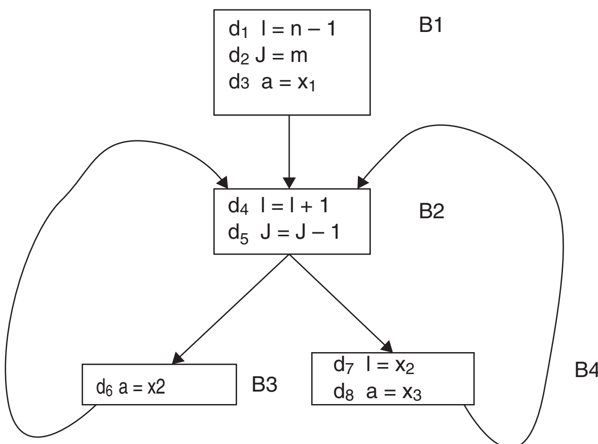
Example 16: The above algorithm when applied to the flow graph of Example 15, we can observe that at the end, the GEN and KILL values for each block are shown in Figure 10.23.

According to the algorithm, first initialize the values of IN[B] with all zeros and based on GEN information find the values for OUT for each block.

PASS 1:

IN[B1] remains the same as there is no predecessor block.

OUT[B1] = GEN[B1] ∪ (IN[B1] – KILL[B1])
= 1110 0000 ∪ (0000 0000 – 0001 1111)
= 1110 0000 ∪ (0000 0000 ∩ 1110 0000)



GEN(B1) = {d1, d2,d3} = 1110 0000
KILL(B1) = {d4,d5,d6,d7,d8} = 0001 1111

GEN(B2) = {d4,d5} = 0001 1000
KILL(B2) = {d1,d2,d7} = 1100 0010

GEN(B3) = {d6} = 000 0100
KILL(B3) = {d3, d8} = 0010 0001

GEN(B4) = {d4, d8} = 0000 0011
KILL(B4) = {d1,d3, d4,d6,} = 1011 0100

Figure 10.23 GEN and KILL Values for Each Block

Block B	Initial		Pass 1		Pass 2	
	In[B]	Out[B]	In[B]	Out[B]	In[B]	Out[B]
B1	0000 0000	1110 0000	0000 0000	1110 0000	0000 0000	1110 0000
B2	0000 0000	0001 1000	1110 0111	0011 1101	1111 1111	0011 1101
B3	0000 0000	0000 0100	0011 1101	0001 1100	0001 1101	0001 1100
B4	0000 0000	0000 0011	0011 1101	0001 1111	0001 1101	0001 1111

$= 1110\ 0000 \cup 0000\ 0000$
$= 1110\ 0000$

IN[B2] = OUT[B1] ∪ OUT[B3] ∪ OUT[B4]
$= 1110\ 0000 \cup 0000\ 0100 \cup 0000\ 0011$
$= 1110\ 0111$

OUT[B2] = GEN[B2] ∪ (IN[B2] – KILL[B2])
$= 0001\ 1000 \cup (1110\ 0111 - 1100\ 0010)$
$= 0001\ 1000 \cup (1110\ 0111 \cap 0011\ 1101)$
$= 0001\ 1000 \cup 0011\ 0101$
$= 0011\ 1101$

IN[B3] = OUT[B2] = 0011 1101

OUT[B3] = GEN[B3] ∪ (IN[B3] – KILL[B3])
$= 0000\ 0100 \cup (0011\ 1101 - 0010\ 0001)$
$= 0000\ 0100 \cup (0011\ 1101 \cap 1101\ 1110)$
$= 0000\ 0100 \cup 0001\ 1100$
$= 0001\ 1100$

IN[B4] = OUT[B2] = 0011 1101

OUT[B4] = GEN[B4] ∪ (IN[B4] – KILL[B4])
$= 0000\ 0011 \cup (0011\ 1101 - 0010\ 0001)$
$= 0000\ 0011 \cup (0011\ 1101 \cap 1101\ 1110)$
$= 0000\ 0011 \cup 0001\ 1100$
$= 0001\ 1111$

PASS 2:
IN[B1] remains same as there is no predecessor block.
OUT[B1] = GEN[B1] ∪ (IN[B1] – KILL[B1])
$= 1110\ 0000 \cup (0000\ 0000 - 0001\ 1111)$
$= 1110\ 0000 \cup (0000\ 0000 \cap 1110\ 0000)$
$= 1110\ 0000 \cup 0000\ 0000$
$= 1110\ 0000$

IN[B2] = OUT[B1] ∪ OUT[B3] ∪ OUT[B4]
$= 1110\ 0000 \cup 0001\ 1100 \cup 0001\ 1111$
$= 1111\ 1111$

OUT[B2] = GEN[B2] ∪ (IN[B2] – KILL[B2])
$= 0001\ 1000 \cup (1111\ 1111 - 1100\ 0010)$
$= 0001\ 1000 \cup (1111\ 1111 \cap 0011\ 1101)$
$= 0001\ 1000 \cup 0011\ 1101$
$= 0011\ 1101$

IN[B3] = OUT[B2] = 0011 1101

OUT[B3] = GEN[B3] ∪ (IN[B3] – KILL[B3])
$= 0000\ 0100 \cup (0011\ 1101 - 0010\ 0001)$
$= 0000\ 0100 \cup (0011\ 1101 \cap 1101\ 1110)$

$= 0000\ 0100 \cup 0001\ 1100$
$= 0001\ 1100$

$IN[B4] = OUT[B2] = 0011\ 1101$

$OUT[B4] = GEN[B4] \cup (IN[B4] - KILL[B4])$
$= 0000\ 0011 \cup (0011\ 1101 - 0010\ 0001)$
$= 0000\ 0011 \cup (0011\ 1101 \cap 1101\ 1110)$
$= 0000\ 0011 \cup 0001\ 1100$
$= 0001\ 1111$

Since the OUT of each block in pass 2 is same with pass 1, it is converged and can be stopped.

The same technique can also be applied for common sub expression and use this information for global optimization.

10.11 Machine-Dependent Optimization

This optimization can be applied on target machine instructions. This includes register allocation, use of addressing modes and peep hole optimization. Instructions involving register operands are faster and shorter (instruction length is small); hence, if we make use of more registers during target code generation, efficient code will be generated. Hence, register allocation and use of addressing modes also contribute to optimization. The most popular optimization that can be applied on target machine is peephole optimization.

Peephole Optimization

Generally code generation algorithms produce code, statement by statement. This may contain redundant instructions and suboptimal constructs. The efficiency of such code can be improved by applying peephole optimization, which is simple but effective optimization on target code. The peephole is considered a small moving window on the target code. The code in peephole need not be contiguous. It improves the performance of the target program by examining and transforming a short sequence of target instructions. The advantage of peephole optimization is that each improvement applied increases opportunities and shows additional improvements. It may need repeated passes to be applied over the target code to get the maximum benefit. It can also be applied directly after intermediate code generation.

In this section, we shall define the following examples of program transformations that are characteristic of peephole optimizations.

10.11.1 Redundant Loads and Stores

The code generation algorithm produces the target code, which is either represented with single operand or two operands or three operands. Let us assume the instructions are with two operands. The following is an example that gives the assembly code for the statement $x = y + z$.

1. MOV y, R_0
2. ADD z, R_0
3. MOV R_0, x

Instruction 1 moves the value of y to register R_0, second instruction performs the addition of value in z with the register content and the result of the operation is stored in the register. The third instruction copies the register content to the location x. At this point the value of x is available in both location of x and the register R_0.

If the above algorithm is applied on the code $a = b + c$, $d = a + e$ then it generates the code given below:

1. MOV b, R_0
2. ADD c, R_0
3. MOV R_0, a
4. MOV a, R_0
5. ADD e, R_0
6. MOV R_0, d

Here we can say that 3 and 4 are redundant load and store instructions. These instructions will not affect the values before or after their execution. Such redundant statements can be eliminated and the resultant code is as follows:

1. MOV b, R_0
2. ADD c, R_0
3. ADD e, R_0
4. MOV R_0, d

10.11.2 Algebraic Simplification

There are few algebraic identities that occur frequently enough and are worth considering.

Look at the following statements.

$x: = x + 0$
$x: = x * 1$

They do not alter the value of x. If we keep them as it is, later when code generation algorithm is applied on it, it may produce six statements that are of no use. Hence, such statements whether they are in three address code or target code can be removed.

10.11.3 Dead Code Elimination

Removal of unreachable code is an opportunity for peephole optimization. A statement immediately after an unconditional jump or a statement that never get a chance to be executed can be identified and eliminated. Such code is called the dead code.

```
For example consider a statement in high level language code
# define x = 0
…….
If (x)
{ ----print value
-----
}
```

```
If this is translated to target code as

If x=1 goto L1
       goto L2
L1: print value
L2: …….. 
```

Here value will never be printed. So whatever code inside the body of “if(x)”is dead code; hence, it can be removed.

10.11.4 Flow-of-Control Optimization

Sometimes when we apply code generation algorithms mechanically we may get jump on jumps as follows:

```
goto L1
L1: goto L2
…….
L2: goto L3
….
L3: if a < b goto L4
L4:
This can be optimized as
goto L3
……….
L3: if a < b goto L4
L4:
```

10.11.5 Reduction in Strength

This optimization mainly deals with replacing expensive operations by cheaper ones. For example

- $x^2 \Rightarrow x * x$
- fixed-point multiplication and division by a power of 2 ⇒ shift
- floating-point division by a constant ⇒ floating-point multiplication by a constant

10.11.6 Use of Machine Idioms

While generating the target code it is better to make use of rich instruction set supported by the target machine, instead of blindly applying available code-generation algorithms. This may produce efficient code. Feature provided by machine architecture may be identified and used wherever applicable to reduce the overall execution time significantly.

For example, consider the statement $x = x + 1$; if we apply the code-generation algorithm mentioned in redundant load/store, we get six instructions but there can be hardware instructions for certain specific operations auto-increment and auto-decrement addressing mode like INR x, which is one instruction only.

Solved Problems

1. Create DAG for the following code:

$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = b[t_1]$
$t_4 = t_2 * t_3$
$pr = pr + t_4$
$i = i + 1$
if i <=20 goto (1)

Solution: The DAG for a given code is shown in Figure 10.24

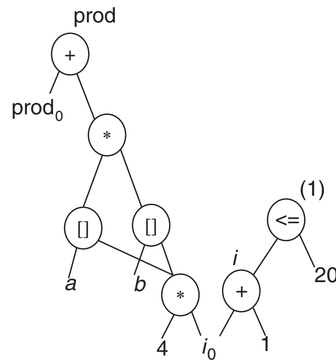


Figure 10.24 DAG for 1

2. Divide the following code, which is for factorial function, into basic blocks and draw flow graph

1. $f = 1$;
2. $i = 2$;
3. If (i>x) goto (8)
4. $f = f * i$;
5. $t_1 = i + 1$;
6. $i = t_1$;
7. goto (3)
8. goto calling program

Solution: By using the algorithm for leaders, we identify leaders as 1, 3, 4, 8.

The statement starting from a leader up to the next leader is a basic block.
Figure 10.25 shows the basic blocks for the factorial program.
The flow graph is shown in Figure 10.26.

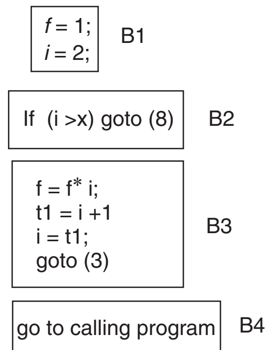


Figure 10.25 Basic Blocks for Factorial Program

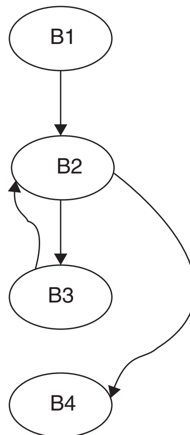


Figure 10.26 Flow Graph for Factorial Program

3. Consider the following part of code.

```
for(i=1; i<n; i++)
      for(j=1; j<n; j++)
            c[i,j] = a[i,j] * b[i,j];
print("done");
```

a. Write its equivalent three address code.
b. Identify the basic blocks.

Solution: a. The three address code for the above program is as follows:

```
 1. i = 1
 2. j = 1
 3. t1 = 4 * i
 4. t2 =i - 1
 5. t3 =t2 + j
 6. t4 = t3 * 4
 7. t5 = a[t4]
 8. t6 = i - 1
 9. t7 = t6 * n
10. t8 = t7 + j
11. t9 = t8 * 4
12. t10 = b[t9]
13. t11 = t5 * t10
14. t12 = i - 1
15. t13 = t12 * n
16. t14 = t13 + j
17. t15 = t14 * 4
18. t16 = c[t15]
19. t16 = t11
20. t17 = j + 1
21. j = t17
22. if j < n go to 2
23. t18 = i + 1
24. i = t18
25. if i < n go to 1
26. print("done");
```

b. To identify the basic blocks, first identify the leader statements. In the above three address code, the leader statements are statement 1, 2, 23, and 26.

```
 1. i = 1                      →leader 1
 2. j = 1                      →leader 2
 3. t1 = 4 * i
 4. t2 = i - 1
 5. t3 = t2 + j
 6. t4 = t3 * 4
 7. t5 = a[t4]
 8. t6 = i - 1
 9. t7 = t6 * n
10. t8 = t7 + j
11. t9 = t8 * 4
12. t10 = b[t9]
13. t11 = t5 * t10
```

```
15. t13 = t12 * n
16. t14 = t13 + j
17. t15 = t14 * 4
18. t16 = c[t15]
19. t16 = t11
20. t17 = j + 1
21. j = t17
22. if j < n go to 2
23. t18 = i + 1                    →leader 3
24. i = t18
25. if i <n go to 1
26. print("done");                 →leader 4
14. t12 = i - 1
```

Basic block 1 has statement 1
Basic block 2 has statements 2 – 22
Basic block 3 has statements 23 – 25
Basic block 4 has statement 26.

The flow graph is shown in Figure 10.27.

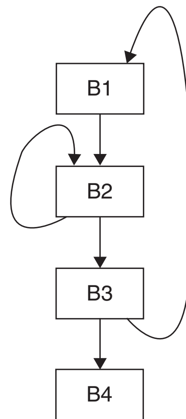


Figure 10.27 Flow Graph for 3

4. Optimize the code applying suitable optimization techniques on the resultant three address code for question 3.

Solution: The three address code is given in the previous problem. On applying redundant sub expression elimination we get it as

```
1. i = 1
2. j = 1
3. t1 = 4 * i
```

```
 4. t2 = i - 1
 5. t3 = t2 + j
 6. t4 = t3 * 4
 7. t5 = a[t4]
 8. t6 = b[t4]
 9. t7 = t5 * t6
10. t8 = c[t4]
11. t8 = t7
12. t9 = j + 1
13. j = t9
14. if j < n go to 2
15. t10 = i + 1
16. i = t10
17. if i <n go to 1
18. print("done");
```

5. For the following statements, draw the DAG.

$t_1 = a + b;$
$t_2 = a * b;$
$t_3 = t_1 * b;$
$c = t_3 + t_2;$

Solution:

The DAG for the above sequence of statements is in Figure 10.28

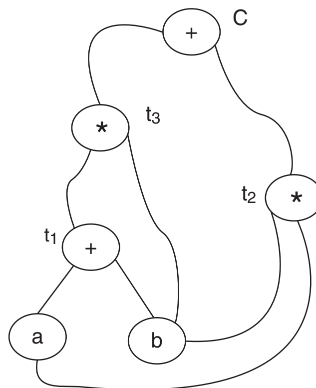


Figure 10.28 DAG for 5

6. Write the procedure to construct the DAG for a statement. Represent the following statement using DAG.

$a = (b + c)*(d–c)+a/(d–c)+(b+c)$

Solution: Procedure for constructing DAG is explained in Section 10.6. The DAG for the given expression is in Figure 10.29.

7. Represent the statement x = a/(b+c) – d*(e+f) using DAG.
Solution:

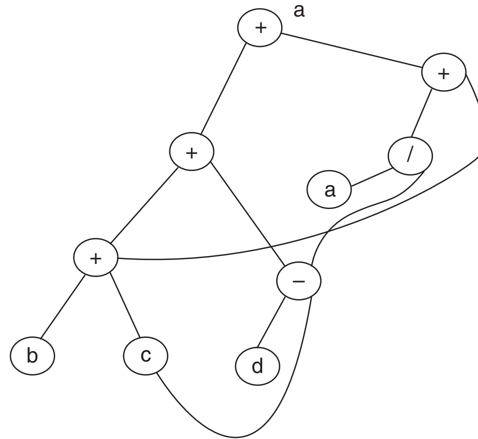


Figure 10.29 DAG for 6

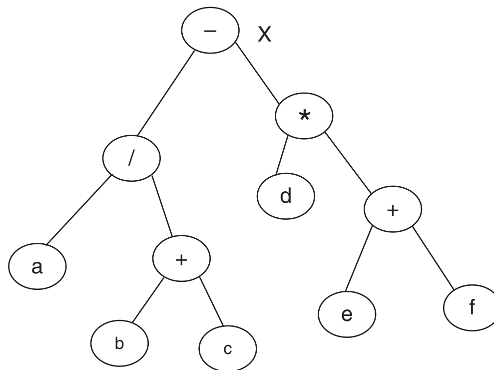


Figure 10.30 DAG for 7

Summary

- Optimization improves the code in terms of space and time.
- Machine-independent optimization techniques are applied on intermediate representation.
- Machine-dependent optimization techniques are applied on target code of machine code.

- Flow graphs are used to represent the three address code where nodes are basic blocks and the edges represent the control flow.
- A basic block is the basic unit of program where all the statements in the block are either executed or ignored depending on the control flow.
- Leader is the first statement in the basic block.
- DAG representation is useful in identifying the redundant sub expression.
- Function-preserving transformations are performed without changing the function it computes.
- Structure-preserving transformations are performed without changing the set of expressions computed in the block.
- Algebraic transformations are useful to change the set of expressions computed by basic block into algebraically equivalent set.

Fill in the Blanks

1. ____________ are the principal sources of optimization.
2. ____________ analysis is used to detect loops in intermediate code.
3. Applying optimization guarantee that generated code is optimal. (Y/N) ____________
3. Replicating the body of the loop to reduce number of tests if the number of iterations is constant is called ____________.
4. Merging the bodies of two loops if the loops have same number of iterations and they use same indices is called ____________.
5. Moving a computation from a high-frequency region to a low-frequency region is called ____________.
6. ____________ is a code which is never used.
7. ____________ are useful data structures for redundancy elimination.
8. Replacing a costlier operation by a cheaper one is called ____________.
9. The usefulness of a variable is listed using ____________.
10. The possible definitions that are reachable at a particular point are listed using ____________.
11. In a path $p_1, p_2, \ldots p_n$, if p_i is the last point in a block then p_{i+1} is ____________.
12. To apply global redundant sub expression elimination, ____________ gives the required information.

Objective Question Bank

1. Folding is a___________ optimization with regard to the basic block.
   (a) global (b) local
   (c) universal (d) none
2. Which of the following is machine-dependent optimization?
   (a) Strength reduction (b) Redundancy elimination
   (c) Flow of control optimization (d) None

3. Which of the following is machine-independent optimization?
   (a) Strength reduction
   (b) redundancy elimination
   (c) loop optimization
   (d) all
4. ________ optimization should not introduce additional errors in the program.
   (a) Folding
   (b) Constant propagation
   (c) Any
   (d) None
5. For finding leaders for basic block, we use the rule as the_______ statement in a leader.
   (a) firs
   (b) last
   (c) statement that is before a conditional jump
   (d) statement that is before a unconditional jump
6. Statement starting from a leader up to the next leader is called_______.
   (a) leader
   (b) DAG
   (c) basic block
   (d) none
7. Which of the following is peephole optimization?
   (a) Common sub expression elimination
   (b) Flow of control optimization
   (c) Loop jamming.
   (d) None
8. Advantages of copy propagation is that_______.
   (a) it often reduces code
   (b) it often turns the copy statement into dead code
   (c) it often makes code to run faster
   (d) None
9. Code motion is_______.
   (a) moving part of computation outside loop.
   (b) moving loop invariant computation before loop
   (c) Either (a) or (b)
   (d) None
10. Folding can be applied to_______.
    (a) subscripted variables
    (b) floating point numbers
    (c) un subscripted variables
    (d) None

Exercises

1. Write the importance of code optimization. Explain the strength-reduction technique in loop with example.
2. Explain with example the various techniques in loop optimization.
3. What is local and global optimization? Explain with example any three local optimization techniques.

4. Consider the following part of code.

```
int main()
{
    int n,k=0;
    scanf("%d,"&n);
    for(i=2; i<n;i++)
    {
        if( (n % i) == 0) break;
    }
    k=1;
    if( i==n)
        printf("number is prime");
    else
        printf("number is not prime");
}
```

   a. Identify the basic blocks in the given program.
   b. Draw the domination tree for the program.
   c. Using dead code elimination identify the statements that are eliminated.

5. Explain the following loop optimization techniques with examples.

   a. Frequency reduction
   b. Strength reduction
   c. Code motion

6. Explain about (a) loop unrolling (b) loop jamming (c) code motion.

7. Explain about (a) folding (b) strength reduction (c) redundant code elimination (d) dead code elimination.

8. Write the procedure to construct the DAG for a statement. Represent the following statement using DAG.

$$a = (x + y) * (x - y) + x / (x - z) + (x + y)$$

9. Explain DAG and its use. Write the procedure to construct the DAG for a statement.

10. What is a basic block? With an example explain the procedure to identify the basic blocks in a given program. Draw the domination tree for the following procedure.

```
(16M) -M
    read a,b
    c=a+b
    If (c>=30) go to 10
        c=c*2
```

```
        go to 20
  10  c=c*3
  20  print a,b,c
```

11. a. Explain DAG and its use.
    b. For the following statements draw the DAG.

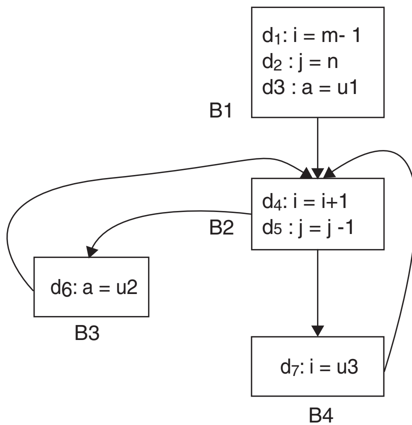
$$
\begin{aligned}
t_8 &= d + e \\
t_6 &= a + b \\
t_5 &= t_6 - c \\
t_4 &= t_5 * t_8 \\
t_3 &= t_4 - e \\
t_2 &= t_6 + t_4 \\
t_1 &= t_2 * t_3
\end{aligned}
$$

12. a. What is local and global optimization? (6 + 5 + 5 M) – M

```
read a,b
c=a+b
If (c>=30) go to 10
      c=c+0
      c=c*2
      go to 20
10    c=c*3
20    print a,b,c
```

    b. Identify the basic blocks in the given program.
    c. Identify the optimization technique that can be applied and write the optimized code.

14. Define the following terms.
    a. Reaching definition.
    b. Live variables.
    c. Use definition chains.
    d. Definition use chains.
15. Explain the formulation of data flow equations for reaching definition in structured programs. Describe the procedure to compute in and out values.
16. Explain about global data flow analysis. Explain the factors that affect the data flow equations.
17. Write the iterative algorithm for reaching definition. Compute in and out for the following flow graph.



18. Explain about global data flow analysis. List the data flow equations for reaching definitions for structured programs.

19. a. Explain the use of algebraic identities in optimization.
    b. Explain strength reduction with example.

20. a. Explain live variable analysis with example.
    b. Explain redundant sub expression elimination with example.

Key for Fill in the Blanks

1. dead code elimination and constant propagation
2. global flow
3. No
4. loop unrolling and loop jamming
5. code motion
6. dead code
7. DAG
8. strength reduction
9. use definition chains
10. definition use chains
11. first point in the successor block
12. data flow analysis

Key for Objective Question Bank

1. b.	2. a.	3. d.	4. b.	5. a.
6. c.	7. b.	8. b.	9. c.	10. c.

CHAPTER 11

Code Generation

Code generation is the final phase in a compiler. Given a code in intermediate form, it uses code generation algorithm and register allocation strategies to generate the final target code.

CHAPTER OUTLINE

11.1 Introduction
11.2 Issues in the Design of a Code Generator
11.3 Approach to Code Generation
11.4 Instruction Costs
11.5 Register Allocation and Assignment
11.6 Code Generation Using DAG

Code generation phase is responsible for generating the target code. This chapter focuses on the issues in code generation phase and the register allocation strategies. Code generation algorithm for three address code and DAG is explained with an example.

11.1 Introduction

Code generator is the last phase in the design of a compiler. It takes three address code or DAG representation of the source program as input and produces an equivalent target program as output. Figure 11.1 shows the position of the location of code generation phase.

The code generator has many limitations regarding the generation of code that is of high quality, accurate, and efficient. In addition to this, the code generator should run efficiently. There are many issues that are to be considered while designing a code generator.

11.2 Issues in the Design of a Code Generator

The code generated is target language dependent and operating system dependent as memory management, instruction selection, register allocation, order of evaluation would affect the efficiency of the code generated. Even the input to the code generation phase is an issue because there are many forms of intermediate codes that are generated by the front end.

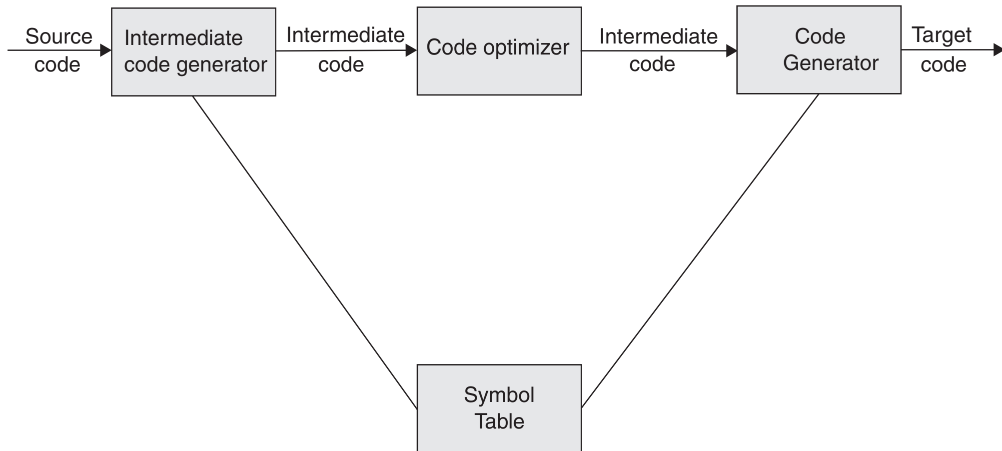


Figure 11.1 Code Generator

11.2.1 Input to the Code Generator

The output of front end is the input to the code generator along with the information in the symbol table that is used to determine the run time address of the data objects denoted by the names in the intermediate representation.

Intermediate code is either in the linear form or in the hierarchical form. Linear representation that includes postfix notation, three address representation (quadruples and triples), and hierarchical representation includes syntax trees and DAGs. The input is intermediate representation and is error free. The values of variable names present in the intermediate code can be represented by target machine in a directly manipulatable form. Since semantic analyzer has already performed the type checking, type conversion operators have been inserted wherever necessary and obvious semantic errors have already been detected.

11.2.2 Target Programs

The output of the code generator is the program in target language. Various possible outputs can be generated.

Absolute machine code—this code is static and is always placed in the same location in memory. This can be loaded and run immediately. Fast in execution but requires same memory locations. This may be suitable for programs like VI editor. Compilers like PL/C produce absolute code.

Relocatable machine code—this allows the subprograms to be compiled separately resulting in a set of relocatable object modules. Loader and linker programmes are needed to link the modules and load the programs into the memory for execution. Although the procedure is expensive, there is flexibility in being able to compile subroutines separately and to

call other previously compiled programs from an object module. If relocation is not carried out automatically by the target machine then the compiler must provide explicit relocation information to the loader. This is used to link the separately compiled object modules.

Assembly code—When we have Assembly language as the output, it makes the process of code generation easier. We can generate mnemonic instructions and use the macro facilities of the assembler to generate code. The cost involved after code generation is less as the other forms require an assembler. Particularly for a machine with a low memory, this choice is reasonable, where a compiler must use more passes.

11.2.3 Memory Management

The role of code generator in coordination with front end is to map the names of variables in the source program to the addresses of the data objects in run time memory. The name in a three address statement refers to a symbol table entry for that name, which is used by code generator.

Each label in the three address code has to be converted to actual memory addresses of instructions and this process is called "back patching." In case of quadruples, if the numbers are referred by labels, then each quadruple is read and the address is computed by maintaining a counter for the words used for the instructions generated so far. The quadruple array within an additional field is used to store the count.

For example, if a reference such as *j: goto i* is encountered, where *i* could be less than or greater than *j*,

- *if i is less than j*, it is a backward jump, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple *i*.
- The jump is forward jump when i is greater than *j*. Hence, *i* exceeds *j*. Here we have store on quadruple list *i* the location of the first instruction generated for quadruple *j*. Then the quadruple *i* is processed. Then for all forward jumps to *i*, proper machine locations are filled.

11.2.4 Instruction Selection

The instruction set of the target machine is an important factor that controls the uniformity of the execution. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Machine idioms and speed of instruction are other important factors to consider in code generation. Instruction selection is straightforward provided the efficiency of the target program is not an important task then, but this results in more computation time.

For example, let us consider a simple statement a: = b + c, where a, b, and c are statically allocated and can be translated into the machine code as follows:

```
MOV b, R0 /* load b into register R0 */
ADD c, R0 /* add c to R0 */
MOV R0, a /* store R0 into a */
```

If the code generator translates statement-by-statement, it often produced a very poor code as given below.

Let the statements in three address code be

a := b + c
d := a + e

would be translated into

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

The fourth statement in the code sequence is redundant. If the value of *a* is not subsequently used, then the third statement is also redundant.

Let us consider another statement, a = a + 1; if this statement is translated into machine code it results in three machine instructions. If the target machine has an increment instruction INC, using this instruction is more efficient as it would perform the same task as that of three instructions generated in normal code generation.

Instruction speeds are needed to design good code sequence but knowing the accurate timing information is often a difficult task. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

11.2.5 Register Allocation

Efficient use of registers is particularly important in code generation as the instructions involving the register operands are short and fast than those instructions involving the operands in memory. The use of registers is often subdivided into two sub problems:

- During the first phase, that is, register allocation phase, we select the set of names that reside in registers at a point in the program.
- During the later phase of register assignment, we pick the register where a variable will reside in

Register assignment to variables is a difficult task as the registers available are to be used by the operating system and also by the programs that are currently running on the system. Mathematically this problem is NP-complete problem and is also complicated because the hardware and/or the operating system of the target machine may require that certain register usage.

Few target machines require register pairs for some operations and to store results. For example, the integer division and multiplication requires register pairs in the IBM System OS 370 machine. The multiplication instruction is of the form

M x, y

The even register of an even/odd register pair will have x, which is the multiplicand. The multiplier y is a single register and the multiplicand value is taken from the odd register pair. The result occupies the complete even/odd register pair. Similarly, the division instruction is of the form

D x, y

In the even/odd register pair, the 64-bit dividend occupies even register x and y represents the divisor. After performing the division operation, the remainder is stored in the even register and the quotient is stored in the odd register.

11.2.6 Choice of Evaluation Order

The efficiency of the target code depends on the evaluation order. The computation order also effects the register requirements to hold intermediate results. Choosing the best order is another difficult task. If the input is in the form of three address code, it may require reordering of the input for efficient code generation. If the input is in the form of DAG, then the best code can be generated by traversing the tree in post order form.

11.3 Approach to Code Generation

An important criterion for a code generator is to generate the code that is correct in terms of meaning and efficiency. Correctness is another important factor because of the number of different cases that code generator must face. Because of prominence in correctness, designing a good target code generator that can be implemented, tested, and maintained easily is an important design goal.

The input for code generator is a sequence of three-address statements partitioned into basic blocks. A simple code generation involves generating code for each three-address statement, taking the advantage of the operands that are in the registers and storing the result in registers as long as possible. The output in register is stored until it is needed for the next computation or just before a procedure call, jump/labeled statement, or end of the 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. This is done to avoid possible error in using the data that reaches that point.

The code generator should keep track of what is currently available in the registers and where the data of a variable is available which will cost less. For this reason it uses two data structures called register descriptor and address descriptor.

Register descriptor: It is a structure that maintains a pointer to the list that contains information about what is currently available in each of the registers. Initially all the registers are set to empty. Whenever a code generator makes a request for register to be allocated, this list is verified, if it already holds the operand it is returned otherwise a free register is allocated.

Address descriptor: It is a structure that keeps track of the locations for each variable name—where the current value of the name can be found at run time. This information can be stored in the symbol table.

The code generator first invokes a function getreg(), that returns a location specifying, where the computation has to be performed by three-address statement. If there is a statement *a* = *b op c,* the getreg() function returns a location L where the computation of *b op c* should be performed. This could be the memory location or a register; the decision depends on the required efficiency.

11.3.1 Algorithm for Code Generation Using Three Address Code

The code generator reads every three-address statement, which is of the form *a* = *b op c* it first invokes the getreg() function, generates the target code and appropriately updates the register and address descriptors as follows:

For every three-address statement of the form *a* = *b op c* in the basic block do

{

1. First invoke the function getreg() to return the register or memory location L in which the operation b op c should be performed. The input for this function is three-address statement *a* = *b* op *c* as a parameter, which can be done by passing the index of this statement in the quadruple array.
2. Check the current location of the operand *b* by consulting its address descriptor, and if the value of *b* is currently present in both memory location and in register, then prefer the register reference. If the value of *b* is currently not available in *L*, then generate a statement MOV b, L (where b as assumed to represent the current location of b).
3. Generate the statement OP c, L, and modify the address descriptor of a to state that value of a is now available in *L*, and if *L* is in a register, then modify its descriptor to indicate that it will contain the run-time value of *a*.
4. If the current values of *b* and /or *c* are in the register, and there is no next use of these variables, that is, they are not live at the end of the block, then alter the register descriptor to indicate that after the execution of the statement *a* = *b* op *c*, these registers will no longer contain value of b and /or *c*.

}

The procedure getreg(), when invoked it returns a location where the computation relating to the given three-address statement *a* = *b* op *c* should be performed. The location to be used to be returned is based on the following conditions:

1. Prefer the register location if one of the register already contains the variable name *b*. If b has no next use after the execution of *a* = *b* op *c*, and if *b* is not live at the end of the block and this register does not hold the value of any other variable, then return the register for L.
2. If such a register is not available then getreg() search is made to find an empty register; and if an empty register is available, then it returns it for *L*.
3. In the absence of an empty register, and if a has next use in the block, or op is an operator, such as indexing, which requires a register for computation, then getreg() searches for a suitable register. If this register is not empty then a statement is generated to empty the register by generating store instruction to store its

value. In the appropriate memory location M, the address descriptor is modified, and the register that is freed is returned for L. It uses the different strategies and one such strategy is least recently used to empty one of the occupied register.

4. Finally, the getreg() procedure returns the memory location L if es that have next use within the block.

Example: Let us consider the sequence of statements as given below.

```
 t = a – b;
u := a – c;
v := t + u;
d := v + u;
```

When these statements are given as input for a code generator, then the operations are performed as follows:

First the address descriptor and register descriptor are set to empty. Let us assume that R0 and R1 registers are available for performing these operations.

For the first statement $t = a - b$, since the registers are empty, the get register would return register R0 for the computation and hence, the instruction MOV *a*, R0 is generated and the register descriptor indicates that R0 holds the value of *a*. Then it generates the instruction SUB *b*, R0 and the address descriptor for t is set to R0. The last instruction generated is MOV R0 *t* and updates the address descriptor indicating that the value of *t* is available in both register R0 and memory location *t*. Similarly for each statement in three address code is processed and the resultant code generated is shown in Table 11.1.

Table 11.1 Contents of register and address descriptor during code generation

Statement	Code generated	Register descriptor	Address descriptor
		R0, R1 are empty	All variables are in memory
$t = a - b$	MOV *a* R0	R0 contains *a*	A in R0 and memory
	SUB *b* R0	R0 contains *t*	A in memory, *t* in R0
	MOV R0 *t*	R0 contains *t*	A in memory, *t* in R0 and memory
$u := a - c$	MOV *a* R1	R0 contains *t* R1 contains *a*	A in R1 and memory, *t* in R0 and memory
	SUB *c* R1	R0 contains *t* R1 contains *u*	A in memory, *t* in R0 and memory, *u* in R1
	MOV R1 *u*	R0 contains *t* R1 contains *u*	A in memory, *t* in R0 and memory *u* in R1 and memory
$v := t + u$	ADD R1 R0	R0 contains *v* R1 contains *u*	A in memory, *t* in memory *u* in R1 and memory, *v* in R0
$d := v + u$	ADD R1 R0	R0 contains *d* R1 contains *u*	A in memory, *t* in memory *u* in R1 and memory, *v* in R0
	MOV R0 *d*	R0 contains *d* R1 contains *u*	A in memory, *t* in memory *u* in R1 and memory, *d* in R0

If the code generator is efficient, then the target code that is generated would have the statements as follows:

The algorithm makes use of the next-use information of each name for efficient use of the registers. 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 *a*,
- And if a statement at the index *j* in the same block uses *a* as an operand,
- If the value assigned to a variable a is not modified in any path from any statement at index *i* to the statement at index *j then*

we say that the value of *a* computed by the statement at index *i* is used in the statement at index *j*. In such cases the next use of the variable a in the statement *i* is statement *j*. For each three address statement *i*, compute the information relating to variables that appear in statement *i*, which has next use in the same block. Backward scanning of basic block allows to attach every statement i under consideration with information of those statements that have next use of each variable name in statement i. The algorithm is as follows:

For each statement i of the form *a* = *b op c* do

{

- compute information relating to the next uses of *a*, *b*, and *c* to statement i
- update the next-use for a as no next-use /* This information can be kept into the symbol table */
- update the information for b and c to be the next use in statement *i*

}

Once we gathered the information regarding the next use, use this information for selecting register allocation strategies.

The register usage reduces the cost of computation. It may not always be true, it depends on the factor how the instruction is written, what addressing modes are used, the number of registers available, number of variables used in the program, and their frequency.

11.4 Instruction Costs

The instruction cost depends on the operation and the addressing modes of the operands involved in the instruction. Addressing modes involving the register have cost zero, while those with a memory location or literal in them have one as the operands have to be stored with the instruction.

1. The instruction MOV R0, R1 has cost one, since it occupies only one word of memory. Both the operands in the instruction are registers which costs zero.
2. The instruction MOV R0, M has cost two, since the address of memory location M is in the word following the instruction.
3. The instruction ADD #1, R0 has cost two, since the constant 1 is in the next word following the instruction.

4. The instruction SUB 4(R0), *12(R1) has the cost three as the constants 4 and 12 are stored in the next two words following the instruction.

 The following table gives the details of the added cost based on the addressing mode.

Mode	Form	Address	Added Cost
Register	R	R	0
Indirect Register	*R	Contents(R)	0
Absolute	M	M	1
Indirect register	*R	Contents (R)	1
Indexed	C(R)	C + contents(R)	1
Indirect Indexed	*C(R)	Contents(C + contents(R))	1

Note: For calculating, the cost of instruction is one plus the added cost of source and, destination, i.e.,
 Cost of instruction = 1 + cost of source address + cost of destination address.

Example:

1. To move register contents to memory (® M)
 MOV R_0, M
 cost = 1 + 0 + 1 = 2.
2. To move the contents from register in indirect indexed mode to memory
 MOV * 4 (R_0), M
 cost = 1 + indirect index + instruction word = 1 + 1 + 1 = 3
3. To move the contents from register in indexed mode to memory
 MOV 4(R_0), M
 cost = 1 + indirect mode + instruction word = 1 + 1 + 1 = 3
4. To move constant or litetral to register
 MOV #1, R_0
 cost = 1 + 1 + 0 = 2
5. To move from memory to memory
 MOV M1, M2
 cost = 1 + 1 + 1 = 3

11.5 Register Allocation and Assignment

From the examples in the previous section, it is clear that the register operands are faster and shorter than the memory operands. A good code generator should consider the availability of registers and the use of variables for generating an efficient code. There are different strategies for identifying the registers and their assignment.

11.5.1 Fixed Registers

The most common and simple strategy is to assign the specific register to specific values. For example, use a separate set of registers for storing the base address, set of registers for storing the stack pointers, set of registers for arithmetic computations and few are reserved by compiler for suitable operations. The advantage of this approach is that the register allocation task is simplified. But the disadvantage is that all the registers are not utilized properly.

11.5.2 Global Register Allocation

When the code is generated for a single block, the variables that are consistently used are stored in the registers. At the end of the block only those variables that are live are stored in the registers. The allocation of variables across the block boundaries is known as global register allocation and this must be consistent. The strategies available for global register allocation are listed below:

- Registers can be fixed to store values for most frequently used variables throughout the loop.
- Number of registers can be fixed to hold the most active values in each inner loop.
- Preference may be given to the free registers if available to one block.

11.5.3 Usage Count

The best way of register allocation is to count the number of times the variable is used in the basic block and then assigning the register to the variable that has the highest usage count. This usage count gives the idea about the reduction in cost of code generated depending on selection of register allocation for specific variable.

$$\sum_{\text{blockBinL}} (use(x,B) + 2 * live\ (x,B))$$

Here use(x,B) gives the number of times the variable x is used in block B, prior to the definition of x. and live(x,B) is 1 if x is live on exit from B, otherwise it is 0.

Let us consider the following program fragment in the Figure 11.2, which has four blocks B1, B2, B3, and B4.

The variables that are used are I, J, a, n, m, x_1, x_2, and x_3. The cost of I is computed across every block as follows:

Block(*B*)	Use(*x*,*B*)	Live(*x*,*B*)	Use count
B1	0	1	0 + 2 * 1 = 2
B2	1	1	1 + 2 * 1 = 3
B3	0	0	0
B4	0	1	0 + 2 * 1 = 2
		Total	7

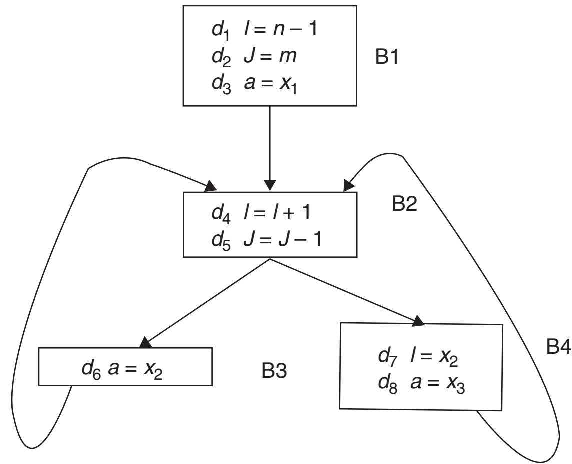


Figure 11.2 Flowgraph

Similarly, the use count for all variables is computed.

Variable	Use count
I	7
J	5
a	6
n	1
m	1
x_1	1
x_2	1
x_3	1

Suppose, there are only two registers then it would be better to reserve them for variable *I* and *a* as their use count is more. If there are four registers, then three can be assigned for *I*, *J*, *a* and the fourth register can be used for other variables.

11.5.4 Register Assignment for Outer Loop

If the program has nested loops, that is, loop L2 inside the loop L1, then while assigning the registers the following criteria may be chosen.

- If *x* is allocated in inner loop L2 then it should not be allocated in outer loop L1-L2.
- If *x* is allocated in L1 and is not used in L2 then store the variable *x* in memory at the entrance to L2 and load while leaving L2.

◆ If x is allocated in L2 and not used in L1 then load z on the entrance of L2 and store z on the exit of L2.

11.5.5 Graph Coloring for Register Assignment

Register allocation can be done using graph coloring. Each variable is represented as a node in the graph. If the variables are interfering, then we place an edge between them and this indicates that the same register cannot be assigned for both of them. If there is no edge between two nodes, that is, they are not adjacent, then we can use the same register for both variables, which will reduce the number of registers.

Let us consider n registers $r_1, r_2 \ldots, r_n$ with n different colors. Now it is required to assign a color so that no two adjacent nodes get the same color. This resembles the graph coloring problem. The solution to this problem will result in minimum number of registers to be used for execution of the program which is cost effective.

11.6 Code Generation Using DAG

Code generation using DAG would result in efficient code as it eliminates redundant instructions and uses minimum number of registers. Target code is generated in two phases—numbering and code generation.

Numbering phase assigns a number to each node in the tree that indicates how many registers are needed to evaluate the subtree of this node. Let us assume that for a node T it requires l register to evaluate its left subtree and r registers to evaluate the right subtree. If one of the numbers is large, say $l > r$, then we can evaluate the left subtree first and store its result into one of the registers R_k. Now the same registers can be used to evaluate the right subtree excluding the register R_k. If $l = r$ we need an extra register R_{l+1} to remember the result of the left subtree. If node T is a leaf then the number of registers to evaluate T is either 1 or 0 depending whether it is a left or a right subtree. For example, the instruction ADD R1, A it is better to handle the right operand A directly without storing it into register. Usually the numbering algorithm starts from the leaf nodes of the tree and assigns 1 or 0 as explained. Then for each node whose children are labeled l and r, if they are equal then the node number is $l + 1$ otherwise it is *maximum* of l and r.

For example, for the expression (A – B) + ((C + D) + (E * F)), which corresponds to the AST/DAG as shown in the following Figure 11.3.

Numbering phase would process this and assign the register number to each node as shown in Figure 11.4.

Code generation phase generates the code based on the numbering assigned to each node T. All the registers available are arranged as a stack to maintain the order of the lower register at the top. This makes an assumption that the required number of registers cannot exceed the number of available registers. In some cases, we may need to spill the intermediate result of a node to memory. This algorithm also does not take advantage of the commutative and associative properties of operators to rearrange the expression tree.

It first checks whether the node T is a leaf node; if yes, it generates a load instruction corresponding to it as ***load top(), T***. If the node T is an internal node, then it checks the left l and right r subtree for the number assigned. There are three possible values, the number on the

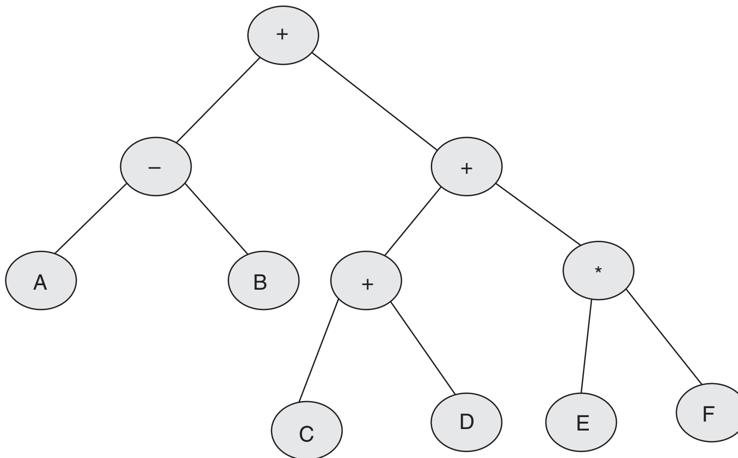


Figure 11.3 AST for (A–B) + ((C+D) + (E ∗ F))

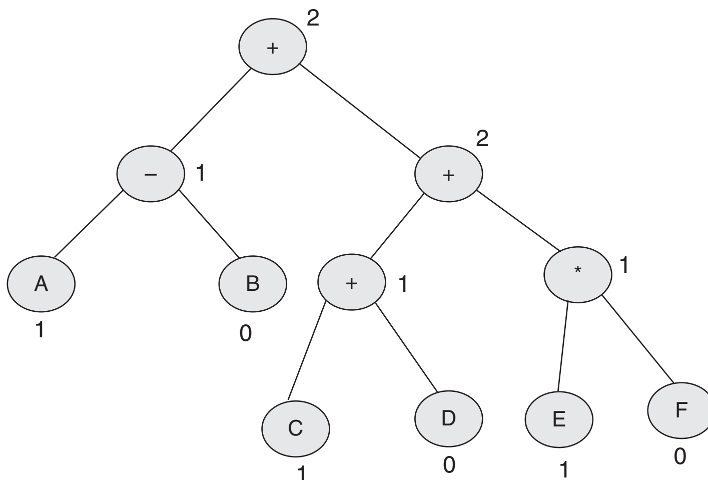


Figure 11.4 Register Allocation

right is 0 or greater than or less than the number on the left. If it is 0 then call the generate() function with left subtree *l* and then generate instruction ***op top(), r***. If the numbering on the left is greater than or equal to right, then call generate() with left subtree, get new register by popping the top, call generate() with right subtree, generate new instruction for ***OP R, top(),*** and push back the used register on to the stack.

If the number on the left is smaller than the number on the right, then first swap the top two elements on the stack, call the generate() function with the right subtree, get new register by popping the top, call generate() with left subtree, generate new instruction for

OP R, top() push back the used register on to the stack, and finally swap the top two elements of the stack.

Algorithm to generate the target code using the numbering information.

```
function CodeGen_DAG(T)
{
    If T is a leaf node then generate Load top(), T
    If T is some internal node then identify the l and r
        children then
            {
          CodeGen_DAG(l)
          Generate statement op top(), r
            }
    If register (l) > register( r ) then
            {
          CodeGen_DAG(l)
          R=pop()
          CodeGen_DAG(r)
          Generate statement op R, top()
          Push(R)
            }
    If(register(l) < register (r) then
            {
          Exchange the top two elements of the stack
          CodeGen_DAG(r)
          R=pop()
          CodeGen_DAG(l)
          Generate statement op R, top()
          Push(R)
          Exchange the top two elements of the stack
            }
}
```

It is clear from the numbering phase that this evaluation requires two registers R1 and R2. Let the registers be arranged with R1 on top of the stack. The code generator first calls the generate function with the root node. Since the right child has the number 2, which is greater than the left, it first swaps the top two register numbers and then calls the generate function with the right sub tree. This procedure is repeated until the leaf is reached and the resultant code generated is as follows:

```
    load R2, C
    add R2, D
    load R1, E
```

```
mult R1, F
add R2, R1
load R1, A
sub R1, B
add R1, R2
```

Solved Problems

1. Generate the target code for the following three address code.

D := B – C
E :=A + B
B := B + C
A := E – D

Solution: Let us assume there are two registers. The code generated is displayed step by step in the following table.

Statement	Code generated	Register descriptor	Address descriptor
		R0, R1 are empty	All variables are in memory
D = B – C	MOV B R0	R0 contains B	B in R0 & memory
	SUB C R0	R0 contains D	B in memory, D in R0
	MOV R0 D	R0 contains D	B in memory, D in R0 & memory
E :=A + B	MOV A R1	R0 contains D R1 contains A	A in R1 & memory, D in R0 & memory
	ADD B R1	R0 contains D R1 contains E	A in memory, D in R0 & memory, E in R1
	MOV R1 E	R0 contains D R1 contains E	A in memory, D in R0 & memory E in R1 & memory
B := B + C	MOV B R0	R0 contains B R1 contains E	B in R0 & memory, D in memory, E in R1 & memory
	ADD C R0	R0 contains B R1 contains E	B in R0, D in memory, E in R1 & memory
	MOV R0 B	R0 contains B R1 contains E	B in R0 & memory, D in memory, E in R1 & memory
A := E – D	SUB D R1	R0 contains B R1 contains A	B in R0 & memory, E in memory, A in R1
	MOV R1 A	R0 contains B R1 contains A	B in R0 & memory, E in memory, A in R1 & memory

2. Draw DAG for the statement $a/(b + c) - d * (e + f)$ and generate the target code.

   Solution: The above expression when represented as a DAG then we get Figure 11.5. After the numbering phase, the number of registers required at each node is computed and displayed in Figure 11.6.

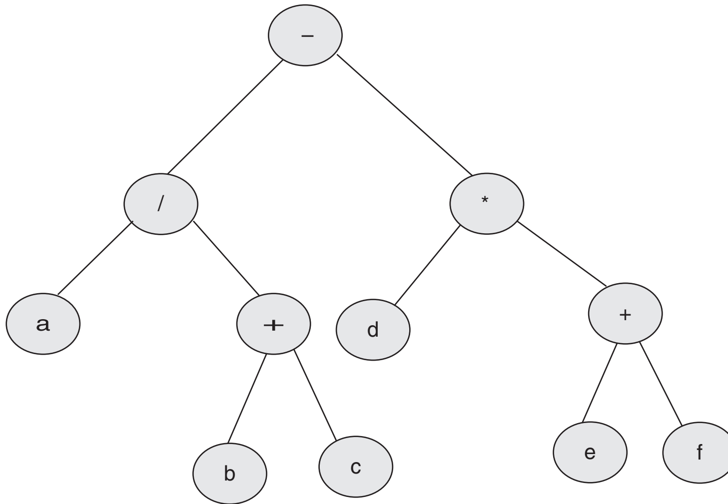


Figure 11.5 AST for a/(b+c) – d ∗ (e +{f)

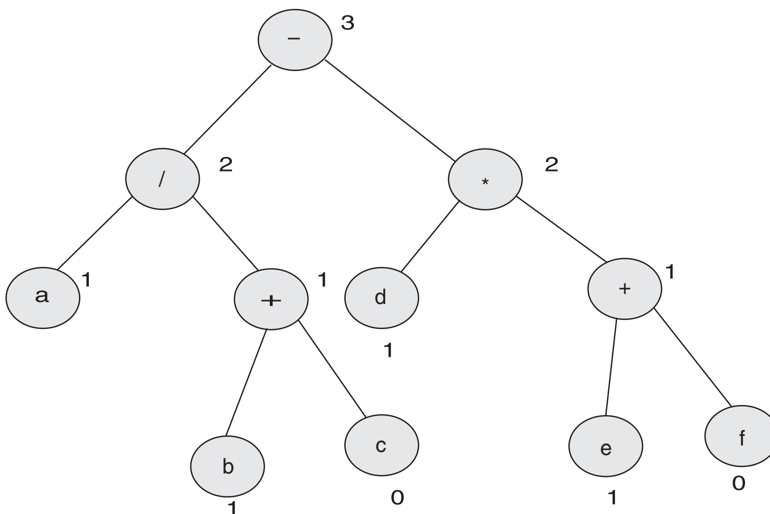


Figure 11.6 Register Allocation

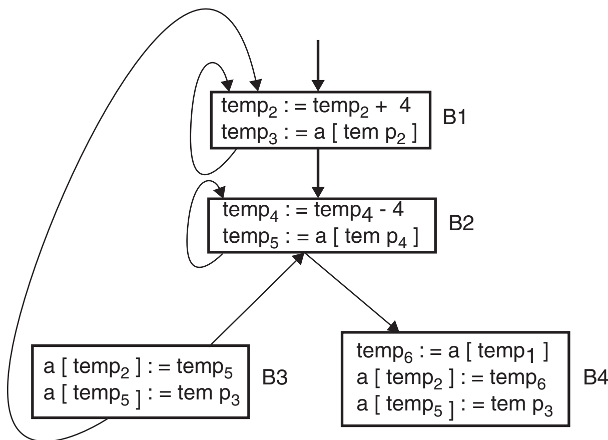
```
MOV A, R1
MOV B R2
ADD C R2
DIV R2 R1
MOV D R2
MOV E R3
ADD F R3
MUL R3 R2
SUB R2 R1
```

Assuming there are three registers available, the target code generated is listed below.

3. If there are three registers r_1, r_2, and r_3, how are these registers allocated for the following program fragment?

Solution:

The variables that are used are $temp_1$, $temp_2$, $temp_3$, $temp_4$, $temp_5$, and $temp_6$. The cost of these variables is computed across every block as follows:



Variable	$\sum_{\text{blockBinL}} (use(x,B) + 2 * live(x,B))$	Total Use count
$temp_1$	1 + 2 * 1	3
$temp_2$	1 + 2 * 3	7
$temp_3$	2 + 2 * 1	4
$temp_4$	1 + 2 * 2	5
$temp_5$	1 + 2 * 1	3
$temp_6$	0 + 2 * 1	2

Similarly, the use count for all variables is computed as shown in the table below.

If there are three registers, then these registers are to be assigned to the variables $temp_2$, $temp_4$, $temp_3$ as their use count is high.

Summary

- Code generator is dependent on the target language.
- Different forms that the input code generator can have are polish notation, three address code, and syntax trees.
- There are three forms of target code—absolute machine code, relocatable machine code, and assembly code.
- Space and time of the object code also depends on the type of instructions used.
- DAG's representation is useful in generating efficient code.
- Register descriptors are maintained to keep track of the contents of the registers.
- Address descriptors are maintained to keep track of the availability of the value of variable.
- The getreg() function returns the register that can be used for the computation of the instruction.
- The cost of instruction is dependent on the register allocation.
- Use count and graph coloring approaches will improve the cost of program execution with minimum number of registers being used.
- Code generation using DAG helps in generating the code that uses minimum number of registers.

Fill in the Blanks

1. ________________ phase is responsible for generating the target code.
2. The intermediate codes that are in linear representation are ________________ and ________________.
3. The intermediate codes that are in hierarchical representation are ________________ and ________________.
4. ________________ code is static and is always placed in the same location in memory.
5. ________________ and ________________ programmes are needed to link the modules and to load the programs into the memory for execution.
6. The conversion of all labels in three address statements to addresses of instructions is known as ________________.
7. Target code for the instruction $a = b * c$ is ________________.
8. Use of ________________ reduces the cost of instruction.
9. ________________ maintains a pointer to the list that contains the information about what is currently available in the registers.

10. ________________ keeps track of locations of each variable.
11. What is the cost of the instruction mov r_5, r_3?
12. ________________ form of writing the instruction would minimize the instruction cost.
13. C(R) indicates that the instruction is in the ________________ mode.
14. The cost of instruction is dependent on the ________________ of the operands.
15. ________________ gives the number of times the variable x is used in block B.

Objective Question Bank

1. The input of code generation phase is.
   (a) Polish notation. (b) Three address code
   (c) Abstract syntax tree (d) Any one of the above
2. Name the programs that are needed to run the code that is in relocatable form.
   (a) Assemblers and loaders (b) Loaders and linkers
   (c) Assemblers and linkers (d) None
3. Back patching is a process that comprises which of the following tasks?
   (a) The labels in three address statements are converted to the addresses of instructions.
   (b) The address of instruction is converted to labels in three address statements.
   (c) Both are valid
   (d) None
4. Which of the following machine idioms perform the task equivalent to $a = a + 1$?
   (a) INC (b) SFT
   (c) Both are valid (d) None
5. Register assignment to variables is ______________ problem.
   (a) P – hard problem (b) P – complete problem
   (c) NP – hard problem (d) NP – complete problem
6. For integer multiplication in IBM System/370 ________ registers are used.
   (a) single (b) pair
   (c) two pairs (d) Any of the above
7. Name the descriptor required to keep track of content of registers.
   (a) Address (b) Register
   (c) Both (d) Any of the above
8. Name the descriptor used to keep track of the availability of value for the variables.
   (a) Address (b) Register
   (c) Both (d) Any of the above
9. Which of the following intermediate code helps in generating the efficient target code?
   (a) Post fix notation (b) Three address code
   (c) Quadruples (d) DAG
10. Code generator is dependent on
    (a) Type of input (b) Type of output
    (c) Register allocation (d) All

Exercises

1. Explain the code generation algorithm, function getreg() with an example.
2. Write about the use of DAG in code generation. Explain the procedure for constructing DAG with an example.
3. (a) Write about the issues in the design of code generator.

   (b) Write about target code forms. Explain how the instruction forms effect the computation time.
4. Write the code generated for the following statements.

   (a) A = B[i] (b) A[i] = B
   (c) A = *p (d) A = B + C
5. (a) Write about global register allocation strategy for loops.
   (b) Explain code generation from DAG. For the following instructions construct DAG.

   t_1: = a + b
   t_2: = a + b
   t_3: = e − t_2
   t_4: = t_1 − t_3
6. For the following instructions, construct DAG and generate code with and without optimization.

   t_1: = a + b
   t_2: = c + d
   t_3: = e − t_2
   t_4: = t_1 − t_3
7. Generate the code for the following C statements using its equivalent three address code.

   (a) a = b + c (b) x = a/(b + c) – d * (e + f)
   (c) * A = p (d) A = B + C
8. Generate the code for the following C statements using its equivalent three address code.

   (a) a = b + 1 (b) x = y + 3
   (c) y = a/b (d) a = b + c
9. Explain the following terms.

   (a) Register descriptor. (b) Address descriptor.
   (c) Instruction costs.
10. (a) What is DAG? Construct the DAG for the following basic block.

    D: = B – C
    E: = A + B
    B: = B + C
    A: = E – D
11. (a) Explain the importance of register allocation with respect to optimization?
    (b) Explain the importance of addressing modes with respect to optimization?
12. (a) Write about global register allocation strategy for loops.
    (b) Explain code generation from DAG. For the following instructions construct DAG.

t_1: = a/b
t_2: = a/b
t_3: = e – t_2
t_4: = t_1 – t_3
t_5: = e – t_2
t_6: = t_4 * t_5

13. (a) Write the three-address code for the following code.

```
fact(x)
{ int f=1;
for(i=2, i >=x, i++)
{
 f=f*i;
 return f;
}
```

(b) Represent the above code using DAG.

14. (a) Explain the importance of register allocation with respect to optimization.
(b) Explain the procedure for constructing DAG with an example. Write the applications of DAG.

Key for Fill in the Blanks

1. Code generation
2. Postfix notation, three address representation.
3. Syntax trees and DAGs
4. Absolute machine code
5. Loader and linker
6. Back patching.
7. mov b R1, mul c R1, mov R1 a
8. Registers
9. Register descriptor.
10. Address descriptor
11. One
12. Register and indirect register
13. Indexed
14. Addressing mode
15. use(x,B)

Key for Objective Question Bank

1. d	2. b	3. a	4. a	5. d
6. b	7. b	8. a	9. d	10. d

This page is intentionally left blank.

Recommended Readings and Websites

Further Readings

1. *Compiler Design*, by Santanu Chattopadhyay. PHI Learning (2006).
2. *Compilers: Principles, Techniques, & Tools*, by Alfred V. Aho. Pearson (1986).
3. *Comprehensive Compiler Design*, by O. G. Kakde. Laxmi Publications (2005).
4. *Formal Languages and Automata Theory*, by A. A.Puntambekar. Technical Publications (2011).

Online References

1. 202.113.80.118:8080/sjjg/upLoad/down/.../20120704005.doc
2. baggins.nottingham.edu.my/~hsooihock/G52CMP/semantic.ppt
3. books.google.co.in/books?isbn=8131707881
4. books.google.co.in/books?isbn=8131759024
5. books.google.co.in/books?isbn=8184314892
6. books.google.co.in › Computers › Programming › Algorithms
7. citeseerx.ist.psu.edu/viewdoc/download?... - United States
8. digitallibrary.srmuniv.ac.in/dspace/bitstream/.../1/3259.pdf
9. dinosaur.compilertools.net/lex/
10. dinosaur.compilertools.net/yacc/index.html
11. docs.oracle.com › ... › Chapter 3 yacc -- A Compiler Compiler
12. dragonbook.stanford.edu/lecture-notes/.../20-Optimization.pdf
13. elearning.algonquincollege.com/coursemat/ranevs/.../NRPP.do..
14. elearning.vtu.ac.in/16/ENotes/.../Unit6-NKC.pdf
15. en.wikipedia.org/wiki/AVL_tree
16. en.wikipedia.org/wiki/Compiler
17. en.wikipedia.org/wiki/Formal_language
18. en.wikipedia.org/wiki/Linker_(computing)
19. home.pcisys.net/~aharon/classnotes3.doc
20. lambda.uta.edu/cse5317/fall02/notes/node39.html
21. lylib.com/books/en/1.424.1.79/1/ - United States
22. man.cat-v.org/plan_9/1/lex
23. nptel.iitm.ac.in/courses/Webcourse.../power...9/9_18.html
24. polaris.cs.uiuc.edu/~padua/cs321/PARSING1.ppt
25. smkfit.files.wordpress.com/2012/01/compiler-notes-unit-iv.pdf
26. suif.stanford.edu/~courses/cs243/lectures/l2.pdf
27. tinman.cs.gsu.edu/~raj/4340/sp12/notes/bottomup-Ullman.pdf
28. uw714doc.sco.com/en/SDK.../_Ambiguity_and_Conflicts.html
29. http://acorwin.wordpress.com/2012/10/18/lexical-analysis-the-role-of-the-lexical-analyzer-section-3-1/
30. http://code.google.com/p/cbse-065/wiki/module10
31. http://computacion.cs.cinvestav.mx/~acaceres/courses/itesm/lp/clases/lp07b.pdf
32. http://csis.bits-pilani.ac.in/faculty/dck/spl/slide/spl4.pdf

33. http://en.wikipedia.org/wiki/Chomsky_hierarchy
34. http://en.wikipedia.org/wiki/Regular_grammar
35. http://en.wikipedia.org/wiki/Syntax_diagram
36. http://flylib.com/books/en/1.424.1.31/1/
37. http://grammar.about.com/od/basicsentencegrammar/a/grammarintro.htm
38. http://lambda.uta.edu/cse5317/notes
39. http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT-KANPUR/compiler-desing/chapter_1/1_1a.html
40. https://parasol.tamu.edu/~rwerger/Courses/434/lec14.pdf
41. http://programming4.us/desktop/392.aspx
42. http://sandbox.mc.edu/~bennet/ada/examples/numbers_adb.html
43. http://svnweb.freebsd.org/csrg/old/yacc/PSD.doc/ss2?revision=62773&view=co&pathrev=62773
44. http://www.augustana.ab.ca/~mohrj/courses/2000.fall/csc370/lecture_notes/ebnf.html
45. http://www.bituh.com/2012/10/19/14bii-explain-the-transformation-of-basic-blocks/
46. http://www.boddunan.com/articles/education/19-engineering.html?start=195
47. http://www.codeduniya.com/slide_folder/Language%20Processors/Scanning.pdf
48. http://www.cs.arizona.edu/classes/cs453/fall12/class_notes/PDF/LexicalAnalysis.pdf
49. http://www.cs.columbia.edu/~aho/cs4115/lectures/13-02-25.htm
50. http://www.eecs.wsu.edu/~cook/tcs/l5.html
51. http://www.fit.vutbr.cz/~meduna/zap/ZAP_SLAJDY.pdf
52. http://www.labautopedia.org/mw/index.php/List_of_programming_and_computer_science_terms
53. http://www.mec.ac.in/resources/notes/notes/compiler/module3/type%20check.htm
54. http://www.univ-orleans.fr/lifo/Members/Mirian.Halfeld/Cours/TLComp/l3-OverviewTL.pdf
55. www.bituh.com/.../14aii-explain-code-generation-phase-with-s...
56. www.cs.auckland.ac.nz/~jmor159/PLDS210/hash_tables.html
57. www.csa.syr.edu/~chapin/cis657/yacc.pdf
58. www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html
59. www.cs.northwestern.edu/academics/courses/322/notes/14.ppt
60. www.cs.nyu.edu/courses/fall06/G22.2130-001/class-notes.html
61. www.cs.rice.edu/~keith/512/2011/.../L08Reassoc-1up.pdf
62. www.cs.wright.edu/~tkprasad/.../L12BUP.pdf - United States
63. www.csee.wvu.edu/~timm/cs310/cs310_12.html
64. www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/translat.htm
65. www.fact-index.com/c/co/compiler.html
66. www.facweb.iitkgp.ernet.in/~niloy/Compiler/.../TACintro.doc
67. www.facweb.iitkgp.ernet.in/~niloy/Compiler/notes/TCBB.doc
68. www.facweb.iitkgp.ernet.in/~niloy/Compiler/notes/PG.doc
69. www.iitg.ernet.in/dgoswami/resource/Code-Optimization.ppt
70. www-inst.eecs.berkeley.edu/~cs164/fa05/lects/f05-13-2x2.pd
71. www.mec.ac.in/resources/notes/notes/automata/enfa.htm
72. www.mec.ac.in/resources/notes/.../compiler/.../codegenissues.h...
73. www.mec.ac.in/resources/notes/notes/.../predective.htm
74. www.nationmaster.com/encyclopedia/Retargetable-compiler
75. www.oocities.org/wael_it2003/2.ppt
76. www.scribd.com/doc/40687350/CD-Shivani
77. www.site.uottawa.ca/~bochmann/.../Notes/.../Error-recovery.p...
78. www.slideshare.net/AdnoisAyyappa/unit8-16576062

INDEX

Note: *Page numbers followed by "f" and "t" denote figures and tables.*

A

Abstract syntax tree (AST), 260, 260*f*
Ambiguous, 94
Augmented grammar, 192
Automatic code generator, 24
Automatic parser generator, 23
AVL trees, 348–352
  deletion, 351–352
  insertion, 348–349
  left–left (LL), 350
  left–right (LR), 350–351
  lookup, 348
  right–left (RL), 349–350
  right–right (RR), 349

B

Back patching, 427
Backus-Naur form (BNF), 84–85
Bootstrapping, 20–21
  advantages, 22–23
  T-diagram, 21
Bottom-up parser, 175*f*
  canonical LR(1) parsers CLR(1)/LR(1), 209–215
    closure(I), 209–210
    CLR(1) grammar, 213–215
    constructing CLR(1) parsing table, 212–213
    creating canonical collection, 211–212
    goto (I,X), 210–211
  handle, 173–174
  LALR(1) parser, 215–222
  LR(0) parser, 197–204
    advantages and disadvantages, 199
    LR(0) grammar, 199
    shift-reduce parsing conflicts, 200–204
  LR grammar, 187
  LR parsers, 187–188
    error recovery, 224
  LR parsing algorithm, 188–191
  LR parsing table, construction, 191–197
    augmented grammar, 192
    closure(I), 193–194
    creating canonical collection, 195
    DFA construction, 195–197
    goto (I,X), 194–195
    LR(0) item, 192–193
  operator precedence parsing, 176–187
    error recovery, 184–185
    operator precedence relation, calculating, 182–184
    parsing algorithm for, 178–179
    precedence relations, 177
    precedence relation table, construction, 179–182
    precedence relation table to precedence function table, 184–185
    recognizing handles, 177–178
  SLR(1) parser, 204–209
  *vs.*top-down parser, 223–224
Brute force technique, 131–133, 132*f*–133*f*

C

Chomsky hierarchy, 81
  context sensitive grammars (CSG), 82
  recursively enumerable languages, 81
  regular grammars (RG), 82
  Unrestricted grammars (URG), 81
CLR(1)/LR(1) grammar, 213
Code generation, 425

approach to, 429–432
issues in design, 425–426, 426*f*
choice of evaluation order, 429
code generator input, 426
instruction selection, 427–428
memory management, 427
register allocation, 428–429
register allocation and assignment, 433–436
target programs, 426–427
using DAG, 436–439
Code optimization, 375–377
basic block identification, 378–380
DAG construction, 381–386
DAG representation of basic block, 381
flow graph, 380–381
function-preserving transformations
common sub-expression elimination, 386–389
constant propagation, 397
copy propagation, 389–394
dead-code elimination, 394–396
global flow analysis, 403–411
loop optimization, 397–403
machine-dependent optimization, 411–413
model, 378*f*
principle source, 386
Coercion, 302
Compiler, 1–2, 8
bootstrapping, 20–23
challenge, 3–4
design of passes, 19–20
design phases
back end, 17–19, 18*f*
code optimizer, 13–14
front end, 16–17, 17*f*, 18*f*
intermediate code generator, 13
lexical analysis, 10–13
symbol table manager and error handler, 15–16
target code generator, 14–15
design principle, application, 24–25
design tools, 23–24
history, 2–3
input and output, 2*f*
modern compilers, 24
retargeting, 20
typical language-processing system, 6–9
vs. interpreter, 4–6
Concrete syntax tree, 260, 260*f*
Context Free Grammars (CFG), 79, 82, 87–89, 88, 125, 241
applications of, 112
derivation, 89–91
Context sensitive grammars (CSG), 82
Copy propagation, 389–394
Cross compiler, 20–21

D

Data-flow engine, 24
Dead-code elimination, 394–396, 412–413
Derivation tree, 93–94
Deterministic finite automata (DFA), 13
Directed acyclic graph (DAG), 262–264, 312
algorithm for construction, 382–384
application of, 384–386
construction, 381–382
representation of basic block, 381

E

Early binding, 291
Encoding of type expressions, 298–299
Error handler, 15
Executable code, 6
Extended backus naur form (EBNF), 85–86

F

Finite automata (FA), 31
Finite state machine, 43–63
automaton model, 44
converting NFA (MN) to DFA (MD), 51–53
deterministic finite dutomaton (DFD), 48–49
eliminating ε-transitions, 55
epsilon closure (ε-closure), 54–55
equivalence of DFA and NFA, 50–51
language acceptance, 46–47
minimization of DFA, 59–63
NFA with epsilon (ε-transitions), 54

nondeterministic finite automaton (NFA), 49–50
transition diagram and table, 45–46
transition function, properties, 45
Function newtemp, 318–319
Function-preserving transformations
common sub-expression elimination, 386–389
constant propagation, 397
copy propagation, 389–394
dead-code elimination, 394–396

G

Global flow analysis, 403–404
live variable analysis, 405
points and paths, 404–405
reaching definition, 405
iterative algorithm for, 408–411
set representation, 406–408
use definition chains, 405
Global optimizer, 24
Grammar, 79–80
ambiguous, 100–103
attributes for symbols, 242–243
inherent ambiguity, 105–106
language defined by, 91–96
left-factoring, 98–100
left recursion, 96–98
LL(1), 145–150
parsing, 125
removing ambiguity, 103–105
representations, 84–87
Backus-Naur form (BNF), 84–85
extended Backus-Naur form (EBNF), 85–86
syntax diagrams, 86–87, 86*f,* 87*f*
simplification of, 106–111
string of symbols, 139–141
types—Chomsky hierarchy, 81–84

H

Handle, 173–174
pruning, 174
Hash tables, 352
chaining, 353–354
collision handling, 353
mapping function, 353
overflow area, 355–356
re-hashing, 354

I

Intermediate code, 309
benefits of, 309
generator role, 309*f*
Intermediate languages
directed acyclic graph, 312
postfix notation, 312–313
syntax trees, 310–311
three address code, 313–314
Intermediate representation (IR), 13, 17
Interpreter, 4, 4*f*
advantages of, 5–6

L

L-attributed, 269–276
L-attributed to S-attributed, converting, 276–278
Left-factoring, 98–100
Leftmost derivation (LMD), 92, 128
Left recursive, 96–98
Lexeme, 35
Lexical analysis, 10–13, 11*f,* 16, 31–64
advantages of separating from syntax analysis, 33
error recovery, 33–34
finite state machine, 43–63
input buffering, 38–40
lexical analyzer secondary tasks, 33
lex tool, 63–64
strategies for implementing, 37–38
tokens, patterns, lexemes, 34–37
tokens specification and recognition, 40–43
Loader/linker, 9–10
Loop optimization, 397–398
induction variables, 399–400
detecting, 400–401
elimination of, 401–403
strength reduction applied to, 401
loop invariant computation, 398–399
LR grammar, 187

M

Machine-dependent optimization, 411
  algebraic simplification, 412
  dead code elimination, 412–413
  flow-of-control optimization, 413
  redundant loads and stores, 411–412
  strength reduction, 413
  use of machine idioms, 413
  Myhill Nerode theorem, 61–63

N

Name equivalence, 299–300
Nondeterministic finite automata (NFA), 31

O

Operator precedence parsing, 176–187
  error recovery, 184–185
  operator precedence relation, calculating, 182–184
  parsing algorithm for, 178–179
  precedence relations, 177
  precedence relation table, construction, 179–182
  precedence relation table to precedence function table, 184–185
  recognizing handles, 177–178

P

Panic Mode Error Recovery, parsing, 127, 150–152
Parsers
  actions in top-down evaluation, 267*f*–268*f*
  error handling in, 126–127
  LR, 188–191, 189*f*
  predictive, 133–134
    algorithm for LL(1) parsing, 137–139
    construction of parsing tables, 144–145
    error recovery, 150–152
    non-recursive descent parser-LL(1) parser, 136–137, 136*f*
    recursive descent parser, 134–136
  types, 128*f*
    top-down parsers (TDP), 128–129, 129*f*
    universal parsers, 128
Parse tree, 243*f,* 244*f,* 245*f,* 246*f,* 247*f,* 249*f,* 250*f,* 253*f,* 254*f,* 255*f,* 256*f,* 258*f,* 259*f,* 262*f,* 263*f,* 266*f,* 272*f,* 277*f*
Pattern, 35
Peephole optimization, 411
Postfix notation, 312–313

R

Recursively enumerable languages, 81
Regular grammars (RG), 82
Rightmost derivation (RMD), 92

S

S-attributed grammar, top-down evaluation of, 265–268
Scanner. *See* Lexical analysis
Scanner generator, 23
Semantic analysis, 291–293
  equivalence of type expressions
  encoding of type expressions, 298–299
  name equivalence, 299–300
  structural equivalence, 297–298
  type graph, 300–301
  functions and operators overloading, 302–303
  polymorphic functions, 303
  role of a semantic analyzer in compilation, 292*f*
  simple type checker, design, 295–296
  type checking
    of expressions, 296
    of functions, 297
    of statements, 296–297
  type conversion, 302
  type expressions, 293–295
  type systems, 293
Shift/reduce conflict, 200–204
Shift reduce (SR) parser, 174–175, 174*t*
  See also Bottom-up parser
Source language, 2
Stack symbol tables, 358–363
Structural equivalence, 297–298

Symbol table, 10, 337–367, 338*t*
block structured language, 356–358
stack-implemented tree/hash-structured symbol tables, 363–367
stack symbol tables, 358–363
entries, 339–340
non-block structured language
AVL trees, 348–352
hash tables, 352–356
hierarchical list, 347–348
linear list, 342
linked list or self-organizing tables, 344–346
ordered list, 342–343
unordered list, 343–344
operations on, 340–341
organization, 341–342
Symbol table manager, 15
Syntax analysis, 125–126, 126*f*
Syntax diagrams, 86–87, 86*f*, 87*f*
Syntax-directed translation (SDT), 24, 241–284
bottom-up evaluation of, 250–260
into three address code, 318–327
addressing array elements, 320–322
assignment statement, 318–320
control statements, 324–327
logical expression, 322–323
postfix, 265
syntax tree, creation, 260–262
types, 264
writing, 243–250
Syntax tree, 260, 310–311

T

Target code generator, 14–15
Target/object language, 2
Three address code, 313–314
representation
comparison, 318
indirect triples, 317
quadruple, 316
triple, 316–317
types, 314–315
Token, 34
recognition, 42–43
specification, 40–41
Top-down parsers (TDP), 128–129, 129*f*
brute force technique, 131–133, 132*f*–133*f*
Translator, 2
Typical language-processing system, 6, 7*f*
loader/linker, 9–10
preprocessor, 6–9

U

Universal parsers, 128
Unrestricted grammars (URG), 81

Y

YACC (Yet Another Compiler Compiler), 278–284