Syllabus

Course Code	Course Name	Credit
CSC602	System Programming and Compiler Construction	4

Course Objectives :

1. To understand the role and functioning of various system programs over application program.
2. To understand basic concepts and designing of assembler, Macro processor and role of static and dynamic loaders and linkers.
3. To understand the need to follow the syntax in writing an application program and to learn the how the analysis phase of compiler is designed to understand the programmer's requirements without ambiguity.
4. To synthesize the analysis phase outcomes to produce the object code that is efficient in terms of space and execution time.

Course Outcomes : On successful completion of course learner will be able to:

1. Identify the relevance of different system programs.
2. Describe the various data structures and passes of assembler design.
3. Identify the need for different features and designing of macros.
4. Distinguish different loaders and linkers and their contribution in developing efficient user applications.
5. Construct different parsers for given context free grammars.
6. Justify the need synthesis phase to produce object code optimized in terms of high execution speed and less memory usage

Prerequisite : Data Structures, Theoretical computer science, Operating system. Computer Organization and Architecture, Microprocessor

Module No.	Unit No.	Topics	Hrs
1	**Introduction to System Software**	Concept of System Software, Goals of system software's, system program and system programming. Introduction to various system programs such as Assembler, Macroprocessor, Loader, Linker, Compiler, Interpreter, Device Drivers, Operating system, Editors, Debuggers. **(Refer chapter 1)**	2
2	**Assemblers**	Elements of Assembly Language programming, Assembly scheme, pass structure of assembler, Assembler Design: Two pass assembler Design and single pass Assembler Design for Hypothetical / X86 family processor, data structures used. **(Refer chapter 2)**	10

Module No.	Unit No.	Topics	Hrs
3	**Macros and Macro Processor**	Introduction, Macro definition and call, Features of Macro facility : Simple, parameterized, conditional and nested. Design of single pass macro processor, data structures used. **(Refer chapter 3)**	8
4	**Loaders and Linkers**	Introduction, functions of loaders, Relocation and Linking concept, Different loading schemes : Relocating loader, Direct Linking Loader, Dynamic linking and loading. **(Refer chapter 4)**	8
5	**Compilers : Analysis Phase**	Introduction to Compilers, Phases of Compilers : Lexical Analysis- Role of Finite State Automata in Lexical Analysis, Design of Lexical analyzer, data structures used. Syntax Analysis- Role of Context Free Grammar in Syntax analysis, Types of Parsers: Top down parser- LL(1), Bottom up parser- Operator precedence parser, SLR, Semantic Analysis, Syntax directed definitions. **(Refer chapter 5)**	12
6	**Compilers : Synthesis phase**	Intermediate Code Generation : Types of Intermediate codes : Syntax tree, Postfix notation, Three address codes: Triples and Quadruples. Code Optimization: Need and sources of optimization, Code optimization techniques: Machine Dependent and Machine Independent. Code Generation: Issues in the design of code generator, code generation algorithm. Basic block and flow graph. **(Refer chapter 6)**	12

❑❑❑

Lab Syllabus

Lab Code	Lab Name	Credit
CSL602	System Software Lab	1

Suggested List of Experiments :

Sr. No.	Title of Experiments
1.	Implementations of two pass Assembler.
2.	Implementation of single pass Macro Processor.
3.	Implementation of Lexical Analyzer.
4.	Implementation of Parser (Any one).
5.	Implementation of Intermediate code generation phase of compiler.
6.	Implementation of code generation phase of compiler.
7.	Study and implement experiments on LEX. YACC, Grey Box Probing.

❑❑❑

Scanned by CamScanner

Module 1

Chapter 1 : Introduction to System Software 1-1 to 1-20

Syllabus : Concept of System Software, Goals of system softwares, system program and system programming; Introduction to various system programs such as Assembler, Macro processor, Loader, Linker, Compiler, Interpreter, Device Drivers, Operating system, Editors, Debuggers.

1.1 Concept of System Software 1-1
- 1.1.1 System Software (**Dec. 2014, Dec. 2017**) 1-1
- 1.1.2 Programming Software 1-2
- 1.1.3 Application Software (**Dec. 2014**) 1-2
- 1.1.4 Difference between Application Program and System Program (**May 2015, Dec. 2015, Dec. 2017**) 1-3
- 1.1.5 Goals of System Software 1-4
- 1.1.6 Difference between System Program and System Programming 1-4

1.2 Introduction to Language Processors 1-5
- 1.2.1 What are Language Processors ? 1-6
- 1.2.2 Spectrum of Language Processors 1-6

1.3 Language Processing Activities 1-6
- 1.3.1 Program Generation 1-7
- 1.3.2 Program Execution 1-7

1.4 Introduction to Various System Programs (**Dec. 2015**) 1-7
- 1.4.1 Assemblers 1-8
- 1.4.1(A) Different Types of Assemblers 1-8
- 1.4.2 Loaders (**Dec. 2016**) 1-9
- 1.4.2(A) Functions of a Loader (**Dec. 2015, Dec. 2016**) 1-9
- 1.4.2(B) Types of Loader 1-10
- 1.4.3 Linkers 1-10
- 1.4.3(A) Various Functions Performed by a Linker 1-10
- 1.4.4 Macro Processors 1-11
- 1.4.4(A) Types of Macros 1-12
- 1.4.5 Compilers 1-12
- 1.4.5(A) Applications of Compiler 1-13
- 1.4.5(B) Phases of Compiler 1-13
- 1.4.6 Interpreters 1-14
- 1.4.6(A) Difference between Interpreter and Compiler (**May 2016**) 1-14
- 1.4.7 Operating Systems 1-14
- 1.4.7(A) Features of an Operating System 1-15
- 1.4.8 Device Drivers 1-15
- 1.4.8(A) Purpose of Device Drivers 1-15
- 1.4.8(B) Virtual Device Drivers 1-16

Scanned by CamScanner

1.4.9 Debug Monitor 1-16
1.4.10 Editors (**Dec. 2017**) 1-16
1.4.10(A) Types of Text Editors 1-16
1.4.10(B) Features of Text Editors 1-17
1.4.11 Debuggers 1-17
1.4.11(A) Features of Debuggers 1-17
1.4.11(B) Reverse Debugging 1-17
1.4.11(C) Various Debuggers 1-18
1.4.11(D) Comparison of various Debuggers 1-18
1.5 Binding (**May 2014**) 1-19
• **Chapter Ends** 1-20

Module 2

Chapter 2 : Assemblers ..... 2-1 to 2-32

Syllabus : Elements of Assembly Language programming, Assembly scheme, pass structure of assembler. **Assembler Design :** Two pass assembler design and single pass Assembler, Design for Hypothetical / X86 family processor, data structures used.

2.1 Introduction 2-1
2.1.1 Advantages of Assembly Language 2-1
2.1.2 Disadvantages of Assembly Language 2-1
2.2 Elements of an Assembly Language Programming 2-2
2.2.1 Basic Feature of an Assembler 2-2
2.2.2 Assembly Language Program Contains Three Kinds of Statements 2-3
2.2.3 An Assembly Language Program 2-4
2.3 Basic Assembler Functions 2-6
2.3.1 Essential Assembler Functions 2-6
2.3.1(A) Assembler Directives 2-6
2.4 Assembly Scheme 2-7
2.5 Pass Structure of Assembler 2-7
2.6 Assembler Design 2-8
2.6.1 Forward Reference Problem (**Dec. 2014**) 2-8
2.6.2 Phases used by the Assembly Language 2-9
2.7 Two Pass Assembler Design (**May 2014, May 2015, Dec. 2015, May 2016, Dec. 2016, Dec. 2017**) 2-10
2.8 One Pass Assembler (**Dec. 2014**) 2-21
• **Chapter Ends** 2-32

Scanned by CamScanner

Module 3

Chapter 3 : Macros & Macro Processor 3-1 to 3-22

Syllabus : Introduction, Macro definition and call, Features of Macro facility : Simple, parameterized, conditional and nested. Design of single pass macro processor, data structures used.

3.1 Introduction 3-1
3.2 Macro Definition and Call 3-1
3.2.1 Functions of MACRO Instructions 3-2
3.2.2 Example of Macro Instruction 3-4
3.3 Features of Macro Facility **(May 2014)** 3-5
3.4 Implementation of Macro Processors **(May 2014)** 3-16
3.5 Design of Single Pass Macro Processor 3-17
3.5.1 Data Structures used in One Pass Macro Processor 3-17
3.5.2 Algorithm Execution Steps 3-17
3.5.3 Format of Databases 3-18
3.6 Difference between MACROS and FUNCTIONS 3-20
3.6.1 Difference between Macro and Subroutine 3-20
3.7 Difference between Procedure Calls and Macro Calls 3-21
• **Chapter Ends** 3-22

Module 4

Chapter 4 : Loaders and Linkers 4-1 to 4-26

Syllabus : Introduction, functions of loaders, Relocation and Linking concept, Different loading schemes : Relocating loader, Direct Linking Loader, Dynamic linking and loading.

4.1 Introduction 4-1
4.1.1 Evolution of Linker and Loader 4-1
4.2 Linking vs. Loading 4-3
4.2.1 Difference between Linker and Loader 4-6
4.3 Functions of Loader **(Dec. 2014, May 2015)** 4-6
4.4 Relocation and Linking Concept 4-7
4.5 Different Types of Loaders 4-9
4.5.1 Compile and Go Loader 4-9
4.5.1(A) General Loading Scheme 4-10
4.5.2 Absolute Loader **(May 2014)** 4-11
4.5.2(A) Bootstrap Loader 4-12
4.5.3 Relocating Loader 4-13
4.5.4 Direct Linking Loader **(Dec. 2014, Dec. 2017)** 4-14
4.5.4(A) Design of Direct Linking Loader 4-14

Scanned by CamScanner

4.5.4(B) Example Program for Direct Linking Loader 4-15
4.5.4(C) Databases Required for Direct Linking Loader 4-18
4.5.4(D) Algorithm and Flowchart for Direct Linking Loader 4-19
4.5.5 Dynamic Loading Loader 4-21
4.5.6 Dynamic Linking Loader **(Dec. 2015)** 4-24
4.5.7 Difference between Dynamic Loading and Dynamic Linking **(May 2016)** 4-25
- **Chapter Ends** 4-26

Module 5

Chapter 5 : Compilers : Analysis Phase **5-1 to 5-184**

Syllabus : Introduction to compilers, Phases of compilers ;

Lexical Analysis : Role of Finite State Automata in Lexical Analysis, Design of Lexical analyser, data structures used. **Syntax Analysis :** Role of Context Free Grammar in Syntax analysis, Types of Parsers : Top down parser- LL(1), Bottom up parser - Operator precedence parser, SLR; **Semantic Analysis**, Syntax directed definitions.

5.1 Evolution of Compilers 5-1
5.2 Features of a Compiler 5-3
5.3 Compiler Requirements 5-4
5.4 Types of Compilers 5-4
5.4.1 One Pass Compiler 5-5
5.4.2 Multi-pass Compiler 5-6
5.5 Phases of Compiler **(Dec. 2015, May 2016, Dec. 2017)** 5-6
5.6 Description of Different Phases of Compilation **(Dec.2015, May 2016, Dec. 2017)** 5-9
5.6.1 Lexical Analysis 5-9
5.6.2 Syntax Analysis 5-10
5.6.3 Semantic Analysis 5-11
5.6.4 Intermediate Code Generation 5-12
5.6.5 Code Optimization 5-12
5.6.6 Code Generation 5-12
5.6.7 Symbol Table Management 5-13
5.6.8 Error Detection and Reporting 5-13
5.7 Example of Compilation **(Dec. 2017)** 5-14
5.8 Interpreter Vs Compiler **(May 2014)** 5-20
5.9 Lexical Analysis : Introduction 5-21
5.10 Role of Lexical Analyzers **(Dec. 2014)** 5-21
5.10.1 Features of Lexical Analyser 5-22
5.10.2 Role of Finite Automata in Lexical Analysers **(Dec. 2014, May. 2016, Dec. 2016)** 5-23
5.11 Issues in Lexical Analysis 5-24

Scanned by CamScanner

5.12 Basic Definitions 5-24
5.12.1 Lexemes 5-24
5.12.2 Tokens 5-25
5.12.3 Patterns 5-25
5.13 Design of Lexical Analyzer **(Dec. 2014, May 2015, Dec. 2017)** 5-26
5.13.1 Data Structure for Lexical Analyzer 5-28
5.13.2 Using Lex with Other Programming Tools 5-33
5.14 Introduction 5-33
5.15 Role of the Parser 5-35
5.16 Role of Context Free Grammars in Syntax Analysis 5-35
5.16.1 Notational Conventions for Context Free Grammars 5-38
5.16.2 Derivation 5-39
5.16.2(A) Leftmost Derivation 5-41
5.16.2(B) Rightmost Derivation 5-42
5.16.3 Derivation of Parse Trees 5-44
5.16.4 Ambiguity 5-47
5.16.5 Left Recursion **(May 2015)** 5-51
5.16.6 Left Factoring **(May 2015)** 5-53
5.16.7 Context Free Language (CFL) 5-54
5.17 Parsing 5-56
5.18 Type of Parsers 5-56
5.18.1 Parsing Table 5-57
5.19 Top Down Parsing 5-57
5.19.1 Recursive Descent Parser **(Dec. 2016)** 5-60
5.19.2 Predictive Parsers 5-65
5.19.3 LL-Parser 5-70
5.19.3(A) LL (1) Parser - Parser Actions 5-73
5.19.3(B) Construction of Predictive Parser Table using First and Follow 5-81
5.19.3(C) First Sets and Follow Sets **(Dec. 2014)** 5-82
5.19.3(D) Finding First and Follow Sets 5-84
5.19.4 LL (1) Parse Tables 5-92
5.19.4(A) LL (1) Grammar 5-100
5.19.4(B) A Grammar which is not LL (1) 5-103
5.19.4(C) Properties of LL (1) Grammar 5-103
5.19.4(D) Error Recovery Techniques in Predictive Parsing 5-104
5.20 Bottom-Up Parsing **(Dec. 2014, Dec. 2016)** 5-105
5.20.1 Shift-Reduce Parser 5-108
5.20.2 Operator Precedence Parser **(Dec. 2014, May 2015, Dec. 2017)** 5-111
5.20.2(A) Operator Precedence Parsing Algorithm **(Dec. 2014)** 5-114
5.20.2(B) How to Create Operator Precedence Relations **(Dec. 2014)** 5-115
5.20.2(C) Handling Unary Minus in Operator Precedence Parser 5-116

Scanned by CamScanner

5.20.2(D) Precedence Functions in Operator Precedence Parser 5-116
5.20.2(E) Algorithm for Constructing Precedence Functions 5-117
5.20.2(F) Advantages and Disadvantages of Operator Precedence Parsing 5-118
5.20.2(G) Error Recovery in Operator-Precedence Parsing 5-118
5.20.3 LR Parsers 5-119
5.20.3(A) The LR Parsing Algorithm 5-120
5.20.3(B) Construction of Action and Goto Table/Parser Table 5-121
5.20.3(C) Rightmost Parsers 5-121
5.20.3(D) LR Parse Tables 5-123
5.20.4 SLR Parser 5-145
5.20.5 LR (1) Parsers 5-152
5.20.6 LR (k) Parsers 5-159
5.21 Difference between LL Parser and LR Parser **(Dec. 2017)** 5-160
5.22 Difference between Top down and Bottom up Parsing 5-161
5.23 Semantic Analysis - Introduction **(May 2014, Dec. 2014, Dec. 2016, Dec. 2017)** 5-161
5.24 Syntax Directed Definitions **(May 2014, Dec. 2014, Dec. 2016, Dec. 2017)** 5-163
5.25 Attribute Grammar 5-166
5.25.1 Synthesized Attributes **(Dec. 2015, May 2016, Dec. 2016)** 5-167
5.25.2 Inherited Attribute **(Dec. 2015, May 2016, Dec. 2016)** 5-169
5.26 Evaluation Order of Syntax Directed Definitions 5-171
5.26.1 Dependency Graphs 5-172
5.26.2 Evaluation Order 5-174
5.27 Differentiate between Parse Tree and Syntax Tree 5-174
5.27.1 Uses of Syntax tree 5-175
- **Chapter Ends** 5-184

Module 6

Chapter 6 : Compilers : Synthesis Phase **6-1 to 6-67**

Syllabus : Intermediate Code Generation : Types of Intermediate codes - Syntax tree, Postfix notation, Three address codes : Triples and Quadruples. **Code Optimization :** Need and sources of optimization, Code optimization techniques : Machine Dependent and Machine Independent. **Code Generation :** Issues in the design of code generator, code generation algorithm. Basic block and flow graph.

6.1 Intermediate Code Generation-Introduction 6-1
6.1.1 Intermediate Representation (IR) Gives 6-5
6.1.2 Importance of Intermediate Representation 6-5
6.1.3 Advantages of using An Intermediate Language 6-6
6.1.4 Issues in Designing an Intermediate Representation 6-7
6.1.5 Multiple Level Intermediate Representations 6-7

Scanned by CamScanner

6.1.6 Generally used different forms of Intermediate Representation **(May 2014, Dec. 2017)** 6-7

6.2 Intermediate Languages **(May 2014, Dec. 2016)** 6-8

6.2.1 Graphical Representation 6-9

6.2.1(A) Syntax Trees 6-9

6.2.1(B) Directed Acyclic Graph (DAG) **(Dec. 2014)** 6-10

6.3 Three-Address Code **(Dec. 2014)** 6-11

6.4 Implementation of Three-Address Statements 6-15

6.4.1 Quadruples **(May 2015, May 2016)** 6-15

6.4.2 Triples **(May 2015, May 2016)** 6-17

6.4.3 Indirect Triples **(May 2015, May 2016)** 6-19

6.4.4 Comparison of Representation 6-19

6.5 Types of Three-Address Statements 6-20

6.6 Syntax-Directed Translation into Three-Address Code 6-20

6.7 Code Optimization-Introduction **(Dec. 2014, Dec. 2015, May 2016)** 6-22

6.7.1 Importance of Optimization 6-24

6.7.2 Is Programmer can Perform as Code Optimizer ? 6-25

6.7.3 Problems Faced in Optimizing Compiler 6-25

6.7.4 Criterion for Code Optimization 6-25

6.7.5 Getting Better Performance through Optimization 6-26

6.7.6 Factors Influencing Code Optimization 6-27

6.7.7 Aims behind the Optimization Techniques 6-27

6.8 Principal Sources of Code Optimization **(Dec. 2014, May 2016)** 6-28

6.8.1 Function Preserving Transformations 6-28

6.8.1(A) Compile Time Evaluation 6-29

6.8.1(B) Common Sub-Expression Elimination 6-29

6.8.1(C) Dead Code Elimination 6-30

6.8.1(D) Copy Propagation 6-31

6.8.2 Loop Optimizations **(May 2015, Dec. 2016)** 6-32

6.8.2(A) Code Motion **(May 2015)** 6-32

6.8.2(B) Strength Reduction 6-34

6.8.2(C) Induction Variable Strength Reduction 6-34

6.8.2(D) Loop Induction Variable 6-36

6.8.2(E) Loop Fission 6-36

6.8.2(F) Loop Fusion or Loop Combining 6-37

6.8.2(G) Loop Interchange 6-37

6.8.2(H) Loop Unrolling 6-37

6.8.2(I) Loop Splitting 6-37

6.8.3 Optimization for Basic Blocks 6-38

6.8.3(A) Common Sub-Expression Elimination 6-38

6.8.3(B) Dead Code Elimination 6-44

Scanned by CamScanner

6.8.3(C) Algebraic Identities Simplification 6-44
6.8.4 Reducible Flow Graphs 6-44
6.8.5 Global Optimizations 6-44
6.8.5(A) Control Flow Analysis 6-44
6.8.5(B) Data Flow Analysis 6-45
6.8.6 Machine Dependent / Machine Independent Optimization 6-45
6.8.6(A) Machine Dependent Optimizations 6-46
6.8.6(B) Machine Independent Optimization 6-48
6.9 Peephole Optimization **(Dec. 2015)** 6-49
6.10 Local Optimization 6-49
6.11 Code Generation-Introduction 6-50
6.12 Issues in the Design of Code Generator **(May 2015, May 2016, Dec. 2017)** 6-51
6.13 Code Generation Algorithm 6-52
6.14 Basic Block and Flow Graphs **(May 2015)** 6-55
6.14.1 Control-Flow Analysis 6-55
6.14.2 Partition Algorithm for Basic Block 6-56
6.15 Directed Acyclic Graphs 6-60
6.16 DAG Representation of Basic Block 6-64
- **Chapter Ends** 6-67
- **Lab Manual** ..... L-1 to L-46

❑❑❑

List of Experiments

Experiment No.	Name of the Experiment	Page No.
1.	Design of an Assembler (PASS-1).	L-1
2.	Implementation of a design of an Assembler (PASS-2).	L-8
3.	Implementation of Macro-Processors.	L-16
4.	Implementation of Lexical Analysis.	L-20
5.	Implementation of Shift Reduce Parser.	L-24
6.	Intermediate Code Generation.	L-27
7.	Implementation of Code Optimization.	L-33
8.	To convert NFA to DFA.	L-36

❑❑❑

Scanned by CamScanner

Module 1

CHAPTER 1

Introduction to System Software

Syllabus Topics

Concept of System Software, Goals of system softwares, system program and system programming; Introduction to various system programs such as Assembler, Macro processor, Loader, Linker, Compiler, Interpreter, Device Drivers, Operating system, Editors, Debuggers.

1.1 Concept of System Software

Computer instructions or data is anything that can be stored electrically is software. The storage devices and display devices are hardware.

The distinction between software and hardware is sometimes confusing because they are so integrally linked. System programs (e.g. compilers, loaders, macro-processors and operating systems) were developed to make computers better adapted to the needs of their users.

Computer system divides the system software's into three major categories:

1. System Software
2. Programming Software
3. Application Software

1.1.1 System Software

→ (MU - Dec. 2014, Dec. 2017)

Q.	What is system software ?	Dec. 2014, 5 Marks
Q.	Define system programming.	Dec. 2017, 2 Marks

Scanned by CamScanner

It consists of a variety of programs that supports the operation of a computer. **System software** or system program is a computer program that provides the infrastructure over which programs can operate. This software makes it possible for the users to focus on an application or other problems to be solved, without needing to know the details of how machine works internally. System software is the combination of the following :

- Device Drivers
- Operating Systems
- Utilities
- Windowing System.
- Servers

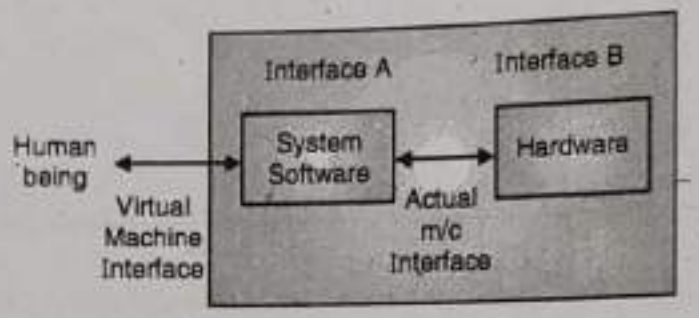


Fig. 1.1.1 : The Role of System Software

1.1.2 Programming Software

It usually provides tools to assist a programmer in writing computer programs, and software using different programming languages.

The programming software tool includes

- Compilers
- Debuggers
- Interpreters
- Linkers
- Text Editors

An Integrated Development Environment (IDE) is a single application that attempts to manage all these functions.

1.1.3 Application Software

→ **(MU - Dec. 2014)**

Q. What is application software? **Dec. 2014, 5 Marks**

Application Software is any tool that functions and is operated by means of a computer and allow end users to accomplish one or more specific (not directly computer development related) tasks. Typically application software includes :

- Industrial Automation
- Office Suites (e.g. : Microsoft office).
- Business Software's
- Computer Games
- Databases
- Medical Software's
- Media Players etc.

Scanned by CamScanner

As shown in Fig. 1.1.2 shows the layered structure of Software and Hardware.

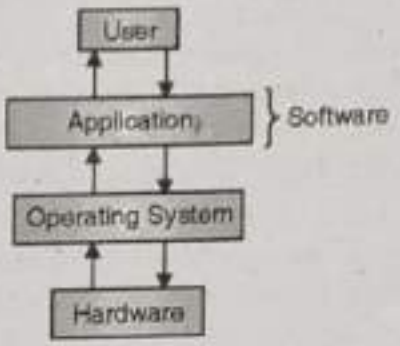


Fig. 1.1.2 : A Layered structure showing where operating system is located generally

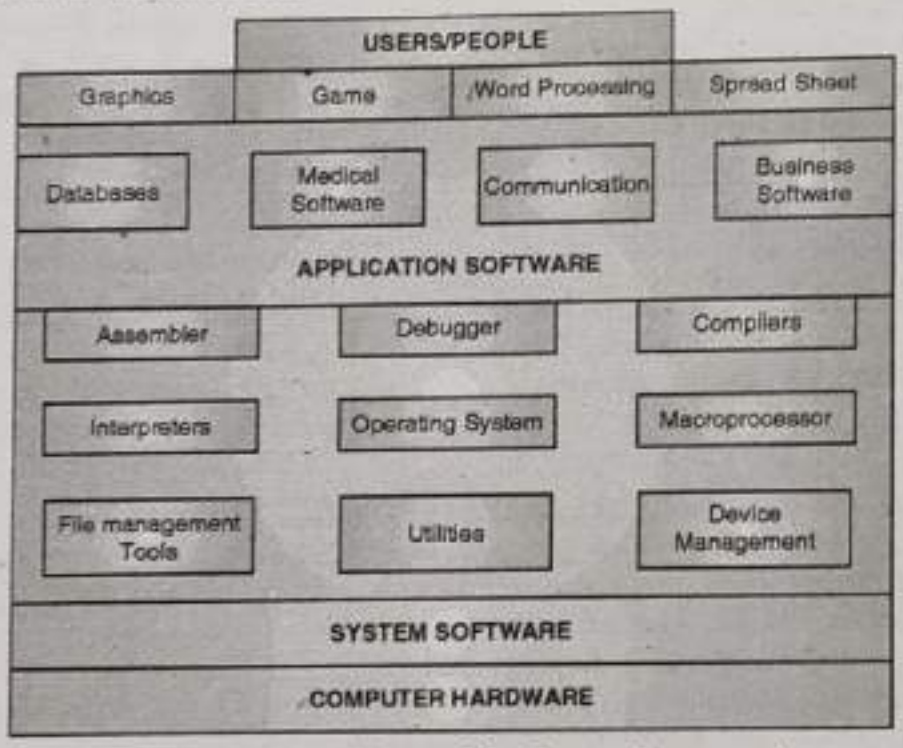


Fig. 1.1.3 : Foundations of the System Programming

Fig. 1.1.3 shows the various components of the application and the system software which together performs productive tasks for the users. System programming is the activity of programming system software. The primary distinguishing characteristic of system program when compared to application programming is that application programming aims to produce software which provides services to the users whereas system programming aims to produce software which provides services to the computer hardware.

1.1.4 Difference between Application Program and System Program

→ **(MU - May 2015, Dec. 2015, Dec. 2017)**

Q. Differentiate between a system program and an application program with examples. **May 2015, Dec. 2015, Dec. 2017, 5 Marks**

Q. State difference between application programs and system programs. **Dec. 2017, 3 Marks**

Scanned by CamScanner

Sr. No.	System Program	Application Program
1.	System software is any computer software which manages and controls computer hardware so that application software can perform a task.	Application software, are programs that enable the end-user to perform specific, productive tasks, such as word processing or image manipulation.
2.	Example : Operating systems, such as Microsoft Windows, Mac OS X or Linux, Compilers, Linkers etc.	Example : Microsoft Word, Oracle, Paint brush, spread sheets, anti-virus tools etc.
3.	The function of the systems software is to manage the resources of the computer, automate its operation and facilitate program development. It is generally provided by the computer manufacturer or a specialized programming firm	Application software's are designed to perform specific data processing or computational tasks for the user. These programs are specifically designed to meet end-user requirements.
4.	System software is the Operating System which handles devices, etc.	Application Software is third party software that seeks help from the system software
5.	System software or Operating System is the software that allows the computer to boot.	Application software are programs that run inside of or on top of the Operating System and allows you to do things like email or word processing.
6.	System software performs tasks like transferring data from memory to disk, or rendering text onto a display device. Specific kinds of system software include loading programs, operating systems, device drivers, programming tools, compilers, assemblers, linkers, and utility software.	Application software are used to meet the user specific needs.

1.1.5 Goals of System Software

System software provides the important functionality to users by not writing the different programs to solve their problems. It provides the platform to the user from where different software's can be generated as per the user's choice.

1.1.6 Difference between System Program and System Programming

System Program	System Programming
System programs are computer programs that provide platform for **application** programs to work.	A **system programming** language is basically referred to a **programming** language designed for writing **system** software, which usually requires different development approaches when compared with application software.

Scanned by CamScanner

1.2 Introduction to Language Processors

Language processors comprises of assemblers, compilers and interpreters. Programmers write software's in a variety of languages. Compilers and interpreters translate these programming languages into instructions that are understood by the computer at machine level. Compilers and Interpreters are contained within the 'Systems services' category of software and they provide a facility for the execution of high level programs (programs such as C and BASIC).

Language processing activities occur due to difference between how software is made by the programmer and how it is implemented by the computer. The software programmer mentions the two domains i.e. application domain to present the ideas and execution domain for the carrying of these ideas.

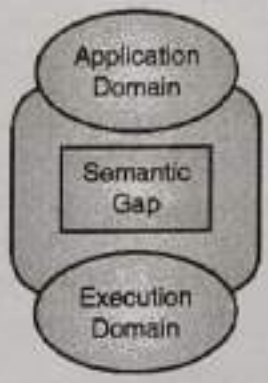


Fig. 1.2.1 : Semantic Gap between Application domain and Execution Domain

- The word semantics gives the set of rules used in different domains.
- Semantic Gap is the gap between the application and execution domains. Semantic gap takes a large development time, large efforts, and poor quality of software.
- **Programming Language** *(PL's)* domain is used to bridge the gap between the application domain and the execution domain.
  - The gap between the application and PL domain is given by Specification gap and is the space between the **two specifications** of the **same** task. This type of task is done by software development team.
  - The gap between the PL and execution domain is given by **Execution Gap.** It is the gap between the semantics of two programs written in different languages.
- Each domain has its own specification language to execute its tasks.

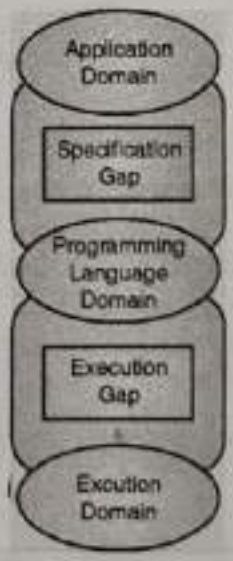


Fig. 1.2.2 : Various Programming Language Domain

Scanned by CamScanner

Need of Language Domain

1. It helps in minimizing the errors caused by the semantic gap.
2. It helps in bridging the gap between the different domains and find out if there are any errors in the input of any domain.

1.2.1 What are Language Processors ?

A Language Processor (LP) **is a program that performs tasks, such as translating and interpreting, required for processing a specified programming language.** It also bridges the specification or execution gap. The program form input to a LP is *source program* and to its output as the *target program*.

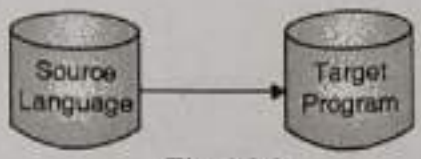


Fig. 1.2.3

1.2.2 Spectrum of Language Processors

- **System programming :** System programming aims to produce software which provides services to the computer hardware.

  A spectrum of language processors is defined as :

- **Language Translator :** It bridges an execution gap between PL domain and execution domain e.g. Assembler, Compiler.
- **Detranslator :** Same as language translator but used in reverse direction i.e. from execution domain to PL domain.
- **Pre-processor :** It is also used to bridge the gap of the execution domain. Language processor converts a C++ program into a C program, hence it is a pre-processor.

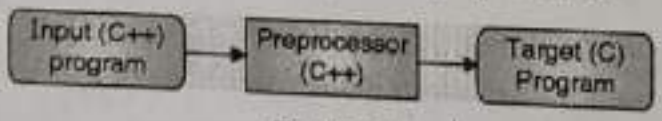


Fig. 1.2.4

- **Language Migrator :** It fills the specification gap between two PL's.

1.3 Language Processing Activities

The *Language Processing Activities* can be divided into those that bridge the specification gap and those that bridge the execution gap. There are two types of such activities :

- Program Generation Activities
- Program Execution Activities

Scanned by CamScanner

1.3.1 Program Generation

It generates the target program in machine language by accepting the specification of the source program.

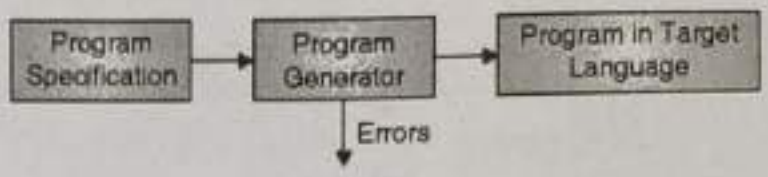


Fig. 1.3.1 : Program Generation Phase

1.3.2 Program Execution

Program execution phase is divided into two parts :

1. **Program Translation :** This activity is used to convert the source program into machine code. Features of the program translation are as :
   - Translation of program should be done before it is executed.
   - Translated program should be saved for repetitive execution.
   - Retranslation of the program should be done if there are any changes.

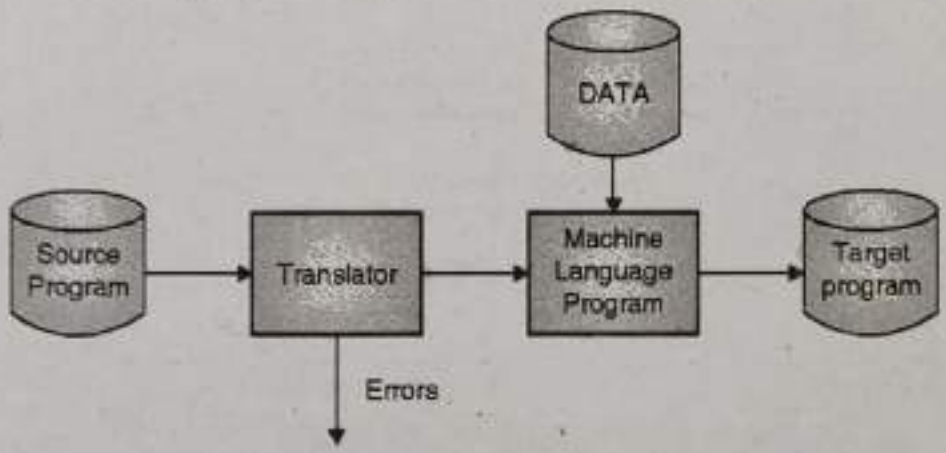


Fig. 1.3.2 : Program translation structure

2. **Program Interpretation :** Interpretation of a input program finds its meaning and implement actions based on its meaning and after that stores in memory for further execution. Interpretation of program consists of following steps :
   1. The code is fetched
   2. Analyse the code.
   3. Execute the meaning of the code.

1.4 Introduction to Various System Programs

→ **(MU - Dec. 2015)**

Q. Indicate the order in which the following system programs are used, from developing program upto its execution: Assemblers, loaders, linker, microprocessor, compiler, editor.

Dec. 2015, 5 Marks

Q. What is system programming? List some system programs and write their functions. **(5 Marks)**

Scanned by CamScanner

The order in which following system program are used from developing program upto its execution are as :

Editor → Macroprocessor → Assembler → Linker → Compiler → Loader

Developing----------→ Execution

1.4.1 Assemblers

Q. Write assemblers functions. (5 Marks)

An assembler creates object code by translating assembly instructions mnemonics into op-codes, and by resolving symbolic names for memory locations and other entities.

In other words we can say that an assembler is a program that accepts input as assembly language program and obtain its machine language equivalent code along with information for the loader. An assembler is a translator of a simple language. Fig. 1.4.1 shows the basic function of an assembler and a Fig. 1.4.2 shows program translation using assembler.

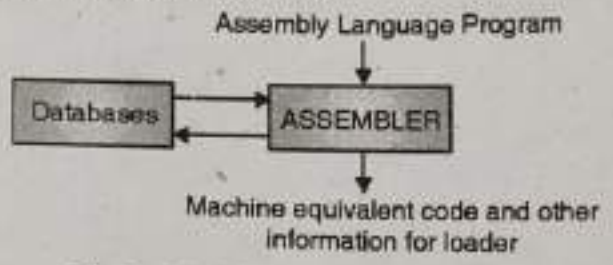


Fig. 1.4.1 : Function of an assembler

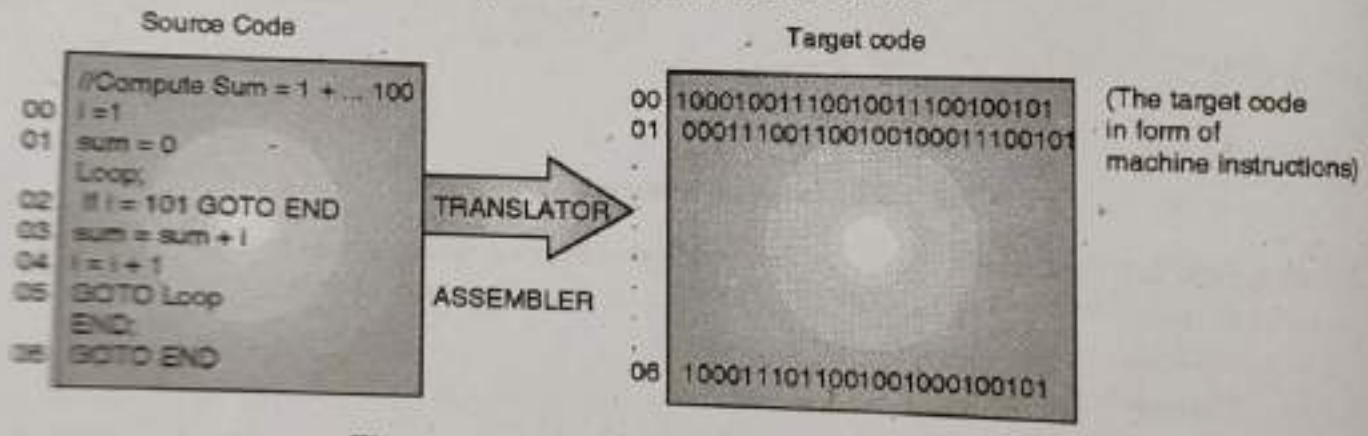


Fig. 1.4.2 : Program Translation using Assemblers

1.4.1(A) Different Types of Assemblers

There are three types of assemblers

1. ***Single Pass Assembler*** : It requires a single pass to assemble the program into machine code.
2. ***Two Pass Assembler*** : It requires two passes to convert the assembly language program into machine equivalent code.

Scanned by CamScanner

3. **Multipass Assembler :** It may require single or a multiple passes as per the requirement of the assembly language.

   Example of ASSEMBLERS SPARC ASSEMBLERS, MASM ASSEMBLERS MIPS ASSEMBLERS etc.

1.4.2 Loaders

→ (MU - Dec. 2016)

Q. What is Loader ? Explain functions of loader with examples. Dec. 2016, 10 Marks

Loading of the program in the memory for the purpose of execution is done by the loader. A loader is a program that accepts the object program and prepares these programs for execution by loading them into the memory.

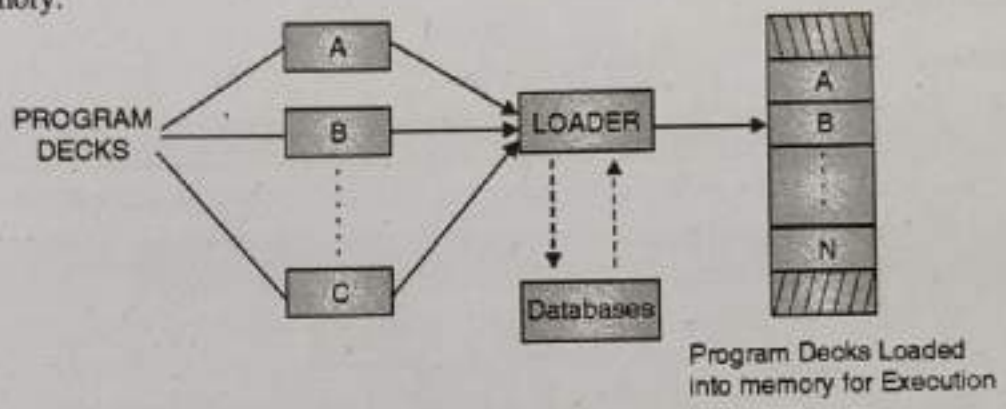


Fig. 1.4.3 : General loading structure

1.4.2(A) Functions of a Loader

→ (MU - Dec. 2015, Dec. 2016)

Q. What are different functions of loader? Dec. 2015, Dec. 2016, 10/5 Marks

The loader mainly performs the following four functions. Fig. 1.4.4 shows the four functions of the loader.

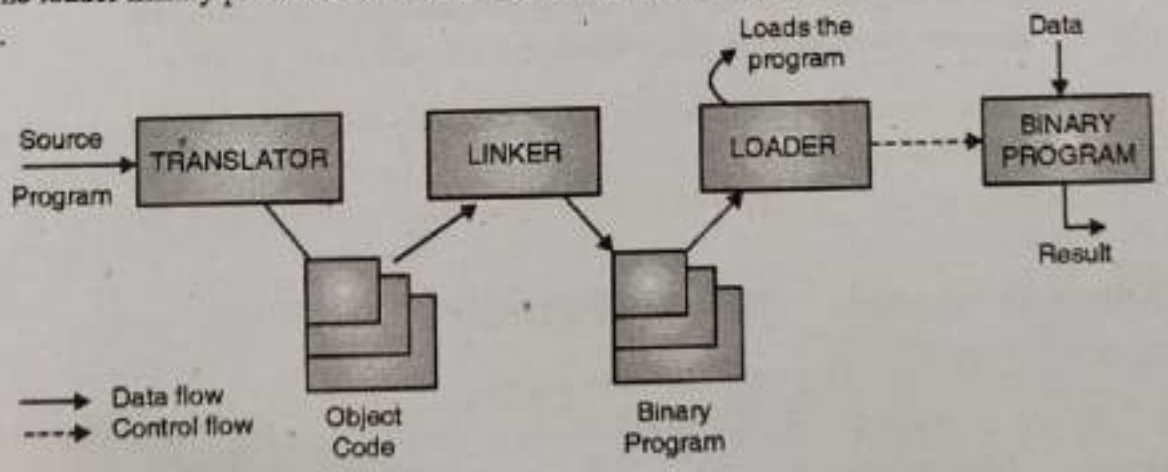


Fig. 1.4.4 : Basic Structure of Program Execution

- **ALLOCATION :** It uses to allocate space in the memory for the programs.
- **LINKING :** It links two or more separate object codes and provides the information needed to allow reference between them.

Scanned by CamScanner

- **RELOCATION :** It modifies the object program by changing certain instruction, so that it can be loaded at an address different from the location originally specified.
- **LOADING :** It brings the object program into the memory for execution.

1.4.2(B) Types of Loader

There are various types of loader :

- Compile and Go Loader/Assemble and Go loader.
- Absolute Loader (Bootstrap loader)
- Relocating Loader
- Direct Linking Loader
- Dynamic Linking loader.

1.4.3 Linkers

Q. What are different functions of linkers? (5 Marks)

A linker or a link editor is a program that takes one or more objects generated by a compiler or assemblers and combines them into a single executable program. Relocation and linking of all object modules is done by linker.

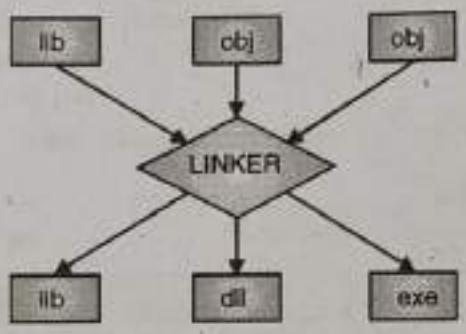


Fig. 1.4.5 : An illustration of the linking process (object files and static libraries are assembled into a new library or executable program)

E.g. In IBM mainframe environments such as OS/360 this program is known as linkage editor, and MS-DOS LINKER for Intel 8088/8086 processors.

When a program comprise of multiple object files, the linker combines these files into a unified executable program.

1.4.3(A) Various Functions Performed by a Linker

- Linkers can take objects from a collection called a **library.**
- The linker also takes care of arranging the objects in a programs address space.
- It performs complex function of linking.
- Linker passes linked modules for allocation and reallocation through the module loader.
- Only references to external data or programs are handled by linkage editor.
- Fig. 1.4.6 shows the flow diagram of a Linker.

Scanned by CamScanner

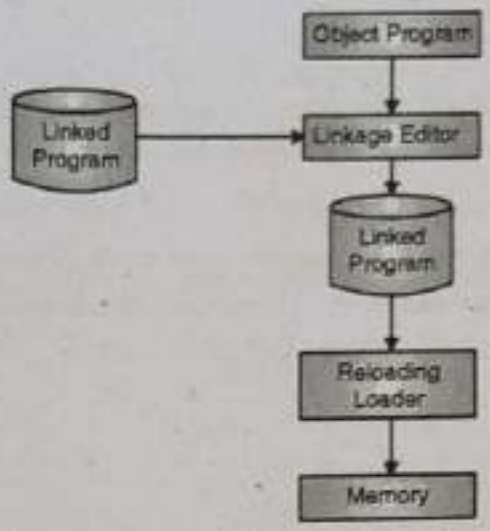


Fig. 1.4.6 : Flow diagram of LINKER

1.4.4 Macro Processors

Q. What are different functions of Macro processors ? (5 Marks)

Macro (meaning "large" or "wide"). A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the interpreter or compiler by the contents of the macro. The purpose of macros is to either automate frequently used sequences or enable a more powerful abstraction.

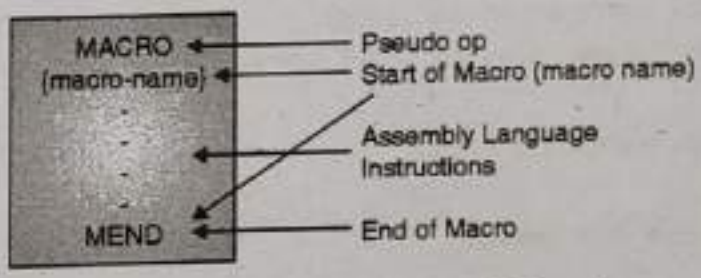
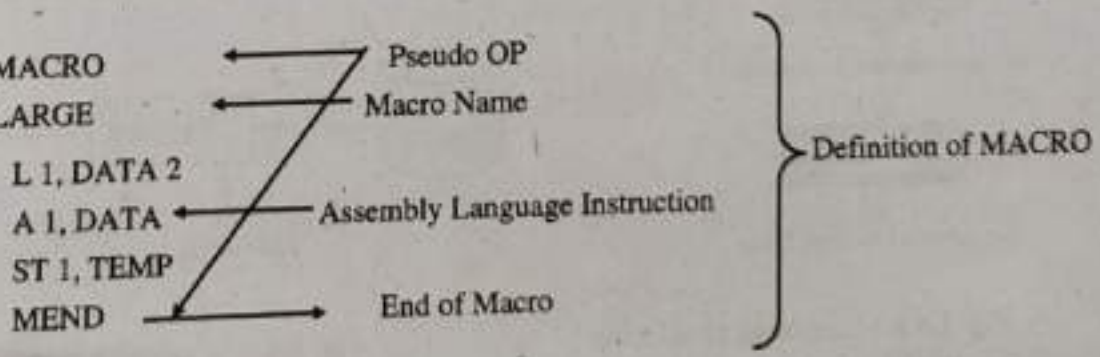


Fig. 1.4.7 : Basic structure of Macro's

"Macro Instructions (macros) are single-line abbreviation for a group of instructions shown in Fig. 1.4.7.

E.g.



Scanned by CamScanner

1.4.4(A) Types of Macros

There are two kinds of macro.

- **Object like Macros :** An object like macros is a simple identifier. It is commonly used to give symbolic names to numeric constants e.g. #define No 1, #define P1 3.14.

  or # define <identifier> (<replacement token list>)
- **Function like Macros :** We can define macros whose use look like a function call.

  e.g. # define <identifier> (<parameter list>) <replacement token list>

  e.g.

```
# define CUBE (X) ( (X) * (X) * (X) )
main ( )
{
    Volume = CUBE (atb) ;
}
// This will expand to Volume = ((atb)* (atb)* (atb));
```

1.4.5 Compilers

Q. What are different functions of compilers ? (5 Marks)

A compiler is a computer program that transforms source code written in a computer language (the source language) into another language (target languages often having a binary form known as object code). The most common reasons for having transform source code is to create an executable program.

A compiler translates a program written in high level language to generate assembly code or relocatable machine code (object code) Fig. 1.4.8 shows the language processing model using compilation.

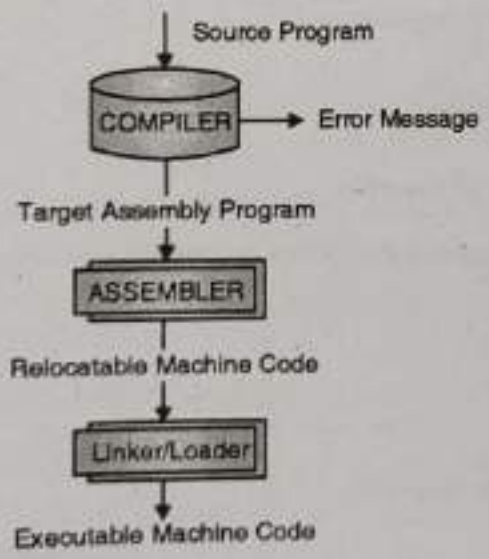


Fig. 1.4.8 : Language processing model using compilation

Fig. 1.4.9 : Compiler Input-Output

Scanned by CamScanner

1.4.5(A) Applications of Compiler

- Machine code generation
- Formal converters
- Silicon compilation
- Query interpretation
- Text formatting

1.4.5(B) Phases of Compiler

To ease the process of development and understanding, a complier can be divided into a number of phases, mainly.

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Target Code Generation
- Code Optimization
- Symbol table Management
- Error handling and Recovery.

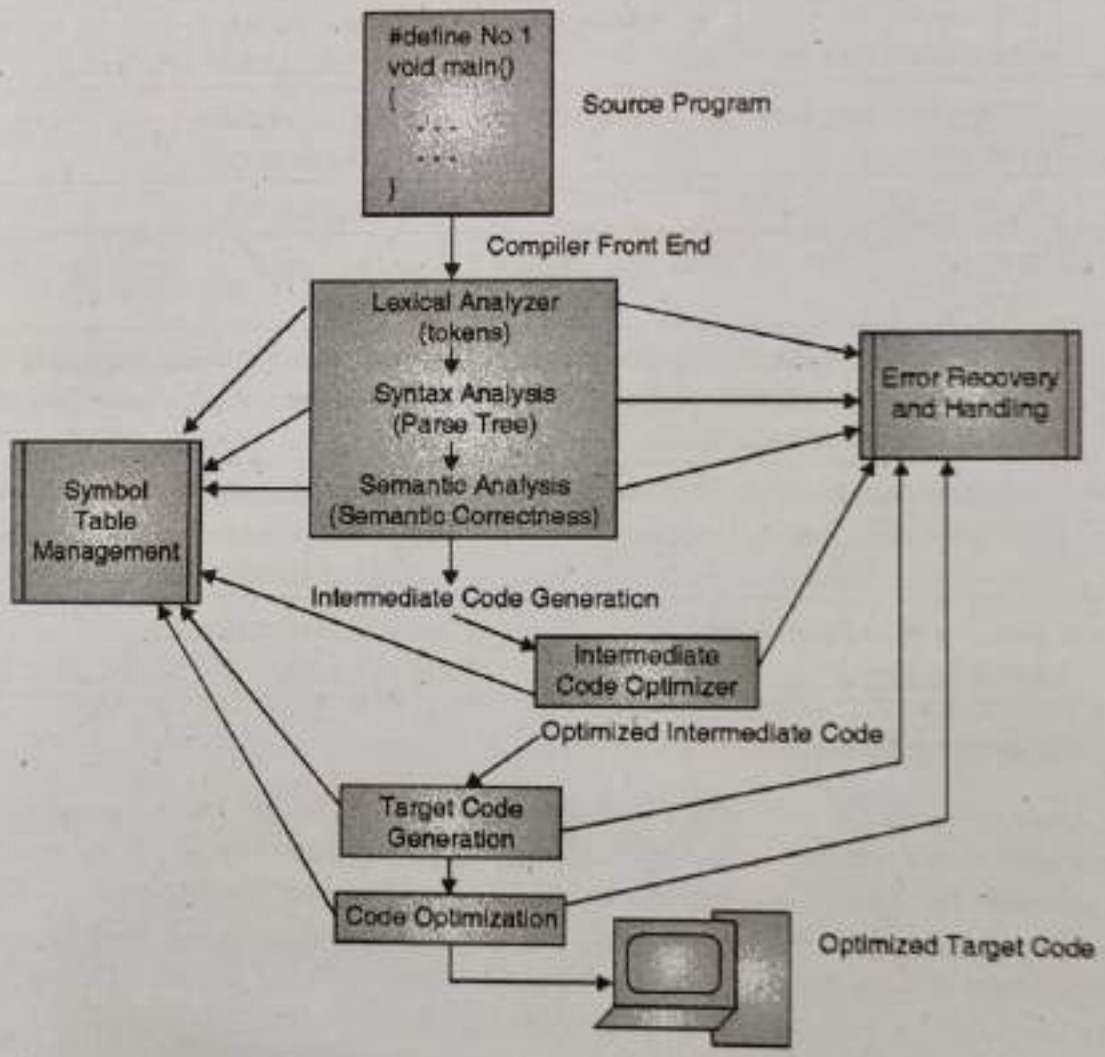


Fig. 1.4.10 : Phases of compiler

Scanned by CamScanner

1.4.6 Interpreters

Q. What is function of an interpreter ? **(5 Marks)**

An interpreters translates high level instructions into an intermediate form. An interpreter may be program that either,

- Executes the source code directly.
- Translates source code into some efficient intermediate representation and executes them.
- E.g. of Interpreter is LISP, BASIC etc.

1.4.6(A) Difference between Interpreter and Compiler

→ (MU - May 2016)

Q. What is the difference between interpreter and compiler ? **May 2016, 5 Marks**

Sr. No.	Interpreter	Compiler
1.	It translates the high level language into intermediate form line-by line.	It compiles the whole program at one time directly into machine language.
2.	Interpreters are slow in comparison to compiler.	Compiled programs run faster than interpreted program.
3.	Less compact.	More compact.
4.	Interpreted code is good for simple applications.	Compiled code is good for complex application.
5.	The advantage of line by line interpretation is that, it can interrupt in between can change the code and start again.	Translation of the whole program is done at a one time resulting in faster execution of program.
6.	The disadvantage is that every line has to be translated every time it is executed.	The disadvantages are that we cannot change the program without going back to the original source code.
7.	Example of Interpreters are JAVA SCRPIT, LISP, FORTH, BASIC	Example of compilers are VISUAL BASIC, C, C++, C#, COBOL etc.

1.4.7 Operating Systems

- An operating system is an interface between hardware and the user. An operating system is responsible for the management and co-ordination of activities and the sharing of the resources of the computer. Fig. 1.4.11 shows the basic architecture of an operating system.
- Operating system offers a number of services to application program and users. Applications access these services through application Programming Interfaces (API's) or **System calls**. E.g. of OS, BSD, GNU/LINUX, SUNOS (Solaris/Open Solaris) and WINOWS (NT/XP/VISTA).

Scanned by CamScanner

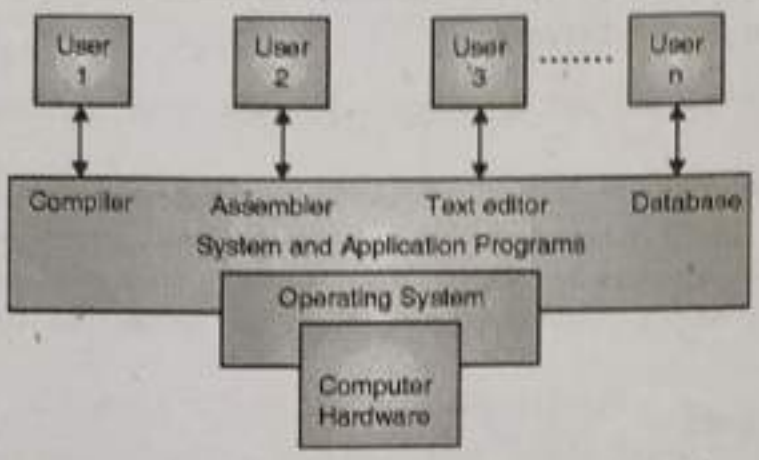


Fig. 1.4.11 : Basic Architecture of an Operating System

1.4.7(A) Features of an Operating System

1) Program Execution
2) Interrupt Handling
3) Memory Management
4) File Management
5) Device Management
6) Kernel Pre-emption
7) Virtual Memory
8) Networking
9) Security
10) Multitasking
11) Efficient Resource Allocation
12) Deadlock Detection and Error Detection.

1.4.8 Device Drivers

Device driver or driver or a hardware driver is a collection of software files that helps one or more hardware devices to communicate with the operating system. There are device drivers for the CD-ROMs, printers, diskettes, mouse, displays and scanner etc.

- Computer bus is used by the driver to communicate with the hardware device to which it is connected.
- Device drivers will differ from system to system i.e. they are hardware and operating system dependent.
- The operating system "calls" the driver, and the driver "drives" the device. When routine is invoked by a calling program in the driver, the driver issues commands to the device. Once the device sends data back to the driver the driver may invoke routines in the original calling program.
- The driver is the link between the operating system and the peripheral device.
- The device drivers present a uniform device-access interface to the I/O subsystem.

1.4.8(A) Purpose of Device Drivers

- A device driver simplifies programming by acting as an abstraction layer between a hardware device and the applications or operating system that uses it.
- Device drivers can be abstracted into logical and physical layers. Logic layers process data for a class of devices such as Ethernet ports or disk devices.
- A physical layer communicates with a specific device instances for e.g. serial port,

Scanned by CamScanner

1.4.8(B) Virtual Device Drivers

Virtual Device Drivers are used to emulate a hardware device, particularly in virtualization environments.

For e.g. when a DOS program is run on Microsoft windows computers or when a guest operating system is run on, for e.g. **XEN** host, instead of enabling the guest operating system to dialog with hardware Virtual device drivers take opposite role and emulate a piece of hardware so that the guest operating system and its drivers running inside a virtual machine can have the illusion of accessing real hardware.

1.4.9 Debug Monitor

- A debug monitor is very powerful graphical or console mode tool that monitors all the activities that are handled by the win driver kernel. With the help of the debug monitor we are able to view execution of all the commands sent to the kernel.
- A debug monitor is a tool that helps to find and reduce the number of bugs and defects in a computer program in order to make it at act the way it should.
- If the debugging monitor locates a bug or defect in any of the equipment, it will first try to reproduce the problems which will allow a programmer to view each string that was within the bug or defect range and try to fix it.

1.4.10 Editors

Q. Explain different type of text editors in brief. → **(MU - Dec. 2017)**

Dec. 2017, 5 Marks

A computer program used to edit the text is known as editors or text editors. These programs are provided with operating systems and software development packages. The various tasks done by the editors are :

- Editing configuration files
- Editing documentation files
- Editing programming language source code.

1.4.10(A) Types of Text Editors

- **Microsoft Windows :** Notepad, Word Pad.
- **Unix :** Pico, Vi and Emacs editors.
- **Mac OS :** SimpleText and TextEdit.
- World Wide Web offers a variety of HTML editors for creating web pages.
- Source Code Editors for writing the source code of various languages.
- Tex or LaTex used by various mathematicians, physicists, scientists etc.

Scanned by CamScanner

1.4.10(B) Features of Text Editors

1. Cut, copy and paste of texts between the document and files.
2. Find and replace.
3. Text formatting
4. Redo and undo
5. **Macro or procedure definition :** It helps to write the new commands or features as combinations of prior commands or other macros.
6. **Multi-file editing :** It helps in editing the multiple files during an edit-session.
7. **Data transformation :** Reading or merging the contents of another text file into the file currently being edited.
8. **Filtering :** It allows the editor to send all or sections of the file being edited to another utility and read the result back into the file in place of the lines being "filtered". Filtering is useful in sorting of data alphabetically or numerically, doing mathematical computations, source code indentation etc.
9. **Syntax Highlighting :** It highlights source code, markup languages, config files and other text in predictable format.

1.4.11 Debuggers

- **Debugging** is a process to identify and resolve the defects and problems in the computer program that helps in the execution of the correct program or software.
- A **debugger** is a tool that is used to test or debug the target program.
- The code to be tested might run on instruction set simulator, a program that has ability to halt when specific conditions are encountered. A trap forms when program cannot normally execute due to programming bug or invalid data. For example syntax error or semantic error or program might tried to use an instruction not available on current version of the CPU etc.

1.4.11(A) Features of Debuggers

1. Query processor
2. Symbol Resolver
3. Expression interpreter
4. Helps in running program step by step
5. Stopping of program by using breakpoint
6. Tracking the values of variables
7. Performance analyser
8. Fault coverage

1.4.11(B) Reverse Debugging

As name suggests these debuggers make it possible to step a program's execution backwards in time. It is also known as **backward debugging** or **historical debugging**. For example Microsoft Visual Studio (2010 Ultimate edition, 2012 Ultimate, 2013 Ultimate, and 2015 Enterprise edition) offers IntelliTrace reverse debugging for C#, Visual Basic .NET, and some other languages, but not C++. Reverse debuggers also exist for C, C++, Java, Python, Perl etc.

Scanned by CamScanner

1.4.11(C) Various Debuggers

1. Eclipse IDE (Java)
2. Nodeclipse (Java Script)
3. Microsoft Visual Studio Debugger
4. Firefox JavaScript Debugger
5. Radare 2
6. Valgrind
7. WinDbg

1.4.11(D) Comparison of various Debuggers

List of various debuggers that are used to test and debug other programs.

Name	Description	Language	OS
adb	Unix debugger	Any compiled to machine code	Unix
GDB	GNU Debugger	Any compiled to machine code	Unix-like systems, Windows
IDB	Intel Debugger	Any compiled to machine code	Windows, Linux, OS X
mdb	Solaris standard debugger (adb) successor	Any compiled to machine code	Solaris
TotalView	Source code and memory serial and parallel debugger	C++, C, CUDA, FORTRAN, MPI, OpenMP	Linux, AIX, Solaris, OS X, Cray, Blue Gene
Valgrind	tool suite for debugging and profiling Linux programs	C, C++, Java, Perl, Python, assembly code, Fortran, Ada, etc.	Linux, Darwin (Mac OS X), Android
Visual Studio Debugger	Debugger in Microsoft Visual Studio	C++, JavaScript, netlanguages	Windows 7, Windows Server 2008 R2, Windows 8, Windows Server 2012, Windows 8.1, Windows Server 2012 R2, Windows 10, Windows Server 2016

Scanned by CamScanner

1.5 Binding

→ (MU - May 2014)

Q. What is binding? Explain static and dynamic binding. **May 2014, 10 Marks**

- A **binding** is the association between the memory address attribute of a data item and address of a memory area.
- Binding is an association of an attribute of a program entity with a value.
  (1) Binding time is the time at which a binding is performed.
  (2) Memory allocation is the procedure used to perform memory binding.
  (3) The binding ceases to exist when memory is deallocated.
  (4) Memory binding can be static or dynamic.
- There are certain tasks on which we calculate binding times.
  (1) Language definition time of L.
  (2) Language implementation time of L.
  (3) Compilation time of P. (L-Programming language)
  (4) Execution time of procedure. (P-Program written in L)

Importance of binding time

The binding time of a program entity determines the manner in which a language processor can handle the use of the entity.

Static Binding

If a compiler can resolve the binding at the compile time only then such a binding is called static Binding or Early Binding. All the instance method calls are always resolved at runtime, but all the static method calls are resolved at compile time itself and we have static binding for static method calls. Static binding is used when we call a simple class method.

Dynamic Binding

It refers to the case where compiler is not able to resolve the call and the binding is done at runtime only. When we have class hierarchies and virtual methods compiler will use dynamic binding.

Review Questions

Q. 1 What is System software, Programming software and Application software ? Give e.g. of each.

Q. 2 Explain various types of System programming components in brief.

Q. 3 Differentiate between linker, loader and assembler.

Q. 4 Differentiate between ccompiler and interpreter give one example of each.

Q. 5 What are the various phases of compiler ? Draw the neat and labeled diagram for it.

Q. 6 What is an operating system and what are its various functions ?

Scanned by CamScanner

Q. 7 Explain Virtual Device Drivers.

Q. 8 What are device drivers and what is the purpose of device drivers ?

Q. 9 What are Text Editors ? Explain.

Q. 10 State importance of debuggers and editors in computer programming ?

Chapter Ends

❑❑❑

Scanned by CamScanner

Module 2

CHAPTER 2

Assemblers

Syllabus Topics

Elements of Assembly Language programming, Assembly scheme, pass structure of assembler.

Assembler Design : Two pass assembler design and single pass Assembler, Design for Hypothetical / X86 family processor, data structures used.

2.1 Introduction

Assembly languages are a family of low-level languages for programming computers, micro controllers, microprocessors and other integrated circuits. They implement a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. An assembly level language in thus specific to a certain physical or virtual computer architecture.

An assembler is a program used to convert the assembly level language statements into the target machine code. The assembler performs a one-to-one mapping from mnemonic instructions to machine statements and data. Mnemonic is a code usually from 1 to 5 letters that represents an op-code followed by one or more numbers. This is in contrast with high level languages in which single statement generally results in many machine instructions. An assembly language is a machine dependent, low level programming language which is specific to a particular computer system.

2.1.1 Advantages of Assembly Language

- It helps to get the better performance out of the processor as possible.
- It helps us to gain access to specific characteristics of the hardware that might not be possible from the higher level language.

2.1.2 Disadvantages of Assembly Language

- It always requires the use of an assembler to translate a source program into an object code.
- Assembly languages are specific to a given micro-processor/computer and hence are not portable.

Scanned by CamScanner

- An **assembler** is a program that accepts an assembly language as input program and produces its machine language equivalent along with the information for the loader. Fig. 2.1.1 shows the basic schematic of an assembler.

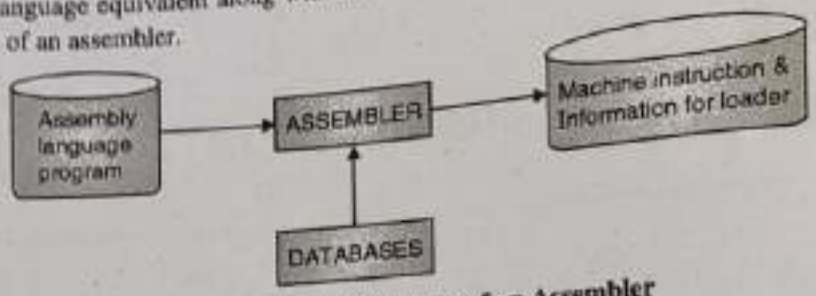


Fig. 2.1.1 : Basic Structure of an Assembler

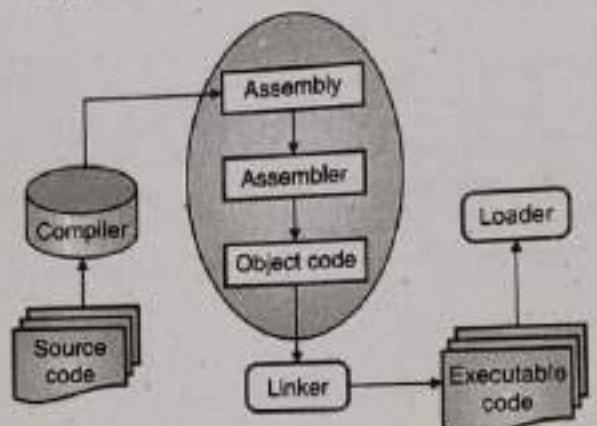


Fig. 2.1.2 : Working of an Assembler

2.2 Elements of an Assembly Language Programming

2.2.1 Basic Feature of an Assembler

An Assembler translates the mnemonic operation codes to their machine language equivalents and resolves symbolic names for memory locations and other entities. Assigning machine addresses to the symbolic saves tedious calculations and manual address updates after program modifications has been done.

Assemblers provide three basic features which simplify and ease the programming.

1) Opcode Mnemonics

An opcode is a symbolic name for a single executable machine instruction, and there is at least one opcode mnemonic defined for each of the machine language instruction. It eliminates the need to memorize numeric operation codes.

Scanned by CamScanner

E.g. MNEMONIC [DESTINATION], [SOURCE]

General format of mnemonic

MOV EAX, 1

↓ ↓ ↓

Mnemonic Destination Source

In this the value 1 is "moved" into the EAX operand (register).

2) Data Sections

They are instructions used to define data elements to hold data and variables. They define type of data, length and alignment of data.

3) Assembly Directives/Pseudo Operators

Assembly directives are instructions that are executed by an assembler at the assembly time. Symbolic assemblers allow programmers to associate arbitrary names (labels or symbols) with memory locations.

2.2.2 Assembly Language Program Contains Three Kinds of Statements

1) Imperative Statements

They are executable statements understood by the machines. It indicates an action to be performed during the execution of the assembled program. Each imperative statements typically translates into one machine instruction those are understood by the machine.

E.g. MOV A, 1

It states "Move" value one to register A

2) Declaration Statements

These are non executable statements. They declare constants or storage areas in a program.

E.g. X DS 2

(In this DS = Declare Storage. It reserves memory area of 1 word and associates the name X with it).

3) Assembler Directives

It instructs the assembler to perform certain actions during the assembly of a program.

E.g. ABC START 1000

END

(It indicates where to start assembly of the input program is performed).

Scanned by CamScanner

2.2.3 An Assembly Language Program

When the user wants to communicate with the computer, he has a lot of languages :

English — Best for programmer

C/ C++/JAVA

:

:

Assembly language

Machine language — Best for machines

We will now discussed assembly language which is the most machine dependent language

	PROGRAM		COMMENTS
TEST	START		Identifies the name of the program
BEGIN	BALR	15, 0	Set register 15 to the address of the next instruction
	USING	BEGIN + 2, 15	Pseudo-opcode indicating to assembler register 15 is base register, and its content is address of next instruction
	SR	4, 4	Clear register 4
	L	3, ELEVEN	Load the number eleven into register 3
LOOP	L	2, DATA (6)	Load data into register 2
	A	2, FORTYSIX	Add 46 to register 2
	ST	2, DATA (6)	Store updated value of the data
	A	4, SIX	Add six to register four
	BCT	3, LOOP	Decrement register 3 by 1, if result non-zero, branch back to loop
	BR	14	Branch back to caller
ELEVEN	DC	F '11'	Constant 11
SIX	DC	F '6'	Constant 6
FORTYSIX	DC	F '46'	Constant 46
DATA	DC	F '1, 3, 3, 3, 4, 5, 8, 9, 0'	Words to be processed
	END		

The programmer is not aware where to load the program and from where to start the program. The BALR instruction is used for loading the base register.

Scanned by CamScanner

Description

1) **USING**

The using instruction tells what register can be used as base register and what that register will contain. Using is a pseudo-opcode to indicate the type of the register. A pseudo-opcode is an assembly language instruction that specifies an operation of the assembler. It is distinguished from the machine-op code which represents to the assembler a machine instruction.

2) **BALR**

Base register is loaded with the BALR instruction whereas USING pseudo-opcode provides information to the assembler.

3) **START**

It tells from where the beginning of the program is to be done.

4) **END**

It tells where the program is to be end.

5) **DS and DC**

The Define Storage and Define Constant are pseudo-opcodes that instructs an assembler to place a 10, 6, and 46 in 3 consecutive full words ("F") in memory and leave 10 more spaces for the data.

6) **LITERALS**

Literal constants (often referred to as literals or constants) are explicit invariant values contained with source programs.

Assembled version of Example program

Assembly Language Program			Relative Location	Assembled Mnemonic Program	
TEST	START				
BEGIN	BALR	15, 0	0	BALR	15, 0
	USING	BEGIN +2, 15	2		
	SR	4, 4	4	SR	4, 4
	L	3 = F '10'	8	L	3, 30 (0, 15)
LOOP	L	2, DATA (6)	12	L	2, 42 (6, 15)
	A	2, = F '46'	16	A	2, 34 (0, 15)
	ST	2, DATA (6)	20	ST	2, 42 (4, 15)
	A	4, = F '4'	24	A	4, 38 (0, 15)
	BCT	3, * – 16	28	BCT	3, 6 (0, 15)
			:		
DATA	DC	F '1, 3, 3, 3, 3, 4, 5, 9, 0'	44	1	
			48	3	
			52	3	
		END			

Scanned by CamScanner

e.g. A 3, 30 (0, 15)

Index Register Value

offset

Base Register Value

It means A number is to be added to the contents of register 3.

2.3 Basic Assembler Functions

There are certain fundamental functions that any assembler should perform such as

- **A simple SIC (Simplified Instructional Computer) Assembler :** It gives the basic functions performed by a assembler and describes a way to carry out these functions
- **Assembler Algorithms, Tables, Data Structure and Logic :** It describes the data structure table and logic used by almost all of the assemblers.

2.3.1 Essential Assembler Functions

- Mnemonic opcodes are converted into equivalent machine instructions.
- Symbolic operands are converted into machine addresses.
- Machine instructions are building to perform operations.
- Data constants are transformed into internal representations.
- Object program is obtained.

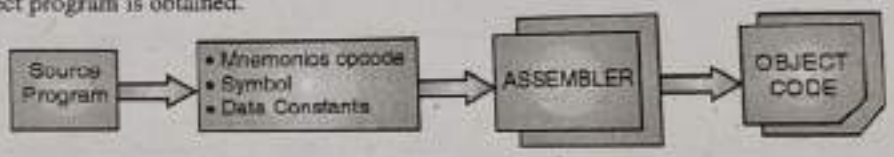


Fig. 2.3.1 : Basic assembler functions

2.3.1(A) Assembler Directives

Q. What are assembler directives ? Explain with example. (5 Marks)

Assembler directives are pseudo-instructions; they provide instructions to the assembler itself and are not translated into machine operation codes. It helps us to do to the following tasks :

- Assemble code and data into specified sections
- Reserve space in memory for uninitialized variables
- Control the appearance of listings
  - Initialize memory
  - Assemble conditional blocks
  - Define global variables
  - Specify libraries from which the assembler can obtain macros
  - Examine symbolic debugging information

Scanned by CamScanner

2.4 Assembly Scheme

The steps that should be followed by the programmer for designing of an assembler are mentioned as following :

1. Specify the problem to identify the information necessary to perform a task.
2. Design a suitable data structure to record and store the information.
3. Determine the process necessary to obtain and maintain the information.
4. Specify the algorithm.
5. Determine the process necessary to perform the task.
6. At last the programmer should look for the modularity of the program (i.e. program to be subdivided into independent programming units). It helps all the modules to be tested and deployed separately.

   Basic steps required in designing the modular system are as :

   - Modules created should be opaque to the rest of the system and initialized through a well-known interface.
   - Modules should not directly reference with one another or the application that loaded them.
   - Modules should not communicate directly they should use loosely coupled techniques, such as shared services, to communicate with the application or with other modules.
   - Modules are not responsible for managing their dependencies.
   - Modules should not rely on static methods that can inhibit testability.

   The mentioned basic steps are also used for the design of the loader, and complier.

2.5 Pass Structure of Assembler

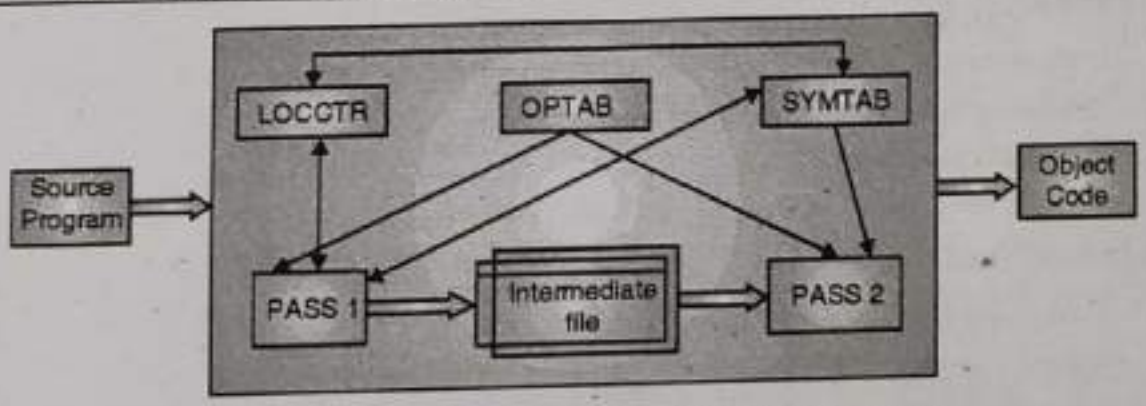


Fig. 2.5.1 : Assembler Algorithms and Data Structures

The simpler assembler uses three major internal data structures :

1. **OPTAB :** Operation Code Table
2. **LOCCTR :** Location Counter
3. **SYMTAB :** Symbol Table

– The intermediate file includes each source statement, assigned address and error indicator.

Scanned by CamScanner

1. **OPTAB :** It includes
   - Mnemonic operation code ⇔ Machine code
   - It has instruction format and length
     - LOCCTR ← LOCCTR + (instruction length)
   - Implementation
     - It is a static table (No updates)
     - Array or hash table. (It is usually organized as a hash table, with mnemonic operation code as the key)

       In **PASS 1 :** 1) It validates the op-codes 2) Computes Instruction length

       PASS 2 : Translate op-code into machine language.

2. **LOCCTR :** It includes
   - It initialises the beginning of the "START" address.
   - LOCTR ← LOCCTR + (instruction length)
   - The current value of LOCCTR gives the address of the label encountered.

3. **SYMTAB :** It includes
   - It is used to store values assigned to the labels.
   - It includes the name and value for each label in the source program together with type, length and flag.
   - It is a dynamic table.
   - It has the operation like insert, delete and search.
   - It usually uses a hash table.

In PASS 1

- Enter labels and addresses to all statements as they are encountered in source program.
- Processing for assembler directives is performed.
- Enter addresses from LOCCTR.
- Save addresses for all values assigned to all labels for case in PASS 2.

In PASS 2 (Multi pass)

- Look up operand labels for addresses.
- Assemble instructions.
- Write the object program.

2.6 Assembler Design

2.6.1 Forward Reference Problem

→ (MU - Dec. 2014)

Q. What is forward reference problem ? How is it is resolved in two-pass assembler ?

Dec. 2014, 5 Marks

When a symbol is referred before it is defined, it is called a ***forward reference***.

Scanned by CamScanner

The function of the assembler is to replace each symbol by its machine address and if we refer to that symbol before it is defined its address is not known by the assembler such a problem is called **forward reference problem**. Forward reference problem is solved by making different passes (i.e. One pass, Two pass, Multi pass) over the assembly code. E.g.:

```
class C {
public :
void muta for (int x) {myValue = x;}
int accessor () {return myValue;}
private :
int myValue;
};
```

In this there are two reference to myValue before it is declared.

2.6.2 Phases used by the Assembly Language

Basically Assembly language uses two phases.

1) Analysis Phase

The tasks performed by this phase are as :

(a) It isolates the labels, mnemonic opcodes (An opcode is the portion of a machine language instruction that specifies the operation to be performed) and operand fields of a statement.

(b) If a label is present enter the pair (symbol, location counter (LC) contents) in a new entry of a symbol table.

(c) Check the validity of the mnemonic opcode through mnemonic table.

(d) Perform processing of location counter i.e. update the value contained in LC by considering the opcode and operands of the statements.

2) Synthesis Phase

Synthesis phase includes :

(a) Obtain the machine opcode corresponding to the mnemonic from the mnemonics table.

(b) Obtain addresses of a memory operand from the symbol table.

(c) Synthesize a machine instruction.

Scanned by CamScanner

Fig. 2.6.1 shows the two phases of an assembler.

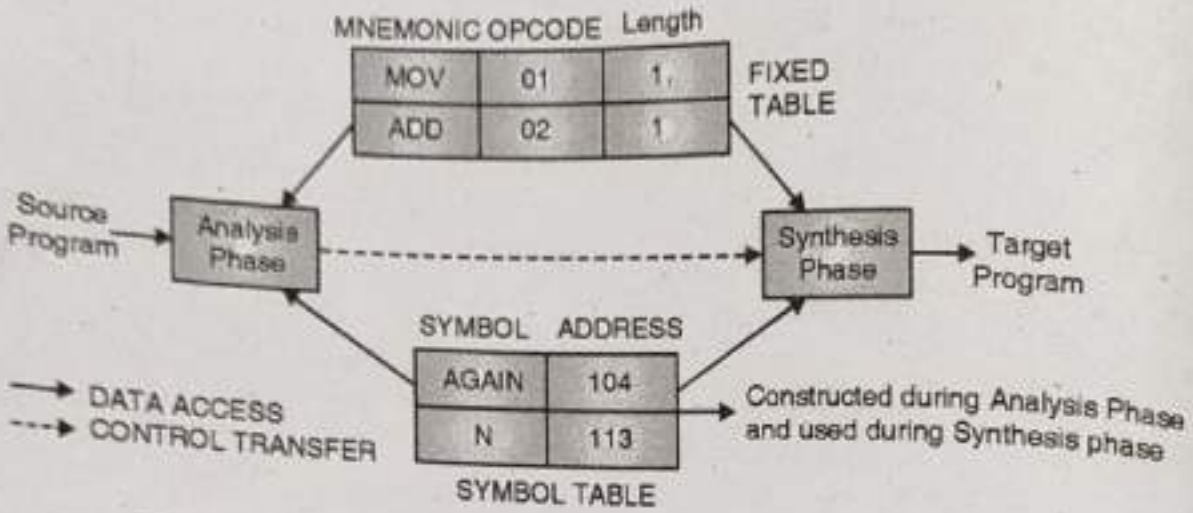


Fig. 2.6.1 : Two Phases of an Assembler used by the Various Passes

2.7 Two Pass Assembler Design

(MU - May 2014, May 2015, Dec. 2015, May 2016, Dec. 2016, Dec. 2017)

Q. What are advantages of assemblers with multiple passes ? How many passes should be there in a typical compiler ? **May 2014, 5 Marks**

Q. State the reason for the assembler to be a multi pass program. **May 2015, 5 Marks**

Q. With reference to assembler explain the following tables with suitable examples :
(i) POT (ii) MOT (iii) ST (iv) LT **Dec. 2015, May 2016, 10 Marks**

Q. What are various databases used in two pass assembler design. Explain with example. **Dec. 2016, 8 Marks**

Q. Explain the design of two pass assembler with flowchart and databases. **Dec. 2017, 10 Marks**

Q. State the reasons for the assembler to be two pass program. **(5 Marks)**

Q. Explain the databases used by each pass of the 2 - pass assembler. Explain how these databases are used by the 2 - pass assembler when it processes the source program with an example. Clearly show all the entries in the databases built by the 2 - pass assembler. **(10 Marks)**

Q. Explain two pass assembler with flowchart. **(10 Marks)**

A two pass assembler means more than one pass is used by assembler. Examples of the two pass assemblers are IBM 360 / 370 processor. The advantages of two pass assemble over single pass assembler are as follows:

- It is used to eliminate forward reference problem.
- Numbers of passes are created in multi pass assembler to process the definition of symbols.
- Multi pass assembler does the work in two pass
- First Pass:
  - Scans the code.
  - Validate the tokens.
  - Create Symbol Table.

Scanned by CamScanner

- Second Pass:
  - Solves forward reference problem.
  - Converts the code to the machine code.
- A two pass assembler does two passes over the source file. Below mentioned is the detail of task accomplished by each pass. The forward reference problem is solved by making two passes over the assembly code.

(I) **Pass 1 :** It looks for label definition and introduces them into the symbol table. It defines the symbols and literals.

(a) It keeps the track of location counter (LC)

(b) Determines the length of machine instructions (MOT).

(c) It keeps the track of the values of symbols until pass 2 is done (ST).

(d) Process some pseudo-opcode e.g. EQU, DS DC (POT).

(e) It stores the literals (LT).

(II) **Pass 2 :** After the symbol table is completed in PASS 1 it does the actual assembly by translating the operations and so on. The purpose of PASS 2 is to generate object code.

(a) It looks up the values of symbols (ST).

(b) It generates the instructions (MOT).

(c) It generates the data (For DS, DC and Literals).

(d) Process some pseudo-opcodes e.g. USING, DROP (POT).

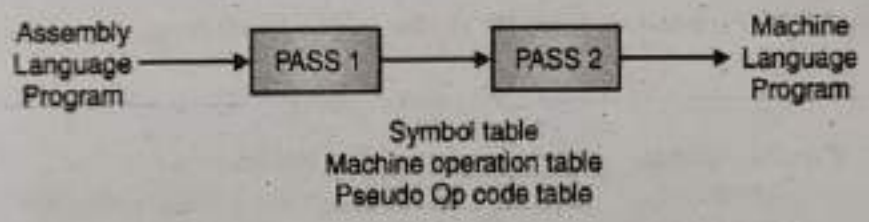


Fig. 2.7.1 : A Simple two Pass Assembler

(III) **FORMAT of DATABASES :** In this we specify the format and content of each of the data bases used in the PASS 2.

Machine Opcode Table (MOT)

MOT is a fixed length table i.e. the contents of these tables are not filled in or altered during the assembly process.

Scanned by CamScanner

Table 2.7.1 : Machine-Op Table (MOT) for pass 1 and pass 2

← 6-bytes per entry →

Mnemonic op-code (4-bytes) (characters)	Binaryop-code (1-byte) (hexadecimal)	Instruction length(2-bits) (binary)	Instruction format(3-bits) (binary)	Not used in this design (3-bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	
"ARbb"	1A	01	000	
...	...	...	...	
"MVCb"	D2	11	100	

b–represents the character "blank"

Codes

Instruction length

01 = 1 Half-Words = 2 bytes
10 = 2 Half-Words = 4 bytes
11 = 3 Half-Words = 6 bytes

Instruction format

000 = RR – Register to Register
001 = RX – Register and Indexed Storage Operation
010 = RS – Register and Storage Operation
011 = SI – Storage and Immediate Operation
100 = SS – Operation using all implied operand and storage

Pseudo Opcode Table (POT) : POT is fixed length table i.e. the contents of these tables are not filled in or altered during the assembly process.

Table 2.7.2 : Pseudo-Op Table (POT) for pass 1 (similar table for pass 2)

← 8-bytes per entry →

Pseudo-Opcode (5-bytes) (character)	Address of routine to process Pseudo-Opcode (3-bytes = 24 bit address)
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING

These are presumably lables of routines in pass 1; the table will actually contains the physical addresses.

P1→PASS1

Scanned by CamScanner

Symbol Table (ST)

Table 2.7.3 : Symbol Table (ST) for pass 1 and pass 2

← 14-bytes per entry →

Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-bytes) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

Symbol Table includes for each entry :

i) Name of the symbol

ii) Symbols assembly time value

iii) Length (in bytes) and

iv) Relative Location indicator (R/A)

Operand can be either immediate (typically one byte values coded in the instruction itself/)or the addresses of the data located elsewhere

(Relative/Absolute value)

R - Relative value

A - Absolute value

Literal Table (LT)

Literal is equivalent to define a constant explicitly and assign an address label for it literal table is often organized as a hash table, using literal name or value as the key.

Literal Table includes for each entry

i) Name of the literal

ii) Literal assembly time value

iii) Length (in bytes) and

iv) Relative Location Indicator (R/A)

(The format is same as that of symbol Table - with required changes) based on the definition of the Literal table.

Name of Literal (1-byte)	Value of Literal (4-byte)	Length of Literal (1-byte)	Relative Location Indicator (1-byte)

Base Table (BT)

It is used by the assembler to generate the proper base register reference in machine instructions and its correct offsets.

Scanned by CamScanner

Table 2.7.4 : Base Table (BT) for pass 2

4-bytes per entry

	Availability indicator (1-byte) (character)	Designated relative-address Contents of base register (3-bytes = 24 bit address) (hexadecimal)
1	"N"	–
2	"N"	–
:	:	–
14	"N"	00 00 00
15	"Y"	

15 entries

Availability

Y – register specified in USING pseudo-opcode

N – register never specified in USING pseudo-opcode or subsequent made unavailable by the DROP pseudo-opcode

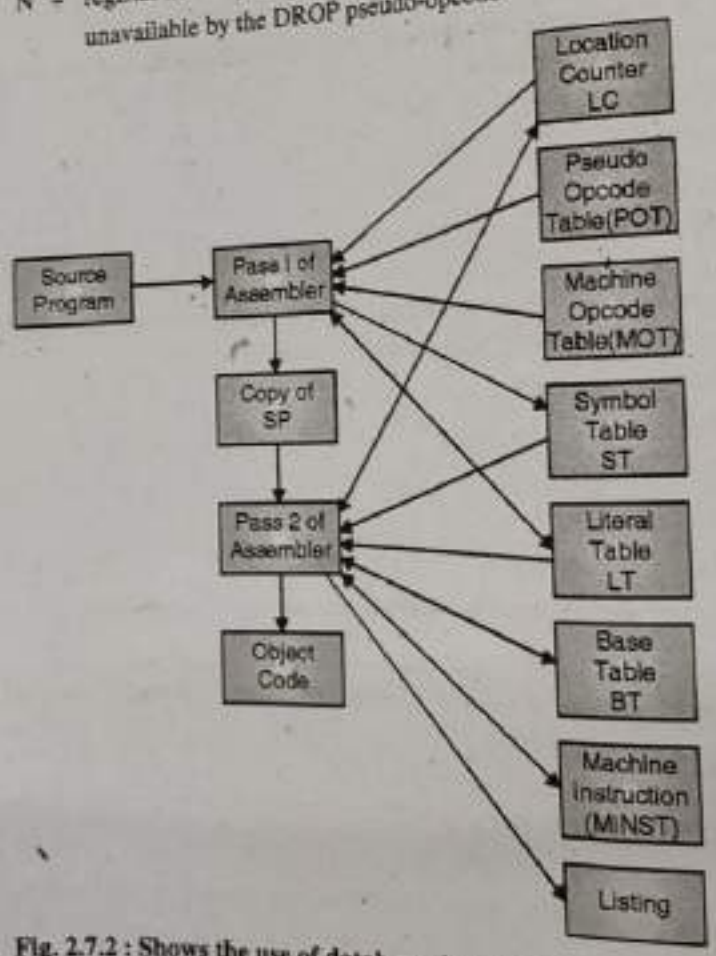


Fig. 2.7.2 : Shows the use of databases by two-pass assembler

(IV) Databases Requir

1) Input source pro
2) **Location Cour** each symbol de
3) **Machine Oper** and its length.
4) **Pseudo Opera** in pass 1.
5) **Symbol Table**
6) **Literal Table** assigned locati
7) A **copy** of the

(V) Databases Requ

1) **Copy** of sourc
2) Location Cou
3) **MOT** that inc
   (a) Symboli
   (c) Binary r
4) **POT** that ind in pass 2.
5) **Symbol Tab**
6) **Literal Tab** location.
7) **Base Table** pseudo-opco
8) **INSTRUCT** being assem
9) **PRINT LI**
10) **PUNCH C** the loader.
11) An **output**

(VI) PASS 1 : Defi

First pass assig symbols appear

1) Set locatio
2) Read the i
3) Opcode is statements
   a. If th instr

Scanned by CamScanner

(IV) Databases Required for Pass 1

1) Input source program.
2) **Location Counter (LC)** to keep track of each instructions location and to assign address for each symbol defined.
3) **Machine Operation Table (MOT)** that indicates the symbolic mnemonic for each instruction and its length.
4) **Pseudo Operation Table (POT)** that indicates the pseudo-opcode and the action to be taken in pass 1.
5) **Symbol Table (ST)** that is used to store each label and its corresponding value.
6) **Literal Table (LT)** that is used to store each literal encountered and its corresponding assigned location.
7) A **copy** of the input to be used later by pass 2.

(V) Databases Required for Pass 2

1) **Copy** of source program input to pass 1.
2) Location Counter (LC)
3) **MOT** that indicates for each instruction
   (a) Symbolic mnemonic (b) Length
   (c) Binary machine op-code (d) Format (RR, RX, ...)
4) **POT** that indicates for each pseudo-opcode the symbolic mnemonic and the action to be taken in pass 2.
5) **Symbol Table (ST)** prepared by pass 1, containing each label and its corresponding value.
6) **Literal Table (LT)** prepared by pass 1, containing each literal encountered and its assigned location.
7) **Base Table (BT)** that indicated which registers are specified as base registers by USING pseudo-opcodes and what the specified contents of these registers are.
8) **INSTRUCTION workspace** which is used to hold each instruction as its various parts is being assembled together.
9) **PRINT LINE workspace** used to produce printed listing.
10) **PUNCH CARD workspace** used to punch the assembled instructions in the format needed by the loader.
11) An **output deck** of assembled instructions in the format needed by the loader.

(VI) PASS 1 : Define Symbols and Literals

First pass assigns a location to each instruction and data and it also defines the value for each symbols appearing in the label fields of the input program.

1) Set location counter to 0.
2) Read the input program.
3) Opcode is checked to find whether it is pseudo-opcode or MOT to locate the match for source statements.
   a. If the entry found in MOT is matched then the MOT entry gives the length of the instruction.

Scanned by CamScanner

b. Operand field is used to check for the presence of a literal and these literals are stored in literal table for processing.

c. After literals label field is checked for the presence of a symbol. If found, enter the symbol in the symbol table with the existing value of the location counter.

d. At last location counter is incremented by the length of the instruction.

e. Copy of input source deck is made available for use by pass 2.

Process of Pseudo-Opcodes

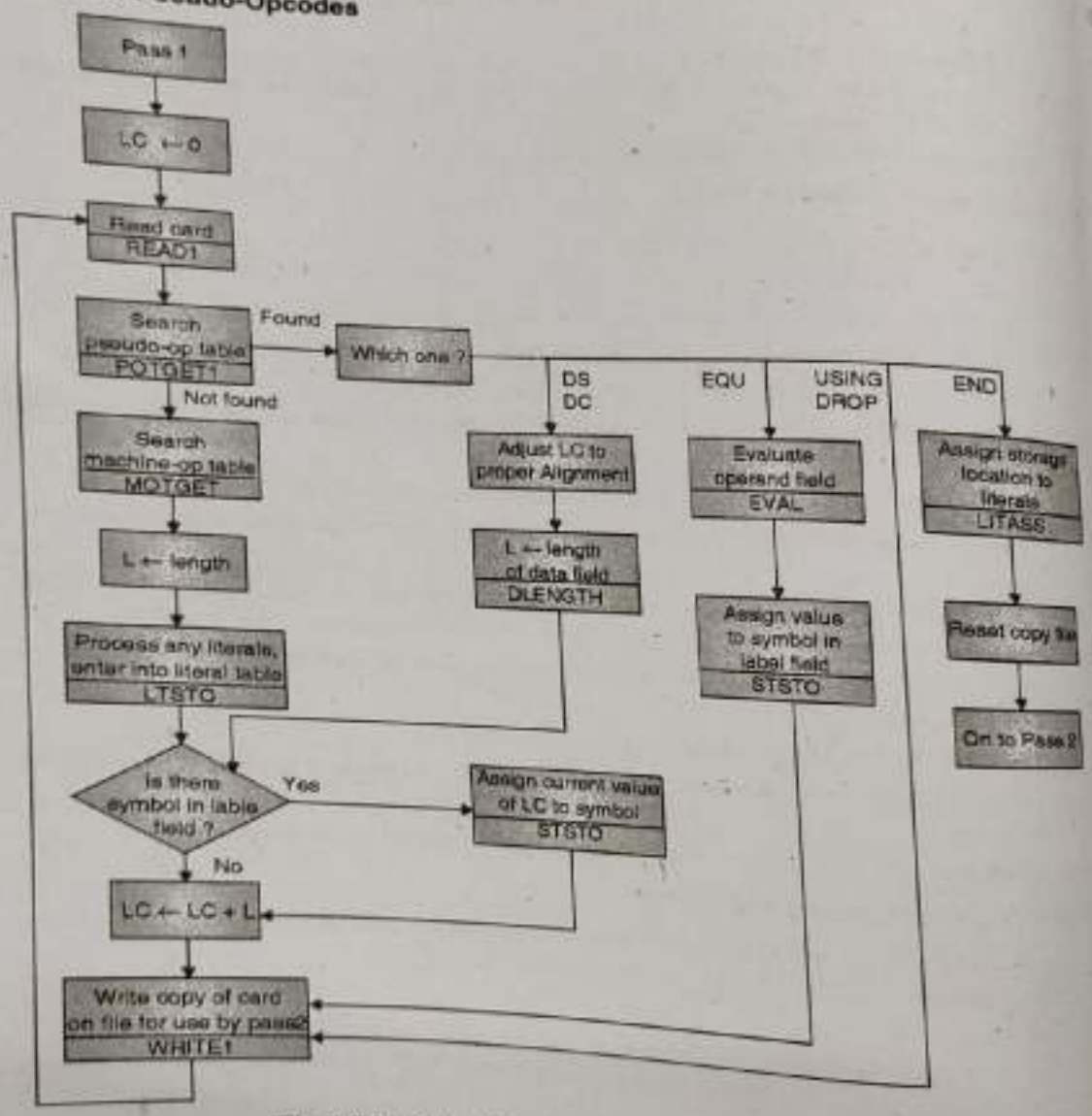


Fig. 2.7.3 : A flowchart for pass 1 of Two Pass Assembler

Scanned by CamScanner

i) **DS and DC :** These both are the assembler directive that tells the assembler something it needs to know in order to carry out the assembely process.

DC stands for Define constant. In assembly language specified as

<label> DC <value> Define Constant

<label> DS <value> Define Storage

<label> = Symbolic name, <value> = It indicates the value supplied by the programmer.

They can affect both LC and the definition of the symbol in the pass 1.

ii) **EQU :** Define the symbol and assign the value evaluate by the expression in the operand field.

iii) **USING and DROP :** These pseudo-opcodes neither define symbols labels nor affect the location counter. So no processing required only save the USING and DROP cards for pass 2.

iv) **END :** When END is encountered pass 1 is terminated and literals are assigned the storage locations and we goto pass 2. The mentioned steps are repeated for the next instructions.

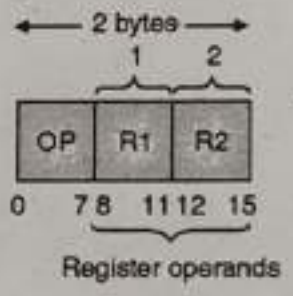
(VII) Pass 2 : Object Code is Generated

It generates object code into the appropriate format for later processing by the loader.

1) Location Counter is initialized.
2) Statement is read from the copy file created by pass1.
3) Opcode is checked to find whether it is pseudo-opcode or MOT to locate the match for source statements.. If the entry found in MOT is matched then the MOT entry gives the length of the instruction.

a) RR - Format Instruction

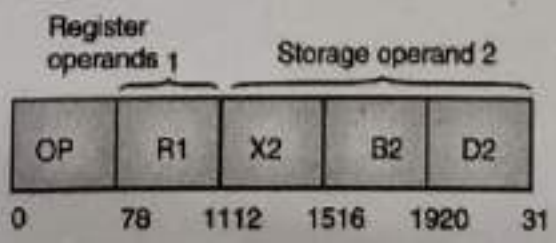
- It denotes register to register operation
- Each of the register specification field is evaluated and the two fields are inserted into their respective four-bit fields in the second byte of the RR-instruction.
- RR has 8-bit opcode and two 4-bit general register field that typically specifies any of 16 general registers.



b) RX - Register and Indexed Storage Operations

- The register (R1) and index fields are evaluated and the two fields are inserted into their respective four-bit fields.
- We calculate the effective address of the operand and we determine appropriate displacement and base register and also ensure D < 4096 and put base and displacement in the 3rd and 4th bytes.

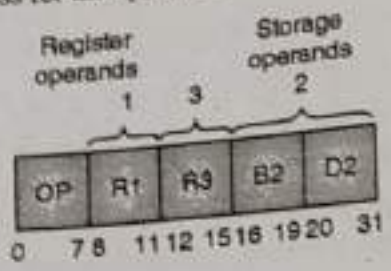
RX -format



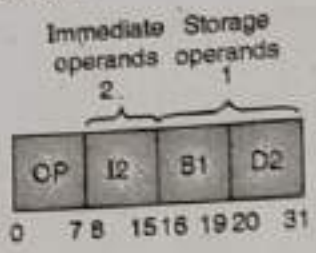
Scanned by CamScanner

c) **RS - Register and Storage Operations**

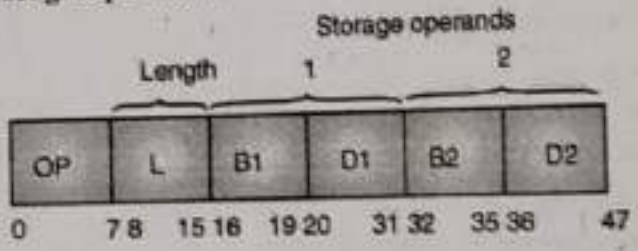
- This is a five-byte instruction of the form OP R1, R3, D2, B2. The first byte contains the 8-bit instruction code.
- The second byte contains two 4-bit fields, each of which encodes a register number.
- The third and fourth byte contains a 4-bit register number and 12-bit displacement, used to specify the memory address for the operand in storage.



d) **SI - Storage and Immediate Operations**



e) **SS - Storage and Storage Operations**



OP – Operation code

R_i – Contents of general register used as operand

X_i – Contents of general register used as Index

B_i – Contents of general register used as Base

D_i – Displacement

I_i – Immediate data

L – Operand length

Scanned by CamScanner

Processing of Pseudo-Opcodes

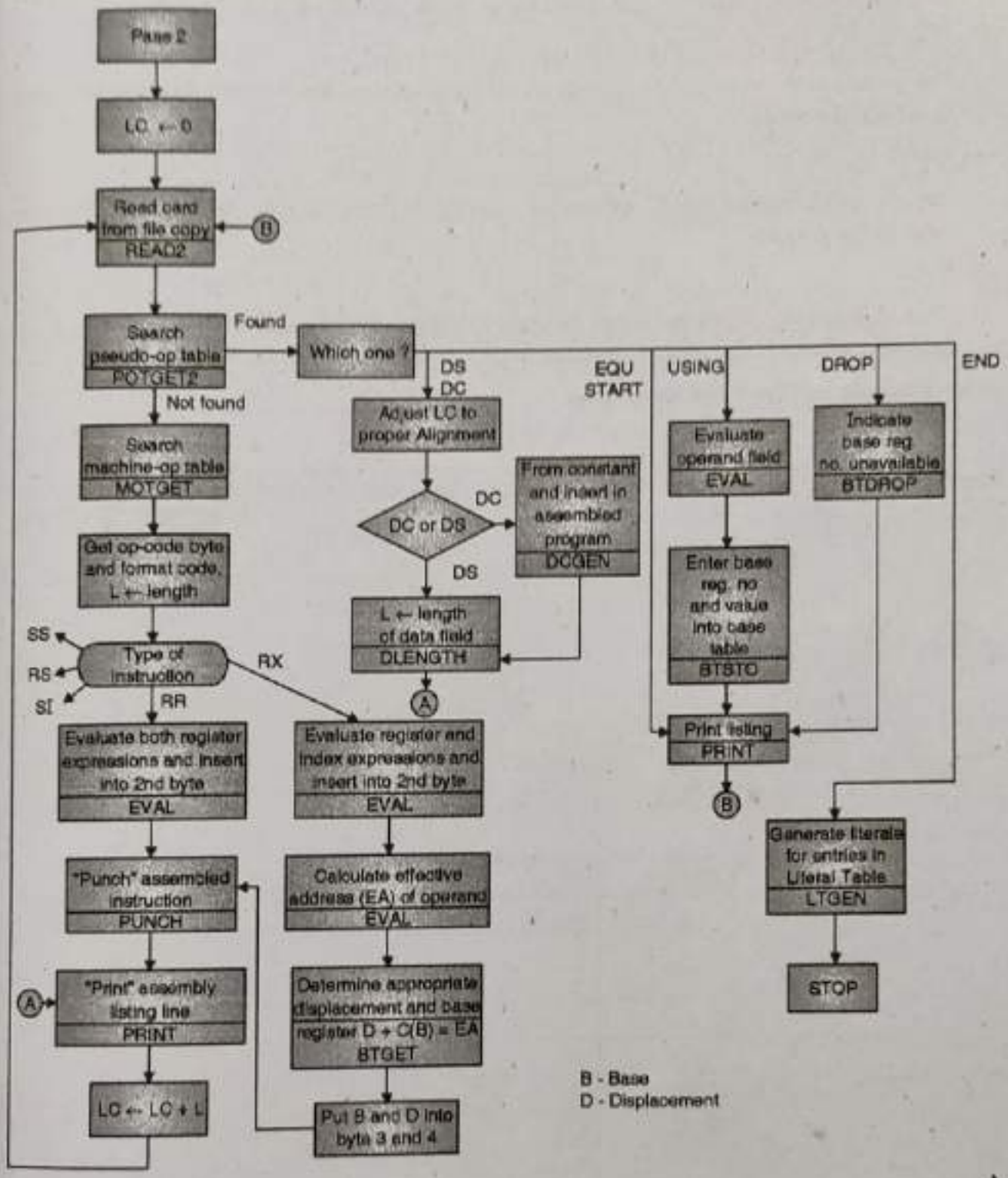


Fig. 2.7.4 : Flowchart for pass 2 of Two Pass Assembler

i) DS and DC

Due to the requirements that the full words must start on a byte whose address is a multiple of four we adjust the location counter. In case of DC we form constants and insert in the assembled program and increment the LC as per the length of the data field.

Scanned by CamScanner

ii) **EQU**

EQU pseudo-op only prints the EQU card as a part of the printed listing.

iii) **USING**

The operand fields of the pseudo-opcodes are evaluated and then the corresponding base table entry in marked as available.

iv) **DROP**

The operand field of the pseudo-opcodes are evaluated and then the corresponding base table entry is marked as unavailable.

v) **END**

END indicates the end of the source program and terminated the assembly, and the literals are generated for entries in Literal Table (LT).

(VIII) Example of Two Pass Assembler

➔ (MU - May 2015)

Example 2.7.1 For following code what will be output generated by Pass-I and Pass-II for two pass assembler. Explain with database. **May 2015, 10 Marks**

```
ABC     START   0
        USING   *, 15
        L       1, FIVE
        A       1, FOUR
        ST      1, TEMP
FOUR    DC      F '4'
FIVE    DC      F '5'
TEMP    DS      IF
        END
```

Solution :

PRG	START	0
	USING	*, 15
	L	1, FIVE
	A	1, FOUR
	ST	1, TEMP
FOUR	DC	F '4'
FIVE	DC	F '5'
TEMP	DS	IF
	END	

The above mentioned program was assumed to be called by another program that left the address of the first instruction in register 15.

Scanned by CamScanner

Pass 1

Sample Program			Relative Address	Mnemonic Instruction
PRG	START	0		
	USING	*,15		
	L	1, FIVE	0	L 1, – (0, 15)
	A	1, FOUR	4	A 1, – (0, 15)
	ST	1, TEMP	8	ST 1, – (0, 15)
FOUR	DC	F '4'	12	4
FIVE	DC	F '5'	16	5
TEMP	DS	1F	20	–
	END			

- We look up bit configuration for the mnemonic in a table and put the bit configuration for the L in the appropriate place.
- Now we need the address of five. At this point we don't know where FIVE is, so we cannot supply its address. And no index register is used. We put in 0 for the index register and we know that the register 15 is being using as the base register, but we cannot calculate offset so we write 1, – (0, 15).
- We maintain a location counter indicating the relative address of the instruction being processed this counter is incremented by 4.
- The next instruction is an Add instruction. We look up the opcode but we do not know the offset for four. The same thing happens with store (ST) instruction.
- The DC instruction is a pseudo-opcode directing us to define some data.

Pass 2

- For 'four' we have to produce a '4'. This word has been stored at relative location 12, from the pass 1. So we will give the value 12 to the Load (L) instruction in pass 2. Same case is with Add and store instruction and their location counter value will be given as 12 and 20.

2.8 One Pass Assembler

→ **(MU - Dec. 2014)**

Q. Explain one pass assembler with flowchart and respective databases. **Dec. 2014, 10 Marks**

Q. Explain with suitable flow chart working of single pass assembler. **(10 Marks)**

Q. Explain the working of single pass assembles. Show the structure of its databases used. **(10 Marks)**

It is also known as the Single Pass Assembler for the Intel 8086 or x86 / 8088 processor used in IBM PC. In single pass assembler the assembler passes over the source exactly once and converts equivalent machine code of the whole program at one time. Due to this it requires less time but more memory space as it has a single chance to convert the program.

Scanned by CamScanner

(I) Problem faced by Single Pass Assembler

(a) **Forward reference :** It occurs when a symbol is used before it is declared.

(b) **Difficulty in determining the length of the branch instruction :** If there is a forward reference for a branch instruction like "JUMP NEXT" where "NEXT" could be within 128 bytes from the instruction or more. Thus the length of the instruction depends on the location of "NEXT".

(II) Solution for the above problem

(a) The forward reference problem can be solved by using the FORWARD REFERENCE TABLE.

(b) It requires that all areas must be defined before they are referenced.

(c) It is possible, although inconvenient, to do so for data items.

(d) Forward JUMP to instruction items cannot be easily eliminated.

- Insert (label, address-to-be modified) to SYMTAB
- Usually address-to-be-modified is stored in a linked list.

(e) Forward Reference Table in one-pass assembler :

(i) Removes the operand address if the symbol is not defined.

(ii) Undefined symbol is entered into SYMTAB and marked as not defined.

(iii) The address of the operand is added to the forward references associated with the SYMTAB entry.

(iv) As soon as the definition for the symbol is found, scans the reference list and inserts the addresses.

(v) At the end reports the error if there are still SYMTAB entries indicating undefined symbols.

(III) Two types of One Pass Assembler

There are mainly two types of single-pass assembler :

1) First type produces object code directly in memory for immediate execution e.g. **LOAD AND GO ASSEMLBER**. In this type of assembler no object program is written out, no loader is needed. It is useful in a system with frequent program development and testing. In this program are re-assembled nearly every time they are executed in load and go assembler forward references became less difficult because the object program is produced in memory rather than being written out on secondary storage.

2) The second type produces the usual kind of object program for later execution E.g. X86, 8088 Assembler, MASM. Assembler.

Scanned by CamScanner

(IV) Single Pass Assembler (Intel x86 Assembler)

1) Memory and Registers

(a) The basic units of memory are as follows :

Units of memory	Bytes	Length in bits
Byte	1	8
Word	2	16
Double word	4	32
Quadra word	8	64
Tetra word	10	80

(b) The architecture supports a stack for storing subroutine address as, parameters and other data.

(c) Two stack pointer registers called SP (Stack pointer) and BP (Base Pointer) are provided to address the stack.

(d) The index registers SI and DI are used to index the source and destination addresses.

(e) The memory is used to store three components of a program to be executed, they are program code, data and stack. CPU contains four 16 bit registers to address these components.

2) Declarative Statements

Declaration of constants and reservation of storage are both done in this.

Table 2.8.1 : Declarative statements

Sr. No.	Declarative statements	Description	Length
1	DB	Reserve a byte	1
2	DW	Reserve a word	2
3	DD	Reserve a double word	4
4	DQ	Reserve a Quadra word	8
5	DT	Reserve a Tetra word	10

3) Assembler Directive Statements

Q. What is the use of ORG (ORIGIN)? (3 Marks)

(a) **SEGMENT :** It indicates the start of the segment.

(b) **ASSUME :** The assembler has to determine its offset (offset refers to a value added to a base address to produce a second address. For example, if B represents address 100, then the expression B + 5 would signify the address 105. The 5 in the expression is the offset specifying address using an offset is called relative addressing because the resulting address is relative to some other point. Another word for offset is displacement) from the start of the corresponding segment. To facilitate this, the programmer has to perform two tasks.

(i) Load a segment register with the segment base.

(ii) Let the assembler know what address a segment register contains.

(c) **DB :** This pseudo-opcode is used to define bytes.

Scanned by CamScanner

(d) **ORG** : This pseudo-opcode is used for the manipulation of the value of the location counter.

(e) **EQU** : It indicates whether the symbol is defined with EQU statements.

(f) **END** : It indicates the end of the segment.

4) Format of Databases

(a) **Machine Operation Table (MOT)** : The MOT is organized and contains opcodes for assembly statements. The routine id is field of a MOT entry contains identifier of the routine which processes that particular opcode.

Table 2.8.2 : Format of MOT

Mnemonic opcode (6)	Machine opcode (2)	Alignment information 1	Routine (id)
Add	-	FFH	R1
"JMPbbb"	-	FFH	R2
JNE	75 H	00H	R2

Machine understandable code (operational codes)

00H-Single Introduction format is supported

FFH-Multiple Instruction format is supported.

Registers

(b) **SYMBOL TABLE** : (SYMTAB) It is a hash organized and includes all relevant information about the symbol defined and used in the source program.

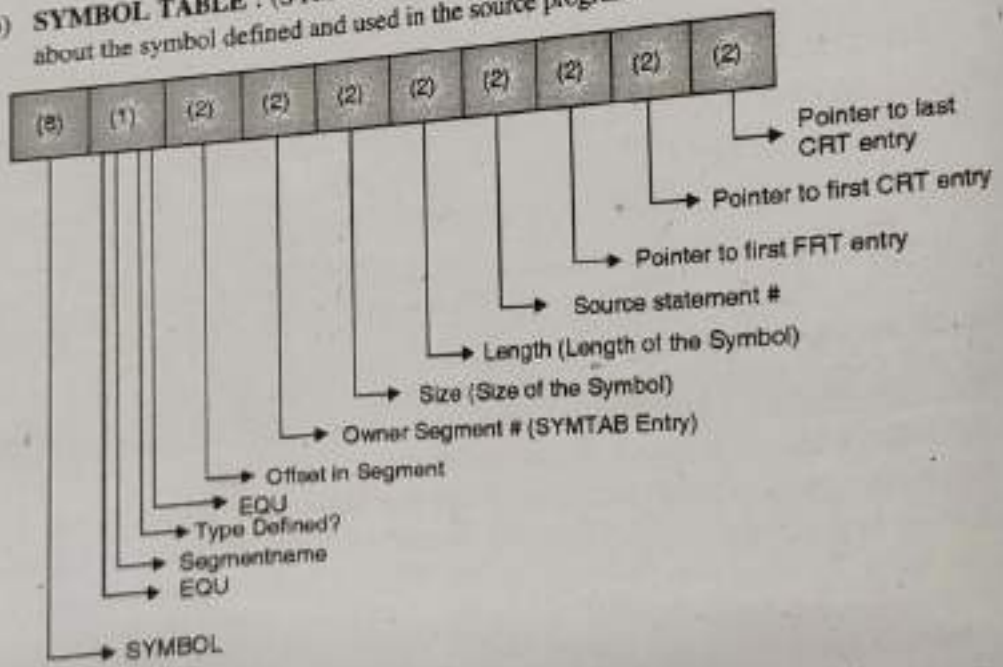


Fig. 2.8.1 : Symbol table

The segment in which the symbol is defined is given by the owner segment field which contains the SYMTAB entry. EQU symbol type indicates whether the symbol is to be given numerical value or text value.

Scanned by CamScanner

(c) **SEGMENT REGISTER TABLE :** (SRTAB) whenever an assembler encounters ASSUME statement it stores the previous SRTAB on the stack and new SRTAB is created.

Table 2.8.3 : Segment register table

SEGMENT REG.	SEGMENT NAME

(d) **STORED SEGMENT REGISTER TABLE (STSRT)**

Table 2.8.4 : Stored segment register table

	Segment Register	Segment Name	
00	ES		STSRT #1
01	CS (Code segment)		
10	SS (Stack segment)		
11	DS (Data segment)		
00	ES		STSRT #2
01	CS		
10	SS		
11	DS		

(e) **FORWARD REFERENCE TABLE (FRT) :** Information regarding forward referenced is put into the FRT in a linked list fashion.

Table 2.8.5 : Forward reference table

Pointer to Next Entry (2)	Entry # in STSRT (1)	Instruction Address (2)	Usage Code (1)	Source Statement (2)

- It contains the address of the instruction making the forward reference.
- Kind of reference made.
- Segment registers to be used for the references.
- Statement number at which the forward reference occurred.
- Usage code is used to indicate the type of forward reference that was made to the symbol.

(f) **CROSS REFERENCE TABLE (CRT) :** CRT is used for producing a cross reference Directory. This directory lists all references to a symbol in the ascending order of statement numbers.

Scanned by CamScanner

Table 2.8.6 : Cross reference table

Pointer to Next Entry	Source Statement #

(V) Flowchart of a Single Pass Assembler

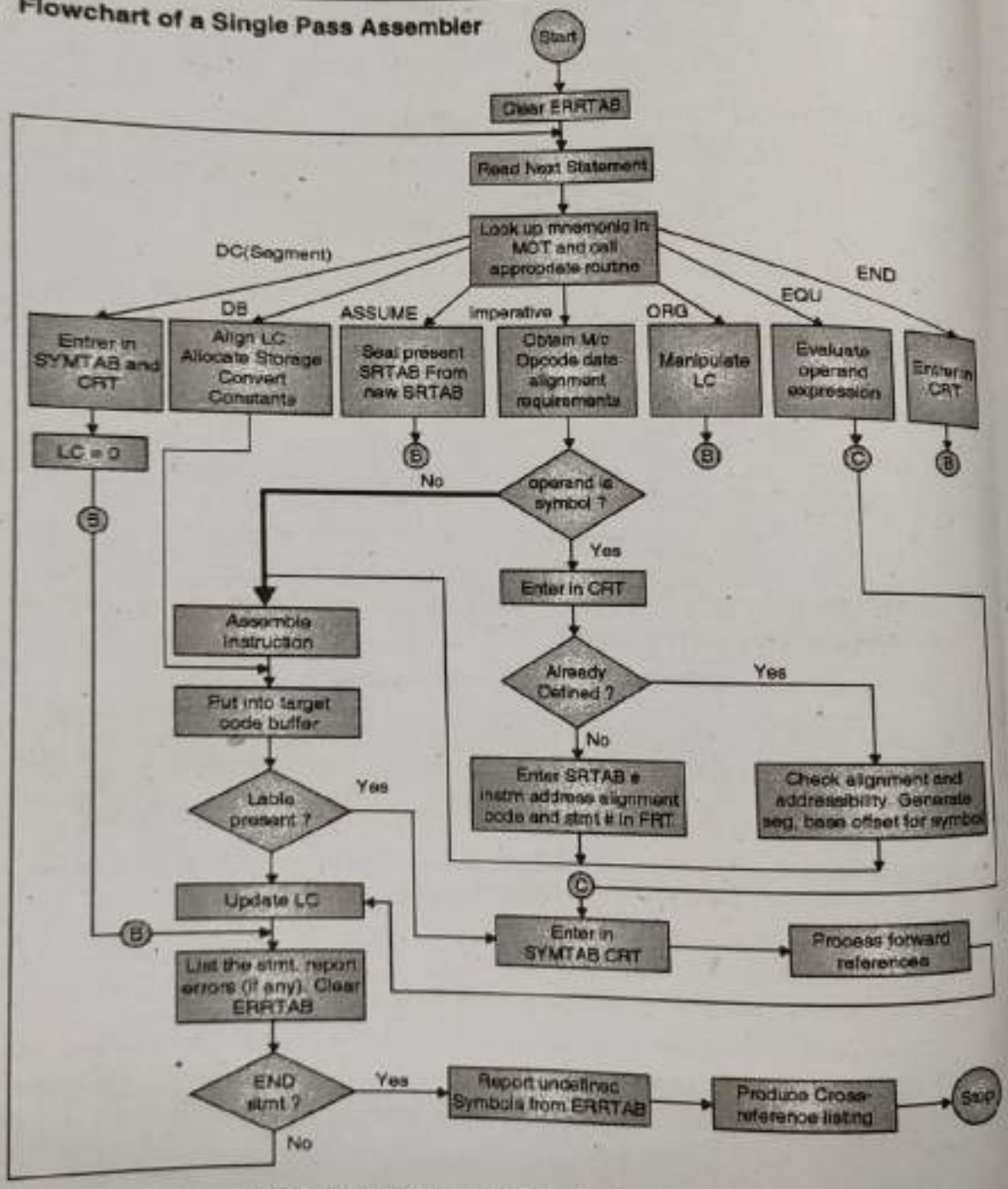


Fig. 2.8.2 : Flowchart of the Single Pass Assembler

Scanned by CamScanner

(VI) Example of One Pass Assembler using various Databases

Example 2.8.1

Table P. 2.8.1 : Example for One Pass Assembler

SRC STMT #	OFFSET	LABEL	PROGRAM	OPERAND
1	0000	XYZ	SEGMENT	
2	–	–	ASSUME	CS : XYZ (# 1),
:				DS : PQR (# 2),
:				ES : LMN (# 3)
10	0006		MOV	COUNT, 0000
:				
:				–
90	0020		ASSUME	DS : LMN (# 1)
				ES : PQR (# 2)

Solution :

Statement

SRC STMT #	OFFSET	LABEL	PROGRAM	OPERAND
1	0000	XYZ	SEGMENT	

Symbol Table Entries for above statement (SRC statement 1)

	SYMBOL	EQU (Y/N)	SEG. REG. (Y/N)	DEFINED (Y/N)	TYPE	OFFSET IN SEG.	OWNERSEG.	LENGTH (2)	SIZE (2)	SOURCE STMT # (2)	PTR. TO 1ST FRT ENTRY (2)	PTR. TO LAST CRT ENTRY (2)	PTR. TO 1ST CRT ENTRY (2)
#1	XYZ	N	Y	Y	ALIGN-MENT INFO	00 00	# 1	...		1	NULL	NULL	NULL

Next Statement (SRC Statement 2)

2	:	:	ASSUME	CS : XYZ (# 1),
				DS : PQR (# 2),
				ES : LMN (# 3)

Scanned by CamScanner

SRTAB Entries for above statement

SRTAB

SEGMENT REG.	SEGMENT NAME
ES (00)	# 3
CS (01)	# 1
SS (10)	–
DS (11)	# 2

Symbol Table Entries for statements (#2 and #3)

	SYMBOL	EQU (Y/N)	SEG-REG- (Y/N)	DEFINED (Y/N)	TYPE	OFFSET IN SEG-	OWNER SEG-	LENGTH (2)	SIZE (2)	SOURCE STMT #(2)	PTR TO 1ST FRT ENTRY (2)	PTR-TO LAST CRT ENTRY (2)	PTR-TO 1ST CRT ENTRY (2)
											NULL	NULL	NULL
#2	POR	N	Y	N							NULL	NULL	NULL
#3	LMN	N	Y	N									

Next Statement of example (SRC Statement 10)

10	0006	:	MOV	COUNT, 0000

Cross Reference Table Entries for above statement : CRT maintained as Queue

PTR - TO NXT CRTS -	SOURCE STMT #
	10

Forward Reference Table : (Maintained as Stack)

PTR to NXT FRT ENTRY (2)	ENTRY # in STSRT (1)	INSTRUCTION ADDRESS (2)	USAGE CODE (1)	SOURCE STMT # (2)
STACK TOP	# 1	0006	XXX	10

Scanned by CamScanner

SYMTAB ENTRIES

SYMBOL	EQ (Y/N)	SEG-REG (Y/N)	DEFIN ED (Y/N)	TYPE	OFFSET IN SEG	OWNER SEG	LENGTH (2)	SIZE (2)	SOURCE STMT # (2)	PTR TO 1ST FRT ENTRY (2)	PTR TO LAST CRT ENTRY (2)	PTR TO 1ST CRT ENTRY (2)
COUNT	N	N	Y	--	0020	#2	1		90	TO FRT	TO FRONT OF CRT	TO REAR OF CRT

Statement of example

90	0020		ASSUME	DS : LMN (# 1) , ES : PQR (# 2)

SRTAB Entries for above Statement

SEGMENT REG.	SEGMENT NAME
ES (00)	~~#3~~ # 2
CS (01)	# 1
SS (10)	--
DS (11)	~~#2~~ # 3

Stored Segment Register Table Entries for Statements

SEGMENT REG.	SEGMENT NAME
ES (00)	# 3
CS (01)	# 1
SS (10)	--
DS (11)	# 2

STSRT # 1

ES (00)	~~#3~~ # 2
CS (01)	# 1
SS (10)	--
DS (11)	~~#2~~ # 3

STSRT # 2

STSRTAB

Scanned by CamScanner

Sample program

```
PRG     START
        USING *, 15
        L   1, FOUR
        A   1, FIVE
        A   1, = F'2'
        ST  1, TEMP
FOUR    DC  F'4'
FIVE    DC  F'5'
TEMP    DS  1F
        END
```

First Pass

Relative Address	Mnemonic Instruction
0	L 1, –(0,15)
4	A 1, – (0, 15)
8	ST 1 – (0,15)
12	4
16	5
20	
24	–

Second Pass

Relative Address	Mnemonic Instruction
0	L 1, 12 (0,15)
4	A 1, 16 (0, 15)
8	ST 1, 20 (0,15)
12	4
16	5
20	
24	–

Symbol Table

Symbol (8 Bytes)	Value (4 Bytes)	Length (1 Byte)	Relocation (1 Byte)
PRG	0	01	R
FOUR	12	04	R
FIVE	16	04	R
TEMP	20	04	R

Scanned by CamScanner

Literal Table

Literal	Value	Length	Relocatable
= F'2'	28	04	R

Base Table

	Availability Indicator	Contents Of Base Reg
0	N	
1	N	
. . .	. . .	
15	Y	00

Review Questions

Q. 1 What features of assembly language required us to build a multi pass assembler ?

Q. 2 Define the following terms.

a) Imperative Statements

b) Declarative Statements

c) Assembler Directives

Q. 3 Write a short note on assembler functions.

Q. 4 What is a Forward Reference Problem and how it is solved in one pass and two pass assembler ?

Q. 5 Explain the databases used in one pass assembler.

Q. 6 Explain the format of databases used in multi pass assembler.

Q. 7 What are advantages and disadvantages of two pass assembler ?

Q. 8 For the following program:

a) Show the symbol tables at the end of pass 1.

b) Show the literal tables at the end of pass 2.

c) Show the changes in the base register table during pass 2.

d) Show the generated "machine" code from pass 2.

Scanned by CamScanner

SIMPLE	START	15, 0
	BALR	*, 15
	USING	R1, TWO
LOOP	L	R1, TWO
	A	R1, FOUR
	ST	FOUR + 3, 4
	CL1	LOOP
	BNE	14
	BR	1
R1	EQU	F '2'
TWO	DC	F
FOUR	DS	
	END	

Q. 9 Explain with neat flowchart and database, the working of two pass assembler.

Q. 10 Using any assembler language, write a sample assembly program, with respect to that program, describe how a 2 pass assembler will translate that.

Q. 11 Explain with the help of databases of each of the passes of two pass assembler.

Q. 12 Give the analysis and design of two pass assembler with respect of flowchart, data structures and algorithms.

Q. 13 What is assembly language ? What features of assembly language required us to build a two pass assembler ?

Q. 14 Following program is for 8086 processor. Give result for program after first pass and second pass of assembler with relative address of each instruction.

SAMPLE	START	4000
	USING	*, BX
	MOV	BL, NUM1
	ADD	BL, NUM2
	MOV	RESULT, BL
	DC	10H
NUM1	DC	20H
NUM2	DS	?
RESULT	END	

Q. 15 Give analysis and design of single pass assembler with respect to flowchart, data structure and algorithm.

Chapter End

Scanned by CamScanner

Module 3

CHAPTER 3

Macros & Macro Processor

Syllabus Topics

Introduction, Macro definition and call, Features of Macro facility : Simple, parameterized, conditional and nested. Design of single pass macro processor, data structures used.

3.1 Introduction

A macro processor helps us to define and use macros in an assembly level language. During macro definition a group of statements is provided that we want to associate with a macro name. The assembler replaces the macro name with the statements specified in the macro definition.

Macros provide several advantages when writing assembly programs.

- The frequent use of macros can reduce the programmer-induced errors. A macro allows us to define an instruction sequences that are used repetitively throughout the program. This reduces the likelihood of errors introduced in repetitive programming sequences.
- The scope of symbols used in macro is limited.
- Macros are well suited for creating simple code tables
- Macros make an assembly program more readable.

The assembly language programmer often finds necessary to repeat some blocks of code many times. The block may consist of series of operation like addition, subtraction etc. again and again. In this situation the programmer will find macro instruction facility very helpful and useful.

3.2 Macro Definition and Call

Q.	Explain when will a macro be used in a program ?	(5 Marks)
Q.	Define macro.	(2 Marks)
Q.	Explain the concept of macro definition within a macro.	(5 Marks)

Macro (meaning "large" or "wide"). A macro is a block of code which has been given a name. Whenever the name is invoked it is replaced by the interpreter or compiler by the contents of the macro.

Scanned by CamScanner

A single line meaning for group of statements is called macro and it is a notational convenience for the programmer. Macro instruction replaces a macro with a group of statements whenever they are called. This activity of replacing the code is called **expansion of macros**.

In complied languages, macro-expansion happens at compile time. Thus macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macro processor.

- The purpose of macros is to either automate frequently used sequences or enable a more powerful abstraction.
- Macro processors are viewed as an extension of the basic assembler algorithm.
- Macro processors are not directly related with the architecture of the computer on which it is run.
- It is used with high level languages, operating system command languages.
- Macro replaces each instruction with the corresponding group of statements. For e.g.
- SIC requires a sequence of instructions to save the contents of all registers. This can be done by writing one macro like save registers etc.

3.2.1 Functions of MACRO Instructions

The macro processor is used to

1. Identify the macro definition
2. Save the macro definition
3. Identify the macro calls
4. Expand the macro calls.

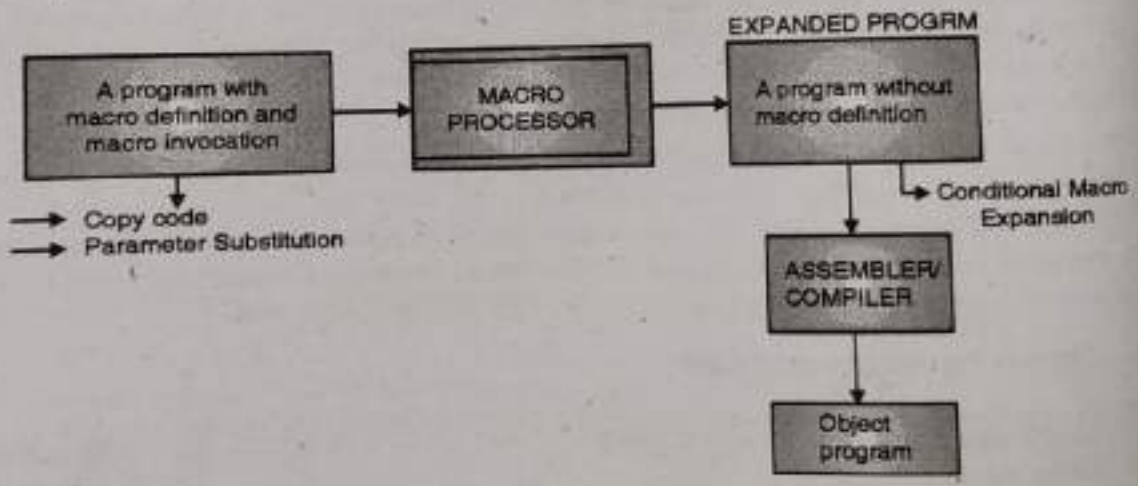


Fig. 3.2.1 : Functions of MACRO Processors

Scanned by CamScanner

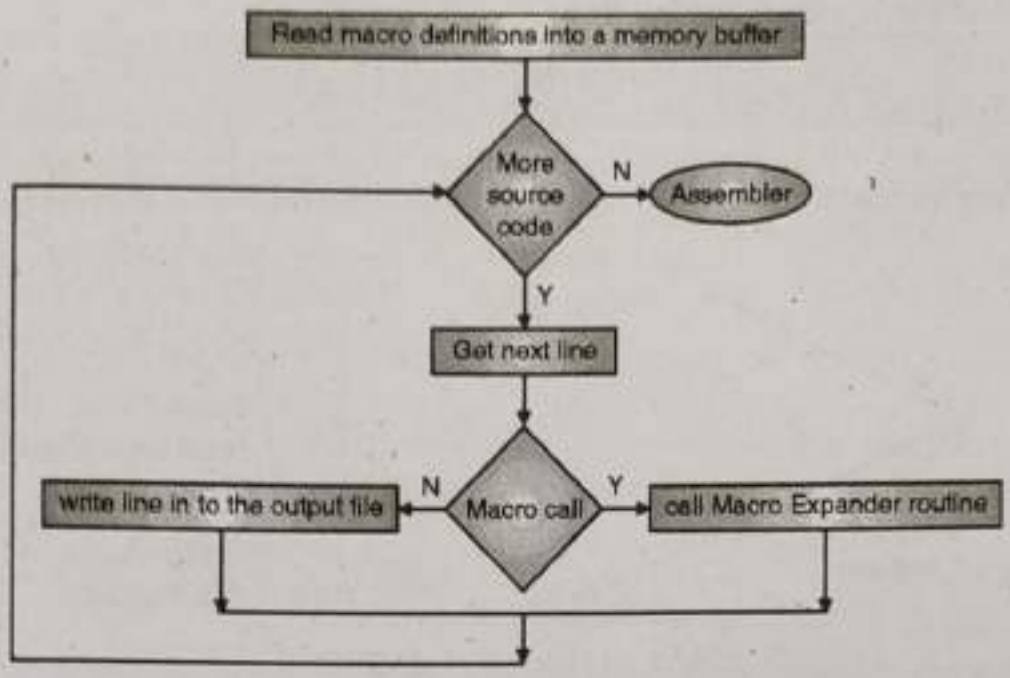


Fig. 3.2.2 : A macro processor front end for an assembler

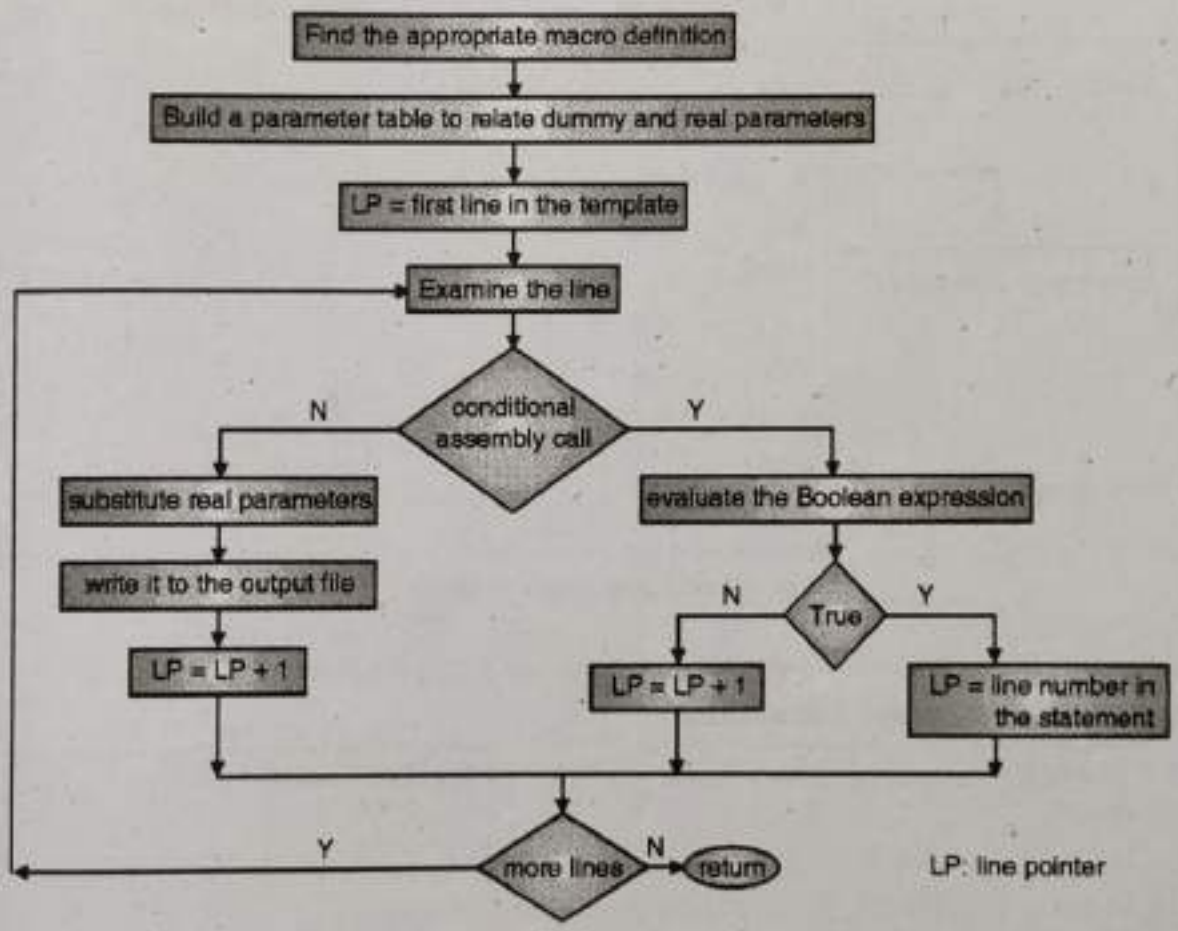


Fig. 3.2.3 : Macro expander routine

Scanned by CamScanner

3.2.2 Example of Macro Instruction

Q. Explain macro calls within macro giving example. **(5 Marks)**

Q. Explain the concept of macro definition within a macro. **(5 Marks)**

A **macro** instruction is an abbreviation for a sequence of operations. The macro processor replaces each macro instruction with a group of source language statements. This is called **expansion of macro**.

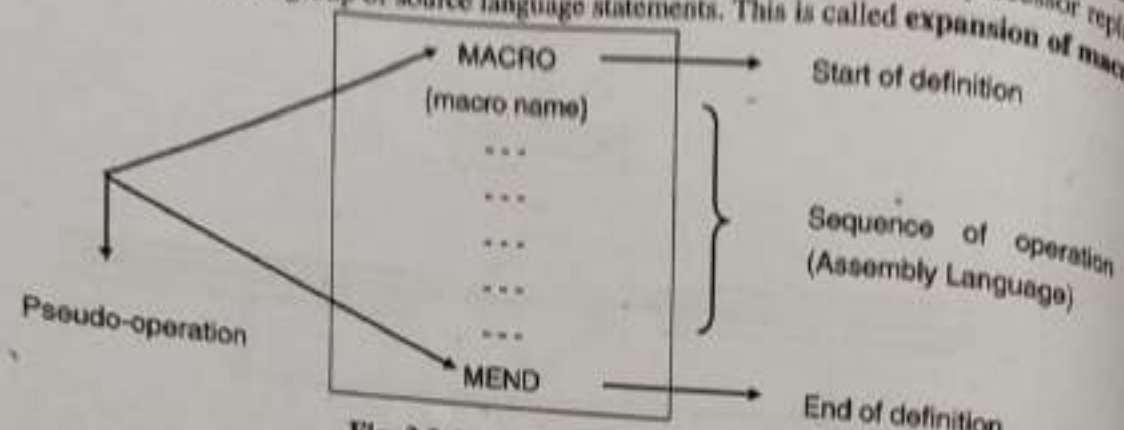


Fig. 3.2.4 : Format of MACRO

1. Copy Code : Example

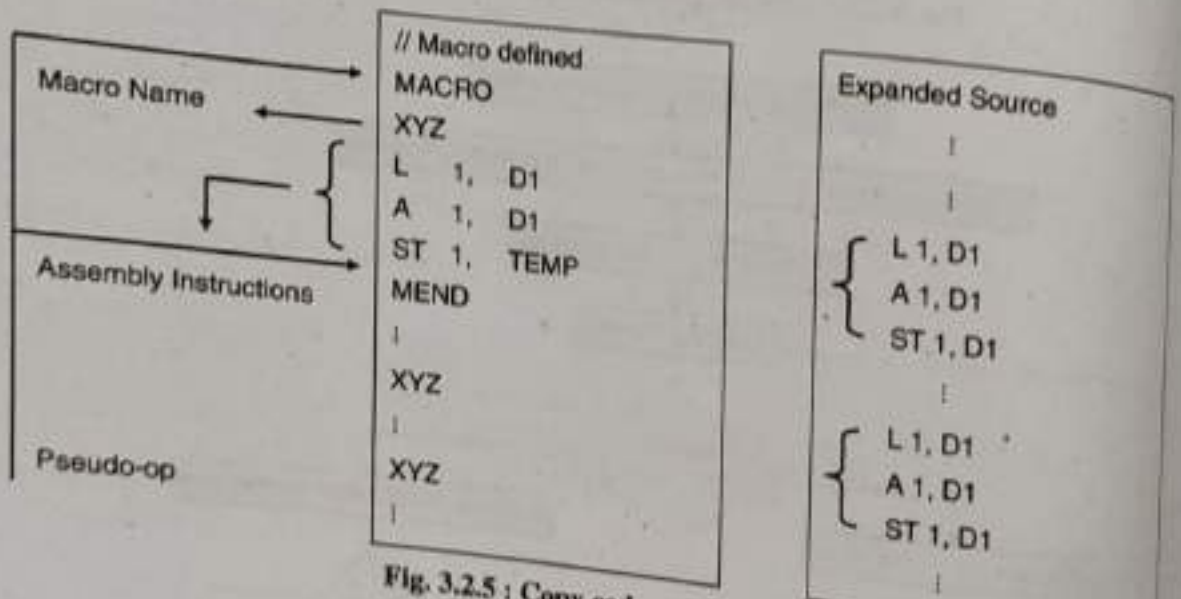
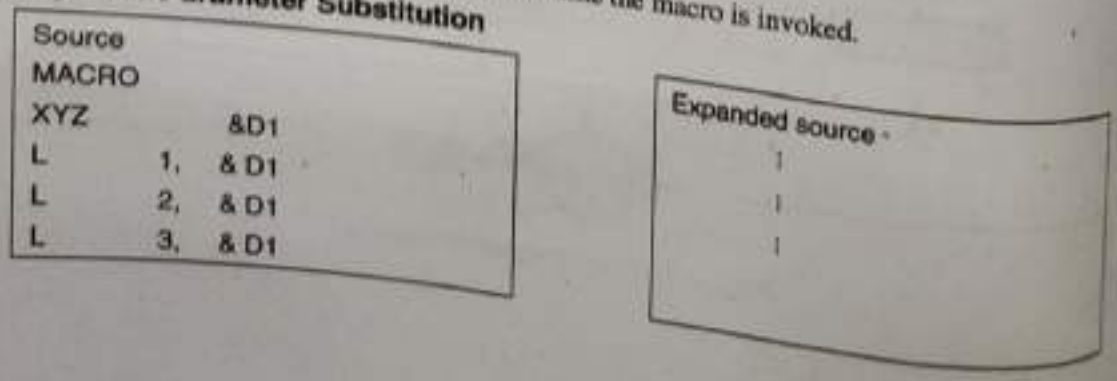


Fig. 3.2.5 : Copy code - example

The statements of expansion are generated each time the macro is invoked.

2. Example for Parameter Substitution



Scanned by CamScanner

```
MEND
:
INCR        RECORD1
:
INCR        RECORD2
:
DATA1  DS   F' 40'
DATA 2 DC   F' 50'
```

```
L   1,  RECORD1
L   2,  RECORD1
L   3,  RECORD1
        |
L   1,  RECORD2
L   2,  RECORD2
L   3,  RECORD2
        |
DATA 1  DS  F' 40'
DATA 2  DC  F' 50'
```

Fig. 3.2.6 : MACRO XYZ has single argument

3.3 Features of Macro Facility

→ **(MU - May 2014)**

Q.	What are the features of macro to be included in the processors?	**May 2014 5 Marks**
Q.	Detail the different features used in macro processing.	**(10 Marks)**

1. Simple Macro Processor

In a simple macro processor each macro processor is expanded with its code as shown in the Fig. 3.3.1.

```
// Macro defined
MACRO
TEST
L   1,   F1
A   1,   F1
ST  1,   SWAP
MEND
:
TEST
:
TEST
:
TEST
```

```
Expanded Source
        :
        :
    L 1, F1
    A 1, F1
    ST 1, SWAP
        :
    L 1, F1
    A 1, F1
    ST 1, SWAP
        :
    L 1, F1
    A 1, F1
    ST 1, SWAP
```

Fig. 3.3.1 : Example of Simple Macro

Scanned by CamScanner

2. Parameterised Macro / Positional Macro (Macro definition with one, two or many arguments)

→ (MU - Dec. 2014)

Dec. 2014, 5 Marks

Q. Explain positional parameter in macro.

Q. Explain the different ways of parameter passing in macros. (5 Marks)

A parameterised macro is a type of macro that is able to insert given objects into its expansion. Parameterized macro is not able to modify the instruction that replaces the call. To solve this problem we use macro calls within arguments, parameters, corresponding to these dummy arguments will appear in macro definition. Parameterized macros are useful mechanism for performing in line expansion.

Source

```
:
MACRO
&STRG  LAB    &A1, &A2, &A3
&STRG     A   1, &A1
          A   2, &A2
          A   3, &A3
       MEND
:
LOOP1  LAB    RECORD1,
              RECORD2, RECORD3
              DATA3
:
:
LOOP2  LAB    RECORD3,
              RECORD2,
              RECORD1
:
:
DATA1  DC     F'6'
DATA 2 DS     F'12'
DATA 3 DS     F'24'
:
```

Expanded source

```
:
:
:
LOOP1     A  1,   RECORD1
          A  2,   RECORD2
          A  3,   RECORD3
:
LOOP2     A  1,   RECORD3
          A  2,   RECORD2
          A  3,   RECORD1
:
:
DATA 1    DC F'6'
DATA 2    DS F'12'
DATA 3    DS F'24'
```

Fig. 3.3.2 : Example of parameterized macro

In mentioned example we have specified four arguments including **STRG** argument. In parameterised macros the parameters and arguments are associated with each other according to their positions in the macro prototype and the macro invocation statements.

Scanned by CamScanner

A certain macro instruction GENER has 10 possible parameters.

E.g. GENER MACRO &1, &2, & type - - - - - - - &10

GENER , , DIRECT, 3

There are generally three ways of specifying arguments to a macro call.

a) Positional arguments

In this arguments are matched with the dummy arguments according to the order in which they appear with positional parameter the programmer must be careful enough to specify the arguments in the proper order.

E.g. STRG, DATA1 DATA2, DATA3

GENER, DIRECT, TYPE etc.

b) Keyword arguments

They allow reference to the dummy arguments by name as well as by the position. This is useful when a macro has arguments that are not always used.

STRG a3 = DATA1, a2 = DATA2, a1 = DATA3

c) Default arguments

It is an argument to a function that a programmer is not required to specify. In most programming languages function may take one or more arguments E.g.

define min (x, y) ((x) < (y) ? (x) : (y))

x = min (a, b); ⇒ x = ((a) < (b) ? (a) : (b));

y = min (1, 2); ⇒ y = ((1) < (2) ? (1) : (2));

3. MACRO Calls within MACROS / NESTED MACROS

In this feature of macro facility when one MACRO expansion is taking place we can encounter another macro call or during one macro call another macro call is invoked. In this macro bodies contain macro calls and processes macro definition during expansion time.

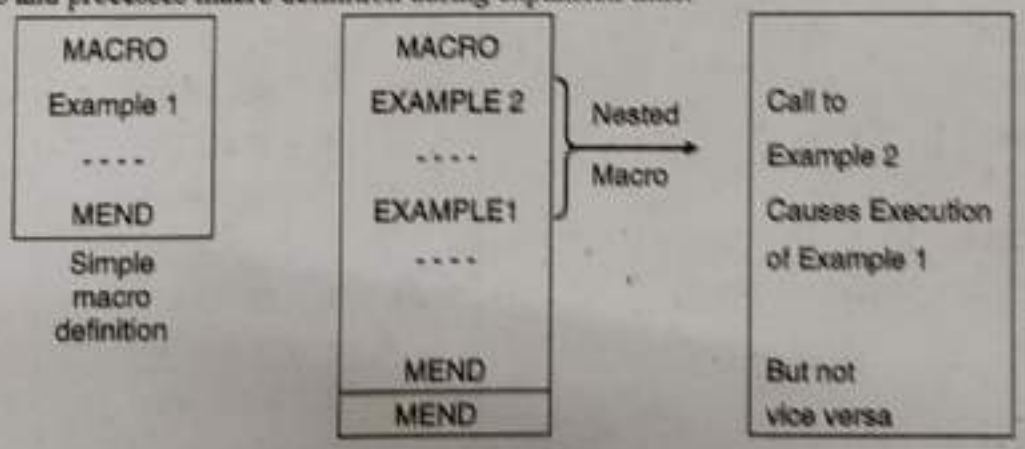


Fig. 3.3.3 : MACRO CALLS WITHIN MACROS

Scanned by CamScanner

Call to
Example 1
causes execution
of Example 2
AND we
cannot call
Example 2
as it is

```
MACRO
Example 1
----
----
MACRO
Example 2
----
----
MEND
----
----
MEND
```

Fig. 3.3.4 : Example of NESTED MACROS

Source

```
MACRO
INSIDE      & A1
L           1, &A1
A           1, = F' 5'
ST          1, &A1
MEND
MACRO
OUTSIDE     &D1, & D2,
            & D3,
INSIDE      &D2
INSIDE      &D3
INSIDE      &D1
MEND
  ⋮
MIDDLE
            DATA1, DATA2,
            DATA3
  ⋮
DATA1       DC  f' 5'
```

Expansion

```
Expansion of MIDDLE
        ⋮
INSIDE  RECORD2
INSIDE  RECORD3
INSIDE  RECORD1

EXPANSION OF INSIDE
L   1,  RECORD2
A   1  = F' 5'
ST  1,  RECORD2

L   1,  RECORD3
A   1, = F' 5'
ST  1,  RECORD3

L   1,  RECORD1
A   1, = F' 5'
ST  1,  RECORD1
```

Fig. 3.3.5 : Example of NESTED MACROS

Scanned by CamScanner

Within the definition of the macro 'OUTSIDE' are three separate calls to a previously macro 'INSIDE'. The use of the MACRO 'MIDDLE' has shortened the length of the definition of 'MIDDLE' and thus has made it more easily understood. Macro calls within macros can involve several level.

Macro calls within Macros

- Use a stack to keep the order of the macro calls.
- Create a parameter table at each macro call.
- If a dummy parameter appears in the real parameter field of a called macro, search the parameter table of the calling macro and replace it with the appropriate real parameter.

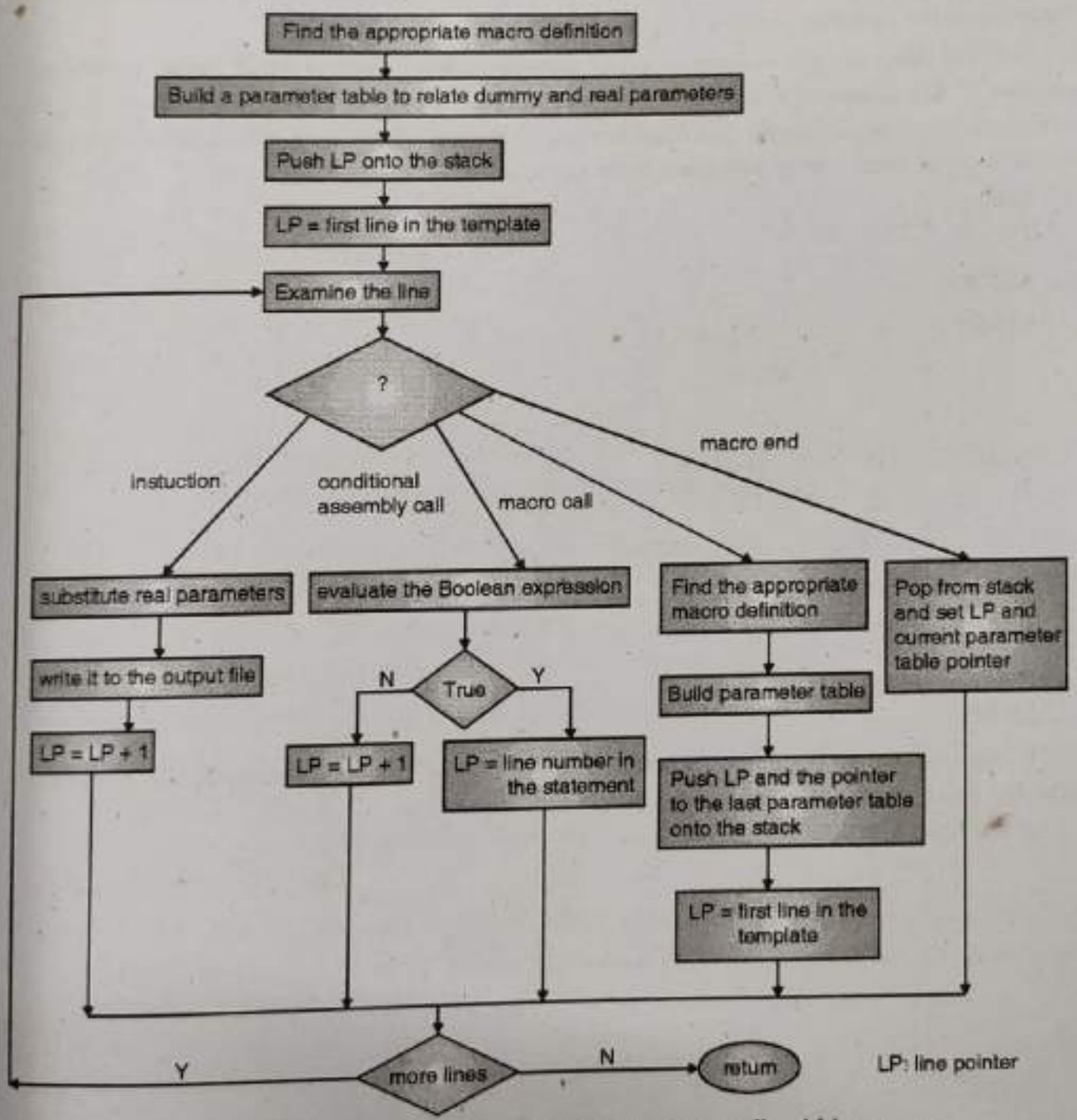


Fig. 3.3.6 : Macro expander routine that allows macro calls within macros

Scanned by CamScanner

4. Conditional MACRO Expansion

→ (MU - Dec. 20

Q. Explain different pseudo-ops used for conditional macro expansion along with an example. Dec. 2015. 10 Marks

In our previous example of macro instruction, each invocation of a particular macro was expand into the same sequence of statements. These statements could be varied by the substitution parameters, but the form of the statements, and the order in which they appeared was unchanged.

Most macro processors can modify the sequence of statements generated from a macro expansion depending on the arguments supplied in the macro-expansion.

AIF and AGO are two conditional macro expansion pseudo opcodes which permit conditional selection of the sequence of the machine instruction that appear in expansion of macro call. In conditional macro expansion only part of the macro is copied out into the code. Which part is copied will be under the control of the parameters in the macro call.

Example

```
:
MACRO
START 1     D    1,    VALUE 1
            D    2,    VALUE 2
:
START 2     D    1,    VALUE 1
            2,    VALUE 2
:
VALUE 1     DC        F' 0'
VALUE 2     DC        F' 1'
:
MEND
```

In this example the labels, the operands and the number of instruction generated change in each sequence. This example can be written as :

Scanned by CamScanner

Source

```
  ⋮
MACRO
& AG  DATA &C, &A2, &A3, & ARG2, &ARG 3
& AG        D       2, & A2
            AIF     (&C EQ 1).FINI
                            (TEST if C = 1)
            D       3, &A3
            AIF     (&C EQ 2).FINI
                            (TEST if C = 2)

MEND
                    ⋮
                    ⋮
START 1     DATA    2, VALUE2, VALUE3
             ⋮
START 2     DATA     1, VALUE 2
                     ⋮
VALUE 1     DC       F'0'
VALUE 2     DC       F'1'
  ⋮
MEND
```

Expanded source

```
        ⋮
        ⋮
        ⋮
        ⋮
        ⋮
        ⋮
        ⋮

START 1     D  2, VALUE 2
            D  3, VALUE 3

START 2     D  2, VALUE 2

        ⋮
```

Fig. 3.3.7 : Example of CONDITIONAL MACROS

- **AIF :** Arithmetic operations are performed by AIF and if nested condition is true then only they are branched. It is conditional opcode.
- **AGO :** It is similar to 'go-to' statement in C/C+ +. It is an unconditional pseudo-opcode. It specifies a label appearing on some other statements in the macro definition.
- The flow of program during macro expansion is controlled by AIF and AGO.
- Other than AIF and AGO we also use IF-ELSE-ENDIF and WHILE-ENDW structure for the conditional branching.
- Symbol table is maintained to store the values of all macro variables. This table is used to look up the current value of a macro.
- Labels with a period (·) such as · FINI do not appear in the output of the macro processor.
- AIF (&C EQ 1)· FINI is a statement which guides the macro processor not to execute the statement labelled· FINI if the value corresponding to C = 1.

Scanned by CamScanner

The macro time IF-ELSE-ENDIF structure

It provides the mechanism for either generating or skipping selected statements in the macro body. Whenever an IF statement is found during the expansion of macro the value is computed in terms of Boolean expression. If the executed value is true IF block is executed otherwise Else is evaluated.

The macro time looping WHILE-ENDW

The **While** conditional statement is useful because it allows us to repeat a block of code as long as a conditional test is **True**, rather than just a simple number of times. It is useful in situations where we need to loop through code, but do not know how many times we need to run the portion of code.

E.g. of WHILE - ENDW Structure

```
MACRO
WHILE (N! = 0)
N = N % 10
N = N / 10
SUM = SUM + N
ENDW
:
MEND
```

Explanation for the given example

1. The expression is evaluated.
2. If expression is initially false, the body of the while statement is never executed, and control passes from the while statement to the next statement in the program.
3. If expression is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.
4. As soon as ENDW statement is found, the macro processor checks the Boolean expression by returning the call to the WHILE, and action is taken based on the new value of the expression.

Examples on conditional macro expansion

➢ **Example 3.3.1**

The following macro called DOIO has three dummy parameters, namely &DEV, &BUF and &CODE. If the actual parameter corresponding to &CODE is R, then the expansion of the macro results in statements that read one character from the device specified by &DEV and store the character in the memory location (byte) specified by &BUF. If the actual parameter corresponding to &CODE is not R, then the expansion of the macro results in statements that write the character stored in the memory location (byte) specified by &BUF to the device specified by &DEV.

```
DOIO MACRO          &DEV, &BUF, &CODE
Initialize register A depending upon &CODE.
        IF          (&CODE EQ 'R')
        CLEAR  A
        ELSE
```

Scanned by CamScanner

```
        LDCH        &BUF
        ENDIF
Generate RD or WD instruction depending on &CODE.
$CHECKTD            =X' &DEV'
        JEQ         $CHECK
        IF          (&CODE  EQ  'R')
        RD          =X' &DEV'
        STCH        &BUF
        ELSE
        WD          =X' &DEV'
        ENDIF
        MEND
```

Notes :

1. When the DOIO macro is invoked by the statement

   DOIO A5, LOC, R

   The expanded version of the macro is as follows :

```
            CLEAR     A
AACHECK     TD        =X' A5'
            JEQ       AACHECK
            RD        =X'A5'
            STCH      LOC
```

2. When the DOIO macro is invoked a second time by the statement

   DOIO A7, LOC, W

   The expanded version of the macro is as follows :

```
            LDCH      LOC
ABCHECK     TD        =X'A7'
            JEQ       ABCHECK
                      WD = X' A7'
```

3. When the macro is invoked using the statement

   DOIO A7, LOC

   The expansion would be similar to that in (2) above. (In this case, the SIC/XE assembler treats the value of the parameter &CODE as the null character.)

➢ **Example 3.3.2**

The purpose of this example is to point out a common mistake in writing macros. Suppose you are asked to write a macro CHANGET that sets register T to 0 if register A contains 0; otherwise, it sets register T to – 1.

Scanned by CamScanner

A wrong solution : (Why is this wrong ?)

```
CHANGET     MACRO
            IF        (A EQ 0)
            CLEAR     T
            ELSE
            LDT       #-1
            ENDIF
            MEND
```

A correct solution : (Does not use conditional expansion)

```
CHANGET     MACRO
            CLEAR     T
            COMP      #0
            JEQ       $EXIT
            LDT       #-1
$EXIT       OR        #0        #This is a nop.
            MEND
```

5. Recursive MACRO Calls

It allows the macro invocation statements appearing within the body of a macro instruction. Macros that invoke themselves are called **recursive macros**. If macros are calling themselves, one speaks of recursive macro calls. In this case, there must be some stop criterion, to prevent the macro of calling itself over and over until the assembler is running out of memory. Recursive macros executes until some command in its definition command sequences fails. To write a recursive macro we need to have

- Recursive function.
- Termination condition.

E. g.

```
PROGRAM  MACRO        &PROG1, &INPUT
                      / * Macro to read record  * /
         CLEAR        X    (Clear loop Counter)
&DATA    RDCHAR       &INPUT Read characters into Reg A
         :
         JEQ          &QUIT  Exit loop if END OF RECORD
         JLT          & DATA
&QUIT    MEND
```

(a) RECURSIVE MACRO

```
RDCHAR   MACRO   & INPUT1
         TD      = X '& IN'   /*Test Input Device*/
         :
         :
         MEND
```

(b) RDCHAR

Fig. 3.3.8

Scanned by CamScanner

- In Fig. 3.3.8(a) recursive macro has defined.
- RDCHAR in (Fig. 3.3.8(b)) calls recursive macro INPUT1 and INPUT1 exits from the loop as end of record is found. We can see that macro &DATA is also called.
- The procedure EXPAND would be called when the macro was recognized and expansion of macro begins.
- When the end of RDCHAR will known, expansion of macro call would stop this will cause problem because macro processor would 'forget' that it had been in the middle of the expansion of macro PROG1
- Macro-processors written in high level language (PASCAL OR C) does not have such problems as they allow recursive macro calls. Example of recursive macro in C

```
#define min(a, b) (((a)<(b))?(a):(b))
main()
{
    int w, x, y, z, small;
    //Read w, x, y, z.
    small=min(w, min(x,min(y,z)));
    printf("Minimum of 4 numbers is %d", small);
}
```

- If a programming language does not support macro calls then there must be return address.

6. MACRO Instructions Defining MACROs

A single macro instruction might be used to simplify the process of defining a group of similar macros. To call the inner macro it is necessary to define the outer macro first.

Definition of MACRO DEFINE	MACRO DEFINE	& ADD	Macro Name DEFINE
Definition of MACRO ADD	MACRO & ADD	&X	Dummy MACRO NAME
	CNOP	0, 5	Alignment of boundary
	BAL	3 * + 10	Set register 3 to parameter list pointer
	DC	B(&X)	Parameter list pointer
	L	20, = V (& ADD)	Address of subroutine ADD
	BALR	19, 20	call transferred to subroutine
End of MACRO	MEND		
	MEND		

Scanned by CamScanner

Let us call this macro with

```
DEFINE    SIGN
```

A new macro call is created by

```
SIGN      ARG
```

and the generated calling sequence by macro processor A is

```
BAL     3, * + 10
DC      A (ARG)
L       20, = V (SIGN)
BALR    19, 20
```

3.4 Implementation of Macro Processors

→ (MU - May 2014)

Q. Explain the working of a macro processor with neat flow charts and data bases. **May 2014, 10 Marks**

The four basic steps that any macro processor should perform are as :

1. **Recognize Macro Definitions**

   Proper match of all the nested and recursive macro calls should be done and macro processor should identify macro definitions given by op-codes MACRO and MEND.

2. **Save the Definitions**

   To expand the macro calls and substituting the expanding source code a macro processor should store the macro definitions.

3. **Recognize Calls**

   The processor should recognize all the macro calls that appear as mnemonic op-codes.

   In this all the macro calls are identified from various mnemonic op-codes.

4. **Expand Calls and Substitute Arguments**

   Macro definition arguments are replaced by the corresponding macro calls.

   In this macro processor has to take a different decision like :

   - Able to identify dummy arguments in macro definition.
   - Syntax and semantics of all the allowed arguments.

3.5 Design of

It processes all m

1. **Macro Name Ta**
2. **Macro Definitio**
3. Auxiliary inform in MDT.
4. All the statement
5. For each macro
   (a) Macro nam
   (b) Informatio
   (c) All macro
   (d) Expansion

Process the stat the MEND statemen relations between va

3.5.1 Data St

Three main dat

1. DEFTAB or MI

- It stores t
- Comment
- Referenc efficienc

2. NAMTAB or M

- It store and end

3. ARGTAB

- Used du
- When a accordi

3.5.2 Algor

1. The macro n
2. NAMTAB c

Scanned by CamScanner

3.5 Design of Single Pass Macro Processor

It processes all macro definitions for each macro name defined.

1. **Macro Name Table (MNT) :** It is used to enter macro name.
2. **Macro Definition Table (MDT) :** It is used to store the entire macro definitions.
3. Auxiliary information is added to the MNT. It indicates where the definition of macro can be found in MDT.
4. All the statements are processed in the assembly source program to detect all macro calls.
5. For each macro call defined in macro definitions:
   (a) Macro name is located in the MNT.
   (b) Information is obtained from the MNT regarding the location of macro definition in MDT.
   (c) All macro calls are processed to build balance between all formal parameters and their values.
   (d) Expansion of all macro call is done by the process given in step 3

Process the statements in the macro definition as found in MDT in their expansion time order until the MEND statement is encountered. AIF and AGO will make changes in the certain expansion time relations between values of formal parameters and expansion time variables.

3.5.1 Data Structures used in One Pass Macro Processor

Three main data structures involved in a one-pass macro processor are as:

1. DEFTAB or MDT

- It stores the macro definition including macro prototype and macro body.
- Comment lines are omitted.
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

2. NAMTAB or MNT

- It store macro names, which serves an index to DEFTAB contain pointers to the beginning and end of the definition.

3. ARGTAB

- Used during the expansion of macro invocations.
- When a macro invocation statement is encountered, the arguments are stored in this table according to their position in the argument list.

3.5.2 Algorithm Execution Steps

1. The macro names are entered into NAMTAB,
2. NAMTAB contains two pointers to the beginning and the end of the definition in DEFTAB.

Scanned by CamScanner

3. The third data structure is an argument table ARGTAB, which is used during the expansion of macro invocations.
4. The arguments are stored in ARGTAB according to their position in the argument list.

3.5.3 Format of Databases

1. In MDT, if we store the definition in its source form, then each statement would have to be processed during expansion to determine whether a formal parameter or an expansion time variable has been referenced to it. At the time of entering a statement in MDT, any reference to formal parameter or expansion time variables would be replaced by index markers #.
2. **Actual parameter list :** For every macro call an APL is formed by processing the macro call statement. This list contains the actual parameters corresponding to formal parameters in the MDT. The actual parameter would be determined from the macro call and the default specifications for the keyword parameters.

Table 3.5.1 : Actual parameter list

	APL
# 1	M
# 2	N
# 3	MULT

3. **Macro Name Table (MNT) :** It specifies the number of positional and keyword parameters for the macro using the KPT field of MNT we can find the keyword parameters in KPT and their default values.

Table 3.5.2 : Macro name table

Macro Name	MDTP	#PD	#KP	#KDTP	
EVAL	26	2	2	6	0

4. **Keyword Parameter Table (KPT) :** for the keyword parameters in the macro KPT is prepared which would be used to store the default values and would be used if they are not passes when the macro is called.

Table 3.5.3: Keyword parameter table

KPT

	Keyword	Default
# 6	OP	MULT
# 7	LAB	–

Scanned by CamScanner

5. **Expansion Variable Storage (EVS) :**When the macro is called, EVS is allocated for all the local variables and used in the macro Expansion Time Variable Storage (EVS).
6. **Macro Expansion Counter (MEC) :**MEC is maintained at the time of macro expansion which would point in the statements stored in MDT.

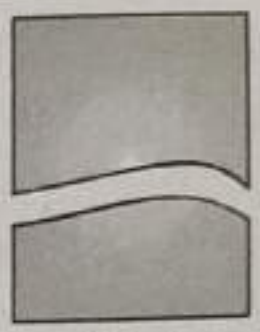


Fig. 3.5.1 : Expansion variable storage

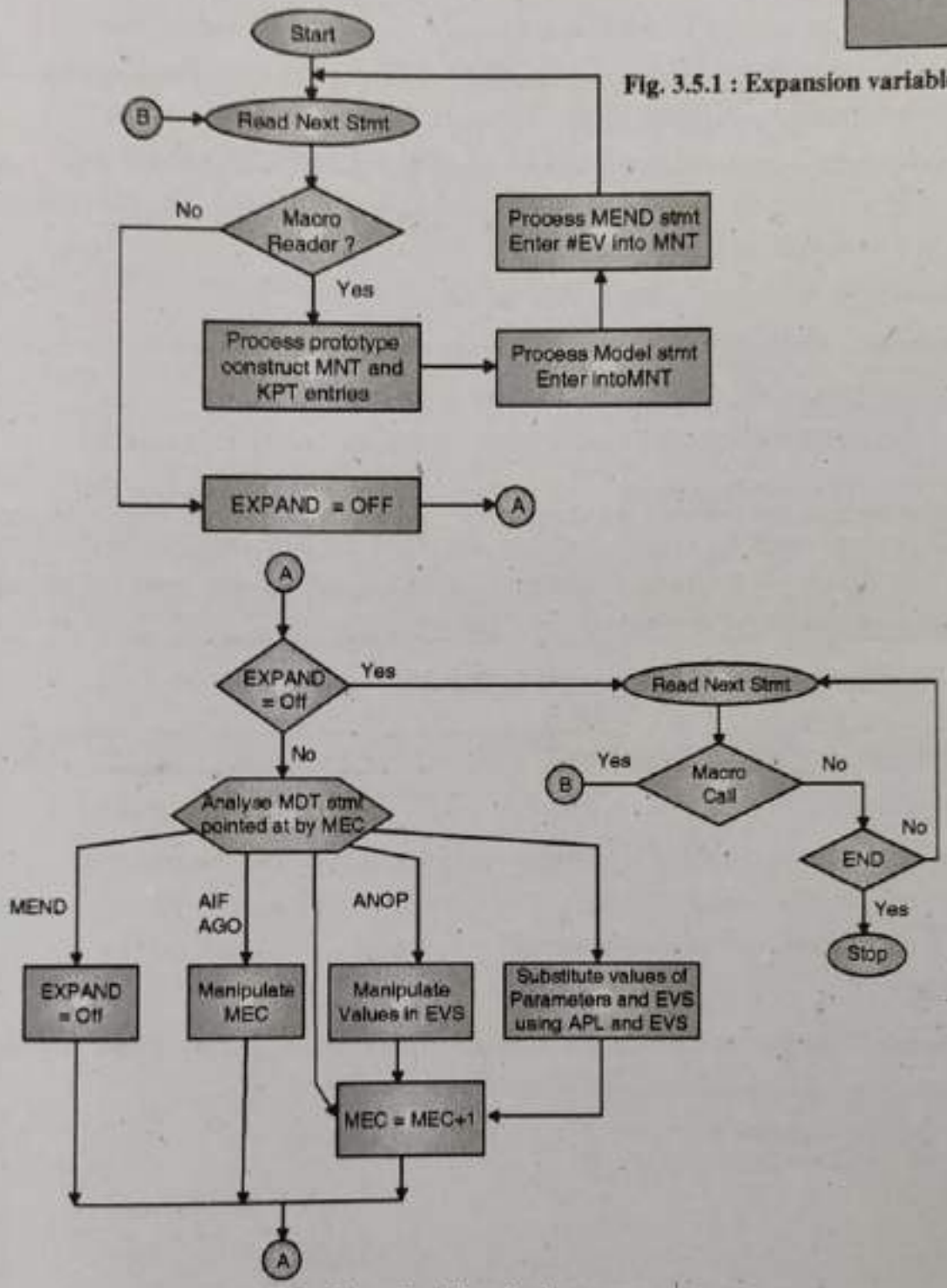


Fig. 3.5.2 : Flow chart for single pass macro processor

Scanned by CamScanner

3.6 Difference between MACROS and FUNCTIONS

1. A macro just replaces the code assigned to it for e.g. if a macro 'P1' is defined with a value 3.14. Function does not replace code but it performs what it is supposed to do.
2. Macro are faster because it avoids function call overhead.
3. No type restriction is placed in arguments so that one macro may serve for several data types.
4. Macros check the number of arguments; they do not check argument types. Functions checks both number of arguments and arguments type.
5. The macro replaces the shorthand text, with the longer definition. If a macro is used twenty times in code in all instances, the longer definition will be substituted in and thus code grow. A function definition occurs only once.
6. Macros do not have return type but a function has a return type.
7. Debugging in macro is difficult as compared to function as size of the code grows.
8. Macro should be used for small tasks and functions for the complex tasks.
9. Macro does not alters the flow of execution but the function alters the flow of execution.
10. Macro calls a process at the translation time function calls a process at the execution time.

Note : The only difference between function and subroutine is that function returns a value but subroutine does not. A function should not change the values of the actual arguments whereas a subroutine changes the values of the actual arguments.

3.6.1 Difference between Macro and Subroutine

Q. How is macro different from subroutine ? **(5 Marks)**

Macro

- Macros can only be used in the program they are defined in and only after the definition.
- Macros can take max 9 parameters.
- Macros are expanded at compilation / generation.

Subroutine

- Subroutines (FORM) can be called from both the program they are defined in and other programs.
- Subroutines can take any amount of parameters.
- Subroutines are 'expanded' at runtime

Scanned by CamScanner

3.7 Difference between Procedure Calls and Macro Calls

Q. Explain difference between : Procedure calls and macro calls. **(10 Marks)**

Sr. No.	Procedure Calls	Macro Calls
1.	A procedure call or subroutine (function, procedure, or subprogram) or a sequence of code which performs a specific task, as part of a larger program that is grouped as one or more statement blocks with the typical intention of doing one thing well.	A macro is a group of repetitive instructions in a program which are codified only once and can be used as many time as necessary.
2.	In procedure the passage of parameters is not possible.	In macro the passage of parameters is possible.
3.	Procedure is an extension of a determined program.	Macro is a module with specific functions which can be used by different programs.
4.	To call a procedure the use of a directive is required.	The call of macros is done as if it were an assembler instruction.
5.	Syntax of Procedure: CALL Name of the procedure. The part which make a procedure are : Declaration of the procedure Code of the procedure Return directive Termination of the procedure	Syntax of Macro: Declaration of the macro Code of the macro Macro termination directive The declaration of the macro is done the following way: Name Macro MACRO [parameter 1, parameter 2...]

6.	Example of Procedure Call:
	Adding ProcNear ; //Declaration of the procedure
	Mov Bx, 0 ; //Content of the procedure
	Mov B1, Ah
	Mov Ah, 00
	Add Bx, Ax
	Ret ; //Return directive
	Add Endp ; //End of procedure declaration
		Example of Macro Call ;
		Position MACRO // Macro name
		PUSH AX
		PUSH BX
		PUSH DX
		MOV AH, 02H
		MEND // End of Marco

Scanned by CamScanner

Review Questions

Q. 1 Explain the recursive macro expansion and nested macro calls with the help of an example.

Q. 2 What are the advantages of using macro in place of subroutine/function and when will be a macro used.

Q. 3 Explain the difference between functions and macros.

Q. 4 Describe the different features offered by the macro facility. Give example.

Q. 5 Explain the function of the macro processor.

Q. 6 Explain conditional macro expansion with suitable example.

Q. 7 Explain macro and databases used for one pass macro.

Q. 8 Explain the design of one pass macro processor.

Chapter Ends...

CHAPTER 4

Syllabus Topics

Introduction, functions
Relocating loader, Direc

4.1 Introduction

4.1.1 Evolution of

- Initially the program
  - Programmers v into the machir
  - The symbols w
  - The names are
- Assemblers solved of symbolic names
  - The work of a
- Loaders and Link memory.
- Relocation and Li
  - Mauchly (19 the catalog code to refle
  - These two assemblers.
  - The reloca addresses a

Scanned by CamScanner

Module 4

CHAPTER 4

Loaders and Linkers

Syllabus Topics

Introduction, functions of loaders, Relocation and Linking concept, Different loading schemes : Relocating loader, Direct Linking Loader, Dynamic linking and loading.

4.1 Introduction

4.1.1 Evolution of Linker and Loader

- Initially the programs in the computers were programmed using the machine language.
  - Programmers would write out the symbolic programs on sheets of paper, hands assemble them into the machine code, and then toggle the machine code into the computer.
  - The symbols were bound to addresses as the programmer did the translation from his hands.
  - The names are bound to addresses too early in the process.
- Assemblers solved the mentioned problem by allowing the programmers to write programs in terms of symbolic names and binding the names to the machine addresses.
  - The work of assigning the addresses is pushed off from the programmer to the computer.
- Loaders and Linkers has solved the mentioned problem by loading and linking the program in the memory.
- Relocation and Library Search
  - Mauchly (1947) has written about loading the programs along with subprograms selected from the catalog of programs stored on the magnetic tapes and, try to relocate the subprograms' code to reflect the addresses at which they were loaded.
  - These two basic linker functions-relocation and library search-appear to predate even assemblers.
  - The relocating loader will load the program anywhere in memory, altering the various addresses as required to ensure correct referencing

Scanned by CamScanner

- Before this there was operating systems, each program had the machine's entire memory at its disposal, so the program could be assembled and linked for fixed memory addresses.
- With the advancement in operating system, relocating loaders became necessary. The actual addresses at which the programs would be running were not known until the operating system loads the program into the memory.
- Linkers does the address binding, assigning relative addresses within each program, while the loader assigns the actual addresses.
- Linker also take care of complex name management and address binding. Linkers had to deal with object code generated by High-level programming languages such as FORTRAN, PASCHAL, and COBOL etc.
- When program is larger than **available memory, then linker provides Overlay structure.** It is a technique that let programmers share the same memory for different parts of a program to, with each overlay located on demand when another part of the program called into it.
  - They faded again as virtual memory appeared on PCs in the 1990s.
  - Overlays are still used in memory-limited embedded environments.
- With the advancement in hardware relocation and virtual memory, linkers and loaders actually became less complex, because each program could again have an entire address space.
- Programs loaded, could be linked at a fixed addresses with the hardware rather than software relocation taking care of any load-time relocation.
- As multiple copies of the same program were frequently run at the same time, compliers and assemblers were modified to create object code in multiple sections with one section for ready-only code and another section for writable data.
- The code is same among all running copies while other parts are unique to each copy.
- The linker has to combine all the sections of each type so that the linked program would have all the code in one place and all the data in another.

Control Sections

- When different programs are running on a computer, these different programs usually turn out to share a lot of common code. Like every C program uses routines such as fopen, and printf.

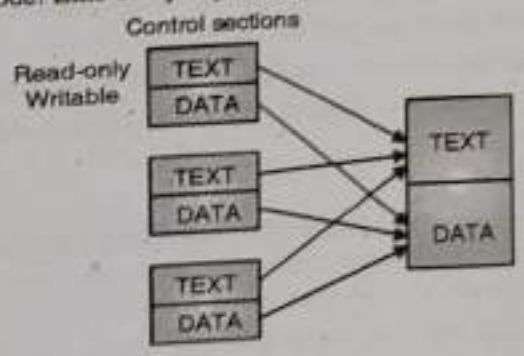


Fig. 4.1.1 : Object Code for Multiple Sections

- Programs running under a GUI such as X Windows, or MS Windows all uses a piece of the GUI library.

Scanned by CamScanner

- Most systems now provide shared libraries for programs to use, so all the programs that use a library can share a single copy of it.
  - **Static Shared Library :** The linker binds program references to library routines to those specific addresses at link time.
  - **Dynamic Linked Library :** Library sections and symbols are not bound to actual addresses until the program that uses the library starts running

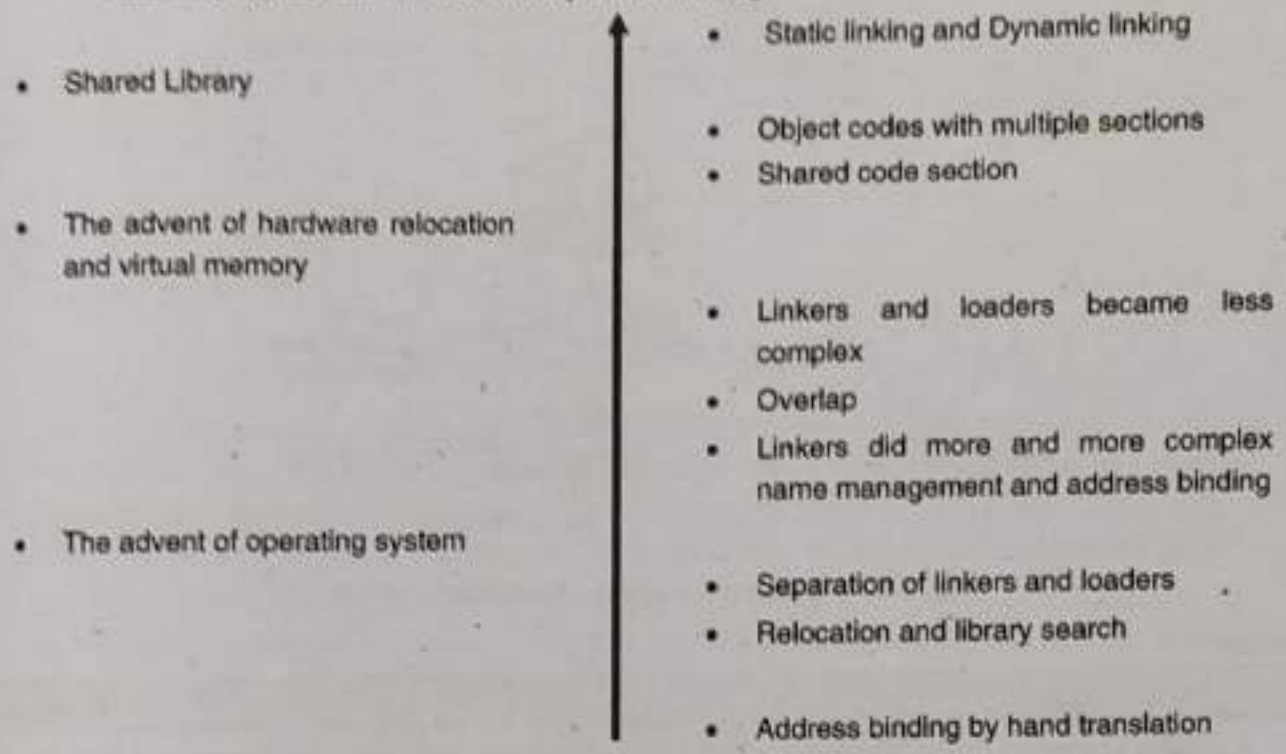


Fig. 4.1.2 : Basic Structure for Storing of Program into Memory

4.2 Linking vs. Loading

Q. Loading and linking scheme. (5 Marks)

Execution of program written in a language L performs the following steps

1. Program translation.
2. Linking of the program with other programs needed for the execution.
3. Relocation of the program to execute from the specific memory area allocated to it.
4. Loading of the program in the main memory for the execution.

For performing these steps we need different language processors. Step 1 is performed using **Language Translators**, step 2 and 3 is done with the help of **Linker** and finally step 4 is done with the help of **Loaders**.

Scanned by CamScanner

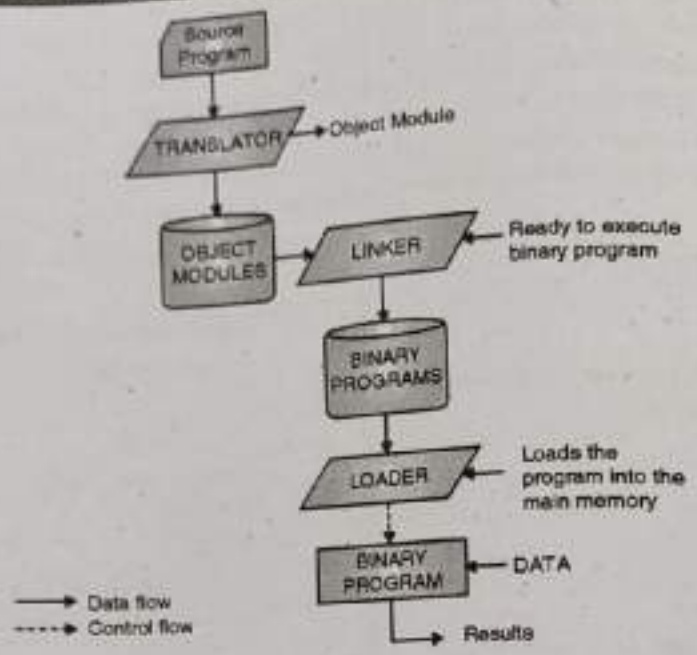


Fig. 4.2.1 : General Language Processor Scheme

Loader

Q. Define Loader. **(2 Marks)**

A **loader** is a program which accepts the various object program decks, and prepares these decks for the execution by loading them into the main memory for execution.

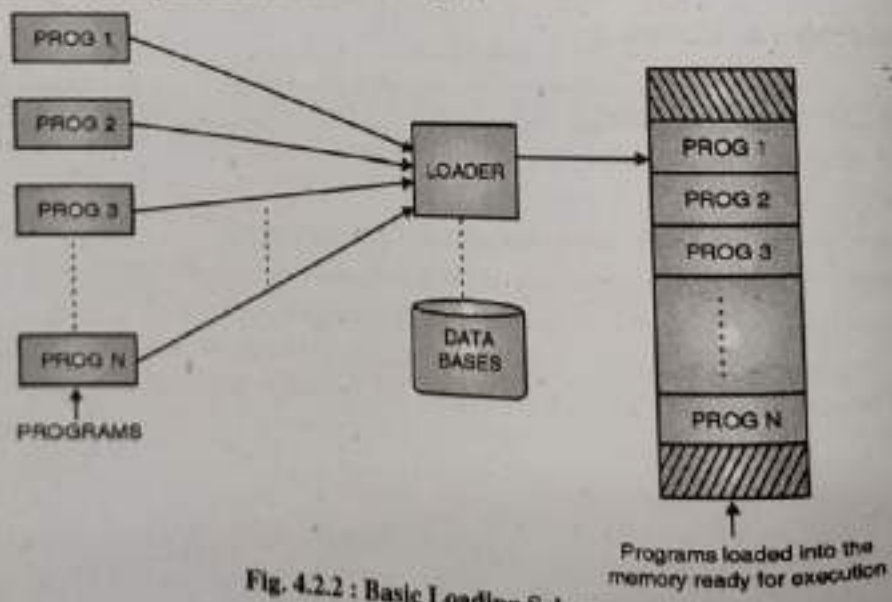


Fig. 4.2.2 : Basic Loading Scheme

Scanned by CamScanner

Linker

A "**Linker**" or **link editor** is a program that takes a collection of object modules (created by compilers or assemblers) and combines them into a single executable program. A loader places the linked program in main memory.

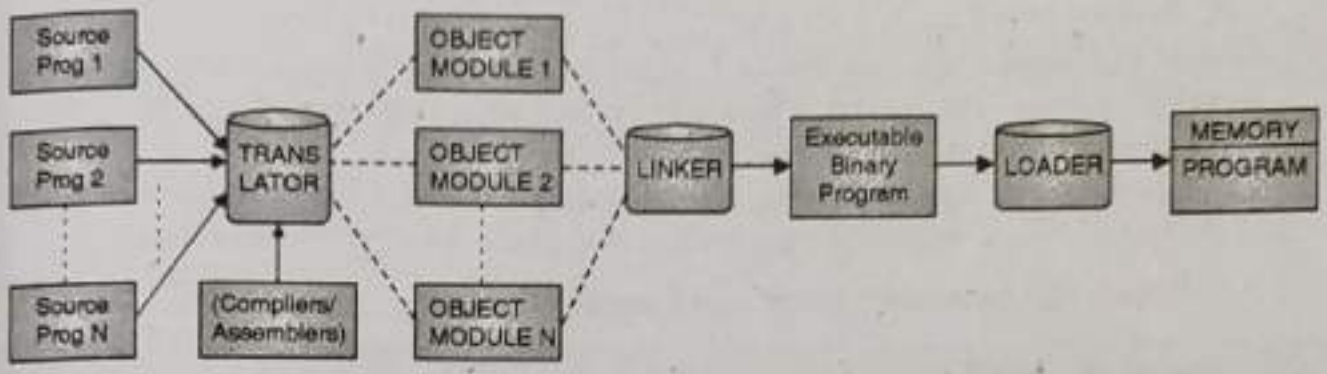


Fig. 4.2.3 : Basic scheme for LINKING of Object Modules

Linking is the process of combining various pieces of code and data together to form a single executable that can be loaded in the memory. Linker is also known as link editor or binder; many programming languages allow us to write different pieces of code, called modules, separately. This simplifies the programming task because we can break a large program into small, more manageable pieces and at end linker put all the modules together.

In addition to combining modules, a linker also replaces symbolic addresses with real addresses. Therefore, we may need to link a program even if it contains only one module.

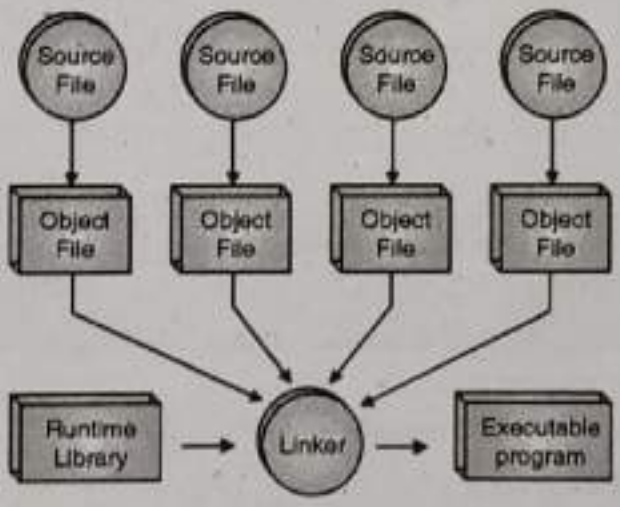


Fig 4.2.4 : Basic Structure of Linker

Linkers and Loaders perform several related but conceptually separate tasks :

- **Program loading :** In this a program is copied from the secondary storage into a main memory for execution. Loading of data also involves
  - Allocation of storage for data,
  - setting protection bits, and
  - Organizing data for virtual memory to map virtual addresses to disk pages.

Scanned by CamScanner

- **Relocation :** During relocation a load address is assigned to the different parts of the program by adjusting the code and data in the program to show the assigned addresses. When a program consists of a several subprograms made from various subprograms, all the subprograms have to be put at non-overlapping addresses so that they do not conflict with each other. Linker links all these subprograms and creates one linked output program that starts from zero within this the various subprograms relocated.
- **Symbol Resolution :** The main task of a symbol resolution is to translate named symbols into memory locations. The process typically involves a few stages: identifying the symbols, segmenting them, and determining the actual address.

Although there is considerable overlap between linking and loading, one important feature that linkers and loaders share is that they both patch object code. This is a uniquely powerful feature.

4.2.1 Difference between Linker and Loader

Q. Explain difference between : Linker and Loaders. (10 Marks)

Linkers	Loaders
A Linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.	Loader is a program that loads machine codes of a program into the system memory.
Linker is a part of a library files	Loader is a part of an operating system.
Linker performs the linking operations.	Loader loads the program for the execution.
It also links the user defined functions to the user defined libraries.	Loading a program involves reading the contents of executable file into memory

4.3 Functions of Loader

→ **(MU - Dec. 2014, May 2015)**

Q. What are the different functions of loader explain in brief? Dec. 2014, 5 Marks
Q. Explain Functions of loader. May 2015, 5 Marks
Q. Explain relocation. (2 Marks)

- The fundamental task of a loader is to
  - bringing an object program into memory and
  - starting its execution
- An object program contains translated instructions and data from the source program. It also specifies addresses in memory where these items are to be loaded.

To execute a program a loader performs the following four functions

1. **Allocation :** It is used to allocate space in memory for the object programs. Translators cannot allocate space since overlap may occur or large wastage of memory takes place.
2. **Linking :** It combines two or more separate object programs and resolve symbolic references between object decks. It also supplies the information needed to allow to reference between them.

Scanned by CamScanner

3. **Relocation :** It modifies the object program so that it can be loaded at an address different from the location originally specified and adjusts all address dependent location.
4. **Loading :** Physically it places the machine instructions and data into the memory for the execution.

4.4 Relocation and Linking Concept

- Computer programs comprises of several fragments or modules all these fragments are need not be controlled or contained within a single object file. In such cases these modules are referred to each other by means of symbols. Typically, an object file has three kinds of symbols :

1. Publicly defined symbols
2. Externally defined symbols
3. Local symbols

1. **Publicly defined symbols :** These symbols are allowed to be called by other modules, also called as **public definition**.
2. **Externally defined symbols :** These are also known as **undefined symbols** or **external references.** They call the other modules where these symbols are defined.
3. **Local symbols :** It is used internally within the object file to facilitate relocation.

- In addition to combining modules, a linker also replaces symbolic addresses with real addresses. Therefore, we may need to link a program even if it contains only one module.

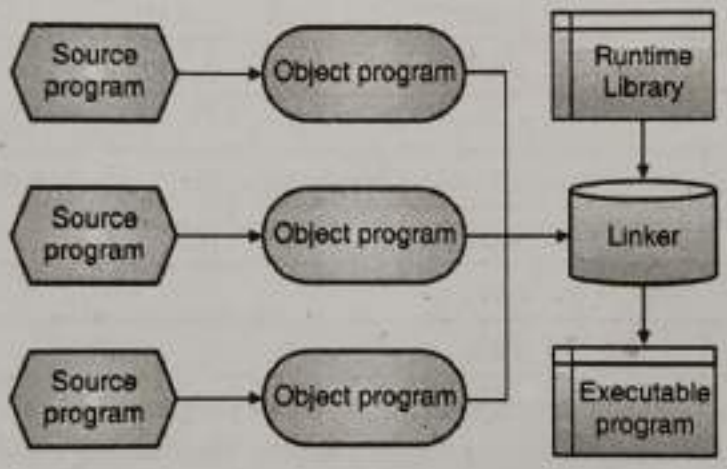


Fig. 4.4.1 : Relocation of linking concept

- Linkers do not include the full library in the output. It only includes the objects from a group of files called a library. It only includes its symbols that are referenced from other object files and arranges the objects in a program's address space. It has a feature to involve the relocating code that assumes a specific base address to another base.
- As compiler is not aware where an object will reside, it often assumes a fixed base location. Relocating machine code may involve re-targeting of absolute jumps, loads and stores. The executable code by the linker may need another relocation pass when it is finally loaded into memory.

Scanned by CamScanner

Linking is of two types :

1. Static linking	2. Dynamic linking

1. **Static linking :** It copies all library routines used in the program into the executable image. It requires more disk space and memory.
2. **Dynamic linking :** It resolves the undefined symbols while program is running. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for the same.

Relocation and linking requirements in segment address

Segment addressing helps to reduce the relocation requirement of a program.

Sr. No.		Statement		Offset
0001	DATA 1	SEGMENT		0000
0002	ABC	DW 25		0002
0003	B	DW ?		
0012	SAMPLE	SEGMENT		
0013		ASSUME	CS : SAMPLE DS : DAMI	
0014		MOV	AX, DAMI	0000
0015		MOV	DS, AX	0003
0016		JMP A		0005
0017	A	MOV	AL, B	0008
0027	S	MOV	AX, BX	0196
0043	SAMPLE	END S		
0044		END		

In this program, the ASSUME statement is used to declare segment register CS and DS for memory addressing. Reference to A is assembled as a displacement of 196 from the content of the CS register in statement 16. This avoids the use of an absolute address.

Linking requirement

- Linking is required by all subprogram calls.
- Programs written in PASCAL are typically nested inside the main program, hence procedure references do not require linking.

Object module

Object module is a sequence of object records, each object record describing specific aspects of the program.

Scanned by CamScanner

The object module contains the following basic information.

(1) Binary image

(2) External references

(3) Public definitions

(4) Debugging information

(5) Miscellaneous information

Each object record contains variable length information.

4.5 Different Types of Loaders

Q. Explain in brief various types of loader with their advantages and disadvantages. (10 Marks)

The various type of loaders are as follows

1. Compile and Go Assembler / Compile and Go Loader
2. Absolute Loader
3. Relocating Loader
4. Direct Linking Loader
5. Dynamic Loading Loader
6. Dynamic Linking Loader

These loaders primarily differ in the manner in which the four basic functions are accomplished.

4.5.1 Compile and Go Loader

A compile and go loader is one in which the assembler itself does the processes of compiling then place the assembled instruction in the designated memory locations. The assembly process is first executed and then the assembler causes a transfer to the first instruction of the program.

Advantages

- Simple to implement.
- No extra procedures are involved.

Disadvantages

- Wastage of memory.
- It retranslates the user's program code every time it is executed.
- When the source code is in different program is in different languages it is hard to handle multiple segments.

Scanned by CamScanner

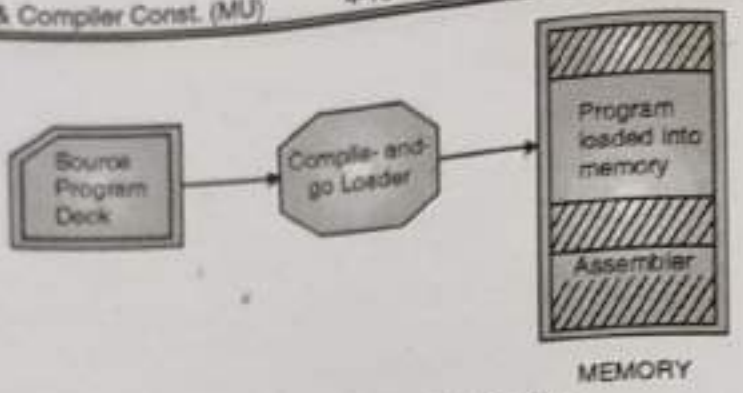


Fig. 4.5.1 : Assemble and Go Loader

4.5.1(A) General Loading Scheme

Loader is a utility program that accepts the object code and load them into the main memory for the execution. Loader involves the storage management function of operating systems and is usually performed later than compilation. In loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instructions and data in an executable form at their assigned memory. The loader occupies some portion of main memory. Load time refers to the period of loading the program in memory and preparing object program for execution.

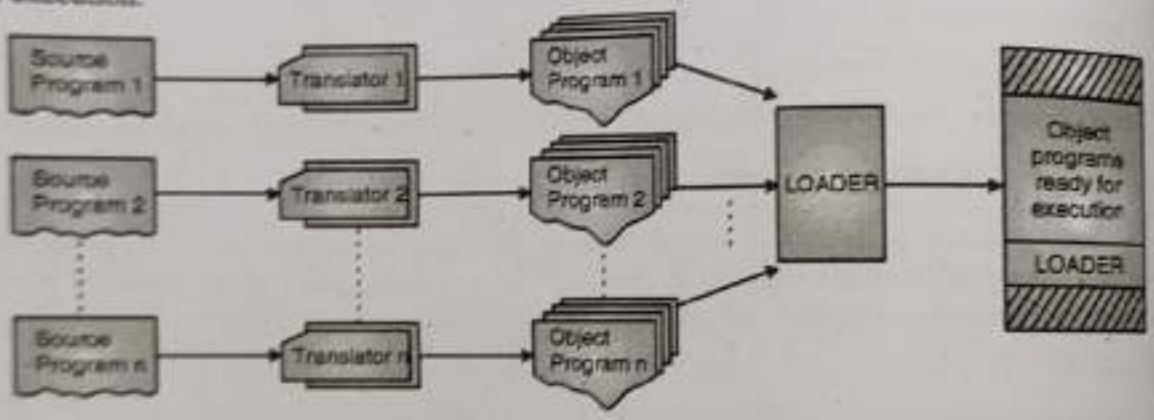


Fig. 4.5.2 : General Loading Scheme

All disadvantages of translate and go loader are overcome using this scheme since

- Loader is smaller than the translator hence less wastage of memory.
- Retranslation is not necessary every time the program is executed since object decks are used.
- As the loader would be in the same machine language it is possible to write subroutines in different languages for the object decks.
- Loading involves binding instructions and data to the physical memory addresses. Three approaches are used for loading.
  - **Absolute Loading :** Binding can be done beforehand, at compile time (writing once)
  - **Relocatable Loading :** Binding is done at load time (Rewriting)
  - **Dynamic Runtime Loading :** Binding is postponed till execution time (Not writing).

Scanned by CamScanner

4.5.2 Absolute Loader

→ (MU - May 2014)

Q. Explain the design of absolute loaders and mention all data structures in detail. **Dec. 2014, 10 Marks**

The type of loader scheme which is used for Fig. 4.5.3 is called an **Absolute Loader**. The loader simply accepts machine language code and places it into main memory specified by the assembler. It is similar to the **"Compile And Go"** loaders. The loader accepts the machine language instructions and places it into the core, then initiates execution by transferring control to the starting of the program in this the data is store on cards instead of being placed in main memory for execution. The object code is loaded at the specified locations in the memory. At the end the loader jumps to the specified location to begin the execution of the loaded program. In absolute loader no linking or relocation is done. Absolute loader requires single pass operation, in that it checks.

- **Single pass operation of Absolute Loader :**
  - Check **H (Header Record) record to verify that correct program** has been presented for loading.
  - Read each **T (Text Record) record, and move object code into** the indicated address in memory.
  - At **E (End Record) record, jump to the specified address to** begin execution of the loaded program.

Algorithm for Absolute Loader

```
Start
    read Header record
    Verify program length and name
    read first text record
    while record type!=E do
            begin
                if object code is in character form converts it into internal representation
                 move object code to specified location in memory
                 read next object program record
            end
        Jump to address specified in End record
Stop
```

Advantages

Simple and efficient to implement.

Disadvantages

- The programmer must specify to the assembler the address in the core where the program is to be loaded.
- In case of multiple subroutines the programmer must remember the address of each and use that absolute address explicitly in other subroutines to perform a subroutine linkage.

Scanned by CamScanner

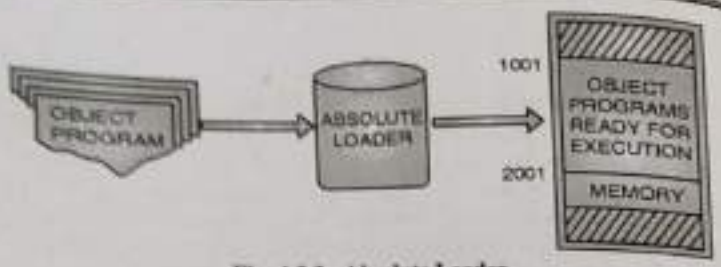


Fig. 4.5.3 : Absolute Loader

The four basic functions of an absolute loader are as :

i. **Allocation** – done by programmer
ii. **Linking** – done by programmer
iii. **Relocation** – loaded where assembler assigns.
iv. **Loading** – By loader

4.5.2(A) Bootstrap Loader

Bootstrap Loader is a type of an absolute loader. When the computer is first powered on, it does not have an operating system in Memory. The computer alone cannot perform complex actions such as loading of a program from the disk. The solution for this is to use a special type of program called **Bootstrap Loader**. The program's only job is to load software for the operating system to start. The name bootstrap loader comes from the image of pulling one oneself up by ones bootstraps.

A bootstrap loader is a program that resides in the computers EPROM, ROM, or other non-volatile memory that automatically executed by the processor when turning on the computer. The booting loader reads the hard drives boot sector to continue the process of loading the computers operating system. The term boostrap comes from the old phrase "Pull yourself up by your bootstraps." It is also known as **bootstrapping** or **boot loader**. The boot loader is now part of the EFI BIOS.

The basic functions of the bootstrap loader are as following :

- It helps user to select an operating system for starting.
- It loads the operating system files from the boot partition.
- It controls the operating system selection and hardware detection prior to the kernel initialization.

In computing, **bootstrapping** is a technique by which a simple computer program activates a more complicated system of programs. In the startup process of a computer system, a small program such as BIOS, initializes and tests that hardware, peripherals and external memory devices are connected, then loads a program from one of them and passes control to it, thus allowing the loading of larger programs such as an operating system.

Bootstrapping

- Actions taken when a computer is first powered on.
- The hardware logic reads a program from address 0 from ROM (Read Only Memory)
  - ROM is installed by the manufacturer
  - ROM contains bootstrapping program and some other routines that controls hardware (e.g. BIOS)

Scanned by CamScanner

Bootstrapping in a loader

- Loads OS from disk into memory and makes it run.
- The location of OS on disk (or floppy) usually starts at the first sector
- Starting address in memory is usually fixed to 0
  - No need to relocation
- This kind of loader is simple
  - No relocation
  - No linking
  - Called "absolute loader"

4.5.3 Relocating Loader

The task of relocating loader is to avoid reassembling of all subroutines when a subroutine is changed and to execute tasks of allocation and linking for programmer. Relocating loaders adjust addresses of the executable code to compensate for variations in the address at which loading starts. The computers which need relocating loaders are those in which pointers are absolute addresses rather than offset from the programs base address. **Binary Symbolic Subroutine (BSS)** loader is an example of Relocating Loader. BSS performs all the four functions of the loader.

- **Allocation** : It is done with the help of the program length and the information.
- **Linking** : It is done by transfer vector.
- **Relocation** : With the help of the relocation bits.
- **Loading** : It is done by the loader.

For each Source program the assembler outputs

- The machine translation code of the program.
- Transfer vector are used to store the addresses containing names of the subroutines referenced by the source program.
- Relocation bits are associated with each instruction or address field which includes :
  - Length of the entire program.
  - Length of the transfer vector portion.
  - Operation performed by the Relocating Loaders.
- Assembler assembles each segment independently and passes on to the loader, the text and the information for the relocation and the intersegment references.
- The BSS loader allows many procedure segments and only single data segment.
- The loader uses the transfer vector to solve the problem of the subroutine linkage
- The length of the whole program and the length of the transfer vector are given by the assembler.
- Relocating loader will load all the information regarding subroutines and would place the equivalent addresses of the subroutines to each position of the transfer vector. It will also load the transfer vector into the core.
- The output of relocation assembler using a BSS scheme is the object program and the information about all other programs it referenced.
- If the relocation bit is one, then the corresponding address bit must be relocated otherwise the field is not relocated.

Scanned by CamScanner

Disadvantages

- The transfer vector is used only for transfers and not used for loading or storing of external data.
- The size of the object program in the memory is increased due to transfer vector.
- The BSS loader processes procedure segments but does not facilitate access to the data segments that can be shared.

4.5.4 Direct Linking Loader → (MU - Dec. 2014, Dec. 2017)

Q. Explain working of a direct linking loader with a proper example. Clearly show the entries in different databases built by the direct linking loader. **Dec. 2014, 10 Marks**

Q. Explain working of direct linking loader with example showing entries in different databases built by DLL. **Dec. 2017, 10 Marks**

Q. Explain the working of Direct Linking Loader with neat flow charts. **(10 Marks)**

A Direct Linking loader is the most popular loading scheme used. It allows the programmer to use the multiple procedure and data segments giving user a complete choice in referencing data or instruction enclosed in other parts of segments. It provides flexible intersegment referencing accessing ability. Assembler provides following information to loader with respect to each procedure or data segment :

- Segment length.
- Records of all the symbols that may be referenced by other symbols and their relative location.
- Records of all the symbols referenced in the segment but not defined in the segment are also maintained.
- Explanation about the address constants, their location in the segment and information about their revise values.
- Information about the source codes machine code translation and the relative addresses assigned.

4.5.4(A) Design of Direct Linking Loader

Q. Explain the design of direct linking loader in detail. **(10 Marks)**

Direct linking loader uses four types of records in the object file they are as ESD, TXT, RLD, END.

1. External Symbol Directory (ESD)

It combines information about all symbols that are mentioned in the program but that may be referenced elsewhere.

2. Test Record (TXT)

It contains the information about the actual object code translated version of the source program.

3. Relocation and Linkage Directory (RLD)

RLD cards are used to store those locations and addresses on which the program content is dependent.

Scanned by CamScanner

4. END Record

It specifies the end of the object file and starting address for execution if the assembled routine is in the main program.

4.5.4(B) Example Program for Direct Linking Loader

Source	Relative		Sample Program
1	0	LOOP1	START
2			ENTRY L0START0, L0START1
3			EXTRN L1START0, L1START1
4	8	L0START0	
5	12	L0START1	
6	16		DC B (L0START0)
7	20		DC B (L0START1 + 10)
8	24		DC B (L0START1 – L0START0 – 2)
9	28		DC B (L1START1)
10	32		DC B(L1START0 + L1START1 + 5)
11			END
12	0	L1START1	START
13			ENTRY L1START0
14			EXTRN L0START0, L0START1
15	4	L1START0	
16	14	DC	A (L0START0)
17	18	DC	A (L0START1 + 10)
18	22	DC	A (L0START1 – L0START0 – 5)
19	END		

SAMPLE PROGRAM FOR LOOP1 and L1START1

Object deck program for LOOP1

ESD Records

Source Card Reference	Variable Name	ESD		Relative address	Length
		Type	ID		
1	LOOP1	SD	01	0	36 (STARTING FROM 0 AND ENDING AT 36)
2	L0START0	LD	02	08	-

Scanned by CamScanner

Source Card Reference	Variable Name	ESD		Relative address	Length
		Type	ID		
2	L0START1	LD	02	12	-
3	L1START1	ER	03	-	-
3	L1START0	ER	03	-	-

Where,

SD – Segment Definition (Hexadecimal code = 01)

LD – Local Definition (Hexadecimal code = 02)

ER – External Reference (Hexadecimal code = 03)

TXT Records

Source Card Reference	Relative Address	Content	What the assembler did ?
6	16-19	08	
7	20-23	22	12 + 10 = 22
8	24-27	02	12 – 08 – 02 = 02
9	28-31	' 0	Not known to LOOP1
10	32-35	+ 5	(5) L1START0, L1START1 NOT KNOWN TO ASSEMBLER.

RLD Records

Variable Name	Source Card Reference	ESD ID	Length (by k)	Arithmetic Operator	Relative Address
L0START0	6	02	4	+	16
L0START1	7	02	4	+	20
L1START1	9	03	4	+	24
L1START0	10	03	4	+	28
L1START1	10	03	4	+	28

Object Deck Program for L1START1

ESD Records

Source Card Reference	Variable Name	Type	ESD ID	Relative Address	Length
12	L1START1	SD	01	0	26(STARTING FROM 0 AND ENDING AT 26)
13	L1START0	LD	02	04	N/A
14	L0START0	ER	03	Unknown	N/A
14	L0START1	ER	03	Unknown	N/A

Scanned by CamScanner

TXT Records

Source Card Reference	Relative Address	Content	What the assembler did?
16	14	0	0 L1START0 Not known
17	18	10	10 L0START1 Not known
18	22	– 5	– 5 L0START0, L0START1 Not known

RLD Records

Variable Name	Source Card Reference	ESD ID	Length	Arithmetic Operator	Relative Address
L0START0	16	03	4	+	14
L0START1	17	03	4	+	18
L0START1	18	03	4	+	22
L0START0	18	03	4	–	22

Global External Symbol Table (GEST)

GEST is used to store the external symbols and their corresponding core address in table which are defined by the SD or LD entry on an ESD card.

12 bytes per entry

External symbol	Assigned Core Address
"LOOP1bb"	100
"L0START0bb"	120
"L0START1bb"	130
"L1START1bb"	150
"L1START0b"	180

Scanned by CamScanner

Local External Symbol Table(LESA)

The Relation between the core address and the ID number on the RLD card is given by the LESA.

		Assigned core address of corresponding symbol (4 –bytes) decimal
Indexed by identifier	1	100
	2	120
	3	130
	4	–
	:	
	:	:
	255	:

4.5.4(C) Databases Required for Direct Linking Loader

(I) PASS 1

1) Object files used for input.
2) Initial Program Load Address (IPLA) gives the address to load the first segment given by operating system or programmer.
3) Program Load Address (PLA) it keeps the track of each location for the segments assigned to it.
4) To store external symbol and its corresponding, assigned core address GEST is used.
5) Output of PASS1 is used as input for PASS2.
6) Load map is used to specify external symbols and their assigned output values.

(II) PASS 2

1) Output of PASS1 is used as input for PASS2.
2) The **Initial Program Load Address Parameter (IPLA) from PASS1**
3) Program load Address Counter prepared by PASS1.
4) GEST prepared by PASS1.
5) The correspondence between the external symbols core address and name mentioned on ESD and RLD cards is given by **Local External Symbol** array.

Scanned by CamScanner

4.5.4(D) Algorithm and Flowchart for Direct Linking Loader

PASS 1 : Gives the definition of symbols and allocate segments

- It gives the value of all the external symbols and their assigned location for every segment.
- In the beginning of PASS1 PLA is the IPLA. An object code is read and used for PASS 2.
- The record may be one of the five types ESD, TXT, RLD, END or LDT/EOF.
- As soon as TXT or RLD record is encountered no processing is done during PASS 1 so the next record is read.
- ESD record is processed depending upon SD, LD or ER.
- **SD (Segment Definition) :** During SD length is temporarily stored in SLENGTH and the assigned value to this symbol is set to the current value of the PLA.
- After that value is stored in GEST to check whether the symbol exist or not.
- If the symbol exists in the GEST error duplicate entry is given otherwise symbol and its value are printed as part of the load map.
- **LD (Local Definition) : During** LD the value assigned PLA plus the relative ADDR given by the ESD record.
- **ER (External reference) :** No processing is done for ER.
- **END :** During END record the PLA is incremented by adding the SLENGTH, in it and giving the PLA for the next segment.
- **LDT/EOF :** When LDT or EOF ESD card is read, control transfers to the PASS 2 after completing PASS1.

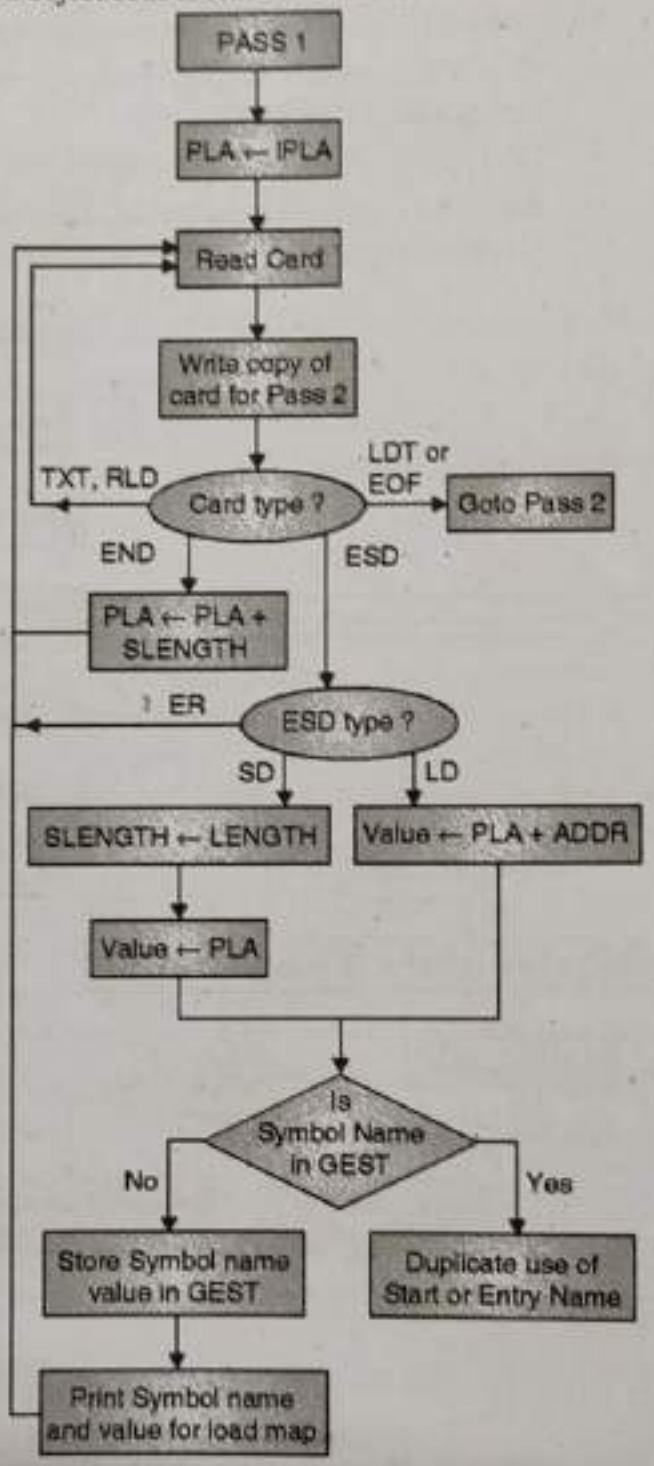


Fig. 4.5.4 : Flowchart for PASS 1

Scanned by CamScanner

PASS 2 : Load the Actual Program Text.

In PASS 2 the control is transferred to the loaded program by the loader after all the segments have been assigned locations and the external symbols have been defined by PASS 1.

1) At starting of PASS2 PLA is the IPLA same as PASS 1 and the execution start address (EXADDR) is set to IPLA.
2) Each of the ESD records is processed in different manner.
   - **SD :** In this length is temporarily saved to the variable SLENGTH and LESA(ID) is set to the current value of the PLA.
   - **LD :** No processing is done during PASS 2.
   - **ER :** GEST obtained form PASS1 is searched for match with an ER symbol if found its value is extracted LESA (ID) is set to one otherwise error message is displayed .

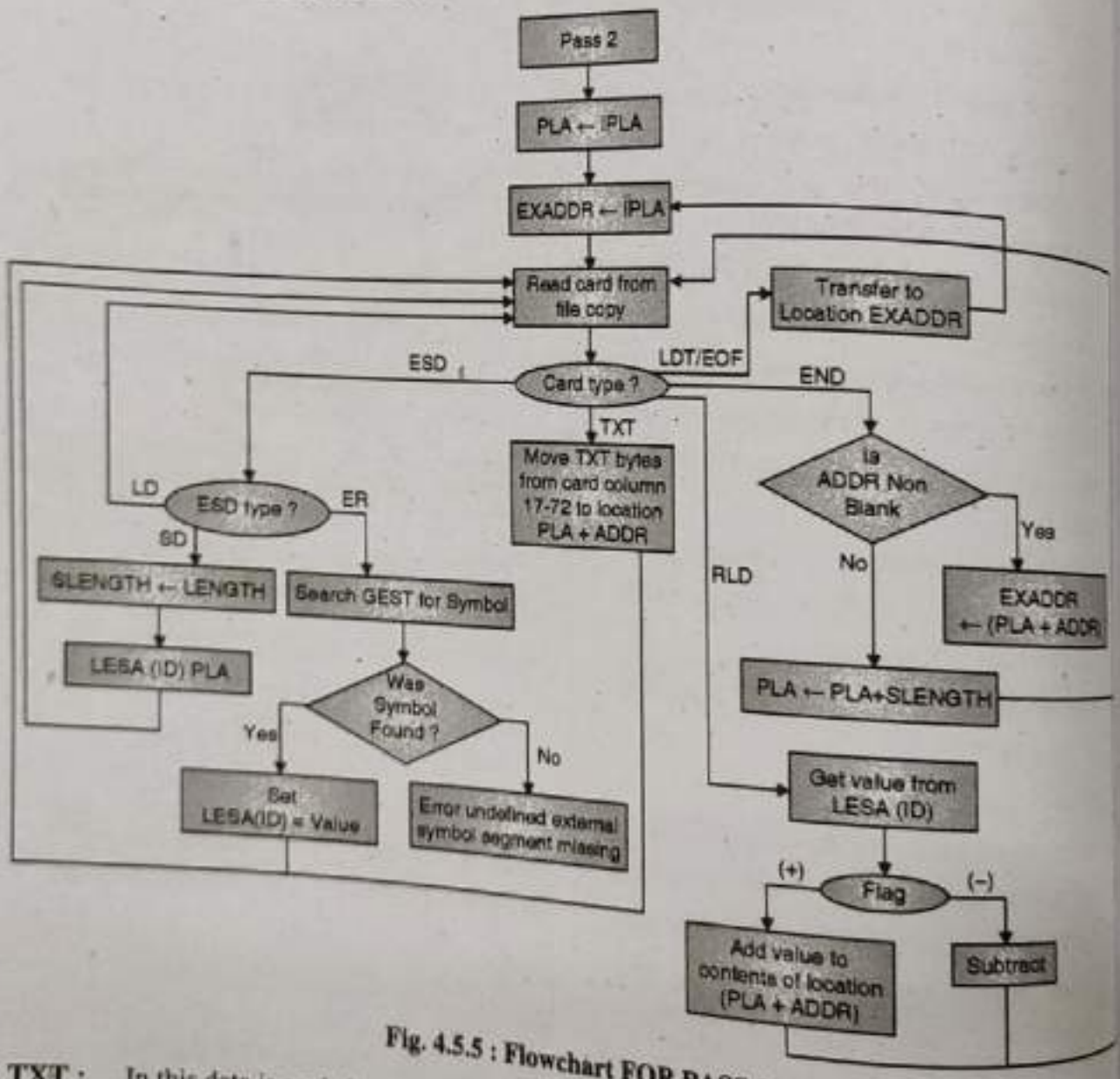


Fig. 4.5.5 : Flowchart FOR PASS 2

TXT : In this data is copied to the appropriate relocated core location (PLA+ADDR).

Scanned by CamScanner

RLD : RLD record is used to calculate the actual relocated address as the sum of the PLA and the ADDR field.

END : In END record the PLA is incremented by adding the SLENGTH, in it and giving the PLA for the next segment. It is used to store execution start address EXADDR after being relocated by the PLA.

LDT/EOF: **As soon as LDT/EOF is encountered the** control is transferred to the current address of the loaded program given by EXADDR.

Disadvantages

- In direct-linking loader, it is necessary to allocate, relocate, link and load all of the subroutines each time in order to execute the program. So loading process can be extremely time consuming.
- Even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space.

The mentioned problem can be solved by dividing the loading process into two separate programs :

1. **Binder : A *binder*** is a program that performs the same function as a direct-linking loader in "binding" subroutines together. The binder performs ***allocation, relocation*** and ***linking***.
2. **A Module Loader** : It outputs the text in a file rather than memory. The module loader merely has to physically load the module into the memory.

4.5.5 Dynamic Loading Loader

If all the subroutines are loaded simultaneously into the core, then there may be a chance, that the core available may be insufficient for the subroutines, this may lead to further complications Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such situations only those segments are resident in the memories that are actually needed at the time of execution. But the question arises what will happen if the required segment is not present in the memory? Naturally the execution process will be delayed until the required segment gets loaded in the memory.

Overlay structure is used to specify the inter dependency between all the segments. It consists of nodes and edges and segment is represented by the nodes. If the two are on the same path they can lie in the main memory. For solving such problems techniques like segmentation and paging is used. In these the subroutines are loaded into core at a different time because the subroutines in a program are needed at different times i.e. they may be mutually exclusive, by identifying which subroutine can calls other subroutines to produce an Overlay Structure.

Overlay Structure is used to :

- Keep in memory only those instructions and data that are needed at any given time.
- Overlay structure needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

The flipper or overlay supervisor is the portion of the loader that actually intercepts the "calls" and loads the necessary procedure.

Let us take an example to understand the concept. Various segments along with their memory requirements are shown further. The Fig. 4.5.6 illustrates a program consisting of six subprograms. (PROG1, PROG2, PROG3, PROG4, PROG5, PROG6)

Scanned by CamScanner

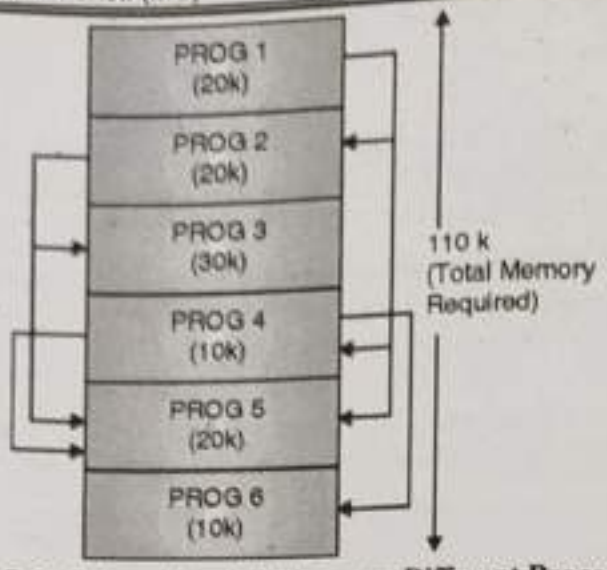


Fig. 4.5.6 : Subroutine Calls between Different Procedures

Subroutine call between the procedures

```
PROG 1
{
    ==  PROG 2
    {
    ==  PROG 3
        {
        .
        .
        .
        }
    PROG 5
    { }
    }
    PROG 4
    {
    ==  PROG6
        {
        —
        }
    }
    PROG 5
    {
    —
    }
}
```

Scanned by CamScanner

PROG 1 – Calls (PROG 2, PROG 4, PROG 5)

PROG 2 – Calls (PROG 3, PROG 5)

PROG 4 – Calls (PROG 6)

PROG 3 and PROG 5 do not call any other routines.

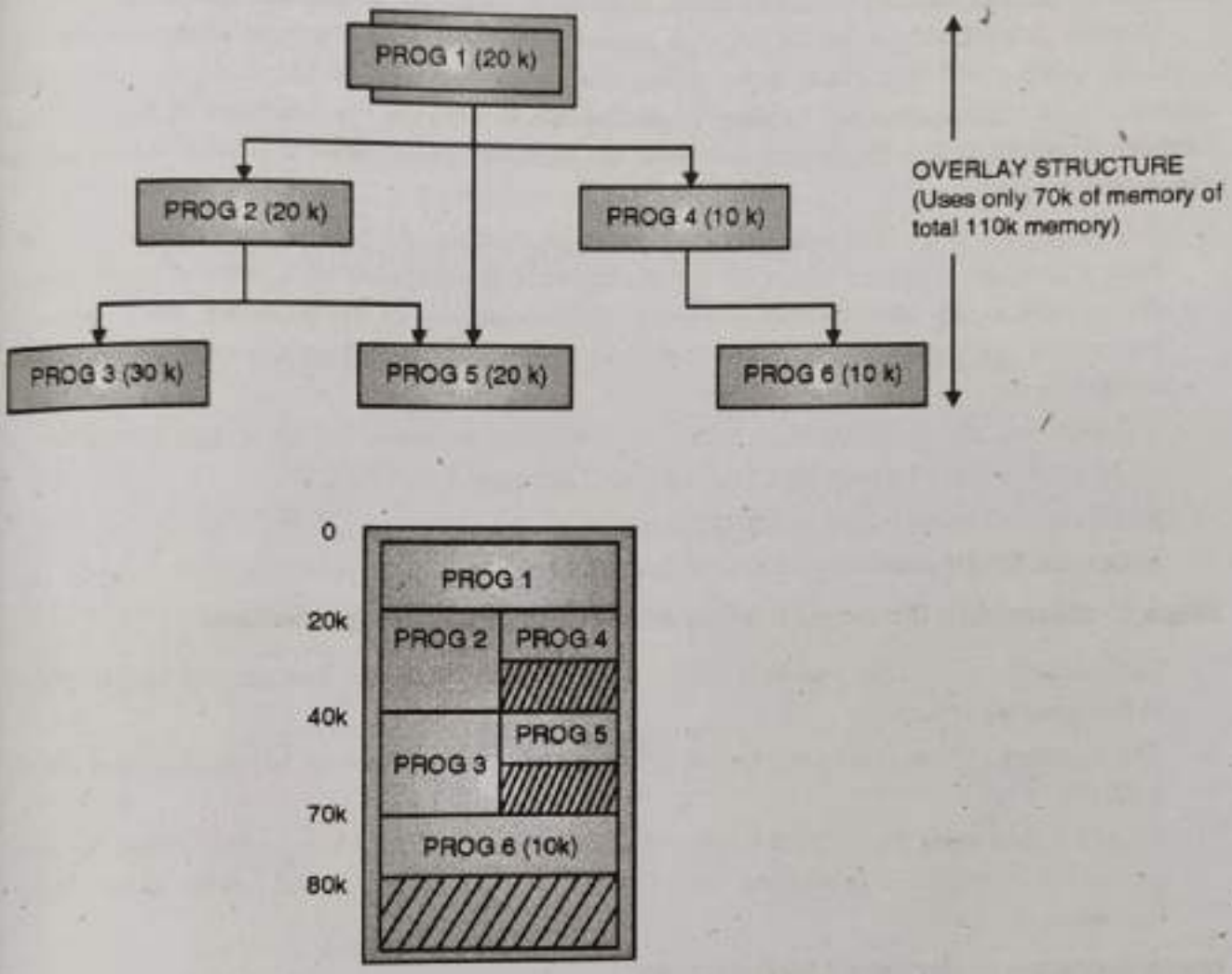


Fig. 4.5.7 : Storage Allocation of Each Procedure

Note : Procedures PROG 2 and PROG 4 are never in use at the same time; neither are C and E.

If we load only these procedures that are actually be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure or flipper. Here it happens to be 70K (Procedures PROG 1, PROG 2, and PROG 3).

Advantages

- No wastage of memory.
- A system can be dynamically loaded whenever it is required.

Scanned by CamScanner

4.5.6 Dynamic Linking Loader

→ (MU - Dec. 2015)

Q. Explain Design of Dynamic Linking Loader along with example. Dec. 2015, 10 Marks

Q. Explain dynamic linking loader. (5 Marks)

Dynamic linker Loader a special part of an operating system that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. This approach is also called **dynamic linking** or **late linking**. It retrieves the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory.

1) It is often used to allow several executing programs to share one copy of a subroutine or library. For e.g., run time support routines for C could be stored in Dynamic Link Library. A single copy of the routines in this library could be loaded into the memory of the computer. All C programs currently in execution could be linked to this one copy, instead of linking a separate copy into each object program.
2) It provides the ability to load the routines only when they are needed so lot of time and memory is saved if subroutines are large with lots of external references.
3) It helps in not loading the entire library for execution.
4) In dynamic linking loader is used to load the main program.

Steps to accomplish the actual loading and linking of a called procedures.

1) The symbolic name of the routine in the program is used to make the load and call service request to the operating system.
2) The operating system checks its internal tables to determine whether or not the routine is already loaded.
3) Control is then being passed from the operating system to the routine being called. When the called subroutine completes its processing, the operating system then returns the control of the program that issued this request.

Implementation of Dynamic Linking Loader

1. Dynamic-link library, or DLL, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. These libraries usually have thefile extension DLL, OCX (for libraries containing ActiveX controls), or DRV (for legacy system drivers).
2. In Unix operating systems, the dynamic linker is actually an external executable that the operating system kernel loads and executes first in a process address space newly constructed as a result of calling exec or posix_spawn functions. When an executable file is loaded, the operating system kernel reads the path of the dynamic linker from it and then attempts to load and execute this other executable binary.
3. In Apple Darwin operating system, OS X and iOS operating systems the dynamically loaded shared libraries can be identified either by the filename suffix .dylib or by their placement inside the bundle for a framework.
4. In Unix-like operating systems using XCOFF, such as AIX, dynamically-loaded shared libraries use the filename suffix .a.

Scanned by CamScanner

5. Dynamic linking from Assembler language programs in IBM OS/360 and its successors is done typically using a LINK macro instruction containing a Supervisor Call instruction that activates the operating system routines that makes the library module to be linked available to the program.

Advantage

1) No over head is incurred unless the procedure to be called or referenced is actually used.
2) A further advantage is that the system can be dynamically configured.

Disadvantage

1) Considerable overhead and complexity incurved, due to the fact that we have postponed most of
2) The binding process until execution time.

4.5.7 Difference between Dynamic Loading and Dynamic Linking

→ (MU - May 2016)

Q. What is the difference between Dynamic Loading and Dynamic Linking explain with an example? **May 2016, 10 Marks**

Sr. No	Dynamic Loading	Dynamic Linking
1.	It refers to mapping (or less often copying) an executable or library into a process's memory after is has started.	Dynamic linking refers to resolving symbols - associating their names with addresses or offsets - after compile time.
2.	It is similar to plugins that is an exe can actually execute before the dynamic loading happens.	The linker while creating the exe does minimal work. For the dynamic linker to work it actually has to load the libraries.
3.	Dynamic loading is concerned with loading the subroutines of a program as and when required during run-time, instead of loading all the subroutines at once before the program execution starts.	Dynamic linking is concerned with linking library routines at run-time instead of combining them with the executable program code before the program execution starts i.e. static linking. To achieve this, a small code call 'stub' is inserted in the program code wherever library routine is required. The stub contains information about where to find the routine if it is not already in the main memory.
4.	Efficient use of memory.	Efficient use of memory.
5.	It does not require special support from Operating system; it is the responsibility of the programmer to check whether the routine that is to be loaded does not exist in main memory.	Dynamic Linking requires special support from operating system, the routine loaded through dynamic linking can be shared across various processes.

Scanned by CamScanner

Review Questions

Q. 1 What is difference between loader and linker ? Give one example of each.

Q. 2 What is loader ? Explain the four basic functions of loader.

Q. 3 Explain the different types of loaders.

Q. 4 Differentiate absolute loader versus relocating loader starting who performs which function in the particular loading scheme.

Q. 5 Explain the relocation and linking concepts.

Q. 6 Describe the working of a Direct Linking Loader. Explain in detail the various data structures used.

Q. 7 What is Bootstrap Loader ? Explain.

Q. 8 Explain Dynamic Loading and Linking with the help of example.

Q. 9 At what point in time do each of the following loading schemes perform binding ?

a. BSS Loader

b. Direct Linking Loader

c. Absolute Loader

Q. 10 Explain the Compile and Go Loader scheme. Also give its advantages and disadvantages.

Q. 11 What is loader ? Explain the loader functionality and its components in detail.

Q. 12 Explain the design of direct linking loader

Q. 13 Write a short note on dynamic loading and dynamic linking.

Q. 14 Explain the design of Absolute Loader.

Q. 15 Explain the relocation and linking concepts.

Chapter Ends

□□□

Scanned by CamScanner

Module 5

CHAPTER

5 Compilers : Analysis Phase

Syllabus Topics

Introduction to compilers, Phases of compilers :

Lexical Analysis : Role of Finite State Automata in Lexical Analysis, Design of Lexical analyser, data structures used.

Syntax Analysis : Role of Context Free Grammar in Syntax analysis, Types of Parsers : Top down parser- LL(1), Bottom up parser- Operator precedence parser, SLR

Semantic Analysis, Syntax directed definitions.

5.1 Evolution of Compilers

Early computers did not use compilers. Compilers had not yet been invented because computers had a very little memory and programs. Programmer soon began to create programs to help with the tedious task of entering machine code, for this a program was developed, which would translate an alphabetic letter into a corresponding numeric instruction which could be executed by the machine. In this manner, assembly languages, the primitive compiler and the assembler emerged. The term compiler was coined in the late forties, by Grace Murray Hopper, a pioneer who raised the challenges of programming the first computers.

During 1960's with the evolution of programming languages and the increasing power of computers, compilers are becoming more and more complex to bridge the gap between different programming languages to increase the performance of machines.

A compiler is itself a complex computer program written in some implementation language. The first compiler was created by HART and LEVIN at MIT in 1962 for LISP. Since the 1970's it has become common practice to implement a compiler in the language it compiles.

A **compiler** is a program that accepts an input a program text in a certain language and produces a output a program text in another language, while preserving the meaning of the text. Almost all the compilers translate from one input language i.e. the **source language**, to one output language, i.e. the **target language**. The compiler is itself written in the language called the **implementation language**.

To have the executable code we run a compiler on input source program which generates the executable code. This code is again given to another compiler whose input data is a file in some another language and it generates the executable machine instructions.

Scanned by CamScanner

These instructions are given to the Linker and then to the loader for execution which generates the final output of the program.

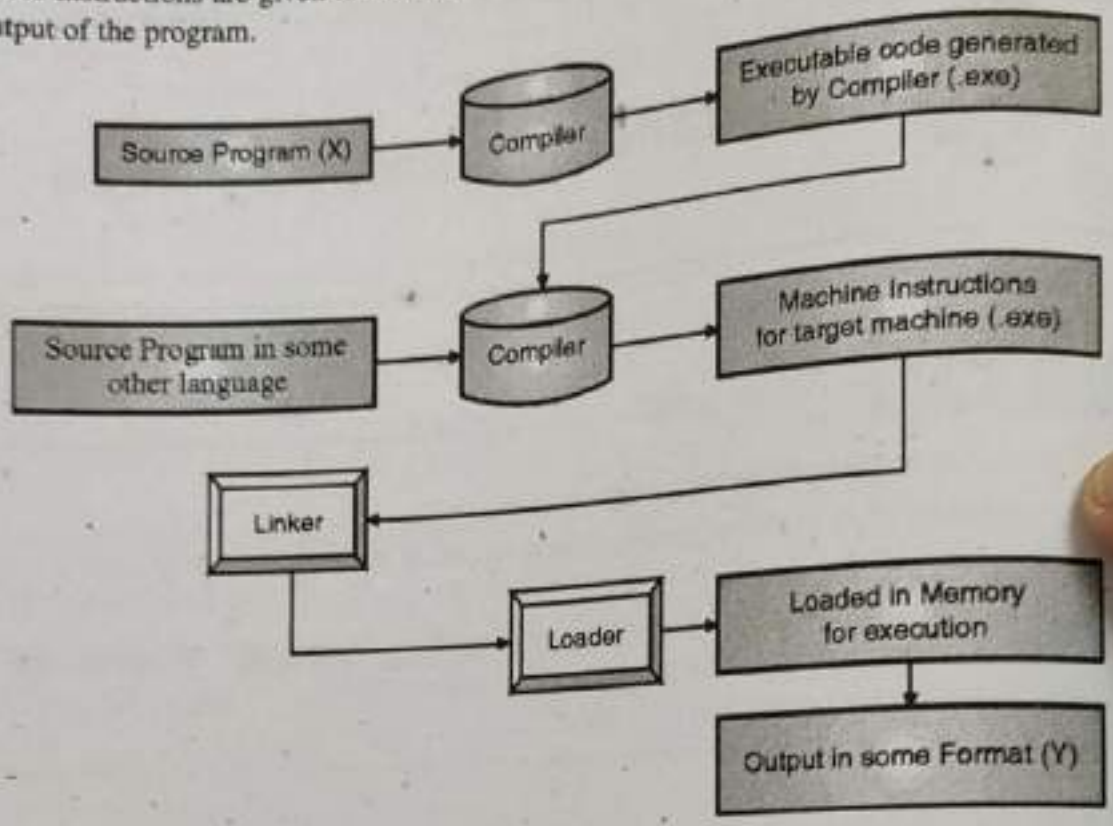


Fig. 5.1.1 : Steps in Compiling and Running a Program

The compiler is important part of system programming due to two reasons :

- Input given to the compiler is a high level language and its structure is already defined in its reference manual.
- Semantics of the language are also mentioned in the manual and attached to the program which enable, the compiler to **'understand'** the program.

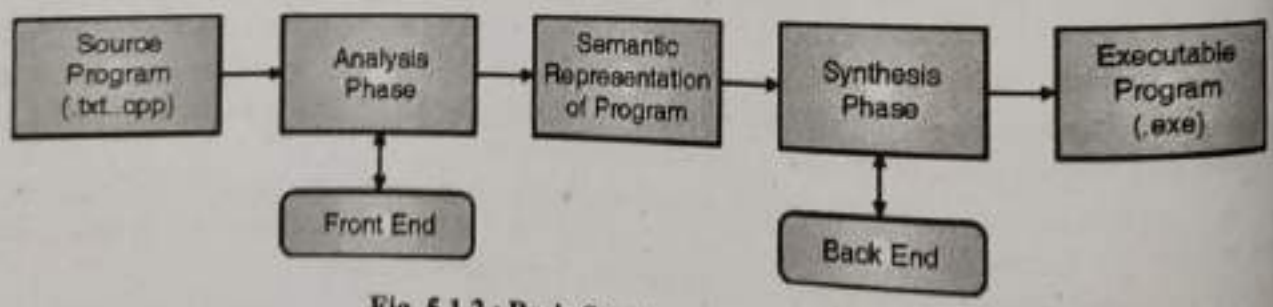


Fig. 5.1.2 : Basic Structure of a Compiler

Analysis of the source program is done in the **frond-end** of compiler and synthesis of the target program is done in the **back-end** of the compiler.

"A compiler is a complex computer program (or set of programs) that translates text written in a computer language (the source language) into another computer language (the target language)".

Scanned by CamScanner

OR

"A compiler is a computer program that translates a computer program written in one computer language into an equivalent program written in another computer language. The process of translation is called compilation".

Compiler programs often divided into several distinct chunks, called passes which communicate with one another via temporary files. These passes are the part of the compilation process.

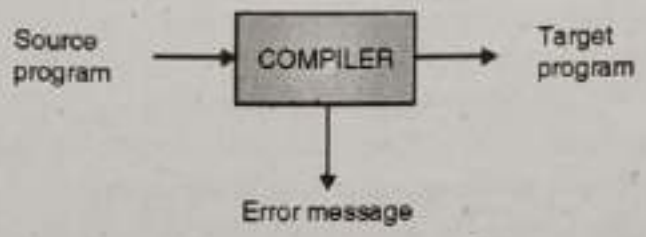


Fig. 5.1.3 : Basic Block Diagram of Compiler

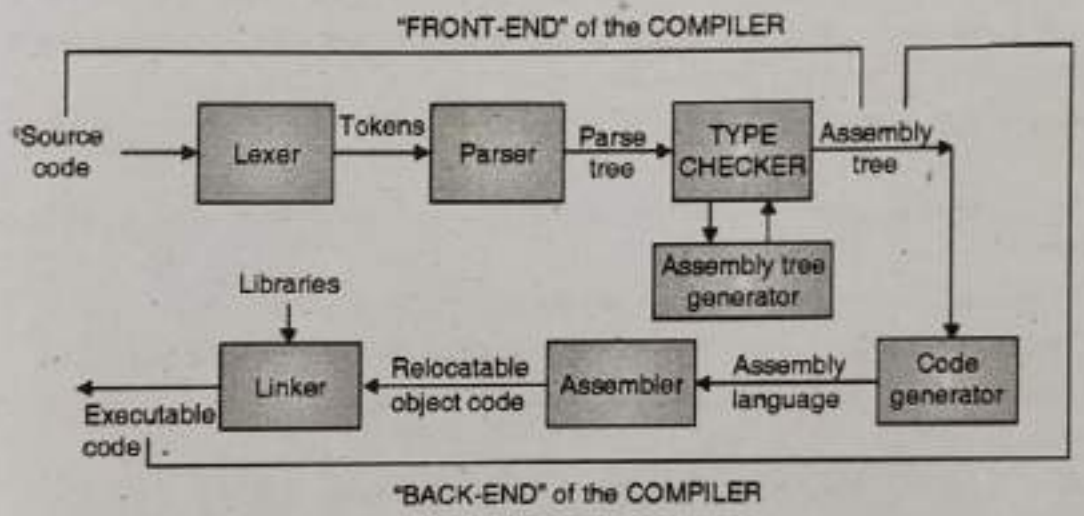


Fig. 5.1.4 : Basic Steps in Compilation Process During Various Passes

5.2 Features of a Compiler

1. The compiler should be bug free.
2. The compiler should run fast.
3. The compiler should generate correct machine code.
4. The generated machine code should run fast.
5. The compiler should have consistent and predictable optimizations.
6. The compiled target code should work well with existing debuggers.
7. Compiler should be efficient enough to print error messages.
8. A good compiler should be portable.
9. The compile should be easy and simple to implement.

Scanned by CamScanner

5.3 Compiler Requirements

Compiler is the basic building block in system programming. Compiler performs lot of work to make programmer task easy the compiler should satisfy the following requirements so that the performance of the computer increases.

The basic requirements which every compiler should follow are as :

1) **Correctness :** It is absolutely paramount. A buggy compiler is next to useless. The correctness of the compiler means that the semantics of the source language and the target language should be well defined and understood by the users and the programmers.
2) **Efficiency :** The compiler should be efficient enough to generate the target code and must be able to respond to changes in the source language and target language.
3) **Interoperability :** Compiler should be able to link all the files with the library code before they are executed.
4) **Usability :** A compiler interacts with the programmer primarily when there are errors in the program. Compiler should be able enough to generate error messages and it should also help users to debug runtime errors during program execution.
5) **Retargetability :** Compilers should be able to generate more than one target from a given source code.

Building a good compiler requires all the above mentioned features, but it also requires a knowledge of :

1) Programming languages (parameter passing, variable scoping, memory allocation etc.).
2) Theory of Automata, context free languages, Context free grammars, Regular expressions etc.
3) Algorithms and various data structures (hash tables, graph algorithms, dynamic programming etc.).
4) Knowledge of Computer Architecture and Organization (assembly language).
5) Software Engineering concepts.

5.4 Types of Compilers

There are many ways to classify compilers according to the input, output, internal structure, and runtime behavior, based on that there are basically four types of compilers. Compiling involves a program lot of computation and early computers did not have enough memory to contain a single program. So compilers were split up into smaller programs, which each made a pass over the source program.

1) One pass compiler
2) Multi pass compiler
3) Load and Go compiler
4) Optimizing compilers
5) Source to Source Compiler
6) Native Compiler
7) Cross Compiler

Scanned by CamScanner

5.4.1 One Pass Compiler

A one pass compiler is a type of compiler that passes through the source code of each compilation unit only once. In a sense a one pass complier can't look back at the code it processed previously. Another terms sometimes used for this is a narrow compiler, which emphasizes the limited scope, a one pass compiler is obliged to use.

One pass or single pass compiler is faster than the multi-pass compiler due to limited scope of availability. Some programming languages have been designed specifically to be compiled with the one pass compilers. Only an example of such is a construct in forward declaration in **PASCAL**. Normally PASCAL requires that the procedures should be fully defined before use.

E.g. One Pass Compiler

```
function odd (n : integer) : boolean ;
    begin
            if n = 0 then
                odd : = true
            else if n < 0 then
                odd : = not even (n + 1) {Compiler error : 'even' not defined}
            else
                odd : = not even (n − 1)
            end ;
    function even (n : integer) : boolean;
    begin
        if n = 0 then
            even : = true
        else if n < 0 then
            even : = not odd (n + 1)
        else
            even : = not odd (n − 1)
        end ;
```

We can remove the error by adding a forward declaration for the function 'even' before the function 'odd'.

function even (n : integer) : boolean, forward;

function even (n : integer) : boolean;

This will help in removing the compilation error.

Disadvantages

- It is not possible to perform many of the sophisticated optimizations needed to generate high quality code.
- Not able to generate efficient programs due to limited scope of availability.

Scanned by CamScanner

5.4.2 Multi-pass Compiler

A multi-pass compiler is a type of compiler that processes the source code or Abstract Syntax Tree(AST) of a program several times. They are also known as a wide compilers referring to the greater scope of the availability of the passes. In this the entire program is compiled instead of just a small portion of a program. In addition some languages cannot be compiled in a single pass, as a result of their design. The typical multi-pass compiler outputs machine code from its final pass. The multi-pass compiler consists of various passes, each pass will forward its output to the next pass as it will act as the input for further processing for other passes.

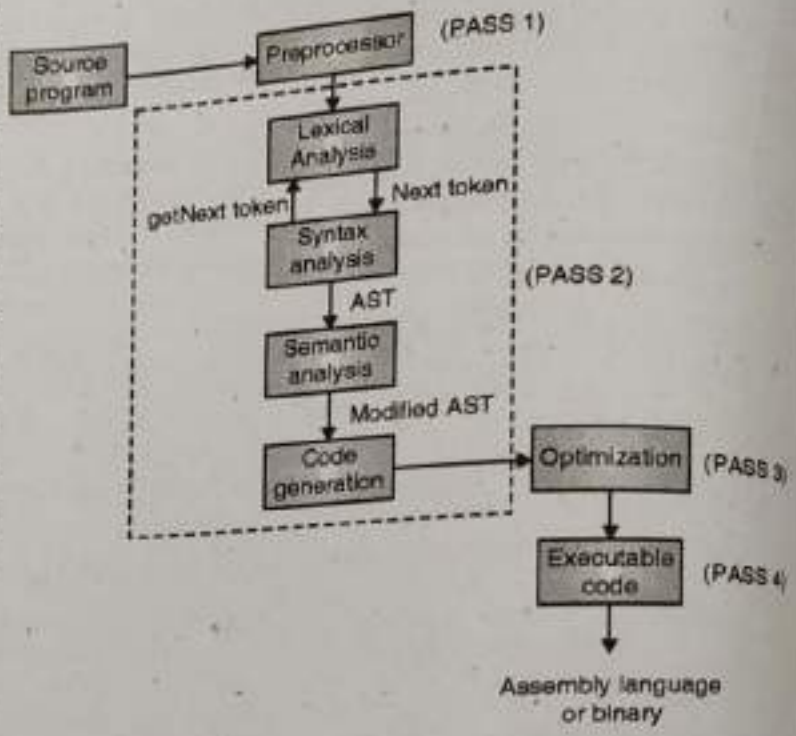


Fig. 5.4.1 : Structure of typical Four Pass Compiler

5.5 Phases of Compiler

→ **(MU - Dec. 2015, May 2016, Dec. 2017)**

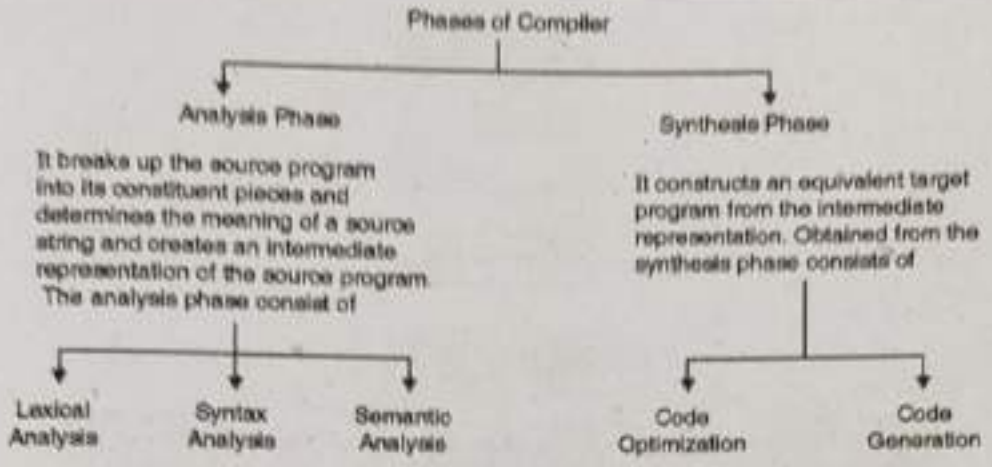
Q.	What are the different phases of Compiler?	Dec. 2015, May 2016, 5 Marks
Q.	Explain different phases of compiler with suitable examples.	Dec. 2017, 10 Marks
Q.	Explain different phases of compiler in details.	(10 Marks)

A Compiler operates in various phases, each of which transforms the source program from one representation to another. A phase is an independent task in the compilation process which transforms the source program from one representation to another.

The fundamental language processing model for compilation consists of two steps for processing of a source program.

- Analysis of Source Program
- Synthesis of Source Program

Scanned by CamScanner



There are two other activities or phases which interact with all the other phases of the compiler at the same line. These phases are as :

- Symbol Table Management
- Error Detection and Handling.

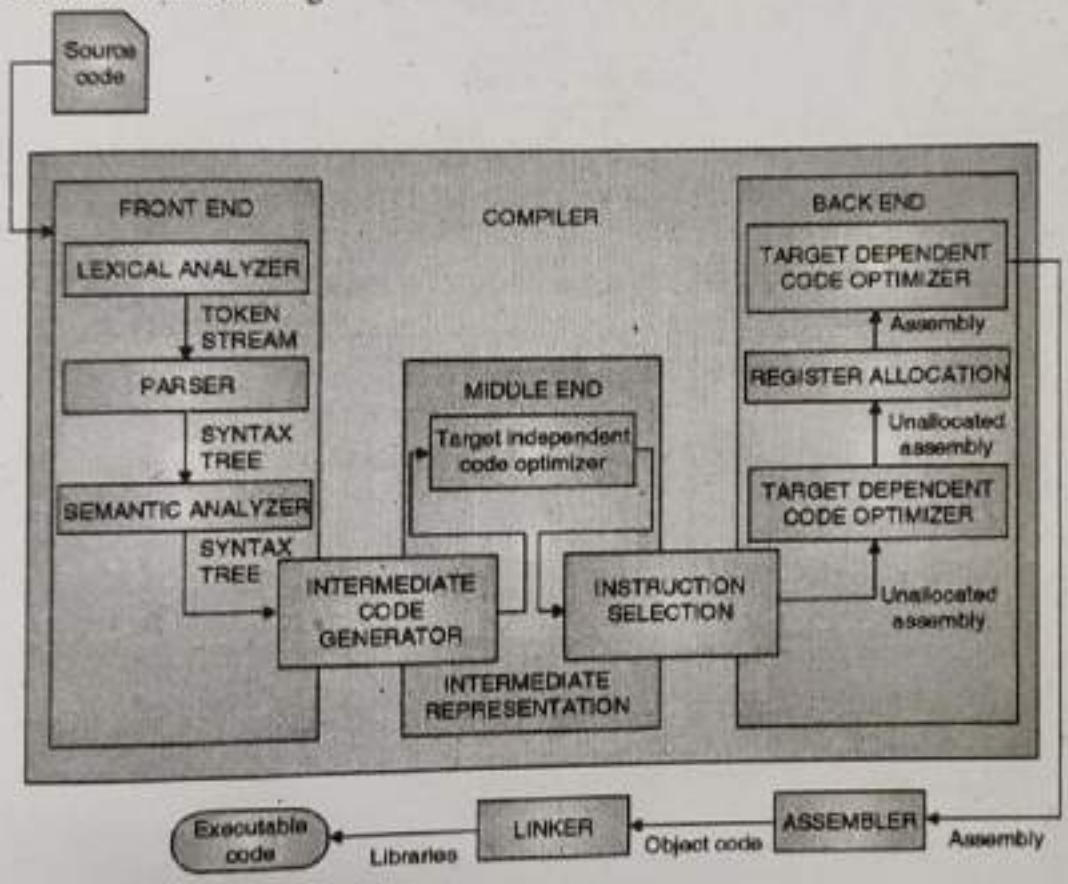


Fig. 5.5.1 : Compiler Design Overview

Compiler phases are divided into two parts :

1. Front end includes all the analysis phases and the intermediate code generator. It evaluates the source program and produces the intermediate code.
2. Back end includes the code optimization phase and final code generation phase. Back end synthesizes the target program from the intermediate code and produces the executable code for execution.

Scanned by CamScanner

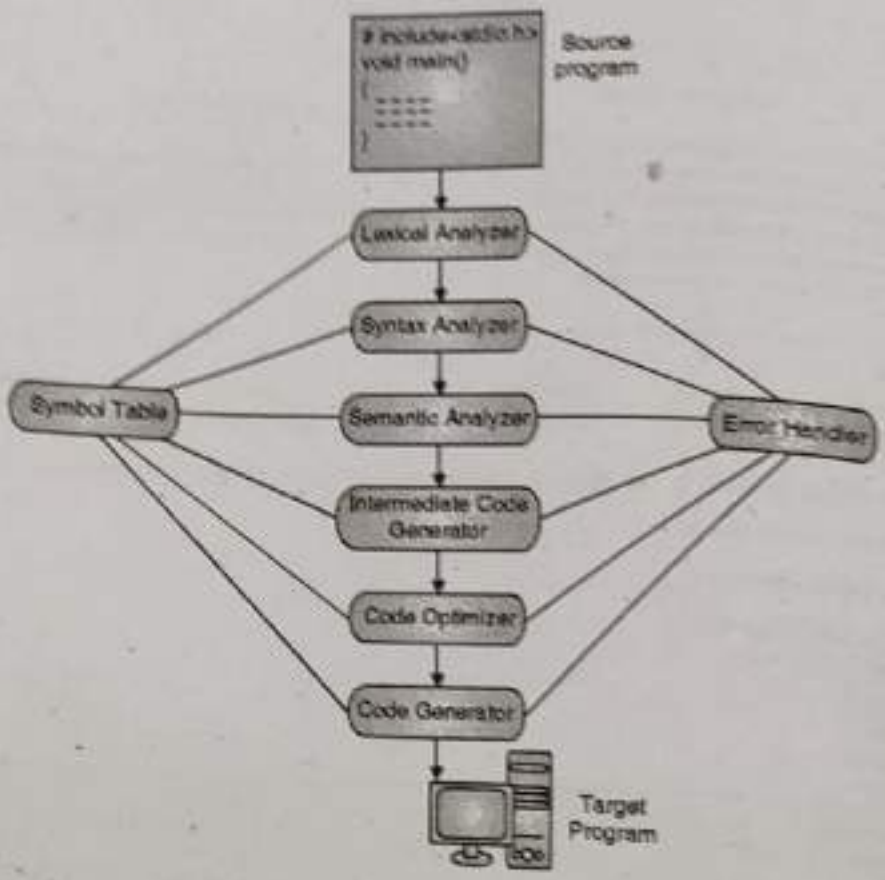


Fig. 5.5.2 : Phases of the Compiler

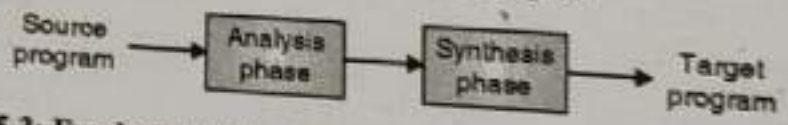


Fig. 5.5.3: Fundamental Analysis and Synthesis Phase of Compiler

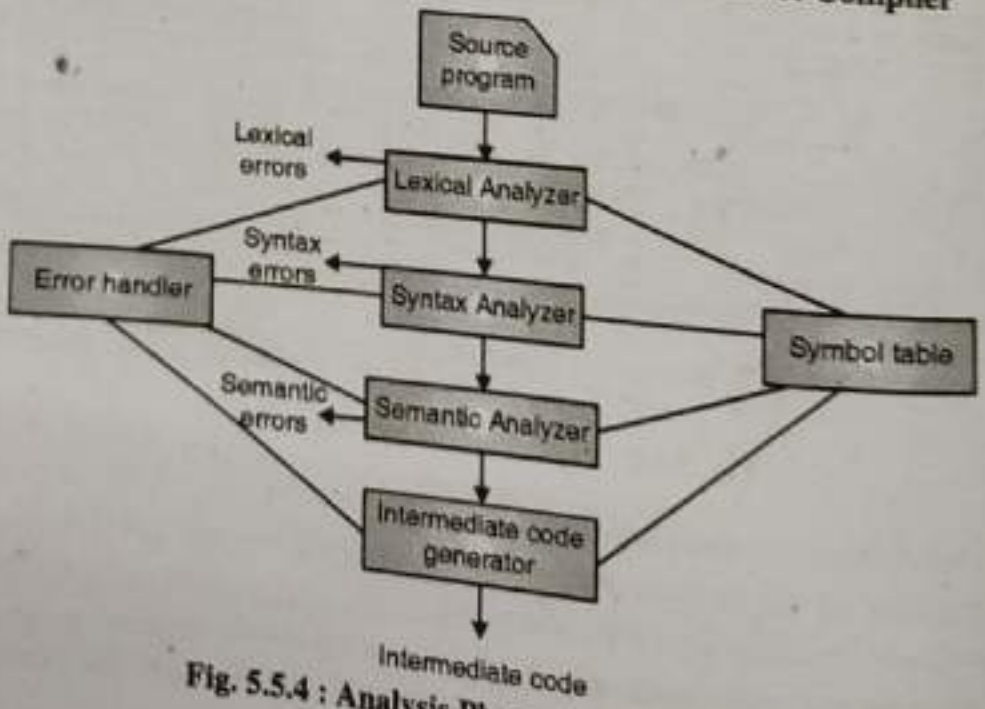


Fig. 5.5.4 : Analysis Phase of the Compiler

Scanned by CamScanner

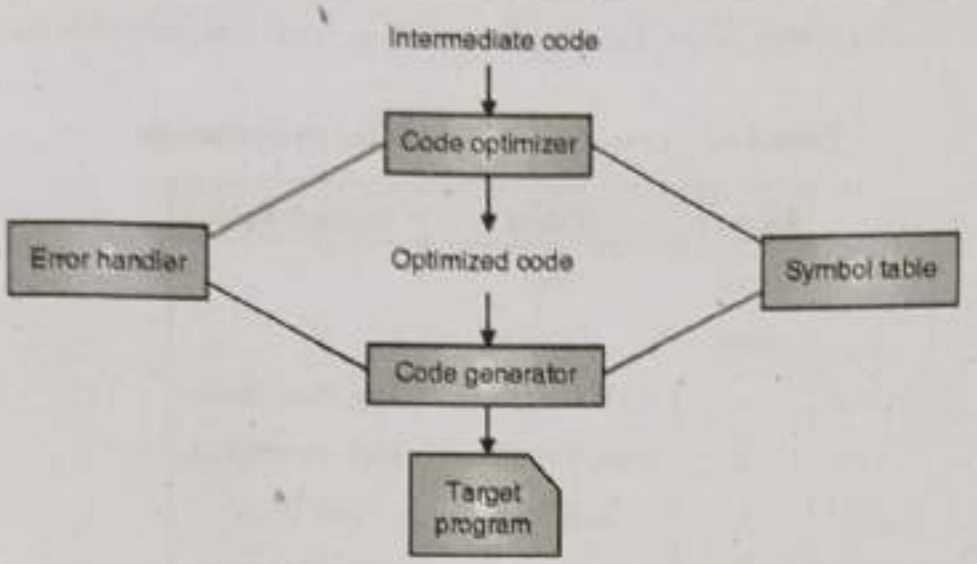


Fig. 5.5.5 : Synthesis Phase of the Compiler

5.6 Description of Different Phases of Compilation

➔ **(MU - Dec.2015, May 2016, Dec. 2017)**

Q.	What are the different phases of Compiler?	Dec. 2015, May 2016, 5 Marks
Q.	Explain various phases of compiler with suitable example.	Dec. 2017, 5 Marks
Q.	Explain different phases of compiler in details.	(10 Marks)
Q.	Explain in brief what activities are performed in various phases of the compiler.	(8 Marks)

Below mentioned description using a small example is a brief overview of the different phases of compilation. Each phase is covered in depth in next following chapters.

For example : If we write a small code in C language

```
void main ( )
{
    int x, y, z ;
    float a ;
    char b ;
    x = 10, y = 25, z = 40 ;
    if (y > x)
    a = x + y + z * 7.5 ;
    else
    a = x + y + z ;
}
```

5.6.1 Lexical Analysis

Lexical analysis is also known as linear analysis or scanning and works as interface between source program and syntax analyser. It determines the lexical constituents in a source string by reading the stream of characters from left-to-right and grouping them into tokens. **A token is a sequence of characters having a collective meaning.**

Scanned by CamScanner

When the above example passes through lexical analysis phase then the following actions take place :

Table 5.6.1 : Lexical Analysis table for given example

Lexeme	Token	Symbol Table
void	void	-
main	main	-
(	Left Paranthesis	Left Paranthesis
)	Right Paranthesis	Right Paranthesis
{	Left Brace	Left Brace
X	id 1	<id1 entry of X>
,	comma	-
Y	id 2	<id2 entry of Y>
Z	id 3	<id3 entry of Z>
;	SEMICOLON	-
A	id 4	<id4 entry of a>
B	id 5	<id5 entry of b>
=	RELOPERATOR	<assign-op>
if	IF	-
>	RELOPERATOR	<GT>
+	ADD-OP	<add-op>
*	MULT-OP	<mult-op>
7.5	Constant	-
else	ELSE	-
}	Right Brace	-

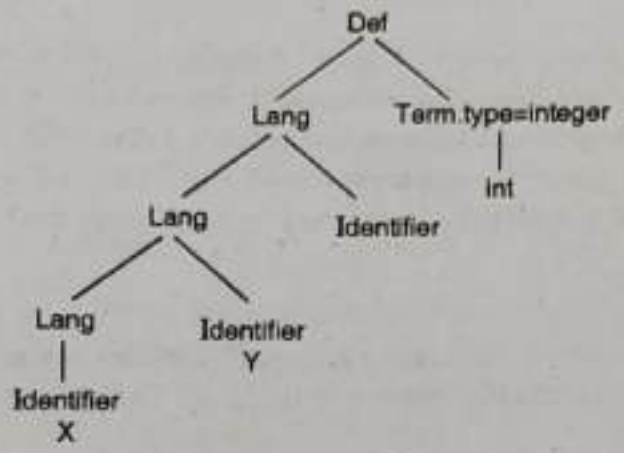
5.6.2 Syntax Analysis

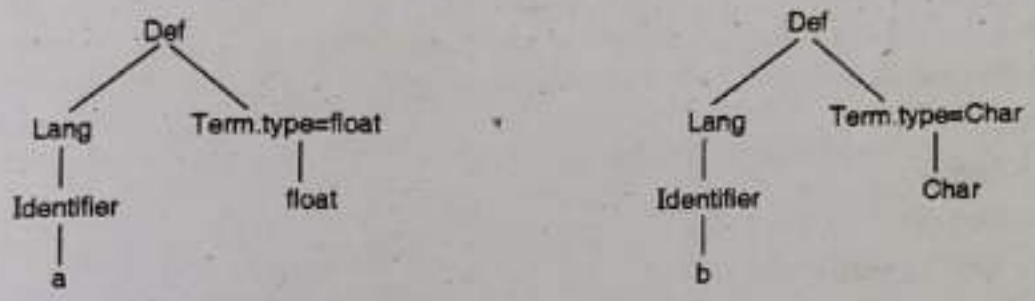
It is also known as hierarchal or parsing analysing. Parsing is the process of analyzing a sequence of tokens in order to determine its grammatical structure with respect to a given formal grammar. It determines the structure of the source string by grouping the tokens into nested collections with same meanings. The output of the syntax analyser is abstract syntax tree, **a tree with a finite**, labelled directed tree where the internal nodes are labelled by operators and leaf nodes represents the operands of the operators.

Scanned by CamScanner

The task of the parser is essentially to determine if and how the input can be derived from the start symbol within the rules of grammar. This can be done using

1. Top Down Parsing 2. Bottom Up Parsing





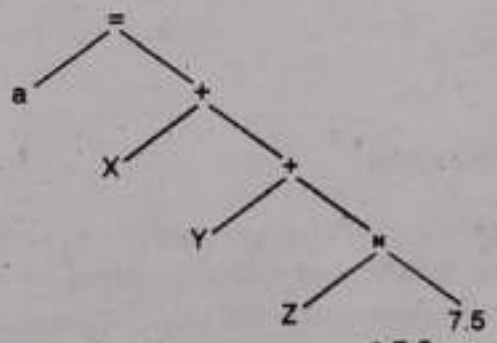
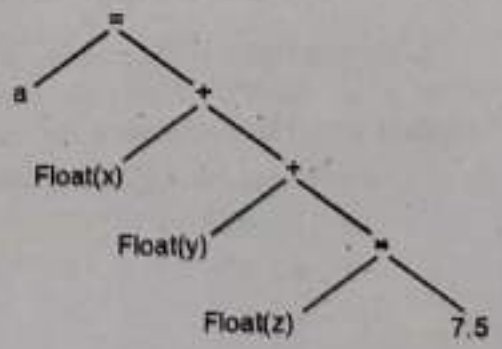


Fig. 5.6.1 : Parse Tree for a = x + y + z * 7.5

5.6.3 Semantic Analysis



Semantic analysis is used to describe the meaning of the source code. It determines the meaning of the source string and ensure that the components of a program fit together meaning fully. It adds the semantically information to the parse tree and performs certain checks. Semantic analysis performs the type checking and objects binding (binding of variables and function names to their definitions)

Scanned by CamScanner

In the above example x, y, z are defined as integer variable while 'a' is defined as float variable. So expression "**a = x + y + z * 7.5**" show error (type mismatch error)

5.6.4 Intermediate Code Generation

After syntax and semantics analysis, some compilers generate an explicit intermediate representation of the source program. The intermediate representation should have two important properties, it should be easy to produces and easy to translate into the target program. The intermediate code is represented in a term called "**three-address-code**". Each TAC can be described as a quadruple (operator, operand 1, operand 2, result), triples (operator, operand 1, operand 2). Each statement has the general form of

z = x op y

where, x, y and z are variables, constants or temporary variables generated by the compiler. 'OP' represents any operator, e.g. an arithmetic operator for the above e.g. the intermediate code will be

0 x = 10
1 y = 25
2 z = 40
3 if y > x then goto 5
4 goto next
5 temp1 = float (z)
6 temp2 = float (y)
7 temp3 = temp2 + temp1 * 7.5
8 temp4 = float (x)
9 temp5 = temp3 + temp 4
10 a = temp5

5.6.5 Code Optimization

It tries to improve the intermediate code to achieve faster running machine code. Code Optimization uses various technique as common in them are renovation, avoid redundancy, less code, straight line code, avoid memory accesses and loop optimizations.

5.6.6 Code Generation

Code generation is the process in which a code generator converts some internal representation of source code into a machine code that can be readily executed by a machine. The output of a code generator would be a machine code, assembly code and code for an abstract machine (like JVM).

For above example the code generation will do work as

R0 = X
R1 = Y
R2 = Z
R3 = a
R4 = b
R0 = 10

Scanned by CamScanner

R1 = 25
R2 = 40
R5 = 7.5
R6 = R2 * R5
R7 = R0 + R1 + R6
R3 = R7

There are two other activities or phases which interact with all the other phases of the compiler time to time.

5.6.7 Symbol Table Management

In computer science, a symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location. The attributes associated with each identifier may provide information about the storage allocated for an identifier, its type, its scope and in the case of procedure names it provides its arguments, the method of passing each argument, and the type returned. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from the record. A symbol table is used by a language translator such as compiler or interpreter where, each identifier in program's source code is associated with information relating to its declaration or appearance in the source, such as type, scope level and sometime its location. A symbol (or token) has at least the following attributes :

1. Symbol Name
2. Symbol Type (int, real, char, ...)
3. Symbol class (Static, automatic, constant, ...)
4. Symbol tables are typically implemented using hash table because good efficiency for the look up is needed.

5.6.8 Error Detection and Reporting

Programs are written by programmers and hence cannot be free from errors. Each phase encounters errors. The lexical analyzer detects all errors when the remaining characters in the input do not form any token. The syntax analyzer usually detects a large number of errors where the token stream violates the structural rules of the language. Semantic analyzer detects mismatch errors.

Error can be encountered by all the phases of a compiler they are as :

1) The lexical analyzer may be unable to proceed because the next token in the source program is misspelled undefined symbol, mistakes done like if, while, then.
2) The syntax analyzer may be unable to infer a structure for its input because a syntactic error such as missing parenthesis has occurred, type mismatch, missing "" delimiter etc.
3) The intermediate code generator may detect an operator whose operands have incompatible types.
4) The code optimizes generator may detect that certain statements can never be reached. It gives warning like unreachable code, symbol defined but never used etc.
5) The code generator may find a compiler created constant that is too large to fit in a word of the target machine. e.g. constant value too large.

Scanned by CamScanner

6) The book-keeping routine in the symbol table discovers an identifier that has been declared with contradictory attributes.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message and compiler modify the phase detecting the error.

5.7 Example of Compilation

→ (MU - Dec. 2017)

➢ **Example 5.7.1**

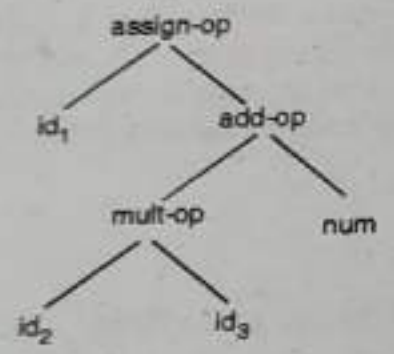
$$\boxed{a = b * z + 10}$$

$$a = b * z + 10$$

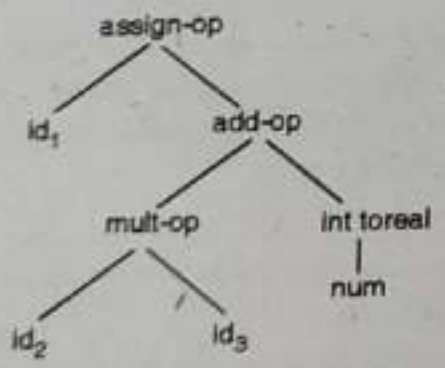
⇓ LEXICAL ANALYSIS

id_1 assign-op id_2 mult-op id_3 add-op num

⇓ SYNTAX ANALYSIS



⇓ SEMANTIC ANALYSIS



⇓ INTEMEDIATE CODE GENERATION

temp1 = id_2 * id_3

temp2 = int to real (num)

temp3 = temp1 + temp 2

id1 = temp3

Scanned by CamScanner

⇓ CODE OPTMIZATION

$t1 = id_2 * id_3$

$id1 = t1 + r_{num}$

⇓ CODE GENERATION

movf	id_3, r_2
mulf	id_2, r_2
movf	r_{num}, r_1
addf	r_2, r_1
movf	r_1, id_1

f – floating point arithmetic

r_1, r_2 two registers to store the values

➢ **Example 5.7.2**

$\boxed{A = B * C + D / E}$

$A = B * C + D / E$

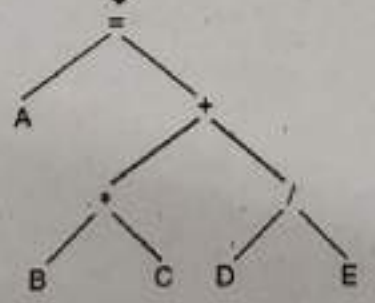
⇓

Lexical Analyzer

⇓

Lexeme	Token	Symbol Table
A	id1	<id1, entry of A in ST>
=	RELOP	<assign>
B	id2	<id2, entry of B in ST>
*	MULOP	<MUL,>
C	id3	<id3, entry of C in ST>
+	PLUSOP	<PLUS,>
D	id4	<id4, entry of D in ST>
/	DIVOP	<DIV,>
E	id5	<id5, entry of E in ST>

⇓ Syntax Analysis



⇓ Semantic Analysis

No action required during this phase as all variables and constants are in same case or type

Scanned by CamScanner

⇓ **Intermediate Code Generation**

t1 = B*C
t2 = D/E
t3 = t1 + t2
A = t3

⇓ Code Optimization

t1 = B * C
t2 = D/E
A = t1 + t2

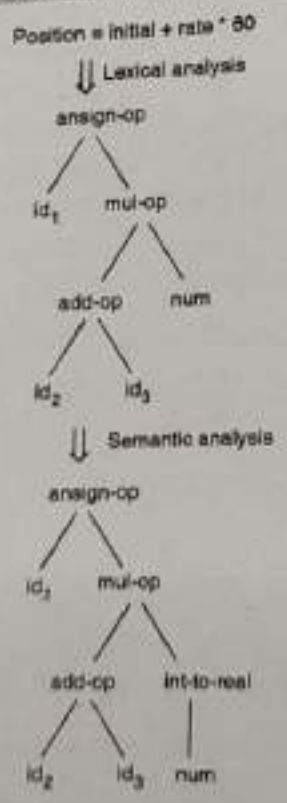
⇓ CODE GENERATION

R1 ← B
R2 ← C
R3 ← B*C
R1 ← D
R2 ← E
R4 ← D/E
R1 ← R3 + R4
A ← R1

Above mentioned both examples show how the action is performed in each phase.

Example 5.7.3 : Illustrate compilers internal presentation of source program for following statement after each phase position = initial + rate * 60. **MU - Dec. 2015, 5 Marks**

Solution :



Scanned by CamScanner

⇓ Intermediate code generation

temp 1 = $id_2 + id_3$

temp 2 = int to real (num)

temp 3 = temp 1 * temp 2

id1 = temp 3

⇓ Code optimization

t1 = $id_2 + id_3$

id_1 = t1 + num

⇓ Code generation

mov f id_3, r_2

add f id_2, r_2

mov f r_{num}, r_1

mult f r_2, r_1

mov f r_1, id_1

f - floating point arithmetic

r_1, r_2 – two registers to store the values.

Example 5.7.4 : What are the various phases of a compiler taking the following as input statement while (a > b), c = b + 2.0 * a – 50; **MU - May 2014, 10 Marks**

Solution :

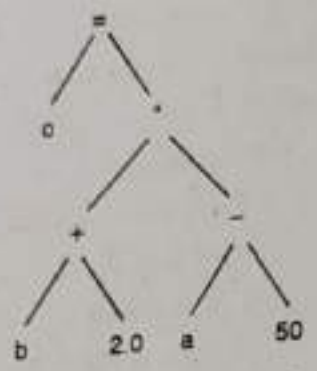
$$C = b + 2.0 \times a - 50$$

⇓ Lexical analyzer

Lexeme	Token	Symbol Table
C	id_1	< id_1, entry of C in ST >
=	RELOP	< assign >
b	id_2	< id_2, entry of b in ST >
+	ADDOP	< ADD >
2.0	Num	< CONSTANT >
*	MULOP	< MUL >
a	id_3	< id_3, entry of a in ST >
–	MINUSOP	< MINUS >
50	NUM	< CONSTANT >

Scanned by CamScanner

⇓ Syntax analysis



⇓ Semantic analysis

No action are required during the phase as all variables and constants are in same case or type.

⇓ Intermediate code generation

t1 = b + 2.0
t2 = a – 50
t3 = t1 * t2
C = t3

⇓ code optimization

t1 = b + 2.0
t2 = a – 50
C = t1 * t2

⇓ Code generation

R1 ← B	R3 ← B + 2.0	R4 ← a – 50
R2 ← 2.0	R1 ← a	R1 ← R3 * R4
	R2 ← 50	C ← R1

Example 5.7.5 : Illustrate compilers internal representation of source program or following statement after each phase.
Amount = P + P * N * R / 100

MU - May 2016, 5 Marks

Solution :

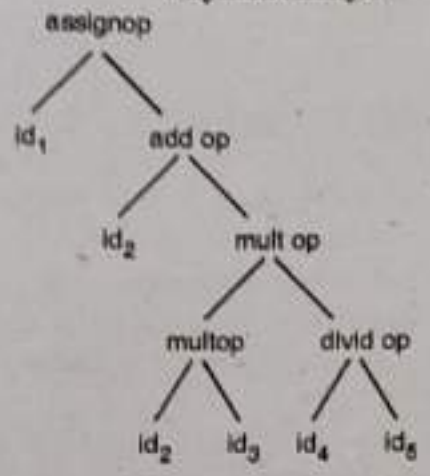
Amount = P + P * N * R / 100

⇓ **Lexical Analysis**

id, assign-op id_2 addop id_3 multop id_4 multop id_5 divide op num

Scanned by CamScanner

⇓ Syntax Analysis



⇓ semantic analysis

Convert all the variables into one form either into int or float by type caste int to float or float to int

⇓ Intermediate code Generation.

t0 = P
t1 = P * N
t2 = R | 100
t3 = t1 * t2
t4 = t0 + t3
Amount = t4

⇓ code optimization

t0 = P
t1 = t0 * N
t2 = R / 100
t3 = t1 * t2
Amount = t0 + t3

⇓ code generation

R1 ←P
R2 ←N
R3 ←R
R4 ←R1 * R2
R5 ←R| 100
R6 ←R4* R5
R7 ←R1 + R6

Scanned by CamScanner

5.8 Interpreter Vs Compiler

→ (MU - May 2014)

May 2014, 5 Marks

Q. Compare the performance of compilers and interpreters.

As we have already seen, the only language that a computer can understand is so called *machine language*. There are the translators between the high level programming language and the machine languages. There exists two sorts of *translators*.

- **Interpreter :** It is a type of translator which translates a programming language into a specified language called intermediate code which can be directly executed using a program called an interpreter. Interpreter translates program line by line. Hence, it is usually smaller than that of a corresponding compiled objects program.
- **Compiler :** Compiler is a program which takes a High Level Language as input and produces a output program which is in other language. That means source language is a high level language such as FORTRAN, COBOL and object language is a low-level language such as an assembly language, machine language etc.

Sr. No.	Interpreter	Compiler
1.	Interpreter executes program line by line	Compiler execute whole program at a time.
2.	Interpreter analyzes each statement in the program each time it is executed, that is why interpreting code is slower.	Compiler analyzes the whole program at once before executing, so compiling code is faster.
3.	Execution time is larger	Execution time is less.
4.	Size of interpreter in small (in sense of space complexity).	Size of compiler is large (in sense of space complexity).
5.	Duet to slower execution, except small in space complexity interpreter is not efficient.	Compiler is more efficient due to fast execution, except large in space complexity.
6.	In interpretation the resulting intermediate form, some system-specific data structure, is high-to-medium level.	In compilation the resulting intermediate form, machine-specific binary executable code is low-level.
7.	In interpretation the interpreting mechanism is a software program.	In compilation the interpreting mechanism is the hardware CPU.
8.	Using an interpreter, a single source file can produce equal results even in vastly different systems (e.g. a PC and a Play Station 3).	Using a compiler, a single source file can produce equal results only if it is compiled to distinct, system-specific executables.
9.	It does not create an object file.	It creates an object file.
10.	Software cost is less.	Software cost is high.
11.	The code size of an interpreter is smaller.	The code size of the compiler is much larger.
12.	For every new run of the program, the program needs to be translated.	Program needs to be compiled only once and can be executed repeatedly.
13.	Example-Basic Interpreter.	Example-C Compiler.

Scanned by CamScanner

5.9 Lexical Analysis : Introduction

Q. Write a short note on Lexical Analyzer. (5 Marks)

Lexical Analysis is the first phase of a compiler. The main task accomplished by lexical analysis is to identify the set of a valid word of the language that occurs in an input stream. Lexical analysis is the interface of the compiler to the outside world. The task of this phase is to scan the input source program and identify the valid words within it. It also does the additional job of removing cosmetics like

- Extra white space, comments added by the user etc from the program,
- Expanding the user defined macros like # include and # define in C, reporting of foreign words that are the words, not belonging to the language of the source program if any.

In computer science, lexical analysis is the first stage of processing the language. It is the process of converting a sequence of characters into a sequence of tokens. Stream of characters making up the source program or other input is read one at a time and grouped into lexemes (or "tokens"). A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner. A lexer often exists as a single entity which is called by the parser or another function.

This chapter elaborates the concept and tools involved in the process of lexical analysis. A simple way to build the lexical analyzer is to construct a diagram that illustrates the structure of the tokens of source language.

Software tools have been developed to write the lexical analysers. The best known such utility is Lex. Lex is a lexical analyzer generator for the UNIX operating system, targeted to the C programming language. Lexical analyzer generator is that can utilize the best known pattern matching algorithms for developing lexical analysers.

5.10 Role of Lexical Analyzers

→ (MU - Dec. 2014)

Q. Explain role of lexical analyser. Dec. 2014, 5 Marks
Q. Write a short note on Lexical Analyzer. (5 Marks)

Lexical analysis or scanning is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens, which may be handled more easily by a parser. It acts as an interface between the input source program to be compiled and the later stages of the compiler. Each token is a sequence of characters that represents a unit of information in the source program.

- Lexical analyzer inputs the source text and, it accomplish various secondary tasks like:
- Eliminate comments and white spaces in the form of blanks, tabs and new line characters.
  - It helps in finding out the errors from the compiler with the source program.
  - Error messages are linked with the line number. The analyzer keeps track of the line number by tracking the number of new line characters found.

Scanned by CamScanner

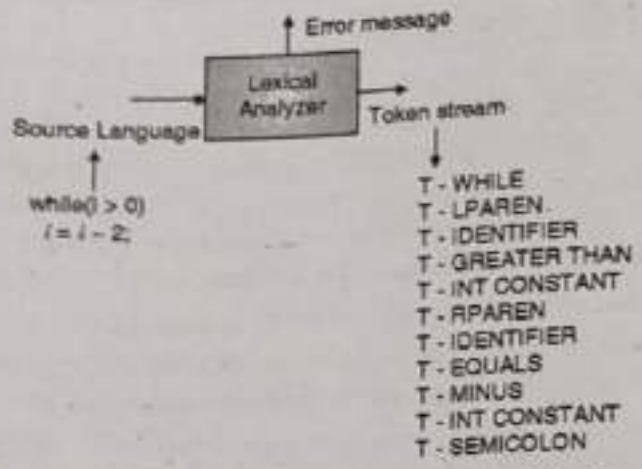


Fig. 5.10.1 : Basic Block Diagram of Lexical Analyser

- The relationship between lexical analyser and the parser is well defined. The parser calls a single lexical analyser subroutine every time it needs a new token and that subroutine reads input characters until it can identify the next token and returns it to the parser. Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token. This relationship is shown in Fig. 5.10.2.

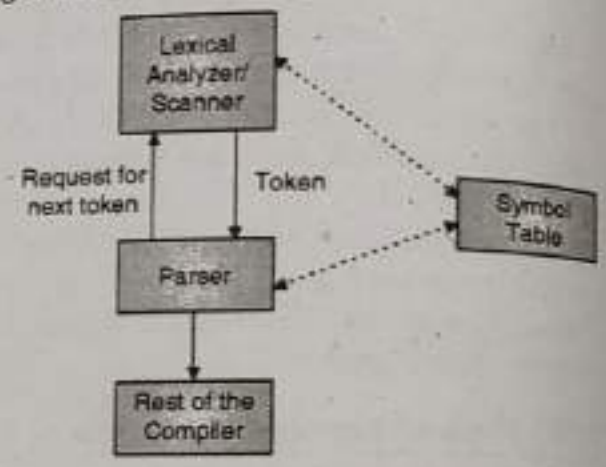


Fig. 5.10.2 : Interaction between Scanner and the Parser

Sometimes lexical analysers are divided into a cascade of two phases, the first called "scanning" and the second "lexical analysis". The former is responsible for doing simple tasks while the later does the complex tasks.

5.10.1 Features of Lexical Analyser

Various features of Lexical Analyser are as follows :

- Return a token to syntax analyser.
- Eliminate white spaces.
- Keeps track of the line numbers.
- Generates output listing with errors marked.
- Delete comments.
- If language has macros it expands.
- Converts number to internal form.

Scanned by CamScanner

5.10.2 Role of Finite Automata in Lexical Analysers

➔ (MU - Dec. 2014, May. 2016, Dec. 2016)

Q.	Explain role of finite automata in compiler.	Dec. 2014, 5 Marks
Q.	What is the role of an automata in compiler design ?	May 2016, 5 Marks
Q.	What is role of finite automata in compiler theory ?	Dec. 2016, 5 Marks

An automaton (plural : automaton) is a self-operating machine. The word is sometimes used to describe a robot, more specifically an autonomous robot. Automaton is more often used to describe non-electronic moving machines, especially those that have been made to resemble human being or animal actions.

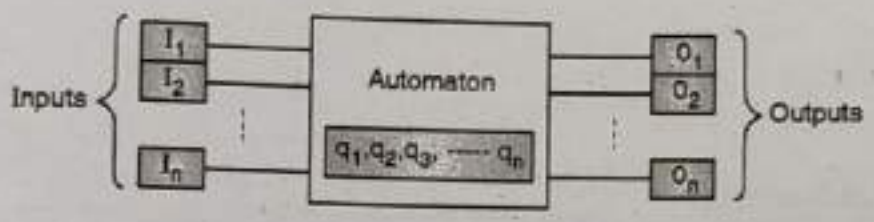


Fig. 5.10.3 : Block diagram of Automata

Characteristics of Automata

(i) **INPUT :** At each of the discrete instants of time input values $I_1, I_2 \ldots I_n$ are applied to the input side of the model.

(ii) **OUTPUT :** $O_1, O_2 \ldots O_n$ are the output of the model.

(iii) **STATES :** At any time automata is anyone of the state $q_1, q_2, q_3, q_4 \ldots$

(iv) **TRANSITION RELATION :** The next state of automata is determined by present state of the automata.

(v) **OUTPUT RELATION :** Output of automata is depend only on state or both, input and state.

Finite Automata

Finite Automata is a machine with a finite number of states in which machine can perform. There is a distinctive start state, from which the machine starts. Finite Automata is a recognized for a language that takes as input a string x and answer "yes" if x is a sentence of the language and "no" otherwise. We compile a regular expression into a recognizer by constructing a generalized transition diagram called **finite automata.**

- Strictly speaking, an FSM consists of the following :

(a) A finite set of states.

(b) A set of transition (or moves) from one state to another. Each transition is labelled with a character from the input alphabet.

(c) A special start state.

(d) A set of final or accepting states.

Scanned by CamScanner

5.11 Issues in Lexical Analysis

Analysis phase is divided into two parts first is called parsing and second is known as lexical analysis due to several

1. The separation leads to simpler design which helps in good and efficient language design.
2. Efficiency of compiler is improved which helps in building an efficient processor for the various tasks like input buffering etc.
3. Enhancement in compiler portability.
4. Minimal compiler size
5. Minimal size of object code and production of efficient object code
6. Ease of maintenance
7. Optimal usability

5.12 Basic Definitions

Lexical analyzer does not have to be an individual phase. But having a separate phase simplifies the design and improves the efficiency and portability. A lexical analyzer, or lexer for short, can be thought of having two stages, as scanner and lex analysis.

- During first phase, the scanner is usually based on a finite state machine. The information is encoded on the possible sequences of characters that can be contained within any of the tokens it handles.
- In second phase of lexical analyzer it evaluates lex to construct a token, which goes over the characters of the lexeme to produce a value. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser.

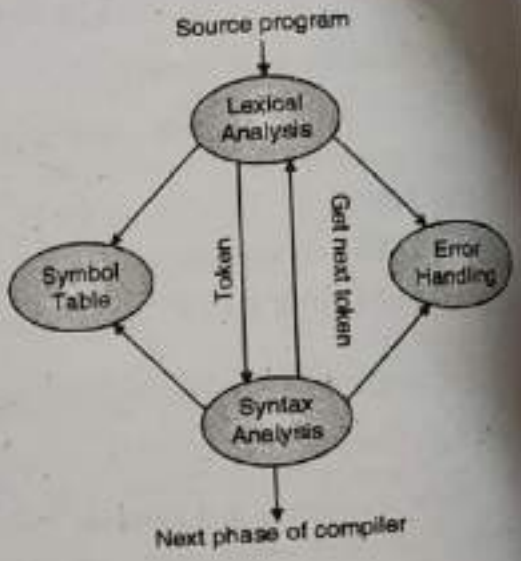


Fig. 5.12.1 : Role of Lexical Analyzer

5.12.1 Lexemes

A lexeme is a basic lexical unit of a language consisting of one word or several words. That corresponds to a set of words that are different forms of the same word. For example, run, runs, ran and running' are terms of the same lexeme.

Lexemes are the smallest logical units of a program. It is a sequence of characters in the source program for which a token is produced. For example, if 10.0, t, etc.

Scanned by CamScanner

5.12.2 Tokens

"Lexical token/tokens are a sequence of characters that can be treated as a unit in the grammar of the programming languages".

Example of tokens are as :

1. **Type tokens** ("id" {a-z, A - Z, 0 - 9}, "num" {0 - 9}, "real, ...)
2. **Punctuation tokens** ("if", "void", "return", "begin", "call", "const", "do", "end", "if", "while", "then", ...).
3. **Alphabetic tokens** ("keywords")
4. **Symbol tokens** ('+', '–', '*', '/', '=' '(' ')', ' : =' '<', '<=', '< >', --)
5. **Non-tokens**

   Comments, pre-processor directive, macros, blanks, tabs, new line ...

5.12.3 Patterns

A pattern is a term, template or model (or, more abstractly, a set of rules) which can be used to make or to generate things or part of a thing especially if the things that are generated have enough in common for the underlying pattern to be inferred.

'A pattern is a set of a string in input for which the same token is produced as output. These set of string is described by a rule called **pattern**'.

OR

"A pattern is a rule describing the set of lexeme that represent particular token."

➢ **Example 5.12.1**

Token	Lexeme	Pattern
Literal	"hello"	chars with in " "
Number	0.31, 5.0	List of numbers
Identifier	my var, your var	Letter followed by list of letters AND/OR Digits.
RELOP	>, =, < = , > =	List of relational operators
VAR	VAR	VAR (keyword usually reserved word)

➢ **Example 5.12.2 : Const a = 10;**

Token	Lexeme	Pattern
const	const	const
if	If	if
relation	<, < =, =, < >	< or < = or < > or > =
id	pi, count D2	letter followed by letter and digits
num	3.14, 0, 6.02	any numeric constant
literal	"core dumped"	any characters between "a to z"

Scanned by CamScanner

Example 5.12.3 : Consider the following function definition in C.

```
Sum (int n1, int n2)
{
    int sum1 = 1 ;
    sum1 = n1 + n2 ;
    return (sum1) ;
}
```

Solution :

Tokens	Lexemes
keywords	return, int
identifier	sum, sum1, n1, n2
Delimeter	(, ; ,), {,}
Assign op	=
Add op	+

Example 5.12.4 : This book-by-xyz = (Project-manager)

Solution :

Lexeme	Token	Symbol Table
this book-by-xyz	id1	<id1, entry of this book by xyz>
=	RELOP	<assign-op>
(	LP	LP
Project	id2	<id2, entry of project>
–		–
manager	id3	<id3, entry of manager>
)	RP	RP

RP – Right Parenthesis, LP – Left Parenthesis

5.13 Design of Lexical Analyzer

→ **(MU - Dec. 2014, May 2015, Dec. 2017)**

Q.	Explain LEX.	Dec. 2014, 2.5 Marks
Q.	Write short note on : LEX.	May 2015, Dec. 2017, 2.5/10 Marks

Lex is commonly used with the YACC parser generator. It was originally written by Eric Schmidt and Mike Lesk. Lex is a program generator designed for lexical processing of character input streams. The code written in Lex recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. Lex helps in writing the programs whose control flow is directed by instances of regular expressions in the input stream.

Scanned by CamScanner

The regular expressions are given to the Lex and it associates the regular expressions and the program fragments. Lex is not a complete language, but rather a generation representing a new language feature which can be added to different programming languages, called "host languages." Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user.

As lex is adaptable to various environments and users so it provides the compatible run-time libraries for the different host languages. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.

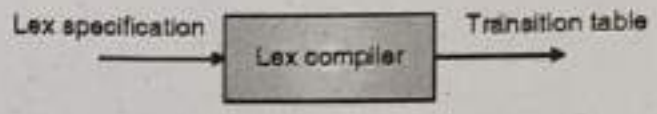


Fig. 5.13.1 : Lex compiler

A lexical analyzer generator creates a lexical analyzer using a set of specifications usually in the format.

d_1 $\{act_1\}$
d_2 $\{act_2\}$
d_3 $\{act_3\}$
d_4 $\{act_4\}$
. . . . . .
d_n $\{act_n\}$

d_i is a regular expression

act_i is a program code for execution

When a lexeme match is found in the input the pattern that matches the longest lexeme is chosen for the execution.

The tool accepts a specification of the tokens in terms of extended regular expressions along with supporting routines. It produces a C-program (*lex.yy.c*) as an output, which can be compiled to obtain the executable version of the lexical analyzer.

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the programming language. Lex can also be used in combination with the parsers where parsers can call the function *yylex* generated automatically. The overall strategy for building the lexical analyzer is shown in Fig. 5.13.2.

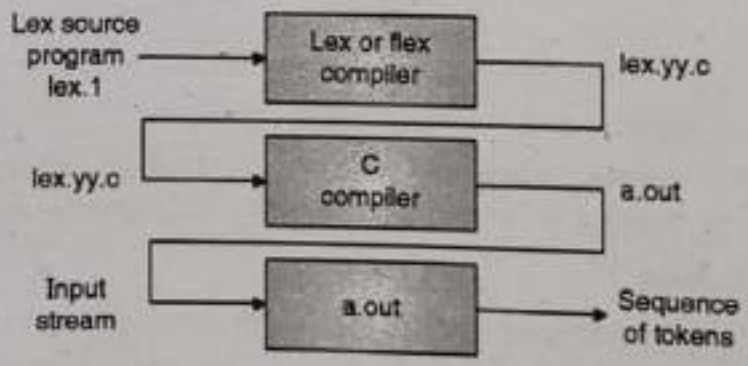


Fig. 5.13.2 : Creating a Lexical Analyzer with LEX and FLEX

The lexical source program, e.g. *lex.l*, is given to *lex or flex* compiler to produce the **C program** file known as *lex.yy.c*, which is then compiled to create an *a.out*. This generated lexical analyzer (a.out) can now be passed with an input stream to find and generate the tokens in it.

Scanned by CamScanner

Lexical analyzer is generally carried out using a finite automaton. Transition table for the finite automaton is built from the regular expressions patterns in the lexical analyzer generator. The lexical analyzer itself consists of a finite automaton simulator that uses this transition table to look for the regular expression patterns in the input buffer.

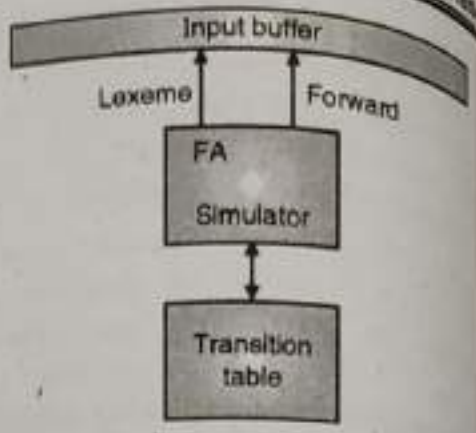


Fig. 5.13.3 : Model of a lex compiler

5.13.1 Data Structure for Lexical Analyzer

Any Lex program consists of three parts, separated by lines that contain only two percent signs, as follows :

{definitions}

% %

{rules}

% %

{user subroutines}

- The **definition** section is the place to define macros and to import header files written in C. It includes variable declarations, constant declarations, and regular definitions etc.
- The **rules** section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. The rule section contains the specification of tokens and the associated actions. This section is of form :

  p_1 {action $_1$}

  p_2 {action $_2$}

  . . . . . .

  p_n (action $_n$}

  Here, each p_i is a regular expression.
- The **user subroutine** contains code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link during the compile time.

Example 1

The following is an example lex file for the flex(**fast lexical analyzer generator** is a free software alternative to lex. It is frequently used with the free Bison parser generator. Flex was written in C by Vern Paxson around 1987.) version of lex. It recognizes strings of numbers (integers) in the input, and simply prints them out.

```
/*** Definition section ***/
% {
/* C code to be copied verbatim */
```

Scanned by CamScanner

```
#include <stdio.h>
    %}
    /* This tells flex to read only one input file */
    % option noyywrap
%%
        /*** Rules section ***/
        /* [0 – 9] + matches a string of one or more digits */
[0 – 9] + {
            /* yytext is a string containing the matched text. */
            printf ("Saw an integer : %s\n", yytext) ;
        }
        {   /* Ignore all other characters. */ }
    %%
/*** C code section ***/
int main (void)
    {
        /* Call the lexer, then quit. */
        yylex ( ) ;
        return 0 ;
    }
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input :

abc123z. ! & * 2ghj6

the program will print :

Saw an integer : 123

Saw an integer : 2

Saw an integer : 6

Table 5.13.1 : Lex pattern matching primitives

Metacharacter	Matches
.	any character except newline
\n	Newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line

Scanned by CamScanner

Metacharacter	Matches
a/b	a or b
(ab) +	one or more copies of ab (grouping)
"a + b"	literal "a + b"
[ ]	character class

The most simple Lex program is the following single-line one :

```
%%
```

Input is copied to output, one character at a time. The first %% is always required, as there must be a rules section. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. This rule is used to match everything and provide the output. Lex uses stdin (standard input) and stdout (standard output) for calling input and output routines. For example

```
%%
        /* match everything except newline */
        ECHO ;
```

Lexical Analysis Tool-Lex

Table 5.13.2 : Pattern Matching Examples

Expression	Matches
Abc	Abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of a, b, c
[a-z]	any letter between a-z
[a\-z]	one of a, -, z
[-az]	one of -, a, z
[A-Za-z0-9] +	one or more alphanumeric character
[\t\n] +	Whitespace
[^ ab]	anything except a b
[a ^ b]	one of a, ^, b
[a|b]	one of a, |, b
a|b	one of a, b

Scanned by CamScanner

In more complex actions, the user will often want to know the actual text that matched some expression like [a-z]+. Lex leaves this text in an external character array named yytext. Variable yytext is a pointer to the matched string (NULL-terminated), and yyleng is the length of the matched string. Thus, to print the name found, a rule like

```
[a-z]+   printf("%s", yytext);
```

will print the string in yytext.In this case, the format is ``print string" (% indicating data conversion, and s indicating string type), and the data are the characters in yytext.. Table 5.13.3 presents some of the most widely used predefined variables and functions of Lex.

Table 5.13.3 : Lex predefined variables and functions

Name	Function
int yylex (void)	call to invoke lexer, returns tokens
char *yytext	pointer to matched string
Yyleng	length of matched string
Yylval	value associated with the token
yymore()	indicate that the next input expression is recognized
yyless()	used to reprocess text in various circumstances
int yywrap (void)	return 1 if done, 0 otherwise (more processing is required)
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start
ECHO	write matched string
input( )	which returns the next input character (I/O routines in Lex)
output(c)	which writes the character c on the output
unput(c)	pushes the character c back onto the input stream to be read later by input().

Example 2

Here is a program that recognizes strings of numbers (integers) in the input, and simply prints them out.

```
/*** Definition section ***/

%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
```

Scanned by CamScanner

```
%option noyywrap

%%
  /*** Rules section ***/

  /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
        /* yytext is a string containing the matched text. */
        printf("Saw an integer: %s\n", yytext);
    }

    {  /* Ignore all other characters. */  }

%%
/*** C Code section ***/

int main(void)
{
  /* Call the lexer, then quit. */
  yylex();
  return 0;
}
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input :

```
abc123z.!&*2ghj6
```

the program will print:

```
Saw an integer: 123
Saw an integer: 2
Saw an integer: 6
```

Debugging Lex

Lex provides debugging of the program. This feature may vary with different versions of Lex. Below mentioned are some commands for the debugging of Lex.

'-b, --backup, %option backup'

Generate backing-up information to lex.backup. This is a list of scanner states which require backing up and the input characters on which they do so.

'-d, --debug, %option debug'

makes the generated scanner run in *debug* mode.

'-p, --perf-report, %option perf-report'

Scanned by CamScanner

generates a performance report to stderr. The report consists of comments regarding features of the flex input file which will cause a serious loss of performance in the resulting scanner.

'-s, --nodefault, %option nodefault'

causes the *default rule* (that unmatched scanner input is echoed to stdout) to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error.

'-w, --nowarn, %option nowarn'

suppresses warning messages.

'--warn, %option warn'

warn about certain things. In particular, if the default rule can be matched but no default rule has been given, the flex will warn you. We recommend using this option always.

The code generated by Lex in the file lex.yy.c includes debugging statements that are enabled by specifying command line option "-d".

5.13.2 Using Lex with Other Programming Tools

Using Lex with Yacc

Lex and Yacc (a parser generator) are commonly used together. Yacc uses a formal grammar to parse an input stream, sometime which Lex cannot do using simple regular expressions (Lex is limited to simple finite state automata). However, Yacc cannot read from a simple input stream-it requires a series of tokens. Lex is often used to provide Yacc with these tokens.

5.14 Introduction

Every programming language has a rules that prescribes the syntactic structure of well formed programs. Syntax analysis is the next phase after lexical analysis in the compiler design. Syntax analysis or parsing is a major component of the front end of a compiler.

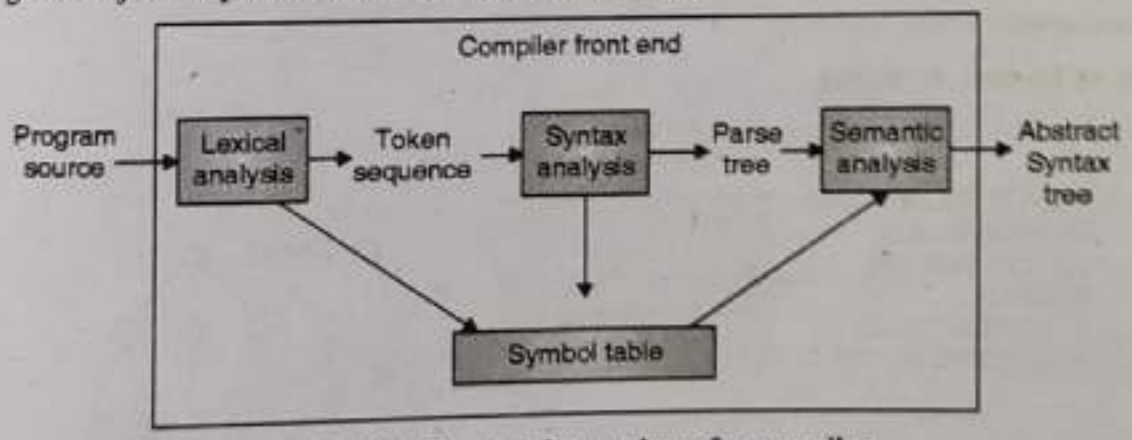


Fig. 5.14.1 : Frontend overview of a compiler

The syntactic analysis of a source code entails the transformation of a linear sequence of tokens into a hierarchical syntax tree or abstract syntax trees. Syntax analysis, also known as parsing, extracts the syntactic units within the input stream, it works in hand-in hand with lexical analyzers. The lexical analyzers determines the input stream of tokens and gives them to the parser and syntax analyzer checks the tokens whether they are valid or not.

Scanned by CamScanner

Lexical Analysis

Identify dynamic language constructs. Each type of construct is represented by a token.

For e.g. 3.14 → float, if → IF, a → ID

Syntax Analysis

It checks if a token sequence is correct with respect to the language specification.

Semantic Analysis

It checks type of relations and consistency rules for different tokens generated.

Each step involves a transformation from a program representation to another.

Overview of Lexical Analysis

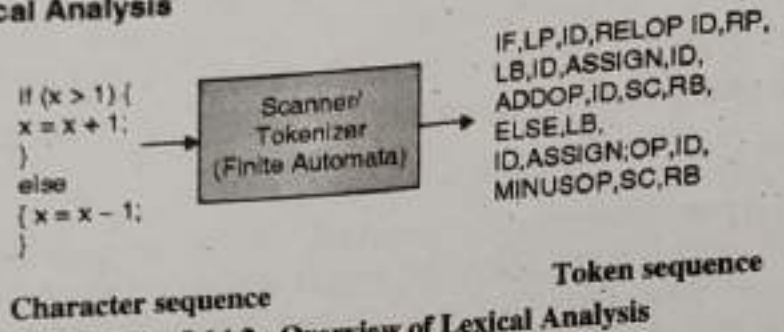


Fig. 5.14.2 : Overview of Lexical Analysis

In Lexical Analysis the :

1. **Input program representation :** Character sequence
2. **Output program representation :** Token sequence
3. **Analysis specification :** Regular Expressions
4. **Recognizing (abstract) machine :** Finite Automata
5. **Implementation :** Finite Automata

Overview of Syntax Analysis

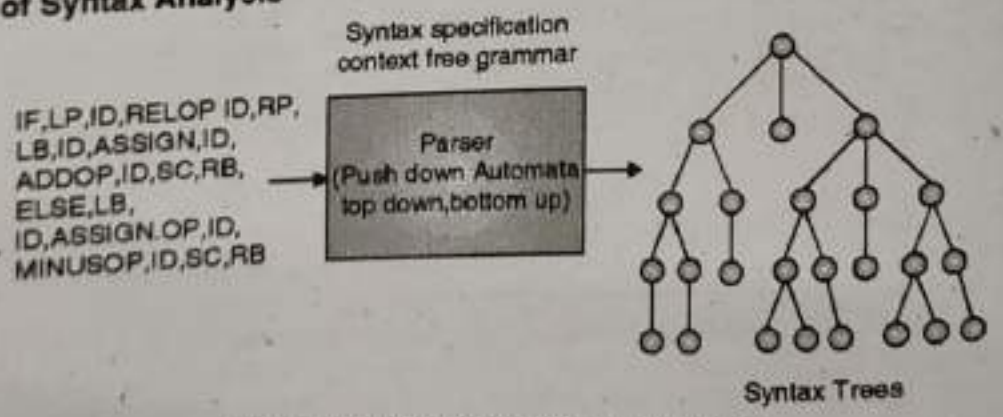


Fig. 5.14.3 : Overview of Syntax Analysis

In Syntax Analysis the :

1. **Input program representation :** Token sequence
2. **Output program representation :** Parse (or syntax) tree

Scanned by CamScanner

3. **Analysis specification :** Context free grammar
4. **Recognizing (abstract) machine :** Push down-automata
5. **Implementation :** Top-down or Bottom up parsers.

In this chapter, we will deal with the basic concepts and definitions such as context free grammar parse tree, non-ambiguous grammar, ambiguity and left recursion with the help of illustrative examples.

Parsing Algorithms are broadly classifieds into top-down and bottom-up techniques based upon the complexity of the parsing strategies. Within each category, there are several parsers – like **predictive** parsers (recursive and non-recursive) shift-reduce parsers, operator precedence, LR **parsers** etc.

5.15 Role of the Parser

The parser takes the stream of tokens generated by the lexical analyzer and take out the constructs of the language within the stream of tokens. Thus the role of parser is given as

1. It eliminates the language constructs present in a given input stream of token and identifies whether the tokens are valid or not. If tokens are valid then input is represented in form of parse tree.
2. Parser finds all the larger structure in the program and group them together. This process is done to validate that the string can be produced by the grammar for the source program.
3. Parser checks the input and if it is grammatically wrong then parser declares the errors in it and no parse tree can be generated from it. The syntax analysis phase also try to recover from the errors so that it can process the root of the input.

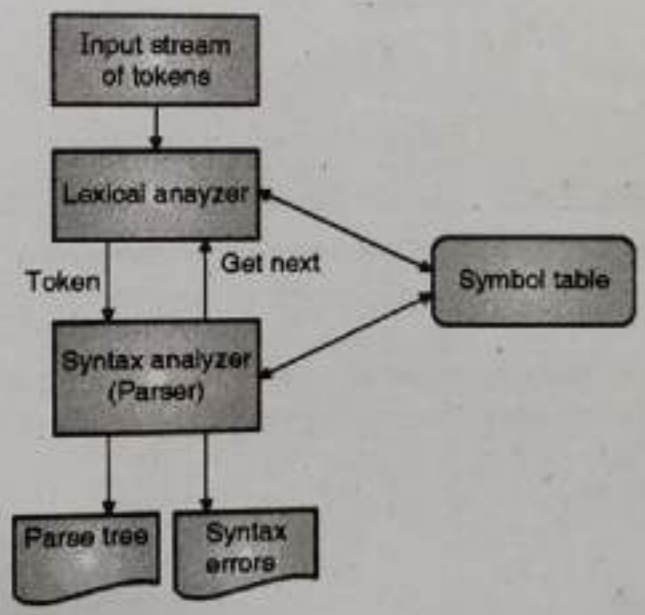


Fig. 5.15.1: Role of Parser

5.16 Role of Context Free Grammars In Syntax Analysis

Q. Define and explain the following terms with suitable example: Context Free Grammars. **(5 Marks)**

Scanned by CamScanner

A context free grammar (CFG) has 4 tuple $<V_N, V_T, P, S>$ where

(i) V_N : Set of non terminal symbols for writing the grammar.

(ii) V_T : Set of terminal symbol T, used as tokens for the language.

(iii) P : Set of production rules. The left hand side of each rewriting rule is a single non-terminal. The right-hand side of each rewriting rule is a string of terminals and non-terminals.

(iv) S : A special non-terminal SEN, which is the start symbol.

We know regular expressions generate strings of characters, the context free grammars generate strings of tokens. A string of tokens generated by the CFG in the following way.

(1) The initial input string is the start symbol S.

(2) While there are non-terminals on the left of the string :

(a) Pick any non terminal on the input string A.

(b) Replace a string occurrence of A in the string with the right hand side of any rule that has A as the left hand side.

Terminals = {id, num, if, then, else, print, = , {, }, ; , (,) }.

Non terminals = {S, E, L}

Rules = (1) S → print (E) ;
(2) S → while (B) do S
(3) S → {L}
(4) E → id
(5) E → num
(6) B → E > E
(7) L → S
(8) L → SL

Start symbol = S

Fig. 5.16.1 : A CFG for some simple statements

Example 5.16.1

Consider a grammar to generate arithmetic expressions consisting of numbers and operator symbols +, −, *, / and ↑. The rules for the grammar can be written as,

E → EAE
E → (E)
E → −E
E → number
A → +
A → −
A → *
A → /
A → ↑

We can apply the above mentioned production rules to derive the expressions "6*(8 + 2*4)"

Scanned by CamScanner

$E \rightarrow EAE \rightarrow EA(E) \rightarrow EA\ (EAE) \rightarrow EA\ (EAEAE)$

$\rightarrow EA\ (EAEA4) \rightarrow EA\ (EAE*4) \rightarrow EA(EA2*4)$

$\rightarrow EA\ (E + 2*4) \rightarrow EA\ (8 + 2 * 4) \rightarrow E * (8 + 2 * 4) \rightarrow 6 * (8 + 2 * 4)$

In this E, A are non-terminals +, −, * are terminals.

Example 5.16.2

$S \rightarrow U/v$

$U \rightarrow TaU \mid TaT$

$V \rightarrow TbV \mid TbT$

$T \rightarrow aTbT \mid bTaT \mid \varepsilon$

Here the non terminal T can generate all strings with the same number of a's and b's. The non terminal U generates all strings with more a's then b's and the non terminal V generates all strings with fewer a's than b's.

Example 5.16.3

Consider the following grammar G which will generate the palindrome string of 0's and 1's.

$G = (\{P\}, \{0, 1\}, A, P)$

Where A is defined by the set of rules

$P = \varepsilon \quad P \Rightarrow 1\,P\,1$

$P = 0 \quad \Rightarrow 11P11$

$P = 1 \quad \Rightarrow 110P011$

$P = 0P0 \quad \Rightarrow 1100P0011$

$P = 1P1 \quad \Rightarrow 110010011$ (Palindrome string)

Example 5.16.4

<sentence> = <noun phrase> <predicate>

<noun phrase> = <article> <noun>

<predicate> = <verb>

<article> = a

<article> ⇒ the

<noun> ⇒ cat

<noun> ⇒ dog

<verb> ⇒ run

<verb> ⇒ walk

Derivation of the sentence "the dog walks" is done as :

<sentence ⇒ <noun phrase> <predicate>

⇒ <noun phrase> <verb>

⇒ <article> <noun> <verb>

⇒ the <noun> <verb>

Scanned by CamScanner

⇒ the dog <verb>

⇒ the dog walks.

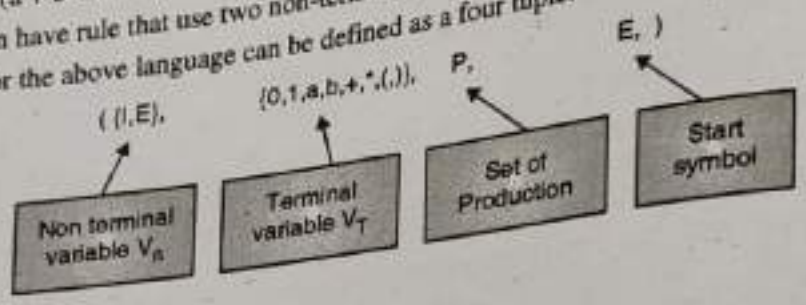
➢ **Example 5.16.5**

Let us consider a regular expression.

R = (a + b) (a + b + 0 + 1)

Here we can have rule that use two non-terminal variables say E and I.

The CFG for the above language can be defined as a four tuple.



Production rules for given G.

(1) E ⇒ I
(2) E ⇒ E + E
(3) E ⇒ E * E
(4) E ⇒ (E)
(5) I ⇒ a
(6) I ⇒ b
(7) I ⇒ Ia
(8) I ⇒ Ib
(9) I ⇒ I0
(10) I ⇒ I1

5.16.1 Notational Conventions for Context Free Grammars

Notational conventions used for the context free grammars throughout the remainder of this chapter/book are as

(1) The list of terminals :
- Lower case letters occurring starting with e, f, g etc.
- Operators such as +, −, =, etc.
- Punctuation such as brackets, parenthesis, comma etc.
- Digits from 0 ..... 9.
- Boldface strings like id, number, while, if etc.

(2) The list of non terminals are :

Upper case letters starting from A, B, C etc.

(i) S is used to signify start symbol.

(ii) Lower case words like expr, stmt are used to indicate nonterminals.

(3) Upper case letters like X, Y, Z represent symbols may be non-terminals or terminals.

(4) Strings of terminals is given by lower case greek letters like α, β, etc.

(5) Lower case letters like x, y, z represent strings of terminals.

(6) Set of productions rules with same left hand side like, $A \to \alpha_1$, $A \to \alpha_2$, ..., $A \to \alpha_k$ is written as $A \to \alpha_1 \mid \alpha_2 \mid \ldots \alpha_k$.

Scanned by CamScanner

(7) If there is no start symbol, then the non terminal appearing in the left hand side of the production rule is used to represent as the start symbol.

(8) '$\rightarrow$' symbol stands for production. e.g. if there is a production, say $A \rightarrow b$ we can say, "A is producing b".

(9) '$\Rightarrow$' Symbol stands for the process of derivation or generation of any string from the language e.g. if the set 'P' is consisting of production.

$$S \rightarrow AB,\ A \rightarrow a,\ B \rightarrow b$$

$$S \Rightarrow aB$$

Here the symbol '$\Rightarrow$' stands for "deriving in single step".

(10) '$\overset{*}{\Rightarrow}$' stands for "deriving in zero or more number of steps" e.g. we can write,

$S \overset{*}{\Rightarrow} ab$ for above generation process.

(11) '$\overset{+}{\Rightarrow}$' stands for "deriving in one or more steps".

Example 5.16.6

The grammar with the following productions defines simple arithmetic expressions.

expr $\rightarrow$ expr op expr

expr $\rightarrow$ (expr)

expr $\rightarrow$ – expr

expr $\rightarrow$ id

op $\rightarrow$ +

op $\rightarrow$ –

op $\rightarrow$ *

op $\rightarrow$ /

op $\rightarrow \uparrow$

Using these standards, we would write the grammar of the Example 5.16.6 as follows.

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow \mid$ $\rightarrow$"OR" symbol is not part of the grammar.

E, A $\rightarrow$ Non terminals

E $\rightarrow$ Start Symbol.

5.16.2 Derivation

The chain of intermediate strings generated to enlarge the start symbol of the grammar to a desired string of terminals is called a derivation. Every move in derivation generates an intermediary string to a new one by replacing a substring. The derivation can also be represented in the form of tree, called **parse tree**.

Scanned by CamScanner

➢ **Example 5.16.7**

The derivation of a + b + c + d can be represented as

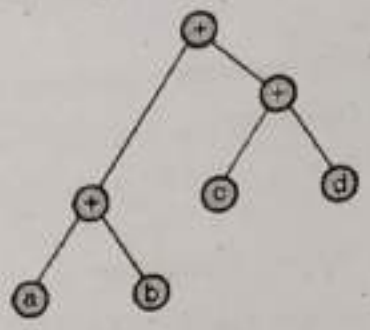


Fig. 5.16.2 : Parse tree for "a + b + c + d"

➢ **Example 5.16.8**

Consider the grammar G = ({I, E}, {0, 1, a, b, +, *, (,)}, P, E)

Find out the strings in the language defined by G. Consider the production rule mentioned in Example 5.16.5:

(i) Production rule 5 (I ⇒ a) implies that string a is in L.

(ii) Production rule 6 (I ⇒ b) implies that string b is in L.

(iii) Production rule 9 (I ⇒ I0) implies that strings a0 is in L.

Sr. No.	String Inferred	For Language of	Production used	Strings used
(i)	A	I	I ⇒ a	
(ii)	B	I	I ⇒ b	
(iii)	b0	I	I ⇒ I0	(ii)
(iv)	b00	I	I ⇒ I0	(iii)
(v)	A	E	E ⇒ I	(i)
(vi)	b00	E	E ⇒ I	(iv)
(vii)	a + b00	E	E ⇒ E + E	(v), (vi)
(viii)	(a + b00)	E	E ⇒ (E)	(vii)
(ix)	a* (a + b00)	E	E ⇒ E * E	(v), (viii)

Fig. 5.16.3 : Inferred strings for "a* (a + b00)"

➢ **Example 5.16.9**

Consider the following statement

while (id > num) do print (id);

Scanned by CamScanner

We can generate this from the given CFG in Fig. 5.16.1 as follows.

Start with the start symbol	S ⇒
rewrite S using rule (2) we get	while (B) do S ⇒
rewrite the B using rule (6) to get	while (E > E) do S ⇒
rewrite the first E using rule (4) to get	while (id > E) do S ⇒
rewrite the remaining E using rule (5) to get	while (id > num) do S ⇒
rewrite the S using rule (1) to get	while (id > num) do print (E), ⇒
rewrite the E using rule (4) to get	while (id > num) do print (id);

We can also generate

{print (id); print (num); } from the CFG given in Fig. 5.16.1 as follows

Start with symbol	S ⇒
rewrite the S using rule (3) to get	{L} ⇒
using rule (8) we get	{SL} ⇒
using rule (1) we get	{print (E) ; L} ⇒
using rule (4) we get	{print (id) ; L} ⇒
using rule (7) we get	{print (id) ; S} ⇒
using rule (1) we get	{print (id) ; print (E); } ⇒
using rule (5) we get	{print (id) ; print (num) ;}

There are two common ways to describe how a given string can be derived from the start symbol of a given grammar. The two different derivations possible to derive the given string are as,

5.16.2(A) Leftmost Derivation

The derivation obtained by replacing the leftmost variable by one of its production bodies at each step, is called the **leftmost derivation**. Leftmost derivation is indicated by a $\underset{lm}{\Rightarrow}$.

➤ **Example 5.16.10**

Derivation of string a* (a + b00) using leftmost derivation is as :

$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow} a^* (E)$

$\underset{lm}{\Rightarrow} a^* (E + E) \underset{lm}{\Rightarrow} a^* (I + E) \underset{lm}{\Rightarrow} a^* (a + E) \underset{lm}{\Rightarrow} a^* (a + I) \underset{lm}{\Rightarrow} a^* (a + I0) \underset{lm}{\Rightarrow} a^* (a + I00)$

$\underset{lm}{\Rightarrow} a^* (a + b00)$

➤ **Example 5.16.11**

Let us consider the following grammar

(1) $S \rightarrow S + S$ (2) $S \rightarrow 1$ (3) $S \rightarrow a$

Derive the string "1 + 1 + a" using a leftmost derivation

$S \rightarrow S + S$ (By (1))

$S \rightarrow S + S + S$ (By (1))

Scanned by CamScanner

$S \rightarrow 1 + S + S$ (By (2))
$S \rightarrow 1 + 1 + S$ (By (2))
$S \rightarrow 1 + 1 + a$ (By (3))

➢ **Example 5.16.12**

Let the grammar G be consist of ({E}, {+, *, id}, P, E)

Where P is

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow id$

Generate the string "id + id" using leftmost derivation.

$E \Rightarrow E + E$
$E \Rightarrow id + E$ (using (3))
$E \Rightarrow id + id$ (using (3))

At each step in the derivation we get some string containing both terminals and non terminals i.e. sentential form. In leftmost derivation they are termed as left sentential forms.

e.g. "E + E", "id + E" and "id + id" are left sentential form.

5.16.2(B) Rightmost Derivation

The derivation obtained by replacing the rightmost variable by one of its production bodies at each step, is called the **rightmost derivation.** A rightmost derivation is indicated by a $\underset{rm}{\Rightarrow}$.

It can also be termed as a canonical derivation.

➢ **Example 5.16.13**

Rightmost derivation of string "a* (a + b00)" is as

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E) \underset{rm}{\Rightarrow}$$

$$E * (E + I) \underset{rm}{\Rightarrow} E * (E + I0) \underset{rm}{\Rightarrow} E * (E + I00) \underset{rm}{\Rightarrow} E * (E + b00) \underset{rm}{\Rightarrow} E * (I + b00) \underset{rm}{\Rightarrow} E * (a + b00)$$

$$\underset{rm}{\Rightarrow} I * (a + b00) \underset{rm}{\Rightarrow} a * (a + b00)$$

➢ **Example 5.16.14**

Considering the grammar defined in Example 5.16.11 the rightmost derivation of string "1 + 1 + a" is as

$S \rightarrow S + S$ (by (1))
$S \rightarrow S + S + S$ (by (1))
$S \rightarrow S + S + a$ (by (3))

Scanned by CamScanner

$S \rightarrow S + 1 + a$ (by (2))

$S \rightarrow 1 + 1 + a$ (by (2))

Example 5.16.15

Consider the same grammar defined in Example 5.16.12 derive the string "id + id * id" using rightmost derivation.

$E \Rightarrow E + E$ (using 1)

$E \Rightarrow E + E * E$ (using 2)

$E \Rightarrow E + E * id$ (using 3)

$E \Rightarrow E + id * id$ (using 3)

$E \Rightarrow id + id * id$ (using 3)

At each step in the derivation we get some string containing both terminals and non-terminals sentential forms. In rightmost derivation they are termed as **right sentential form**.

E.g. "E + id * id", "E + E", "E + E * id" are right sentential forms

Example 5.16.16

Consider the grammar given as, G = ({S, A, B}, {a, b}, P, S}

Where P consists of

(1) $S \rightarrow aB / bA$

(2) $A \rightarrow a / aS / bAA$

(3) $B \rightarrow b / bS / aBB$

derive "bbaaba" using leftmost and rightmost derivations.

(i) Leftmost derivation

$S \Rightarrow bA$ rule (1)

$S \Rightarrow bbAA$ rule (2)

$S \Rightarrow bbaA$ rule (2)

$S \Rightarrow bbaaS$ rule (2)

$S \Rightarrow bbaabA$ rule (1)

$S \Rightarrow bbaaba$ rule (2)

(ii) Rightmost derivation

$S \Rightarrow bA$ rule (1)

$S \Rightarrow bbAA$ rule (1)

$S \Rightarrow bbAa$ rule (2)

$S \Rightarrow bbaSa$ rule (2)

$S \Rightarrow bbaaBa$ rule (1)

$S \Rightarrow bbaaba$ rule (3)

Scanned by CamScanner

➤ **Example 5.16.17**

Consider the grammar

G = ({S, A, B,}, {a, b, c}, S, P)

Where P is

$S \rightarrow 0B / 1A$

$A \rightarrow 0 / 0S / 1AA$

$B \rightarrow 1 / 1S / 0BB$

With this grammar, for the string "00110101" find

(a) Leftmost Derivation
(b) Rightmost Derivation

a) Leftmost Derivation

$$\begin{aligned} S &\Rightarrow 0B \\ &\Rightarrow 00BB \\ &\Rightarrow 001B \\ &\Rightarrow 0011S \\ &\Rightarrow 00110B \\ &\Rightarrow 001101S \\ &\Rightarrow 0011010B \\ &\Rightarrow 00110101 \end{aligned}$$

b) Rightmost Derivation

$$\begin{aligned} S &\Rightarrow 0B \\ &\Rightarrow 00BB \\ &\Rightarrow 00B1 \\ &\Rightarrow 001S1 \\ &\Rightarrow 0011A1 \\ &\Rightarrow 00110S1 \\ &\Rightarrow 001101A1 \\ &\Rightarrow 00110101 \end{aligned}$$

5.16.3 Derivation of Parse Trees

A syntax tree or parse tree is an ordered tree that is used to represents the syntactic structure of a string according to some given grammar or production rules. The non-terminals are used to represent the internal nodes and the terminals are used to represents the leaf nodes in the parse tree.

A program that produces such trees is called a **parser**. It is made up of nodes and branches. Parse trees may be generated for sentences in natural languages. Parse trees are distinct from abstract syntax trees (also known simply as syntax trees), in that their structure and elements more concretely reflect the syntax of the input language.

Scanned by CamScanner

Below mentioned image shows the parse tree for the English sentence.
"John hits the ball"

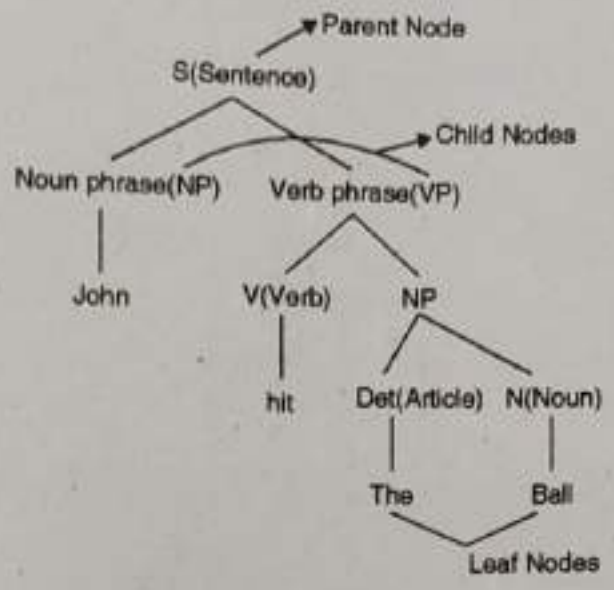


Fig 5.16.4 : Parse tree for English sentence "John hit the ball"

Example 5.16.18

Consider CFG :

$S \rightarrow SS+$

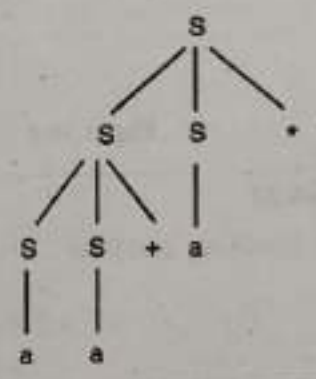
$S \rightarrow SS*$

$S \rightarrow a$

and generate 'aa + a*' and construct parse tree also

$$S \Rightarrow SS*$$
$$\Rightarrow SS+S*$$
$$\Rightarrow aa+a*$$

Parse tree :



Example 5.16.19

Consider the following grammar :

$S \rightarrow (L)$

$S \rightarrow a$

$L \rightarrow L, S$

$L \rightarrow S$

Under this grammar, the parse tree of the sentence (a, ((a, a), a)) is :

Scanned by CamScanner

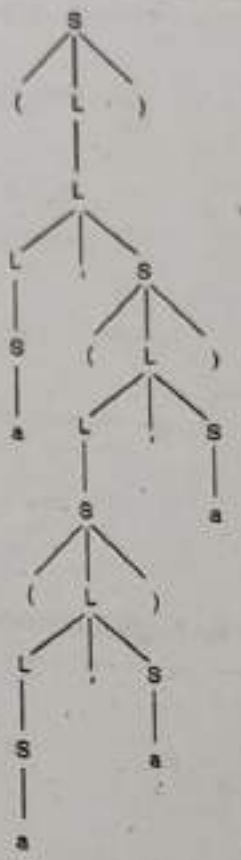


Fig. 5.16.5 : Parse tree for "(a, ((a, a), a))"

➢ **Example 5.16.20**

Consider the following grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

$F \rightarrow num$

Derive the string "id + num * id" and construct a parse tree.

$$
\begin{aligned}
E &= E + T \\
&\Rightarrow E + T * F \\
&\Rightarrow T + T * F \\
&\Rightarrow T + F * F \\
&\Rightarrow T + num * F \\
&\Rightarrow F + num * F \\
&\Rightarrow id + num * F \\
&\Rightarrow id + num * id
\end{aligned}
$$

Scanned by CamScanner

Parse tree

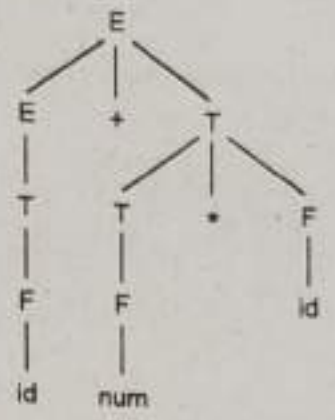


Fig. 5.16.6 : Parse tree for string "(id + num* id)"

5.16.4 Ambiguity

Q. What is ambiguity? Explain the techniques to eliminate the ambiguity with an example. **(10 Marks)**

Q. What do you mean by ambiguity in grammar ? How will you remove ambiguity from a grammar? Use suitable examples. **(10 Marks)**

A grammar is said to be ambiguous if it produces more than one parse tree for same sentence in the grammar. An ambiguous grammar can have more than one leftmost and rightmost derivations. For general parsing, a grammar should be made unambiguous; if grammar is ambiguous then the parser cannot uniquely determine which parse tree is to be select for a given sentence.

➢ **Example 5.16.21**

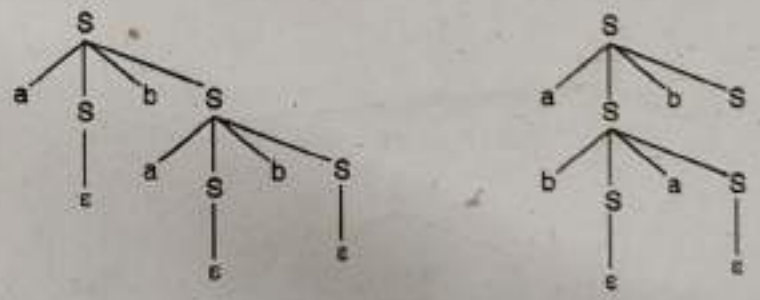
Consider the following grammar

$S \rightarrow a\,S\,b\,S$

$S \rightarrow b\,S\,a\,S$

$S \rightarrow \varepsilon$

For the string "abab", we have two different parse trees as shown in Fig. 5.16.7.



(a) **(b)**

Fig. 5.16.7 : Two different parse trees for "abab"

Scanned by CamScanner

➢ **Example 5.16.22**

Consider the sentence "id-id/id" has two parse trees as shown in Fig. 5.16.8.

$E \rightarrow E\ OPE$

$E \rightarrow (E)$

$E \rightarrow -E$

$E \rightarrow id$

$OP \rightarrow +$

$OP \rightarrow -$

$OP \rightarrow *$

$OP \rightarrow /$

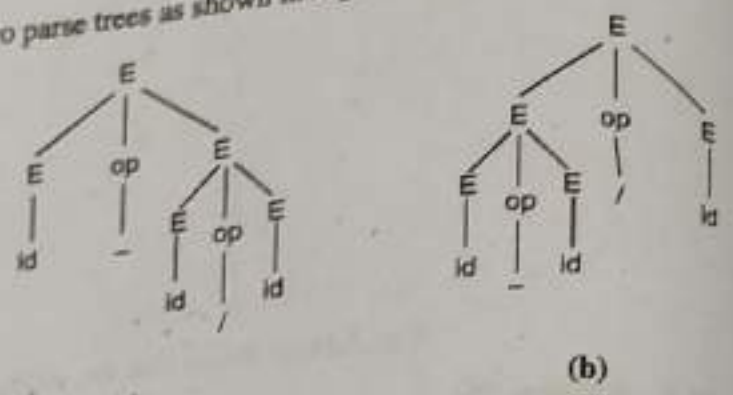


Fig. 5.16.8 : The ambiguous parse tree for "id-id/id"

➢ **Example 5.16.23**

Consider the following grammar that is ambiguous.

stmt → if cond then stmt | if cond then stmt else stmt

The above stated grammar is ambiguous as associating else with if, it. For example the sentence if C1 then E1 if C2 then S1 else S2 can lead to two parse trees which are shown as below :

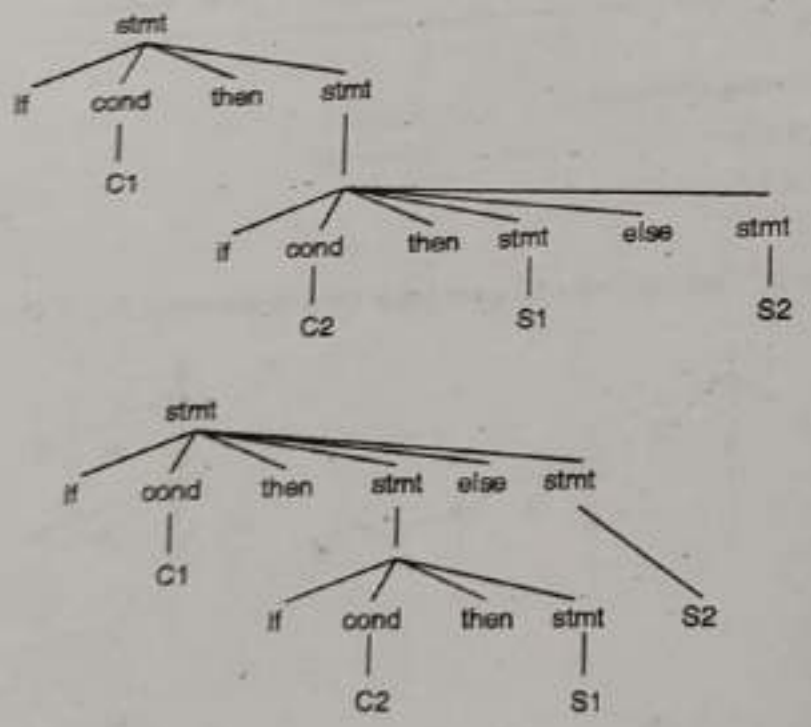


Fig. 5.16.9 : Two parse trees for if-then-else statements

Thus from the above parse trees it can be seen that the above grammar is ambiguous.

Scanned by CamScanner

Eliminating ambiguity

Ambiguity in the above grammar can be eliminated by rewriting the grammar. For example, the if-then-else grammar may be written by associating the else with the closest if. Then the grammar can be written as

stmt → ustat | mstat

mstat → if cond then mstat else mstat

mstat → other_stmt

ustat → if cond then stmt | if cond then mstat else ustat

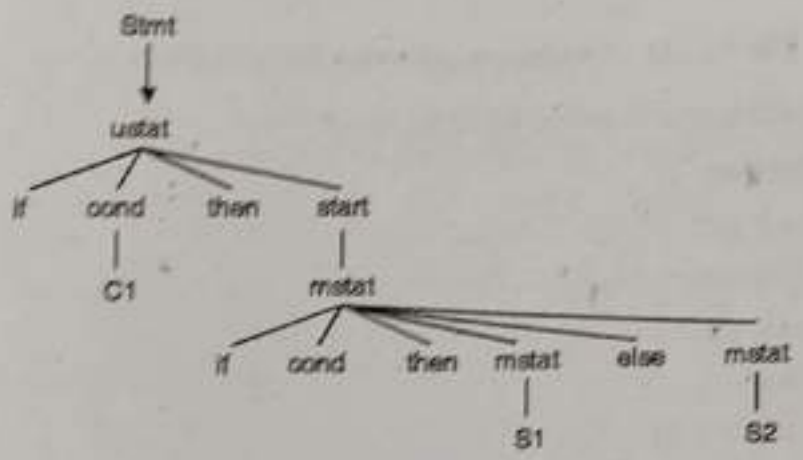


Fig. 5.16.10 : Parse tree for unambiguous if-then-else statements

The grammar is broken in to two pieces matched statements and unmatched statements. Matched statements contain as many if's and else's and unmatched statements contain greater number of if's than else's.

Another technique to resolve ambiguity is to modify the language a bit. Many languages require that a *if* should have a matching *endif*. Thus, the grammar is modified as,

stmt → if condition then stmt else stmt endif

| if condition then stmt endif

The other method to remove ambiguity from an expression grammar, is to postpone the higher priority operation or assign a seperate non-terminal to take care of then

Consider the grammar

$$E \rightarrow E + E \mid E * E \mid id$$

Here the problem of ambiguity is due to the fact that 'E' is the only non-terminal deriving two separate operations with different priorities. Here, '*' has higher priority then '+'.

The grammar after removing the ambiguity i.e. the unambiguous grammar can be written as follows :

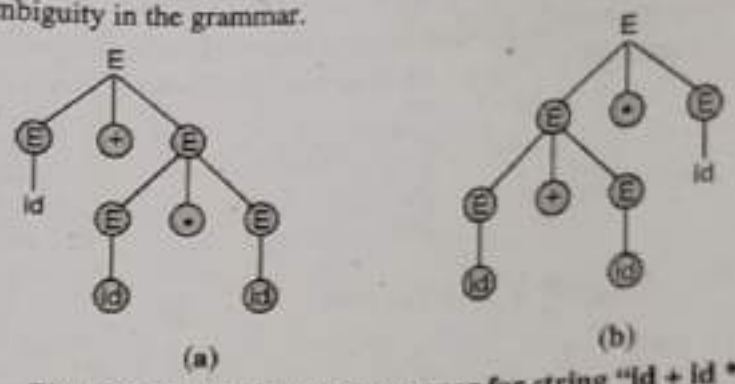
$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow id$$

Scanned by CamScanner

Let us derive the string the same string "id + id * id", the from the above production rule, for which we had detected ambiguity in the grammar.



(a) (b)

Fig. 5.16.11 : Ambiguous grammar for string "id + id * id"

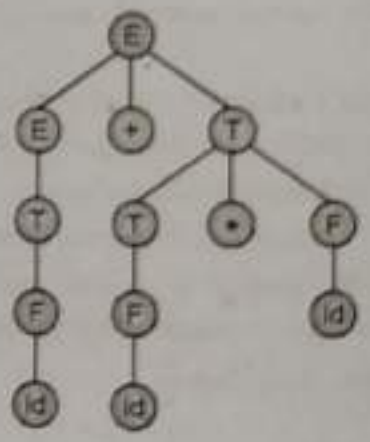
Derivation of unambiguous Grammar for string "id + id * id".

(i) Leftmost Derivation

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow id + T \\
&\Rightarrow id + T * F \\
&\Rightarrow id + F * F \\
&\Rightarrow id + id * F \\
&\Rightarrow id + id * id
\end{aligned}
$$

(ii) Rightmost Derivation

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow E + T * F \\
&\Rightarrow E + T * id \\
&\Rightarrow E + F * id \\
&\Rightarrow E + id * id \\
&\Rightarrow T + id * id \\
&\Rightarrow F + id * id \\
&\Rightarrow id + id * id
\end{aligned}
$$



Derivation tree for "id + id * id"

Fig. 5.16.12 : Unambiguous Grammar

Note : If a grammar has single leftmost and rightmost derivation then it is an unambiguous grammar but if a grammar has a two different type of leftmost derivation then that grammar will be ambiguous grammar for which we can draw two parse trees.

Scanned by CamScanner

5.16.5 Left Recursion

→ (MU - May 2015)

Q. Write short note on : Left recursion. **May 2015, 2.5 Marks**

Q. With example explain the process of elimination of left recursion. **(2.5 Marks)**

A grammar is said to be left recursive if it has a non-terminal, say A, such that there is a derivation A => Aα, for some string α. Presence of left recursion creates difficulties while designing the parsers. Left recursion may be of two types :

1. Immediate left recursion
2. General left recursion

Top-down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed. It means driving the input in one or more steps. Here A is non terminal and α denotes some input string. This expansion of A causes further expansion of A only and due to generation of A, A α, Aαα, Aαaα the input pointer will not be advanced. This causes major problem in top down parsing and therefore elimination of left recursion is a must.

An immediate left recursion happens with a non-terminal A having production rule of the form given below

(i) A -> Aαlβ or it is written as

A -> Aα

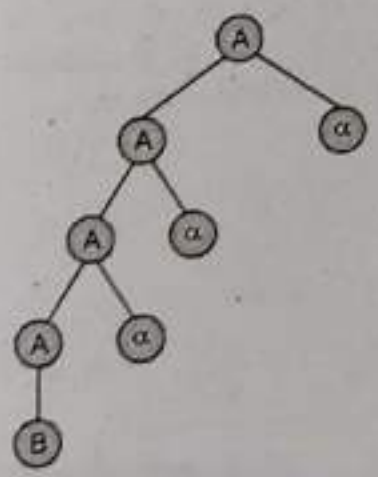
A -> β

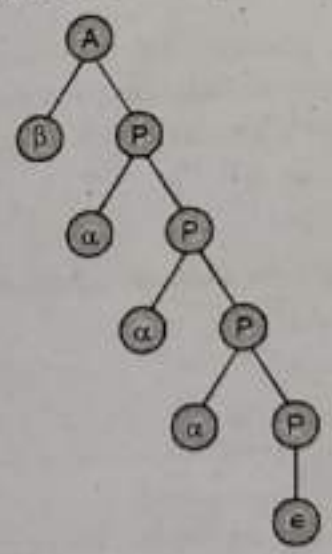
The parser can go into an infinite loop. The immediate left recursion can be eliminated by introducing a new nonterminal symbol, say P thus corresponding grammar without left recursion :

A -> βP

P -> αPlε

We can also verify whether the modified grammar is equivalent to original or not. Take an input string "βaaaa".





The above two parse tree shows that both the grammars are equivalent

No matter how many A-productions there are we can eliminate immediate left recursion from them by the following technique. First we group the A-productions as :

Scanned by CamScanner

$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid A\alpha 3 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \beta 2 \mid \beta 3 \mid \ldots \mid \beta n$.

where no βi begins with an A. Then we replace the A-productions by

$A \rightarrow \beta 1P \mid \beta 2P \mid \beta 3P \mid \ldots \mid \beta nP$

$P \rightarrow \alpha 1P \mid \alpha 2P \mid \alpha 3P \mid \ldots \mid \alpha mP \mid \varepsilon$

The above process removes all immediate left recursion but does not remove recursions involving derivations of two or more steps.

➢ **Example 5.16.24**

Consider the grammar :

$S \rightarrow Aa$

$A \rightarrow Sb \mid c$

Here the grammar does not have immediate left recursion. But has a left recursion because $S \Rightarrow Aa \Rightarrow Sba$

In such cases we set a "hierarchy" among non-terminals and implement the following algorithm.

1. Arrange the non-terminals in some order A1, A2, ...An (setting the hierarchy)
2.

```
for i=1 to n do
{
for j=1 to i-1 do
{
Replace each production of the form Ai -> Ajγ
by the production Ai->δ1γ | δ2γ | ... |δkγ
where Aj -> δ1 | δ2 | δ3 | ..... | δk are all the current Aj positions
}
Eliminate the immediate left recursion among the Ai productions
}
```

For the above grammar, Let the order of non-terminals be S, A. For i=1, the rule $S \rightarrow Aa$ is through, since there is no immediate left recursion. For i=2, $A \rightarrow Sb \mid c$ is modified as, $A \rightarrow Aab \mid c$, which has immediate left recursion and hence is eliminated by modifying the rules as,

$S \rightarrow Aa$

$A \rightarrow Sb|c$

After removing immediate left recursion :

$S \rightarrow Aa$

$A \rightarrow cA'$

$A' \rightarrow abA'|\varepsilon$

➢ **Example 5.16.25**

Consider the following left-recursive grammar for arithmetic expressions.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Scanned by CamScanner

Elimination of immediate left recursion from the rules modifies the grammar as,

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (F) \mid id$$

Example 5.16.26 : Eliminate left recursion present in following grammar (Remove Direct and Indirect recursion both).

$S \longrightarrow Aa \mid b$

$A \longrightarrow Ac \mid Sd \mid \epsilon$

MU - Dec. 2015, May 2016, 5 Marks

Solution :

Algorithm : Indirect left-recursion

Arrange the nonterminals in some order $A_1,\ldots,A_n$.

for (i in 1..n) {

for (j in 1..i-1) {

replace each production of the form $A_i ::= A_j g$ by the

productions $A_i ::= d_1 g \mid d_2 g \mid \ldots \mid d_k g$ where

$A_j ::= d_1 \mid d_2 \mid \ldots \mid d_k$

}

eliminate the immediate left recursion among A_i productions }

For the above grammar, Let the order of nonterminals be S, A. For i=1, the rule $S \rightarrow Aa$ is through, since there is no immediate left recursion. For i=2, $A \rightarrow Sb \mid c$ is modified as, $A \rightarrow Aab \mid c$, which has immediate left recursion and hence is eliminated by modifying the rules as,

$$S \rightarrow Aa$$
$$A \rightarrow Sb \mid c$$

After removing immediate left recursion :

$$S \rightarrow Aa$$
$$A \rightarrow cA'$$
$$A' \rightarrow abA' \mid \varepsilon$$

5.16.6 Left Factoring

(MU - May 2015)

Q. Write short note on : Left Factoring removal technique. **May 2015, 2.5 Marks**

A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*. The basic idea behind left factoring is that when we are not clear which production rule is to be used for the expansion of any non-terminal.

Scanned by CamScanner

For example, if we have the two productions

$S \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

On seeing this we cannot identify which productions are used to expand A whether $\alpha\beta1$ or $\alpha\beta2$ this will lead us to the infinite parse tree. However, we can make the grammar left factored by expanding A to $\alpha A'$. Then, after seeing the input derived fromα, we may expand S' to $\beta1$ or $\beta2$. So the expression changes to

$S \rightarrow \alpha S'$

$S' \rightarrow \beta1|\beta2$

Algorithm :

Construction of Left factoring grammar

Input : Grammar

Output : Left factored grammar

Steps

- For each non-terminal S with two or more alternatives (production rules) with a common non-empty prefix, let say

  $S \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$

  convert it into

  $S \rightarrow \alpha S' \mid \gamma_1 \mid \ldots \mid \gamma_m$

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

Here S' is a new non terminal. Repeat this transformation until no two alternatives for a non terminal have a common prefix.

➢ **Example 5.16.27**

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cdg} \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

5.16.7 Context Free Language (CFL)

It is a language generated by CFG i.e. context free grammar.

L (G) = {w/w CT* and it can be derivable from the start symbol 'S'}

Scanned by CamScanner

Example 5.16.28

The context free grammar is given as

$S \rightarrow aSb \mid ab$

Find the CFL generated by the above grammar.

Let us start listing or generating the strings that we can generate with the above CFG.

Let us start with minimal length string. Let us number the production as,

(1) $S \rightarrow aSb$ (2) $S \rightarrow ab$

(i) from production (2) we can device string "ab" in one step as

$S \Rightarrow ab$

(ii) to start with new derivative, we can have

$S \Rightarrow aSb$ rule (1)

$\Rightarrow aabb$ rule (2)

(iii)

$S \Rightarrow aSb$

$\Rightarrow aaSbb$ rule (1)

$\Rightarrow aaabbb$ rule (2)

(iv)

$S \Rightarrow aSb$ rule (1)

$\Rightarrow aaSbb$ rule (1)

$\Rightarrow aaaSbbb$ rule (1)

$\Rightarrow aaaabbbb$ rule (2)

Thus the language can be listed in the form of set as,

$L = ab, aabb, aaabbb \ldots\}$

i.e. $L = \{a^n b^n \mid n \geq 1\}$

Example 5.16.29

Find the CFL generated by

$S \rightarrow +SS \mid -SS \mid a$

Case (1)

$S \rightarrow +SS$

$\rightarrow +-SSS$

$\rightarrow +-aaa$

Case (2)

$S \rightarrow +SS$

$\rightarrow ++SSS$

$\rightarrow ++-SSSS$

$\rightarrow ++-aaaa$

$L(G)$ = {Postfix expression for a combination of a}

Scanned by CamScanner

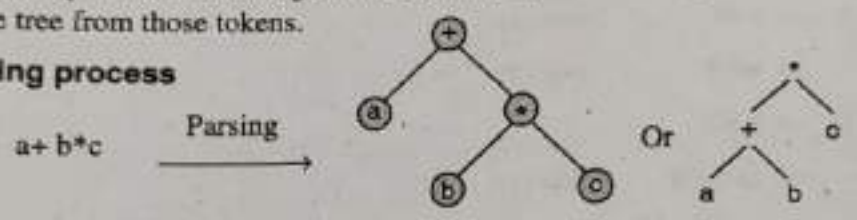
5.17 Parsing

Parsing is the process of analyzing a sequence of tokens in order to determine its grammatical structure with respect to a given formal grammar. It is formally named as syntax analysis. A parser scans an input token stream, from left to right, and group tokens in order to check the syntax of an input string by identifying a derivation by using the production rules of the grammar. We can define a parser formally as a program P, for a context free grammar G, which takes a string W as input and outputs.

1. A parse tree for W if W is a sentence of G i.e. $W \in L(G)$
2. Or an message if W is not a sentence. It may also provide information about the type of error and the location of the error.

Generally, parsers operates in two stages, first identifying the meaningful tokens in the input, and then building a parse tree from those tokens.

Overview of parsing process



The above mentioned example shows the general case of parsing a computer language with two levels of grammar i.e. lexical and syntactic analysis.

The first stage is the token generation, or lexical analysis. The next stage is syntactic parsing or syntax analysis, which is the checking of tokens from an allowable expression.

5.18 Type of Parsers

Different parsing strategies differ in the manner they construct a derivation for the input. There are two basic approaches for parsing. Each corresponds to one of the two possible ways in which a parse tree can be constructed.

(1) Top Down Parsers

It creates the parse tree from the root and expand it till all the leaves are reached. The different type of top down parsers are as

(1) Recursive Descent Parser
(2) LL Parser
(3) Packrat Parser
(4) Unger Parser
(5) Early Parser

In this chapter we will mainly deal with the **recursive descent** and **LL parsers**.

(2) Bottom-up Parsers

A parser starts with the input and attempt to rewrite it to the start symbol, or it creates the parse tree from leaves upwards to the root. There are many different type of bottom-up parsers they are as

(1) Precedence parsing
(2) Bounded Context Parsing
(3) LR Parser
(4) SLR parser
(5) LALR parser
(6) Canonical LR parser
(7) GLR parser
(8) CYK parser

Scanned by CamScanner

In the following chapter we will discuss precedence parsing, LR parser, SLR parser and LALR parsers.

5.18.1 Parsing Table

Parsing table is a tabular illustrations of a pushdown automaton that is built from context free grammars. They are used to describe what action its parser should take when a given input comes while it is in a given state. Parse table is used to recognize each viable prefix (The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes that is, the sequence of input that can be set aside before a parser needs to either perform a reduction or return an error) and determines what needs to be done next.

A parsing table is comprised of a stack and an input buffer. Initially the stack is empty and the symbols are shifted onto the stack one by one from the input buffer with associated states. In each step, the parsing table is referred to decide what action to be taken.

The parsing table consists of two parts :

1. **Action Table :** It defines what to do when a terminal is encountered. Action table pushes the state at the top of the stack and the next symbol (also known as "lookahead" symbol) in the input buffer and tells what action is to be taken.
2. **Goto Table :** It defines what to do when a non-terminal come across. It returns the next state for the parser when a reduction exposes a new state on the top of the stack.

Action table consists of three operations.

Reduction

In reduction when the parser recognizes a sequence of symbols to represent a single non-terminal, and reduces that non-terminal for the sequence. During reduction the parser will pop 2*N entries from the stack and push the recognized non-terminal on its place. There is no state related to this non-terminal, so to determine the next parse state the parser will use the go to table. (N is the length of the sequence, this number is doubled because when each token is pushed and removed from the stack we have two states to consider).

Shift

In shift operation parser shifts the next symbol onto the stack. The token is moved from the input buffer to the stack along with an associated state, also specified in the action table entry, which will be the next parser state.

Accept

When a parser is complete and the sequence of tokens is matched to an accept state in the parse table.

5.19 Top Down Parsing

A top down parser may be considered as an attempt to construct a parse tree starting from the root and create the nodes of the parse tree in pre-order. This means it starts the derivation from the start symbol of the grammar.

Scanned by CamScanner

It is desirable to construct a derivation that produces terminals in the left to right fashion in the sentential forms as the Input tokens are scanned from left to right. This requirement may be fulfilled by constructing a left most derivation.

A top down parser starts constructing the left-most derivation from the start symbol. The next step is to find a suitable production rule in such a way that by using that rule it can move from a leftmost sentential form to its succeeding one.

If the leftmost nonterminal has more than one production rule, a selection may be made depending on whether backtracking is permitted or not. If back-tracking is permitted, parser can make repeated scans of the input. It may try out all the alternates, in any order, till it successfully parser the string i.e. it finds a leftmost derivation or it fails to construct a leftmost derivation. If no back-tracking is permitted, parser has to select the correct alternate at each step of derivation process. The selection of the correct rule is the most important task in such parsers.

Example 5.19.1 : Consider the following CFG grammar which generates a subset of the types used in C language.

(i) P → type list ; (ii) type → int ;

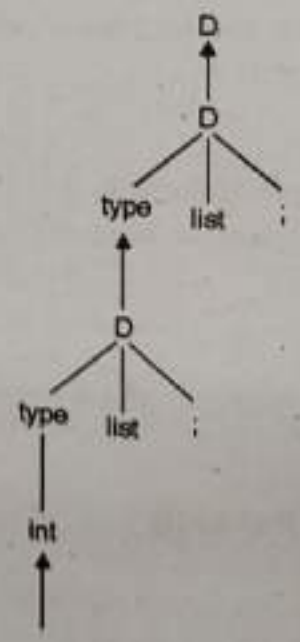
(iii) type → float (iv) list → id, list

(v) list → id

Solution :

Let **int a, b;** be the sentence to be parsed using a top down parser. The parser would start construction of a parse tree by the root labelled with the starting non terminal D and repeatedly selecting one of the production for the non terminal node n and constructing children at n for the right hand side of the production. It then finds the next node at which a subtree is to be constructed.

This is shown in Fig. P. 5.19.1.

Input	Parse Tree
(a) int a, b,	
(b) int a, b,	
(c) int a, b,	

Scanned by CamScanner

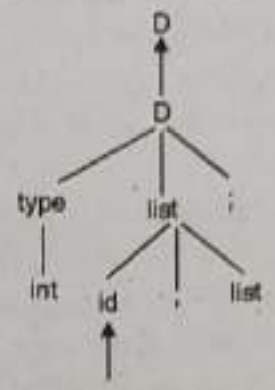
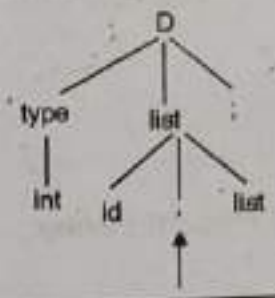
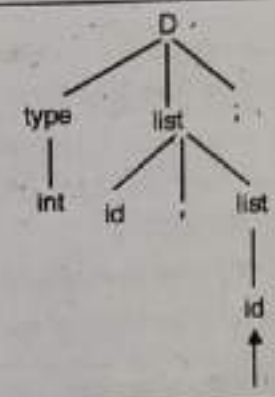
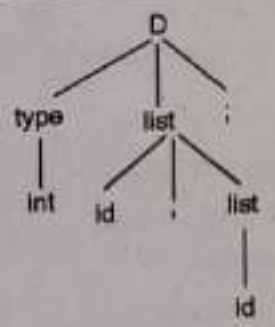
Input	Parse Tree
(d) int a, b,	
(e) int a,b,	
(f) int a, b,	
(g) int a,b,	

Fig. P. 5.19.1 : Top down parsing of a string "int a, b ;" from left to right

The initial situation is shown in Fig. P. 5.19.1(a) the current token being scanned is referred to as the look ahead symbol. The leftmost token of the input string e.g. int, in this case, is the look ahead symbol initially and the parse tree consists of a root labelled with non-terminal D.

Scanned by CamScanner

We apply rule (i) to D in order to derive a string that starts with the look ahead symbols int. So we construct the children of the root labelled with the symbols on the right side of the production. This situation is shown as Fig. P. 5.19.1(b).

Then rule (ii) is applied to type which is the leftmost child as in Fig. P. 5.19.1(c). Input string shows an arrow marking the look ahead symbol in the input and an arrow at the node in the parse tree that is being considered.

At this stage the current token int matches the terminal int at the node being considered in the parse tree, both the arrows are moved and parsing continues.

The next token a in the input becomes the new look ahead symbol and the next child in the parse tree is considered. At this stage we apply rule (iv) to non terminal list as shown in Fig. P. 5.19.1(d). We may note that rule (v) may also be applied here but that may not be the correct choice. The Fig. P. 5.19.1 shows the complete parsing of the input which is parsed successfully.

Important

(1) Prior to top-down parsing, left recursion must be compulsorily removed.

(2) Left factoring should be done, as it helps to remove ambiguity.

5.19.1 Recursive Descent Parser

→ (MU - Dec. 2016)

Q.	Write short note on : Recursive Descent Parser.	Dec. 2016, 5 Marks
Q.	Explain the working of Recursive Descent Parser.	(5 Marks)

A recursive descent parser is a top-down method of syntax analysis which uses a collection of recursive procedures to process the input. A procedure is associated with each non-terminal of the grammar. In recursive descent parsing :

(i) We backtrack for other production rules if a choice of production rule does not work.

(ii) It tries to find the left most derivations and it is a general parsing technique but not widely used.

Example 5.19.2 : Consider the grammar

(1) $E \rightarrow T + E \mid T$ (2) $T \rightarrow int \mid int * T \mid (E)$

Input string is "$int_5 * int_2$"

Solution :

Start with top-level non-terminal E

(i) Using given production rule we get

$E_0 \rightarrow T_1 + E_2$ (using (1))

$T_1 \rightarrow (E_3)$ (using (2))

But It does not match. Input token int_5

(ii) Try $T_1 \rightarrow int$ – (Token matches)

But + after T_1 does not match input token +

(iii) Try another rule

$T_1 \rightarrow int * T_2$ (using rule (2))

Scanned by CamScanner

This will match but + after T_1 will be unmatched.

$$E_0 \rightarrow int * T_2 + E_2$$

The above are the exhausted choices for T, we will backtrack again to have the right solution for input string "$int_5 * int_2$"

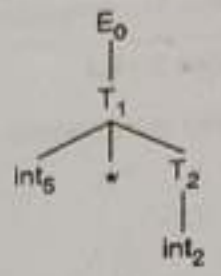
$$E_0 \rightarrow T_1$$

Follow same steps as before for T_1

$$T_1 \rightarrow int * T_2$$

$$T_2 \rightarrow int$$

With the help following parse tree is



Example 5.19.3 : Consider the grammar

$S \rightarrow cAd$

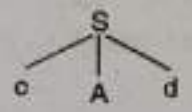
$A \rightarrow ab \mid a$

Input string X = cad. Is parsed using Recursive Descent Parser.

Solution :

To construct parse tree for this string top-down we initially create a tree consisting of a single node labeled S. An input pointer points to C the first symbol of X.

Step 1 : Following the first rule $S \rightarrow cAd$ to parse S



Step 2 : The next term in line A is parsed using first rule, $A \rightarrow ab$, but twins out INCORRECT, parser backtracks.

Step 3 : Next rule to parse A is taken $A \rightarrow a$, twins out CORRECT, string parsed completely power stops.

Note : A left recursive grammar can cause a recursive descent parser, even the one with backtracking to go into an infinite loop. An attempt to expand A, we may find ourselves expanding A, without having consumed any input.

Recursive descent parser is a top down parser built from a set of mutually recursive procedures where each such procedure usually implements one of the production rules of the grammar.

Scanned by CamScanner

Recursive descent with backup or back tracking is a technique that determines which production to use by trying each production in turn. Although predictive parsers are widely used, programmers often prefer to create LR or LALR parsers via parser generators without transforming the grammar into LL (k)* form. Recursive descent parsers are particularly easy to implement in functional languages.

➢ **Example 5.19.4**

Consider is a grammar for recursive descent parser

Program → statement program/statement
Statement → assignment
Assignment → 1D EQUAL expr

➢ **Example 5.19.5**

Consider the following grammar as

Expression → expression op term | term
Term → term op | factor | factor
Factor → expression | number
Op → + | −
op1 → *

Now we present a procedure that recognizes a factor. In this procedure there is a 'token' variable that keeps the current text token in the input and 'match' is an another procedure that match current next token with its parameter.

Procedure factor

```
begin
    case token
    (match) ( ) ;
    expression ;
    match ( ) ;
    number : match (number) ;
    else Error
    end case
end
procedure match (expected token) ;
begin
    If token = expected token then
    Get next token
    else
    Error
    end of
end
```

Scanned by CamScanner

Example 5.19.6

Recursive descent calculator for simple integer arithmetic. For this purpose we consider the following grammar as :

```
# include <stdio.h>
# include <conio.h>
char token ;
int exp (void) ;
int term (void) ;
void factor (void) ;
void error void
{
    fprintf (stderr, "error") ;
    exit (1) ;
}
void match (void)
{
    if (token = = expected Token)
    token = getchar ( );
    else error ( );
}
int exp (void)
{
    int temp = term ( ) ; -
    while ( (token = = '+' || token = = '–') ) ;
    switch (token)
    {
      case '+' : match ('+') ;
      temp + = term ( ) ;
      break ;
      case '–' : match ('–')
      temp – = temp ( ) ;
      break ;
    }
    return temp ;
}
int term (void)
{
    int temp = factor ( ) ;
```

Scanned by CamScanner

```
        while (token = = '*')
        {
          match ('*')
          temp * = factor ( ) ;
        }
        return temp ;
    }
    int factor (void)
    {
        int temp ;
        if (token = = '(')
        {
          match (' (')
          temp = exp ( ) ;
          match (')') ;
        }
        else if (is digit (token))
        {
          ungete (token, stdin) ;
          scanf ("%d", & temp) ;
          token = getchar ( ) ;
        }
          else error
          return temp ;
    }
    void main ( )
    {
        int result ;
        token = getchar ( );
        result = exp ( );
        if (token = = '\n')
        printf ("result = %id", result) ;
        else
        error ( );
        return 0;
    }
```

Recursive descent parser is a simple and general parsing strategy. For this left recursive must be eliminated first. This is unpopular because of backtracking. In practice recursive descent parser is easy to construct if the programming language used for coding permits recursion.

Scanned by CamScanner

5.19.2 Predictive Parsers

A **predictive parser** is a recursive descent parser with no backup or backtracking.

It is top-down parser that needs no backtracking. At each step, the choice of rule to be expanded is made upon the next terminal symbol.

Suppose

$$A \rightarrow A1 \mid A2 \mid \ldots \mid An$$

If the non-terminal to be expanded next is 'A', then the choice of the rule is made on the basis of the current input symbol 'a' only.

Algorithm for Predictive Parser.

1. Make a transition diagram (like DFA/NFA) for every rule of the grammar.
2. Optimize the DFA by reducing the number of states, yielding the final transition diagram.
3. To parse a string, simulate the string on the transition diagram.
4. If after consuming the input the transition diagram reaches an accept state, it is parsed.

Each of these steps are expanded below :

➢ **Example 5.19.7**

Consider the following grammar :

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

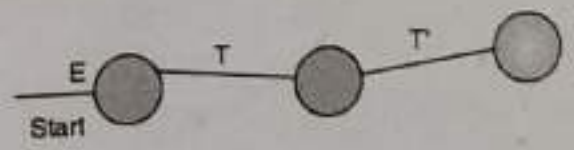
After removing left-recursion, left-factoring

The rules are as follows :

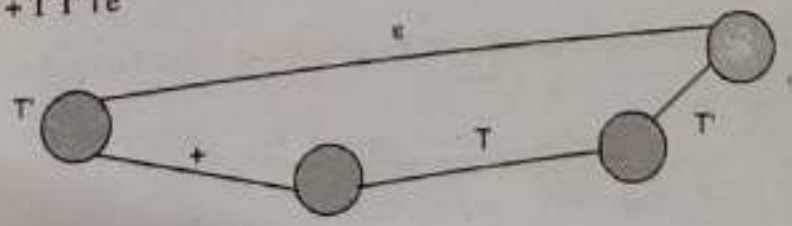
- $E \rightarrow T\,T'$
- $T' \rightarrow + T\,T' \mid \varepsilon$
- $T \rightarrow F\,T''$
- $T'' \rightarrow * F\,T'' \mid \varepsilon$
- $F \rightarrow (E) \mid id$

Step 1 : Make DFA-like transition diagrams for each rule :

(a) $E \rightarrow T\,T'$

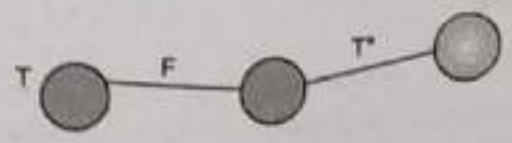


(b) $T' \rightarrow + T\,T' \mid e$

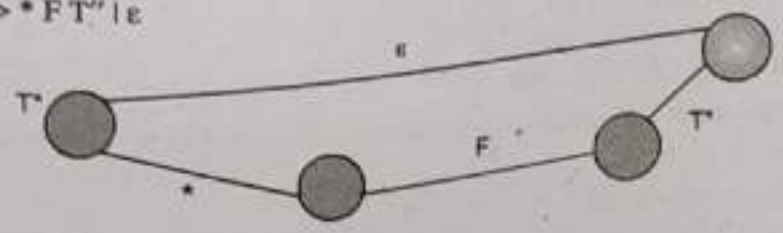


Scanned by CamScanner

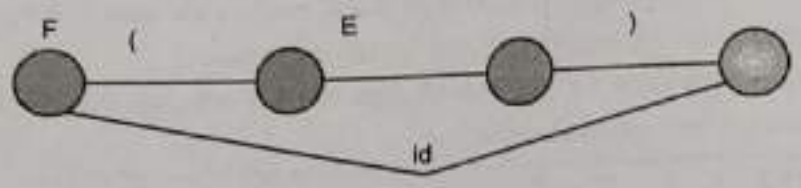
(c) $T' \rightarrow F\,T''$



(d) $T'' \rightarrow * F\,T'' \mid \varepsilon$

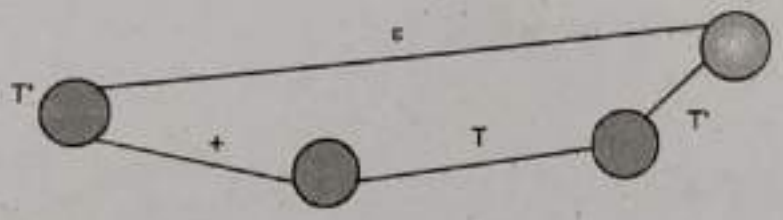


(e) $F \rightarrow (E) \mid id$

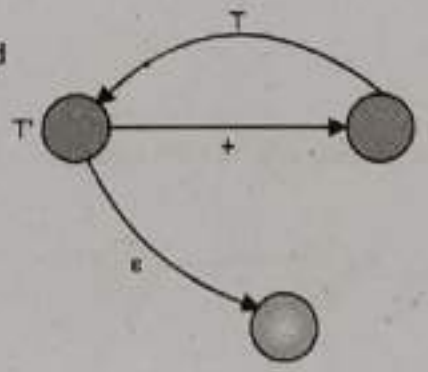


Step 2 : Optimization : The purpose of optimization is to reduce the number of states, let's optimize the states.

(f) Consider the DFA for $T' \rightarrow + T\,T' \mid \varepsilon$ shown below :



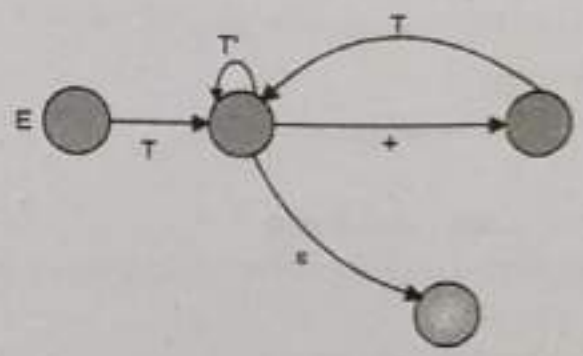
(g) It can be optimized



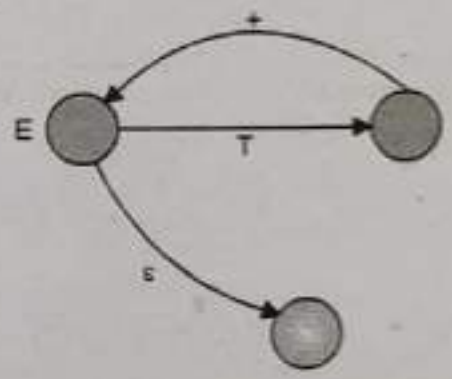
(h) Which can be optimized further by combining with DFA for $E \rightarrow T\,T'$ shown below

Scanned by CamScanner

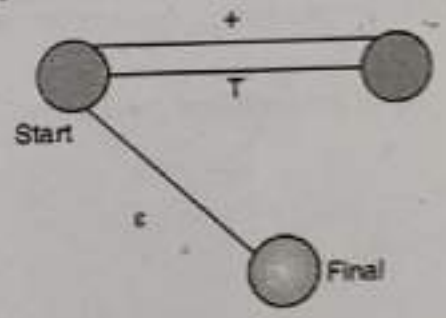
(i) It yields

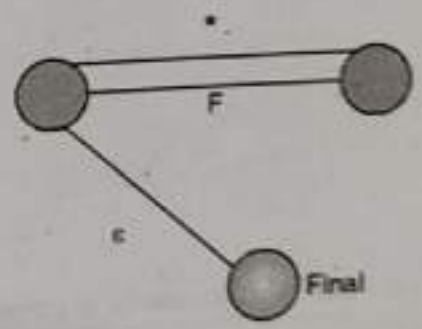


(j) We can even further optimize it to produce



(k) Similarly, we optimize other structures to produce the following DFA's :





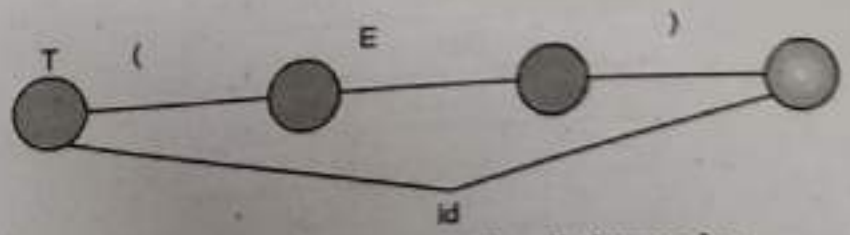


Fig. 5.19.1 : Different steps of predictive parsing

Scanned by CamScanner

After optimization is over we start simulation over these "DFA's"

Step 3 : Simulation on the input string :

Following steps list out the simulation procedure briefly.

- Start from the start state
- If a terminal comes consume it, move to next state
- If a non-terminal comes go to the state of the "DFA" of the non-terminal and return on reaching the final state
- Return to the original "DFA" and continue parsing.
- If one completion (reading input string completely), you reach a final state, and string is successfully parsed.

Example 5.19.8 : Modify the given grammar and construct a predictive parser table explaining each step

$E \rightarrow E + T \mid T$

$T \rightarrow T * V \mid V$

$V \rightarrow id$ **(10 Marks)**

Solution :

For construction of the Predictive parser we have to follow the following steps :

Computation of FIRST and FOLLOW function. Construct the predictive parser table with the help of the FIRST and follow functions.

Input the parse string with the help of the predictive parse table.

The above given grammar is-

$E \rightarrow E + T \mid T$

$T \rightarrow T * V \mid V$

$V \rightarrow id$

After modifying the grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow VT'$

$T' \rightarrow *VT' \mid \varepsilon$

$V \rightarrow id$

Step -1 : Find the first and follow functions for the above grammar

Symbols	FIRST	FOLLOW
E	{id}	{$}
E'	{+, ε}	{$}
T	{id}	{+, $}
T'	{*, ε}	{+, $}
V	{id}	{+, *, id, $}

Scanned by CamScanner

Parse table for the above given grammar is :

	+	*	id	$
E			E → TE′	
E′	E → +TE′			E′ → ε
T			T – VT′	
T′	T′ → ε	T′ → *VT′		T′ → ε
V			V → id	

Example 5.19.9 : Design a predictive parser for the given grammar. Mention all the steps

E → TQ
T → FR
Q → +TQ | –TQ | E
R → *ER | /ER | E
F → (E) | id

MU - May 2014, 10 Marks

Solution :

First remove the left recursion from the given grammar so that given production rules.

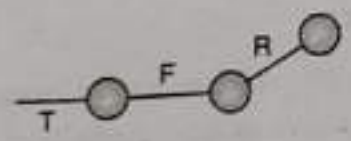
E → TQ
T → FR
Q → +TQ | –TQ | E
R → *ER | /ER | E
F → (E) | id

does not has left recursion and left factoring.

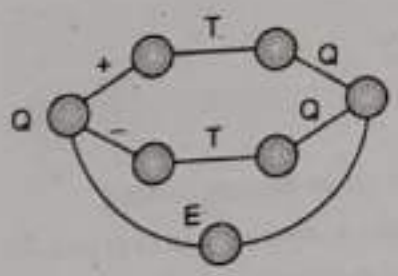
Step 1 : make DFA – like transition diagram for each rule.

(a) E → TQ

(b) T → FR

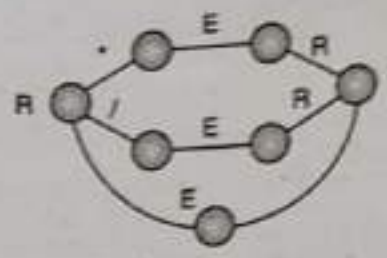


(c) Q → +TQ | –TQ | E

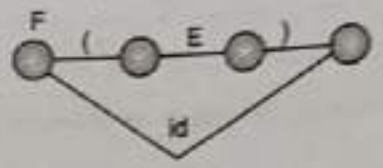


Scanned by CamScanner

(d) $R \rightarrow * ER \mid / ER \mid E$

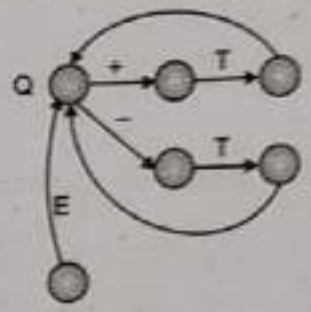


(e) $F \rightarrow (E) \mid id$

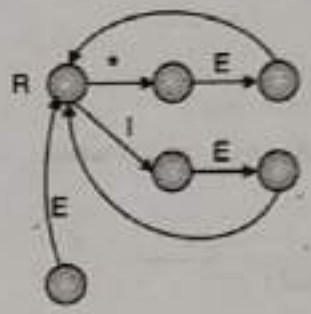


Step 2 : The purpose of optimization is to reduce the number of states.

(f)



(g)



Disadvantages of predictive parser

- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.
- Doing optimization may not be as simple as the complexity of grammar grows.
- To remove this recursion, we use LL-parser, which uses a table for lookup.

5.19.3 LL-Parser

It requires :

- No backtracking.
- It is efficient in comparison to recursive descent parsing.
- It needs a special form of grammars known as (LL(1) grammar).
- Non recursive (Table driven) predictive parser is also known as **LL (1) parser**.

Scanned by CamScanner

An LL parser is a top-down parser for a subset of the context-free grammar. It parses the input from left to right, and constructs a left most derivation of the sentence. The class of grammar which are parsable in this way are known as the **LL grammar**.

The LL parser stands for

- "L" – left to right scan of input
- "L" – leftmost derivation

An LL parser is called an LL (k) parser if uses

- "k" – It uses K token of look ahead when parsing a sentence.

If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called LL (k) grammar. Of these grammar, LL (1) grammar, are very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions. Predictive parsing is possible only for the class of LL (k) grammar, which are the context-free-grammar for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. A predictive parser runs in linear time.

LL parser has an input buffer, a stack on which it keeps symbols from the grammar, a parsing table which tells it what grammar rule to use given the symbols on top of its stack and its input tape. It also has an output stream as shown in Fig. 5.19.2.

An LL parser is called LL (*) parser if it is not restricted to a finite k tokens of lookahead. The parser applies the rule found in the table by matching the top most symbol on the stack (row) with the current symbol in the input stream (column).

When the parser starts, the stack, already contains two symbols : [S, $] where '$' is a special terminal to indicate the bottom of the stack and the end of the input stream and 'S' is the start symbol.

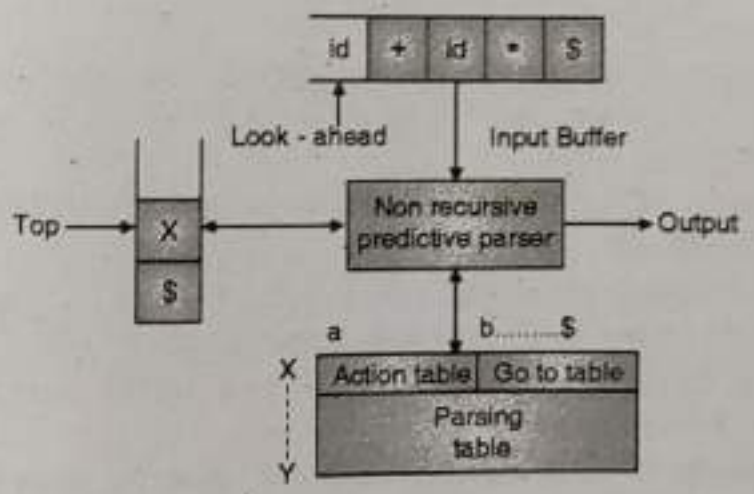


Fig. 5.19.2 : Basic block diagram of a Non Recursive Predictive Parser

Example 5.19.10

To explain its working we will use the following grammar :

1. $E \rightarrow F$
2. $E \rightarrow (E + F + S)$
3. $F \rightarrow 0$
4. $S \rightarrow 1$

Scanned by CamScanner

and parse the following input :

(0 + 0 + 1)

The parsing table for this grammar looks as follows :

Input Symbol

Stack Top Symbol

	(	)	0	1	+	$
E	2	–	1		–	–
F	–	–	3		–	–
S				4		

Instructs parser to apply rule

Fig. 5.19.3 : The Rule given by the Parser to apply on input String

(Note that there is also a column for the special terminal, represented here as $, that is used to indicate the end of the input stream.)

Parsing Procedure

Parsing Technique

1. The parser reads the next-available symbol from the input stream, and the top-most symbol from the stack.
2. If the input token and the top symbol of the stack match, the parser removes the both symbol and leaves behind the unmatched symbol.

Parsing Procedure for String "(0+0+0)"

- Parser reads the input symbol '('and the stack-top symbol 'E'. This matching is done based on the column given by '('and the row headed by the stack-top symbol 'E'; this cell has the '2', which indicates the parser to apply rule 2.
- The Parser has to rewrite 'E' to '(E+F+S)' on the stack and write the rule 2 to the output. The stack then becomes **[(, E, +, F ,+ ,S ,), $].**
- The '(' from the input stream did not match the top-most symbol, 'E', from the stack, it was not removed, and remains the next input token for the following step.
- The parser removes the '(' from its input stream and from its stack, since they match. The new stack will be **[E, +, F ,+ ,S ,), $].**
- **Now the parser has the '0' on its input stream and 'E' as its top of stack it will apply rule 1 E→F. The stack becomes [F, +, F ,+ ,S ,), $].**
- **The parser now has an 0 on its input stream** and an 'F' as its top of stack it will apply rule 3from the production rule F→0. The stack becomes **[0, +, F ,+ ,S ,), $].**
- In the next two steps the parser reads the '0' and '+' from the input string and, since they match the removes them from the stack. This result in stack as: **[F, +, S,), $].**
- **In the next four steps the parser will replace 'F' on the stack by '0' by rule number 3 and 'S' on the stack by '1' by rule number 4 to the output stream and remove '0', '1' and ')'** from both the stack and the input stream. The parser thus ends with '$' on both its stack and its input stream.

Scanned by CamScanner

- **At last the parser will report that it has accepted the input string and writes the rule number to the output stream: [2, 1, 3, 3, 4].**
- This is indeed a list of rules for a leftmost derivation of the input string, which is :
- $E \rightarrow (E+F+S) \rightarrow (F+F+S) \rightarrow (0+F+S) \rightarrow (0+0+S) \rightarrow (0+0+1)$

Remarks

Three type of actions are performed by the parser depending whether the top of the stack is a non-terminal, a terminal or a special symbol $:

- If there exists a non-terminal at the top of the stack then the rule number is written to the output stream else it reports an error and stop.
- If there exists a terminal at the top of the stack then it compares it to the symbol on the input stream and if they are equal they are both removed else it reports an error and stops.
- If there exists a $ on top of the stack and on the input stream then the parser reports that it has successfully parsed the input and accepts the given string. These steps are repeated until the parser stops.

A table driven parser has an input buffer, a stack, a parsing table, and an output stream. LL (1) parser consist of the following.

The output from the parser may be the production rule used by the parser. The working of the parser is described in a tabular term as in Fig. 5.19.3.

Input buffer

The string to be parsed. We will assume that its end is marked with a special symbol $.

Output

A production rule representing a step of the derivation sequence of the string in the input buffer.

Stack

- It has the grammar symbols.
- Bottom of the stack has special end marker symbol $.
- Initially the stack has $ and start symbol S.
- When there is only $ left in the stack, it indicates that parsing is completed.

Parsing table

- Two dimensional matrix M [A, a]
- Each row contains a non-terminal symbol and column has a terminal symbol or the special symbol $.
- Each entry in the parsing table holds a production rule.

5.19.3(A) LL (1) Parser - Parser Actions

- The parser action is determined by the symbol at the top of the stack (say X) and the current symbol in the string (say a).

Scanned by CamScanner

- There are four possible parser actions.
  1. If X and a are $ → parser halts (successful completion)
  2. If X and a are the same terminal symbol (different from $)
     → parser pops X from the stack, and moves the next symbol in the input buffer.
  3. If X is a non-terminal
     → parser looks at the entries done in the parsing table. If M[X, a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack.
     → $X \rightarrow Y_1 Y_2 \ldots Y_k$ is given by the parser to the output to represent a step of the derivation.
  4. None of the above action takes place then there is an error
     - Parsing table with all blank entries represent errors.
     - If X is a terminal symbol different from a, this is also an error.

Top of stack	Lookahead symbol	Parser action
$	$	Parsing successful halts
t	T	pop t, advance lookahead pointer continue
t	r	Error
X	t	Consult table entry [X, t] if Table [X, t] = X → ABC pop ; push C, Push B ; Push A; else give error message

Fig. 5.19.4 : Working of a Non Recursive Predictive Parser

➢ **Example 5.19.11**

Consider the following grammar containing six production rules from R1 to R6.

D → type list ; R1
list → id tlist R2
t list → ,id tlist R3
t list → ε R4
type → int R5
type → float R6

The parsing table for this grammar is shown as :

	id	,	;	int	float	$
D	e	e	e	R1	R1	e
list	R2	e	e	e	e	e
t list	e	R3	R4	e	e	e
type	e	e	e	R5	R6	e

Fig. 5.19.5 : Parsing table for Example 5.19.11

Scanned by CamScanner

The working of the parser for the input a, b is shown in Fig. 5.19.6.

Stack	Remaining input	Output
$ D	int a, b; $	D → type list ;
$; list type	int a, b; $	R1
$; list int	int a, b ; $	R5
$; list	a, b; $	Match
$; list id	a, b; $	R2
$; t list	, b; $	Match
$; t list id,	, b; $	R3
$; t list id	b; $	Match
$; t list	; $	Match
$;	; $	R4
$	$	match and the parser halts with a successful parsing

Fig. 5.19.6 : Sequence of moves made by the parser on input int a, b;

Example 5.19.12

Consider a LL (1) Parsing Table for the given grammar

S → aBa

B → bB | ε

	A	B	$
S	S → aBa		
B	B → ε	B → bB	

Fig. 5.19.7: Parsing table for a given grammar in Example 5.19.12

Stack	Input	Output
$S	abba$	S → aBa
$aBa	abba$	
$aB	bba$	B → bB
$aBb	bba$	
$aB	ba$	B → bB
$aBb	ba$	
$aB	a$	B → ε
$a	a$	
$	$	Accept, successful completion

Fig. 5.19.8 : Sequence of moves made by the parser for input "abba".

Scanned by CamScanner

Outputs : $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation (left-most) : $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$

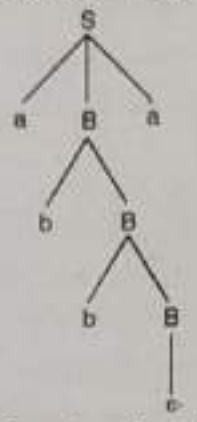


Fig. 5.19.9 : Parse tree for string "abba"

➢ **Example 5.19.13**

Construct a LL (1) Parsing table for the given grammar

$E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$ $\quad T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E′		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T′		$T \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Fig. 5.19.10 : LL (1) Parsing table

Stack	Input	Output
$E	id+id$	$E \rightarrow TE'$
$E′ T	id+id$	$T \rightarrow FT'$
$E′ T′F	id+id$	$F \rightarrow id$
$ E′ T′ id	id+id$	
$ E′ T′	+id$	$T' \rightarrow \varepsilon$
$ E′	+id$	$E' \rightarrow +TE'$
$ E′ T+	+id$	
$ E′ T	id$	$T \rightarrow FT'$
$ E′ T′ F	id$	$F \rightarrow id$
$ E′ T′ id	Id$	
$ E′ T′	$	$T' \rightarrow \varepsilon$
$ E′	$	$E' \rightarrow \varepsilon$
$	$	accept

Fig. 5.19.11 : Sequence of moves made by the parser for input "id + id"

Scanned by CamScanner

Algorithm : Non recursive predictive parsing.

Input : A string W and a parsing table M for grammar G.

Output : If W is in L (G), a leftmost derivation of W, otherwise, an error indication.

Method

```
While stack is not empty do
        Let X be the top stack symbol on the stack
        Let U be the next input symbol
        if X ε T then
                if X = U then
                pop X from the stack and advance the input
                else
                        parsing error
                else  /* X is a non terminal */
                    if  M [X, U] = A → y1, y2 ... yk then
                        pop X from the stack ;
                        push Yk, Yk-1,..., Y1 onto the stack;
                        with Y1 on top ;
                        output the production X → Y1 Y2 ... Yk
                    else
                        parsing error
```

Fig. 5.19.12 : Non recursive predicting parser program

Example 5.19.14

Consider a grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non Terminal	Input Symbol					
	Id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Fig. 5.19.13 : Parsing table

Scanned by CamScanner

With input id + id * id the predictive parsers makes the following sequence of input

Stack	Input Action	Output
$ E	id + id * id $	E → TE′
$ E′ T	id + id * id $	T → FT′
$ E′ T′ F	id + id * id $	F → id
$ E′ T′ id	id + id * id $	match
$ E′ T′	+ id * id $	T → ε
$ E′	+ id * id $	E′ → + TE′
$ E′ T +	+ id * id $	match
$ E′ T	id* id $	T → FT′
$ E′ T′ F	id* id $	F → id
$ E′ T′ id	id* id $	match
$ E′ T′	* id $	T′ → * FT′
$ E′ T′ F *	* id $	match
$ E′ T′ F	id $	F → id
$ E′ T′ id	id $	match
$ E′ T′	$	T′ → ε
$ E′	$	E′ → ε
$	$	Accept

Fig. 5.19.14 : Moves made by predictive parser on input "id + id * id"

Example 5.19.15 : Test whether the grammar is LL (1) or not, and construct a predictive parsing table for it.

S → AaAb / BbBa

A → ε

B → ε

(10 Marks)

Solution : Consider the grammar.

S → AaAb / BbBa

A → ε

B → ε

Now we will compute first and follow sets of given grammar.

First (S) → {a, b} follow (S) → {$}

First (A) → {ε} follow (A) → {a, b}

First (B) → {∈} follow (B) → {a, b}

Scanned by CamScanner

The predictive parsing table is

	a	b	$
S	S → Aa Ab	S → Bb Ba	
A	A → ∈	A → ∈	
B	B → ∈	B → ∈	

Now consider the string "ab" for passing.

Stack	Input	Action
$ s	ab $	S → Aa Ab
$ b Aa A	ab $	A → ε
$ b Aa	ab $	
$ bA	b $	A → ε
$ b	b $	
$	$	Accept

This shows that the given grammar is LL (1).

Example 5.19.16 : Test whether following grammar is LL (1) or not. Construct LL (1) Parse table.

S → AB | g Da
A → ab | c
B → dC
C → gC | g
D → fD | g

(10 Marks)

Solution :

S′ → S$
S → AB | g D a
A → ab I c
B → dC
C → gC I g
D → fD I g

For this we have to find the FIRST and FOLLOW sets. The FIRST and FOLLOW sets give the information about the next characters expected in the grammar.

FIRST (S) = {a, c, g}
FIRST (A) = {a, c}
FIRST (B) = { d }
FIRST (C) = { g }
FIRST (D) = {f, g}
FOLLOW (S) = { $ }

Scanned by CamScanner

FOLLOW (A) = { d }
FOLLOW (B) = { $ }
FOLLOW (C) = { $ }
FOLLOW (D) = { a }

Table P. 5.19.16 : LL(1) Parse table

	A	b	C	d	f	g
S	$S' \to S\$$		$S' \to S\$$			$S' \to S\$$
A	$A \to ab$		$A \to C$			
B				$B \to dC$		
C	$C \to gc$					$C \to g$
D					$D \to fD$	$D \to g$

The above given grammar is LL (1) as their is no multiple entry in one column.

Example 5.19.16(A) : Construct LL(1) parsing table

$S \to aBDh$

$B \to Cc$

$C \to bC \mid \varepsilon$

$D \to EF$

$E \to g \mid \varepsilon$

$F \to f \mid \varepsilon$

Check whether the string 'acbgh' is valid or not. **MU - Dec. 2017, 10 Marks**

Solution :

$S' \to S\$$

$S \to aBDh$

$B \to cC$

$C \to bC \mid \varepsilon$

$D \to EF$

$E \to g \mid \varepsilon$

$F \to f \mid \varepsilon$

FIRST (S)	= {a}	FOLLOW(S)	= {$}
FIRST (B)	= {c}	FOLLOW(B)	= {g, f}
FIRST (C)	= {b, ε}	FOLLOW(C)	= {$}
FIRST (D)	= {g, f}	FOLLOW(D)	= {f, h}
FIRST (E)	= {g, ε}	FOLLOW(E)	= {$}
FIRST (F)	= {f, ε}	FOLLOW(F)	= {$}
FIRST (S')	= {a}		

Scanned by CamScanner

LL(1) Passing Table

	a	b	c	g	f	h	$	ε
S'	S' → S$							
S	S → aBDh							
B			B → cC					
C		c → bC						C → ε
D				D → EF	D → EF			
E				E → g				E → ε
F					F → f			F → ε

Parse the string "acbgh"

S → aBDh
S → acCDh (B → cC)
S → acbCDh (C → bC)
S → acbεDh (C → ε)
S → acbEFh (D → EF)
S → acbgFh (E → g)
S → acbgεh (F → ε)
S → **acbgh**

The above string "acbgh" is valid

5.19.3(B) Construction of Predictive Parser Table using First and Follow

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions are known as :

(i) FIRST
(ii) FOLLOW

These two functions allow us to fill the entries in the predictive parsing table for G.

FIRST (α) is set of the terminal symbols which occurs as first symbols in strings derived from α where α is any string of the grammar symbols.

If α derives to ε, then ε is also in First (α)

FOLLOW (A) is the set of the terminals which occur immediately after (follow) the non terminal A in the strings derived from the starting symbol.

- a terminal a is in FOLLOW (A)

$$\text{if } S \overset{*}{\Rightarrow} \alpha A a B$$

- $ is in FOLLOW (A)

$$\text{if } S \overset{*}{\Rightarrow} \alpha A$$

Scanned by CamScanner

5.19.3(C) First Sets and Follow Sets

→ (MU - Dec. 2014)

Dec. 2014, 3 Marks

Q. Explain First and Follow Sets ?

First Sets

FIRST sets are the right-hand sides of each of the grammar's productions. The idea is that for a sequence α of symbols, FIRST(α) is the set of terminals that begin the strings derivable from α, and also, if α can derive ε, then ε is in FIRST(α). By allowing ε to be in FIRST(α), we can avoid defining the notion of nullable. More formally : FIRST(α) = {t | (t is a terminal and α =>* tβ) or (t = εand α =>*ε)}

To define FIRST(α) for arbitrary α, we start by defining FIRST(X), for a single symbol X (a terminal, a non-terminal, or ε):

- X is a terminal: FIRST(X) = X
- X is ε: FIRST(X) = ε
- X is a nonterminal. In this case, we must look at all grammar productions with X on the left, i.e., productions of the form : $X \rightarrow Y_1Y_2Y_3 \ldots Y_k$
  where each Y_k is a single terminal or nonterminal (or there is just one Y, and it is ε). For each such production, we perform the following actions :
  - Put FIRST(Y1) – {ε} into FIRST(X).
  - If ε is in FIRST(Y1), then put FIRST(Y2) – {ε} into FIRST(X).
  - If ε is in FIRST(Y2), then put FIRST(Y3) – {ε} into FIRST(X).
  - . . .
  - If e is in FIRST(Yi) for 1 <= i<=k (all production right- hand sides) then put ε into FIRST(X).

Consider the Context Free Grammar in Fig. 5.19.15.

Terminals = {a, b, c, d,} Non terminals = {S, A, B, C, D}
(0) S' → S$ (1) S → A (2) A → BC | DBC (3) B → Bb | ε (4) C → c | ε (5) D → a | d Start symbol = S

Fig. 5.19.15 : A CFG, with $ as end of file marker

In this grammar, the set of all strings derivable from A are {aBC, b, c, dBC} and hence first (A) = {a, b, c, d}. The first sets of all of the non terminals in Fig. 5.19.15 can be found in Fig. 5.19.16.

FIRST(S) = {a, b, c, d, ε}
FIRST(A) = { a, b, c, d, ε}
FIRST(B) = {b, ε }
FIRST(C) = {c, ε}
FIRST(D) = {a, d}

Fig. 5.19.16 : The first sets for all the non-terminals for the CFG from Fig. 5.19.15

Scanned by CamScanner

To derive the first set to a string of terminals and non-terminals. Consider a partial derivation, such as

$$S \Rightarrow A \Rightarrow BC$$
$$S \Rightarrow A \Rightarrow DBC$$

There are four ways to complete this derivation.

Either $BC \Rightarrow C \Rightarrow c$ or

$BC \Rightarrow B \Rightarrow Bb \Rightarrow b$

DBC=>aBC=>a

DBC=>dBC=>d

Thus the set of all non-terminals that could created from the partial derivation of BC is {b, c} and from DBC is {a, d}. The first (γ) is the set of all terminals that can appear at the beginning of the string. If a string of non-terminals can derive empty string, the ε will also be in first (γ).

Follow Sets

FOLLOW sets are only defined for single non-terminals. The definition is as follows :

For a non-terminal A, FOLLOW (A) is the set of terminals that can appear immediately to the right of A in some partial derivation; i.e., terminal t is in FOLLOW(A) if:

S => +. . . A t . . .

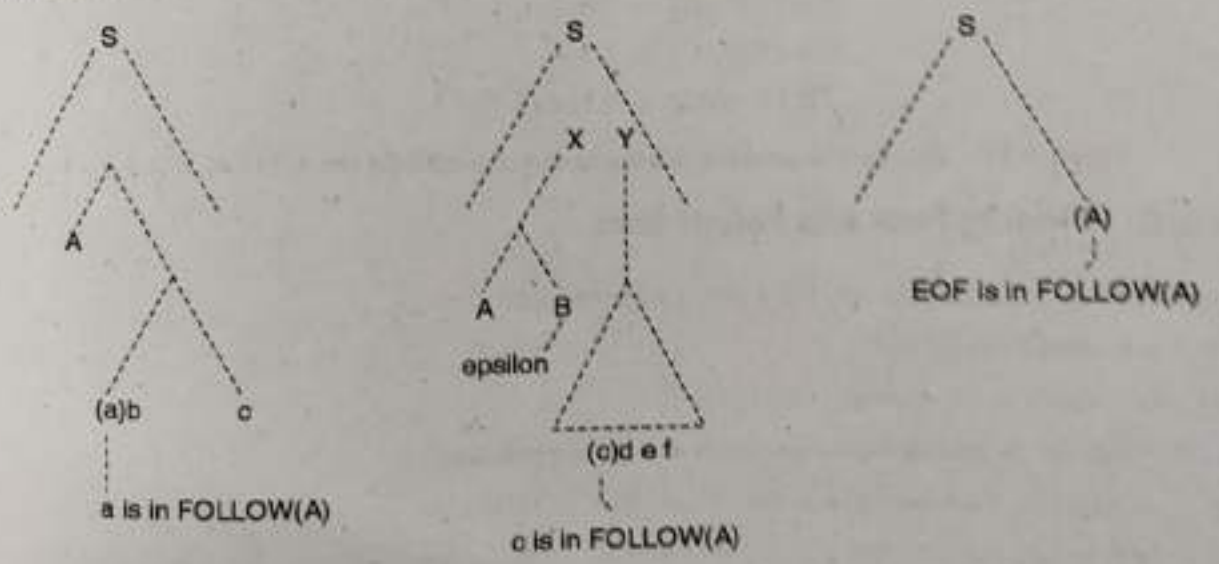
where t is a terminal

Furthermore, if A can be the rightmost symbol in a derivation, then EOF is in FOLLOW (A). It is worth noting that epsilon is never in a FOLLOW set.

Notationally

FOLLOW (A) = {t | (t is a terminal and S => + α A t β) or (t is EOF and S => * αA)}

Some conditions under which symbols a, c, and EOF are in the FOLLOW set of non-terminal A :



How to compute FOLLOW (A) for each non-terminal A :

If the production rule starts with non-terminal A then put EOF in FOLLOW (A).

Scanned by CamScanner

Find all the productions with A on the right-hand-side:

- If production is of type $X \rightarrow \alpha A \beta$, put FIRST $(\beta) - \{\varepsilon\}$ in FOLLOW (A).
- If ε is in FIRST (β) then put FOLLOW(X) into FOLLOW (A).
- For each production $X \rightarrow \alpha A$, put FOLLOW(X) into FOLLOW (A).

If we again see the grammar in Fig. 5.19.15, then what terminals can follow A in a derivation consider the derivation.

$$S' \Rightarrow S\$$$
$$\Rightarrow A\$$$

Since \$ follows A in this (partial) derivation, \$ is in the follow set of A. Again it we see Again we will see what terminals follow B in a derivation for this consider the following derivation

$$A \Rightarrow BC\$$$
$$\Rightarrow Bc\$, \text{ and}$$
$$B \Rightarrow Bb$$

Since b, c follows B in this (partial) derivation, so b, c is in the follow set of B. Again we see

$$A \rightarrow DBC$$
$$\Rightarrow DC$$
$$\Rightarrow Dc$$

Since both \$ and c follow D in this derivation so c and \$ are in the follow set of D.

The follow set for all the non-terminals of the CFG in Fig. 5.19.15 can be found in Fig. 5.19.17.

FOLLOW(S) = { \$ }
FOLLOW(A) = { \$ }
FOLLOW(B) = { b, c, \$ }
FOLLOW(C) = { \$ }
FOLLOW(D) = { b, c, \$ }

Fig. 5.9.17 : The follow sets for all the non-terminals in the CFG of Fig. 5.9.15

5.19.3(D) Finding First and Follow Sets

Algorithm : To calculate first set, for a string of **terminals**.

- If β is ε, then First $(\beta) = \varepsilon$
- If first symbol in β is b, then First $(\beta) = \{b\}$
- If β is $B\beta'$ for B, and string of terminals and non-terminals is β'.
  - If first (B) does not have ε, then First (β) = First (B)
  - If first (A) have ε, then First $(\beta = (\text{First (B)} - \{\varepsilon\}) \cup \text{First } (\beta')$

Algorithm : How to calculate the first sets for all non-terminals CFG.

1. For every non-terminal B in G, First (B) = { }.
2. For each rule $B \rightarrow \beta$ (where β is a string of terminals and non-terminals, then add all the elements of First (β) to First (B). That is :

Scanned by CamScanner

- If β is ε, then add ε to First (B)
- If the first character in β is the terminal b, then add b to first (B).
- If β is $B\beta'$ for some non-terminal B1, and First (B1) does not contain ε, then add all elements of First (B1) to First (B) and recursively add all elements of First (β') to First (B).

3. If there is any change in step 2 go and repeat all the steps of the step 2 again.

Since both \$ and c follow D in this derivation so c and \$ are in the follow set of D.

The follow set for all the non-terminals of the CFG in Fig. 5.19.15 can be found in Fig. 5.19.17.

FOLLOW(S) = { \$ }
FOLLOW(A) = { \$ }
FOLLOW(B) = { b, c, \$ }
FOLLOW(C) = { \$ }
FOLLOW(D) = { b, c, \$ }

Fig. 5.19.18 : The follow sets for all the non-terminals in the CFG of Fig. 5.19.15

Consider the CFG in Fig. 5.19.16, to find the First sets

(0) $S' \rightarrow S\$$ Add { } to First (S')
(1) $S \rightarrow A$ Add { } to First (S)
(2) $A \rightarrow BC \mid DBC$ Add { } to First (A)
(3) $B \rightarrow Bb \mid \varepsilon$ Add { ε } to First (B)
(4) $C \rightarrow c \mid \varepsilon$ Add { c } to First (C)
(5) $D \rightarrow a \mid d$ Add { a, d } to First (D)

Non terminals	First
S'	{ }
S	{ }
A	{ }
B	{ε}
C	{c}
D	{a, d }

Due to some changes the another iteration for finding first sets are :

(0) $S' \rightarrow S\$$ Add { } to First (S')
(1) $S \rightarrow A$ Add { } to First (S)
(2) $A \rightarrow BC \mid DBC$ Add {a, d, ε } to First (A)
(3) $B \rightarrow Bb \mid \varepsilon$ Add { b, ε } to First (B)
(4) $C \rightarrow c \mid \varepsilon$ Add { c, ε } to First (C)
(5) $D \rightarrow a \mid d$ Add { a, d } to First (D)

Scanned by CamScanner

Non terminals	First
S'	{ }
S	{ }
A	{a, d, ε}
B	{b, ε}
C	{c, ε}
D	{a, d}

As there are more changes, the another iteration for finding first sets are :

(0) S' → S$ Add { } to First (S')

(1) S → A Add {a, b, c, d, ε } to First (S)

(2) A → BC | DBC Add {a, b, c, d, ε } to First (A)

(3) B → Bb | ε Add { b, ε } to First (B)

(4) C → c | ε Add { c, ε } to First (C)

(5) D → a | d Add { a, d } to First (D)

Non terminals	First
S'	{ }
S	{a, b, c, d}
A	{a, b, c, d}
B	{b, ε}
C	{c, ε}
D	{a, d}

The final iteration for finding the first set is

S' → S $ Add {a, b, c, d, ε} to First (S')

Final table for first set mentioned in Fig. 5.19.15 given as

Non terminals	First
S'	{a, b, c, d}
S	{a, b, c, d}
A	{a, b, c, d}
B	{b, ε}
C	{c, ε}
D	{a, d}

Follow sets for Non Terminals

Let us see how to find the follow set for given example.

(1) F → (E)

{)} is in the follow set of (E). This is because in this rule) exists immediately after E.

Scanned by CamScanner

(2) If the grammar has the following production rules

$E \rightarrow TF$

$E \rightarrow T, F$

$F \rightarrow 0$

$F \rightarrow 1$

Both {0, 1, ,} are in follow set of T. This can be seen by this following derivation.

$E \Rightarrow TF \Rightarrow T0$

$S \Rightarrow TF \Rightarrow T1$

(3) Consider the following rules

$E \rightarrow FRT$

$R \rightarrow =$

$R \rightarrow +$

$T \rightarrow \varepsilon$

$T \rightarrow ;$

$T \rightarrow *$

Then {=,+} are all in follow set of F and {;,* } are in follow of R

$E \Rightarrow FRT \Rightarrow F{=}T$

$E \Rightarrow FRT \Rightarrow F{+}T$

$E \rightarrow FRT \Rightarrow FR;$

$E \rightarrow FRT \Rightarrow FR*$

Algorithm for finding follow sets for all non-**terminals** in CFG are :

(1) Find First (A) for all non-terminals A in given grammar

(2) Place Follow (A) = { } for all non-terminals A in given grammar.

(3) For rule $A \rightarrow \beta$ in grammar ($A \rightarrow$ non terminal and $\beta \rightarrow$ string of terminals and non-terminals). For every non-terminal A_i in β

(i) Consider the rule $A \rightarrow \alpha A_1 \gamma$ (α and $\gamma \rightarrow$ strings of terminals and non-terminals, and First (γ) does not contain (ε) add all elements of First (γ) to follow (A).

(ii) Consider rule is $A \rightarrow \alpha A_1 \gamma$, same as (i) but in this β does contain ε, add all elements of First (γ) except ε to Follow (A_1), and add all elements of follow (A) to Follow (A_1).

(4) Repeat step 3 if there are any changes.

Find out the follow sets for all non-terminals mentioned in grammar.

Terminals = {+, 0, 1, *}

Non-terminals = {X, W, Y, Z}

Rules = (0) $X' \rightarrow X\$$

(1) $X \rightarrow YZ$

(2) $Z \rightarrow 0WY$

(3) $Z \rightarrow Y+$

(4) $Y \rightarrow \varepsilon$

(5) $Y \rightarrow +W1$

Scanned by CamScanner

(6) $W \rightarrow *$

(7) $W \rightarrow Z\ 1$

Start symbol = X′

Non terminals	First
X′	{0, +}
X	{0, +}
W	{*, 0, +}
Y	{+, *, 1, ε}
Z	{0, +, 1}

Place follow (X) = { } for all non-terminals in S, and then check every rule in given grammar.

(0) $X' \rightarrow X\$$ Add {$} to Follow(X)

(1) $X \rightarrow YZ$ Add First(Z);{0} to Follow(Y)
Add Follow(X);{$} to Follow (Z)

(2) $Z \rightarrow 0WY$ Add First(Y);{+} to Follow(W)
Add Follow(Z);{$} to Follow(W)
Add Follow(Z);{$} to Follow(Y)

(3) $Z \rightarrow Y+$ Add {+} to Follow(Y)

(4) $Y \rightarrow \varepsilon$ No change

(5) $Y \rightarrow +\ W\ 1$ Add {1} to Follow(W)

(6) $W \rightarrow *$ No change

(7) $W \rightarrow Z\ 1$ Add {1} to Follow(Z)

Non terminals	Follow
X′	{ }
X	{$}
W	{+, 1, $}
Y	{0, +, $}
Z	{1, $}

So the final table for the follow sets is given as

Non terminals	Follow
X′	{$}
X	{$}
W	{+, 1, $}
Y	{0, +, $}
Z	{1, $}

Scanned by CamScanner

Another algorithm for finding FIRST and FOLLOW can also be written as :

Given a grammar with rules $A_1 \rightarrow w_1, \ldots, A_n \rightarrow w_n$, We can compute the FIRST $\{w_1\}$ and FIRST $\{A_i\}$ for every rule as follows :

1. Initialize every FIRST $\{w_1\}$ and FIRST $\{A_i\}$ with the empty set $\{\ \}$.
2. Add FIRST $\{w_i\}$ to FIRST $\{A_i\}$ for every rule $A_i \rightarrow w_i$ where FIRST is defined as follows :
   - If w_i is terminal then add FIRST $\{A_i\} = \{a\}$.
   - If w_i is ε then add FIRST $\{A_i\} = \{\varepsilon\}$
   - If w_i is non terminal and $w_i \rightarrow z_1 z_2 z_3 z_4 \ldots z_{i-1} z_i \ldots z_k$ then add FIRST $\{A_i\}$ $= \{z_1\}$ if z_1 is not derived ε else discard z_i up to ε is derived and add FIRST $\{A_i\} = \{z_{i+1}, \varepsilon\}$
3. Add FIRST $\{w_i\}$ to FIRST $\{A_i\}$ for every rule $A_i \rightarrow w_i$
4. Do steps 2 and 3 until all FIRST sets stay the same.

Algorithm

```
FIRST (A)
For all non terminal A do first (A) = { }
while there are changes to any FIRST (A)
for each production choice A→ X1 X2 ... Xn do.
    k = 1;
    continue = true
    while continue = true and k <= n do
        add FIRST (Xk) = {ε} to FIRST (A)
        If ε is not in first (Xk) then
Continue = False
k = k + 1
If continue = true then
add ε to FIRST (A) ;
```

Example 5.19.17

Consider a grammar

(1) $S \rightarrow Aa \mid b$

(2) $A \rightarrow bdZ \mid eZ$

(3) $Z \rightarrow cZ \mid adZ \mid \varepsilon$

- FIRST (S) = {FIRST (A), FIRST (b)} = {b, e} (by rule 3)
- FIRST (A) = {FIRST (b), FIRST (e)} = {b, e} (by rule 2 and 1)
- FIRST (Z) = {FIRST (c), FIRST (a), FIRST {ε}} = {a, c, ε}

Scanned by CamScanner

➢ **Example 5.19.18**

Find the FIRST sets for the following grammar :

(1) $S \rightarrow ACB \mid CbB \mid Ba$

(2) $A \rightarrow da \mid BC$

(3) $B \rightarrow g \mid \varepsilon$

(4) $C \rightarrow h \mid \varepsilon$

First sets

$$\text{FIRST}(S) = \text{FIRST}(ACB) \cup \text{FIRST}(CbB) \cup \text{FIRST}(Ba) \quad \ldots(1)$$

$$\text{FIRST}(A) = \text{FIRST}(da) \cup \text{FIRST}(BC) \quad \ldots(2)$$

$$\text{FIRST}(B) = \text{FIRST}(g) \cup \text{FIRST}(\varepsilon) \quad \ldots(3)$$

$$\text{FIRST}(C) = \text{FIRST}(h) \cup \text{FIRST}(\varepsilon) \quad \ldots(4)$$

So $\text{FIRST}(B) = \{g, \varepsilon\}$ ...(5) from (3)

$\text{FIRST}(C) = \{h, \varepsilon\}$ ...(6) from (4)

$$\begin{aligned}\text{FIRST}(BC) &= [\text{FIRST}(B) - \{\varepsilon\}] \cup \text{FIRST}(C)\\ &= [\{g, \varepsilon\} - \{\varepsilon\}] \cup \{h, \varepsilon\}\\ &= \{g\} \cup \{h, \varepsilon\} \quad \ldots(7)\\ &= \{g, h, \varepsilon\} \quad \ldots(8) \text{ from (2 and 7)}\end{aligned}$$

So $\text{FIRST}(A) = \{d, g, h, \varepsilon\}$

$$\begin{aligned}\text{FIRST}(ACB) &= [\text{FIRST}(A) - \{\varepsilon\}] \cup [\text{FIRST}(C) - \{\varepsilon\}] \cup \text{FIRST}(B)\\ &= [\{d, g, h, \varepsilon\} - \{\varepsilon\}] \cup [\{h, \varepsilon\} - \{\varepsilon\}] \cup [\{g, \varepsilon\}]\\ &= \{d, g, h, \varepsilon\} \quad \ldots(9)\end{aligned}$$

$$\begin{aligned}\text{FIRST}(CbB) &= [\text{FIRST}(C) - \{\varepsilon\}] \cup [\text{FIRST}(b)] \cup [\text{FIRST}(B)]\\ &= [\{h, \varepsilon\} - \{\varepsilon\}] \cup [\{b\}] \cup [\{g, \varepsilon\}]\\ &= \{b, g, h, \varepsilon\} \quad \ldots(10)\end{aligned}$$

$$\begin{aligned}\text{FIRST}(Ba) &= [\text{FIRST}(B) - \{\varepsilon\}] \cup \text{FIRST}(a)\\ &= \{g, a\} \quad \ldots(11)\end{aligned}$$

So $\text{FIRST}(S) = \{a, b, d, g, h, \varepsilon\}$ ...from (1, 9, 10, and 11)

FOLLOW (w)

Written as FOLLOW (w) here, which is defined as the set of terminals such that there is a string of symbols α AaB that can be derived from the start symbol. Computing the follow-sets for the non terminals in a grammar can be done as follows :

1. Initialize every FOLLOW (A_i) with the empty set and a entry of \$ at the end of this set.
2. If there is a rule of the form $A_j \rightarrow wA_i w'$ then,

   If the terminal a is in FIRST (w'), then add a to FOLLOW (A_i)

   If ε is in FIRST (w'), then add FOLLOW (A_j) to FOLLOW (A_i) or any production of form occur like $A_j \rightarrow wA_i$
3. Repeat step 2 until all FOLLOW sets stay the same.

Scanned by CamScanner

Algorithm

FOLLOW (A)

FOLLOW (Start_Symbol) = {$}

for all non terminal ≠ start symbol do

FOLLOW (A) = { }

While there are changes to any FOLLOW set do

for each production $A \rightarrow X_1 X_2 X_3 \ldots X_n$ do

for each X_i that is a nonterminal do

add first $\{X_{i+1} X_{i+2} \ldots X_n\} - \{\varepsilon\}$

to FOLLOW (X_i)

If ε is in FIRST $(X_{i+1} X_{i+2} X_n)$ then

Add FOLLOW (A) to FOLLOW (X_i)

Example 5.19.19

Consider a simple grammar as :

(1) $S \rightarrow Aa \mid b$

(2) $A \rightarrow bdZ \mid eZ$

(3) $Z \rightarrow cZ \mid adZ \mid \varepsilon$

FOLLOW (S) = {$}

FOLLOW (A) = {a}

FOLLOW {Z} = {FOLLOW (A)} = {a}

The use of **FIRST ( ) and FOLLOW ( )**

FIRST and FOLLOW give us information about the next characters expected in the grammar.

Note :

To compute FIRST(A) we must look for A on a production's left-hand side.

To compute FOLLOW(A) we must look for A on a production's right-hand side.

FIRST and FOLLOW sets are always sets of terminals (plus, perhaps, ε for FIRST sets, and EOF for follow sets). Non-terminals are never in a FIRST or a FOLLOW set.

Example 5.19.20 : Find first and follow set for given grammar below :

$E \longrightarrow TE'$ $\quad E' \longrightarrow + TE' \mid \in$

$T \longrightarrow FT'$ $\quad T' \longrightarrow * FT' \mid \in$

$F \longrightarrow (E)$ $\quad F \longrightarrow id$

MU - Dec. 2015, May 2016, 5 Marks

Scanned by CamScanner

Solution :

FIRST and Follow for the following Grammar :

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Now FIRST Set's

FIRST (F)	= {(, id}	FIRST (TE′)	= {(, id}
FIRST (T′)	= {*, ε}	FIRST (+ TE′)	= {+}
FIRST (T)	= {(, id}	FIRST (ε)	= {ε}
FIRST (E′)	= {+, ε}	FIRST (FT′)	= {(,id}
FIRST (E)	= {(,id}	FIRST (*FT′)	= {*}
FIRST (ε)	= {ε}		
FIRST ((E))	= {(}		
FIRST (id)	= {id}		

FOLLOW Set's

FOLLOW (E) = {$,) }

FOLLOW (E) = {$,) }

FOLLOW (T) = {+,), $}

FOLLOW (T) = {+,), $}

FOLLOW (F) = {+, *,), $}

5.19.4 LL (1) Parse Tables

PARSE TABLE can be build once FIRST and FOLLOW sets for all non-terminals in the context free grammar are obtained. Rows represents non-terminals and terminals are located in columns of the pare table. Each entry in the parse table is either empty, or contains a grammar rule.

The rule placed at row of non-terminal and a column of a parse table tells us which rule to apply when we are trying to parse the non-terminal S, and the next input symbol in the input is a. Consider the following context free grammar:

Terminals = {+, 0, 1, *}

Non-terminals = {X, W, Y, Z}

Rules = (0) $X' \rightarrow XS$

Scanned by CamScanner

(1) $X \rightarrow YZ$

(2) $Z \rightarrow 0WY$

(3) $Z \rightarrow Y+$

(4) $Y \rightarrow \varepsilon$

(5) $Y \rightarrow +W1$

(6) $W \rightarrow *$

(7) $W \rightarrow Z1$

Start symbol = X'

The First Sets for the above given CFG is

Non terminals	First
X'	{0, +}
X	{0, +}
W	{*, 0, +}
Y	{+, *, 1, ε}
Z	{0, +, 1, ε}

The Follow Sets for the above given CFG is

Non terminals	Follow
X'	{$}
X	{$}
W	{+, 1, $}
Y	{0, +, $}
Z	{1, $}

Parse table for the above grammar is

	0	+	*	1
X'	$X' \rightarrow X\$$	$X' \rightarrow X\$$		
X	$X \rightarrow YZ$	$X \rightarrow YZ$		$X \rightarrow YZ$
W	$W \rightarrow Z1$	$W \rightarrow Z1$	$W \rightarrow *$	$W \rightarrow Z1$
Y	$Y \rightarrow \varepsilon$	$Y \rightarrow +W1$ $Y \rightarrow \varepsilon$		
Z	$Z \rightarrow 0WY$	$Z \rightarrow Y+$		

Once the parse table is created, it becomes easy to create recursive descent parser.

Scanned by CamScanner

Creating parse tables A parse table is created as follows :

- Rows of the parse table are represent the non-terminals of the grammar.
- Columns of the parse table represent terminals of the grammar.
- Every label entered in the parse table is either empty, or contains grammar rule.
  - Place each rule of the form $S \rightarrow \gamma$ in row S, in each column in first (γ).
  - Place each rule of the form $S \rightarrow \gamma$, where first (γ) contains ε, in row S in each column in FOLLOW (S)

Consider again the CFG of Fig. 5.19.15. We have already computed the FIRST and FOLLOW sets of each non-terminal.

Non-terminal	FIRST	FOLLOW
S'	{a, b, c, d}	{$}
S	{a, b, c, d}	{$}
A	{a, b, c, d}	{$}
B	{b, ε}	{b}
C	{c, ε}	{$}
D	{a, d}	{b, c}

Parse table for this grammar. Looking at each rule in turn :

(0) $S' \rightarrow S\$$ First (S') = {a, b, c, d} — $S' \rightarrow S\$$ goes in row S', columns a, b, c and d.

(1) $S \rightarrow A$ FIRST (A) = {a, b, c, d} — $S \rightarrow A$ goes in row S, column a, b, c and d.

(2) $A \rightarrow BC$ FIRST (BC) = {b, c} — $A \rightarrow BC$ goes in row A, column b and c

(3) $A \rightarrow DBC$ FIRST (DBC) = {a, d} — $A \rightarrow DBC$ goes in row A, column a and d.

(4) $B \rightarrow Bb$ FIRST (Bb) = {b} — $B \rightarrow Bb$ goes in row B, column b.

(5) $B \rightarrow \varepsilon$ FIRST (ε) = { ε} — $B \rightarrow \varepsilon$ goes in row B, column c.

(6) $C \rightarrow c$ FIRST (c) = {c} — $C \rightarrow c$ goes in row C, column c.

(7) $C \rightarrow \varepsilon$ FIRST (ε) = { ε } — $C \rightarrow \varepsilon$

(8) $D \rightarrow a$ FIRST (a) = {a} — $D \rightarrow a$ goes in row D, column a.

(9) $D \rightarrow d$ FIRST (d) = {d} — $D \rightarrow d$ goes in row D, column d.

Resulting parse table

	A	b	c	d
S'	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$
S	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$
A	$A \rightarrow DBC$	$A \rightarrow BC$	$A \rightarrow BC$	$A \rightarrow DBC$
B		$B \rightarrow Bb$	$B \rightarrow \varepsilon$	
C			$C \rightarrow c$	
D	$D \rightarrow a$			$D \rightarrow d$

Scanned by CamScanner

Consider the context tree grammar in Fig. 5.19.19.

Session → Facts question | (Session) Session
Facts → Fact Facts | ∈
Fact → ! STRING
Question → STRING

Fig. 5.19.19 : A CFG for simple statements

Non-terminal	FIRST	FOLLOW
Session	{(, ?, !}	{), #}
Facts	{ε, !}	{?}
Fact	{!}	{ !}
Question	{?}	{#,) }

Since the first and follow sets are given, we can create parse table for the grammar.

	(	)	#	!	?
Session	(Session) Session			Facts Question	Facts Question
Facts				Fact Facts	ε
Fact				! STRING	
Question					? STRING

Example 5.19.21

(1) S → F
(2) S → (S + F)
(3) F → 1

FIRST SETS

FIRST (S) = FIRST {F} = {1}
FIRST (S) = FIRST {(} = {(}
FIRST (F) = FIRST {1} = {1}

Non-Terminal	FIRST
S	{1, (}
F	{1}

FOLLOW SET

FOLLOW (S) = FOLLOW {F, $}
FOLLOW (S) = FOLLOW {+ F), $}
FOLLOW (S +) = FOLLOW {F), $}

Scanned by CamScanner

FOLLOW (S + F) = FOLLOW {), $}

FOLLOW (S) = FOLLOW {, $}

FOLLOW (F) = FOLLOW {1}

So

Non-Terminal	FOLLOW
S	{$}
F	{1}

LL (1) TABLE

Non-Terminals	Input Symbols				
	(	)	1	+	$
S	S → (S + F)		S → F		
F			F → 1		

➢ **Example 5.19.22**

Consider the following grammar and find FIRST and FOLLOW SET and construct the parsing table.

S → stmtS′

S′ → ; S | ε

Stmt → s

Now FIRST and FOLLOW SET for the given grammar as

Non-Terminal	FIRST SETS	FOLLOW SETS
S	{s}	{$}
S′	{; ε}	{$}
Stmt	{s}	{; , $}

Parsing table

	S	;	$
S	S → Stmt S′		
S′		S′ → ; S	S′ → ε
Stmt	Stmt → s		

➢ **Example 5.19.23**

Consider the following grammar

1. S → P $
2. P → {D ; C}
3. D → d, D | d
4. C → c, C | c

Scanned by CamScanner

The above grammar corresponds loosely to the structure of programs. Need to left factor the grammar first

1. $S \rightarrow P\$$
2. $P \rightarrow \{D ; C\}$
3. $D \rightarrow d\ D2$
4. $D2 \rightarrow , D \mid \varepsilon$
5. $C \rightarrow c\ C2$
6. $C2 \rightarrow , C / \varepsilon$

Now we find the first and follow sets as

	FIRST	FOLLOW
S	{{}	{$}
P	{{}	{$}
D	{d}	{;}
D2	{, ε}	{;}
C	{c}	{}}
C2	{, ε}	{}}

Now constructing LL Parser table.

	{	}	;	,	c	d	$
S	S → P $						
P	P → {D; C}						
D							
D2			D2 → ε	D2 → , D		D → dD2	
C					C → cC2		
C2		C2 → ε		C2 → , C			

Example 5.19.24

Find the FIRST and FOLLOW of the following grammar and then parse it via LL (1) parser.

(1) exp → term exp1

(2) exp1 → op1 term exp1 | ε

(3) term → factor term 1

(4) term 1 → op2 factor term 1 | ε

(5) factor → (exp) | num

(6) op1 → + | –

(7) op2 → *

Scanned by CamScanner

The FIRST and FOLLOW set are as

	FIRST SETS	FOLLOW SETS
exp	{(, num}	{), $}
exp1	{+, −, ε}	{), $}
term	{(, num}	{), +, − $}
term 1	{*, ε}	{), +, −, $}
factor	{(, num}	{), +, −, *, $}
op1	{+, −}	−
op2	{*}	−

Now we parse given grammar and design LL (1) parser table.

	(	num	)	+	−	*	$
exp	exp → term exp1	exp → term exp1					
exp1			exp → ε	exp1 → op1 term exp1	exp1 → op1 term exp1		exp → ε
term	term → factor term1	term → factor term 1					
term1			term1 → ε	term1 → ε	term1 → ε	term1 → op2 factor term1	term1 → ε
factor	factor → (exp)	factor → num					
op1				op1 → +	op1 → −		
op2						op2 → *	

➢ **Example 5.19.25**

Find the FOLLOW for the following grammar, if possible otherwise parse it using LL (1).

1. S → P
2. P → pQ
3. Q → qQR / t
4. R → s

In the above grammar there is no ε production, so there is no need to calculate FOLLOW SETS. We can parse it directly with the help of

Scanned by CamScanner

FIRST SETS

So,

FIRST (S) = {p}
FIRST (Q) = {q, t}
FIRST (P) = {p}
FIRST (R) = {s}

Parse table

	p	q	t	s	D	$
S	S → P					
P	P → pQ					
Q		Q → qQR	Q → t			
R				R → s		

Example 5.19.26 : Construct the LL (1) parser table using FIRST and Follow for the following Grammar :

E → TE'
E' → + TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Solution :

Now FIRST Set's

FIRST (F) = {(, id}
FIRST (T') = {*, ε}
FIRST (T) = {(, id}
FIRST (E') = {+, ε}
FIRST (E) = {(,id}

FIRST (TE') = {(, id}
FIRST (+ TE') = {+}
FIRST (ε) = {ε}
FIRST (FT') = {(,id}
FIRST (*FT') = {*}
FIRST (ε) = {ε}
FIRST ((E)) = {(}
FIRST (id) = {id}

FOLLOW Set's

FOLLOW (E) = {$,) }
FOLLOW (E) = {$,) }
FOLLOW (T) = {+,), $}
FOLLOW (T) = {+,), $}
FOLLOW (F) = {+, *,), $}

Scanned by CamScanner

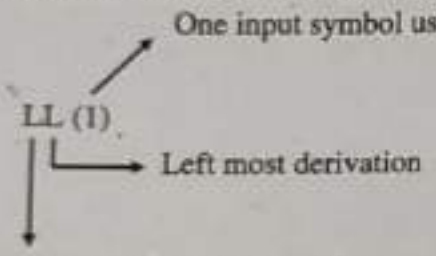
LL (1) Parsing Table

	Id	+	*	(	)
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T'			$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$	

Fig. P. 5.19.26 : LL (1) parsing table

5.19.4(A) LL (1) Grammar

- A grammar whose parsing table has no multiple-defined entries is said to be LL (1) grammar.



- If a parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL (1) grammar.

➢ **Example 5.19.27**

Grammar which is not LL (1)

$S \rightarrow iCtSE \mid a$

$E \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

FIRST (iCtSE) = {i}

FIRST (a) = {a}

FIRST (eS) = {e}

FIRST (ε) = {ε}

FIRST (b) = {b}

FOLLOW (S) = {\$, e}

FOLLOW (E) = {\$, e}

FOLLOW (C) = {t}

Scanned by CamScanner

	A	b	E	i	t	$
S	S → a			S → iCtSE		
E			E → eS			
E → ε			E → ε			
C		C → b				

(Two production rules M [E, e])

Problem in the above example → Ambiguity

Example 5.19.28 : For the following grammar construct LL(1) parser table.
S → F S → (S – F) F → a
And Parse the string (a – a). Show contents of stack and i/p buffer and action taken after each step.
MU - Dec. 2015, 10 Marks

Solution :

S → F

S → (S – F)

F → a

construct LL(1) parser table

Step 1 : Find the first and follow set of the given expression.

FIRST (S) = First (P) = {a,(}

FIRST (f) = {a}

FOLLOW (S) = { $ }

FOLLOW (f) = { – }

LL (1) Table :

Non Terminals	Input symbols			
	a	–	$	(
S	S → f	S → (S – F)		S → (S – F)
F	F → a			

Parse the string (a – a)

Stack	Input	Output
$ S	(a – a)	S → (S – f)
$) S – f (	(a – a)	S → (S – f)
$) S – f	a – a)	f → a
$) S – a	a – a)	f → a
$) S –	– a)	

Scanned by CamScanner

Stack	Input	Output
$) S	a)	S → f
$) f	a)	f → a
$) a	a)	f – a
$)	)	
$	accept	

Example 5.19.29 : Consider the following grammar with terminals (,[,) and].

S → TS | [S]S |)S | ε

T → (X)

X → TX | [X]x | ε

(i) Construct first and follow set for the non-terminals.

(ii) Construct its LL(1) parsing table.

(iii) Is this LL(1) grammar ?

MU - Dec. 2014, 10 Marks

Solution :

S → TS | [S] S | | S | ε

T → (X)

X → T X | [X] X | ε

(i)

FIRST SETS

Non terminal	FIRST SET
S	(, [,), ε,]
T	(,
X	(, [,ε,)

FOLLOW SETS

Non terminal	Follow set
S	],
T	[,)
X	),]

(ii) LL(1) Parsing table

N.T. \ T	(	[	)	]	ε
S	S → TS	S → [S] S	S →) S	S → [S] S	S → ε
T	T → (X)				
X	X → TX	X → [X] X	X → TX		X → ε

(iii) The given grammar is LL(1)

Example 5.19.30 : Construct predictive parsing table for following grammar.

S → A

A → aB | Ad

B → bBC | f

C → g

MU - May 2015, 10 Marks

Scanned by CamScanner

Solution :

$S \rightarrow A$	$S' \rightarrow S\$$
$A \rightarrow aB \mid Ad$	$S \rightarrow A$
$B \rightarrow bBC \mid f$	$A \rightarrow aB \mid Ad$
$C \rightarrow g$	$B \rightarrow bBC \mid f$
	$C \rightarrow g$

FIRST (S) → { a }	FOLLOW (S) → { $ }
FIRST (A) → { a }	FOLLOW (A) → { d }
FIRST (B) → { b, f }	FOLLOW (B) → { g }
FIRST (C) → { g, f }	FOLLOW (C) → { $ }
FIRST (S') → { a }	

Predictive Parser table or LL(1) table

	a	b	g	f	$
S'	$S \rightarrow S\$$				
S	$S \rightarrow A$				
A	$A \rightarrow aB \mid Ad$				
B		$B \rightarrow bBC$		$B \rightarrow f$	
C			$c \rightarrow g$	$B \rightarrow bBC$	

5.19.4(B) A Grammar which is not LL (1)

- If the parsing table has a multiple entries in it then we need to do the following:
- Eliminate the left recursion from the grammar. A left recursive grammar cannot be LL(1) this can be explained from the following example
  - $A \rightarrow A\alpha \mid \beta$
  - → any terminal that appears in FIRST $\{\beta\}$ also appears in FIRST $(A\alpha)$ because $A\alpha \Rightarrow \beta\alpha$.
  - → If β is ε, any terminal that appears in FIRST (α) also appears in FIRST $(A\alpha)$ and FOLLOW (A).
- Left factor the grammar. If grammar is not left factored then left factor the grammar this can be explained with the help of below mentioned example
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
  - → any terminal that appears in FIRST $(\alpha\beta_1)$ also appears in FIRST $(\alpha\beta_2)$.
- If parsing table still has multiple entries, then that grammar is an ambiguous or it is inherently not a LL (1) grammar.

5.19.4(C) Properties of LL (1) Grammar

A given grammar G will be LL (1) if and only if it holds the following property for two distinct production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$.

Scanned by CamScanner

1. Production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$ cannot derive strings starting with same terminals.
2. Anyone of α and β has derive to ϵ.
3. If β can derive to ϵ, then α cannot derive to any string starting with a terminal in FOLLOW(A).

5.19.4(D) Error Recovery Techniques in Predictive Parsing

- An error may occur in the predictive parsing or LL(1) parsing the reasons for occurring error are as :
  - The terminal symbol on the top of stack does not match with the current input symbol.
  - The top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A, a] is an empty entry.
- How parser should handle such type of errors?
  - The parser should generate a meaningful error message. Parser should be able to recover from that error case, and able to continue the parsing with the rest of the input.
  - Proceed forward after detecting 1st error → Error Recovery (ER).
  - Error recovery technique should be simple and fast.

➢ **Example 5.19.31 : Panic-Mode Error Recovery Example**

$S \rightarrow AbS \mid e \mid \varepsilon$

$A \rightarrow a \mid cAd$

FOLLOW (S) = {$}

FOLLOW (A) = {b, d}

Stack	Input	Output
$S	aab$	$S \rightarrow AbS$
$SbA	aab$	$A \rightarrow a$
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	$S \rightarrow AbS$
$SbA	ab$	$A \rightarrow a$
$Sba	ab$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	Accept

Stack	Input	Output
$S	ceadb$	$S \rightarrow AbS$
$SbA	ceadb$	$A \rightarrow cAd$
$SbdAc	ceadb$	

Scanned by CamScanner

Stack	Input	Output
$SbdA	eadb$	Error : unexpected e (illegal A) (Remove all input tokens until first b or d, pop A)
$Sbd	db$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	Accept

Last Time ... How to Build a LL Parser

Have a complete recipe for building a parser

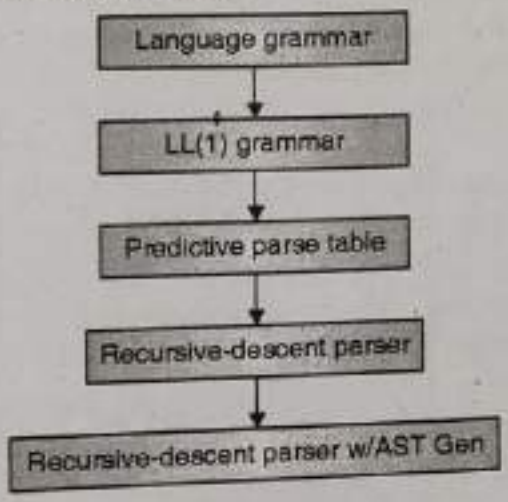


Fig. 5.19.20 : Complete recipe for building a parser

5.20 Bottom-Up Parsing

→ **(MU - Dec. 2014, Dec. 2016)**

Q. Explain Handle. **Dec. 2014, 2 Marks**

Q. What is handle pruning ? **Dec. 2016, 5 Marks**

Q. Define and explain the term with suitable example : Handles. **(5 Marks)**

Bottom-up parsing is based on the reverse process to top-down. This technique attempts to construct a parse tree for an input string beginning from the leaves (the bottom) and working up towards the root (the top). **Bottom-up parsing** (also known as **shift-reduce parsing**) is a strategy for analyzing unknown data relationship that attempts to identify the most fundamental units first, and then to infer higher-order structures from them.

It attempts to build trees upward toward the start symbol. Instead of expanding successive non-terminals according to production rules, a current string or *right-sentential form* is collapsed each time until the start non-terminal is reached to predict the legal next symbol.

Scanned by CamScanner

➤ **Example 5.20.1**

Consider a Grammar

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)

E

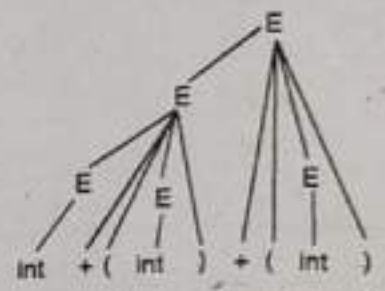


Fig. 5.20.1 : Bottom up parsing for the string "int + (int) + (int)"

Bottom up parsing is a process can be thought of as "reducing" a given string to the start symbol of a grammar. During each ***reduction*** step a particular substring matching the right side of a production is replaced by the symbol on the left of that production.

$E \rightarrow E + T$	generation ↓	E	parsing ↑
$E \rightarrow T$		E + T	
$T \rightarrow T * F$		E + T * F	
$T \rightarrow F$		E + T * id	
$F \rightarrow (E)$		E + F * id	
$F \rightarrow id$		E + id * id	
		T + id * id	
		F + id * id	
		id + id * id	

Fig. 5.20.2 : Right sentential forms

Each of the lines in Fig. 5.20.2 is a (right) sentential form. The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation.

Suppose we have a grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

And the input string is

abbcde

Then an instance of bottom-up parsing can be given as

abbcde

aAbcde

aAde

aABe

S

Scanned by CamScanner

Thus this process is like tracking out the right most derivations in reverse.

Rightmost derivation : In this we replace the rightmost non-terminal in each step of the derivation.

In context of the above example right derivations can be shown as

S -> aABe -> aAde -> aAbcde -> abbcde

Here in each step we have expanded the rightmost terminal.

Note : A left-sentential form occurs in a leftmost derivation.
A right-sentential form occurs in a rightmost derivation.

Handle

A handle is a right sentential sequence which is the rightmost derivation within the string being parsed. A handle of a string is a substring that matches the right hand side of a production and whose reduction to the nonterminal on the left hand side of the production represents one step along the reverse of a **rightmost derivation** that ultimately leads to the start symbol. If replacing a string does not ultimately lead to the start symbol it can't be a handle.

If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle. The parser must find a substring β of the tree's such that it matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation. This substring β is a handle.

The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning. Handle pruning forms the basis for a bottom-up parsing method.

In the above example **Abc** is an example of a handle.

$S \rightarrow aABe \rightarrow aAde \rightarrow a\mathbf{Abc}de \rightarrow abbcde$

Example 5.20.2

Consider the grammar for input string "id + id + id"

$E \rightarrow E+T \mid T$ Right-Most Derivation of id+id+id

$T \rightarrow T+F \mid F$ $E \Rightarrow E+T \Rightarrow E+T+F \Rightarrow E+T+id \Rightarrow E+F+id$

$F \rightarrow (E) \mid id$ $\Rightarrow E+id+id \Rightarrow id+id+id \rightarrow$ The rightmost derivation

Example 5.20.3

Consider the grammar mentioned in Example 5.20.2 and the input string is "id1+id2+id3"

Right-Most Sentential Form	Reducing Production
id+id+id	$F \rightarrow id$
<u>F</u>+id+id	$T \rightarrow F$
<u>T</u>+id+id	$E \rightarrow T$
E+<u>id</u>+id	$F \rightarrow id$
E+<u>F</u>+id	$T \rightarrow F$
E+T+<u>id</u>	$F \rightarrow id$
E+<u>T+F</u>	$T \rightarrow T*F$
<u>E+T</u>	$E \rightarrow E+T$
E	

Scanned by CamScanner

Handles are red and underlined in the right-sentential forms.

Here it can be noted that string appearing to the right of a handle contains only terminal symbols. It should be observe that the sequence of right sentential forms in this Example is just the reverse of the sequence in the first rightmost derivation in Example.

Since here the grammar is ambiguous the choices for the handles can be different depending upon right derivations used.

The main problems faced by the bottom up parsing using handles are as :

1. Identifying the handle
2. Identifying the rule to reduce.

The most important aspect of bottom up parsing is the process of detecting handles and using them in reductions. Bottom up parsing is a parsing method that works by identifying terminal symbols first, and combines them successively to produce non-terminals. The productions of the parser can be used to build a parse tree. This process can be implementing using ***Shift-Reduce Parsing***.

5.20.1 Shift-Reduce Parser

Q. Explain shift reduce parser in detail. **(5 Marks)**

Shift reduce parsers use the principle of the bottom-up parsing. Shift reduce parsers *shifts* input symbols until a handle is found. Then, *reduce* the substring to the non-terminal on the left hand side of the corresponding production. One scheme to implement a handle-pruning, bottom-up parser is called a **shift-reduce parser.** The various data structures required for a shift-reduce parser are as follows :

(i) A stack to hold grammar symbols and to detect handles

(ii) An input buffer to hold the input string to be parsed.

(iii) A data structure for storing and accessing the left hand side and right hand side of the production rules.

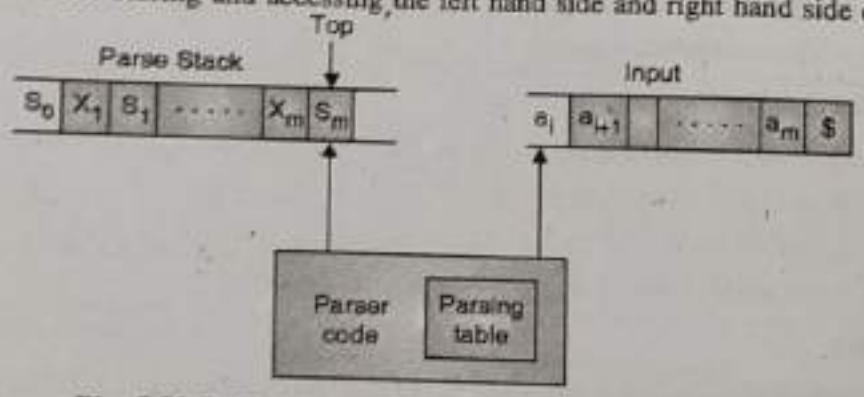


Fig. 5.20.3 : Basic block diagram of shift reduce parser

The shift reduce parser performs the following four actions :

(i) **Shift** – next input symbol is shifted onto the stack

Move ↑ one place to the right

Shifts a terminal to the left string

E + (↑ int) ⇒ E + (int ↑)

(ii) **Reduce – It handle is at the top of the stack**

- pop handle
- push appropriate left hand side

Scanned by CamScanner

- Apply an inverse production at the right end of the left string
- If $E \rightarrow E + (E)$ is a production, then
- $E \rightarrow (E + (E) \uparrow) \Rightarrow E + (E \uparrow)$
- **Accept** – stop parsing and report success
- **Error** – call error reporting/recovery routine.

Algorithm for Shift Reduce Parser

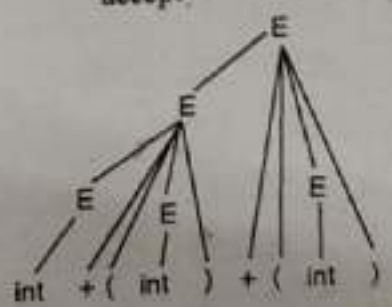
- Shift-reduce parsers use a **stack** and an **input buffer**

1. Initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is **eof**
   a) **find the handle**
      if we don't have a handle on top of the stack, **shift** an input symbol onto the stack.
   b) **prune the handle**
      if we have a handle ($A ::= \beta, k$) on top of the stack, **reduce**
      i) pop $|\beta|$ symbols off the stack
      ii) push A onto the stack

Example 5.20.4

Shift-reduce parsing $E \rightarrow int$, $E \rightarrow E + (E)$

$\uparrow$ int + (int) + (int) $	shift
int $\uparrow$ + (int) + (int) $	red. $E \rightarrow int$
E $\uparrow$ + (int) + (int) $	shift 3 times
E + (int $\uparrow$) + (int) $	red. $E \rightarrow int$
E + (E $\uparrow$) + (int) $	shift
E + (E) $\uparrow$ + (int) $	red. $E \rightarrow E + (E)$
E $\uparrow$ + (int) $	shift 3 times
E + (int $\uparrow$) $	red. $E \rightarrow int$
E + (E $\uparrow$) $	shift
E + (E) $\uparrow$ $	red. $E \rightarrow E + (E)$
E $\uparrow$ $	accept



Scanned by CamScanner

➢ **Example 5.20.5**

Shift-reduce parsing

Grammar :

1. S → E
2. E → E + E
3. E → E * E
4. E → **num**
5. E → **id**

Input to parse is :

id_1 + num * id_2

Handles : <u>underlined</u>

Stack	Action
$	Shift
$ id_1	Reduce (rule 5)
$ E	Shift
$ E +	Shift
$ E + <u>num</u>	Reduce (rule 4)
$ E + E	Shift
$ E + E *	Shift
$ E+ E * <u>id_2</u>	Reduce (rule 5)
$ E + E * E	Reduce (rule 3)
$ <u>E + E</u>	Reduce (rule 2)
$ <u>E</u>	Reduce (rule 1)
$ S	Accept

The main two problems stated above are :

Shift-reduce conflicts : It occurs when the parser cannot decide whether to shift a token onto the stack or reduce a handle

Reduce-reduce conflicts : In this parser is not able to decide which reduction to make.

➢ **Example 5.20.6**

Consider the following actions of a shift-reduce parsers for the string "x – 2*y" defined in Fig. 5.20.2.

0	Goal → Expr
1	Expr → Expr + Term
2	Expr → Expr – Term
3	Expr → Term
4	Term → Term * Factor
5	Term → Term / Factor
6	Term → Factor
7	Factor → number
8	Factor → id
9	Factor → (Expr)

Scanned by CamScanner

Stack	Input	Handle	Action
$	id - num * id	*none*	shift
$ id	- num * id	9,1	reduce 9
$ <factor>	- num * id	7,1	reduce 7
$ <term>	- num * id	4,1	reduce 4
$<expr>	- num * id	*none*	shift
$<expr> -	num * id	*none*	shift
$<expr> - num	* id	8,3	reduce 8
$<expr> - <factor>	* id	7,3	reduce 7
$<expr> - <term>	* id	*none*	shift
$<expr> - <term> *	id	*none*	shift
$<expr> - <term> * id		9,5	reduce 9
$<expr> - <term> * <factor>		5,5	reduce 5
$ <expr> - <term>		3,3	reduce 3
$ <expr>		1,1	reduce 1
$<goal>		*none*	accept

Fig. 5.20.4 : Actions of a shift reduce parsers for the string "x-2*y"

There are CFG's for which shift-reduce parsing cannot be used. For each grammars the Shift-reduce or reduce-reduce conflicts cannot be resolved.

Now, we will look into main classes of bottom-up parsing i.e. operator precedence and LR parsing. Operator precedence is a simpler parser to construct and of course has lesser power than LR parsers.

5.20.2 Operator Precedence Parser

→ (MU - Dec. 2014, May 2015, Dec. 2017)

Q. Explain operator preceding parsing with example. **Dec. 2014, May 2015, 10 Marks**

Q. What do you mean by operator precedence parser ? **Dec. 2017, 5 Marks**

Q. Explain operator precedence parsing. **(5 Marks)**

Q. Explain working of operator precedence parsing with the help of an example. **(5 Marks)**

It is a bottom-up parser that interprets an operator-precedence grammar. For example, most calculators use operator precedence parsers to convert from the human-readable infix notation. The operator precedence parsing technique can be applied to operator grammars.

Operator Grammar are defined as the grammars with the following properties :

1. **No epsilon** in the right hand side of any production
2. **No adjacent non-terminals** in the right hand side of any production.

Scanned by CamScanner

This property enables the implementation of efficient operator-precedence parsers. These parsers rely on the following three precedence relations :

Relation	Meaning
a <· b	a yields precedence to b
a =· b	a has the same precedence as b
a ·> b	a takes precedence over b

Fig. 5.20.5 : Precedence relations

These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end. Also, **a <· b** need not imply **b ·> a**

Let us assume that between the symbols ai and ai + 1 there is exactly one precedence relation. Suppose that $ is the end of the string.

Then for all terminals we can write : $ <· b and b ·> $. If we remove all nonterminals and place the correct precedence relation : <·, =· , ·> between the remaining terminals, there remain strings that can be analyzed by developed parser.

For example, the following operator precedence relations can be introduced for simple expressions :

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Difference between operator precedence parsing and Recursive descent parsing

➔ **(MU - May 2014)**

Q. State the differences between operator precedence parsing and Recursive descent parsing with examples. **May 2014, 10 Marks**

Sr. No.	Operator parsing	Recursive parsing
1.	Bottom up parsing technique	Top down parsing technique
2.	Applied to operator grammar	Backtracking is needed
3.	Build the parse tree from leaves to root	Left most derivation is used in this.
4.	Applied to small set of grammars	Not widely used.

Example 5.20.7 : Consider the grammar

$E \rightarrow E + E \mid E * E \mid id$

MU - Dec. 2015, Dec. 2016, 10 Marks

Scanned by CamScanner

Precedence Table

	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
Id	·>	·>	·>	·>
$	<·	<·	<·	<·

Consider the input string "id1 + id2 * id3"

The string with the precedence relations inserted from the above table is

$ <· id1 ·> + <· id2 ·> * <·id3 ·> $

Having precedence relations allows identifying handles as follows :

- scan the string from left until seeing ·>
- scan backwards the string from right to left until seeing <·
- everything between the two relations <· a ·> forms the handle

The handles can be found as

If <· Push in stack

If ·> Pop till a <· is found and reduce

$ <· id1 ·> + <·id2 ·> * <·id3·> $

$ <· E + <·id2 ·> * <·id3·> $

$ <· E + <·id2 ·> * <·id3·> $

$ <· E + <·id2 ·> * <·id3·> $

$ <· E + <· E * <·id3·> $

$ <· E + <· E * E ·> $

$ <· E + E· > $

$ <· E ·> $

Example 5.20.8 : With the help of following grammar and given string explain role of operator precedence parser.

E → E + T / T

T → T * V / V

V → a / b / c / d

String to parse 'a + b*c*d'

MU - Dec. 2017, 10 Marks

Solution :

Operator precedence table for string "a + b * c * d" using the algorithm mentioned in section (5.20.2(A))

Stack	Input	Stack TOP	Current Input	Action
$	a+b*c*d$	$	a	Shift a
$a	+b*c*d$	A	+	Reduce using V → a
$v	+ b * c * d $	V	+	Reduce using T → V

Scanned by CamScanner

Stack	Input	Stack TOP	Current Input	Action
$T	+b*c*d$	T	+	Reduce using E → T
$E	+b*c*d$	E	+	Shift +
$E+	b*c*d$	+	b	Shift b
$E+b	*c*d$	B	*	Reduce using V→ b
$E+V	*c*d$	V	*	Reduce using T → V
$E+T	*c*d$	T	*	Shift *
$E+T*	c*d$	*	c	Shift C
$E+T*C	*d$	C	*	Reduce using V→ C
$E+T*V	*d$	V	*	Reduce using T → T*V
$E+T	*d$	T	*	Shift *
$E+T*	d$	*	d	Shift d
$E+T*d	$	D	$	Reduce using V → d
$E+T*V	$	V	$	Reduce using T → T*v
$E+T	$	T	$	Reduce using E → E+T
$E	$	E	$	Accept +

5.20.2(A) Operator Precedence Parsing Algorithm

→ (MU - Dec. 2014)

1. If the front of input has $ and top of stack both have $, it's done
   Else
2. Compare front of input b with ·>
   if b! = '·>'
   then push b
   Scan the next input symbol
3. If b == '·>'
   Then pop till <· and store it in a string S
   pop <· also
   reduce the popped string
   If (top of stack) <· (Front of input)
   then push <· S
   If (top of stack) ·> (Front of input)
   then push S and goto 3

OR

Initialize stack to $
While ( ) do

Scanned by CamScanner

```
Let U be the topmost terminal on the stack
Let V be the next input symbol
If U = $ and V = $ then return
If U <· V or U =· V then
    Shift V onto the stack
    Advance the input pointer
Elseif U ·> V then
    Do
        Pop the topmost symbol, call it V, from the stack
    Until the top of the stack is <· V
Else
    Error
End
end
```

Operator Precedence table for the string "id1 + id2 * id3" using the above algorithm explained below

	Stack	Input	Action
1	$	<·id1·>+<·id2·>*<·id3·>$	Shift
2	$ <·id1	·>+ <·id2·>*<·id3·> $	Reduce (E → id)
3	$<·E	+<·id2·>*<·id3·>$	Shift
4	$<·E+<·id2	·>*<·id3·>$	Reduce (E → id)
5	$<·E+<·E	*<·id3·>$	Shift
6	$<·E+<·E*<·id3	·>$	Reduce (E → id)
7	$<·E+<·E*E	·>$	Reduce (E → E*E)
8	$<·E+E	·>$	Reduce (E → E+E)
9	$E	$	Accept

Fig. 5.20.6 : Operator Precedence Table Based on Algorithm

5.20.2(B) How to Create Operator Precedence Relations

➔ (MU - Dec. 2014)

We use associativity and precedence relations among operators to see that the operator-precedence parsing algorithm will work correctly when guided by them.

1. X has higher precedence than Y i.e. X ·> Y and X <· Y. If + has higher precedence then – then it ensure that expression of the form Expr + Expr – Expr -Expr, the Expr + Expr is the handle that will be reduced first.

Scanned by CamScanner

2. X has equal precedence than Y, then
   - they are left-associative → X ·> Y and Y ·> X
   - they are right-associative → X <· Y and Y <· X
3. For all operators X, X <· id, id ·> X, X <· (, (<· X, X ·>),) ·> X, X ·> $ and $ <· θ for all operators X.

➢ **Example 5.20.9**

Figure contains the operator precedence relations for grammar

E → E + E|E – E | E*E |E/E| (E) | – E | id assuming

1. ↑ is of highest precedence and right associative
2. * and / are of next highest precedence and left associative, and
3. + and – are of lowest precedence and left-associative.

	+	–	*	/	^	Id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
–	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

5.20.2(C) Handling Unary Minus in Operator Precedence Parser

- Operator-Precedence parsing is not able to handle the both unary minus and the binary minus in the grammar.
- Instead of operator precedence parser lexical analyzer handle this problem.
  - Lexical analyzer will return two different tokens for the unary minus and the binary minus.
  - It will have a look ahead to distinguish the binary minus from the unary minus.
- Then, we make

  θ <· unary-minus for any operator.

  unary-minus ·> θ if unary-minus has higher precedence than θ.

  unary-minus <· θ if unary-minus has lower (or equal) precedence than θ.

5.20.2(D) Precedence Functions in Operator Precedence Parser

- Operator precedence parser can be encoded by two precedence functions x and y that map terminal symbols into integers.

Scanned by CamScanner

- For symbols a and b.

  $x(a) < y(b)$ where $a \lessdot b$

  $x(a) = y(b)$ where $a \doteq b$

  $x(a) > y(b)$ where $a \gtrdot b$

5.20.2(E) Algorithm for Constructing Precedence Functions

1. Generate functions $\mathbf{x_a}$ for each grammar terminal **a** and for the end of the string symbol.
2. Partition the symbols in groups so that $\mathbf{x_a}$ and $\mathbf{y_b}$ are in the same group if $a \doteq b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Generate a directed graph whose nodes are in the groups, for each symbols a and b do: place an edge from the **group of $\mathbf{y_b}$ to the group of $\mathbf{x_a}$ if a** $\lessdot$ b, otherwise if $a \gtrdot b$ place on edge from the **group of $\mathbf{x_a}$** to that of $\mathbf{y_b}$;
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of x_a and y_b respectively.

Example 5.20.10

Consider the following

		y_{id}	y_+	y_*	$y_\$$
		id	**+**	*****	**$**
x_{id}	**Id**		$\gtrdot$	$\gtrdot$	$\gtrdot$
x_+	**+**	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
x_*	*****	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$x_\$$	**$**	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$

Columns represents function $\mathbf{y_a}$ and Row represents function $\mathbf{x_a}$

Using the algorithm leads to the following graph :

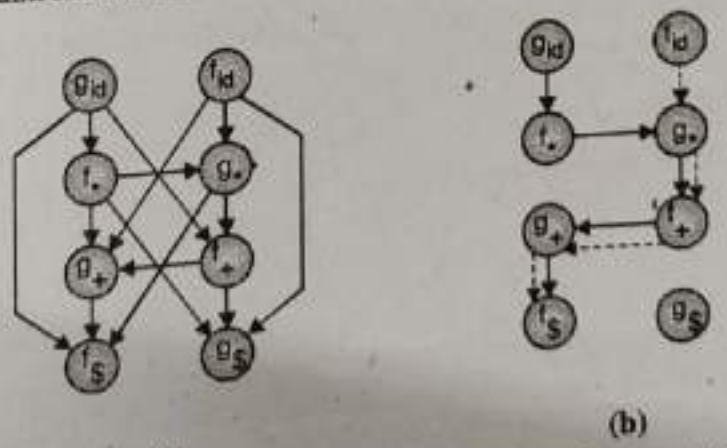


Fig. 5.20.7 : Graph based on above precedence relations

Scanned by CamScanner

	Id	+	*	$
X	4	2	4	0
Y	5	1	3	0

From which we extract the following precedence functions :

The precedence function is calculated as :

For e.g. x_{id} – precedence function = 4

It is calculated by taking the longest path from x_{id} to $x_\$$ or $y_\$$. In the above Fig. 5.20.7(b) the dotted line shows the longest path from x_{id} to $x_\$$ and solid line shows the longest path for y_{id} to $y_\$$. Similarly the longest path is calculated for every functions.

5.20.2(F) Advantages and Disadvantages of Operator Precedence Parsing

Disadvantages

- It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
- Small class of grammars.
- Difficult to decide which language is recognized by the grammar.

Advantages

- Simple
- Powerful enough for expressions in programming languages

5.20.2(G) Error Recovery In Operator-Precedence Parsing

Error Cases

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side.

Error recovery

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle "looks like" which right hand side. And tries to recover from that situation.

Handling shift/reduce errors

To take care of such type of errors we must modify :

1. Stack
2. Input or
3. Both

Note : It should be noted that we don't get into an infinite loop during shift/reduce errors.

➢ **Example 5.20.11**

Consider the table mentioned in Example 5.20.9 the empty entries in that table is filled with error entries.

Scanned by CamScanner

	Id	(	)	$
id	error3	error3	·>	·>
(	<·	<·	=·	error4
)	error3	error3	·>	·>
$	<·	<·	error2	error1

Fig. 5.20.8 : Matrix with error entries

error1 : Missing Operand
error2 : Unbalanced Right Parenthesis
error3 : Missing Operator
error4 : Missing Right Parenthesis

5.20.3 LR Parsers

Q. Explain the LR parser. Write an algorithm for it. Show the working of this algorithm with an example. **(10 Marks)**

In previous sections we have found the handle to reduce or shift the given grammar. For a given grammar we know when we have to found a handle by

1. Analyzing the grammar beforehand
2. Building tables
3. Look ahead in the input.

LR parsing is a bottom up parsing technique which can be used to parse a large class of context free grammars. An LR parser uses the shift-reduce technique discussed earlier. LR parsers read their input from left to right and produce a rightmost derivation LR (k) stands for.

L : It stands for left-to-right scanning of the input

R : It stands for constructing rightmost derivation in reverse

K : It stands for the numbers of input symbols of look ahead which are used to make parsing decisions.

An LR parser is said to perform bottom-up-parsing because it attempts to reduce the top level grammar productions by building up from the leaves. These are the most powerful of all deterministic parsers in practice. The three representative of this family are.

1. **SLR**

Simple LR parsing. It is easy to implement, but is less powerful than the other parsing methods. It uses LR (0) items i.e. it has no look ahead symbol.

2. **Canonical LR parser**

It is the most general and powerful, but is tedious and costly to implement. It uses LR (1) items, i.e. it uses one look ahead symbol.

3. **LALR parser**

It stands for look ahead LR. It is a combination of SLR and canonical LR and can be implemented efficiently. It contains the same number of states as the simple LR parser for the same grammar.

Scanned by CamScanner

LR parsing is attractive for a variety of reasons.

(1) LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.

(2) It is the most general non-back-tracking shift-reduce parsing method.

(3) The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

(4) An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

(5) It can be implemented in a very efficient manner.

The principal drawback of the method is that it is too much work to construct an LR parser by hands for a typical programming language grammar. After discussing the operation of a LR parser, we present three techniques for constructing a LR parsing table for grammar. The three methods used are

(1) Simple LR parser (SLR)

(2) Canonical LR parser

(3) Look ahead LR (LALR) parser

5.20.3(A) The LR Parsing Algorithm

Q. Explain the LR parser. Write an algorithm for it. Show the working of this algorithm with an example. (10 Marks)

The LR parser is a recursive program that can be proven correct by direct computation and can be implemented more efficiently as a recursive descent parser. The schematic form of an LR parser is shown in Fig. 5.20.9.

The LR parser consists of :

1. An input buffer.
2. A stack to store the list of states.
3. A goto table that suggests to which new state it should move forward.
4. An action table that gives a grammar rule to apply given the current state and the current terminal in the input stream.
5. A set of context free language rules.

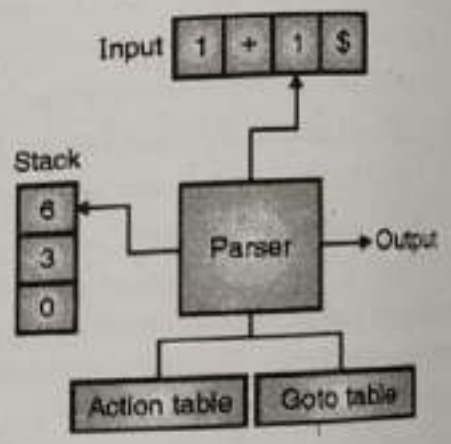


Fig. 5.20.9 : Architecture of an LR parser

LR parsing algorithm works by creating a *leftmost reduction* of the input. It reads input from left to right and uses reductions instead of derivations to process the input. To generate the rightmost derivation it reversed the end result obtained by the leftmost reductions.

The LR parsing algorithm works as follows :

1. The top of the stack will always be the current state and is initialized with [0].
2. Action table is used for the action to be taken when the current state and the current terminal on the input stream is given. There are four cases used in the action table :

Scanned by CamScanner

A shift action (sn)

- In shift action the present terminal is removed from the input stream and state n is pushed onto the stack and it becomes the new present state.

A reduce action (rm)

- The number m is written to the output stream.
- The symbol m mentioned in the right-hand side of rule m says that state is removed from the stack (i.e. if the right-hand side of rule m consists of 2 symbols then 2 top states are removed from the stack).
- The symbol m mentioned in the left-hand side of rule m says that anew state is looked up in the goto table and made the new current state by pushing it onto the stack.

an accept : the string is accepted

no action : a syntax error is reported

Step 2 is repeated until either the string is accepted or a syntax error is reported.

The function goto takes a state and grammar symbol as arguments and produce a state. We shall see that goto function of a parsing table constructed from a grammar G using the SLR, canonical LR, or LALR method is the transition function of a deterministic finite automata that recognizes the viable prefixes of G. Viable prefixes of G are those prefixes of right sentential forms that can appear on the stack of a shift reduce parser, because they do not extend past the right-most handle.

5.20.3(B) Construction of Action and Goto Table/Parser Table

We construct the parse table as follows :

1. The columns for non terminals are copied to the goto table
2. The columns for the terminals are copied to the action table as shift actions.
3. An extra column for '$' (end of input) is added to the action table that contains accept for every item set that contains $S \rightarrow E$.
4. If an item set i contain an item of the form $A \rightarrow w$ and $A \rightarrow w$ is rule m with $m > 0$ then the row for state i in the action table is completely filled with the reduce action rm.

5.20.3(C) Rightmost Parsers

In this section we will see the working of the LR. In LL (1) parsers by looking at the first terminal of the right hand side we can decide which rule is to be applied. Both LL parsers and LR parsers creates a rightmost derivation. LR parsers uses the bottom up derivation in reverse order. On the other hand LR parsers are not able to decide which rule to apply until the entire right hand side of the rule has been parsed. Stacks are used by the right most parsers. The input is put onto the stack, until the entire right hand side of a rule is scanned. Then "reduce" method is applied for, reducing the right-hand-side of the

Consider the following grammar.

$$\begin{aligned} \text{Terminal} &= \{a, b\} \\ \text{Non-terminals} &= \{S', S\} \\ \text{Rules} &= (1)\ S' \rightarrow S\ \$ \\ &\quad\ (2)\ S \rightarrow AB \\ &\quad\ (3)\ A \rightarrow a \\ &\quad\ (4)\ B \rightarrow b \end{aligned}$$

Scanned by CamScanner

Start symbol = S

In the above grammar :

$ → End-of-file

This grammar is not LL (1) we can tell it by creating a parse table which has duplicate entry like

	a	B
S'	S' → S $	S' → S $
S	S → AB A→a	S → AB B → b
A	A→ a	
B		B → b

Now we will see how LR parser handles this grammar for the string "ab". We will start with an empty stack.

Stack	Input	
<empty>	ab $	(Shift initial input token to the top of the stack)
a	b $	Shift a
A	b $	Reduce (A → a) by rule 3
Ab	$	Shift b
AB	$	Reduce (B → b) by rule 4
S		Reduce (S → AB) by rule 2

Once we shift the end-of-file token $ the stack contains the right hand side of the initial rule. Consider the grammar of Fig. 5.20.10.

Terminals = {id, +, *}
Non-Terminals = {E, T}
= (0) E' → E$
(1) E → E + T
(2) E → E*T
(3) E→ T
(4) T → T * id / id
Start symbol = E

Fig. 5.20.10 : A Context Free Grammar

Stack	Input	
<empty>	id * id+ id*id $	(Shift id)
Id	* id+ id*id $	(By rule 4 reduce T → id)
T	* id+ id*id $	(By rule 3 reduce E → T)
E	* id+ id*id $	(shift *)
E *	id+ id*id $	(Shift id)
E * id	+ id*id $	(By rule 4 reduce T → id)

Scanned by CamScanner

Stack	Input	
E *T	+ id*id $	By rule (2) we reduce
E	+ id*id $	(Shift +)
E+	id*id $	(Shift id)
E+id	*id $	(By rule 4 reduce T → id)
E+T	*id $	By rule (1) we reduce
E	*id	(shift *)
E*	id$	(Shift id)
E*id	$	(By rule 4 reduce T→id)
E*T	$	By rule (2) we reduce
E$		
E'	<empty>	We can now shift the $, and reduce by rule (0) E' → E$

The input string is empty, and the initial non-terminal of the grammar is on the top of the stack, so the string is in the language defined by the grammar.

If we follow the reduction in the previous example in reverse order we get,

$$
\begin{aligned}
E' &\Rightarrow E\,\$ \\
&\Rightarrow E + T\,\$ \\
&\Rightarrow E * T + T\,\$ \\
&\Rightarrow E * T + T * id\,\$ \\
&\Rightarrow E * T + id * id\,\$ \\
&\Rightarrow E * id + id * id\,\$ \\
&\Rightarrow id * id + id * id\,\$
\end{aligned}
$$

In this the contents of the stack and the input left to process, combine to form right sentential form.

5.20.3(D) LR Parse Tables

An LR parser is a method of parsing for context free grammar. LR parsers read their input from left to right and produce a rightmost derivation. The parser has an input buffer, a stack on which it keeps a list of states it has been in, an action table and a goto table that tell it to what new state it should move or which grammar rule it should use given the state it is currently in and the terminal or non-terminal it has just read on the input stream.

The action table is indexed by a state of the parser and a terminal and contains three types of actions :

1. **shift**, written as 'sn' and indicates the next state in n.
2. **reduce**, written as 'rm' it indicates that a reduction with grammar rule m should be performed.
3. **accept**, it indicates that the parser accepts the string
4. The goto table is used to indicate what the next state of the parser will be if it has recognized a certain non-terminal.

Scanned by CamScanner

Stack implementation of LR parser

(1) Stack is initialized with [$]. The current state will always be the state that is on top of the stack.

(2) Given the current state and the current terminal on the input stream an action is looked up in the action table. There are four cases :

(a) a shift sn :

- The current terminal is removed from the input stream.
- The state n is pushed onto the stack and becomes the current state.

(b) a reduce rm :

(i) The number m is written to the output stream.

(ii) Remove all the symbols from the stack that matches the RHS of rule (m).

(iii) Given the state that is top of the stack and the LHS of rule m a new state is looked up in the goto table and made a new current state by pushing it onto the stack.

(c) an accept : the string is accepted.

(d) no action : syntax error.

(3) The previous step is repeated until the string is accepted or a syntax error is reported.

Let us take an example mentioned below and the parse table for this grammar is mentioned in Fig. 5.20.11.

Terminals = { +, *,0,1}
Non-Terminals = {E, B}
= (1) E → E * B
(2) E → E +B
(3) E→ B
(4) B→0
(5) B→1
Start symbol = E

State	Action					Goto	
	*	+	0	1	$	E	B
0			S1	S2		G3	G4
1	R4	R4	R4	R4	R4		
2	R5	R5	R5	R5	R5		
3	S5	S6			acc		
4	R3	R3	R3	R3	R3		
5			S1	S2			G7
6			S1	S2			G8
7	R1	R1	R1	R1	R1		
8	R2	R2	R2	R2	R2		

Fig. 5.20.11 : The LR Parse Table

Scanned by CamScanner

Stack Implementation of LR parser shown in Fig. 5.20.11.

Stack	Input Stream	Output Stream	Actions
0	1+1 $		By seeing at row 0, column 1 we have S2. So we shift into state 2 (parser always starts with the initial state)
1 0 2	+1 $		At row 2, column + we have R5. Pop the RHS of rule (5) i.e. (1) from stack
0	+1 $	[5]	As the LHS of rule (5) is a B, push B on top of the stack. As we know top of the stack is 0, and looking at row 0, column B of the table we see the entry G4 i.e. go to state 4. Push B and 4 on the stack
B 0 4	+1 $	[5, 3]	Row (4), column +, we see R3 so pop the RHS of rule (3) B from the stack
0	+1 $	[5, 3]	LHS of rule (3), is E, push E on the stack. As we know top of the stack is state 0, and row 0, column E in the table is G3, push an E and 3 on the stack
E 0 3	+1 $	[5, 3]	At row 3, column + we have S6, so we shift into state 6.
E+ 0 3 6	1 $	[5, 3]	At row 6, column 1 we have S2, so we shift into state 2.
E+1 0362	$	[5, 3,5]	At row 2, column $ we see R5 pop the RHS of rule (5) – 1 from the stack.
E+ 036	$	[5, 3,5]	LHS of rule(5) is B, push B on the stack. As we know top of the stack is 6, and row 6, column B in the table is G8 push a B and 8 on the stack
E+B 0368	$	[5, 3,5]	At row 8 and column $ we see R2, so pop the RHS of rule (2) (E + B) from the stack
0	$	[5, 3,5,2]	LHS of rule(2) is E push E on the stack. As we know top of the stack is 0, and row 0, column E in the table is G3 push a E and 3 on the stack
E 0 3	$	[5, 3,5,2]	Row 3, column $ is "accept" so we accept

The rule numbers that we have written in the output stream column is [5,3,5,2] which is indeed a right most derivation of the string "1+1" in reverse.

Scanned by CamScanner

Let see another example.

Stack	Input Stream	Output Stream	Action
0	1+1+1$		S2
1 0 2	+1+1$	[5]	R5
B 0 4	+1+1$	[5,3]	R3
E 0 3	+1+1$	[5,3]	S6
E + 0 3 6	1+1$	[5,3]	S2
E + 1 0 3 6 2	+ 1 $	[5,3,5]	R5
E + B 0 3 6 8	+ 1 $	[5,3,5,2]	R2
E 0 3	+ 1 $	[5,3,5,2]	S6
E + 0 3 6	1 $	[5,3,5,2]	S2
E + 1 0 3 6 2	$	[5,3,5,2,5]	R5
E + B 0 3 6 8	$	[5,3,5,2,5,2]	R2
E 0 3	$		ACCEPT

The rule numbers that we have written in the output stream column is [5,3,5,2,5,2] which is indeed a right most derivation of the string "1+1+1" in reverse.

[5,3,5,2,5,2] => Start from right side i.e. from rule (2)

E → E+B (Rule 2)

E → E+1 (Rule 5)

E → E+B+1 (Rule 2)

E → E+1+1 (Rule 5)

E → B+1+1 (Rule 3)

E → 1+1+1 (Rule 5)

Hence we get the string "1+1+1" from the grammar mentioned in Fig. 5.20.11.

Scanned by CamScanner

Construction of LR(0) Parsing Tables

Augmented Grammar

The grammar is always augmented with an extra rule

(0) S → E

Where S is a new start symbol and E the old start symbol. The parser uses this rule for reduction exactly when it has accepted the input string.

So our new grammar is

(0) S → E
(1) E → E * B
(2) E → E +B
(3) E→ B
(4) B→0
(5) B→1

Items

The construction of LR parsing table is generated with the help of the LR(0) items. These items are created with the grammar rules with a special dot added somewhere in the RHS.

The grammar rule S → S +E represents the four corresponding items

S → .S+E
S → S.+E
S → S+.E
S → S+E.

Rules of the form A→ ε have only single item A→. .These rules are used to denote the state of the parser.

Item Sets

The set of items is used to depict the state of the parser. As it is usually not possible to find the state of the parser with a single item because it may not know in advance which rule it is going to use for the reductions. Therefore we will characterize the state of the parser by a set of items.

Closure of Item Sets

If there exists a grammar rule of the form S→a.Eb in an item set and there is another grammar rule of the form E→b then the item E→.b should also be in the item set. Such type of extension of grammar rule is known as the closure of an item set and written as closure(I) where I is an item set.

An item with a dot in front of a nonterminal, such as E→E+. B, indicates that the parser expects to parse the nonterminal B next. It must include all the items describing how B itself will be parsed.

Finding the reachable item sets and he transitions between them:

The transitions between the closed item sets will be determined as if we are considering a finite automaton that can read terminals as well as non-terminals. The start state of this automaton is always represented as the closure of the first item of the rule. It is denoted as :

Scanned by CamScanner

Item Set 0:

S→.E

E→.E*B

E → .E+B

E →.B

B → .0

B → .1

Item Set 1 : For terminal '0'

B → 0.

Item Set 2 : For terminal '1'

B → 1.

Item Set 3 : For nonterminal 'E'

S → E.

E → E.*B

E → E.+B

Item Set 4 : For nonterminal 'B'

E → B.

For the item sets 1,2,4 there will be no transitions since the dot in front of any symbol. For item set 3 we see that the dots are in the front of terminals '*' and '+'.

Item Set 5 : For terminal '*'

E → E*.B

B → .0

B → .1

Item Set 6 : For terminal '+'

E → E+.B

B → .0

B → .1

From the item set 5 the transitions for B are

Item Set 7 : For nonterminal 'B'

E → E*B.

Item Set 8 : For nonterminal 'B'

E → E+B.

These final item sets have no symbols behind their dots so no more new item sets are added and we are in finished state. The transition table for automaton looks like this in this we fill the entries by seeing that from given item set we are reaching to which item set. For example for Item set 0 we are having the transitions to item set 1 and item set 2 so we have fill 1 and 2 in row of item set 0 for terminals 0 and 1 of action table. These entries are filled because no more new item sets are found for B → .0 and B → .1. This way we fill the full table.

Scanned by CamScanner

Item Set	Action					Goto	
	*	+	0	1	$	E	B
0			1	2		3	4
1							
2							
3	5	6					
4							
5			1	2			7
6			1	2			8
7							
8							

Construction of Parse Table

1. The goto table is filled with the columns for non-terminals.
2. The action table is copied with the columns for the terminals as shift actions.
3. The action table has an extra column for '$' is added that contains accept state for every item set that contains $S \rightarrow E$.
4. If an item set i contains an item of the form $A \rightarrow w.$ and $A \rightarrow w$ is rule m with m>0 then the row for state i in the action table is completely filled with reduce action rm.

State	Action					Goto	
	*	+	0	1	$	E	B
0			S1	S2		G3	G4
1	R4	R4	R4	R4	R4		
2	R5	R5	R5	R5	R5		
3	S5	S6			Acc		
4	R3	R3	R3	R3	R3		
5			S1	S2			G7
6			S1	S2			G8
7	R1	R1	R1	R1	R1		
8	R2	R2	R2	R2	R2		

Algorithm for constructing LR parse Table

Now we will consider the simplest LR parsers – LR (0)

LR (0) stands for

L → Left-to-Right

R → Rightmost Derivation

0 → With Zero Look-ahead

Scanned by CamScanner

LR parsing generates the rightmost derivation, but since it works from the bottom up, it reduces sequence of tokens on the left hand side first. But when we read from the top, the derivation sequence appears to expand rightmost non-terminals first.

(1) The stack is empty and we are looking to reduce the rule by $S' \rightarrow S\ \$$.

(2) Using a "•" in the rule represents how much of the rule is already on the stack

(3) A dotted item, or simply, item, is a production rule with a dot in it to indicate how much of the RHS has so far been recognized. The closure of an item is used to see which production rules could be used to extend the current structure. It is calculated as follows:

a) From the given grammar rules add the first item itself to the first closure set.

b) If there exists an item in the closure of the form $A \rightarrow \alpha . \beta \gamma$ where the next symbol after the dot is a nonterminal, add the production rules of that symbol where the dot is before the first item (e.g., for each rule β:- γ, add items: β :-.γ).

c) Repeat steps b and c for any new items added under (b).

Example : Given the grammar

1. E:- E $ 2. E :- T 3. E :- E + T

4. T :- id 5. T :- (E)

Closure set for

E:- . E $:

E:- . E $ (by rule a)

E :- . T (by rule b)

E :- . E + T (by rule b)

T :- . id (by rule c, b)

T :- . (E) (by rule c, b)

Consider the following grammar for the LR(0) Parsing

1. E :- T 2. E :- E + T 3. T :- id 4. T :- (E)

Building the DFA

For creating a DFA for an LR (0) parser table two tasks are accomplished

- Construct the closure set of a given state.
- Find transitions from one state to another.

Steps for constructing the DFA for an LR(0) parser are as follows:

1) Augment the grammar with a new state

0. E:- E $

2) Build the node for State S0:

a. Create an item from the rule introduced in (1), with the dot before the first item.

b. Build the closure for an LR(0) state for this rule :

Scanned by CamScanner

1. If there is an any item in the state of the form $S \rightarrow v\ S_0\ w$ where v and w are terminals and S_0 is a non-terminal.
2. For every rule of the form $S_0 \rightarrow v$ add the item $S_1 \rightarrow .v$ if it is not already there.
3. Repeat step 1 and 2 until all the item are added.

For example

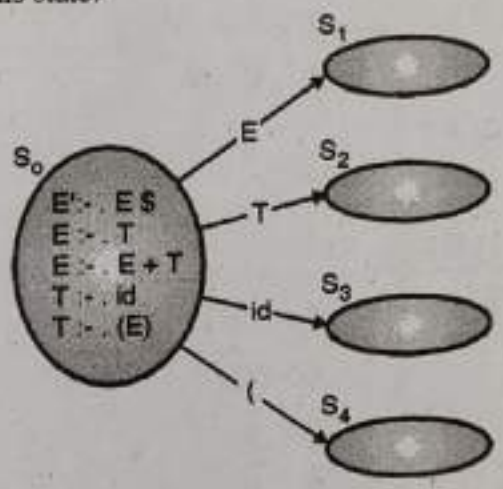
E':- . E $
E :- . T
E :- . E + T
T :- . id
T :- . (E)

c. Generate a node for S0 for this closure set

3) For each symbol which follows a dot in the closure set (S0) :

a. Create a new state for each symbol and link the new state to the original state.

b. Add a DFA node for this state.



c. The closure of the new states is calculated as follows :

i) It contains only those items from the previous state where the recognized symbol was the symbol after the dot.

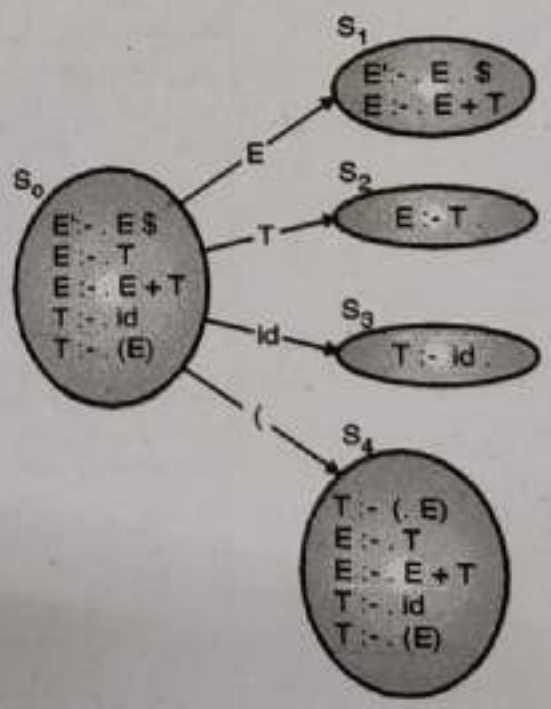
ii) For each of these items, move the dot AFTER the recognised symbols.

iii) Build a new empty state.

iv) If the state is of form $S \rightarrow \beta \cdot b\ \gamma$, where α and γ are terminals and non-terminal add the item $S \rightarrow \beta\ b \cdot \gamma$ to the new state then find the closure of new state.

v) With the help of transition add original state to the new state, labelled with b.

vi) Where the new next symbol is a nonterminal, add the closure of these items to the closure (note in the state S4, we started with just



Scanned by CamScanner

T -> (. E) but as the next symbol was E, we closure of this item added 3 other rules :

4) Repeat step 3 for each of the new states, except where a new state which has the same closure as an existing one, link to that state. The final DFA for the given grammar is mentioned below.

5) If terminal $ (end of file) appears after the "." in some actual state of item, build a transition from the actual state to an "accept" state, transitioning on $.

6) Same steps are repeated for strings of non-terminals.

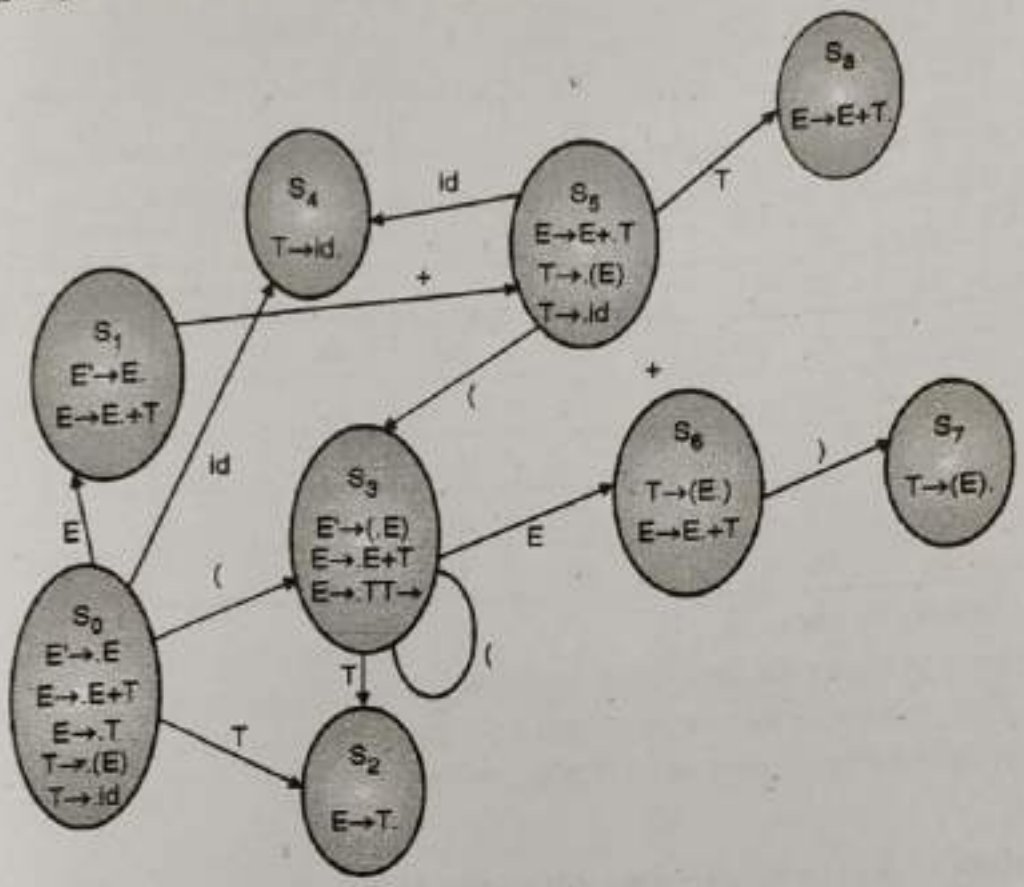
Based on the above algorithm the remaining item sets for the grammar mentioned above are as:

1. E':- E $
2. E :- T
3. E :- E + T
4. T :- id
5. T :- (E)

S_0: E':- .E $ E :- .T E :- .E + T T :- .id T :- . (E)	S_1: E':- E.$ E :- E. + T
S_2: E :- T.	S_3: T :- (. E) E :- .T E :- .E + T T :- .id T :- . (E)
S_4: T :- id.	S_5: E :- E + .T T :- . (E) T :- .id T :- (E.)
S_6: T :- (E.) E :- E. + T	S_7: T :- (E) .
S_8: E :- E + T.	

Scanned by CamScanner

With the help of these items we have drawn DFA:



Steps required in the construction of the parse table

1) In parse table each row represents the state and column is used to enter terminal and nonterminal in the grammar (except for the augmented grammar symbol E').
2) Rows represents state numbers, and the columns are labelled with terminals (including $) and non-terminals.
3) In action table add a column for the each state.
4) For each transition arc between Si and Sj, with label X: enter j in the cell (Si, X) (i.e. enter the end state in the row of the start state under the arc label)
5) For each state with outgoing arcs, enter 'shift' in the action column of the state. Add SP (for "shift to state P") to the table in a row n, column a where P stands for state number, if state n contains a transition labelled with a terminal a to state S,
6) For reduce state, enter 'reduce' in the state's row of the action column, followed by the number of the rule which is completed in this state. Add r <rule number> to the table in row n, in all columns labelled with terminals if state n contains an item with the "." at the end of a rule,
7) In place of the action E' :- E $, put the 'accept' in desired column.
8) Add GP (for "go to state P") to the table in row n, column S, if a state contains a transition labelled with a non-terminal S to a state P,

Scanned by CamScanner

The parse table for the above mentioned grammar is as :

State	Action					Goto	
	id	+	(	)	$	E	T
S_0	S4		S3			G1	G2
S_1		S5			Accept		
S_2	R2	R2	R2	R2	R2		
S_3	S4		S3			G6	G2
S_4	R4	R4	R4	R4	R4		
S_5	S4		S3				G8
S_6		S5		S7			
S_7	R5	R5	R5	R5	R5		
S_8	R3	R3	R3	R3	R3		

Parse Table : Initially we have :

1. Input vector: a vector for the input tokens, terminated by $.
2. Token pointer: pointer for the next input token to be processed.
3. The Stack : initially first closure set S0 is placed on the stack.

Alternatively

1. **Action Table :** Based on the entry done in the action table for the current state apply the action on the topmost state in the stack.

   a. If the action mentioned in the column is shift, move the next token onto the top of the stack, and move the pointer to the next token.

   b. If action is reduce, look up the rule given in the action, and remove n*2 items from the stack (where n is the number of symbols on the RHS of the rule). Then place the LHS of the production on the stack.

   c. If action is accept, then finish parsing.

2. **Goto State table :** If top of the stack contains a state and a terminal or nonterminal then in the Goto table, look up the row for the state, and the column symbol.

   a. If there is a number there, put this number on the top of the stack. It represents the new state.

   b. If there is no Goto, then return ERROR.

➢ **Example 5.20.12**

Consider a non LR (0) grammar with a shift-reduce conflict is :

1. $E \rightarrow 1E$
2. $E \rightarrow 1$

Scanned by CamScanner

One of the item sets we then find is :

$E \rightarrow 1 \cdot E$

$E \rightarrow 1 \cdot$

$E \rightarrow E \cdot$

There is a shift reduce conflict in this item set because in the cell in the action table for this item set and the terminal '1' there will be both a shift action to state 1 and a reduce action with rule 2.

Example 5.20.13

Consider a non LR (0) grammar with a reduce-reduce conflict as.

1. $E \rightarrow A1$
2. $E \rightarrow B2$
3. $A \rightarrow 1$
4. $B \rightarrow 1$

In this case we obtain the following item set

$A \rightarrow 1 \cdot$

$B \rightarrow 1 \cdot$

There is a reduce-reduce conflict in this item set because in the cells in the action table for this item set there will be both a reduce action for rule 3 and are for rule 4.

Example 5.20.14

(1) $S X \$$

(2) $X \rightarrow Y/id$

(3) $Y \rightarrow id$

The two item sets for the following grammar are as

I0 $S \rightarrow \cdot X \$$

$X \rightarrow \cdot Y$

$X \rightarrow \cdot id$

$Y \rightarrow \cdot id$

I1 $X \rightarrow id \cdot$

$Y \rightarrow id \cdot$

State I1 has reduce-reduce conflict.

Example 5.20.15

1. $S \rightarrow E \$$
2. $E \rightarrow L = R/R$
3. $L \rightarrow * R/id$
4. $R \rightarrow L$

Scanned by CamScanner

The item sets for the following grammar are as

I0 $E \rightarrow \cdot E\$$

$E \rightarrow \cdot L = R/R$

$E \rightarrow \cdot * R$

$R \rightarrow \cdot L$

$E \rightarrow \cdot id$

I1 $E \rightarrow L \cdot = R \mid R$

$R \rightarrow L \cdot$

It has shift-reduce conflict in I1

➢ **Example 5.20.16**

1. $S \rightarrow Aa\ Ab$
2. $S \rightarrow Bb\ Ba$
3. $A \rightarrow \varepsilon$
4. $B \rightarrow \varepsilon$

(i) Augmented grammar (0) $S' \rightarrow S$

(1) $S \rightarrow AaAb$

(2) $S \rightarrow BbBa$

(3) $A \rightarrow E$

(4) $B \rightarrow E$

(ii) Set of items

I0 ⇒ $S' \rightarrow \cdot S$

$S \rightarrow \cdot Aa\ Ab$

$S \rightarrow \cdot Bb\ Ba$

$A \rightarrow \cdot \varepsilon$

$B \rightarrow \cdot \varepsilon$

I1 ⇒ $S' \rightarrow S \cdot$

I2 ⇒ $S \rightarrow A \cdot a\ Ab$

I3 ⇒ $S \rightarrow B \cdot bBa$

I4 ⇒ $S \rightarrow Aa \cdot Ab$

I5 ⇒ $S \rightarrow Bb \cdot Ba$

I6 ⇒ $S \rightarrow AaA \cdot b$

I7 ⇒ $S \rightarrow BbB \cdot a$

I8 ⇒ $S \rightarrow Aa\ Ab \cdot$

I9 ⇒ $S \rightarrow Bb\ Ba \cdot$

Parsing table

	INPUT/ACTION TABLE			GO TO TABLE		
	A	b	$	S	A	B
I0	R3/R4	R3/R4				
I1			Accept			

Scanned by CamScanner

	INPUT/ACTION TABLE			GO TO TABLE		
	A	b	$	S	A	B
I2	S4					
I3		S5				
I4	R3	R3			G6	
I5	R4	R4				G7
I6		S8				
I7	S9					
I8						
I9						

In this it shows the reduce-reduce conflict in state I0.

Example 5.20.17

1. $S \rightarrow E\,S$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id$
5. $T \rightarrow (E)$

Above grammar has the following item sets.

I0 ⇒
$S \rightarrow \cdot E\,\$$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot id$
$T \rightarrow \cdot (E)$

I1 ⇒
$S \rightarrow E \cdot \$$
$E \rightarrow E \cdot + T$

I2 ⇒
$S \rightarrow E\,\$ \cdot$

I3 ⇒
$E \rightarrow E + \cdot T$
$T \rightarrow \cdot id$
$T \rightarrow \cdot (E)$

T4 ⇒
$E \rightarrow E + T \cdot$

I5 ⇒
$T \rightarrow id \cdot$

I6 ⇒
$T \rightarrow (\cdot E)$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot id$
$T \rightarrow \cdot (E)$

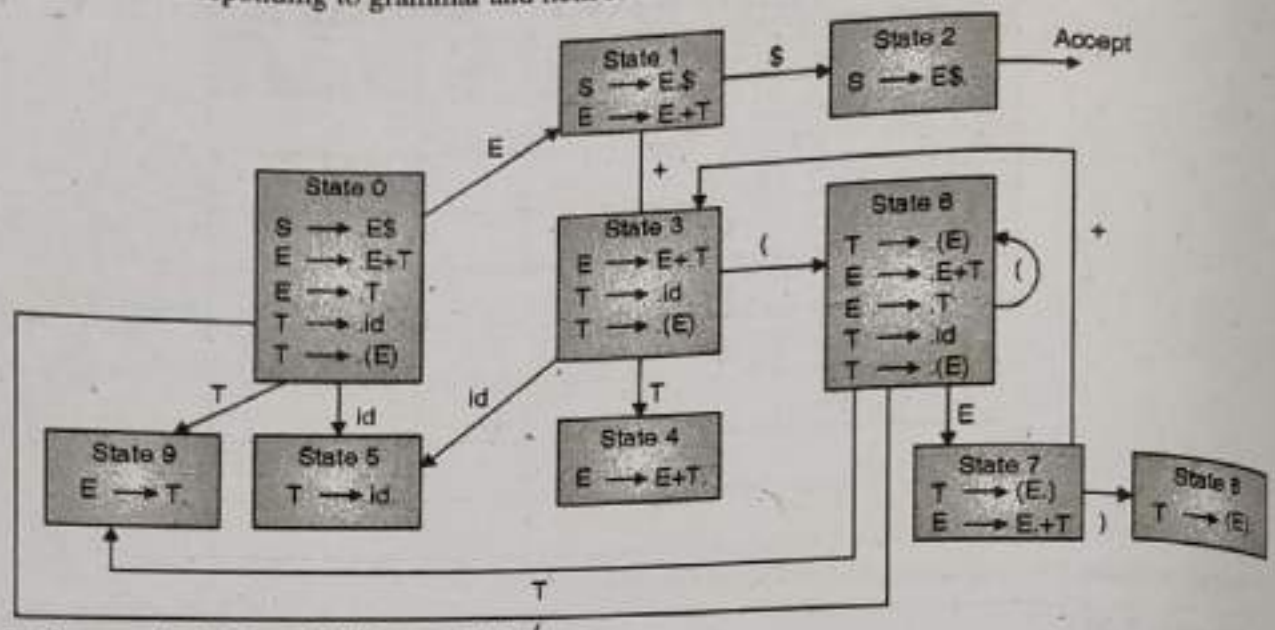
I7 ⇒
$T \rightarrow (E \cdot)$
$E \rightarrow E \cdot + T$

I8 ⇒
$T \rightarrow (E) \cdot$

I9 ⇒
$E \rightarrow T \cdot$

Scanned by CamScanner

DFA corresponding to grammar and item sets as.



Action and Go To table/Parse table

State	Action					Go To		
	id	(	)	+	$	S	E	T
0	S5	S6					G1	G9
1				S3	S2			
2					Accept			
3	S5	S6						G4
4	r2	r2	r2	r2	r2			
5	r4	r4	r4	r4	r4			
6	S5	S6					G7	G9
7			S8	S3				
8	r5	r5	r5	r5	r5			
9	r3	r3	r3	r3	r3			

➢ **Example 5.20.18**

(0) $S' \rightarrow S\$$

(1) $S \rightarrow BB$

(2) $B \rightarrow aB$

(3) $B \rightarrow C$

The following grammar has the item set as

I0 ⇒ $S' \rightarrow \cdot S\$$
$S \rightarrow \cdot BB$
$B \rightarrow \cdot aB$
$B \rightarrow \cdot C$

I4 → $S \rightarrow BB \cdot$

I5 → $B \rightarrow a \cdot B$
$B \rightarrow \cdot aB$
$B \rightarrow \cdot C$

Scanned by CamScanner

I1 → $S' \to S \cdot \$$

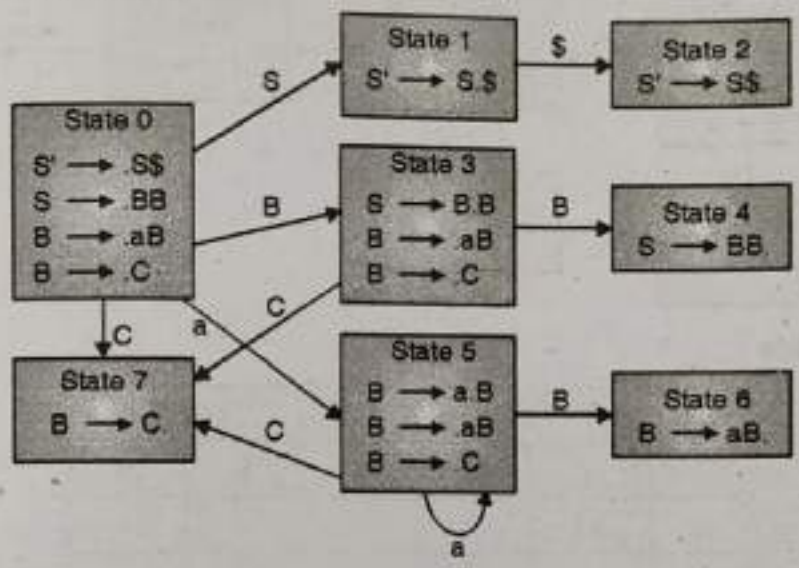
I2 → $S' \to S\$ \cdot$

I3 → $S \to B \cdot B$
$B \to \cdot aB$
$B \to \cdot C$

I6 → $B \to aB \cdot$

I7 → $B \to C \cdot$

DFA for the above grammar is



Parse table for the above grammar is as

State	Action			Go To		
	A	C	$	S'	S	B
0	S5	S7			G1	G3
1			S2			
2			accept			
3	S5	S7				G4
4	r1	r1	r1			
5	S6	S7				G6
6	r2	r2	r2			
7	r3	r3	r3			

Example 5.20.19

Consider the grammar

(1) $S \to R\,\$$

(2) $R \to Rb$

(3) $R \to a$

Scanned by CamScanner

Item sets

I0 ⇒ $S \rightarrow \cdot R\$$

$R \rightarrow \cdot Rb$

$R \rightarrow \cdot a$

I1 ⇒ $S \rightarrow R \cdot \$$

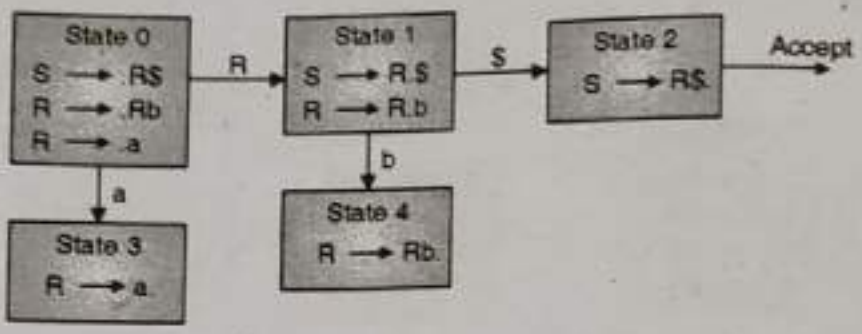
$R \rightarrow R \cdot b$

I2 ⇒ $R \rightarrow R\$ \cdot$

I3 ⇒ $R \rightarrow a \cdot$

I4 ⇒ $R \rightarrow Rb \cdot$

DFA



Action and goto table

State	Action			Go To	
	a	b	$	S	R
0	S3				G1
1		S4	S2		
2			accept		
3	r3	r3	r3		
4	r2	r2	r2		

➢ **Example 5.20.20**

Consider the following grammar and find item sets, DFA and parse table

(1) $S' \rightarrow E$

(2) $E \rightarrow E + T$

(3) $E \rightarrow T$

(4) $T \rightarrow id$

(5) $T \rightarrow (E)$

(6) $T \rightarrow id\ [E]$

Item sets.

I0 → $S' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot (E)$

$T \rightarrow \cdot id$

$T \rightarrow \cdot id\ [E]$

I6 $T \rightarrow (E \cdot)$

$E \rightarrow E \cdot + T$

I7 $T \rightarrow (E) \cdot$

I8 $E \rightarrow E + T \cdot$

Scanned by CamScanner

I1 →
$S' \to E \cdot$
$E \to E \cdot + T$

I2 →
$E \to T \cdot$

I3 →
$T \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot (E)$
$T \to \cdot id$
$T \to \cdot id [(E)]$

I4 →
$T \to id \cdot$
$T \to id \cdot [E]$

I5 →
$E \to E + \cdot T$
$T \to \cdot (E)$
$T \to \cdot id$
$T \to \cdot id [E]$

I9
$T \to id [\cdot E]$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot (E)$
$T \to \cdot id$
$T \to \cdot id [E]$

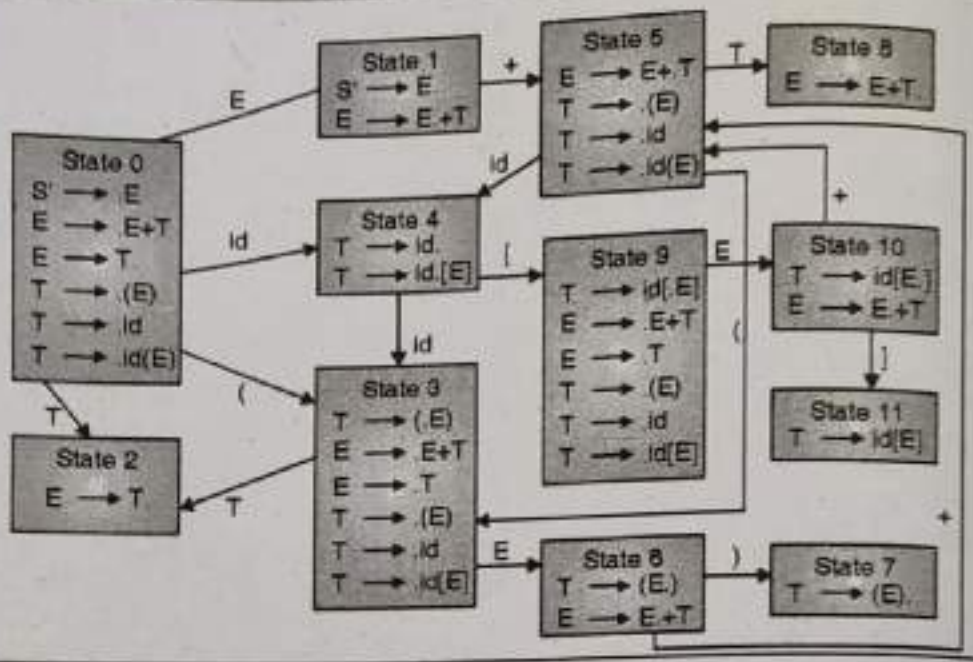
I10
$T \to id [E \cdot]$
$E \to E \cdot + T$

I11
$T \to id [E] \cdot$

DFA

State	Action						Go To		
	+	Id	(	)	[	]	S	E	T
0		S4	S3					G1	G2
1	S5								
2	r3	r3	r3	r3	r3	r3			
3								G6	G2
4		S3			S9				
5		S4	S3						G8
6	S5			S7					
7	r5	r5	r5	r5	r5	r5			
8	r2	r2	r2	r2	r2	r2			
9								G10	
10	S5					S11			
11	r6	r6	r6	r6	r6	r6			

Scanned by CamScanner



➢ **Example 5.20.21**

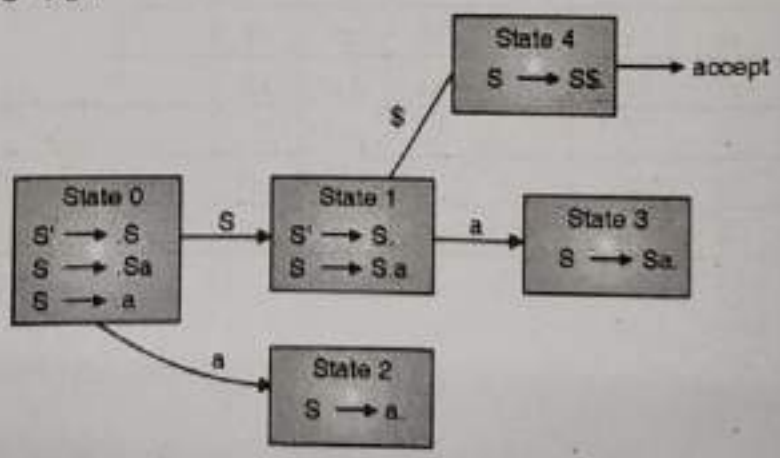
Find only item sets and DFA for the following grammar.

1. $S' \rightarrow S\,\$$
2. $S \rightarrow Sa$
3. $S \rightarrow a$

Item sets

I0 →	$S' \rightarrow \cdot S$	I3	$S \rightarrow Sa \cdot$
	$S \rightarrow \cdot Sa$		
	$S \rightarrow \cdot a$	I4	$S \rightarrow S\,\$ \cdot$
I1 →	$S' \rightarrow S \cdot$		
	$S \rightarrow S \cdot a$		
I2 →	$S \rightarrow a \cdot$		

DFA



Scanned by CamScanner

Parse Table

State	Action		Go To	
	a	$	S	R
0	S2			G1
1	S3	accept		
2	R3	R3		
3	R2	R2		

Example 5.20.22

Consider the following example

1. S → E $
2. E → (L)
3. E → ( )
4. E → id
5. L → L, E
6. L → E

Item sets

I0 →
S → · E $
E → · (L)
E → · ( )
E → · id
L → · L, E
L → · E

I1 → S → E · $

I2 S → E $ ·

I3
E → (· L)
E → (·)
L → · L, E
L → · E
L → · (L)
E → · ( )
E → · id

I4
E → (L ·)
E → L · , E

I5 E → (L) ·

I6
L → L, · E
L → · (L)
E → · ( )
E → · id

I7 L → L, E ·

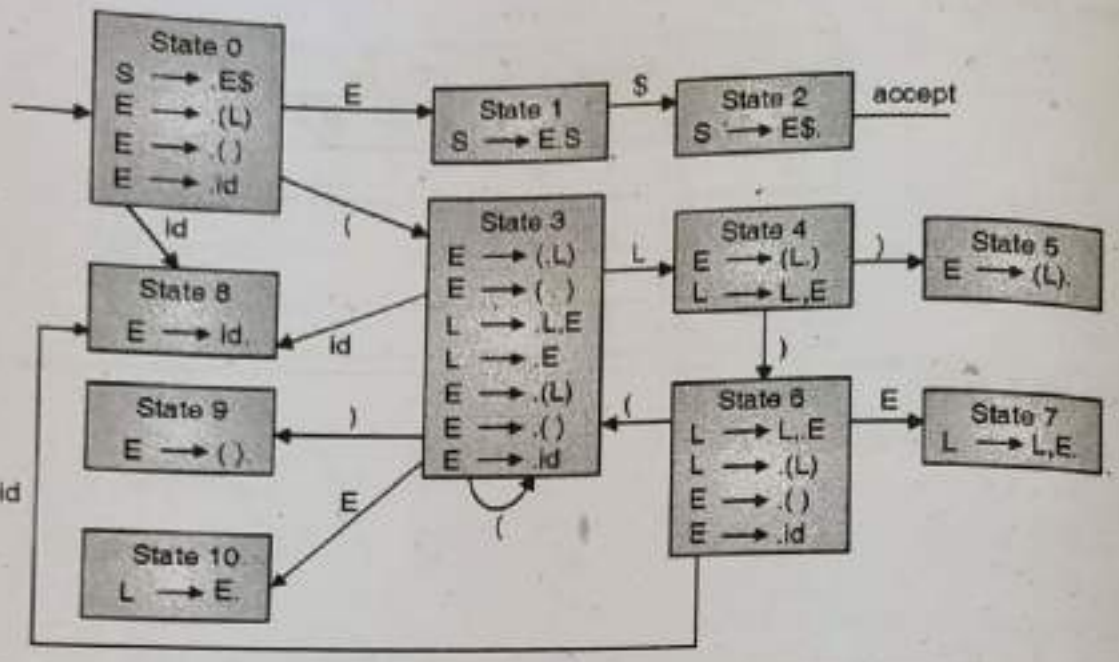
I8 E → id ·

I9 E → ( ) ·

I10 L → E ·

Scanned by CamScanner

DFA



Example 5.20.23 : Consider the following grammar :

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

(i) Frame the transition table and action / goto table of the given grammar.

(ii) Demonstrate if the grammar is LR(0) or not. **MU - Dec. 2014, 10 Marks**

Solution :

I0 :	$S' \rightarrow \cdot S$
	$S \rightarrow \cdot aSbS$
	$S \rightarrow \cdot bSaS$
	$S \rightarrow \cdot$

I1 :	$S' \rightarrow S \cdot$

I2 :	$S \rightarrow a \cdot SbS$
	$S \rightarrow \cdot aSbS$
	$S \rightarrow \cdot bSaS$
	$S \rightarrow \cdot$

I3 :	$S \rightarrow b \cdot SaS$
	$S \rightarrow \cdot aSbS$
	$S \rightarrow \cdot bSaS$
	$S \rightarrow \cdot$

I4 :	$S \rightarrow aS \cdot bS$

I5 :	$S \rightarrow bS \cdot aS$

I6 :	$S \rightarrow aSb \cdot S$
	$S \rightarrow \cdot aSbS$
	$S \rightarrow \cdot bSaS$
	$S \rightarrow \cdot$

I7 :	$S \rightarrow bSa \cdot S$
	$S \rightarrow \cdot aSbS$
	$S \rightarrow \cdot bSaS$
	$S \rightarrow \cdot$

I8 :	$S \rightarrow aSbS \cdot$

I9 :	$S \rightarrow aSbS \cdot$

The Parsing action and goto tables constructed from these sets of items are :

Scanned by CamScanner

State	Action			Goto
	a	B	$	S
0	s2/r (3)	S3/r (3)	r (3)	1
1			accept	
2	s2/r (3)	s3/r (3)	r (3)	4
3	s2/r (3)	s3/r (3)	r (3)	5
4		s6		
5	s7			
6	s2/r (3)	s3/r (3)	r (3)	8
7	s2/r (3)	s3/r (3)	r (3)	9
8	r(1)	r(1)	r(1)	
9	r(2)	r(2)	r(2)	

In the LR(0) grammar parsing table there are multiple shift -reduce conflicts. So the given grammar is not LR(0).

5.20.4 SLR Parser

As we know not all CFG's are LL (1), and LR (0). The grammar in figure is not LL (1), though it was LR (0). Grammar that are not LR (0) requires a more powerful parsing technique. We will now see SLR (1) parsers i.e. simple, left-to-right, rightmost derivation parsers with 1 symbol lookahead SLR (1) parsers are basically LR (0) parsers with a very simple lookahead scheme. LR(0)∈ SLR(1)-A LR parser for which the parsing tables are generated as for a LR(0) parser. SLR parser prevents certain shift-reduce and reduce-reduce conflicts that occur in LR(0) parsers.

Consider a CFG in Fig. 5.20.12 which describes.

Terminals = {0, +, *, $}

Non-Terminals = {S′, S, B}

Rules = (0) S′ → S S

(1) S → S + B

(2) S → B

(3) B → B *0

(4) B → 0

Start symbol = E′

Fig. 5.20.12 : A CFG for simple expression

Scanned by CamScanner

Item sets for the following CFG is as

I1 → S′ → · S$
S → · S + B
S → · B
B → · B * 0
B → · 0

I2 → S′ → S · $
S → S · + B

I3 → S → B ·
B → B · * 0

I4 → B → 0 ·

I5 → S → S + · B
B → · B *0
B → ·0

I6 → B → B * · 0

I7 → S → S + B ·
B → B · *0

I8 → B → B * 0 ·

The final set of states and their transitions are shown in Fig. 5.20.13.

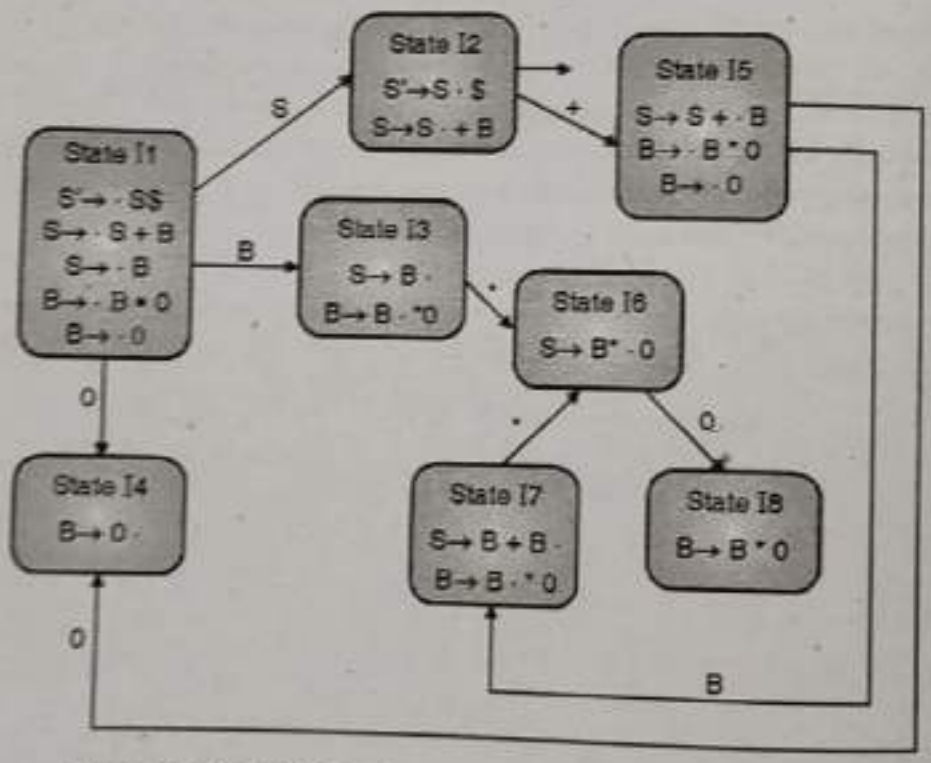


Fig. 5.20.13 : DFA for the grammar in Fig. 5.20.12

Scanned by CamScanner

Creating a parse table from the Fig. 5.20.13 we get

State	Action				Go To	
	0	+	*	$	S	B
1	S4				G2	G3
2		S5		accept		
3	R2	R2	S6, R2	R2		
4	R4	R4	R4	R4		
5	S4					G7
6	S8					
7	R1	R1	S6, R1	R1		
8	R3	R3	R3	R3		

Fig. 5.20.14 : LR (0) parse table for the grammar of Fig. 5.20.12

Note : Note that duplicate entries in the table the grammar is not LR (0)

(1) Adding simple look-ahead to LR (0) parsers

The grammar in Fig. 5.20.12 is not LR (0) because entries are duplicate entries (especially shift reduce conflicts) in the parse table.

- The Shift-Reduce Conflict is caused when the grammar allows a rule to be reduced for particular token, but, at the same time, allowing another rule to be shifted for that same token. This led to an ambiguous grammar as program can be interpreted in two ways whether to shift or reduce for the particular action. It is the most common type of conflict found in grammars and is used by recursive grammar definitions where the system cannot determine when one rule is completed and another is just started.
- A Reduce-Reduce error is a caused when a grammar allows ,two or more different rules to be reduced at the same time, for the same token. This error can be caused when the same rule is reached by more than one path.

These two problem arises because LR (0) parsers reduce, the moment the RHS of a rule appears on the stack. SLR (1) parsers add a little bit of look-ahead information to LR (0) parsers, to avoid some conflicts.

Fig. 5.20.14 shows the shift reduce conflict of row 3, column *. If we reduce by rule (2) (E → T), then the top of the stack will be E. If we now shift *, we will have a problem. E * will be on the top of the stack and * cannot follow E in any partial derivation. Thus we don't want to reduce by rule (2) when the next symbol is *. This reads naturally the formation of SLR (1) parse tables.

(2) Creating SLR (1) parse tables

(A) Create action table

- one row for each item-set which forms a state.
- one column for each terminal including token '$'.

Scanned by CamScanner

(B) Create goto table

- same rows as action.
- column for nonterminal.

(C) For each state n created in step 2 and 3 the ith row, corresponds to the Ith item set, Ii

- If [A → α. aβ] is in Ii, and goto(Ii,a) = Ij then set action[i,a] to shift j,
- If [A → α.] is in Ii, and a is in Follow(A) then set action[i,a] to reduce A → α
- If [S' → S .] is in Ii, then set action[i,$] to accept

In LR (0) parse table , we always reduce when a state contains an item of the form S → γ whereas in SLR (1) parse table , we only reduce when a state contains an item of the form S → γ .and the next input symbol can follow S.

As we can see there is a shift- reduce conflict for state 3 and terminal '*'. However the follow set of B contains {$} so the reduce actions R1 and R2 are only valid in the column of $. Thus the SLR (1) parse table for the grammar of Fig. 5.20.12 is in Fig. 5.20.15.

State	Action				Go To	
	0	+	*	$	S	B
1	S4				G2	G3
2		S5		accept		
3		R2	S6	R2		
4		R4	R4	R4		
5	S4					G7
6	S8					
7		R1	S6	R1		
8		R3	R3	R3		

Fig. 5.20.15 : SLR (1) parse table for the grammar of Fig. 5.20.12

The follow sets of Fig. 5.20.11 are as

FOLLOW (S') = {$}

FOLLOW (S) = {+, $}

FOLLOW (B) = {*, $}

➢ **Example 5.20.24**

Draw the SLR parser table for the following grammar.

(1) E' → E $

(2) E → 1E

(3) E → 1

Constructing the action and goto table as is done for LR (0) parsers would give the following item sets and tables.

Scanned by CamScanner

I0 → $E' \to \cdot E\$$
$E \to \cdot 1E$
$E \to \cdot 1$

I1 → $E \to 1 \cdot E$
$E \to 1 \cdot$

I2 → $E' \to E \cdot$

I3 → $E \to 1E \cdot$

ACTION and GOTO TABLES

State	Action		Go To
	1	$	E
0	S1		G2
1	S1/r3	r3	G3
2		Acc	
3	r2	r2	

As it can be observed that there is a shift-reduce conflict for state 1 and terminal '1'. However, the FOLLOW set of E is {$} so the reduce actions r2 and r3 are only valid in the columns for $. The result are the following conflict less action and goto table.

State	Action		Go To
	1	$	E
0	S1		G2
1	S1	r3	G3
2		Acc	
3		r2	

Example 5.20.25 : Consider the grammar and construct the SLR parser for the given grammar.
$S \to (S)\,S \mid \varepsilon$ **MU - Dec. 2016**

Solution :

Augmenting grammar

1. $S' \to S$
2. $S \to (S)\,S$
3. $S \to \varepsilon$

Find the item sets

I0 → $S' \to \cdot S$
$S \to \cdot (S)\,S$
$S \to \cdot \varepsilon$

I3 → $S \to (S \cdot)\,S$

I4 → $S \to (S) \cdot S$
$S \to \cdot (S)\,S$
$S \to \cdot$

Scanned by CamScanner

I1 → $S' \rightarrow S \cdot$

I2 → $S \rightarrow (\cdot S) S$
$S \rightarrow \cdot (S) S$
$S \rightarrow \cdot$

I5 → $S \rightarrow (S) S \cdot$

And now finally the parsing table is as :

State	Input/Action			Go To
	(	)	$	S
0	S2	R3	R3	G1
1			Accept	
2	S2	R3	R3	G3
3		S4		
4	S2	R3	R3	G5
5		R2	R2	

➢ **Example 5.20.26**

Consider the following grammar and construct the SLR parser for this

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

First of all we augment the grammar as :

Augmented grammar

(0) $E' \rightarrow E$
(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

Find the set of items as

Set of Item ⇒

I0 → $E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

Scanned by CamScanner

I1 → $E' \to E \cdot$
$E \to E \cdot + T$

I2 → $E \to T \cdot$
$T \to T \cdot * F$

I3 → $T \to F \cdot$

I4 → $E \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I5 → $F \to id$

I6 → $E + \cdot T$
$T \to \cdot T * F$

I7 → $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I8 → $F \to (E \cdot)$
$E \to E \cdot + T$

I9 → $E \to E + T \cdot$
$T \to T \cdot * F$

I10 → $T \to T * F \cdot$

I11 → $F \to (E) \cdot$

Parsing table

State	Action/Input						Go To		
	id	+	*	(	)	$	E	T	F
0	S5			S4			G1	G2	G3
1		S6				Accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			G8	G2	G3
5		R6	R6		R6	R6			
6	S5			S4				G9	G3
7	S5			S4					G10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Note : (1) SLR grammar constitute a small subset of CFG, so a SLR parser can only succeed on a small number of CFG.

(2) By comparing the SLR (1) parser with canonical LR parser we find that the CLR parser is more powerful. But the CLR (1) has a greater number of states than the SLR (1) parser, so its storage requirement is also greater than SLR (1) parser.

Scanned by CamScanner

5.20.5 LR (1) Parsers

The parsing table for the canonical LR parser or LR (1) parser is constructed in a similar way as with LR(0) parsers except that the items in the item set also has a follow symbol i.e. a terminal that is expected by the parser after the right hand side of the rule is processed.

A LR(1) item is a production with a marker together with a terminal, for e.g., see [S→xP.Qy, c] Such rule states that how much of a production we have already processed (x, P), what we could expect next(Q ,y), and a look-ahead that agrees with what should follow in the input if we ever reduce by the production S→xP.Qy. The look-ahead of an LR(1) item is used directly only when considering reduce actions (i.e., when the marker is at the right end).

SLR (1) parsers use some look-ahead information. SLR (1) stands for simple LR (1) parsers.

S – Simple, L → Left-to-Right, R – Rightmost derivation with 1 symbol look ahead.

Constructing Canonical LR(1) Parsing Tables

Input : An augmented grammar G'.

Output : The canonical LR parsing table functions action and goto for G'

1. Construct C={I0,I1.....,In}, the collection of sets of LR(1) items for G'.
2. State I of the parser is constructed from Ii. The parsing actions for state I are determined as follows :
   (a) If [A → α. a β, b] is in Ii, and goto(Ii, a) = Ij, then set action[i,a] to "shift j." Here, a is required to be a terminal.
   (b) If [A → α., a] is in Ii, A ≠ S', then set action[i,a] to "reduce A → α."
   (c) If [S'→S.,$] is in Ii, then set action[i ,$] to "accept."
3. Extra information is put into a state by including a terminal symbol as a second component in an item called look-ahead symbol..
4. A LR(1) item is:
   A→α.β,a where **a** is the look-head of the LR(1) item (**a** is a terminal or end-marker)
5. For the given production rule A→ α.β, a if β is not empty, the look-head does not have any affect.
6. When β is empty (A→ α.,a), we do the reduction by A→α only if the next input symbol is a (not for any terminal in FOLLOW(A)).
7. A state will contain $A \rightarrow \alpha.,a_1$ where $\{a_1,...,a_n\} \subseteq FOLLOW(A)$
   $A \rightarrow \alpha.,a_n$
8. The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

   closure(I) is: (where I is a set of LR(1) items)

   Every LR (1) item in I is in closure (I)

   If the rule is of the form A→α.Bβ,a (where α and β are strings of terminals and non-terminals, B is a nonterminal, and a is the look-ahead symbol) in closure(I) and B→γ is a production rule of G; then B→.γ,b will be in the closure(I) for each terminal b in FIRST(β) . If FIRST (β) contain ε then for each rule of the term B → γ add the item $B_1 \rightarrow \cdot \gamma b$, if it is not already there.

Scanned by CamScanner

If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

If $A \rightarrow \alpha.X\beta,a$ in I then every item in **closure**($\{A \rightarrow \alpha X.\beta,a\}$) will be in goto(I,X).

3. The transitions in LR (1) state is same as finding transitions for LR (0) states the look-ahead symbols just ride along with the LR (1) items.

Find items where the "." is not at the end of the rule.

For every terminal 'x' that appears immediately after the "." in at least one item, add a transition to the following state, labelled with 'x':

(i) Start with a blank state.

(ii) If the rule is of the form $A \rightarrow \alpha.B\beta,a$ (where α and β are strings of terminals and non-terminals, B is a nonterminal, and a is the look ahead symbol) in the original state, add the item $A \rightarrow \alpha B \cdot \gamma a$ to the new state.

Find the closure of the new state from step 8

Consider the grammar of Fig. 5.20.15. The LR (0) states for this grammar are in Fig. 5.20.16.

Terminals	=	{a,b}
Non-Terminals	=	{S', S, A,B}
Rules	=	(0) $S' \rightarrow S$
		(1) $S \rightarrow AaAb$
		(2) $S \rightarrow BbBa$
		(3) $A \rightarrow \varepsilon$
		(4) $B \rightarrow \varepsilon$
Start symbol	=	S

Fig. 5.20.16 : A CFG

The grammar of Fig. 5.20.16 is clearly not LR (0) there is a shift reduce conflict in State 2,3,6 and 7. Is this grammar SLR (1) ? The follow sets for all of the non-terminals are in Fig. 5.20.17 and the

LR (0) parse table is in Fig. 5.20.16(a) there are duplicate entries.

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

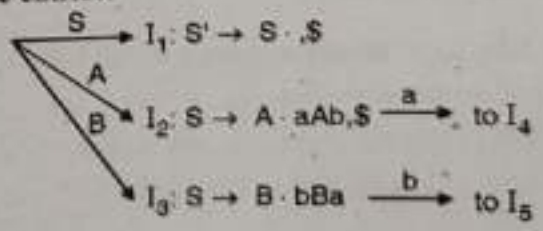
I_0 : $S' \rightarrow S$

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow .$

$B \rightarrow .$



I_4: $S \rightarrow AaAB \xrightarrow{A} I_6 : S \rightarrow AaA.b \xrightarrow{b} I_8 : S \rightarrow AaAb.$

$A \rightarrow .$

I_5: $S \rightarrow BaBa \xrightarrow{B} I_7 : S \rightarrow BbB.a \xrightarrow{a} I_9 : S \rightarrow BbBa.$

$B \rightarrow .$

Fig. 5.20.16(a) : The LR (0) states and transitions for the grammar of Fig. 5.20.16

Scanned by CamScanner

State	Action			Go To		
	a	b	$	S	A	B
0				G1	G2	G3
1			accept			
2	S4/R3					
3		S5/R4				
4					G6	
5						G7
6	R1	S8/R1	R1			
7	S9/R2		R2			
8	R1	R1	R1			
19	R2	R2	R2			

Fig. 5.20.16(b) : LR(0) Parse Table for CFG mentioned in Fig. 5.20.16

Follow sets for CFG mentioned in Fig 5.20.16

Non-Terminal	FOLLOW
S	{$}
A	{a, $}
B	{b, $}

Fig. 5.20.17 : The FOLLOW sets for the non-terminals in the grammar Fig. 5.20.16

Let's take a closer look at the conflict in the SLR (1) parse table of Fig. 5.20.16(b). SLR (1) parser don't keep track of that much information, so we can't tell whether we should shift or reduce. For this LR (1) parser is used.

(1) LR (1) items

The look-ahead symbols in an LR (1) item are the set of all terminals that could follow the rule in the current derivation.

LR (1) items contain a rule with a "·", just like LR (0) items.

We start with $S' \rightarrow \cdot S \$$, and it has non look-ahead information as it has end of file marker.

Rule	Look ahead
$S \rightarrow \cdot S \$$	

Let us expand the rule $S \rightarrow .S \$$, then we will first have to reduce something to an S – and the symbol that will follow that S is a $:

Scanned by CamScanner

Rule	Look ahead
S → · S $	
S → · AaAb	$
S → .BbBa	$

If we want to reduce S → · AaAb. We will look a group of terminals that can be reduced to S.

Rule	Look ahead
S → · S $	
S → · AaAb	$
S → .BbBa	$
A → ·	A
B → ·	B

Creating the LR (1) transitions diagram in Fig. 5.20.18.

S → AaAb
S → BbBa
A → ε
B → ε

I_0: S' → S, $
S → AaAb, $
S → BbBa, $
A → ., a
B → ., b

$\xrightarrow{S}$ I_1: S' → S · $
$\xrightarrow{A}$ I_2: S → A · aAb, $\xrightarrow{a}$ to I_4
$\xrightarrow{B}$ I_3: S → B · bBa, $\xrightarrow{b}$ to I_5

I_4: S → AaAB, $ $\xrightarrow{A}$ I_6: S → AaA.b, $ $\xrightarrow{b}$ I_8: S → AaAb., $
A → ., b

I_5: S → Bb.Ba, $ $\xrightarrow{B}$ I_7: S → BbB.a, $ $\xrightarrow{a}$ I_9: S → BbBa. , $
B → ., a

Fig. 5.20.18 : The LR (1) states and transitions for the grammar of Fig. 5.20.16

State	Action			Go To		
	a	b	$	S	A	B
0	R3	R4		G1	G2	G3
1			accept			
2	S4					
3		S5				
4		R3			G6	
5	R4					G7
6	R1	S8				
7	S9					
8			R1			
9			R2			

Fig. 5.20.19 : LR (1) parser table for the grammar of Fig. 5.20.16

Scanned by CamScanner

(2) Creation of DFA based on states and transitions

DFA for LR (1) context free grammar is as :

(1) Add the rule $S' \rightarrow S\ \$$.

(2) $S' \rightarrow \cdot\ S\ \$$ <empty> is the start state of the DFA.

(3) By adding all the new states create all the transitions.

Consider the grammar of Fig. 5.20.20.

Terminals = {+, −, =}

Non-terminals = {E′, E, D, F}

Rules = (0) $E' \rightarrow E\ \$$

(1) $E \rightarrow DF$

(2) $D \rightarrow -D$

(3) $D \rightarrow +$

(4) $E \rightarrow -$

(5) $F \rightarrow =F$

(6) $F \rightarrow \varepsilon$

Start symbol = S

Fig. 5.20.20 : A simple CFG that is neither LL (0) LL (1)

But is SLR (1) and LR (1)

First sets for all of the non-terminals in this grammar.

Non-terminal	First
E	+, −
D	+, −
F	=, ε

Build the first state we start with :

Rule	Lookahead
$E \rightarrow \cdot\ E\ \$$	

Closure of this state $E' \rightarrow \cdot\ E\ \$$, is all rules that have LHS of E to this state. Since a $ follows the E in this item, the lookahead for all of these rules will be $.

Rule	Lookahead
$E \rightarrow \cdot\ E\$$	
$E \rightarrow \cdot -$	$
$E \rightarrow \cdot\ DF$	$

Scanned by CamScanner

Closure of the state E → · DF, $, is all rules that have LHS of B to this state. What should be look-ahead for these items be? Since F follows the "·", the look-ahead for all these items is FIRST (F). as FIRST (F) contains ε, we need to add the look-ahead of the item E → · DF ,$ as well, giving :

Rule	Look-ahead
E → · E $	
E → · –	$
E → · DF	$
D → · +	=, $
D → · - D	=, $

The symbols that appear immediately before the "·" items in this state are E, -, D and +. We can now construct the transitions out of this state.

E on an E we go to the closure of the state.

Rule	Look-ahead
E → E · $	

- on a -, shift into the closure of the state.

Rule	Look-ahead
E → – ·	$
E → – · D	=, $

Which is

Rule	Look-ahead
E → – ·	$
D → – · D	=, $
D → · – D	=, $
D → · +	=, $

D on a D we go to the closure.

Rule	Look-ahead
E → D · F	$
F→ · =F	$
F→ ·	$

The entire set of states and transitions can be found in Fig. 5.20.21.

Scanned by CamScanner

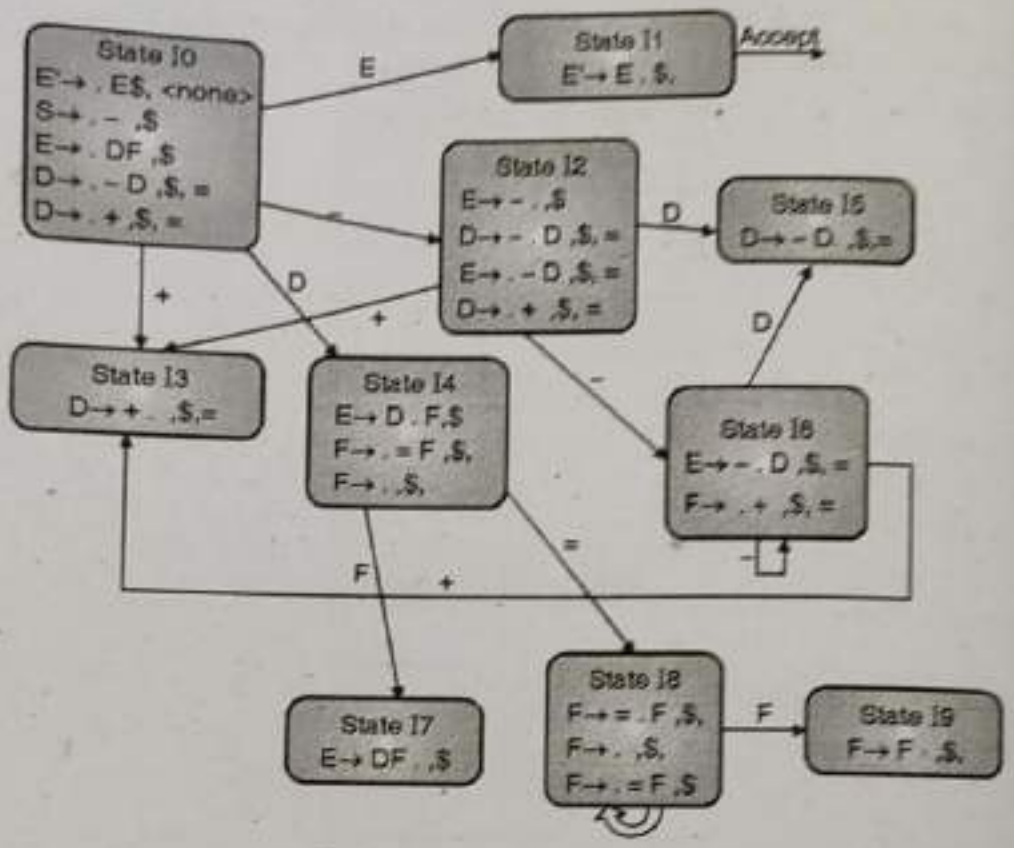


Fig. 5.20.21 : The LR (1) states and transitions for the CFG of Fig. 5.20.20

(3) Creating LR (1) parse tables

1. Construct the canonical collection of sets of LR(1) items for G'. C←{I0,...,In}
2. Create the parsing action table as follows
   - If a is a terminal, A→α.aβ,b in I_i and goto(I_i,a)=I_j then action[i,a] is ***shift j.***
   - If A→α.,a is in I_i, then action[i,a] is ***reduce A→α*** where A≠S'. If state n contains an item of the form A → α ·, a (where α is a (possibly empty) strings of terminals and non terminals and a is the lookahead, add r (<rule number>) to the table in row n, in all columns in a.
   - If S'→S.,$ is in I_i, then action[i,$] is ***accept.***
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains S'→.S,$

Scanned by CamScanner

The LR (1) parse table built from the LR (1) states and transitions of Fig. 5.20.21 is in Fig. 5.20.22.

	Action				Go To		
	+	−	=	$	S	D	F
0	S3	S2			G1	G4	
1				accept			
2	S3	S6		R4		G5	
3			R3	R3			
4			S8	R6			G7
5			R2	R2			
6	S3	S6				G5	
7				R1			
8			S 8	R6			G9
9				R5			

Fig. 5.20.22 : LR (1) parse table for the grammar of Fig. 5.20.21

5.20.6 LR (k) Parsers

As we seen that LL parsers are more powerful due to lookahead symbol attached with it, similarly LR parsers can become more powerful by increasing lookahead. LR (1) items are of the form

$$A \rightarrow \alpha \cdot \gamma\, a$$

α and γ are strings of terminals and non terminals.

a – It represents the set of terminals which is used to represent the characters that could follow the rule $A \rightarrow \alpha\gamma$ in the current derivation

LR (k) items are of the form :

$$A \rightarrow \alpha \cdot \gamma_1\, a$$

Where α and β are strings of terminals and non-terminals, and a is a set of strings of up to K terminals that could follow the rule $A \rightarrow \alpha\gamma$ in the current derivation.

(1) Closure of an LR (k) state

Same steps are involved in finding the closure of an LR (k) which are used in finding the closure of an LR (1) state – the only difference is that we will keep track of lookahead strings of length K, instead of single symbol.

(2) Transitions for an LR (k) state

Same steps are involved in finding the transitions of an LR (k) which are used in finding the transitions of an LR (1) state – the only difference is that we will keep track of lookahead strings of length K, instead of single symbol.

Scanned by CamScanner

5.21 Difference between LL Parser and LR Parser

→ (MU - Dec. 2017)

Q. State difference between LL Parser and LR Parser. Dec. 2017, 5 Marks

Q. Distinguish between : LL Parser and LR Parser. (10 Marks)

Sr. No.	LL Parser	LR Parser
1.	It is also known as top down parsing.	It is also known as bottom-up parsing.
2.	**L**-Left to right i.e. (input is processed in the order it is read) **L** – Leftmost derivation	**L**-Left to right i.e. (input is processed in the order it is read) **R** – Rightmost derivation
3.	LL starts with only the root non-terminal on the stack.	LR ends with only the root non-terminal on the stack.
4.	LL ends when the stack is empty.	LR starts with an empty stack.
5.	LL expands non-terminal.	LR reduces non-terminal.
6.	LL reads terminal when it pops one off the stack	LR reads terminal when it pushes them on the stack
7.	LL uses pre-order traversal of the parse tree.	LR uses post-order traversal of the parse tree.
8.	During an LL parser the parser continuously chooses between two actions **Predict**: Based on the leftmost non-terminal and some number of look-ahead tokens. **Match**: Match the leftmost guessed terminal symbol with the leftmost unconsumed symbol of input	During an LR parser the parser continuously chooses between two actions **Shift**: Add the next token of input to a buffer for consideration. **Reduce**: Reduce a collection of terminals and non-terminals
9.	LL parser are easier to write but less powerful and LL parser also come in many flavours like LL (1) etc.	LR parsers are much powerful and they come in many flavours like (LR(0), SLR(1), LALR(1), LR(1) ...etc)
10.	Example based on LL parser Section 5.19.3(A) Example 5.19.13	Example based on LR parser. Section 5.20.3(A) Example 5.20.17

Scanned by CamScanner

5.22 Difference between Top down and Bottom up Parsing

Q. Differentiate between Top-down and Bottom-up parsing. Explain each of them with suitable example. **(5 Marks)**

Q. Differentiate Top-down and Bottom-up parsing techniques. **(5 Marks)**

Sr. No.	Parameter	Top Down Parsing	Bottom Up Parsing
1.	Definition	Top down parsing performs the parsing from the staring symbol to the input string. It starts from the root level of the parse tree and works down by using the rules of a formal grammar.	A parsing strategy that first looks at the level of the parse tree and works up the parse tree by using the rules of a formal grammar.
2.	Functionality	Occurs from the starting symbol to the input string in top down parsing	Occurs from the input string to the starting symbol in bottom up parsing.
3.	Main Decision	To select what production rule to use in order to construct the string	To select when to use a production rule to reduce the string to get the starting symbol
4.	Method of Construction	Left most derivation	Right most derivation
5.	Examples	Recursive Descent Parser	Shift Reduce parser

5.23 Semantic Analysis - Introduction

➔ **(MU - May 2014, Dec. 2014, Dec. 2016, Dec. 2017)**

Q. Write a short note on Syntax Directed Translation. **May 2014, Dec. 2016, Dec. 2017, 10 Marks**

Q. Explain syntax directed translation. **Dec. 2014, 5 Marks**

Q. Explain syntax directed translation. Give the syntax directed translation equations for infix to prefix conversion. **(10 Marks)**

Syntax-directed translation refers to a method of complier implementation where the source language translation is completely driven by the parser or syntax analysis. In other words, the parsing process and parse trees are used for direct semantic analysis and the translation of the source program. The semantic routines perform interpretation based on the syntax structure. It also attaches the attributes to the grammar symbol and values for these attributes are computed by semantic rules associated with the grammar productions. There are two notations for associating semantic rules with productions, they are as :

(1) Syntax-Directed Definitions.
(2) Translation Schemes.

Scanned by CamScanner

(1) Syntax-Directed Definitions

In compiler construction syntax-directed translation is used for

- Constructing abstract syntax tree
- Type Checking
- Intermediate code generation

It is a method of translating a string into a sequence of actions by attaching one such action to each rule of a grammar. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Thus, parsing a string of the grammar produces a sequence of rule applications. Such grammars are called attribute grammars.

(2) Translation Schemes

Translation scheme refers to the method of compiler implementation where the semantic rules are to be processed based on the production rules.

According to the definitions mentioned above the token is parsed to build the parse tree and then dependency graph is traversed to process the semantic rules at parse tree nodes (see Fig. 5.23.1).

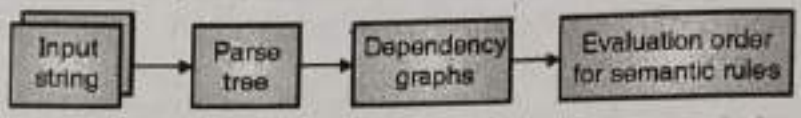


Fig. 5.23.1 : Conceptual view of Syntax-Directed Translation

Each production rule in a grammar is associated with semantic rules which give the information how to calculate the attribute values associated with each grammar symbol in a production. These attributes are evaluated by the semantic rule. The evaluation of these semantic rules gives the following information.

- Intermediate code is generated.
- Symbol table is used to enter the information
- Type checking is performed and error message is issued.

Syntax directed translations can also be implemented through one pass by evaluating semantic rules during parsing without construction of parse tree or a graph showing dependencies between attributes. The translation of the token stream is the result obtained by evaluating the semantic rules.

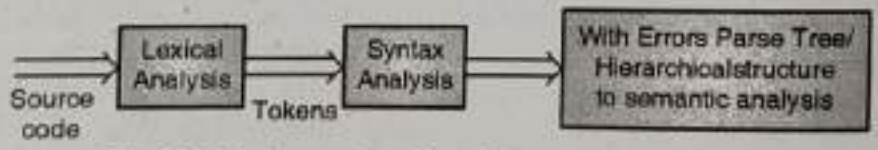


Fig. 5.23.2 : Overview of analysis phase of compiler

In this section we should study how to perform semantic analysis in syntax directed fashion by augmenting the context-free-grammar with semantic rules and the parsing process by evaluation of attributes using semantic rules associated with the grammar productions. We will also study about the different types of attributed schemes.

Scanned by CamScanner

5.24 Syntax Directed Definitions

➔ (MU - May 2014, Dec. 2014, Dec. 2016, Dec. 2017)

Q. Write a short note on Syntax Directed Translation.

May 2014, Dec. 2016, Dec. 2017, 10 Marks

Q. Explain syntax directed translation.

Dec. 2014, 5 Marks

Q. Write note on : Syntax Directed Translation.

(10 Marks)

Q. Explain syntax directed translation. Give the syntax directed translation equations for infix to prefix conversion.

(10 Marks)

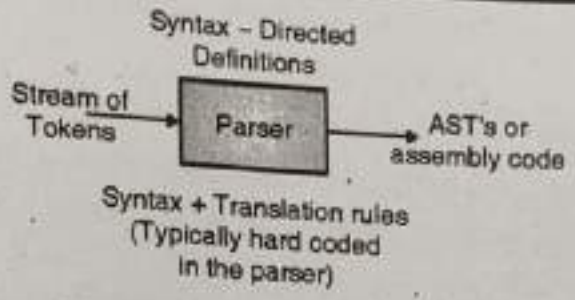


Fig. 5.24.1 : Parser as a Translator

Augmentation of grammar is done by associating attributes with each grammar symbol that describes its properties. An attribute has following information attached with it such as name, value, a string, a number, a type, a memory location, an assigned register or whatever information we need. For example, variables may have an attribute "type" useful for type checking or an integer constant may have an attribute "value" used later to generate code.

- A syntactic structure of input tokens is specified by the context free grammar which uses syntax-directed definition.
- A set of attributes is associated with each grammar symbol.
- A syntax-directed definition associates a set of semantic rules with each production rule.

For example, let the grammar contains the production:

$$X \rightarrow Y\,Z$$

And also let that nodes X, Y and Z have associated attributes X.a, Y.a and Z.a respectively.

The above mentioned production, augmented with a set of actions that use the "a" attribute for node to store the appropriate value. Since in this example, the value of a node X can be determined from 'a' attribute of Y and Z nodes attribute 'a' in a synthesized attribute. Synthesized attributes can be evaluated by a single bottom-up traversal of the parse tree.

term → 0 {term.a = 0}
| 1 {term.a = 1}
...
| 9 {term.a = 9}

Scanned by CamScanner

There are two types of attributes we might come across; synthesized or inherited

(i) An attribute is synthesized if its value at a parent node can be determined from attributes of its children. In this type of grammar the left side attribute is computed from the right-side attributes.

- The attributes for terminals are supplied by the lexical analyzer and the synthesized attributes ones are built up for the non-terminals and passed up the tree.

$$Func \rightarrow A_1 A_2 \ldots A_n$$

$$Func \cdot a = f(A_1 \cdot a, A_2 \cdot a, \ldots A_n \cdot a)$$

(ii) An attribute is inherited if its value is passed down from parent nodes to children in a parse tree i.e., the right-side attributes are derived from the left-side attributes (or other right-side attributes).

$$Func \rightarrow A_1 A_2 \ldots A_n$$

$$A_{i \cdot a} = f(Func \cdot a, A_1 \cdot a, A_2 \cdot a, \ldots, A_{k+1} \cdot a, \ldots, A_n \cdot a)$$

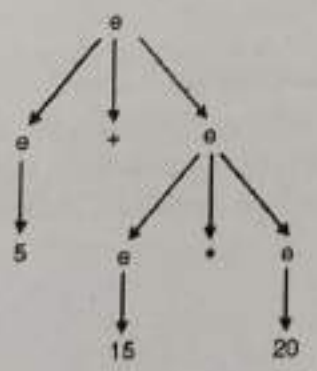
Syntax directed definition is a kind of abstract specification. A semantic rule merely specifies the way in which the attribute of grammar symbols are related to each other. Dependencies between attributes are set up with the help of the semantic rule in form of dependency graph which gives the evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string.

An annotated parse tree shows the value of attributes at each node. The process of computing the attributes values at the nodes is called annotating the parse tree. Values of attributes in nodes of annotated parse tree are either,

- Initialized to constant values given by the lexical analyser.
- Or determined by the semantic rules.
- Following Fig. 5.24.2 shows the example of parse tree and annotated parse tree.

e ::= n | e + e | e − e | e * e | e / e

Parse tree for 5 + 15 * 20 :



Annotated parse tree :

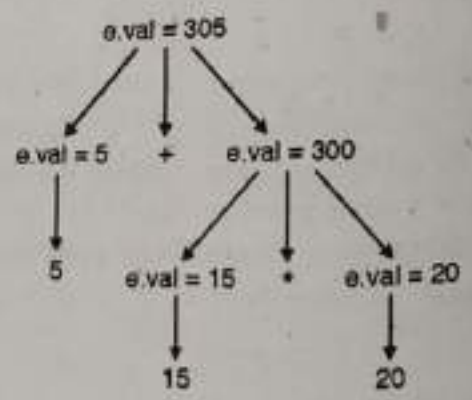


Fig. 5.24.2

Scanned by CamScanner

- Syntax directed definition associates a set of attributes with each grammar symbol and also associates a set of semantic rules with each production. It also specifies how to compute attribute values of the symbols. An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Example 5.24.1

Fig. 5.24.1 shows the example of a synthesized attribute.

```
expr ::= n
       | expr + expr
       | expr * expr
```

- An attribute is synthesized if the attribute value of parent is determined from attribute values of children in the parse tree.

Production	Semantic rules
expr::= n	expr.num = n.num
expr::= expr1 + expr2	expr.num=expr1.num+ expr2.num
expr::= expr1 * expr2	expr.num=expr1.num* expr2.num

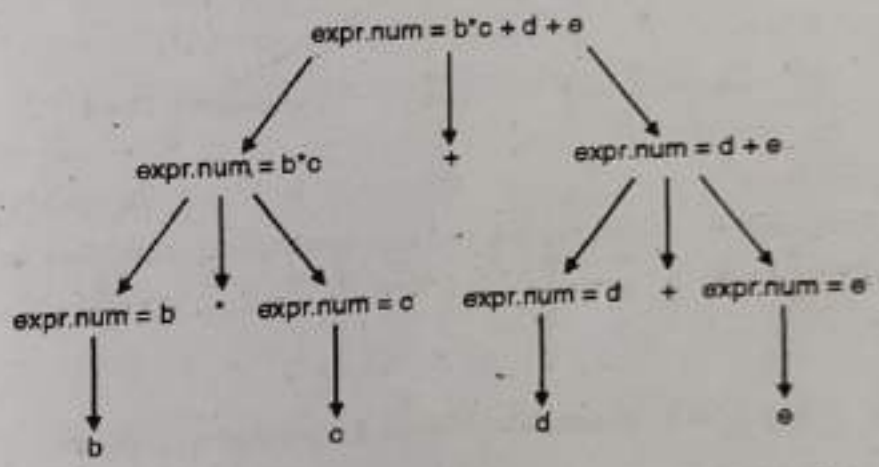


Fig. 5.24.3 : Syntax Directed Definitions for Synthesized Attribute

Symbols expr1 and expr2 are associated with a synthesized attribute *num*. For each expr1 and expr2 production, the semantic rule computes the value of attribute *num* for the non-terminal on the left side from the values of *num* for the non-terminals on the right side.

Example 5.24.2

Based on the syntax-directed definitions mentioned in Example 5.24.1, an annotated parse tree for string 6*2+4+10 is created in Fig. 5.24.4.

Scanned by CamScanner

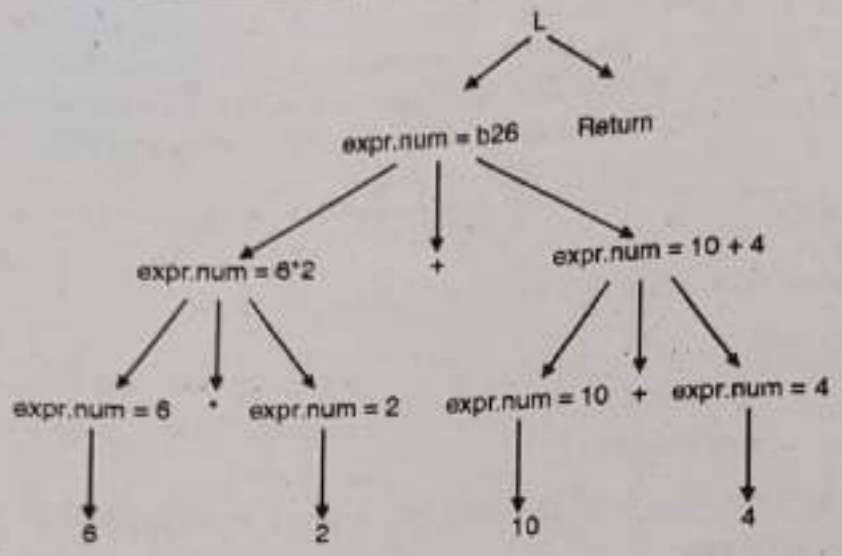


Fig. 5.24.4 : Annotated Parse Tree for String "6*2+10+4"

The dependency graph for the input string "6*2+10+4" is shown in Fig. 5.24.4.

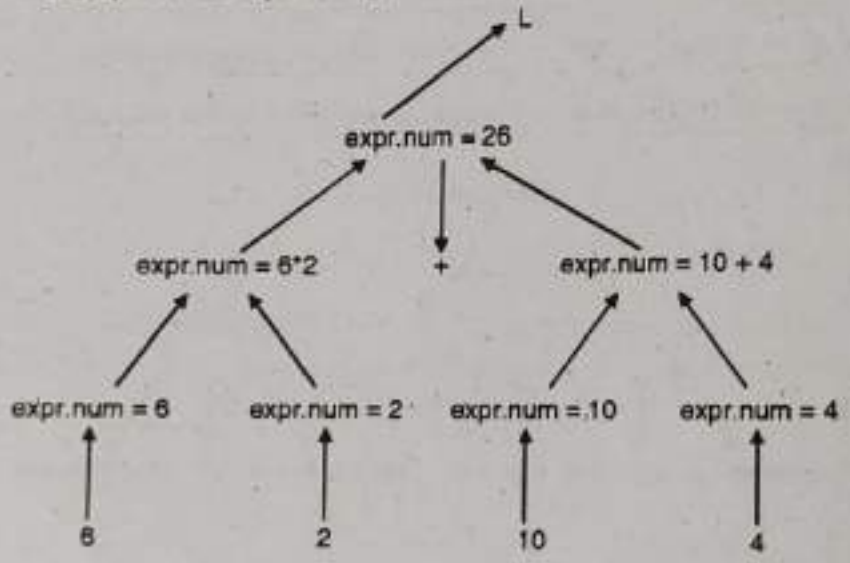


Fig. 5.24.5 : Dependency Graph for String "6*2+10+4"

5.25 Attribute Grammar

Attribute grammars are formalism for expressing syntax directed translation of a context-free grammar. An **attribute grammar** is a formal way to define attributes (*an **attribute** is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance*) for the productions of a formal grammar, associating these attributes to values. The evaluation occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or compiler.

Attributes is a piece of information which determines the properties of a field or tag in a database or a string of characters in a display. Attributes value, time and complexity can be determined during the translation/execution process.

Scanned by CamScanner

Typical examples of attributes are :

1. The data type of a variable.
2. The value of an expression.
3. Variable location in the memory.
4. The object code of a procedure.

The attributes are divided into two groups :

1. Synthesized Attributes
2. Inherited Attributes

Attributed grammars can pass the information from anywhere in the abstract syntax tree to anywhere else, in a controlled way. They can also be used to translate the syntax tree directly into the machine code for some specific machine.

A parse tree showing the values of its attributes is called **annotated parse tree**.

5.25.1 Synthesized Attributes

→ **(MU - Dec. 2015, May 2016, Dec. 2016)**

Q. Explain synthesized attributes used in syntax directed definitions. **Dec. 2015, May 2016, 2.5 Marks**

Q. What are different type of attributes in SDD? Explain with examples. **Dec. 2016, 5 Marks**

An attribute is synthesized if its value at a parent node can be determined from attributes of its children. In general, given a CFG production rule of the form 'A=a B g' then an associated semantic rule of the form "*A.attribute=f(a.attribute, B.attribute, g.attribute)*" is said to specify a synthesized attribute of A.

Synthesized attribute of the node is defined in terms of

- **Attribute values at children of the node.**
- **Attribute value at node itself.**

We have productions of the form : $A \rightarrow X_1 X_2 X_3 \ldots X_n$

- with semantic rules of the form : $b := f(c_1, c_2, c_3, \ldots, c_n)$
- where b and the c's are attributes of the grammar symbols
- b is called a synthesized attribute if :
  - b is an attribute of A (i.e. the LHS), and
  - The c's are all attributes of the X's (symbols on the RHS).

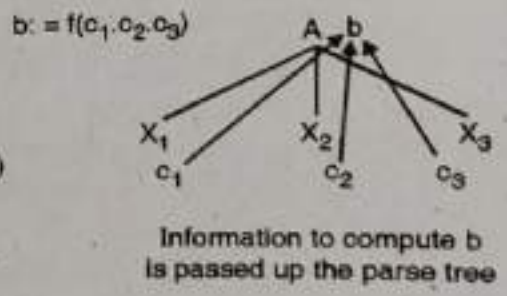


Fig. 5.25.1 : Synthesized Attributes

Synthesized attributes can be evaluated by a single bottom-up traversal of the parse tree.

Scanned by CamScanner

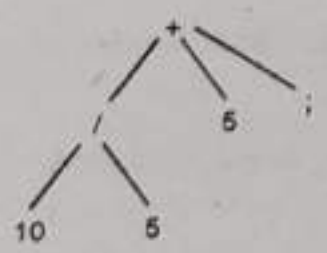
➢ **Example 5.25.1**

Consider the following context tree grammar for computing the synthesized attributes

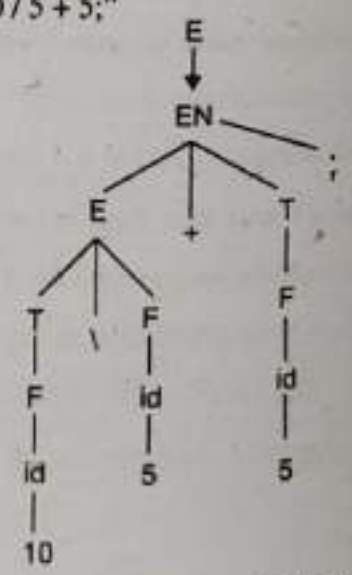
		Production Rule	Semantic Actions
$E \rightarrow EN$		$E \rightarrow EN$	type (E · val)
$E \rightarrow E + T$		$E \rightarrow E + T$	E · val = E · val + T · val
$E \rightarrow T$		$E \rightarrow T$	E · val = T · val
$T \rightarrow T / F$	$\Rightarrow$	$T \rightarrow T * F$	T · val = T · val / F · val
$T \rightarrow F$		$T \rightarrow F$	T · val = F · val
$F \rightarrow (E)$		$F \rightarrow (E)$	F · val = E · val
$F \rightarrow id$		$F \rightarrow id$	F · val = id · lexval
$N \rightarrow ;$		$N \rightarrow ;$	; is terminating symbol so it can be ignored by lexical analyser

Syntax directed definitions for the above grammar.

- E, T and F are non-terminals and their values can be found using the attribute "val".
- "lexval" is a synthesized attribute whose value can be obtained from the lexical analyser. This attribute is associated with token digit.
- Following steps are followed to compute S-attributed definition.
  1. Proper semantic rules are required to write the syntax directed definition for corresponding production rule of the given grammar.
  2. The annotated parse tree is generated in bottom up fashion and attribute values are computed.
  3. The value generated at the root node is supposed to be the final output. Construct parse tree, syntax tree and annotated parse tree for the input string "10 / 5 + 5;"

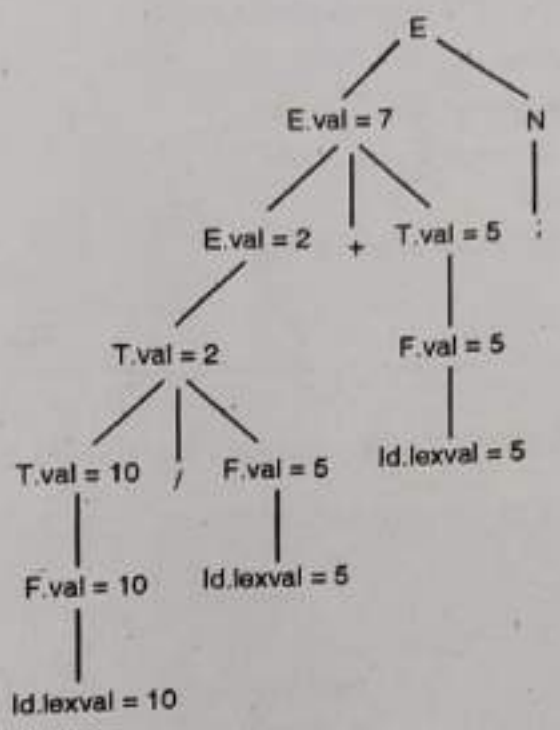


Syntax Tree for String "10 /5+5"



Parse Tree for String "10 / 5 +5"

Scanned by CamScanner



Annonated Parse Tree for String "10 / 5 +5;"

S-attribute definition are the syntax directed definitions that uses synthesized attributes. A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

5.25.2 Inherited Attribute

(MU - Dec. 2015, May 2016, Dec. 2016)

Q. Explain inherited attributes used in syntax directed definitions. Dec. 2015, May 2016, 2.5 Marks

Q. What are different type of attributes in SDD? Explain with examples. Dec. 2016, 5 Marks

An attribute is inherited if the attribute value of a parse-tree node is determined from attribute values of its parent and siblings. In general, given a CFG production rule of the form 'A=a B g' then an associated semantic rule of the form *"B.attribute=f(a.attribute, A.attribute, g.attribute)"* is said to specify an inherited attribute of B. The inherited attributes are passed down from parent nodes to the children nodes of the abstract syntax tree during the semantic analysis of the parsing process.

Inherited attribute of the node is defined in term of

- Attributes values at parent node.
- Attribute values at sibling.
- Attributes value of node itself.

We have the production of the form : $A \rightarrow X_1 X_2 X_3 \ldots X_n$

Scanned by CamScanner

Semantic rules of the form : $b := f(c_1, c_2, c_3, \ldots, c_n)$

- b is an inherited attribute if
  - b is an attribute of one of the X's (i.e. RHS).
  - The c's are the attributes of A and/or one or more of the other X's.
  - Which means they are beside or above where b is needed.

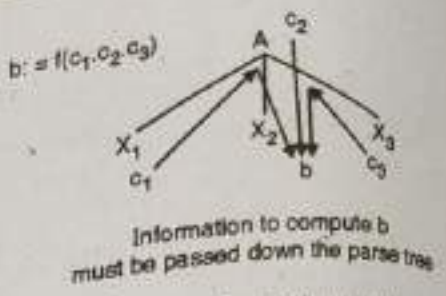


Fig. 5.25.2 : Inherited Attributes

➢ **Example 5.25.2**

Below mentioned context free grammar consists of the keyword int and real, followed by a list of identifiers.

D:: = TL
T::= int | real
L::= L, id | id

Production	Semantic Rules
D → T L	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L1, id	L1.in := L.in symtab(id.entry, L.in)
L → id	symtab(id.entry, L.in)

Fig. 5.25.3 : Inherited Attribute L.in.

- Declaration consists of type T followed by a list of variables.
- Type of T is synthesized and inherited to L :
  - "type" synthesized attribute.
  - "in" inherited attribute.
- Procedure "symtab" adds type of each identifier to its entry in the symbol table pointed to by attribute entry.
- The Fig. 5.25.4 shows an annotated parse tree for the sentence real id_1, id_2, id_3.

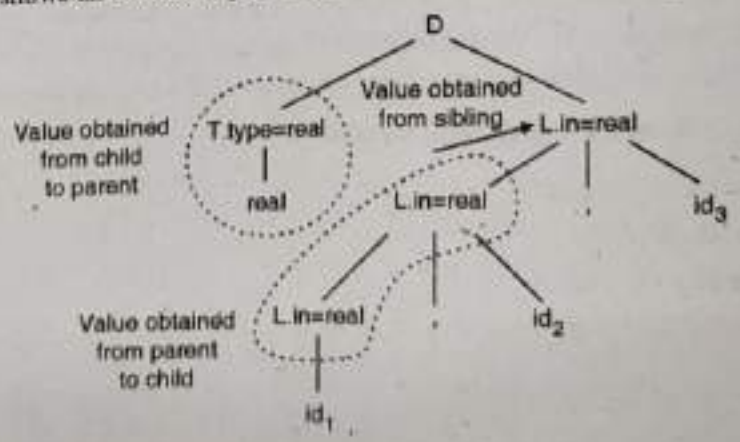


Fig. 5.25.4 : Annotated Parse tree with inherited attributed

Scanned by CamScanner

Example 5.25.3

Consider the following example and build the synthesized and inherited attribute for it.

e ::= n e'

e' ::= +ee' | *ee' | ε

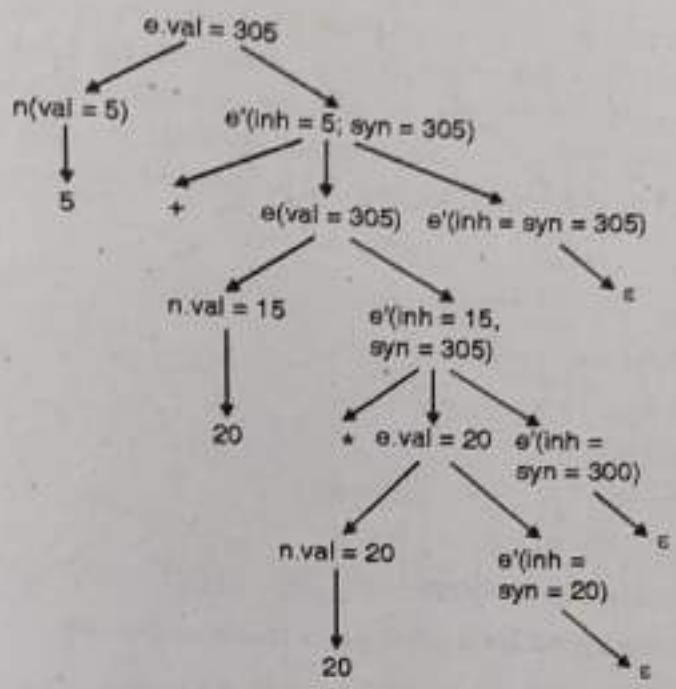


Fig. 5.25.5 : Synthesized Attribute

Production	Semantic rules
e::= ne'	e'.inh = n.val; e'.val = e'.syn
e'::= + ee'1	e'1.inh = e'.inh [+] e.val e'.syn = e'1.syn
e'::= * ee'1	e'1.inh = e'.inh [*] e.val e'.syn = e'1.syn
e'::= ε	e'.syn = e'.inh

Fig. 5.25.6 : Inherited Attribute

5.26 Evaluation Order of Syntax Directed Definitions

Annotated parse tree shows attribute values. Dependency graph help us to determine how those values are computed.

Steps used for evaluating the order of the syntax directed definitions:

1. Build the parse tree and parse the input.
2. Construct the dependency graph for the parse tree.

Scanned by CamScanner

3. Topological sorting is done for the dependency graph.
4. Traverse all nodes in topologically order, and evaluate the value of attributes at each node.

5.26.1 Dependency Graphs

If an attribute 'y' is evaluated after the evolution of 'z' then we say that y is dependent on z this dependency can be shown by parse tree with some other relation known as dependency graph. Dependency graph exists in both synthesized and inherited attributes. The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependency. Below mentioned is the algorithm required to create the dependency graph:

Algorithm

```
for each node n in the parse tree
        for each attribute a of the grammar symbol at n
        construct a node in the dependency graph for a
for each node n in the parse tree
        for each semantic rule b=f(c1,...,ck) associated with
            the production used at n
                construct an edge from each ci to b
```

Properties of the dependency graph for a given parse tree is as follows.

- Evaluate attribute b first if its value is required by attributes a and c.
- Build directed graph without cycles.
- Nodes in the dependency graph should be order in topologically form as n1, n2, : : : , nk such that there is no path from ni to nj with i > j.
- Each arc represents the dependence relationship (flow of values).
- Dependency graph should be acyclic.

➢ **Example 5.26.1**

Fig. 5.26.1 shows the dependency graph for Synthesized attribute.

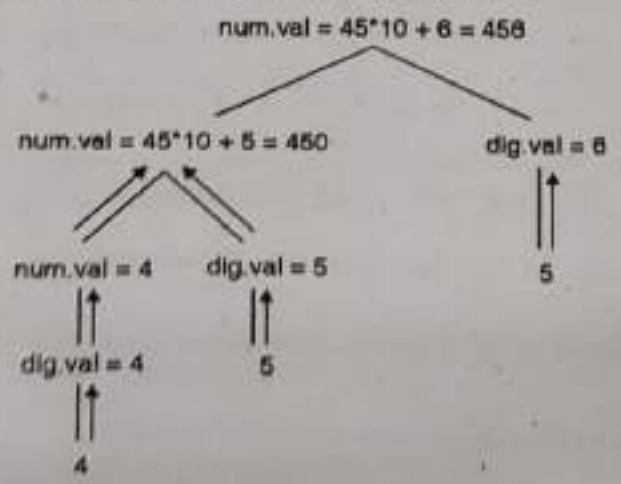


Fig. 5.26.1 : Dependency Graph for Synthesized Attribute

Scanned by CamScanner

Example 5.26.2

Consider the following expression y:=3*x+z and draw the parse tree and dependency graph for this expression.

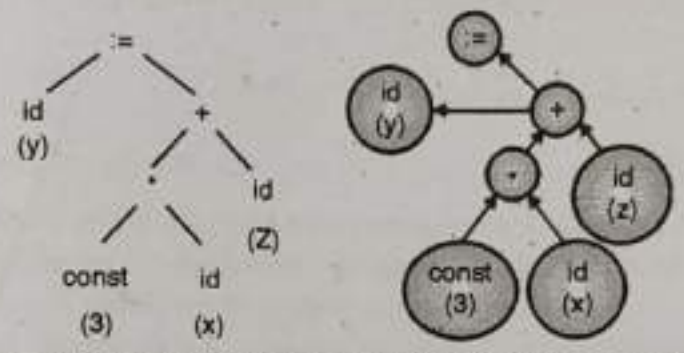


Fig. 5.26.2 : Parse Tree and Dependency Graph for expression y:=3*x+z

The arrows from *const* and *id* to * shows that the value of * depends on *const* and *id*.

Example 5.26.3

Dependency graph for the Fig. 5.25.4 is shown in Fig. 5.26.3.

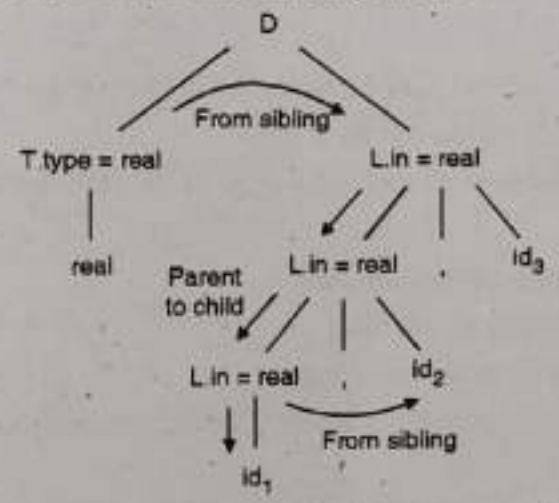


Fig. 5.26.3 : Parse tree with inherited attribute in at each node labeled L for the string "real id_1, id_2, id_3".

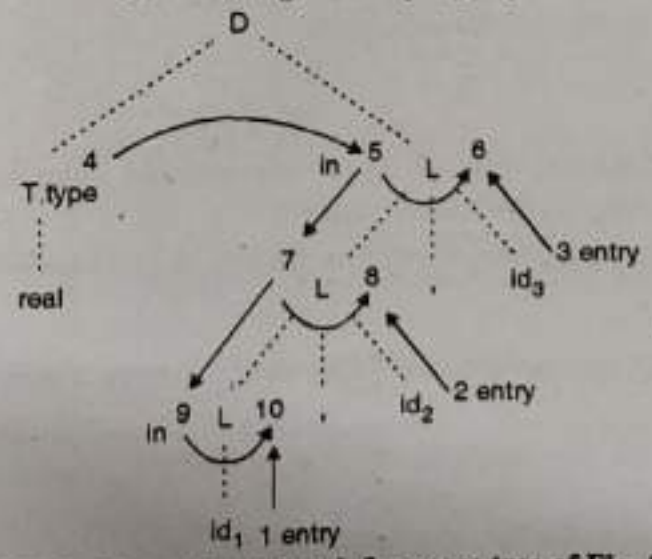


Fig. 5.26.4 : Dependency graph for parse tree of Fig. 5.26.3

Scanned by CamScanner

Fig. 5.24.4 gives the following information about the dependency graph.

Edge from 4 to 5 according to the semantic rule L.in = T.type for the production $D \rightarrow TL$

The two downward edges from 5 to 7 and from 7 to 9 is according to the semantic rule $L_1.in = L.in$ for the production $L \rightarrow L_1, id$.

Nodes 6, 8, and 10 are made to represent dummy attributes.

5.26.2 Evaluation Order

Topological sort of a directed acyclic dependency graph to order attributes. Using these order semantic rules associated with the nodes in parse tree is evaluated. This order depends upon both the grammar and input string. A topological sort of the graph is any ordering $n_1,...,n_k$ of the nodes such that edges of the graph are all from left to right in the ordering; i.e., a node appears in the ordering after all of the nodes it depends on. A topological sort is a legal evaluation order of the attributes.

Note : For the topological sort to make sense there can be no cycles in the graph. Cyclically defined attributes are not legal.

Several methods have been proposed for evaluating semantic rules :

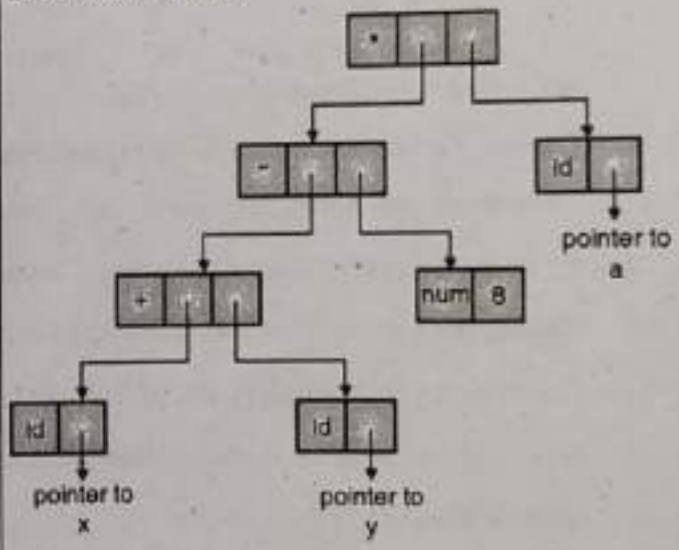
1. **Parse Tree Methods**
   - It is done at compile time and these methods obtain to build a parse tree for each input.
   - Build a dependency graph from the parse tree.
   - Obtain evaluation order from a topological order of the dependency graph.
2. **Rule Based Methods**
   - Analyze rules at compiler-generation time.
   - Determine a static ordering at that time.
   - Evaluate nodes in that order at compile time.
   - Predetermine the order of attribute evaluation for each production.
3. **Oblivious Methods**
   - Evaluation order is independent of semantic rules.
   - Evaluation order forced by parsing methods.
   - Restrictive in acceptable attribute definitions.
4. **Depth-first Traversal**
   - Evaluate attributes.
   - Syntax directed definition does not impose any particular order as long as order computes attribute of parent after all its children's attribute.
   - Rule-based and oblivious methods are more efficient as they do not build dependency graph at compile time.

5.27 Differentiate between Parse Tree and Syntax Tree

Q. Define and explain the following terms with suitable example. **(10 Marks)**
(i) Parse tree (ii) Syntax tree

Q. Differentiate between Parse Tree and Syntax Tree. **(5 Marks)**

Scanned by CamScanner

Sr. No.	Parse Tree	Syntax Tree
1	It is also known as Concrete Syntax Tree (CST)	It is also known as Abstract Syntax Tree (AST)
2	It contains every token of the input as the leaf and the interior nodes are non-terminals in the grammar used to parse.	It omits much of the detail that would be present in a CST. The interior nodes represent language constructs as defined by the grammar.
3	Parse tree may be generated for sentences in natural languages.	Syntax tree is a tree representation of the abstract syntactic structure of source code written in programming language.
4	Parse trees are often built by the power part of the source code translation and compiling process.	In AST subsequent information is added during semantic analysis process.
5	A parse tree is a record of the rules (and token) used to match some input test	Syntax tree records the structure of the input and is insensitive to the grammar that produced it.
6	E.g. x + y – 8 * a **PARSE TREE** 	E.g. x + y – 8 * a **SYNTAX TREE** 

5.27.1 Uses of Syntax tree

Q. Syntax trees and their usage. (5 Marks)

1. It uses for the semantic analysis of the program.
2. A complete traversal of the tree allows verification of the correctness of the program.
3. After verifying correctness, the AST serves as the base for code generation.
4. The AST is often used to generate the 'intermediate representation' '(IR)', sometimes called an intermediate language, for the code generation.

Scanned by CamScanner

Review Questions

Q. 1 Write a brief note on the compiler. Also mention the advantages and disadvantages of a compiler.

Q. 2 Explain various types of compilers.

Q. 3 Explain various phases of compilers.

Q. 4 How source program is interpreted ? What are the similarities and differences between both interpreters and compilers ?

Q. 5 Describe the analysis-synthesis model of a compiler.

Q. 6 Describe various compiler construction tools.

Q. 7 Describe the siblings/relatives of a compiler.

Q. 8 Discuss the actions taken by a complier for the following statements

(a) pas = int + rate * 60 (b) $D = b^2 - 4ac$

(c) Volume $= \frac{4}{3} * 3.14 * r * r * r$

Q. 9 Write a brief note on

(a) Lexical analysis (b) Semantic analysis

(c) Features of compiler (d) Interpreter Vs compiler

(e) Code optimization in compiler

Q. 10 Discuss various errors handle by the error handler in compiler in each phase.

Q. 11 Discuss various errors detected in each phase of compiler.

Q. 12 Describe the role of lexical analyzer in the design of a compiler.

Q. 13 Discus Lexical Analysis with the help of two examples.

Q. 14 Describe the features and various issues in Lexical Analyzer.

Q. 15 What do you mean by a context-free-grammar ? Give an example of CFG.

Q. 16 With reference to CFG explain

(a) Left most derivation (b) Right most derivation

(c) Left recursion (d) Left factoring

Give one example of each.

Q. 17 What is parsing ? Explain various type of parsers.

Q. 18 What is top down parsing ? Explain various types of top-down parsing with the help of an example.

Q. 19 Explain LL (1) grammar in detail.

Scanned by CamScanner

Q.20 Write a short note on

(a) LR parsers

(b) Operator precedence parser

Q.21 Show that no left recursive grammar can be LL (1).

Q.22 Show that no LL (1) grammar can be ambiguous.

Q.23 Show that no LR (1) grammar can be ambiguous.

Q.24 Write an algorithm to create a LR parsing table.

Q.25 Discuss the role of parser in compiler design.

Q.26 Define the following terms

(a)	Terminal	(f)	Canonical LR parser
(b)	Non terminal	(g)	Parsing table
(c)	Ambiguous	(h)	Context free language
(d)	FIRST	(i)	Panic mode recovery
(e)	FOLLOW	(j)	Error production

Q.27 Consider the following grammar.

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

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

$B \rightarrow 0A \mid 1S$

(a) Construct a leftmost derivation for the following sentences.

(i) 0101 (ii) 1100101

(b) Draw its parse tree.

Q.28 Consider the grammar G = { {S, L, L′}, {a, (,), ,}, S, P} where P contains

$S \rightarrow (L)$

$S \rightarrow a$

$L \rightarrow SL'$

$L' \rightarrow , SL' \mid \epsilon$

(a) Construct a leftmost and a rightmost derivation for each of the following sentences.

(i) (a, (a, a))

(ii)(a, ((a, a), (a,a))).

(b) Draw the parse tree for these sentences.

Q.29 (a) What do you mean by an ambiguous grammar ? Give an example. How you can disambiguate a grammar ?

(b) Show whether the following grammar is ambiguous or not.

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

Scanned by CamScanner

Q. 30 Consider the following grammar.

$E \rightarrow T + E \mid T$

$T \rightarrow V * T \mid V$

$V \rightarrow id$

Write down the procedures for the nonterminals of the grammar to make a recursive descent parser.

Q. 31 Write and explain rules for finding the FIRST and FOLLOW sets of any given grammar.

Q. 32 What modifications must be done on a grammar to make it suitable for predictive parsing ? Explain with a suitable example.

Q. 33 Why FOLLOW set is required for the construction of LL (1) parsing table ?

Q. 34 Consider the following grammar.

$S' = S\#$

$S \rightarrow ABC$

$A \rightarrow a \mid bbD$

$B \rightarrow a \mid \in$

$C \rightarrow b \mid \in$

$D \rightarrow c \mid \in$

(a) Construct the first and follow sets for the grammar.

(b) Design a LL (1) parsing table for this grammar.

(c) Is the grammar LL (1) ?

Q. 35 Construct LL (1) parsing table for the following grammar.

$S \rightarrow A$

$A \rightarrow aB \mid aC \mid Ad \mid Ae$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

Q. 36 Consider the following grammar

$S \rightarrow : PS$

$P \rightarrow C \mid B$

$B \rightarrow H{:}T$

$H \rightarrow bd \mid H{:}d$

$T \rightarrow Se \mid S{:}T$

$C \rightarrow bT$

Design a SLR (1) parser for this grammar.

Scanned by CamScanner

Q.37 Consider the following grammar.

$S \rightarrow E_1$

$E_1 \rightarrow T_3 E_1 \mid T_2$

$E_2 \rightarrow T_3 E_2 \mid T_2$

$T_1 \rightarrow a\$ \mid (E_2\$$

$T_2 \rightarrow a) \mid (E_2)$

$T_3 \rightarrow a+ \mid (E_2+$

(a) Construct the sets of LR (0) items for this grammar.

(b) Construct the SLR parsing table for the same.

Q.38 Construct a canonical LR (1) parser for the following grammar

$E \rightarrow T \mid E - T$

$T \rightarrow F \mid * F$

$F \rightarrow id \mid (E)$

Q.39 What are the faced in problems

(a) Top down parsing (b) LR (0) parsing method.

Q.40 Consider the following grammar

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a$

(a) Construct the SLR parsing table for this grammar and show all moves for the parsing of input string "abab" using this table.

(b) Construct the canonical parsing table for this grammar.

Q.41 Explain various conflicts that occur during shift reduce parsing.

Q.42 Define handle of a sentential form and explain its role in bottom up parsing.

Q.43 Find FIRST and FOLLOW of following :

(i) $S \rightarrow aAB \mid bA \mid \varepsilon$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bB \mid C$

(ii) $S \rightarrow v \mid XUW \mid UV \mid uWV$

$U \rightarrow w \mid WvX \mid SUVW \mid SWX \mid \varepsilon$

$V \rightarrow x \mid wX \mid \varepsilon$

$W \rightarrow w \mid UXW$

$X \rightarrow yz \mid SuW \mid SW$

Scanned by CamScanner

Q. 44 Parse the following grammar using LL (1) parser and check whether they satisfy the LL(1) conditions.

(a) $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

(b) $E \rightarrow E + T \mid T$

$T \rightarrow T^* F \mid F$

$F \rightarrow F^* \mid a \mid b$

(c) $S \rightarrow AaAb \mid BbbA$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

(d) $S \rightarrow iEtSS1$

$S1 \rightarrow eS \mid E$

$E \rightarrow b$

(e) exp → atom | list

atom → **num** | **id**

list → (exp′)

exp′ → exp′ exp | exp

(f) dect → type list

type → **int** | float

list → **id**, list | **id**

Q. 45 Show grammar $B \rightarrow Bb \mid a$ is SLR (1) but $B \rightarrow ABb \mid a$, $A \rightarrow \varepsilon$ is not LR (k) for any k.

Q. 46 Find following grammar is operator precedence or not ?

$E \rightarrow ET + IT$

$T \rightarrow TF * IF$

$F \rightarrow FP \mid P \uparrow$

$P \rightarrow E \mid id$

Q. 47 Can you parse following grammar using LR (1) parser.

(i) $E \rightarrow E + T/T$

$T \rightarrow T^*F/F$

$F \rightarrow id$

(ii) $S \rightarrow AA$

$A \rightarrow aA/b$

Scanned by CamScanner

Q. 48 Construct LR (1) parse table for the following grammar :

$S \rightarrow aBD / AB / DAC / b$

$A \rightarrow SCB / SABC / CbD / C / \varepsilon$

$B \rightarrow d / \varepsilon$

$C \rightarrow ADC / c$

$D \rightarrow SaC / SC / fg$

Q. 49 Construct LR (0) parsers for each of the following grammar :

(a) $S \rightarrow cA / ccB$ (b) $S \rightarrow aSSb / aSSS / c$

(c) $A \rightarrow cA / a$ (d) $B \rightarrow ccB / b$

Q. 50 Find the canonical collection of LR (0) items for the following grammar

$E \rightarrow E + T / T$

$T \rightarrow T^* F / F$

$F \rightarrow (E) / id$

Q. 51 Create a set of LR (0) items and transitions and a parse table for the following grammars :

a. Terminals = {a, (,), $}

Non-Terminals = {S′, S, A}

Rules = (0) $S' \rightarrow S\$$

(1) $S \rightarrow SA$

(2) $S \rightarrow A$

(3) $A \rightarrow (S)$

(4) $A \rightarrow a$

Start Symbol = S′

b. Terminals = {a, b, $}

Non-Terminals = {S′, S, A, B}

Rules = (0) $S' \rightarrow S\$$

= (1) $S \rightarrow AB$

= (2) $A \rightarrow Aa$

= (3) $A \rightarrow a$

= (4) $B \rightarrow Bb$

= (5) $B \rightarrow b$

Start Symbol = S′

Q. 52 Create a set of LR (0) items and transitions for the following grammar. Show that the grammar is not LR (0). Give the SLR (1) parse table for the grammar.

Terminals = {num, id, [,], $}

Non-Terminals = {E′, E, V}

Scanned by CamScanner

Rules = (0) $E' \rightarrow E\$$
(1) $E \rightarrow V$
(2) $E \rightarrow num$
(3) $V \rightarrow id$
(4) $V \rightarrow V\ [E]$

Start Symbol = E'

Q. 53 For each of the following grammars, create the set of LR (1) items and transitions and LR (1) parse table, and state if the grammar is SLR (1).

(a)

Terminals = {a, b, \$}
Non-Terminals = {S′, S, A, B}
Rules = (0) $S' \rightarrow S\$$
(1) $S \rightarrow AB$
(2) $S \rightarrow BA$
(3) $A \rightarrow aaB$
(4) $A \rightarrow a$
(5) $B \rightarrow bbA$
(6) $B \rightarrow a$

Start Symbol = S'

(b)

Terminals = {a, b, \$}
Non-Terminals = {S′, S, A}
Rules = (0) $S' \rightarrow S\$$
(1) $S \rightarrow bAb$
(2) $S \rightarrow A$
(3) $A \rightarrow ab$
(4) $A \rightarrow a$

Start Symbol = S'

Q. 54 Consider the following CFG

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid I$
$I \rightarrow a \mid b \mid c$

Remove the left recursion from above grammar.

Q. 55 Explain Recursive-descent parser as push down automata and Error recovery in a top down parser.

Scanned by CamScanner

2.55 Construct predictive parsing table for following grammar.

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

2.57 Explain Recursive descent parser with suitable example.

2.58 Construct LL (1) parsing table for following grammar.

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \in$

$D \rightarrow EF$

$F \rightarrow g \mid \in$

$F \rightarrow f \mid \in$

2.59 (a) For the following grammar construct the predictive table and explain it step by step :

Grammar G :

$E \rightarrow IE'$

$E' \rightarrow TE' / E$

$I \rightarrow FT'$

$T' \rightarrow * FT' / E$

$E \rightarrow (E) / id$

(b) Explain operator precedence parser with the help of suitable example.

2.60 Explain LL (1) parser with the help of example.

2.61 Define the following terms with the help of suitable example :

(i) Parse tree. (ii) Syntax tree.

(iii) Sentential form. (v) Context free grammar.

2.62 (a) Consider the following grammar :

$S \rightarrow A$

$A \rightarrow Ad \mid Ae \mid aB \mid aC$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

Eliminate the left recursion from the above grammar

(b) Construct predictive parsing table for the following grammar :

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \in$

$B \rightarrow \in$

Scanned by CamScanner

Q. 63 (a) Explain meaning of ambiguous Grammar, Left recursive Grammar and left factoring of a Grammar with example.

(b) Give Non-ambiguous Grammar for following pattern of a's, b's and c.

(c) WC w^{-1} where 'w' is any string of a's and b's and w^{-1} represent recurse of string w. e. aabab Cbabaa.

Q. 64 Consider the following grammar

$E \rightarrow E + T$

$E \rightarrow T$

$E \rightarrow T * F$

$T \rightarrow F$

$T \rightarrow (E)$

$T \rightarrow id$

Show the shift reduce parser action for the string id + id + id * id.

Q. 65 What is the role of semantic analyzer in the process of compilation ?

Q. 66 Define a syntax directed definition. Give a suitable example.

Q. 67 Consider the following grammar

$E \rightarrow E + T$

$E \rightarrow T$

$E \rightarrow num.num$

$E \rightarrow num$

(a) Write a syntax directed definition to determine the type of each sub-expression. The result type is integer, if two integers are added otherwise it is real.

(b) Construct an annotated parse tree for input expression "10 + 20 + 15.2".

Q. 68 What do you mean by S-attributed and L-attributed definition?

Q. 69 Explain synthesized and Inherited attributes used in Syntax Directed Definition.

Q. 70 What are different type of attributes in SDD? Explain with examples.

Q. 71 Write short note on Syntax Directed Translation.

Chapter Ends

□□□

Scanned by CamScanner

CHAPTER 6

Module 6

Compilers : Synthesis Phase

Syllabus Topics

Intermediate Code Generation : Types of Intermediate codes : Syntax tree, Postfix notation, Three address codes : Triples and Quadruples.

Code Optimization : Need and sources of optimization, Code optimization techniques : Machine Dependent and Machine Independent.

Code Generation : Issues in the design of code generator, code generation algorithm. Basic block and flow graph.

6.1 Intermediate Code Generation-Introduction

Some compilers generate an explicit intermediate representation of the source program after first three phases of the compiler i.e. lexical, syntax and semantic analysis. The intermediate representation of the source program should acquire two main characteristics

1. Easy to produce and handle
2. Translation into the target program should be easy to understand

If we recall, we have studied that compilation is done in two phase's i.e.

PHASE 1 : Analysis of source program and remember relevant information and,

PHASE 2 : Perform synthesis of target program (Code Generation)

The Language Processor processes the source code in first pass and remembers the information for the second pass. But in second pass it re-reads and processes the complete source code once again which causes :

1. Unnecessary repeated efforts
2. Wastage of memory,
3. Wastage of Processing Time,

Use of Intermediate Representation (IR) phase between first pass and second pass helps to reduce these drawbacks.

Scanned by CamScanner

Language Processor

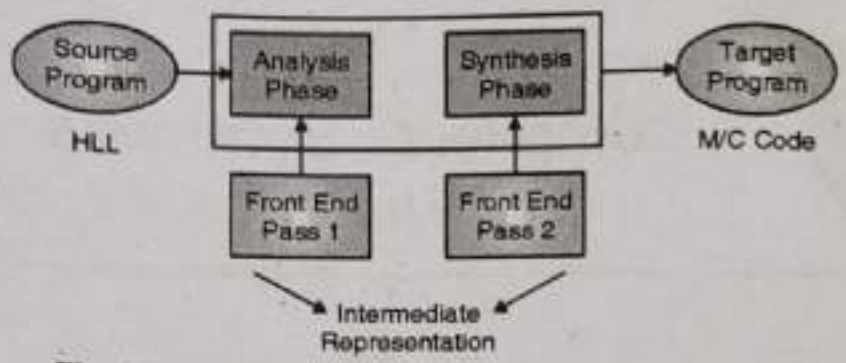


Fig. 6.1.1 : Basic block Diagram of Compilation Process

IR is a representation of output of language processor when source program is partially processed (after First Pass).

Pass 1 : Performs analysis of source program and reflects its results in the intermediate representation (IR) form.

Pass 2 : Reads and analyses the (IR), instead of source program, and performs synthesis of the target program.

From Fig. 6.1.1 we can see that

Front End

- IR produced by front end has two components :
  1. Database (Symbol table, Literal table)
  2. An intermediate code (IC)

  This IC is actually a description of the source program.

➢ **Example 1**

Consider the following example.

float sum; int a=2; int b=3;
sum=sum+a+b;
the symbol table created would be..

these are id nos, position of symbols in symbol table

Sr. No.	Symbol	Type	Length	Address	Other info
1	sum	float	4 bytes	4000H	value =0101
2	a	int	2 bytes	4004H	value =0010
3	b	int	2 bytes	4006H	value =0011
4	a*	These are temp names added internally during semantic analysis of '=' statement. Here a and b are to be treated as float since they would be stored in float			
5	b*				
6	temp				

Scanned by CamScanner

The intermediate code (IC) generated after first pass would be of the form.

1. Convert (ID,# 2) to float giving (ID, #4)
2. Convert (ID,# 3) to float giving (ID, #5)
3. Convert (ID,# 1) to (ID, #4) giving (ID, #6)
4. Add (ID, # 6) to (ID, # 5) giving (ID # 7) (not shown in figure)
5. Store (ID # 7) in (ID # 1)

Sr. No.	Symbol	Type	Length	Address	Other info
1	sum	float	4 bytes	4000H	value =0101
2	a	int	2 bytes	4004H	value =0010
3	b	int	2 bytes	4006H	value =0011
4	a*	These are temp names added internally during semantic analysis of '=' statement. Here a and b are to be treated as float since they would be stored in float			
5	b*				
6	temp				

Back End : Back End takes care of

- **Memory allocation**
- **Code generation**

Here a*b* and temp are not shown since these are LP's internal decision;

whether to allocate separate memory or not

Sr. No.	Symbol	Type	Length	Address	Other info
1	sum	float	4 bytes	4000H	value =0101
2	a	int	2 bytes	4004H	value =0010
3	b	int	2 bytes	4006H	value =0011

In code generation the IC is converted into assembly language depending upon type of processor to be used for further execution

Most of the compilers convert the source program first into some form of *intermediate representation like three address code* and after that into machine code. The intermediate representation is a machine and language-independent version of the original source code. Intermediate representation provides with a clean and separate front and back ends, and adds possibilities for retargeting / cross-compilation.

Scanned by CamScanner

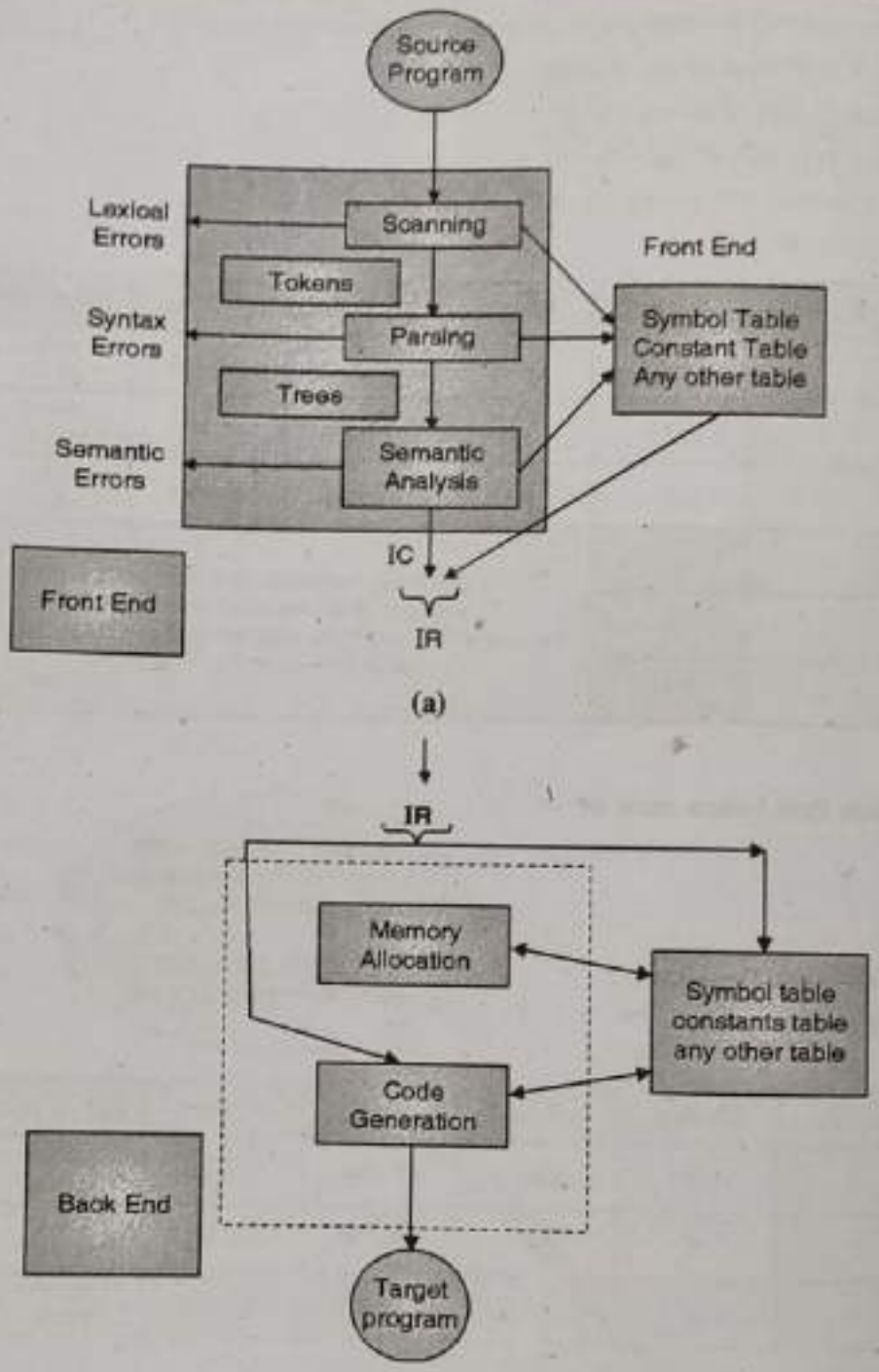


Fig. 6.1.2 : Front-End and Back-End for Intermediate Code generation

There are many intermediate representations in use (some suggests it may be as many as a unique one for each existing compiler) but the various representations are actually more alike than they are different.

Scanned by CamScanner

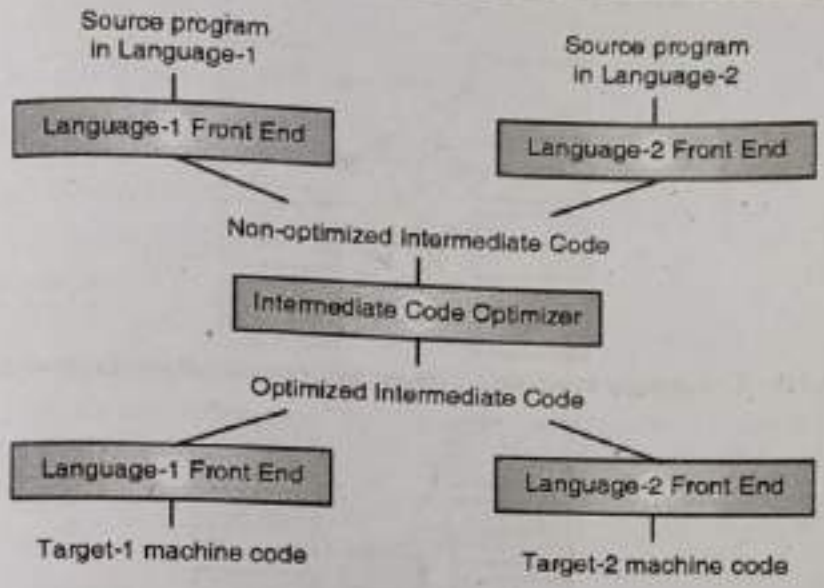


Fig. 6.1.3 : Component Based Approach to Build Compilers

In computing, an **intermediate representation** (IR) is a data structure that is constructed from input data to a program, and from which part or all of the output data of the program is constructed in turn. Intermediate representation is a kind of abstract machine language that can express the target machine operations without describing too much of machine details.

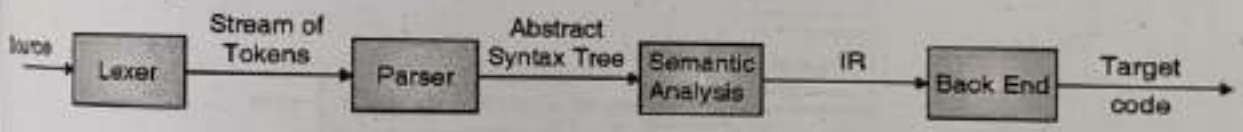


Fig. 6.1.4 : Position of Intermediate Code Generator

6.1.1 Intermediate Representation (IR) Gives

- Clean and abstract machine language
- Target machine operations are expressed by it.
- Independent of machine
- Not dependent on any source language

6.1.2 Importance of Intermediate Representation

- Fig. 6.1.5 shows the languages which are executed directly without any intermediate representation. But we can also convert all the languages to an intermediate representation and could use a common backend to generate a machine independent code.

Scanned by CamScanner

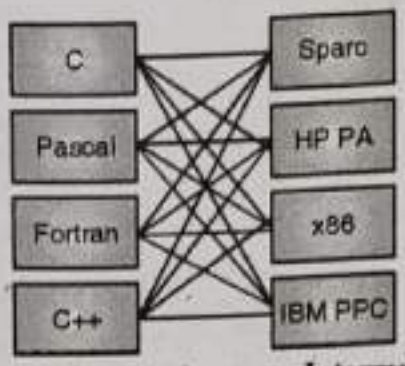


Fig. 6.1.5 : Languages executing without any Intermediate Representation

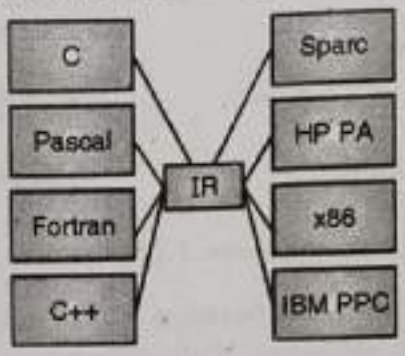


Fig. 6.1.6 : Languages executing with an Intermediate Representation

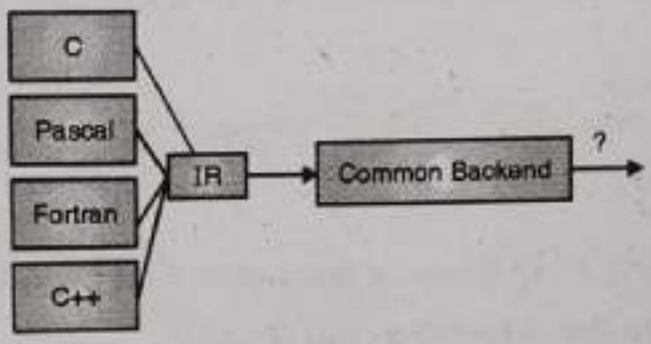


Fig. 6.1.7 : Languages executing with an IR and common backend

- Conversion of source program into an "intermediate language " is
  - Easy
  - Simple to understand
  - Not dependent on CPU,

6.1.3 Advantages of using An Intermediate Language

In the analysis synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back-end generates target code. Some important advantages of machine-independent intermediate form are :

1. **Retargeting :** In this we build a new compiler by attaching a new code to an existing front-end.
2. **Optimization :** Reuse intermediate code optimizers in compilers for different languages and different machines.

Scanned by CamScanner

Note : The terms "intermediate code", "intermediate language", and "intermediate representation" are all used interchangeably.

6.1.4 Issues in Designing an Intermediate Representation

- Whether to use an existing IR
  - if target machine architecture is similar
  - if the new language is similar
- Whether the IR is appropriate for the kind of optimizations to be performed
  - E.g. speculation and predication
  - some transformations may take much longer than they would be a different IR
- Designing a new IR needs to consider
  - Level (how machine dependent it is)
  - Structure
  - Expressiveness
  - Appropriateness for general and special optimizations
  - Appropriateness for code generation
  - Whether multiple IRs should be used

6.1.5 Multiple Level Intermediate Representations

Multiple Level IR helps in various ways that are mentioned as follows :

- Translating from one level to another in the compilation process
- Preserving an existing technology investment
- Some representations may be more appropriate for a particular task.

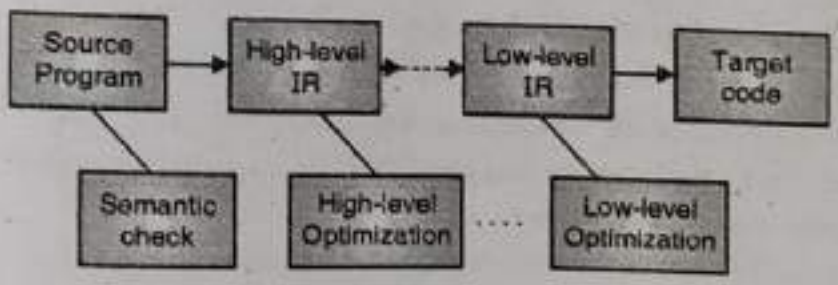


Fig. 6.1.8 : Multiple Level Intermediate Representations

6.1.6 Generally used different forms of Intermediate Representation

→ **(MU - May 2014, Dec. 2017)**

Q. What are the different forms of intermediate code generated? **May 2014, 10 Marks**

Q. Explain the different types of intermediate code representations with examples. **Dec. 2017, 10 Marks**

Q. Explain various form of the intermediate code used by compiler. **(10 Marks)**

Q. Explain the different types of intermediate code representations. **(10 Marks)**

Q. With example explain the different forms of intermediate code generated. **(10 Marks)**

Scanned by CamScanner

Q. What are the different types of intermediate codes? Explain their implementation techniques. (10 Marks)

Intermediate representations of source program are machine independent, but they are close to machine instructions. The intermediate code of any source language is given by the intermediate code generator. Various types of intermediate forms used by the designer are as :

- Graphical representations forms are as:
  - Syntax trees, AST (Abstract Syntax Trees), and DAG (Direct Acyclic Graph)
  - Postfix notation
  - Three-address code (Triples, Indirect triples and Quadruples) can be used as an intermediate language.
- SSA (Static Single Assignment) form
- IR should have individual components that describe simple things
- Programming languages have well defined intermediate languages like JAVA, PROLOG etc.

6.2 Intermediate Languages

→ (MU - May 2014, Dec. 2016)

Q. What are different forms of intermediate code generated ? May 2014, 10 Marks

Q. Discuss various intermediate code forms in detail. Dec. 2016, 10 Marks

Q. Explain the term : Intermediate language. (10 Marks)

Intermediate languages are designed to use by the compilers. They are the language of an abstract machine designed to aid in the analysis of computer programs. In this compiler first translates the source code into an intermediate code as an intermediate step before generating machine code for a target machine. The design of an intermediate language typically differs from that of a practical machine language in three ways :

- Each instruction represents exactly one operation.
- Instruction set does not include control flow information.
- The number of registers available may be large, even limitless.
- Syntax trees and postfix notations are two kinds of intermediate representation the third kind of IR which is commonly used is three address codes.

Postfix notation is a notation in which a mathematical notation wherein every operator follows all of its operands.

The PN of expression :

1. 9* (5+2) is 952+*
2. (a+b)/(c–d) is ab+cd–/

The Rules for Postfix Notation are as :

1. Operands appear before their operators. If E is an expression of the form E1 op E2, the PN of E is E1'E2 'op.
2. If E represents variable/constant, the PN of E is E itself.
3. If E is a parenthesized expression of form (E1), the PN of E is the same as the PN of E1.
4. The expression is evaluated by scanning from left to right.

Scanned by CamScanner

6.2.1 Graphical Representation

Syntax tree is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. A DAG (Directed Acyclic Graph) gives the same information but in more compact way because common sub expressions are identified.

6.2.1(A) Syntax Trees

A parse tree is an example of a very high-level intermediate representation. As parse tree contains all the information about the program that can be used to build the source code. A syntax tree is a graph structure. Nodes in a syntax tree correspond to statements. Edges in a syntax tree represent relations between statements.

Abstract syntax tree is used to drive the semantic processing and code generation. An abstract syntax tree (AST), or just syntax tree, is a tree representation of the syntactic structure of the source code written in any programming language. Each node of the tree denotes a construct occurring in the source code. Each node has reference to its parent and child node.

Consider the following example of a programming language grammar :

```
program -> function_list
function_list -> function_list function | function
function -> PROCEDURE ident ( params ) body
params -> ...
```

A sample program for this language

```
PROCEDURE main()
BEGIN
statement...
END
PROCEDURE factorial(n:INTEGER)
BEGIN
statement...
END
```

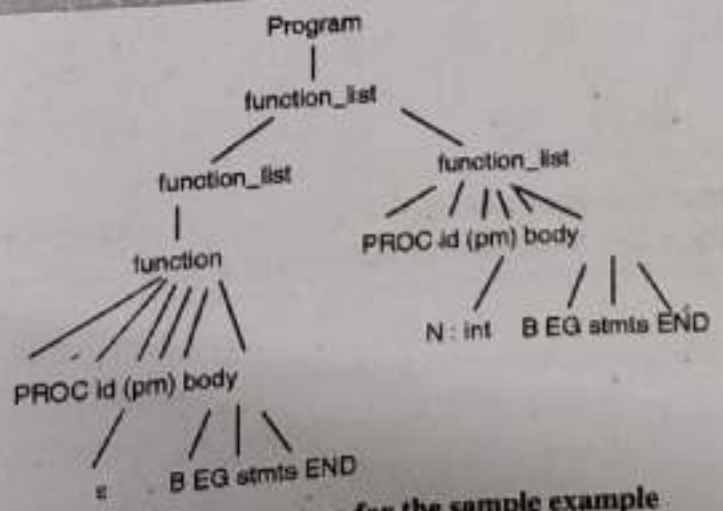


Fig. 6.2.1 : Parse tree for the sample example

Scanned by CamScanner

Consider again the syntax tree for the arithmetic expressions (a + b * c)

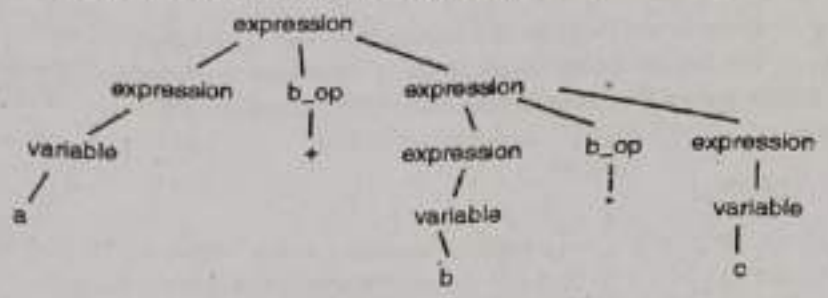


Fig. 6.2.2 : Syntax tree for Aritnmetic Expression

6.2.1(B) Directed Acyclic Graph (DAG)

➔ **(MU - Dec. 2014)**

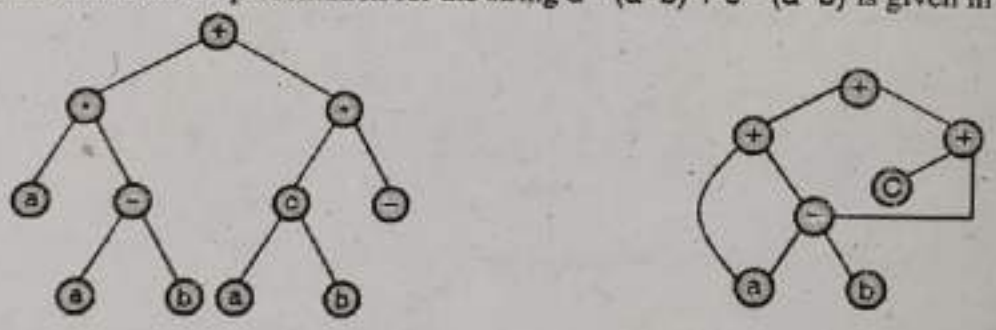
Q. Explain DAG. **Dec. 2014, 5 Marks**

In compilers a tree has a one route from the parent (root) to child (leaf) node. *Directed acyclic graph* (DAG) is used to share the common subtree of a given expression. DAG is used represent more than one path from start symbol to terminals. DAG's are difficult to construct, but provide an obvious savings in space. They also highlight equivalent sections of code and that will be useful later when we use different optimization techniques.

A **directed acyclic graph** is a collection of vertices and directed edges. Each edge connecting one vertex to another, such that there is no way to start at some vertex *v* and follow a sequence of edges that eventually loops back to *v* again. **DAG has** no directed cycles.

➢ **Example 1**

A syntax tree and DAG representation for the string a * (a–b) + c * (a–b) is given in Fig. 6.2.3.



(a) Syntax Tree Representation **(b) DAG Representation**

Fig. 6.2.3 : Graphical Representation of a:=b* –c+b* –c

➢ **Example 2**

D = ((A+B*C) + (A*B*C))/ –C construct a DAG for it.

Scanned by CamScanner

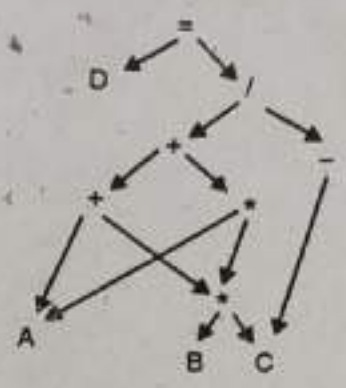


Fig. 6.2.4 : DAG for string "D = ((A+B*C) + (A*B*C))/ – C"

6.3 Three-Address Code

➔ **(MU - Dec. 2014)**

Q. Explain three address code. **Dec. 2014, 5 Marks**

Q. Explain three-address code, its statements, types and also implementation of three address statements. **(10 Marks)**

In three address code, each statement generally contains three addresses, two for the operands and one for the result. This is why it is termed as **three address code**. It is a popular form of intermediate code used in optimizing compilers. It is a sequence of statements having a general form.

$$a = b \text{ op } c$$

Where op is any operator like +, –, * etc. and a, b and c can be variables, constants or the temporary variables generated by the compiler. We can say that three address codes is linearized representation of a syntax tree.

In three-address code "x * 5 – y" might be translated as

$$t1 = x * 5$$

$$t2 = t1 - y$$

In general, when generating three-address statements, the compiler has to create new temporary variables (temporaries) as needed. The advantage of using temporary names for intermediate results is that the three-address code can be easily rearranged. We use a function **newtemp ( )** that returns a new temporary each time it is called. Temporary names must be generated to compute intermediate operations. Addresses are implemented as pointers to their symbol-table entries.

The syntax tree and DAG in Fig. 6.2.4 are represented by the three-address code sequences in Fig. 6.3.1.

Scanned by CamScanner

a * (a-b) + c * (a-b)

$t_1 := a$
$t_2 = b$
$t_3 := t_1 - t_2$
$t_4 := a$
$t_5 := t_4 * t_3$
$t_6 = c$
$t_7 := t_6 * t_3$

(a) Linearized representation of a Syntax Tree

$t_1 := a$
$t_2 = b$
$t_3 := t_1 - t_2$
$t_4 := a$
$t_5 := t_4 * t_3$
$t_6 := t_1 * t_3$
$t_8 := t_5 + t_6$
$t_9 := t_5 + t_7$

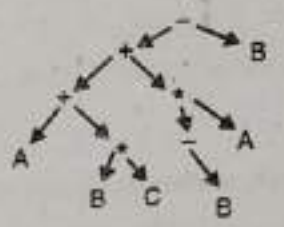
(b) Linearized Representation of a syntax DAG

Fig. 6.3.1 : Three-address codes corresponding to the syntax tree and DAG of Fig. 6.2.4

➢ **Example 1**

Construct the syntax tree and three-address code for the following string

"(A+B*C) + (–B*A) – B"



(a) Syntax Tree

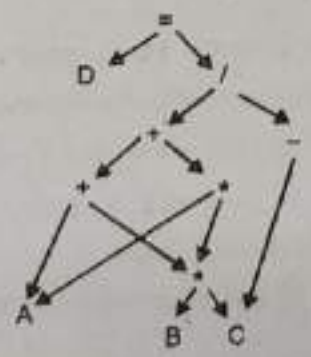
$t_1 = B * C$
$t_2 = A + t_1$
$t_3 = -B$
$t_4 = t_3 * A$
$t_5 = t_2 + t_4$
$t_6 = t_5 - B$

(b) Three-Address Code

Fig. 6.3.2 : Syntax tree and Three-Address code for example 9

➢ **Example 2**

Consider the expression: D = ((A+B*C) + (A*B*C))/ –C. Draw the DAG and three-address code representation for the above expression.



(a) DAG Representation

$T_1 := A$
$T_2 := C$
$T_3 := B * T_2$
$T_4 := T_1 + T_3$
$T_5 := T_1 * T_3$
$T_6 := T_4 + T_5$
$T_7 := -T_2$
$T_8 := T_6/T_7$
$D := T_8$

OR

$T_1 := -B * C$
$T_2 := A + T_1$
$T_3 := A*T_1$
$T_4 := T_2 + T_3$
$T_5 := -C$
$T_6 := T_4 / T_5$
$D := T_6$

(b) Three-Address Code Representation

Fig. 6.3.3 : DAG and Three-Address Code for example 10

Scanned by CamScanner

Example 6.3.1 : Generate three address code for a given expressions.
While (A < B) do
If (C < D) then X = Y + Z **(10 Marks)**

Solution :

Three address code for a given expressions

While (A < B) do

if (C < D) then

X = Y + Z

L1 : while (A < B) goto L2

goto last

L2 : if (C < D) goto L3

Goto last

L3 : $t_1 = Y$

$t_2 = Z$

$t_3 = t_1 + t_2$

$X = t_3$

last :

Example 6.3.2 : Draw Syntax tree and Directed Acylic Graph (DAG) for expression
(a * b) + (c – d) * (a * b) + b **(10 Marks)**

Solution :

Steps for converting syntax tree.

(I) Convert the given expression from infix to postfix expression.

The postfix expression is as;

ab*cd – +ab * *b +

Syntax tree for the given expression ;

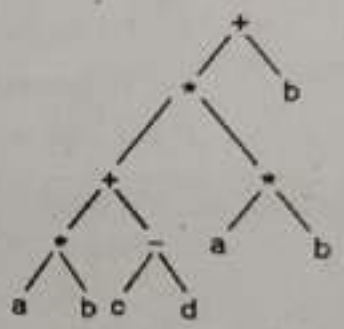


Fig. P. 6.3.2

The sequence of operation for syntax tree is as;

P_1 = mkleaf (id, a)

P_2 = mkleaf (id, b)

P_3 = mknode (*p_1, p_2) → (a*b)

P_4 = mkleaf (id, c)

P_5 = mkleaf (id, d)

P_6 = mknode (–, P_4, P_5) ⇒ (c – d)

P_7 = mknode (+, P_3, P_6) ⇒ (a * b) + (c – d)

P_8 = mkleaf (id, a)

P_9 = mkleaf (id, b)

P_{10} = mknode (*, P_8, P_9) ⇒ (a * b)

P_{11} = mknode (*, P_7, P_{10}) ⇒ (a * b) + (c – d) * (a * b)

Scanned by CamScanner

P_{12} = mkleaf (id, b)

P_{13} = mknode (+, P_{11}, P_{12}) ⇒ (a * b) + (c – d) * (a * b) * b

Directed Acyclic graph of the given expression (a * b) + (c – d) * (a * b) + b

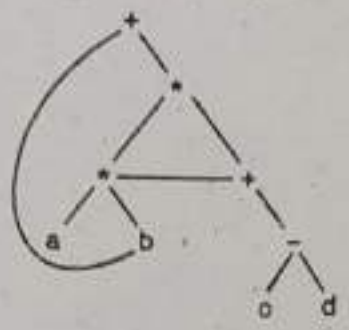


Fig. P. 6.3.2(a)

The sequence of operation for DAG as follows.

P_1 = mkleaf (id, a)

P_2 = mklead (id, b)

P_3 = mknode (*, P_1, P_2) → (a * b)

P_4 = mkleaf (id, c)

P_5 = mkleaf (id, d)

P_6 = mknode (–, P_4, P_5) ⇒ (c – d)

P_7 = mknode (+, P_3, P_6) → (a * b) + (c – d)

P_8 = mknode (* P_7, P_3) → (a * b) + (c – d) * (a * b)

P_9 = mknode (+ P_8, P_2) → (a * b) + (c – d) * (a * b) + b

Example 6.3.3 : Generate three address code for following code.

```
While (a < b) do
  if (c < d)   then
    x = y + 2
  else
    x = y – 2
```

MU - May 2015, 10 Marks

Solution :

Three address code for a given expressions

```
    while (a < b) do
    if (c < d) then
    x = y + 2
    else
    x = y - 2
L1 : while (a < b) goto L2
    goto last
```

Scanned by CamScanner

L2 : if (c < d) goto L3
Goto L4:
L3 : $t_1 = y$
$t_2 = 2$
$t_3 = t_1 + t_2$
$x = t_3$
L4 : $t_1 = y$
$t_2 = 2$
$t_3 = t_1 - t_2$
$x = t_3$
last

6.4 Implementation of Three-Address Statements

Q. Explain three-address code, its statements, types and also implementation of three address statements. (10 Marks)

A three address code can be implemented as records having fields for the operators and operands. **Three-address code** (often abbreviated to TAC or 3AC) is a form of representing intermediate code used by compilers to aid in the implementation of code-improving transformations. In a compiler it can be realized with the fields for the operators and the operands. The three-address code can be implemented in several ways they are as :

1. Quadruples
2. Triples
3. Indirect Triples

6.4.1 Quadruples

→ (MU - May 2015, May 2016)

Q. Discuss with example quadruple. May 2015, 4 Marks
Q. Explain with an example Quadruples. May 2016, 4 Marks

A quadruples is a record structure having four fields as follows :

Operator	Operand 1	Operand 2	Result

Fig. 6.4.1 : Structure of Quadruple

Where operand1 and operand2 are two operands for the operator and result field contains the result of the operation on operand1 and operand2. For example **a:=b+c** will be represented as

+ b c a

The fields operand1, operand2 and result are pointers to the symbol table entries, therefore temporary names must be entered into the symbol table. The three-address statement

x: = a op b

is represented by placing a in operand1, b in operand 2, and x in result.

Scanned by CamScanner

- Unary operators like **arg1:= -arg2 or arg1:=arg2** does not use operand2.
- Procedure call like ***param*** use neither operand2 nor result.
- A conditional and unconditional jump puts the target label in result.
- Operand1, Operand2 and result are usually pointers to symbol entries.
- Quadraples may need to use many temporary names.

➢ **Example 1**

Find out the three-address code and quadruples for the assignment statements :

a: = b*–c + b*–c.

Three-Address code :

t_1: = – c
t_2: = b * t_1
t_3: = – c
t_4: = b * t_3
t_5: = t_2 + t_4
a: = t_5

	Operator	Operand1	Operand2	Result
(0)	Uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	Uminus	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Fig. 6.4.2 : Quadruples representation for the three-address statement

Note : Temporary names must be entered into the symbol table as they are created.

➢ **Example 2**

Fig. 6.4.3 shows the quadruples for an assignment statement

x: = a + b * c + c * d *e

Three address code :

t_1 = b * c
t_2 = c * d
t_3 = e * t_2
t_4 = a + t_1
t_5 = t_3 + t_4
x = t_5

	Operator	Operand1	Operand2	Result
(0)	*	b	c	t_1
(1)	*	c	d	t_2
(2)	*	t_2	e	t_3

Scanned by CamScanner

	Operator	Operand1	Operand2	Result
(3)	+	t_1	a	t_4
(4)	+	t_3	t_4	t_5
(5)	assign	t_5		X

Fig. 6.4.3 : Quadruples for the string "x:= a+b * c+c * d*e"

Advantage

In quadruples it is easy to rearrange code for global optimization.

Disadvantage

In this lots of temporaries are created, so lot of memory is wasted.

6.4.2 Triples

➔ **(MU - May 2015, May 2016)**

Q. Discuss with example triple. **May 2015, 4 Marks**

Q. Explain with an example Triples. **May 2016, 4 Marks**

A triple is a record structure having three fields as follows

Operator	Operand 1	Operand2

Fig. 6.4.4 : Structure of Triples

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. If we do so the three-address code can be represented with only three fields operator, operand1 and operand2 are either pointers to the symbol table or pointers into the triple structure. It refers to the symbol table for user defined names or constants and to the triple structure for temporary values.

The triple **A * B** represents the infix notation ***A B**.

Example 3

Find out the three-address code and triples for the assignment statements.

a: = b*–c + b*–c.

Three-Address Code

t_1: = c

t_2: = b * t_1

t_3: = c

t_4: = b * t_3

t_5: = t_2 + t_4

a: = t_5

Scanned by CamScanner

	Op	arg1	arg2
(0)	Uminus	c	
(1)	*	b	(0)
(2)	Uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	Assign	a	(4)

Fig. 6.4.5 : Triples representation for Three-Address statements

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the name themselves. In practice the information needed to interpret the different kinds of entries in the operand1 and operand2 fields can be encoded into the operator field or some additional fields. The triples in Fig. 6.4.5 correspond to the quadruples in Fig. 6.4.2. Note that the copy statement $a := t_5$ is encoded in triple representation by placing a in the operand field by using the operator assign.

Note : Temporary names are not entered into the symbol table is Triples.

➢ **Example 4**

Ternary operations like

i) func[n]:= z

ii) func:= z[n]

	Operator	argument1	argument2
(0)	[ ] =	func	n
(1)	Assignment	(0)	z

(a) func[n]:=z

	Operator	argument1	argument2
(0)	=[ ] ,	func	n
(1)	Assignment	z	(0)

(b) func:= z[n]

Fig. 6.4.6 : Triples Representation for Ternary Operators

➢ **Example 5**

The sequence of triples produced for the string z:=p+q–r*s*t is as shown in Fig. 6.4.7.

Triple Number	Operator	Operand1	Operand2
(0)	+	P	q
(1)	*	R	s
(2)	*	(1)	t
(3)	-	(0)	(2)
(4)	=	Z	(3)

Fig. 6.4.7 : Triples for the string z:=p+q–r*s*t

Scanned by CamScanner

Advantage : In triples temporaries are implicit.

Disadvantage : In triples it is difficult to rearrange code.

6.4.3 Indirect Triples

→ (MU - May 2015, May 2016)

Q. Discuss with example indirect triple. May 2015, 4 Marks

Q. Explain with an example Indirect triples. May 2016, 4 Marks

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples. In indirect triples we use an array statement to list pointers to triples in the desired order. Then the triples in Fig. 6.4.5 might be represented as in Fig. 6.4.8.

	Statements		#	Operator	Argument 1	
0	100	→	100	uminus	C	
1	101	→	101	*	B	100
2	102	→	102	uminus	C	
3	103	→	103	*	B	102
4	104	→	104	+	101	103
5	105	→	105	=	A	104

Fig. 6.4.8 : Indirect Triples Representation of Three-Address Statements

Advantages

- In indirect triples the temporaries are implicit and easier to rearrange code.
- A statement can be moved by reordering the statement list.

6.4.4 Comparison of Representation

- The difference between triples and quadruples may be regarded as a matter of how much indirection is present in the representation. When we ultimately produce target code, each name, temporary or programmer defined, will be assigned some run-time memory locations.
- This location will be placed in the symbol table entry for the datum.
- Using a quadruple notation, a three-address statement defining or using a temporary variable can immediately access the location for the temporary via the symbol table.
- Using a quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use.
- In the triple notation, moving a statement that defines a temporary value requires us to change all references to that statement in the operand1 and operand2 arrays. This problem makes triples difficult to use in an optimizing compiler.
- In indirect triples a statement can be moved by reordering the statement list. Since pointers to temporary values refer to the operator-operand1-operand2 arrays which are not changed, none of those pointers need to be changed.

Scanned by CamScanner

- Indirect triples can save some space compared with quadruples if the same temporary value is used more than once.

6.5 Types of Three-Address Statements

Q. Explain types and implementation of three address statements. (10 Marks)

Three-Address statements are similar to assembly code. Three address codes is a form of representing intermediate code used by compilers to aid in the implementation of code improving transformations. Statements can have symbolic labels and flow-of-control.

Here are the some common three-address statements used in the remainder of this book :

1.	Assignment Statement	res := arg1 op arg2 res := op arg1	Where res, arg1, arg2 are names, constants, or compiler generated temporaries; op is any binary, arithmetic, logical or unary operation.
2.	Indexed Assignments	arg1 := arg2[j], arg1[j] := arg2	arg1, arg2 and j refer to data objects
3.	Copy Statements	arg1 : = arg2,	value of arg2 is given to arg1.
4.	Unconditional Jump	goto L,	L is a label of the next instruction to be executed.
5.	Conditional Jump	if arg1 relop arg2 goto L.	arg1, arg2 refer to data objects, relop is a binary operator (< = , > =, <, >, != etc.) that compares arg1 to arg2 and indicates true/false, L is a label of the next instruction to be executed if relop indicates true
6.	Procedure Call	Param a, call x,y return b	a is a parameter x is a procedure, y is the number of parameters b is an optional return value.
7.	Address and Pointer Assignments	arg1 := &arg2, arg1 := *arg2, or *arg1 :=arg2;	arg1 is set to the location of arg2, arg1 is set to the value at location arg2, the value pointed by arg1 is set to the r-value of arg2

6.6 Syntax-Directed Translation Into Three-Address Code

When syntax-directed translation of three-address code is generated, temporary nodes are created for the interior nodes of a syntax tree. The value of non-terminal **E** on the left hand side of $E \rightarrow E_1 + E_2$ will be computed into a new temporary variable t.

The S-attributed definition in Fig. 6.6.2 generates three-address code. It uses two attributes:

1. **S.data** - the name that will hold the value of S
2. **S.offset** - hold the three address code statements that evaluate S (this is the 'translation' attribute).

Scanned by CamScanner

Productions	Synthesized attributes	
$S \rightarrow id := E$	S.code	three-address code for S
| while E do S	S.begin	label to start of S or nil
$E \rightarrow E + E$	S.after	label to end of S or nil
| $E * E$	E.code	three-address code for E
| $* E$	E.place	a name holding the value of E
| (E)		
| id		
| num		

Code generation → gen(E.place ':=' E_1.place '+' E_2.place)

$t_3 := t_1 + t_2$

Fig. 6.6.1 : Syntax Directed Translation into Three-Address Code

- The function ***newtemp*** that returns a new temporary variable like $t_1, t_2 \ldots\ldots t_n$ that we can use in response to the successive calls for intermediate computations.
- The function ***gen*** generates three-address code such that :
  1. Everything quoted is taken literally;
  2. The rest is evaluated.
- In practice, code can be sent to an output file instead of being assigned to the code attribute.

Production	Semantic rules

$S \rightarrow L := E$	if L.offset=null then
	emit (L.data':=' E.data);
	else
	emit (L.data'['L.offset']' ':=' E.data);

$E \rightarrow E_1 + E_2$	E.data : = newtemp; emit(E.data ': =' E1.data'+' E2.data);
$E \rightarrow E_1 * E_2$	E.data : = newtemp; emit (E.data ': =' E_1. data '*' E_2. data)

$E \rightarrow L$	if L.offset=null then
	E.data=L.data
	else
	begin
	E.data : = newtemp;
	emit (E.data':='L.data'['L.offset']');
	end

$E \rightarrow (E1)$	E.data := E1.data
$E \rightarrow id$	E.data:=id.data; E.offset:=Null

Fig. 6.6.2 : Syntax Directed Definition to produce Three-Address Code

Scanned by CamScanner

The code for $E \rightarrow$ ***while Expr do Stmt*** is generated using new attributes ***E.start*** and ***E.end*** to mark the first statement in the code for Expr and the statement following the code for Stmt, respectively.

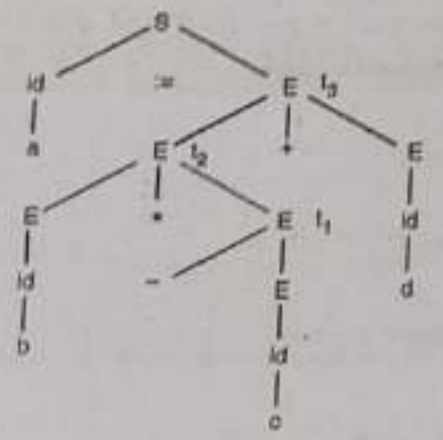
Production

```
E → while Expr do Stmt

Semantic rule
E.start := newtemp()
E.end := newtmp()
E.data := emit(E.start ':') ||
Expr.data ||
emit('if'  Expr.data  '='  '0'  'goto'
      E.end) || Stmt.data ||

emit('goto' E.start) ||
gen(E.end ':')
```

Fig. 6.6.3 : Semantic Rules for while statement

➢ **Example 1**

Given the assignment a := b* –c + d the code generated by the given grammar is :



t_1 := uminus c

t_2 := b + t_1

t_3 := t_2 + d

a := t_3

6.7 Code Optimization-Introduction

➔ **(MU - Dec. 2014, Dec. 2015, May 2016)**

Q.	Explain code optimization in compiler designing with suitable example.	Dec. 2014, 10 Marks
Q.	Explain the role of code optimization in compiler designing?	Dec. 2015, 10 Marks
Q.	Explain the different code optimization techniques along with an example.	May 2016, 10 Marks
Q.	What is code optimization? What are various strategies for code optimization?	(10 Marks)
Q.	Write note on : Code Optimization Techniques.	(10 Marks)

Optimization is the procedure of converting a piece of code into more efficient code in terms of time and space complexity. The transformed optimized code should have the following characteristics in it :

Scanned by CamScanner

- The output of the program should remain same without having any side-effects.
- The optimized code must execute faster and it should consume less memory.

Optimizing code is the core problem of compilers. Most of the research work is carried out for the problems of optimization. Optimizations are broadly classifieds into two types of optimizations; optimizations for speed and for memory space. High-quality optimization is more of an art than a science. Compilers for high level languages aren't judged by how well they parse or analyse the code but instead by the quality of the object code they produce.

The process of modifying a software system to make some aspect of it work more efficiently or use fewer resources is known as **program optimization** or **software optimization**.

In general, a computer program may be optimized so that it executes more rapidly, or is capable of operating with less memory storage or other resources, or draw less power.

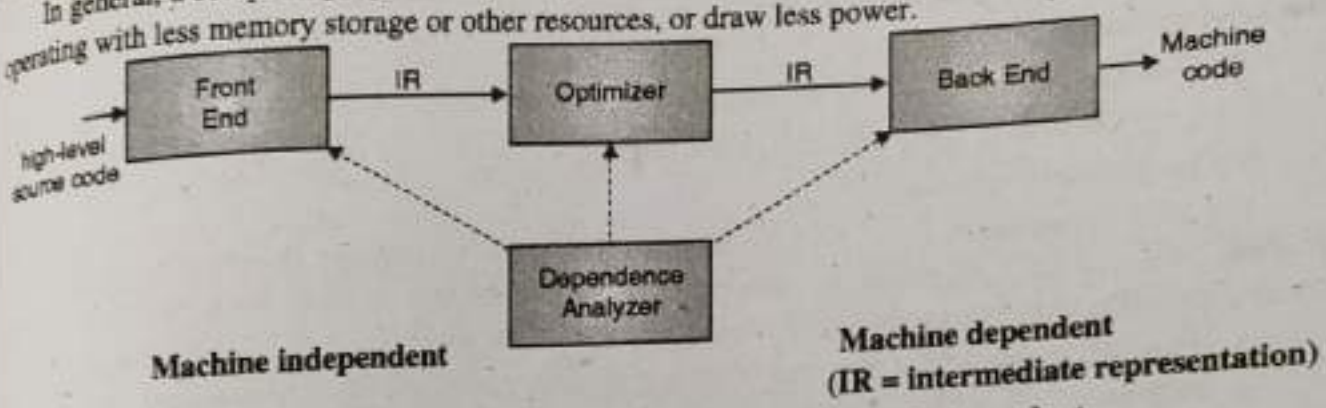


Fig. 6.7.1 : Compiler Structure showing the Machine Independent and Machine Dependent Optimization

Code Improvement (or Optimization)

The basic aim of code improvement is as :

- It analyses IR and rewrites (or transforms) IR.
- Its primary goal is to reduce running time of the compiled code.
- It may also improve space, power consumption.
- It must preserve "meaning" of the code
- It measures the name by the values.

Typical Transformations of Code Optimization Includes

- It discovers and propagates some constant value.
- It moves a computation to a less frequently executed place.
- It specialize some computation based on context.
- It discovers a redundant computation and removes it and also removes a useless or unreachable code.
- It encodes an idiom in some particularly efficient form.

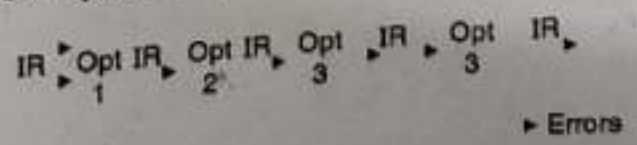


Fig. 6.7.2 : Modern Optimizers are Structured as a Series of Passes

Scanned by CamScanner

Most of the times, the phases of a compiler are collected into a *front-end* and a *back-end*. The ***front-end*** comprises of those phases or at times also parts of the phases which depend on the source language and are independent of the target machine. These include :

1. Lexical Analysis,
2. Syntactic Analysis,
3. Creation of Symbol- table,
4. Semantic Analysis and
5. Generation of Intermediate code.

It also includes some amount of error handling and code optimization that goes along with these phases. The back-end generally includes those phases of the compiler which depend on the target machine. They do not depend on the source language they depend on the just intermediate language. **Backend** includes

(i) Code Optimization,

(ii) Code Generation along with Error Handling and

(iii) Symbol-table operations.

The term optimization in a compiler may be applied to a technique designed to obtain more efficient object code that would be obtained by simple, straight forward code generators. Two constraints on the techniques used to perform code optimizations are

(i) They must ensure that the transformed program is semantically equivalent to the original program.

(ii) The improvement of the program efficiency must be achieved without changing the algorithms which are used in the program.

The input program is generally written in a high level language and the output program can be in a high level language or in a low level language. If both input and output programs are in high level language, the optimizer is known as ***source-to-source optimizer***. If the output is in the low level language, then the optimizer is known as an ***optimizing compiler***.

The advantage of a source to source optimizer is that its output is portable as it is in high level language which can be executed efficiently on any compiler system which has a compiler for that high level language. Whereas, the output of optimizing compiler is not portable as it is in a low level language.

The main **goal of optimization** is to produce target program with high execution efficiency. Optimization can be done at each level as follows :

- At the source code level by the user who can change algorithm or transform loops.
- At the intermediate code level by the compiler which can improves loops by improving address calculations.
- At the target code level by the compiler which can keep the most heavily used variables in registers.

6.7.1 Importance of Optimization

Due to various reasons like :

(i) Slow input operations,

(ii) Difference in the working speed of a human operator and a computer

Scanned by CamScanner

(iii) Availability of more and more source language options.

(iv) The volume of code required for newer applications.

The above mentioned are the major issues used in establishing the need of optimization of code. Following are the points which states the importance of the code optimization.

(i) As programming languages offers much flexibility, often leading to inefficient coding by the programmer. For example, the statement a = a + 1, where a, is of real type, requires a type conversion of 1 to 1.0 at run-time. An optimizer of the statement may modify it to a = a + 1.0 thus avoiding the run-time errors.

(ii) Optimization helps programmer to express their views clearly and allows programmers to write code in straight forward manner. It also allows programmer to select choices about implementation details that lead to an efficient code.

6.7.2 Is Programmer can Perform as Code Optimizer ?

The programmer can perform as a code optimizer the reason behind it are as :

(i) The selection of algorithm is done better by the programmer to solve by the program.

(ii) The programmer may have fair idea of branches being taken as he is aware about the data items in the program.

The reason behind the answer NO is as follows :

(i) Manual optimizations take too much time.

(ii) Certain machine level details are beyond the control of a programmer, namely, instructions and addressing modes supported by the target architecture.

6.7.3 Problems Faced in Optimizing Compiler

Design of an optimized compiler is further complicated due to following reasons :

(i) High level of redundancy is present in the source program as optimizing compiler takes the intermediate representation of a program code and replaces it with better version.

(ii) Optimized compilers usually have to take care of various objectives, for example different architectures, different syntax etc. making it difficult to optimize well for all cases.

(iii) A compiler typically deals with only a small part of an application at a time

These problems can be overcome by the another set of tool known as **post pass optimizer which works on the assembly language, rather** than the intermediate form.

6.7.4 Criterion for Code Optimization

Code optimization applies rules and algorithms to the source program with the goal of making the code faster, smaller, more efficient, and so on. Optimizations can be performed at various levels by various people like software developer, compiler optimizer etc. Often these types of optimizations conflict with each other: for instance, faster code usually ends up larger, not smaller. The important criterions for code optimization are as :

(i) Meaning must be preserved or semantic equivalence of the programs should be preserved.

(ii) The algorithm should not be modified.

(iii) Transformation, on an average should speed up the execution of the program.

Scanned by CamScanner

(iv) Transformation must be worth the effort: Intellectual and compilation effort should not spend on insignificant improvement.

(v) Transformations must be simple enough to have a good effect.

Correctness above All

It may seem obvious, but it bears repeating that optimization should not change the correctness of the generated code. Transforming the code to something that runs faster but incorrectly is of little value. It is expected that the un-optimized and optimized variants give the same output for all inputs. This may not hold for an incorrectly written program (e.g., one that uses an uninitialized variable).

When and Where To Optimize ?

To rearrange or to compress the intermediate code some techniques are applied to reduce the size of the syntax tree or shrink the number of three address code instructions.

6.7.5 Getting Better Performance through Optimization

Optimization can be done in almost all phases of compilation. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.

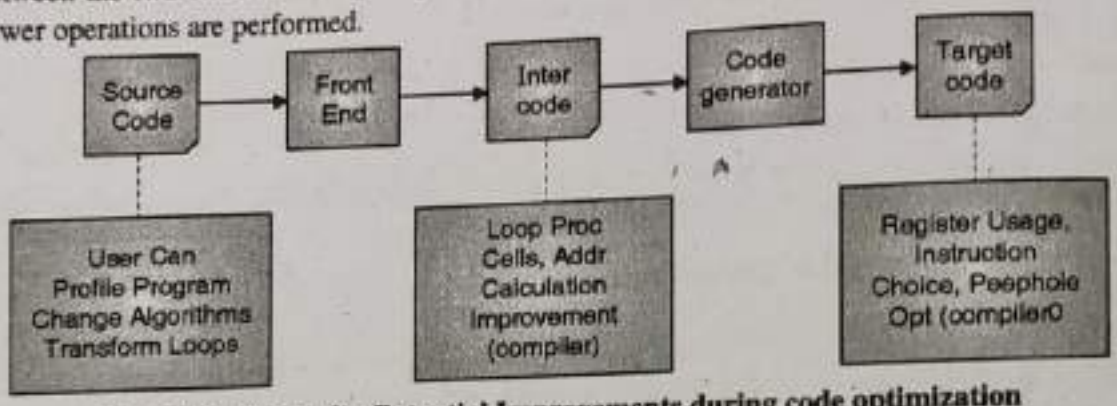


Fig. 6.7.3 : Places for Potential Improvements during code optimization by the User and the Compiler

The main phase of the back end includes the Analysis of the code: This phase includes the collecting of program information from the intermediate representation derived from the input and control flow graph is built during the analysis phase. The analysis phase is followed by the application of transformations.

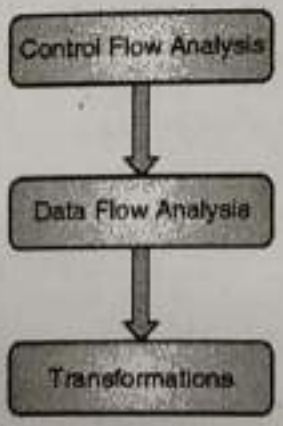


Fig. 6.7.4 : Code Optimizer Organization

Scanned by CamScanner

6.7.6 Factors Influencing Code Optimization

- Machine dependent factors can be parameterized to compiler for fine tuning.

1. Architecture of Target CPU :
   (i) Number of CPU registers
   (ii) RISC(Reduced Instruction Set Computers) vs CISC(Complex Instruction Set Computers)
   (iii) Pipeline Architecture
   (iv) Number of functional units
2. Machine Architecture :
   (i) Cache Size and type
   (ii) Cache/Memory and transfer rate

6.7.7 Aims behind the Optimization Techniques

The main aim behind the optimization techniques are to :

(i) Elimination of unnecessary work (henceforth "strength reduction").
(ii) Selection of the best patterns of instructions.
(iii) The code size should be smaller leads to: less work for CPU, cache, and memory.
(iv) As jumps interfere with code pre-fetch so code should has less number of jumps.
(v) Code Locality : codes executed close together in time are generated close together in memory – increase locality of reference.
(vi) Extract more information about code.

The first two aims overlap each other at the edges since selection of the best pattern of instructions involves preventing the processor doing unnecessary work. Below is a list of the various types of optimizations; the higher level optimizations will be first, with the lowest level of optimizations mentioned in last part

1. **Strength reductions**
   - Dead code elimination
   - Hoisting of invariants from loops
   - Common sub-expression elimination
   - Jump removal
   - Constant folding
   - Arithmetic simplification
   - Loop unrolling
   - Induction variable elimination
2. **Selection of efficient instruction patterns**
   - Loop fusion
   - Loop fission
   - Pre-fetching
   - Blocking

Scanned by CamScanner

- Instruction combination
- Instruction selection (peephole optimization)
- If conversion
- Register allocation
- Register movement elimination

3. Redundancy Elimination

If two computations are equivalent then eliminating one will not affect the program. There are two types of redundancy elimination techniques :

(i) **Value Numbering :** Identify the expressions that have same value and associates symbolic values to computations.

(ii) **Constant/Copy Propagation :** It helps in identifying literals that have constant values and uses the constants in that place of the variables.

6.8 Principal Sources of Code Optimization

→ **(MU - Dec. 2014, May 2016)**

Q.	Explain code optimization in compiler designing with suitable example.	Dec. 2014, 10 Marks
Q.	Explain different code optimization techniques along with an example.	May 2016, 10 Marks
Q.	Write note on : Code Optimization Techniques.	(10 Marks)
Q.	Explain the different code optimization techniques in compiler design.	(10 Marks)
Q.	Explain the role of code optimization in compiler designing?	(10 Marks)

This part of the chapter includes some of the most important code-improving transformations and also methods for applying these transformations. The design objectives for code optimizations are: compiler reliability, code generator portability and object code efficiency. Designing a compiler to meet the above objectives usually requires a lot of effort. To achieve these objectives of compiler reliability, one frequently tries to simplify the compiler design. **Transformation of a code can be performed locally or globally. A transformation of a program is local if it can be done by looking only at the statements in a basic block; otherwise the transformation is global.** Examples of global transformation are DAG, Data Flow Analysis etc. Local transformations are usually performed first.

The major principle source of optimization are as :

1. Function Preserving Transformation
2. Loop Optimization
3. Optimization for Basic Blocks
4. Reducible Flow Graph
5. Global Optimizations
6. Machine Dependent / Machine Independent Optimizations

6.8.1 Function Preserving Transformations

These transformation improves the code without changing the function. They are majorly four type of transformation they are as :

Scanned by CamScanner

1. Compile Time Evaluation
2. Common Sub-Expression Elimination
3. Dead Code Elimination
4. Copy Propagation

6.8.1(A) Compile Time Evaluation

In this technique expressions whose values can be pre-computed at the compilation time and used, perhaps repeatedly, in the execution of the program are identified. Then the computations of such expressions are used during compilation rather than during execution.

There are two possible ways in which compile time evaluation can be carried out :

1. Constant folding
2. Constant propagation

1. Constant Folding

Evaluation of an expression with constant operands to replace the expression with single value.

Example 1

area := (22.0/7.0) * r ** 2 is replaced with
area := 3.14286 * r ** 2

2. Constant Propagation

Replace a variable with constant which has been assigned to it earlier.

Example 2

pi := 3.14286 area = pi * r ** 2
⟶ area = 3.14286 * r ** 2

What does Constant Propagation means ?

- Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
  Similar to copy propagation.
- Extra Feature : It can analyse constant-value conditions to determine whether a branch should be executed or not.

When is it performed ?

- Early in the optimization process.

What is the result of Constant Propagation ?

- Smaller code
- Fewer registers

6.8.1(B) Common Sub-Expression Elimination

Two operations are *common* if they produce the same result. In such case, it is likely more efficient to compute the result once and reference it to the second time rather than re-evaluate it. An expression

Scanned by CamScanner

is *alive* if the operands used to compute the expression have not been changed. An expression that is no longer alive is *dead*.

OR

Identify common sub-expression present in different expression, compute once, and use the result in all the places. The *definition* of the variables involved in sub-expression elimination should not change.

➢ **Example 3**

a := b * c		temp := b * c
....	→	a := temp
...		...
x := b * c + 5		x := temp + 5

➢ **Example 4**

```
double x = d * (lim / max) * sx;
double y = d * (lim / max) * sy;
double depth = d * (lim / max);
double x = depth * sx;
double y = depth * sy;
```

6.8.1(C) Dead Code Elimination

Dead Code is a portion of the program which will not be executed in any path of the program. If an instruction's result is never used in the program, the instruction is considered to be "dead" and can be removed from the instruction stream. So if we have

tmp1 = tmp2 + tmp3;

If **tmp1** is never used again, we can eliminate this instruction altogether. However, we need to be careful and check whether tmp1 holds the result of a function call :

tmp1 = swap( );

Even if **tmp1** is never used again, we cannot eliminate the instruction because we can't be sure that called function has no side-effects. Dead code can occur in the original source program but is more likely to have resulted from some of the optimization techniques run previously.

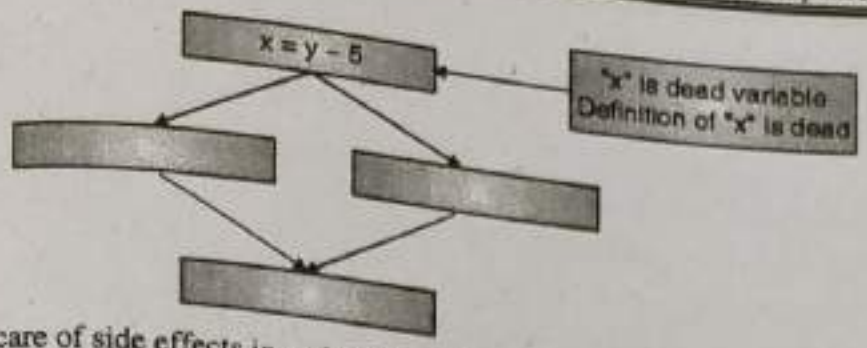
➢ **Example 5**

Program structures where Dead-Code elimination can be applied are as :

- If there is no control flows into the basic block.
- If the value of the variable is not used anywhere in the program.
- If an assignment assigns a value to a dead variable.

Scanned by CamScanner

Example 6



Always take care of side effects in code during dead code elimination.

Examples 7

(a) DEBUG:=0
if (DEBUG) print ← Can be eliminated

(b) i = 0
y (i = 1)
{ a = x + 5;
}

Here if statement in a dead code, as the condition will never get satisfied.

6.8.1(D) Copy Propagation

This optimization is similar to constant propagation, but generalized to non-constant values. Copy propagation is the process of removing them and replacing the copies generate by the compilers in an intermediate form. It often reveals dead-code.

In the optimized version of the code, we eliminated the unnecessary copies and propagated the original variable.

Example 8

tmp2 = tmp1 ;
tmp3 = tmp2 * tmp1;
y=x
z=3+y

⇩

tmp3 = tmp1 * tmp1 ;
z = 3 + z

When Code Propagation is performed ?

- At any level, but usually early in the optimization process.

What is the result of Copy Propagation ?

- Smaller code

In copy propagation f := g are called copy statements or copies. We can use of g for f, whenever possible after copy statement.

Scanned by CamScanner

➢ **Example 9**

x[i] = a; ⟶ x[i] = a;

sum = x[i] + a; sum = a + a;

Local copy propagation : It is performed within basic blocks.

- Algorithm sketch for Local Copy Propagation :
  - Traverse Basic Block from top to bottom.
  - Maintain table of copies encountered so far.
  - Modify applicable instructions as we go.

6.8.2 Loop Optimizations

➔ (MU - May 2015, Dec. 2016)

Q. Explain loop optimization with example. **May 2015, 5 Marks**

Q. Explain various loop optimization techniques with example. **Dec. 2016, 10 Marks**

Loops plays a very important role in code optimizations. Since the body of the loop is executed again and again, and moreover the loops are nested so small changes in the execution of the loop body can results in a remarkable improvement in the performance of the program.

Loop optimization can be viewed as the application of a sequence of specific **loop transformations** (listed below) to the source code or intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). It is most important set of optimizations in this programs are likely to spend more time in loops. The commonly used loop optimization techniques are :

1. Code Motion	2. Strength Reduction
3. Induction Variables and Strength Reduction	4. Loop Induction Variable
5. Loop Fission	6. Loop Fusion or Loop Combining
7. Loop Interchange	8. Loop Unrolling
9. Loop Splitting	

6.8.2(A) Code Motion

➔ (MU - May 2015)

Q. Write short note on : Code motion. **May 2015, 5 Marks**

Code motion (also called *code hoisting*) unifies sequences of code common to one or more basic blocks to reduce the code size and potentially avoid expensive re-evaluation. The most common form of code motion is *loop-invariant* code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. In code motion **invariant expressions should be executed only once.**

OR

In Code Motion the code moves from one part of the program to other without modifying the algorithm. The two advantages of using it are as :

- It reduces the size of the program.
- It reduces the execution frequency of the code subjected to the movement.

Scanned by CamScanner

Example 10

```
for (int i = 0; i < x.length; i++)
x[i] *= Math.PI * Math.cos(y);
```

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
x[i] *= picosy
```

A loop is a set of basic blocks which satisfies two conditions :

1. All blocks are strongly connected, i.e. there is a path between any two blocks.
2. The set has a unique entry point, i.e. every path from outside the loop that reaches any block inside the loop enters through a single node.

Code Motion is of two types :

1. ***Code Space Reduction*** **:** Similar to common sub-expression elimination but with the objective to reduce code size.

Example 11

Consider

```
if (a < b) then
    z := x ** 2
else
    y := x ** 2 + 10
```

```
temp : = x ** 2
if (a < b) then
z := temp
else
y := temp + 10
```

"x ** 2" is computed once in both cases, but the code size in the second case reduces.

2. ***Execution Frequency Reduction*** **:** It reduces execution frequency of partially available expressions (expressions available at least in one path).

Example 12

```
if (a < b) then
z = x * 2

else
y = 10

g = x * 2
```

```
if (a < b) then
temp = x * 2
z = temp
else
y = 10
temp = x * 2
g = temp;
```

In code motion the expression is move out of the loop if the evaluation does not change inside the loop.

Scanned by CamScanner

➢ **Example 13**

```
while ( i < (max-2) ) ...
```

Equivalent to :

```
t := max - 2
while ( i < t )
```

6.8.2(B) Strength Reduction

Operator strength reduction replaces an operator by a "less expensive" one. Given each group of identities below, which operations are the most and least expensive, assuming **f** is a float and **i** is an int ? (Trick question : it all depends on the architectures we need to know the target machine to optimize well!)

➢ **Example 14**

```
i*2 = 2*i = i+i = i << 1
i/2 = (int)(i*0.5) 0-1 = -i
f*2 = 2.0 * f = f + f
```

➢ **Example 15**

```
                          temp = 5;
for i=1 to 10 do          for i=1 to 10 do
...                       ...
x = i * 5      ------>    x = temp
...                       ...
                          temp = temp + 5
end                       end
```

- Typically strength reduction occurs in address calculation of array references.
- Applies to integer expressions involving induction variables (loop optimization)
- This eliminates the multiplication entirely and reduces the need for an extra temporary. By re-writing the loop termination test in terms of *temp*, we could remove the variable i entirely and not bother tracking and incrementing it at all.

6.8.2(C) Induction Variable Strength Reduction

Basic induction variables are created from derived induction variables. The rules required for creating the basic induction variables are as :

x := y op *z; It states that*

- *op* can be any *, <<, +, or –
- *y* is an induction variable and *z* is an invariant
- x is a register and no other statement modifies *x*
- *x* is not *y* or *z*

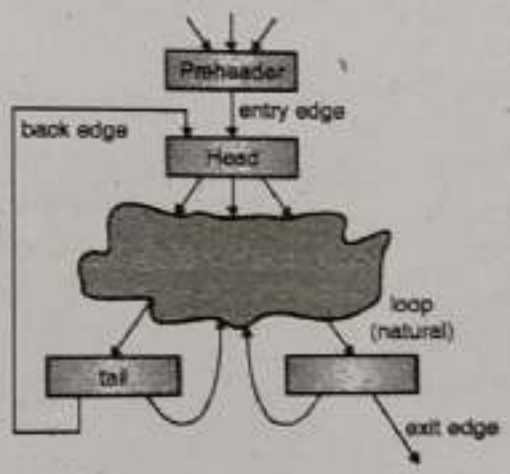
Loop Header : It is also known as the entry point of the loop. It dominates all blocks in the loop body.

Scanned by CamScanner

Loop Pre-Header : For example block B is a dominator with several incoming edges. , some of them being back edges (so M is a loop header). Create a loop preheader - a basic block before the loop header block. When a basic block is divided into two parts B_{pre} and B_{loop} the contents of B and back edges are moved to B_{loop}, the rest of the edges are moved to point into B_{pre}, and a new edge from B_{pre} to B_{loop} is inserted. B_{pre} is called the loop pre-header, and B_{loop} would be the loop header.

Loop Terminology

Loop	:	This is strongly connected component of a control flow graph
Loop entry edge	:	Source entry not in loop but target in loop
Loop exit edge	:	Source entry in loop but target not in loop
Loop header node	:	Target of loop entry edge
Natural loop	:	Loop made up of single loop header
Back edge	:	Target is loop header & source is in the loop
Loop tail node	:	Source of back edge
Loop preheader node	:	Single node that's source of the loop entry edge
Nested loop	:	Loop whose header is inside another loop
Reducible flow graph	:	CFG whose loops are all natural loops



Example 16

The following example shows the Induction Variable Strength Reduction :

```
extern int sum;
int foo(int n) {
 int i, j;
 j = 5;
 for (i=0; i < n; ++i) {
   j += 2;
   sum += j;
 }
 return sum;
}
```

```
extern int sum;
int foo(int n) {
int i;
for (i=0; i < n; ++i) {
 sum += (5 + 2*(i+1));
}
return sum;
}
```

Scanned by CamScanner

6.8.2(D) Loop Induction Variable

Induction variables values are either incremented or decremented whenever the value of the variable is changed. There are three types of induction variable :

- **Basic Induction Variable :** These type of variables are assigns within a loop. There are two types of induction variable. For example

  K = K +/– D where D is a constant.

- **Primary Induction Variable :** The loop execution is controlled in it.

  (for j=0; j<100; j++)

  i is the primary induction variable.

- **Derived Induction Variable :** Derived variable is a linear function of a basic induction variable.

➢ **Example 17**

In following example the different type of variables are as :

Basic : r4, r7, r1

Primary : r1

Derived : r2

	r1 = 0 r7 &A
Loop :	r2 = r1 * 4 r4 = r7 + 3 r7 = r7 + 1 r10 = *r2 r3 = *r4 r9 = r1 * r3 r10 = r9 >> 4 *r2 = r10 r1 = r1 + 4 If (r1 < 100) goto Loop

Here if r1 is incremented by 1 then r2 gets incremented by 4. Hence r1 is induction variables.

6.8.2(E) Loop Fission

Loop Fission tries to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. It improves the locality of reference.

➢ **Example 18**

```
for j=1 to 50 do
        a[j]=...
        b[j]=...
        c[j]=....
        end loop
```

Scanned by CamScanner

Here each value of i the program first refers to an element of the array a and then to b and then to another element from the array c. If there is no data dependency between these three statements in the loop body, then it can be broken into three loops as shown :

```
for j=1 to 50 do
        a[j]=...
end loop
for j=1 to 50 do
        b[j]=...
end loop
for j=1 to 50 do
        c[j]=...
end loop
```

6.8.2(F) Loop Fusion or Loop Combining

The transformation opposite to loop fission is loop fusion. When two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data.

6.8.2(G) Loop Interchange

These optimizations exchange inner loops with outer loops.

6.8.2(H) Loop Unrolling

It duplicates the body of the loop multiple times, in order to decrease the number of times the loop conditions needs to be tested. It also reduces the number of jumps from the end of the loop to the beginning.

6.8.2(I) Loop Splitting

Loop Splitting simplifies a loop or eliminate dependencies by breaking the loop into multiple loops which have the same bodies but iterate over different portions.

Example 19

```
for j=1 to 50 do
        if j < 10 then B1
        if  10 <= j < 30 then B2 else B3
end loop
```

The loop can be split into three loops as follows.

```
        for j= 1 to 9 do
            B1
end loop
```

Scanned by CamScanner

```
        for j = 10 to 29 do
            B2
end loop

        for j = 30 to 50 do
            B3
end loop
```

A useful special case is *loop peeling* - in this erroneous loop is executed separately before entering the main loop.

6.8.3 Optimization for Basic Blocks

A basic block is a linear piece of code in which flow of control moves in at the beginning and exit at the end without halt or possibility of branching except at the end. The analysis and optimization of the basic blocks are easier.

The following sequence of three-address statements forms a basic block :

```
t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5
```

A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block. Optimization in the basic blocks are done in four ways they are as :

(A) Common Sub-Expression Elimination.

(B) Eliminate Dead Code.

(C) Algebraic Optimizations.

6.8.3(A) Common Sub-Expression Elimination

This type of optimizations are explained with the help of an example mentioned below.

➢ **Example 20**

Consider the Three-Address-Code and flow graph for Quick Sort mentioned in Fig. 6.8.1. Use this flow graph to eliminate the common sub-expression.

Scanned by CamScanner

Common sub-expression elimination

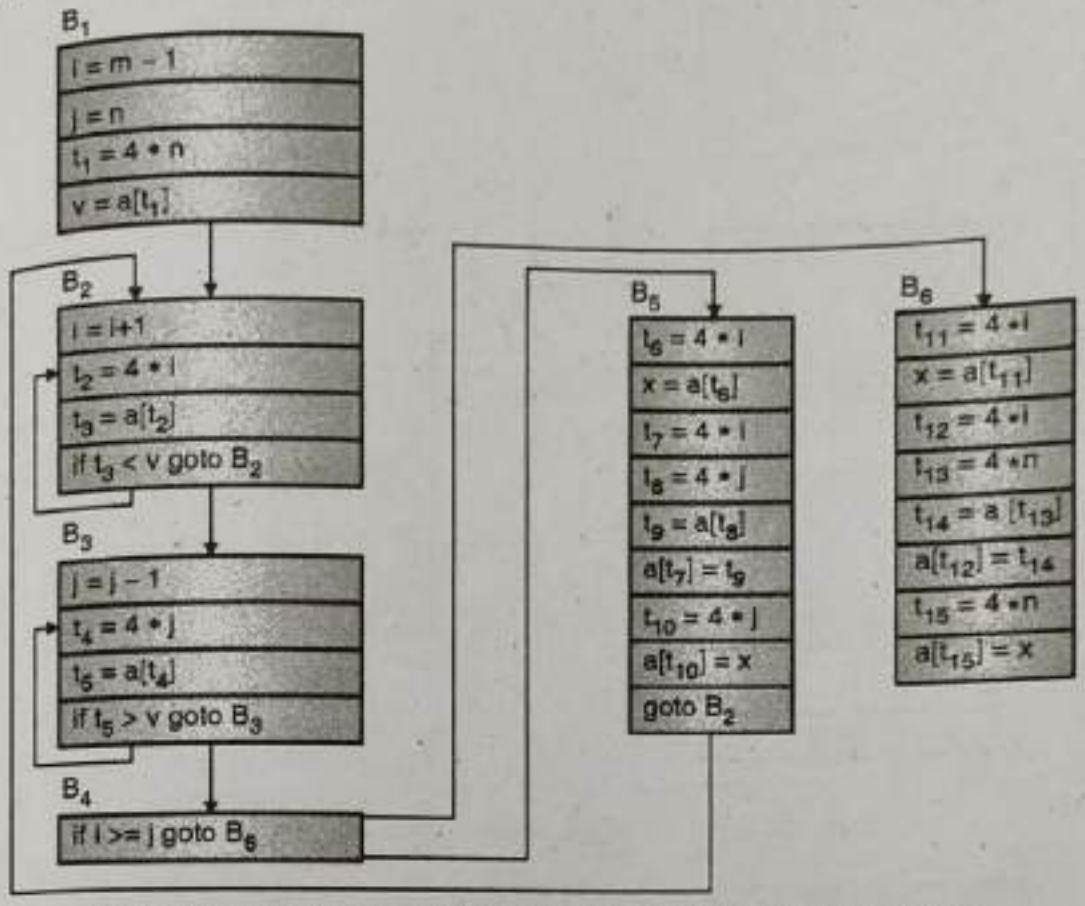
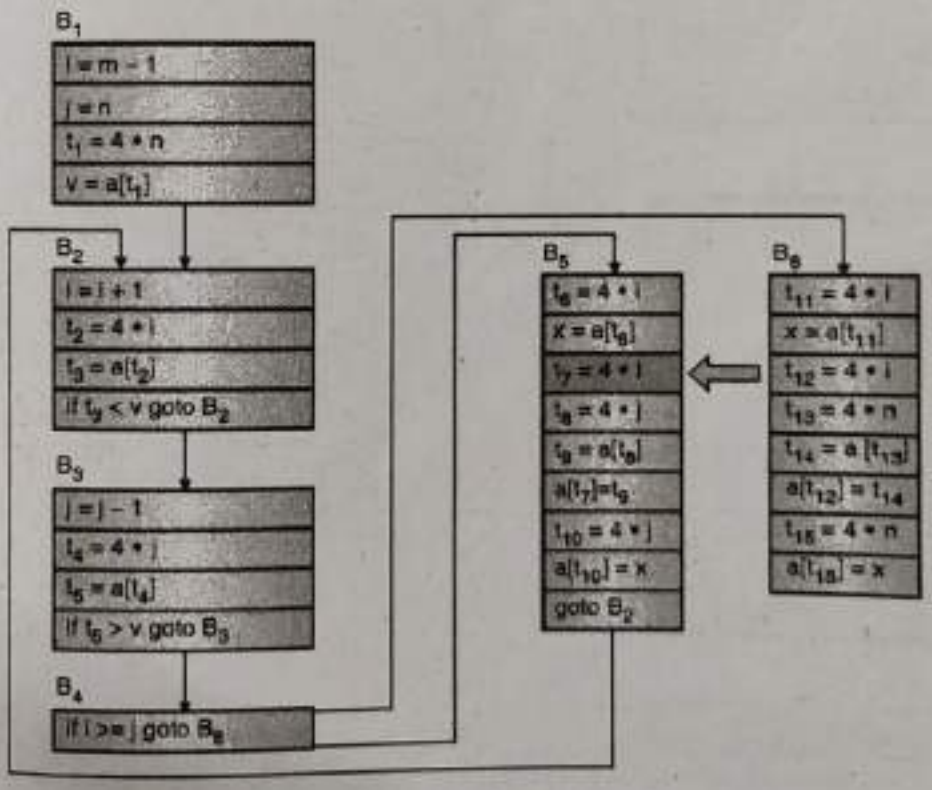
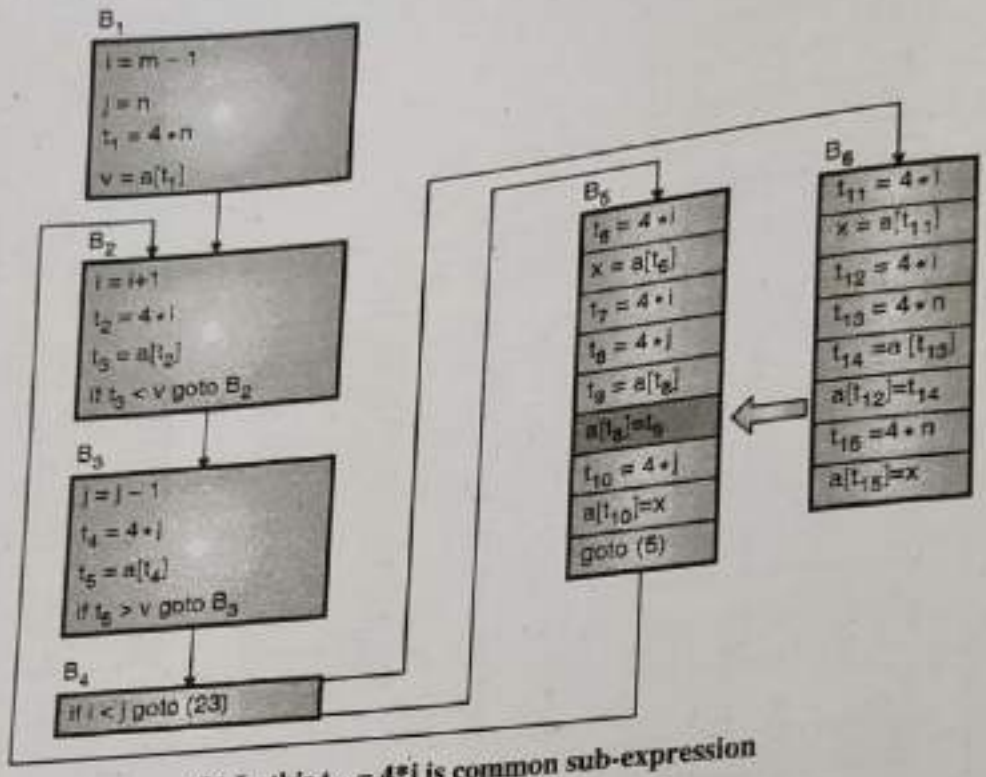


Fig. 6.8.1 : Three Address Code and Flow Graph for Quicksort

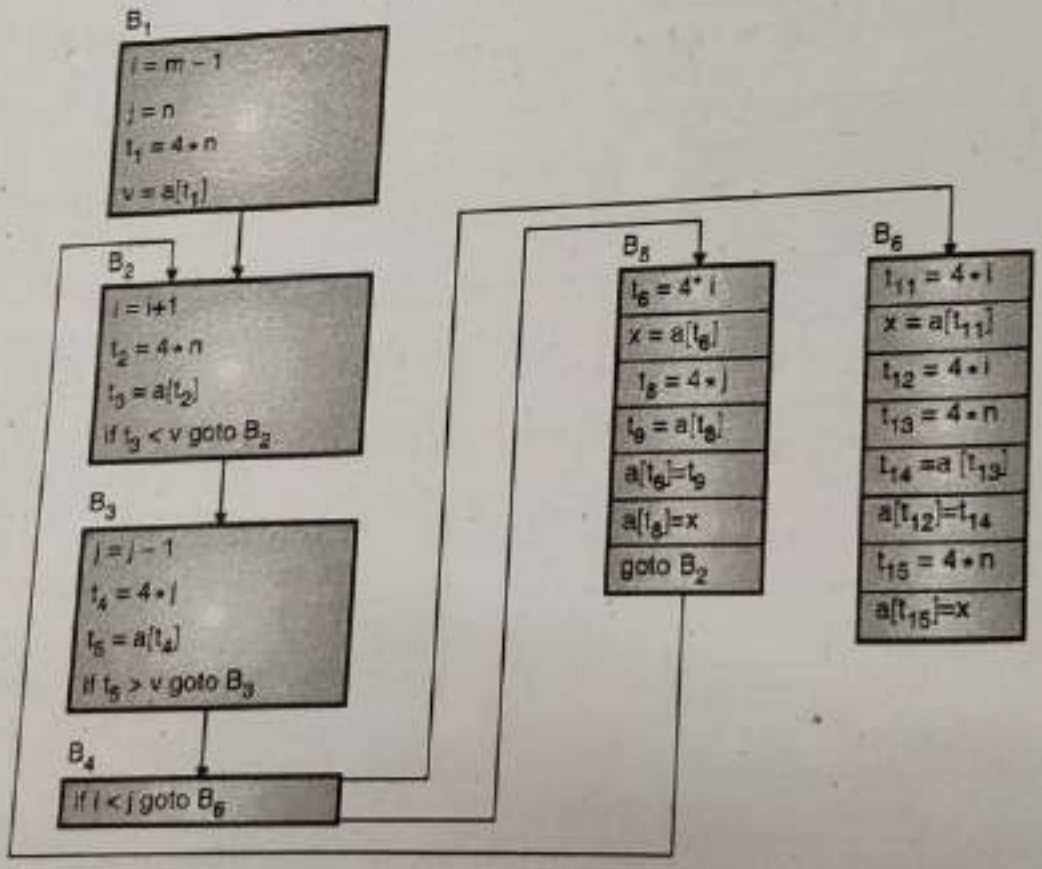


(a) In this $t_7 = 4*i$ is common sub-expression

Scanned by CamScanner

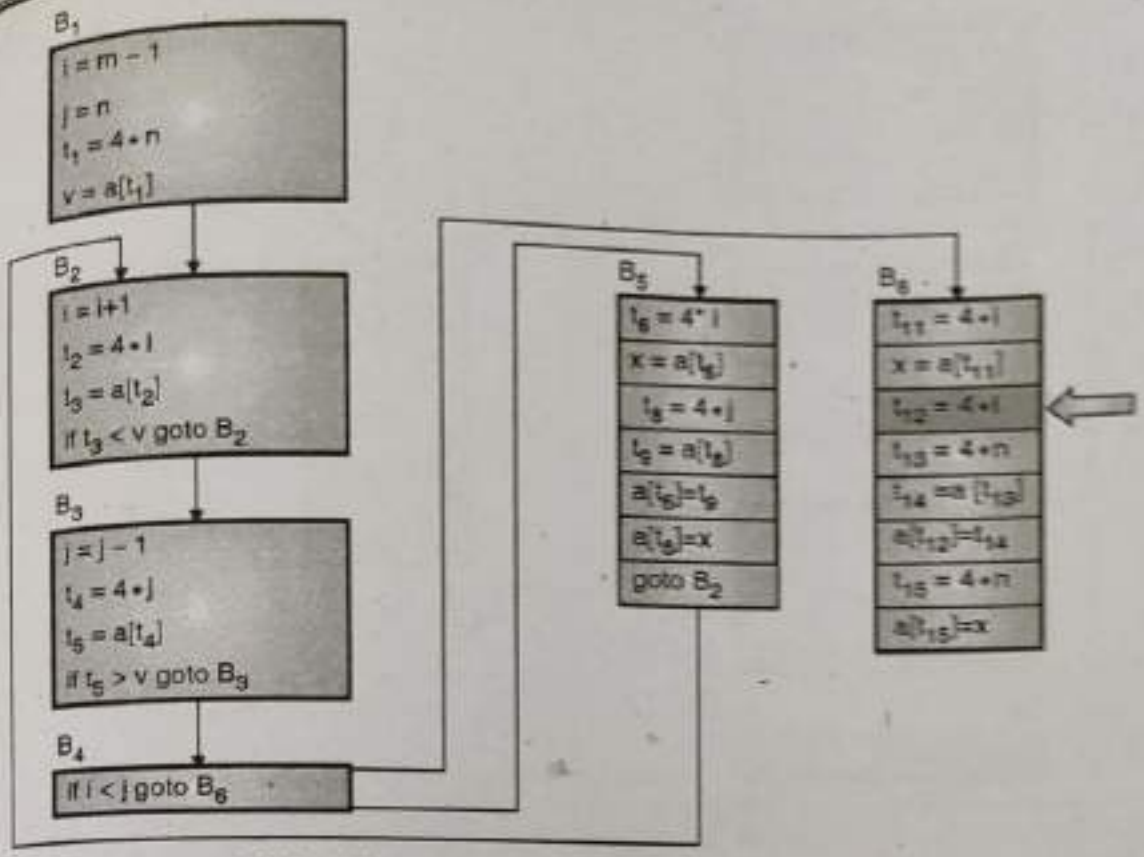


(b) In this $t_{10} = 4*j$ is common sub-expression

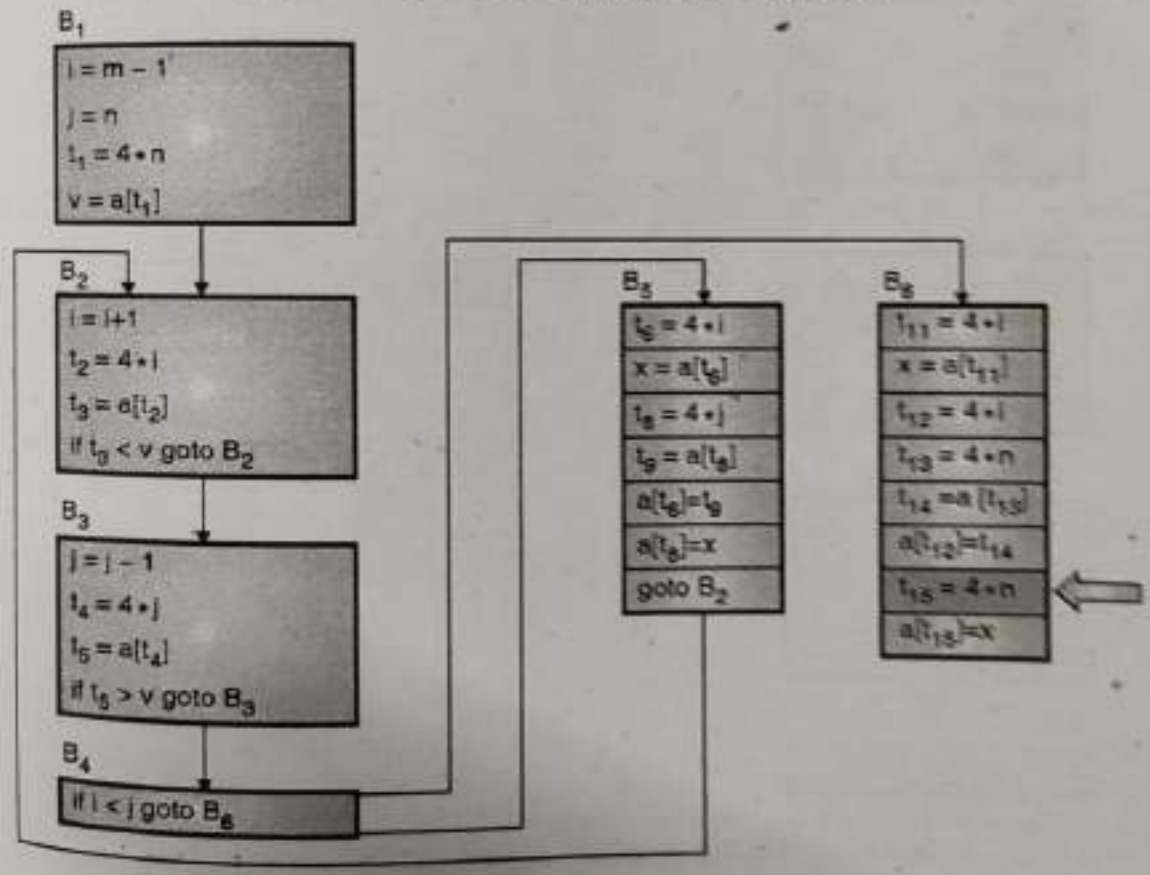


(c) In this common sub-expression t_7 and t_{10} is eliminated

Scanned by CamScanner

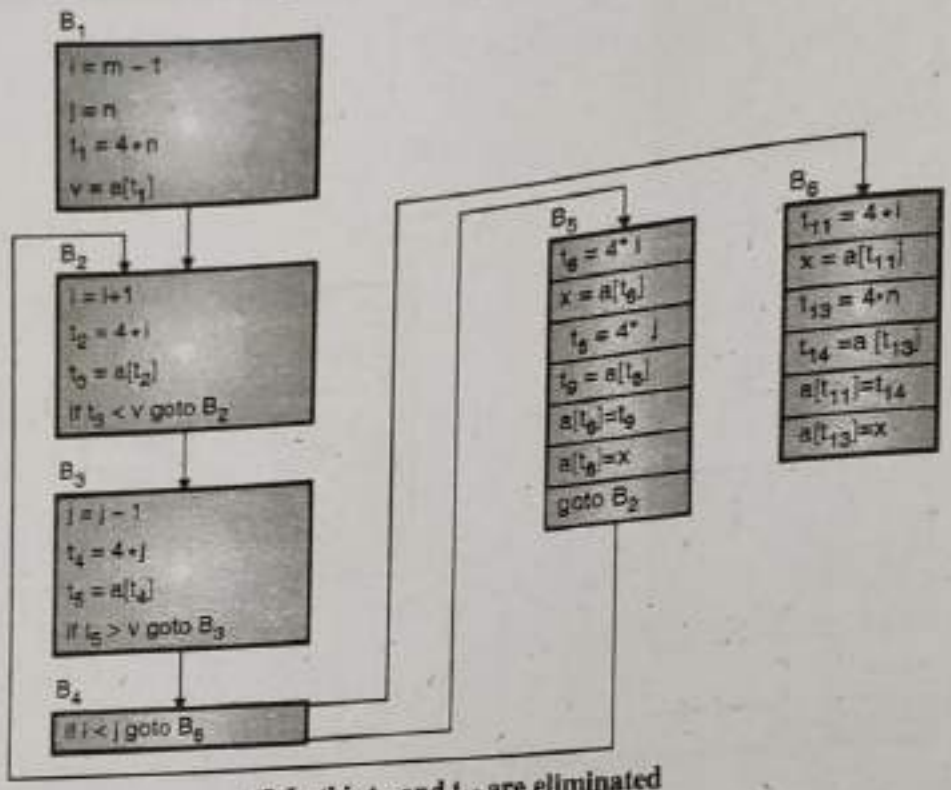


(d) In this t_{12} = 4*i is a common sub-expression

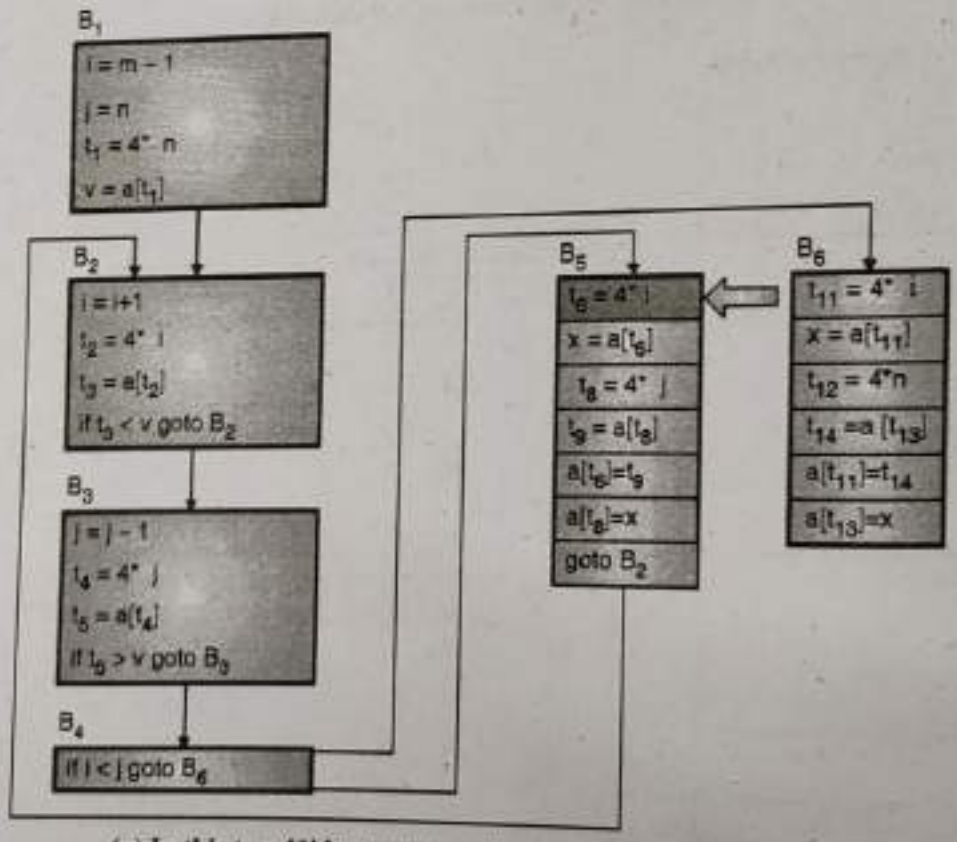


(e) In this t_{15} = 4*n is a common sub-expression

Scanned by CamScanner

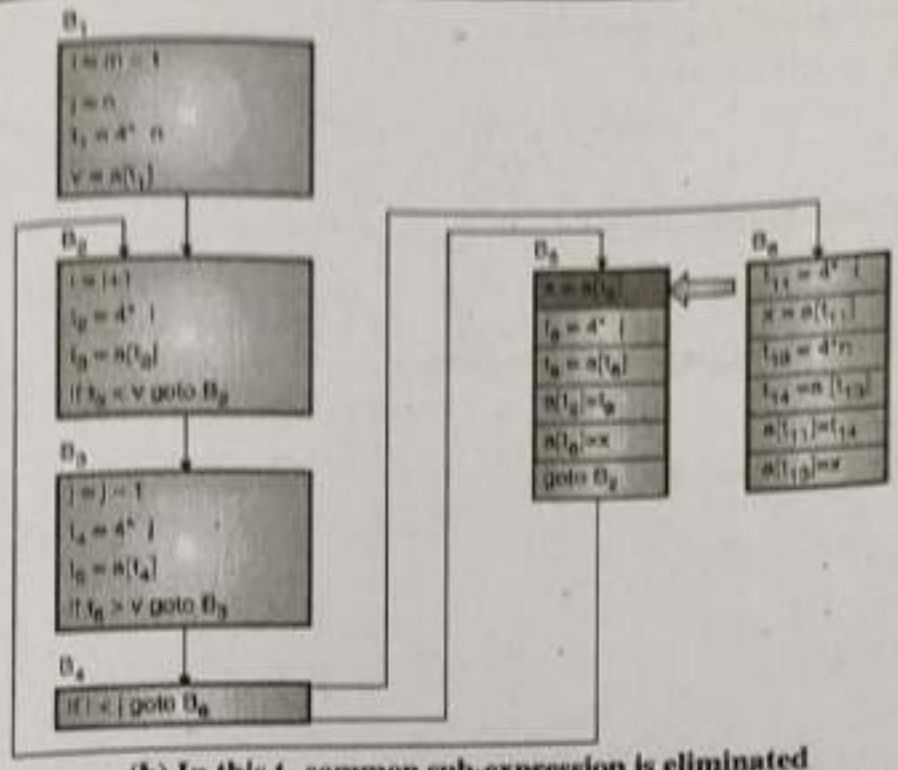


(f) In this t_{12} and t_{15} are eliminated



(g) In this $t_6 = 4*i$ is common sub-expression in Block B2 and B5

Scanned by CamScanner



(h) In this t_6 common sub-expression is eliminated

Fig. 6.8.2 : Common Sub-Expression Elimination for Three Address Code for Quick sort

There are two types of Sub-Expression Elimination :

- Local Common Sub-Expression Elimination
  - Performed within basic blocks
  - Algorithm sketch
  - Traverse Basic Block from top to bottom.
  - Maintain table of expressions evaluated so far.
  - If any operand of the expression is redefined, remove it from the table.
- Modify applicable instructions as you go

  Generate temporary variable, store the expression in it and use the variable next time the expression is encountered.

➢ **Example 21**

c = a + b	t1 = a + b
d = m * n	c = t1
e = b + d	t2 = m * n
f = a + b	d = t2
g = – b	t3 = b + d
h = b + a	e = t3
a = j + a	f = t1
k = m * n	g = – b
j = b + d	h = t1 /* commutative*/
a = – b	a = j + a
if m *n go to L	k = t2
	j = t3
	a = – b
	if t2 go to L

Scanned by CamScanner

Global Common Sub-Expression Elimination

- It is performed on flow graph
- It requires available expression information and where the expression was most recently evaluated.

6.8.3(B) Dead Code Elimination

It is explained in section 6.8.3(A).

6.8.3(C) Algebraic Identities Simplification

These include :

Taking advantage of algebraic identities like

(x*1) is x

Strength reduction

(x*2) is (x << 1)

Simplifications such as

– (– x) is x

(1 || x) is true

(1 && x) is x

*(& x) is x

a=b+1
c=a-1 ⟹ c=b

6.8.4 Reducible Flow Graphs

A flow graph is reducible when the edges can be partitioned into forward edges and back edges. The forward edges must form an acyclic graph in which every node can be reached from the initial node. Structured control-flow statements such as if-then-else, while and break produces reducible control-flow. Irreducible flow graphs cannot be optimized as they create loops. With the help of the duplication code irreducible flow graphs can always be converted into reducible flow graph.

6.8.5 Global Optimizations

The scope of local optimization is limited to short code sequences, whereas the scope of global optimization is a program unit like a procedure or a function body including entire loops. This required considerable analysis of the program structure, but its gains are much higher than those of local optimizations.

In order to perform a global optimization, the program is to be represented in the form of a program flow graph. Two types of analysis are performed on the program flow graph. (A) Control flow analysis and, (B) data flow analysis.

6.8.5(A) Control Flow Analysis

It determines information concerning the structure of the program like the presence of the loops, nesting of the loops, conditional and unconditional jumps, nodes visited before the execution reaches a **Specific node**.

Scanned by CamScanner

A flow graph is a directed graph in which the control flow information is added to the basic blocks.

(i) Basic block represents the nodes to the flow graph.

(ii) Initial block is the block whose leader is the first statement.

(iii) There is a directed edge from block B_1 to block B_2.

6.8.5(B) Data Flow Analysis

It determines information regarding the data flow in a program like how data items are assigned and referenced in a program, what are the values which are available when program execution reaches a specific statement of the program. Data flow analysis is basically a process of computing the values of a set of items of dataflow information which are useful for the purpose of optimizations.

6.8.6 Machine Dependent / Machine Independent Optimization

Q. Differentiate between machine dependent and machine independent code optimization. **(5 Marks)**

The code optimization phase attempts to improve the intermediate code so that faster running machine code will result. **An optimization is classified as machine dependent or machine independent.**

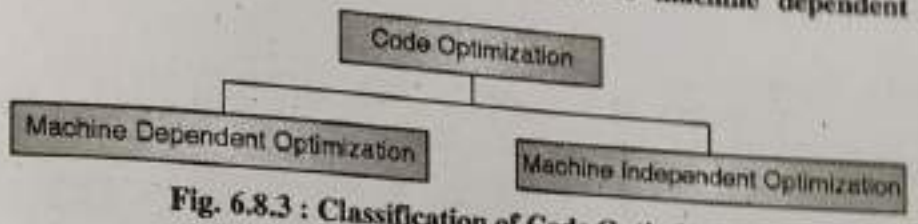


Fig. 6.8.3 : Classification of Code Optimization

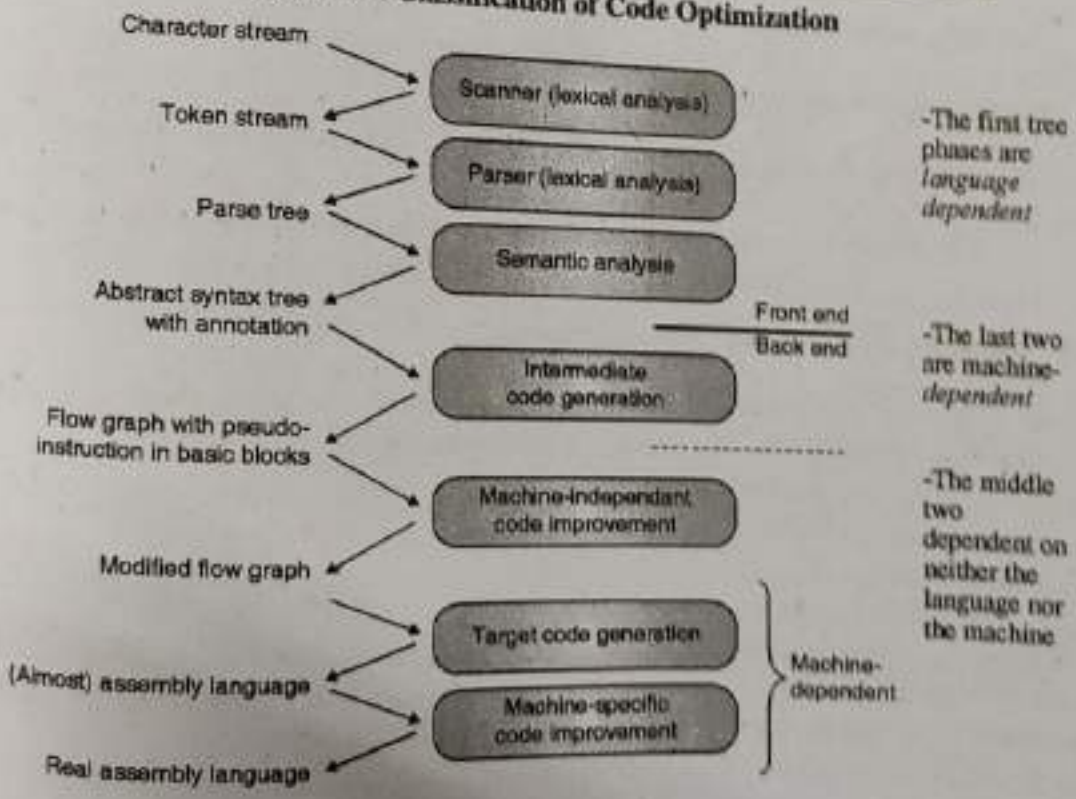


Fig. 6.8.4 : Phases of Compilation divided into Machine dependent Optimization

Scanned by CamScanner

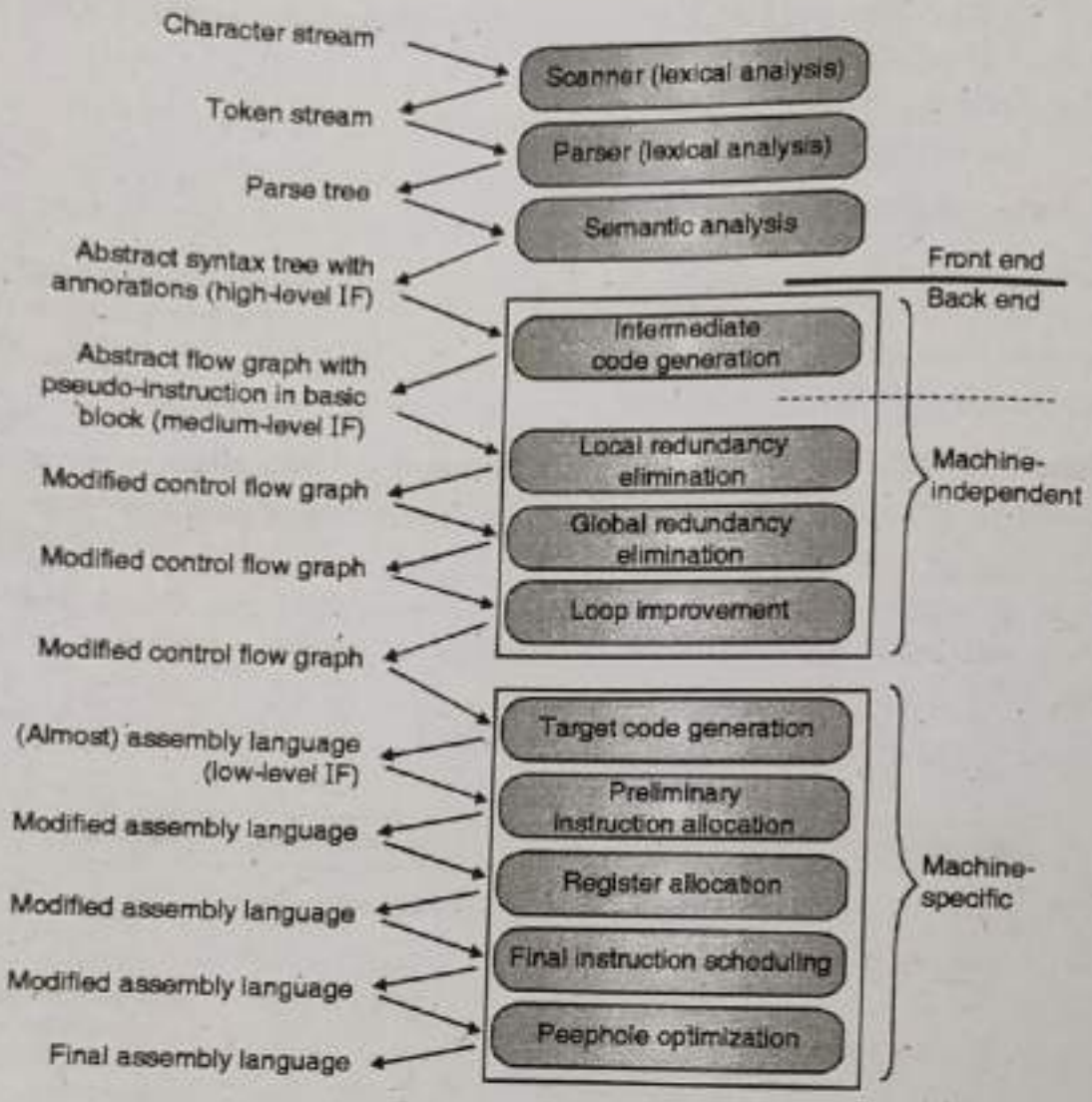


Fig. 6.8.5 : Compilation Phases showing the various techniques for Machine Independent Optimization

6.8.6(A) Machine Dependent Optimizations

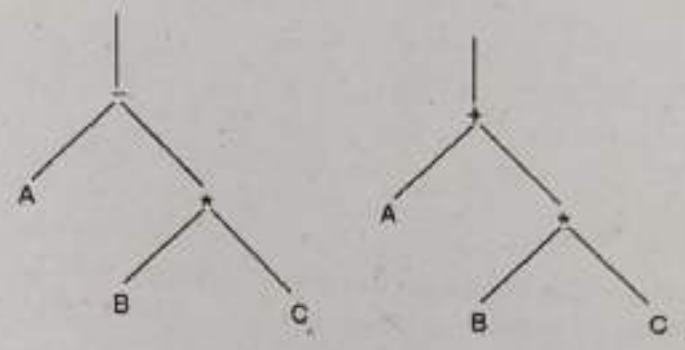
Machine Dependent Optimizations exploit characteristics of the target machine in terms of addressing structures and instruction set to reduce the amount of code of execution time.

During code generation, peculiarities of the target machine can be exploited to obtain a reduction in space and execution time. Examples of machine dependent optimizations include

1. Allocation of scarce resources to achieve high efficiency in program execution e.g. the top-of-stack registers, few arithmetic registers etc.
2. Use of immediate instructions where a value is a part of the instruction. Incrimination is a process where a memory location can be incremented by some constant, indexing or indirection and vector operations are some special machine features that can be gainfully exploited.
3. If data is intermixed with the instruction sequence it can be accessed more efficiently on some machines.

Scanned by CamScanner

Let us consider in Fig. 6.8.6 which illustrates a situation in which we can optimize code by using a LOAD COMPLEMENT instruction



(a) Before optimization **(b) After optimization**

Fig. 6.8.6

The code generated for the tree of Fig. 6.8.6(a) is as follows

LOAD	B
MULTIPLY	C
STORE	T1
LOAD	A
SUBTRCT	T1

Whereas the code generated for the tree of Fig. 6.8.6(b) is as follows.

LOAD COMPLEMENT	B
MULTIPLY	C
ADD	A

Machine Dependent Optimization Techniques differ from machine to machine. If machine is capable of parallel processing than compiler should generate code to take advantage of it. When different instructions are available to perform a task, the most efficient should be chosen.

There are several different possibilities for performing a machine-dependent code optimization.

1. **Assignment and use of registers**
   - Registers is used as instruction operand.
   - The number of registers available is limited
   - Required to find the least used register to replace with new values when needed.
   - Usually the existence of jump instructions creates difficulty in keeping track of registers contents.
2. **Problem is divided into the basic blocks**

   As it is already mentioned basic block is a sequence of straight line code that can be entered only at the beginning and exited only at the end. An arrow from one block to another indicates that control can pass directly from one quadruple to another. This kind of representation is called a **flow group.**
3. Rearrangement of machine instruction to improve the efficiency of execution.
4. **Register Allocation :** It assigns CPU registers for specific use and at the same time information is also maintained for different registers and their use.

Scanned by CamScanner

Main objective of Register Allocation :

- CPU registers utilization is maximized
- References to memory locations are reduced
- Importance of Register Allocation
- Efficiency
- Speed

Register Allocation Algorithm

Register Allocation Algorithm determines how many registers will be needed to evaluate an expression. It also determines the sequence in which sub-expressions should be evaluated to minimize register use.

Databases Required in Machine Dependent Optimization

- Matrix Table

Operator	Operand 1	Operand 2	Forward pointer	Backward pointer

Fig. 6.8.7 : Format of Matrix Table

- Identifier Table
- Literal Table
- Code Productions (Macro Definitions)

6.8.6(B) Machine Independent Optimization

Q. Differentiate between machine dependent and machine independent code optimization. **(5 Marks)**

Machine independent optimizations exploit characteristics of the programming language in terms of program structure and arithmetic properties of the operations involved. Machine Independent Optimizations are base on the mathematical properties of a sequence of source statements. An optimization essentially means analysing the overall purpose of the statement sequence and finding an equivalent sequence which can be translated to the minimum code. In this section we will discuss the methods of program structure analysis on which their application is based.

Databases Required in Machine Independent Optimization

- Matrix
- Identifier table
- Literal table

Optimization Techniques used in Machine Independent Optimizations

- Elimination of Common Sub expression.
- Invariant Code Motion.
- Reduction in Strength.
- Constant Propagation & Dead Code Elimination.

All these techniques are discussed in Section 6.8 (Sources of Optimization)

Scanned by CamScanner

6.9 Peephole Optimization

→ (MU - Dec. 2015)

Q. Explain Peephole optimization along with an example. Dec. 2015, 5 Marks

Peephole optimizations examine at most a few instructions, transforming instructions into other less expensive ones, such as turning a multiplication of x by two into an addition of x with itself. It works on a small set of instructions. The small set is called a "peephole" or a "window". It works by recognising sets of instructions that don't actually do anything, or that can be replaced by a leaner set of instructions.

Peephole optimization is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine dependent code improvements. For example, peephole optimizations might include elimination of multiplication by 1, elimination to load a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect. Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but it can significantly improve code at a very local level and can be useful for cleaning up the final code that resulted from more complex optimizations.

Common techniques used in peephole optimizations are as :

1. **Constant folding :** It finds out the constant sub-expressions in advance.
2. **Strength reduction :** Slow operations are replaced with faster equivalents.
3. **Null sequences :** Useless operations are removed
4. **Combine Operations :** Similar operations are replaced with one equivalent.
5. **Algebraic Laws :** Algebraic laws are used to simplify or to reorder instructions.

6.10 Local Optimization

Q. Explain local code optimization in detail. (5 Marks)

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information; we just work with the statements within the block.

Local optimization has a scope of a few program statements which are executed in an essentially sequential manner. Target code generated statement by statement generally contains redundant instructions. We can improve the quality of such code by applying optimizing transformations locally by examining a short sequence if possible. This technique is called peephole optimization where peephole is a small, moving window on the target program. In order to get the maximum benefits we can have repeated passes over the code. Though the gains of local optimization are limited, it is still considered significant by a practical optimizer. The cost of local optimization is also low, since the amount of program analysis required in order to perform local optimization is limited. Many of the code improving transformations discussed in the last section can be implemented by constructing a DAG (Directed Acyclic Graph) for a basic block. We may recall that a DAG differs from a tree in that it may contain shared nodes where shared nodes indicate common sub-expressions.

Scanned by CamScanner

Initially there is a node for each variable of a basic block which is identified by the symbol table entry of that variable. Each node is labelled with the names of the variables whose values it represents.

Also, there is a node n associated with each operation within the block. The children of n are the operands of the operation it is representing. Whenever a new node m is to be added, it is checked if a matching node n with the same children, in the same order already exists. A new node is not created. **At the node n, a list of variables is attached which contains the name of the left hand side variable if n is the root of the right hand side expression of an assignment statement**

6.11 Code Generation-Introduction

- In this chapter we will be dealing with the last phase of compiler design model i.e. code generation phase. The input given to the code generator is an intermediate representation of the source program and output is target program as indicated in Fig. 6.11.1.
- Code generation phase is after the syntax phase. The parser recognizes a portion of the source program according to the grammar a sub program is executed for that grammar. This sub program is called **semantic routine.**
- A semantic routine generates object code directly and are also called **code generation routines.** Some complex compilers generate an intermediate form of the program.

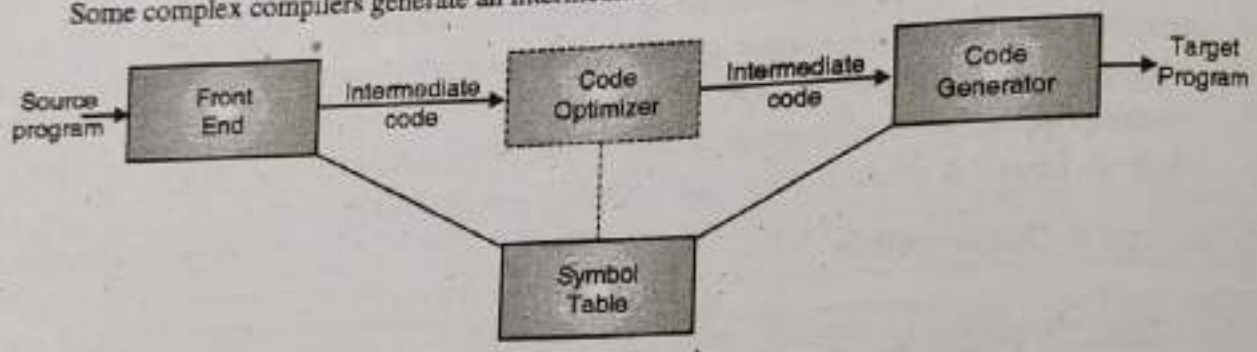


Fig. 6.11.1 : Code Generation Phases

- **Code generation** is the process by which a compiler's **code generator** converts some internal representation of the source code into a form (e.g., machine code) that can be readily executed by a machine. The input to the code generator typically consists of a **parse tree** or an **abstract syntax tree**.
- The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three address code.
- The characteristics of a good code generator are as :
  - The generated target code must be accurate and of good quality.
  - It should be efficient enough to generate an intermediate code.
  - Proper selection of heuristic is important in generating optimal code.
  - Details of code generator are dependent on target language and operating system.

The input to the code generator consists of :

- Intermediate code produced by the front end (and perhaps optimizer).

Scanned by CamScanner

- Intermediate code can come in many forms but we are concentrating on three-address code but several techniques apply to other possibilities as well.
- Information in the symbol table is used to determine runtime addresses of the data objects.

The code generator typically assumes that :

- Input is free of errors.
- Type checking has taken place and necessary type conversion operators have already been done.

The output of the code generator consists of :

The target program is the output of the code generator. The output can take a variety of forms like :

- Absolute machine language
- Re-locatable machine language
  - Can compile subprograms separately
  - Added expense of linking and loading
- Assembly language

Important tasks done by the code generator are as follows :

- Selection of Instruction: Proper selection of target-machine instructions is done to implement the IR statements
- Assignment and Allocation of Registers : In this we decide what values to keep in which registers.
- Ordering of Instructions: Execution order of instruction is decided.

5.12 Issues in the Design of Code Generator

→ (MU - May 2015, May 2016, Dec. 2017)

Q.	What are different issues in code generation, explain in detail.	May 2015, 10 Marks
Q.	What are the different issues in design of Code Generator ? Explain with an example.	May 2016, 10 Marks
Q.	Discuss different issues in design of Code Generator.	Dec. 2017, 10 Marks

1. Input data for the Code Generator

Below mentioned are the following formats used as input for code generator

(i) Three-Address Code (quadruples, triples, indirect triples)

(ii) Virtual machine presentations (byte-code, stack-machine)

(iii) Linear presentation (postfix, infix, prefix)

(iv) Graphical presentation (syntax trees, DAGs, parse tree, syntax directed translation)

2. Target Program

- Knowledge of machine architecture and instruction set are the pre-requisite for the design of good code generator.
- Proper selection of machine architecture helps in producing absolute machine-language program and re-locatable machine-language program

Scanned by CamScanner

3. Instruction Selection

The complexity of mapping IR program into code sequence depends on :

- Level of Intermediate Representation (high-level or low-level)
- Type and nature of instruction set.

4. Register Allocation and Assignment

Selection of set of variables that will reside in registers at each point and choosing specific register helps in faster execution of data.

5. Evaluation Order

Selecting the order in which computations are performed effects the efficiency of the target code.

6.13 Code Generation Algorithm

Code generation is a process by which a compiler's code generator converts some intermediate representation of source code into a form that can be readily executed by a machine.

Code generation algorithm uses the two stacks.

1. **Register stack :** stack of registers like $r_0, r_1 \ldots r_n$
2. **Temporary stack :** stack of temporaries like $t_0, t_1 \ldots t_n$.
3. A call to create code (n) generates code to evaluate a tree T, rooted at node n, into the register top (regstack) and the rest of the regstack remains the same state as the one before call
4. An interchange of the top two registers (regstack) is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.
5. All computations are carried out in registers. Code Generation Algorithm is done in two phases.

(i) Labelling phase
(ii) Code generation phase

(i) Labelling Algorithm

Each node of the tree is labelled with an integer. Few registers are required to evaluate the tree, for leaf nodes

Step 1 : if n is the leftmost child of its parent then

lable (n) : = 1

else lable (n) : = 0

Step 2 : for internal nodes

Label (n) = max (l_1, l_2), if $l_1 \mathrel{!}= l_2$

$= l_1 + 1$ if $l_1 = l_2$

Scanned by CamScanner

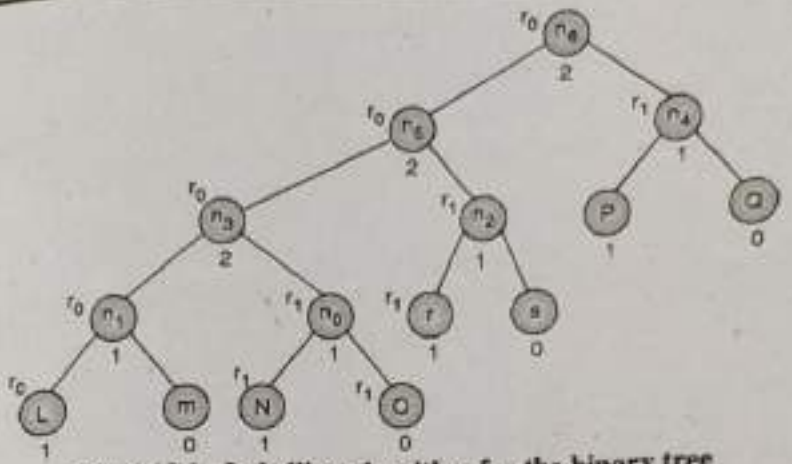


Fig. 6.13.1 : Labelling algorithm for the binary tree

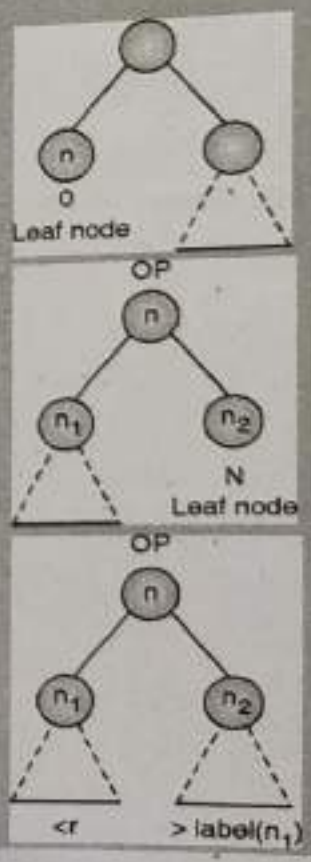
From the given labelling algorithm we have find the values at each node r_0 and r_1 are two register stacks.

Code generation phase

It uses Regstack, interchange tempReg and create code to evaluate the node.

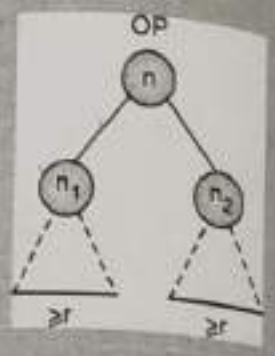
Code generation algorithm :

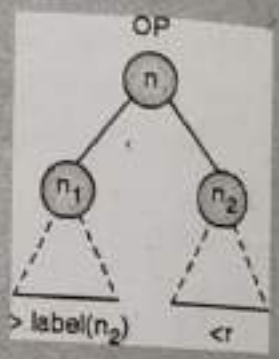
```
Procedure create code (n);    /* case 0 */
{
   if n = leaf with leftmost child of its parent and representing operant O
   then
   print (load 0, top (Reg stack))
   /* case 1 */
else if n is an interior node with operator OP,
left child n1 and right child n2
then
if lable (n2) = = 0 then
{
Let 0 be the operand for n2;
create node (n1)
print (OP n, top (Reg stack));
}
/* case 2 */

else if ((1 < lable (n1) < label (n2)
and (label, (n1) < r))
then
{
   interchange (Regstack);
```



Scanned by CamScanner

```
        create node (n2);
        r = POP (Regstack) ; gencode (n1);
        Print (OP r, top (Regstack));
        Push (Regstack, r);
        Swap (Regstack);
    }
    / * case 3 *
    else if (( 1< label (n2) < label (n1))
    and (lable (n2) < r)
    then
    {
    gencode (n1);
    r = pop (Regstack) ; gencode (n2);
    Print (OP, top (Regstack));
    Push (Regstack, R);
    }
/ * case 4
    else
    {
    create code (n2);
    T : = pop (Regstack);
    Print (load top (Regstack), T);
    createnode (n1);
    Print (OP T, top (Regstack));
    push (Regstack, T);
    }
}
```





Code Generation Phase Example

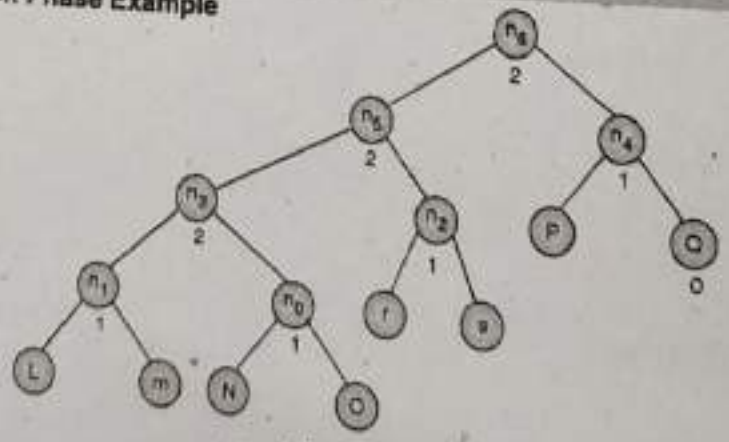


Fig. 6.13.2

No : of registers r = 2

$n_6 \rightarrow n_5 \rightarrow n_3 \rightarrow n_1 \rightarrow L \rightarrow$ load l, r_0

$\rightarrow$ OP n1 m , r_0

$\rightarrow n_2 \rightarrow r \rightarrow$ load r, r_1

Scanned by CamScanner

$\rightarrow$ OP n_2 s, r_1

$\rightarrow$ OP n_5 r_1, r_0

$n_4 \rightarrow e \rightarrow$ load P, r_1

$\rightarrow$ OP n_4 Q, r_1

$\rightarrow$ OP n_6 r_1, r_0

6.14 Basic Block and Flow Graphs

➔ (MU - May 2015)

Q. What is flow graph ? State its signification in code generation. **May 2015, 5 Marks**

6.14.1 Control-Flow Analysis

- The compiler has done an enormous amount of analysis through various phases of compiler like lexical analysis, syntactic analysis, semantic analysis and finally intermediate-code generation, but till now compiler doesn't really know how the program works and what it does.
- During control-flow analysis, the compiler gives the information about the working of the program after removal of syntactic or semantic errors in the code. Control-flow analysis begins by constructing a control-flow graph, which determines the control structure of a program and build a Control Flow Graph.
- ***A flow graph is a graphical representation of a sequence of instructions with control flow edges.*** A flow graph can be determined at the intermediate code level or target level.

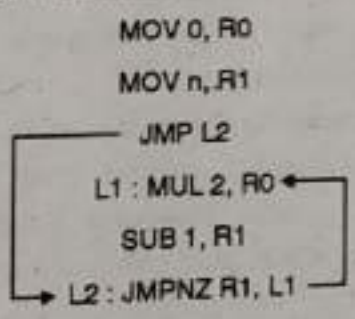


Fig. 6.14.1 : Flow Graph Representation

- To build the graph, we first divide the code into basic blocks. *A basic block is a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end.*

A basic block has exactly one entry and one exit point. It states that in basic block only the first statement can be reached from outside the block and all statements are executed consecutively one after another.

Scanned by CamScanner

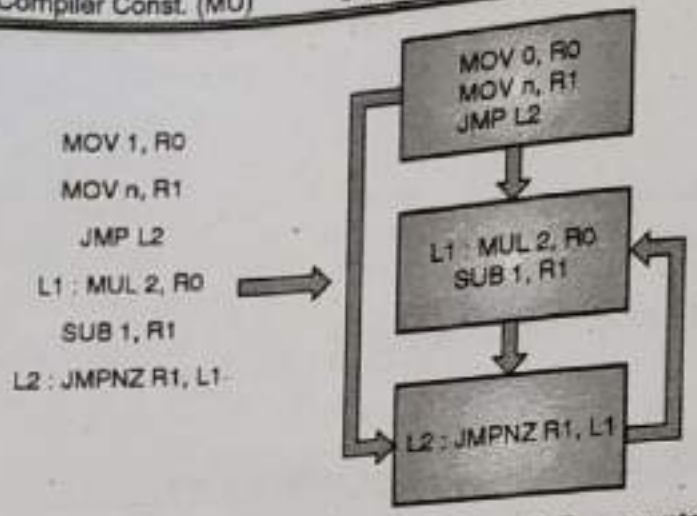


Fig. 6.14.2 : Basic Block and Control Flow Graph Representation

A basic block may starts in one of several ways mentioned below

- Through an entry point into the function or procedure
- The target of a branch or instruction immediately following a branch

A basic block ends in any of the following ways

- Jump statement.
- Conditional or unconditional branch.
- Return statement.

Control flow graph provides the following information

- Each node represents a basic block,
- Directed edge represents jumps in the control flow.
- *Entry block*, through which control enters into the flow graph,
- *Exit block*, through which all control flow, leaves.

These terms are most commonly used during the construction of the control flow graphs.

Entry Block : Through this block all control flows enter in the graph and it has no predecessors.

Exit Block : Through exit block all control flows leaves the graph and it has no successors.

Back Edge : Back edge is an edge which points to the basic block that has already been seen in bottom up parsing.

Critical Edge : **Critical edge is an** edge which is neither leaving its source block, nor the edge entering its destination block. These edges work as a split; where a new block is inserted in the middle of the edge in order to calculate the computations on the edge without affecting any other edges.

Dominator : Block P dominates block Q if every path from the entry block Q has to pass through block P. The entry block dominates all blocks.

6.14.2 Partition Algorithm for Basic Block

Input : A sequence of all the instruction in three-address statements form.

Output : A list of basic blocks and set of leaders with each three-address statement in exactly one block.

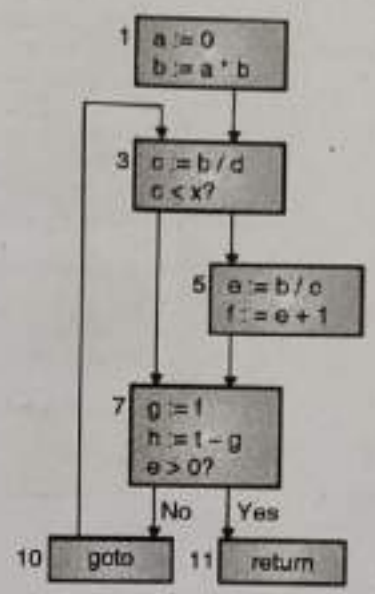
Scanned by CamScanner

Algorithm

1. Identify *leaders*, the first statements of basic blocks
   (a) The first instruction in a procedure, or
   (b) The target of any branch, or
   (c) An instruction immediately following a branch (implicit target)
   (d) Any statement that is the target of a goto is a leader
   (e) Any statement that immediately follows a goto is a leader
   (f) Gobble all subsequent instructions until the next leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Example 1 :

1. a := 0
2. b := a * b
3. L1: c := b/d
4. if c < x goto L2
5. e := b / c
6. f := e + 1
7. L2: g := f
8. h := t - g
9. if e > 0 goto L3
10. goto L1
11. L3: return



Control Flow Graph for Example 1

1. a:=0
2. b:=a*b
3. L1 : c:=b/d
4. if c< x goto L2
5. e:=b/c
6. f:=e*1
7. L2 :g:=f
8. h:=t – g
9. if e > 0 goto L3
10. goto L1.
11. L3 :return

Leaders : – {1, 3, 5, 7, 10, 11}

Basic Blocks

– {1, 2}
– {3, 4}
– {5, 6}
– {7, 8, 9}
– {10}
– {11}

Scanned by CamScanner

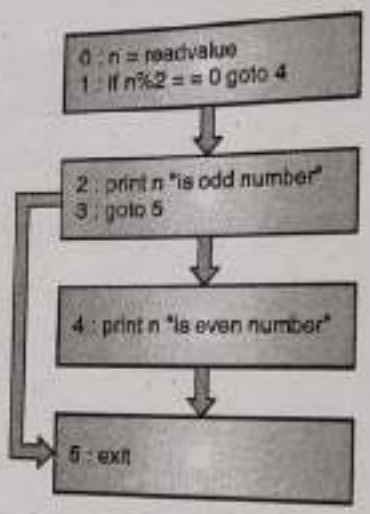
➤ **Example 2**

Consider the following fragment of code :

0: n = readvalue
1: if n % 2 = = 0 goto 4
2: print n "is odd number." A
3: goto 5
4: print n "is even number."
5: exit B

In this code, we have 4 basic blocks:

A from 0 to 1, Entry Block
B from 2 to 3.
C at 4
D at 5. Exit Block
Lines 4 and 5 are jump targets.



➤ **Example 3**

Consider the following C-code for bubble sort.

```
for i:=n-1 to 1 do
    for j:=1 to i do
        if(a[j]>a[j+1]) then
            begin
                temp:=a[j]
                a[j]:=a[j+1]
                a[j+1]:=temp
            end
    end for j loop
end for i-loop
```

Fig. 6.14.3 : C code for Bubble Sort

Scanned by CamScanner

i:=n-1	temp:=a[t9]
L5: if (i<1) goto L1	t10:=j+1
j:=1	t11:=t10-1
L4: if(j>i) goto L2	t12:=4*t11
t1:=j-1;	t13:=a[t12]
t2:=4*t1	t14:=j-1
t3:=a[t2]	t15:=4*t14
t4:=j+1	a[t15]:=t13
t5:=t4-1	t16:=j+1
t6:=4*t5	t17:=t16-1
t7:=a[t6]	t18=4*t17
if(t3<=t7) goto L3	a[t18]:=temp
t8:= j-i	L3: j:=j+1
t9:=4*t8	goto L4
	L2: i=i-1
	goto L5
	L1: stop

Fig. 6.14.4 : Three Address Code for Bubble Sort

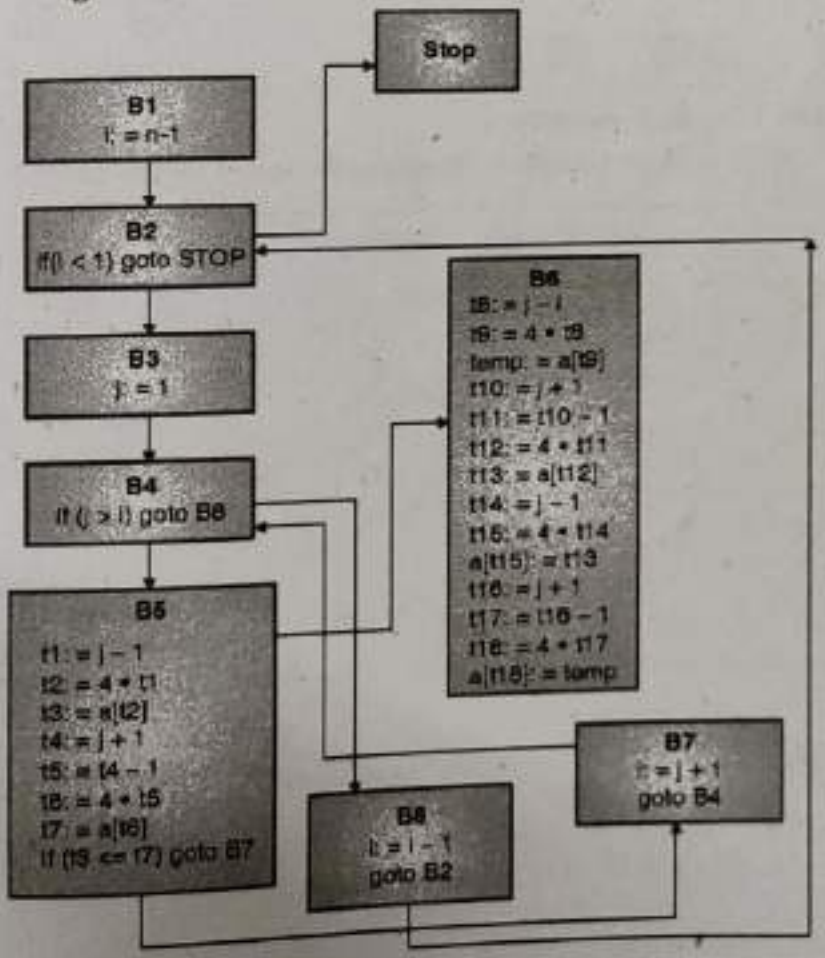


Fig. 6.14.5 : Control Flow Graph for Three Address Code of Bubble Sort

Scanned by CamScanner

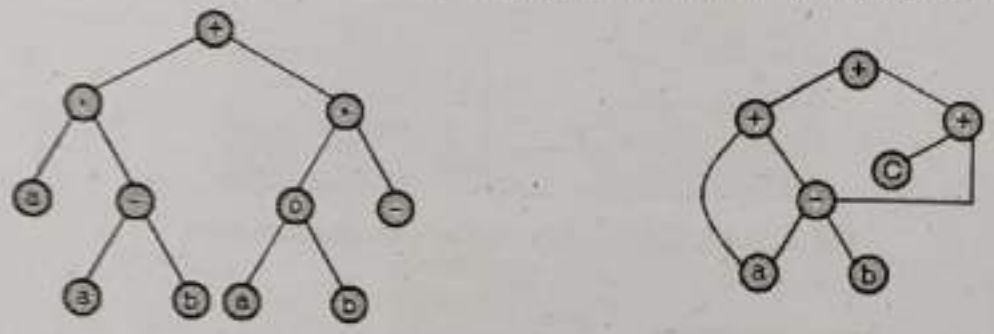
6.15 Directed Acyclic Graphs

Q. Explain DAG. **(5 Marks)**

- In compilers a tree has a one route from the parent (root) to child (leaf) node. *Directed acyclic graph* (DAG) is used to share the common subtree of a given expression. DAG is used represent more than one path from start symbol to terminals. DAG's are difficult to construct, but provide an obvious savings in space. They also highlight equivalent sections of code and that will be useful later when we use different optimization techniques.
- DAG is formed by grouping vertices and directed edges. DAG is a directed graph with no directed cycles. In DAG each edge is used to connect one vertex to another. In DAG there is no way to start at some vertex p and follow a sequence of edges that eventually loops back to p again.

➢ **Example 1**

A syntax tree and DAG representation for the string a * (a–b) + c * (a–b) is given in Fig. 6.15.1.



(a) Syntax Tree Representation **(b) DAG Representation**

Fig. 6.15.1 : Graphical Representation of a:=b*–c+b*–c

➢ **Example 2**

D = ((A+B*C) + (A*B*C))/ –C construct a DAG for it.

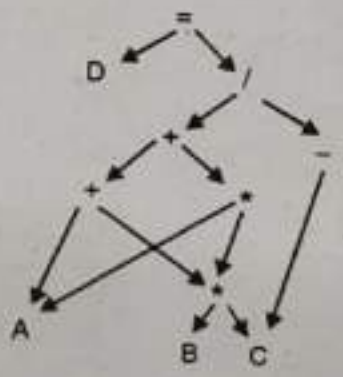


Fig. 6.15.2 : DAG for string "D = ((A+B*C) + (A*B*C))/ –C"

Fig. 6.15.3(a) shows a syntax tree representation and Fig. 6.15.3(b) shows the DAG representation for the expression **if x>0 then x = 3 * (y+1) else y = y + 1.**

Scanned by CamScanner

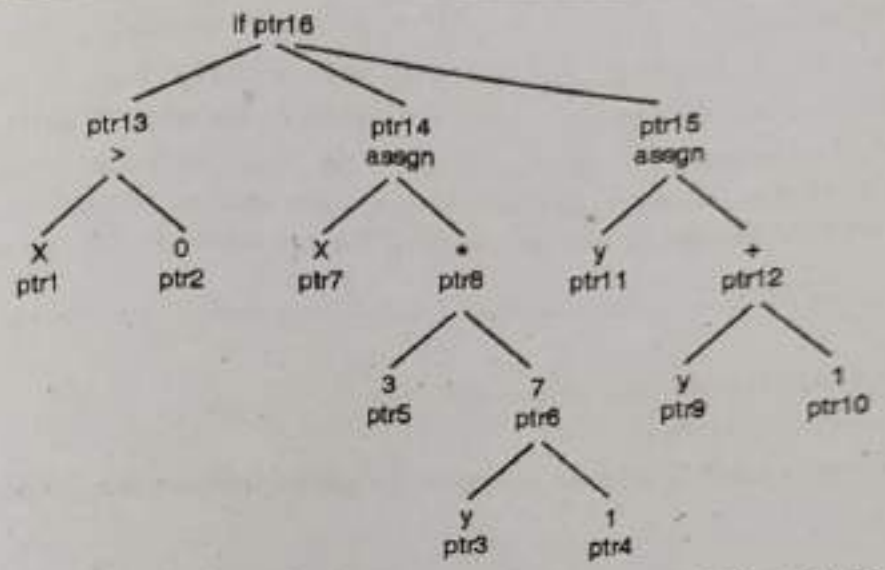


Fig. 6.15.3(a) : Syntax Tree for the expression if x > 0 then x = 3 * (y + 1) else y = y + 1

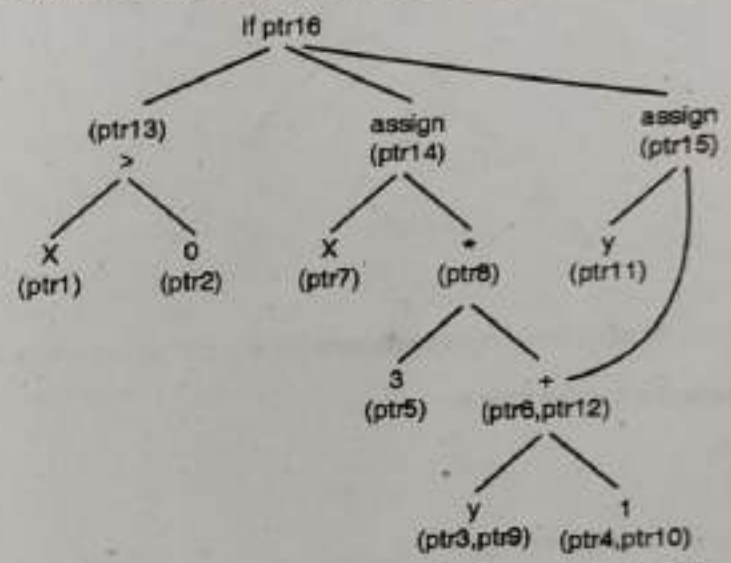


Fig. 6.15.3(b) : DAG for the expression if x > 0 then x = 3 * (y + 1) else y = y + 1

The leaf for **node y and 1 has two** parents because y + 1 is common to the two sub-expressions x and y. Likewise, both occurrences of the common sub-expression **y + 1** are represented by the same node, which also has two parents.

The sequence of instructions for constructing the DAG is listed as below. The functions defined constructs *createnode( )* and *createleaf( )* which creates new nodes only when necessary, returning pointers to existing nodes with the correct label and children whenever possible.

(1) ptr1 := createleaf(id, x);
(2) ptr2 := createleaf(num, 0);
(3) ptr3 := createleaf(id, y);
(4) ptr4 := createleaf(num, 1);
(5) ptr5 := createleaf(num, 3);
(6) ptr6 := createnode(' + ' ,ptr3,ptr4);
(7) ptr7 := createleaf(id, x);
(8) ptr8 := createnode('*',ptr5, ptr6);
(9) ptr9 := createleaf(id, y);
(10) ptr10 := createleaf(num, 1);

Scanned by CamScanner

(11) ptr11 := createleaf(id, y);
(12) ptr12 := createnode(' + ' ,ptr9,ptr10);
(13) ptr13 := createnode(' > ' ,ptr1,ptr2);
(14) ptr14 := createnode(' =' ,ptr7, ptr8)
(15) ptr15 := createnode(' =' ,ptr11, ptr12)
(16) ptr16 := createnode('if',ptr13, ptr14, ptr15)

createleaf(**id** ,y) is repeated on line 3 and line 9 the node constructed by the previous call createleaf(**id**, y) is returned, so ptr3 = ptr9.Similarly,the nodes returned on lines 4 and 10 are same. Hence, the node returned on line 12 must be the same one constructed by the call of createnode on line 6.

➢ **Example 3**

Construct the syntax tree and DAG for the expression

((((x + y) *(r–s)) + (x + y)) *y)

To create the syntax tree first we have to convert the given expression into postfix expression like **xy+rs-*xy++y***

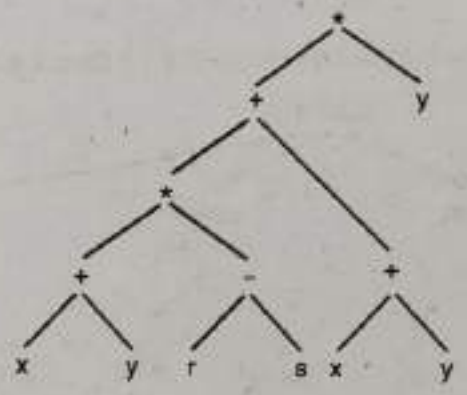


Fig. 6.15.4(a) : Syntax Tree for expression ((((x + y) * (r – s)) + (x + y)) * y)

The sequence of operation for syntax tree is :

(1) ptr1 := createleaf(id, x);
(2) ptr2 := createleaf(id, y);
(3) ptr3 := createnode(+, ptr1, ptr2)→(x+y)
(4) ptr4 := createleaf(id, r);
(5) ptr5 := createleaf(id, s);
(6) ptr6 := createnode(' – ' ,ptr4, ptr5);→(r–s)
(7) ptr7 := createnode('*',ptr3,ptr6);→(x+y) *(r–s)
(8) ptr8 := createleaf(id, x);
(9) ptr9 := createleaf(id, y);
(10) ptr10 := createnode('+', ptr8, ptr9); →(x+y)
(11) ptr11 := createnode('+', ptr7, ptr10);→((x+y)*(r–s))+(x+y)
(12) ptr12 := createleaf(id, y);
(13) ptr13 := createnode(' * ' ,ptr11,ptr12);→((((x+y)*(r-s))+(x+y))+y)

Scanned by CamScanner

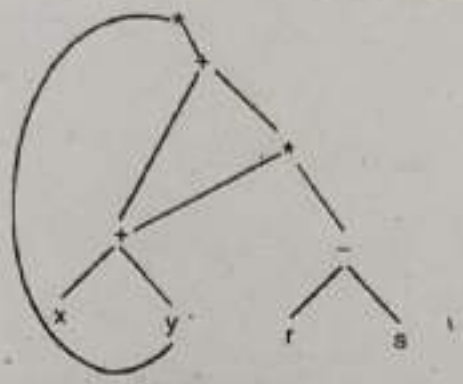


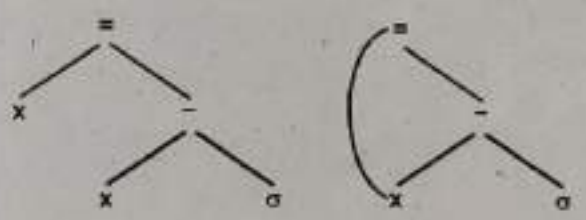
Fig. 6.15.4(b) : DAG for expression ((((x + y) *(r - s)) + (x + y)) * y)

The sequence of operation for DAG is as follows :

1. ptr1 := createleaf(id, x);
2. ptr2 := createleaf(id, y);
3. ptr3 := createnode(+, ptr1, ptr2)→(x+ y)
4. ptr4 := createleaf(id, r);
5. ptr5 := createleaf(id, s);
6. ptr6 := createnode(' – ' ,ptr4, ptr5);→(r–s)
7. ptr7 := createnode('*',ptr3,ptr6);→(x + y) *(r–s)
8. ptr8 := createnode ('+', ptr7, ptr3);→ ((x+y)*(r–s))+(x+y)
9. ptr9 := createnode('*,ptr8,ptr2);

Example 4

Construct a syntax tree and DAG for x=x–6.



(a) Syntax tree **(b) DAG**

Fig. 6.15.5 : Syntax tree and DAG representation of Example 7

Example 6.15.1 : Draw and explain DAG and represent the following example with it.
(a / b) + (a / b) * (c + d) **MU - May 2016, 10 Marks**

Solution :

DAG representation of (a / b) + (a / b) * (c + d)

Scanned by CamScanner

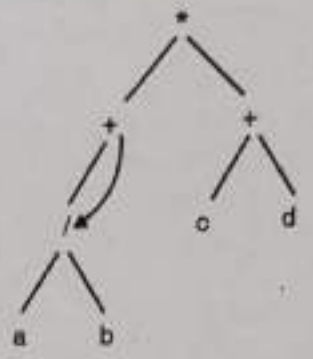


Fig. P. 6.15.1 : DAG Representation

6.16 DAG Representation of Basic Block

Directed Acyclic graphics give the full view of how the value computed by each statement in the basic block and is used in the subsequent statements of the block. A DAG for a basic block is a directed acyclic graph with the following labels on nodes.

(i) Leaf nodes are marked with variable name or constants
   (a) They are unique identifiers
   (b) With the help of an operator we identify l - or r- value
   (c) Subscript with 0 represent initial values

(ii) Operator symbols are represented with the help of on interior nodes.

(iii) Nodes are also given a sequence of identifies for labels
   - Interior node = computed values
   - Identifiers in the sequence

Example of DAG representation

$r_1 = 3KJ$ if J <10 goto 1

$r_2 = a\,[r_1]$

$r_3 = 3 * J$

$r_4 = b\,[r_3]$

$r_5 = r_2 * r_4$ three address code

$r_6 = sum + r_5$

$a = r_6$

$J = r_7$

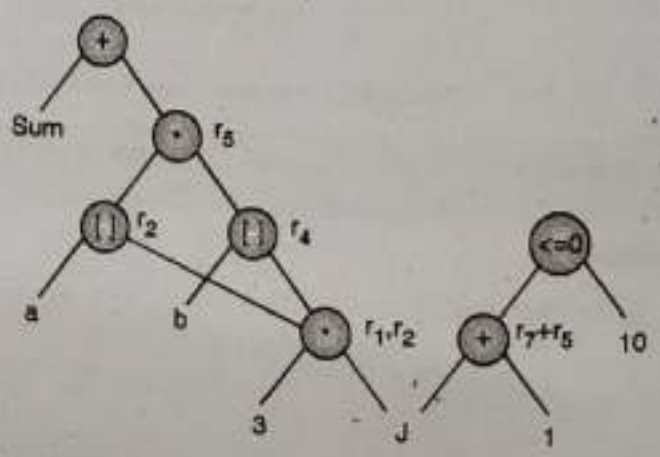


Fig. 6.16.1 : Corresponding DAG of given three address code

Scanned by CamScanner

Constructing a DAG from three address code is a good way to determine :

1. Common sub expressions evaluation.
2. Which names are used inside the block but evaluated outside.
3. Dead code elimination within the block.
4. Which statements of the block could have their computed value used outside the block.
5. Identifying the similar algebraic identities.

Review Questions

Q.1 What is an intermediate code ? Discuss advantages of writing intermediate code.

Q.2 Translate the following program fragment into three address code.

```
int i;
i = 1 ;
while a < 10 do
        if x > y then a = x + y
        else a = x - y
```

Q.3 Discuss the issues in designing an intermediate representation.

Q.4 What are the advantages of using an intermediate language ?

Q.5 Write a short note on commonly used intermediate representation forms.

Q.6 What are the various graphical representation used in intermediate languages ?

Q.7 What are three-address-code ? Explain the types of Three-address-statements.

Q.8 Explain Syntax Directed Translation for three address code.

Q.9 Translate the following program fragment into three address code

```
int i ;
i = 1 ;
while a < 10 do
        if x > y then a = x + y
        else a = x - y
```

Q.10 Give the generated quadruples for the following program segment

```
while  (a << && b < d) do
if a = 1 then c = c++
else
        while a ≤ d do
            a = a + 3
```

Scanned by CamScanner

Q. 11 Generate the three address code for,

(a)
```
while (i<10)
{
x = 0 ;
i = i + 1 ;
}
```

(b)
```
switch (C)
{
Case 1 : C = x * y
    break ;
Case 2 : C = x /y
    break ;
}
```

(c)
```
for (i = 1 ; i < = 5 ;i ++)
{
array (i) = temp * 5 ;
}
```

(d)
```
int i
int array [10] [10]
i = 0
while [i < 10]
{
  array (i) (i) = 1;
  i + + ;
}
```

Q. 12 Describe the necessary and sufficient conditions for performing constant propagation and dead code elimination.

Q. 13 What is peephole optimization? Explain in brief.

Q. 14 Explain machine dependent and machine independent code optimizations. What are their advantages?

Q. 15 Consider the flow graph shown in Fig. 1

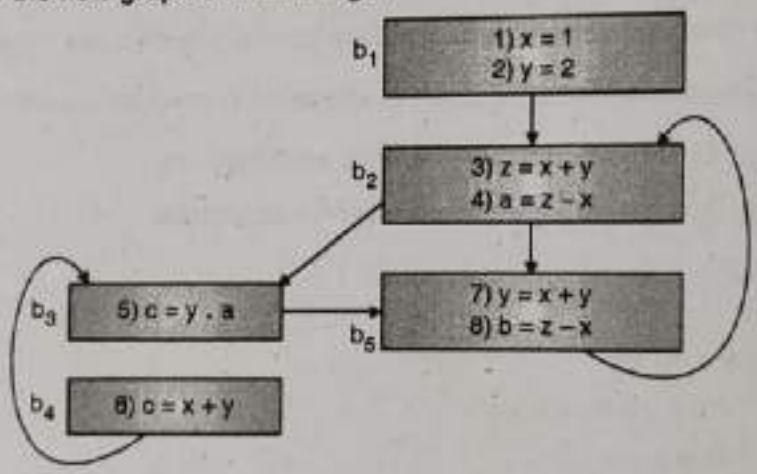


Fig. 1

(a) Compute live variables at the end of each block.

(b) Compute available expressions for each block

(c) Eliminate any common sub-expression for each block.

Scanned by CamScanner

Q.16 Consider the following program which finds the sum and the largest number.

```
main ( )
{   int i, x[10], sum =0, large;
        for(i=0;i<10;i++)
        {
            printf("%d",x[i]);
            sum + =x[i];
        }
    large =x[10];
        for(i=0;i<10;i++)
        {
            if( x[i] > large)
            large = x[i];
        }
    printf("%d, %d", sum, large);
}
```

(a) Translate the program into three-address statements and construct a flow graph from the three address statements.

(b) Eliminate common sub-expression from each basic block.

(c) Find the induction variables of each loop and eliminate them, if possible.

Q.17 Discuss the importance of optimization in compiler design.

Q.18 What are the problems in optimizing compiler ?

Q.19 What are the factors influencing optimization ?

Q.20 Explain the following optimizations with the help of an example:

(a) Common sub-expression elimination

(b) Elimination of dead code

(c) Constant propagation

(d) Frequency reduction

Q.21 What are the various techniques used for sources of optimization ?

Q.22 Explain principle sources of optimization techniques available for code optimizations in compiler design.

Q.23 Explain Local Optimizations and Global Optimization in detail.

Q.24 Explain the following terms :

1. Dominator
2. Header
3. Natural Loops
4. Back Edge

Q.25 Write an algorithm to find the natural loops in the flow graph with the help of an example.

Q.26 Explain optimization of basic blocks with the help of an example.

Chapter Ends

❑❑❑

Scanned by CamScanner

Lab Manual

Experiment 1 : Design of an Assembler (PASS-1).

Solution :

Aim

To implement pass 1 of an assembler which generate the symbols. The purpose of the first pass is to assign a location to each instruction and data defining pseudo instruction, and thus to define value for symbols appearing in the label fields of source program.

Theory

Assembler is programs that accept as input an assembly language program and produces its machine language equivalent along with information for loader. The following data structures are used in pass (1)

1. Input source cards.
2. Location counters (LC) to keep track of each instruction location.
3. A table the MOT that indicates the symbolic mnemonic and action to be taken for each instruction.
4. A Pseudo Opcode Table (POT) that indicates the symbolic mnemonic and action to be taken for each pseudo opcodes.
5. A symbol table that is used to store each label and its corresponding value.
6. A literal table that is used to store each literal encountered and its corresponding assign location.
7. A copy is made that is later used in pass2.

Program Logic

Algorithm

1. Start.
2. Read next entry in the source card.
3. Is there a label on a card if yes, store it instruction with LC value.
4. Search the POT for pseudo opcode.
5. If it is found, POT check the type of pseudo opcode for using (USING) and DROP goto step 2 for END stop. For OS and DC determine length of a data space required.
6. If not found search MOT.
7. If found get the length of the instruction.
8. Process the literals and update LC.
9. Goto step 2 until END.

Scanned by CamScanner

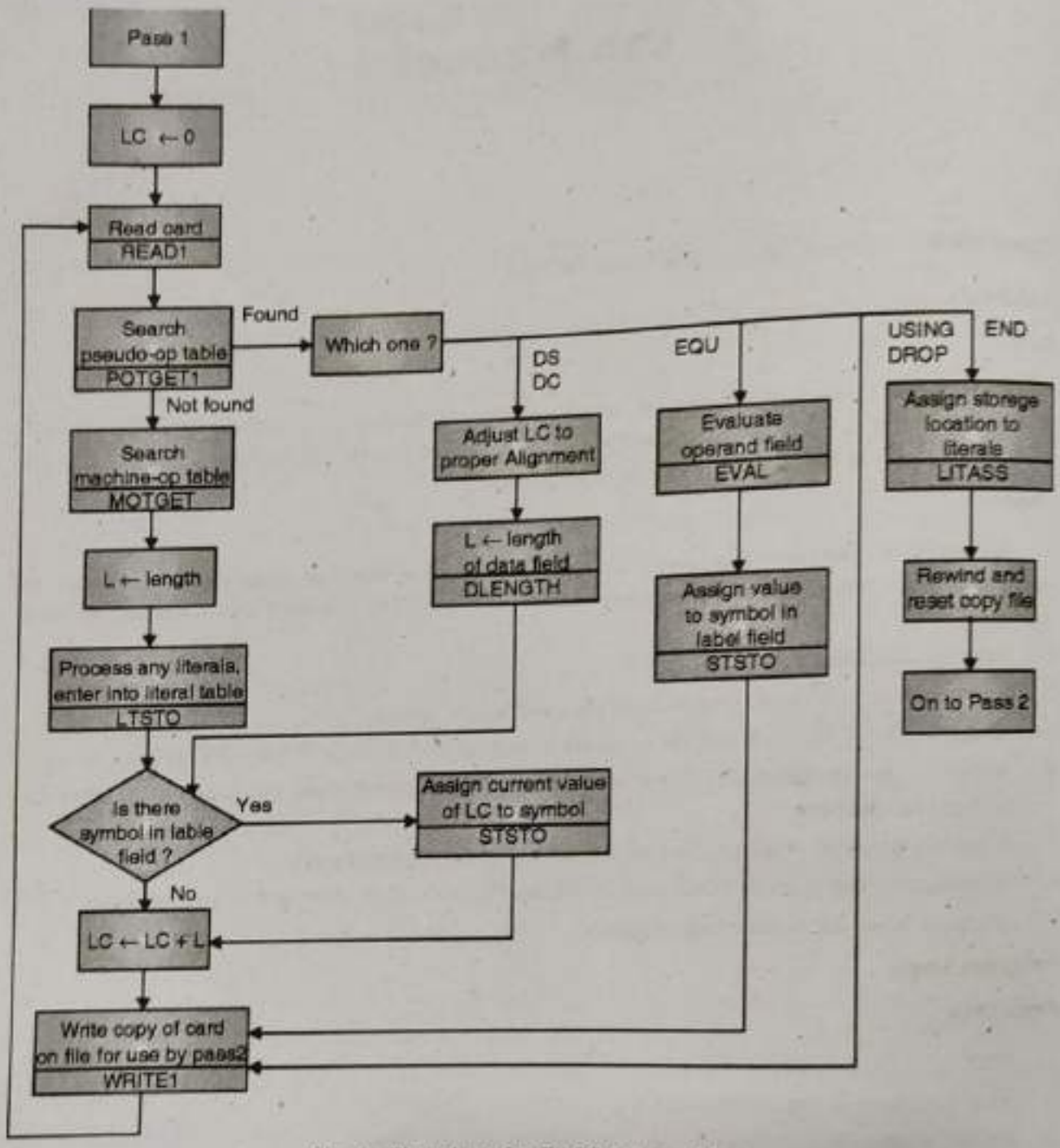


Fig. 1 : Flowchart for PASS-1 Assembler

Conclusion

Thus have implemented pass 1 of assembler and generated following tables like

1. Pseudo opcode table
2. Machine operand table
3. Symbol table.

Scanned by CamScanner

Source Code

```
import java.io.*;
import java.util.*;
class Pass1
{
    public static void main (String args[])throws Exception
    {
        String m[] ={"L","A","ST"};
        String b[]={"1001","1011","1111"};
        String p[]={"DC","EQU","DS","USING","END"};
        int l[]={4,1,4};
        int LC = 0;
        Vector<String>mneumonic = new Vector();
        Vector<String>bincode = new Vector();
        Vector<Integer>mlength = new Vector();
        Vector<String>pseudo = new Vector();
        Vector<Integer>ppos = new Vector();
        Vector<String>symbolname = new Vector();
        Vector<Integer>symbolvalue = new Vector();
        Vector<String>format = new Vector();
        Vector<String>RA = new Vector();
        Vector<Integer>symbollength = new Vector();
        Scanner sc = new Scanner( new File( "prog.txt" ) );
        String xx = sc.nextLine();
        StringTokenizer stt = new StringTokenizer( xx );
        Vector<String>readd = new Vector();
        while( stt.hasMoreTokens() )
        readd.add( stt.nextToken() );
        symbolname.add( readd.elementAt(0) );
        LC = Integer.parseInt( readd.elementAt( readd.size() - 1 ) );
        symbolvalue.add( LC );
        symbollength.add( 1 );
        RA.add( "R" ) ;
        Vector<String>alreadyread = new Vector();
        String prev = "";
        while( sc.hasNext() )
        {
            String x = sc.nextLine();
```

Scanned by CamScanner

```
        StringTokenizer st = new StringTokenizer( x );
        Vector<String>read = new Vector();
        while( st.hasMoreTokens() )
        read.add( st.nextToken() );
        for(int i=0;i<read.size();i++)
        {
            int potpos = getIndex( p , read.elementAt(i) );
            if( potpos == -1 )continue;
            pseudo.add( p[potpos] );
            ppos.add( potpos + 1 );
            if( !p[potpos].equals( "USING" ) && !p[potpos].equals( "END" ) )
            {
                symbolname.add( read.elementAt( i-1 ) );
                symbollength.add( l[potpos] );
            }
        }
    }
    sc = new Scanner( new File( "prog.txt" ) );
    String ss = sc.nextLine();
    while( sc.hasNext() )
    {
        String x = sc.nextLine();
        StringTokenizer st = new StringTokenizer( x );
        Vector<String>read = new Vector();
        while( st.hasMoreTokens() )
        read.add( st.nextToken() );
        boolean flag = false;
        for(int i=0;i<read.size();i++)
        {
            int motpos = getIndex( m , read.elementAt(i) );
            if( motpos !=-1 )
            {
                mneumonic.add( m[motpos] );
                bincode.add( b[motpos] );
                flag = true;
            }
        }
        if( flag )
```

Scanned by CamScanner

```
    {
        boolean pushed = false;
        String last = read.elementAt( read.size() - 1 );
        alreadyread.add( last );
        for(int i=0;i<symbolname.size();i++)
        if( symbolname.elementAt(i).equals( last ) )
        {
            format.add( "RX" );
            pushed = true;
            break;
        }
        if( !pushed )format.add( "AX" );
    }
}
sc = new Scanner( new File( "prog.txt" ) );
ss = sc.nextLine();
while( sc.hasNext() )
{
    String x = sc.nextLine();
    StringTokenizer st = new StringTokenizer( x );
    Vector<String>read = new Vector();
    while( st.hasMoreTokens() )
    read.add( st.nextToken() );
    for(int i=0;i<read.size();i++)
    {
        int motpos = getIndex( m , read.elementAt(i) );
        if( motpos !=-1 )
        {
            LC += 4;
        }
        int potpos = getIndex( p , read.elementAt(i) );
        if( potpos !=-1 && !p[potpos].equals( "USING" ) && !p[potpos].equals("END") )
        {
            symbolvalue.add( LC );
            if( !p[potpos].equals( "EQU" ) )
            LC += 4;
            continue;
        }
```

Scanned by CamScanner

```
            }
            }
        for(int i=0;i<symbolname.size();i++)
        {
            String ii = symbolname.elementAt(i);
            boolean there = false;
            for(int j=0;j<alreadyread.size();j++)
            if( ii.equals( alreadyread.elementAt(j) ) )
            {
                there = true;
                break;
            }
            RA.add( there ? "R" : "A" );
        }
        System.out.println( "MOT" );
        System.out.println( "Mneumonic Binopcode  Length   Format" );
        for(int i=0;i<mneumonic.size();i++)
        {
System.out.printf( "%s\t    %s\t%d\t%s\n" , mneumonic.elementAt(i) , bincode.elementAt(i) , 4 ,
format.elementAt(i) );
        }
        System.out.println( "POT" );
        System.out.println( "Pseudo-opcode   Address" );
        for(int i=0;i<pseudo.size();i++)
        {
            System.out.printf( "%s  \t\t %d\n" , pseudo.elementAt(i) , ppos.elementAt(i) );

        }

        System.out.println( "Symbol Table" );
        System.out.println( "Name   Counter  Length   R/A " );
        for(int i=0;i<symbolname.size();i++)
        {
System.out.printf( "%s\t %d\t %d\t %s\n" , symbolname.elementAt(i) , symbolvalue.elementAt(i) ,
symbollength.elementAt(i)  , RA.elementAt(i) );
        }
    }
    static int getIndex( String st[] , String s )
    {
```

Scanned by CamScanner

```
        for(int i=0;i<st.length;i++)
            if( s.equals( st[i] ))
            return i;
        return -1;
    }
}
```

The program to input in the Pass 1 assembler is written in notepad and will be read by the above program.

```
JOHN START 0
    USING * 15
    L 1 FIVE
    A 1 FOUR
    ST 1 TEMP
FOUR   DC F 4
FIFTY EQU 3
FIVE DC F '5'
TEMP DS IF
```

Output

```
C:\Documents and Settings\teb2\Desktop\alif>javac Pass1.java
Note: Pass1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
C:\Documents and Settings\teb2\Desktop\alif>java Pass1
MOT
Mneumonic Binopcode Length  Format
L       1001     4     RX
A       1011     4     RX
ST      1111     4     RX
POT
Pseudo-opcode  Address
USING          4
DC             1
EQU            2
DC             1
DS             3
Symbol Table
Name  Counter  Length  R/A
JOHN   0       1       R
FOUR   12      4       A
```

Scanned by CamScanner

FIFTY	16	1	R
FIVE	16	4	A
TEMP	20	4	R

Experiment 2 : Implementation of a design of an Assembler (PASS-2).

Solution :

Aim

To implement pass 2 for an assembler which generates the base table and base index values for symbols.

Theory

A common assembler focuses on forward reference problem in which a variable is referred before it is defined. Thus, we use an assembler from pass 1 which defines the symbols and pass 2 generates the machine code.

The data structure modes are :

1. Copy of source program
2. LC to keep track of location
3. A table MOT with symbolic mnemonic length, binary machine opcode
4. A table POT for each pseudo opcode
5. A table ST prepared by pass 1
6. Base table (BT) registers contents and their availability
7. A workspace INST to hold each instruction assembled together
8. Workspace print line used to procedure printed timing
9. Workspace PUNCH CARD used prior to actual output
10. An output check of assembled instruction in the format studied by the assembler.

Program Logic

Algorithm

1. Start
2. Read the next entry from copy of pass 1
3. Search POT
4. If forward check occurs from the type of pseudo opcode, then perform the corresponding actions and go to step 2
5. If not found, search MOT
6. Get instruction length, binary code and type
7. Evaluate operand expressions by searching for values of symbols
8. Assemble together the parts of instructions
9. Update LC
10. Repeat 2-9 until END
11. Stop

Scanned by CamScanner

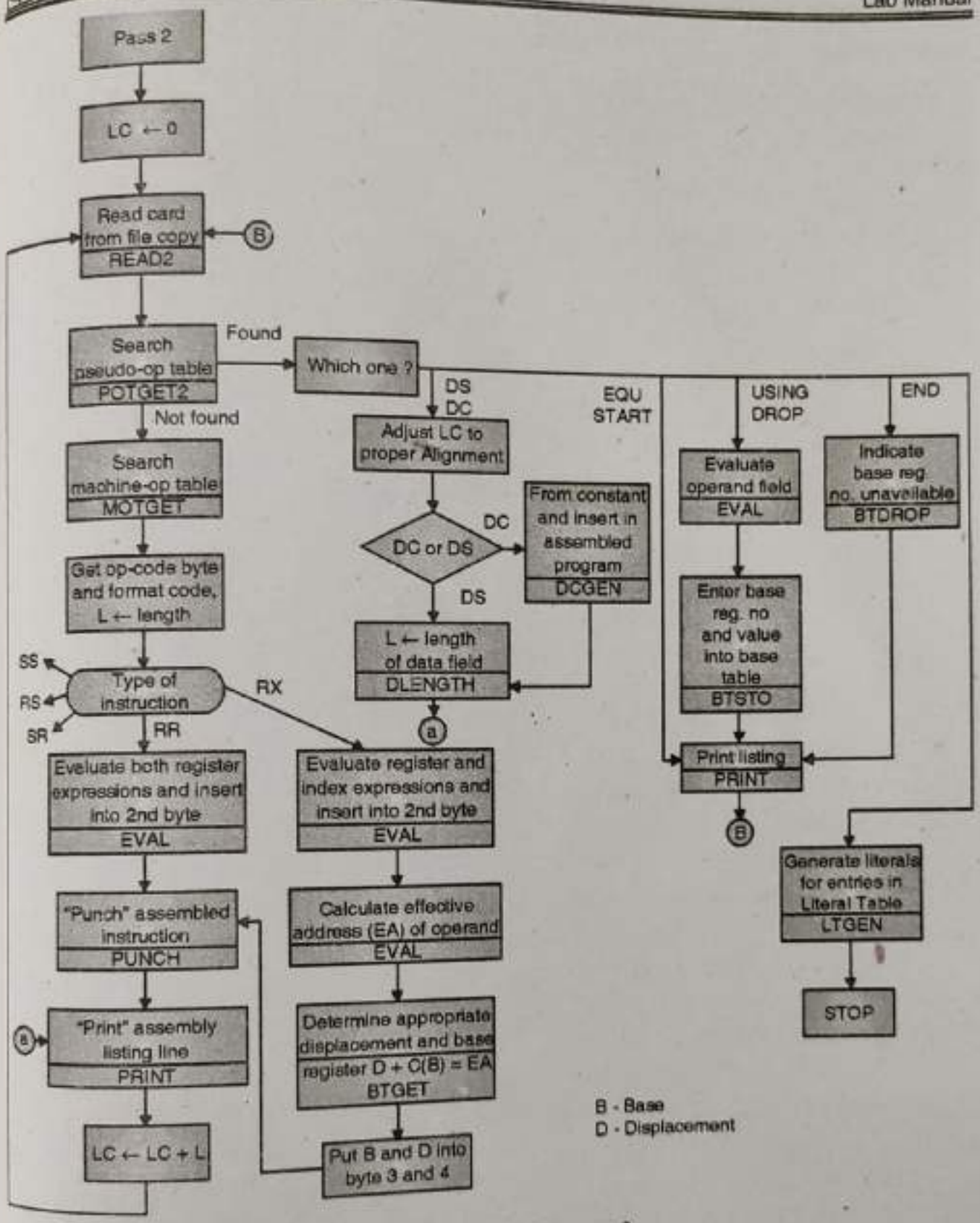


Fig. 2 : Flowchart for pass 2

Conclusion

Thus, we have studied and implemented the pass 2 of assembler and generated base table.

Scanned by CamScanner

Source Code

```
import java.io.*;
import java.util.*;
class Pass2
{
    public static void main (String args[])throws Exception
    {
        String m[] = {"L","A","ST"};
        String b[]={"1001","1011","1111"};
        String p[]={"DC","EQU","DS","USING","END"};
        int l[]={4,1,4};
        int LC = 0;
        Vector<String>mneumonic = new Vector();
        Vector<String>bincode = new Vector();
        Vector<Integer>mlength = new Vector();
        Vector<String>pseudo = new Vector();
        Vector<Integer>ppos = new Vector();
        Vector<String>symbolname = new Vector();
        Vector<Integer>symbolvalue = new Vector();
        Vector<String>format = new Vector();
        Vector<String>RA = new Vector();
        Vector<Integer>symbollength = new Vector();
        Scanner sc = new Scanner( new File( "prog.txt" ) );
        String xx = sc.nextLine();
        StringTokenizer stt = new StringTokenizer( xx );
        Vector<String>readd = new Vector();
        while( stt.hasMoreTokens() )
        readd.add( stt.nextToken() );
        symbolname.add( readd.elementAt(0) );
        LC = Integer.parseInt( readd.elementAt( readd.size() - 1 ) );
        symbolvalue.add( LC );
        symbollength.add( 1 );
        RA.add( "R" ) ;
        int origLC = LC;
        Vector<String>alreadyread = new Vector();
        String prev = "";
        while( sc.hasNext() )
        {
```

Scanned by CamScanner

```
        String x = sc.nextLine();
        StringTokenizer st = new StringTokenizer( x );
        Vector<String>read = new Vector();
        while( st.hasMoreTokens() )
        read.add( st.nextToken() );
        for(int i=0;i<read.size();i++)
        {
            int potpos = getIndex( p , read.elementAt(i) );
            if( potpos == -1 )continue;
            pseudo.add( p[potpos] );
            ppos.add( potpos + 1 );
            if( !p[potpos].equals( "USING" ) && !p[potpos].equals( "END" ) )
            {
                symbolname.add( read.elementAt( i-1 ) );
                symbollength.add( l[potpos] );
            }
        }
    }
    sc = new Scanner( new File( "prog.txt" ) );
    String ss = sc.nextLine();
    while( sc.hasNext() )
    {
        String x = sc.nextLine();
        StringTokenizer st = new StringTokenizer( x );
        Vector<String>read = new Vector();
        while( st.hasMoreTokens() )
        read.add( st.nextToken() );
        boolean flag = false;
        for(int i=0;i<read.size();i++)
        {
            int motpos = getIndex( m , read.elementAt(i) );
            if( motpos !=-1 )
            {
                mneumonic.add( m[motpos] );
                bincode.add( b[motpos] );
                flag = true;
            }
        }
```

Scanned by CamScanner

```
            if( flag )
            {
                boolean pushed = false;
                String last = read.elementAt( read.size() - 1 );
                alreadyread.add( last );
                for(int i=0;i<symbolname.size();i++)
                if( symbolname.elementAt(i).equals( last ) )
                {
                    format.add( "RX" );
                    pushed = true;
                    break;
                }
                if( !pushed )format.add( "AX" );
            }
        }
        sc = new Scanner( new File( "prog.txt" ) );
        ss = sc.nextLine();
        while( sc.hasNext() )
        {
            String x = sc.nextLine();
            StringTokenizer st = new StringTokenizer( x );
            Vector<String>read = new Vector();
            while( st.hasMoreTokens() )
            read.add( st.nextToken() );
            for(int i=0;i<read.size();i++)
            {
                int motpos = getIndex( m , read.elementAt(i) );
                if( motpos !=-1 )
                {
                    LC += 4;
                }
                int potpos = getIndex( p , read.elementAt(i) );
if( potpos !=-1 && !p[potpos].equals( "USING" ) && !p[potpos].equals("END") )
                {
                    symbolvalue.add( LC );
                    if( !p[potpos].equals( "EQU" ) )
                    LC += 4;
                    continue;
```

```
        }
    }
}
Vector<String> mcode = new Vector();
Vector<Integer> mcode1 = new Vector();
Vector<Integer> mcode2 = new Vector();
sc = new Scanner( new File( "prog.txt" ) );
ss = sc.nextLine();
boolean usi = false;
int FF = 0;
while( sc.hasNext() )
{
    String x = sc.nextLine();
    StringTokenizer st = new StringTokenizer( x );
    Vector<String> read = new Vector();
    while( st.hasMoreTokens() )
    read.add( st.nextToken() );
    for(int i=0;i<read.size();i++)
    {
        int potpos = getIndex( p , read.elementAt(i) );
        if( potpos !=-1 && p[potpos].equals( "USING" ) )
        {
        FF = Integer.parseInt( read.elementAt( read.size() - 1 ) );
            usi = true;
            break;
        }
    }
    if( usi )
    {
        for(int i=0;i<read.size();i++)
        {
            int motpos = getIndex( m , read.elementAt(i) );
            if( motpos !=-1 )
            {
                mcode.add( m[motpos] );
                mcode1.add( Integer.parseInt( read.elementAt(i+1) ) );
                String pp = read.elementAt(i+2) ;
                mcode2.add( search( symbolname , symbolvalue , pp ) );
```

Scanned by CamScanner

```
                    break;
                }

            }
        }
    }

    for(int i=0;i<symbolname.size();i++)
    {
        String ii = symbolname.elementAt(i);
        boolean there = false;
        for(int j=0;j<alreadyread.size();j++)
        if( ii.equals( alreadyread.elementAt(j) ) )
        {
            there = true;
            break;
        }

        RA.add( there ? "R" : "A" );
    }
    System.out.println( "MOT" );
    System.out.println( "Mneumonic Binopcode Length  Format" );
    for(int i=0;i<mneumonic.size();i++)
    {
System.out.printf( "%s\t   %s\t%d\t%s\n" , mneumonic.elementAt(i) , bincode.elementAt(i) , 4 ,
format.elementAt(i) );

    }
    System.out.println( "POT" );
    System.out.println( "Pseudo-opcode  Address" );
    for(int i=0;i<pseudo.size();i++)
    {
        System.out.printf( "%s  \t\t %d\n" , pseudo.elementAt(i) , ppos.elementAt(i) );
    }
    System.out.println( "Symbol Table" );
    System.out.println( "Name  Counter Length  R/A " );
    for(int i=0;i<symbolname.size();i++)
    {
```

Scanned by CamScanner

```
System.out.printf( "%s\t %d\t %d\t %s\n" , symbolname.elementAt(i) , symbolvalue.elementAt(i) ,
symbollength.elementAt(i)  , RA.elementAt(i) );
        }
        System.out.println( "Machine code" );
        for(int i=0;i<mcode.size();i++)
        {
System.out.printf( "%s\t%d,%d(%d,%d)\n" , mcode.elementAt(i) , mcode1.elementAt(i) ,
mcode2.elementAt(i)  , origLC , FF  );
        }
        }
    static int getIndex( String st[] , String s )
    {
        for(int i=0;i<st.length;i++)
            if( s.equals( st[i] ))
            return i;
        return -1;
    }
    static int search( Vector<String>v , Vector<Integer>n , String s )
    {
        for(int i=0;i<v.size();i++)
        if( s.equals( v.elementAt(i) ) )
          return n.elementAt(i);
        return 0;
    }
}
```

The program to input in the Pass 2 assembler is written in notepad and will be read by the above program.

```
JOHN START 0
   USING * 15
   L 1 FIVE
   A 1 FOUR
   ST 1 TEMP
FOUR   DC F 4
FIFTY EQU 3
FIVE DC F '5'
TEMP DS IF
```

Scanned by CamScanner

Output

```
C:\Documents and Settings\teb2\Desktop\alif>javac Pass2.java
Note: Pass2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
C:\Documents and Settings\teb2\Desktop\alif>java Pass2
MOT
Mneumonic Binopcode Length Format
L         1001      4      RX
A         1011      4      RX
ST        1111      4      RX
POT
Pseudo-opcode Address
USING         4
DC            1
EQU           2
DC            1
DS            3
Symbol Table
Name  Counter Length R/A
JOHN  0       1      R
FOUR  12      4      A
FIFTY 16      1      R
FIVE  16      4      A
TEMP  20      4      R
Machine code
L   1,16(0,15)
A   1,12(0,15)
ST  1,20(0,15)
```

Experiment 3 : Implementation of Macro-Processors.

Solution :

Aim

To implement a macro processor to handle recursive and nested macro calls.

Theory

In assembly language programs, there is often a need to repeat blocks of statements many times in the course of programs. Many instructions are single line abbreviations for a group of instructions. Many computer manufacturers use macro instructions to automate the writing of 'tailored' operating

Scanned by CamScanner

systems in a process called system generation. Macro instructions are considered an extension of basic assembly language. In employing a macro the programmer defines a single instruction.

Typical Macro

The first line in macro pseudo opcode of the definition identifies the following lines as macro instruction name.

Program Logic

Algorithm

1. Initialize MDTC & MNTC to 1.
2. Read the next source card.
3. If it is macro pseudo opcode then enter macro name and current value of MDTC in MNT with entry number MNTC.
4. Increment MNTC.
5. Prepare ACA table.
6. Increment MNTC until MEND pseudo opcode.
7. Read assembly source code.
8. Search MNT for match with operation code.
9. If macro name found, set MDTP to MDT index from MNT entry.
10. Set up argument list array.
11. Increment MDTP and get line from MDT.
12. Substitute arguments from macro calls.
13. Repeat 7 to 12 till MEND pseudo opcode.
14. Write expanded source code.
15. Stop.

Conclusion :

Thus, we have studied and implemented the macro processor and generated various databases like macro definition table, macro name table and argument list array.

Source Code

```
import java.util.*;
import java.io.*;
class Macro1
{

  public static void main( String args[] )throws Exception
  {

    Scanner sc = new Scanner( new File( "macroin.txt" ) );

    String tmp = sc.nextLine();
```

Scanned by CamScanner

```
Vector< Vector<String>> mdtdefn = new Vector();
Vector<Integer> idx = new Vector();
Vector<Integer> mdtidx = new Vector();
int macroidx = 0;
int lptr = 0;
Vector<String> mdtname = new Vector();
int alaidx = 0;
Vector<String>ala = new Vector();

while( sc.hasNext() )
{
    String s = sc.nextLine();
    if( s.equals( "END" ) )break;

    if( s.equals( "MACRO" ) )
    {
        idx.add( lptr );
        mdtidx.add( macroidx++ );
        lptr++;
        Vector<String> v = new Vector();
        v.add( s );

        s = sc.nextLine();

        lptr++;
        StringTokenizer st = new StringTokenizer( s );

        s = st.nextToken();
        mdtname.add( s );
        v.add( s );

        s = st.nextToken();
        String ss[] = s.split( "," );
        for(int i=0;i<ss.length;i++)ala.add( ss[i] );

        while( sc.hasNext() )
        {
            s = sc.nextLine();
```

Scanned by CamScanner

```
                lptr++;
                v.add( s );
                if( s.equals( "MEND" ) )break;
            }

            mdtdefn.add( v );
        }
    }

    System.out.println( "\nMDT" );
    System.out.println( "Index\tMacroDefn" );

    for(int i=0;i<mdtdefn.size();i++)
    {
        Vector<String>v = mdtdefn.elementAt(i);
        System.out.printf( "%d\t%s\n" , i , v.elementAt(0) );
        for(int j=1;j<v.size();j++)
        System.out.printf( "  \t%s\n" , v.elementAt(j) );
    }

    System.out.println( "MNT" );
    System.out.println( "Index\tName\tMDTIndex" );


    for(int i=0;i<idx.size();i++)
    {
        System.out.printf( "%d\t%s\t%d\n" , idx.elementAt(i) , mdtname.elementAt(i) ,
mdtidx.elementAt(i) );
    }
    System.out.println( "\nALA" );
    System.out.println( "Index\tIndexParameter\tArgument" );

    for(int i=0;i<ala.size();i++)
    {
        System.out.printf( "%d\t#%d\t\t%s\n" , i , i + 1 ,ala.elementAt(i) );
    }

  }
}
```

Scanned by CamScanner

The program to input in the macros is written in notepad(macroin.txt) and will be read by the above program.

```
LOOP1      LAB    &A1, &A2, &A3
& STRG              D      3, #3
# 0                 D      2, #2
                    D      1, #1
                    MEND
```

Output

```
C:\Documents and Settings\teb2\Desktop\alif>javac Macro1.java
Note: Macro1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Documents and Settings\teb2\Desktop\alif>java Macro1
MDT
Index  MacroDefn
0      MACRO
       INCR
       A 1,&ARG1
       A 2,&ARG2
       A 3,&ARG3
       MEND
MNT
Index  Name   MDTIndex
0      INCR     0
ALA
Index  IndexParameter  Argument
0          #1          &ARG1
1          #2          &ARG2
2          #3          &ARG3
C:\Documents and Settings\teb2\Desktop\alif>
```

Experiment 4 : Implementation of Lexical Analysis.

Solution :

Aim

To implement a lexical analyser. It should break the source code into variables, constants, special symbols, keywords, operators, literals.etc.

Scanned by CamScanner

Theory

The source program is given as input to Lexical Analyser. The output of the Lexical Anaylser is given as input to Syntax Analyser. Lexical Analyser generates tokens and gives them to the Syntax Analyser. This phase takes source program and if the elements of the program are correct, then it generates meaningful units called tokens.

Lexical Analysis breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language, for instance, a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automation constructed from a Regular Expression, can be used to recognize it. This phase is also called Lexing or Scanning and the software doing Lexical Analysis is called a Lexical Analyser or Scanner.

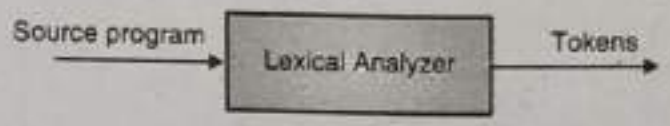


Fig. 3 : Block Diagram for Lexical Analyzer

Program Logic

Algorithm

1. Open the files for storing constants, variables, keywords, operators, special characters etc.
2. Accept the source file for which the tokens have to be generated.
3. Open the source file in read mode.
4. Read a character from the source file.
5. If character is an alphabet or a digit, then
   (i) Append it to the temp array.
   (ii) Now check whether the temp array consists of a keyword. If yes, then print the temp array in the keyword file and flush the temp array for reading a new constant, variable or keyword.

   Else if the character is space then

   If the temp array is not empty, then put the temp array into var and constant file depending on the first character of the temp array and flush the array.

   Else if the character is operator then

   (i) Point the temp array into the var or constant file since occurence of operator indicates end of a literal, constant or variable or keyword.
   (ii) Point the operator in operator file.

   Else (special symbol)

   (i) Point the temp array into var or constant file.
   (ii) Point the special symbol into the corresponding file.
6. Repeat steps 4 and 5 till end of file is encountered.
7. Close all opened files.
8. Stop

Scanned by CamScanner

Conclusion

Thus, we have studied and implemented a Lexical Analyser and we have created different text files to store the variables, constants, keywords, operators, etc.

Source Code

```
import java.util.*;
import java.io.*;
class Demo{
    public static void main(String arg[])throws Exception{
        Scanner sc=new Scanner(new File("abc.txt"));
        String i[]={"main"};
        String l[]={"a","b","c"};
        String t[]={",",";","{","}","(",")"};
        String o[]={"+","="};
        String k[]={"int","void"};
        int a=0;
        FileWriter fw1=new FileWriter(new File("identifier.txt"),true);
        FileWriter fw2=new FileWriter(new File("literal.txt"),true);
        FileWriter fw3=new FileWriter(new File("terminal.txt"),true);
        FileWriter fw4=new FileWriter(new File("operator.txt"),true);
        FileWriter fw5=new FileWriter(new File("keyword.txt"),true);
        while(sc.hasNext()){
            String b=sc.next();
            if(b.equals(i[0])){
                System.out.println(b+" is an identifier");
                fw1.write(b+" ");
            }
            else if((b.equals(l[0]))||(b.equals(l[1]))||(b.equals(l[2]))){
                System.out.println(b+" is a literal");
                fw2.write(b+" ");
            }
            else
if((b.equals(t[0]))||(b.equals(t[1]))||(b.equals(t[2]))||(b.equals(t[3]))||(b.equals(t[4]))||(b.equals(t[5])))
{
                System.out.println(b +" is a terminal");
                fw3.write(b+" ");
            }
            else if((b.equals(o[0]))||(b.equals(o[1]))){
                System.out.println(b +" is an operator");
```

Scanned by CamScanner

```
                fw4.write(b+" ");
            }
            else if((b.equals(k[0])) || (b.equals(k[1]))) {
                System.out.println(b +" is a keyword");
                fw5.write(b+" ");
            }


        }
        fw1.close();
        fw2.close();
        fw3.close();
        fw4.close();
        fw5.close();
    }
}
```

The program to input in the Lexical Analyzer is written in notepad and will be read by the above program.

```
void main ( )
{
int a , b , c ;
c = a + b ;
}
```

Output

```
C:\Documents and Settings\teb2>javac Demo.java
C:\Documents and Settings\teb2>java Demo
void is a keyword
main is an identifier
( is a terminal
) is a terminal
{ is a terminal
int is a keyword
a is a literal
, is a terminal
b is a literal
, is a terminal
c is a literal
; is a terminal
c is a literal
```

Scanned by CamScanner

```
= is an operator
a is a literal
+ is an operator
b is a literal
; is a terminal
} is a terminal

C:\Documents and Settings\teb2> */
```

Experiment 5 : Implementation of Shift Reduce Parser.

Solution :

Aim

To implement a shift reduce parser. Shift Reduce parser parses the tree from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.It requires some data structures i.e.

- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Theory

Basic Operations

- **Shift :** This involves moving of symbols from input buffer onto the stack.
- **Reduce :** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.
- **Accept :** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.
- **Error :** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example 1

Consider the grammar

S ->S + S
S ->S * S
S -> id

Perform Shift Reduce parsing for input string "id + id + id".

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	id+id$	Reduce by S → id
$S	+id+id$	Shift
$S+	id+id$	Shift

Scanned by CamScanner

Stack	Input Buffer	Parsing Action
$S+id	+id$	Reduce by S → id
$S+S	+id$	Shift
$S+S+	id$	Shift
$S+S+id	$	Reduce by S → id
$S+S+S	$	Reduce by S → S+S
$S+S	$	Reduce by S → S+S
$S	$	Accept

Program Logic

Algorithm

1. Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
2. Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.

A String $\xrightarrow[\text{reduce to}]{}$ the starting symbol

3. Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
4. At the shift action, the current symbol in the input string is pushed to a stack.
5. At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Conclusion

Thus, we have studied and implemented a Shift Reduce Parser and we have performed it on the given string.

Source Code

```
import java.io.*;
class ShiftReduceParser
{   public static void main(String[] args) throws Exception
    {   int no, loc=0, i, j;
        String str, stack="", temp;
        String[][] productions;
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter no. of productions: ");
        no = Integer.parseInt(in.readLine());
        productions = new String[no][2];
        System.out.println("Enter the productions:");
        for (i=0; i < no; i++)
```

Scanned by CamScanner

```
{
    System.out.print("LHS for production "+(i+1)+": ");
    productions[i][0] = in.readLine();
    System.out.print("RHS for production "+(i+1)+": ");
    productions[i][1] = in.readLine();
}
System.out.println("The productions are:");
for (i=0; i < no; i++)
{   System.out.println(productions[i][0]+" -> "+productions[i][1]);    }
System.out.print("Enter a string: ");
str = in.readLine();
while (loc < str.length()-1)
{
    temp = str.substring(loc, str.indexOf(' ', loc));
    loc = str.indexOf(' ', loc)+1;
    for (i=0; i < no; i++)
    {
        if (temp.equals(productions[i][1]))
        {
            temp = productions[i][0];
            break;
        }
    }
    stack = stack+temp;
    System.out.println("Stack contents: "+stack);
    for (i=0; i < no; i++)
    {   if (stack.equals(productions[i][1]))
        {
            stack = productions[i][0];
            break;
        }
    }
}
System.out.println("Stack contents: "+stack);
if (stack.equals(productions[0][0]))
    System.out.println("Accepted.");
else
    System.out.println("Rejected.");
}
}
```

Scanned by CamScanner

Output

```
> java ShiftReduceParser
Enter no. of productions: 2
Enter the productions:
LHS for production 1: E
RHS for production 1: E+E
LHS for production 2: E
RHS for production 2: id
The productions are:
E -> E+E
E -> id
Enter a string: id + id          //id(space)+(space)id(space) - pgm will generate error otherwise
Stack contents: E
Stack contents: E+
Stack contents: E+E
Stack contents: E
Accepted.

> java ShiftReduceParser
Enter no. of productions: 2
Enter the productions:
LHS for production 1: E
RHS for production 1: E+E
LHS for production 2: E
RHS for production 2: id
The productions are:
E -> E+E
E -> id
Enter a string: id * id          //id(space)+(space)id(space) - pgm will generate error otherwise
Stack contents: E
Stack contents: E*
Stack contents: E*E
Stack contents: E*E
Rejected.
```

Experiment 6 : Intermediate Code Generation.

Solution :

Aim

Convert the given infix expression into postfix expression and implement intermediate code generation using C.

Scanned by CamScanner

Theory

The output of the semantic analyzer is given as input to the intermediate code generation. The output of the ICG is given as input to the code optimizer. ICG takes as input the parse tree and generates an intermediate representation which helps in breaking the complexity of code and thereby enables faster translation.

Intermediate code generation uses a simple traversal algorithm to linearise the tree block(3 address code) instructions.

Eg : Consider the HLL instruction:

Position = initial+rate*60

We will refer position, initialize and rate with identifier id1, id2 and id3.

Three Address Code Form :

T1 = int to real(60)
T2 = id3+T1
T3 = id2+T2
Td1 = T3

Basic idea of having an intermediate code(3AC form) is to break down source long expression into simple pieces that-

1. Translate easily into real m/c code.
2. Form a linearized representation of a syntax tree.
3. Allow us to check our own work to this point.
4. Allow machine independent code optimization to be performed.
5. Increase the probability of the compiler.

In 3AC form, the statements are of the form

X: = y op z

There is almost 1 operator in addition to the assignment operator. x, y and z are called addresses.

Program Logic

Algorithm

1. Accept the statement from the user.
2. Convert the RHS of c="operator" in a postfix expression.
   (a) Read the postfix expression character by character.
   (b) If an operand is encountered, push it onto the stack.
   (c) If an operator is encountered, pop two elements from the stack and perform the operation for the operator encountered. Then to see whether this expression has to be stored in a temp variable or the LHS variable, check whether any operand is left in the postfix string. If yes, then store that expression in a temp variable and push the temp variable onto the stack. Else store that expression in the LHS variable.
3. Repeat step 2(b) till end of the postfix expression.
4. Display the 3AC instruction in a format understood by the user.
5. Stop

Scanned by CamScanner

Conclusion

Thus, we have studied and implemented intermediate code generation(3AC form).

Source Code

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
    clrscr();
        while(1)
        {
        printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
        scanf("%d",&ch);
        switch(ch)
    {
case 1:
    printf("\nEnter the expression with assignment operator:");
    scanf("%s",exp);
    l=strlen(exp);
    exp2[0]='\0';
    i=0;
        while(exp[i]!='=')
        {
        i++;
        }
    strncat(exp2,exp,i);
    strrev(exp);
    exp1[0]='\0';
    strncat(exp1,exp,l-(i+1));
    strrev(exp1);
    printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
    break;
case 2;
    printf("\nEnter the expression with arithmetic operator:");scanf("%s",ex);
```

Scanned by CamScanner

```
    strcpy(exp,ex);
    l=strlen(exp);
    exp1[0]='\0';
    for(i=0;i<l;i++)
    {
        if(exp[i]=='+'||exp[i]=='-')
        {
            if(exp[i+2]=='/'||exp[i+2]=='*')
                {
                pm();
                    break;
                }
            else
            {
                    plus();
                    break;
            }
        }
        else if(exp[i]=='/'||exp[i]=='*')
                {
                    div();
                    break;
                }
    }
    break;
case 3:
    printf("Enter the expression with relational operator");
    scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp(op,">=")==0)||
(strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
    printf("Expression is error");
else
{
    printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
    addr++;
    printf("\n%d\t T:=0",addr);
    addr++;
    printf("\n%d\t goto %d",addr,addr+2);
    addr++;
```

Scanned by CamScanner

```
    printf("\n%d\t T:=1",addr);
}
    break;
case 4:
    exit(0);
        }
    }
}
void pm()
{
 strrev(exp);
 j=l-i-1;
 strncat(exp1,exp,j);
 strrev(exp1);
 printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
 strncat(exp1,exp,i+2);
 printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
 strncat(exp1,exp,i+2);
 printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
```

Output

```
1. assignment
2. arithmetic
3. relational
4. Exit
Enter the choice:1
Enter the expression with assignment operator:
a=b
Three address code:
temp=b
a=temp

1.assignment
2.arithmetic
3.relational
4.Exit
```

Scanned by CamScanner

```
Enter the choice:2
Enter the expression with arithmetic operator:
a+b-c
Three address code:
temp=a+b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a-b/c
Three address code:
temp=b/c
temp1=a-temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a*b-c
Three address code:
temp=a*b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:a/b*c
Three address code:
temp=a/b
temp1=temp*c

1.assignment
2.arithmetic
3.relational
4.Exit
```

Scanned by CamScanner

```
Enter the choice:3
Enter the expression with relational operator
a<=b
100 if a<=b goto 103
101 T:=0
102 goto 104
103 T:=1

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4
```

Experiment 7 : Implementation of Code Optimization.

Solution :

Aim

Write a program to read a source file and optimize it using code optimization technique.

Theory

The code optimization phase attempts to improve the intermediate so that a faster running machine could be generated. The program that performs code optimization is called optimizer.

Problem Logic

Algorithm

1. Accept Input in un-optimized from the user.
2. Eliminate it by using common sub expression i.e. occurrences of an expression Is called a common sub expression if it occurs previously so that now use just assign previous value it.
3. The transform costly operation by cheaper ones.
4. Eliminate call induction variable and code motion and strength reduction.
5. Eliminate dead code, common sub expression and constant folding.
6. Use copy propagation: the idea behind the copy propagation transformation is to use g for f, whenever possible is after the copy statement. This may not appear to be an improvement but it gives us the opportunity to eliminate the assignment fox.

Conclusion

Thus, we studied and implemented code optimization.

Source Code

```
#include<stdio.h>
#include<stdlib.h>
char temp ='A'-1;
int index1=0;
```

Scanned by CamScanner

```
char addtotable(char, char, char);
struct expr
    {
    char operand1;
    char operand2;
    char operator;
    char result;
    };
%}
%union
{ char symbol; }
%left '+' '-'
%left '*' '/'
%token <symbol> NUMBER ID
%type <symbol> exp
%%
st: ID '=' exp ';' {addtotable((char)$1,(char)$3,'=');};
exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}
   |exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}
   |exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
   |exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
   |'(' exp ')' {$$ = (char)$2;}
   |NUMBER {$$ = (char)$1;}
   |ID {$$=(char)$1;};
%%
struct expr arr[20];
void quad()
{
  int i;
  for(i=0;i<index1;i++)
    {
    if(arr[i].operator=='!') continue;
    printf("%c:=\t",arr[i].result);
    printf("%c\t",arr[i].operand1);
    printf("%c\t",arr[i].operand2);
    printf("%c\n",arr[i].operator);
    }
}
```

Scanned by CamScanner

```
int main()
{ temp='A'-1;
  printf("Enter the expression\n");
  yyparse();
  quad();
  opt();
  printf("After Optimization\n");
  quad();
}
int yywrap()
{   return 1; }
void yyerror(char *s)
{   printf("Error %s",s); }
char addtotable(char a, char b, char c)
{ temp++;
  arr[index1].operand1=a;
  arr[index1].operand2=b;
  arr[index1].operator=c;
  arr[index1].result=temp;
  index1++;
  return temp;
}
void opt()
    {   int i,j;
        for(i=0;i<index1;i++)
        for(int j=i+1;j<index1;j++)
        {
  if(arr[i].operator==arr[j].operator && arr[i].operand1 ==arr[j].operand1 && arr[i].operand2 ==
arr[j].operand2)
            {
                int z;
                for(int z=j+1;z<index1;z++){
                if(arr[z].operand1==arr[j].result) arr[z].operand1=arr[i].result;
                 if(arr[z].operand2==arr[j].result) arr[z].operand2=arr[i].result;
            }
  arr[j].operator='!';
        }
    }
}
```

Scanned by CamScanner

Output

```
Enter the expression
a=b*c+b*c+b*c+b*c;
A:= b c *
B:= b c *
C:= A B +
D:= b c *
E:= C D +
F:= b c *
G:= E F +
H:= a G =
After Optimization
A:= b c *
C:= A A +
E:= C A +
G:= E A +
H:= a G =
```

Experiment 8 : To convert NFA to DFA.

Solution :

Aim : To convert NFA to DFA.

Theory

NFA

In the theory of computation, a nondeterministic finite state machine or nondeterministic finite automaton (NFA) is a finite state machine where for each pair of state and input symbol there may be several possible next states. This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined. Both types of automata recognize only regular languages.

Formal Definition

- The ordinary NFA is defined as a 5-tuple, $(Q, \Sigma, T, q0, F)$, consisting of :
  - A finite set of states Q
  - A finite set of input symbols Σ
  - A transition function $T : Q \times \Sigma \rightarrow P(Q)$.
  - An initial (or start) state $q0 \in Q$
  - A set of states F distinguished as accepting (or final) states $F \subseteq Q$.
- Here, P(Q) denotes the power set of Q. The NFA with ε-moves (also sometimes called NFA-epsilon or NFA-lambda) replaces the transition function with one that allows the empty string ε as a possible input, so that one has instead

  $T : Q \times (\cup \{\varepsilon\}) \rightarrow P(Q)$.
- It can be shown that ordinary NFA and NFA with epsilon moves are equivalent, in that, given either one, one can construct the other, which recognizes the same language.

Scanned by CamScanner

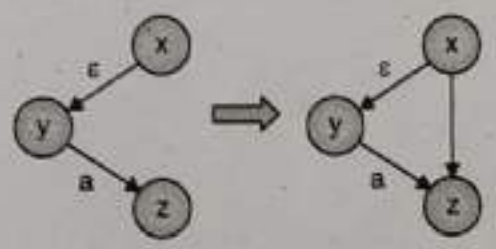
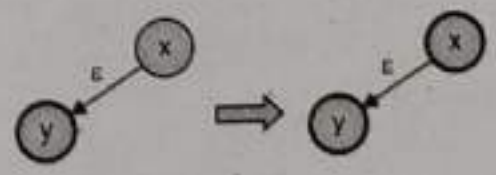
DFA

Formal Definition

- A DFA is a 5-tuple, $(Q, \Sigma, \delta, q0, F)$, consisting of
- A finite set of states (Q)
- A finite set of input symbols called the alphabet (Σ)
- A transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- A start state ($q0 \in Q$)
- A set of accept states ($F \subseteq Q$)

Program Logic

Algorithm : NFA -> DFA

Input	NFA $M = (\Sigma, Q, \Delta, q_0, F)$
Output	DFA $M'' = (\Sigma, Q'', \Delta, q_0'', F'')$
Method	**Step 1 :** Removal of the edges labeled ε. Apply the following rules on the current transition diagram until all edges labelled ε are removed from the diagram. The edges on the left sides of the rules are identified with edges in the current transition diagram. All newly occurring edges on the right side of a rule correspond to newly created edges and thus to new transitions. After all rules that are applicable on an edge labelled ε have been applied, this edge is removed. **(A)**  **(B)**  Let the NFA without edges labelled ε formed by these rules be : $M' = (\Sigma, Q', \Delta, q_0', F')$.

Scanned by CamScanner

	Step 2 : NFA M' without ε-transitions → DFA M". States in M" are sets of states of M'. The starting state of M" is ε-SS(q_0'). The additional states generated for Q" are marked, as soon as their successor states or transitions under all symbols of Σ have been generated. The marking of generated states is determined in the partial function marked : P(Q) → **bool**.

Conclusion

Thus we have studied and implemented NFA to DFA conversion.

Source Code

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
struct state
{
    int count1,count2;
    int on0[5],on1[5];
}temp1,temp2,temp3;
struct newstate
{
    int prev[5],new1,c;
}ns[5];
int change=0;
class table
{
    private:
        int count;
        state p[20];
    public:
        void getdata()
        {
            int i,j;
            printf("Enter the number of states :");
            scanf("%d",&count);

            for(i=0;i<count;i++)
            {
                printf("for state %d\n",i);
                printf("Enter number of new states and their values on 0\n");
                scanf("%d",&p[i].count1);
```

Scanned by CamScanner

```
        for(j=0;j<p[i].count1;j++)
            scanf("%d",&p[i].on0[j]);
        printf("Enter number of new states and their values on 1\n");
        scanf("%d",&p[i].count2);
        for(j=0;j<p[i].count2;j++)
            scanf("%d",&p[i].on1[j]);
    }
}
void putdata()
{
    int i,j;
    printf("state\\input\t0\t1\n");
    for(i=0;i<count;i++)
    {
        printf("\tp%d\t",i);
        for(j=0;j<p[i].count1;j++)
            printf("p%d,",p[i].on0[j]);
        printf("\t");
        for(j=0;j<p[i].count2;j++)
            printf("p%d,",p[i].on1[j]);
        printf("\n");
    }
}
int returncnt()
{
    return(count);
}
int check(int i,int j)
{
    if(j==0)
    {
        if(p[i].count1==1)
            return(1);
    }
    else if(j==1)
    {
        if(p[i].count2==1)
            return(1);
```

Scanned by CamScanner

```
    }
    return(0);
}
state getstate(int i)
{
    return(p[i]);
}
void addstate(state temp)
{
    count++;
    p[count-1]=temp;
}
void init()
{
    count=0;
}
void newtrans(int in)
{
    int i,j,l,t,flag,k=0,arr2[10],arr[10],cnt;
    if(in==0)
    {
        cnt=temp1.count1;
        for(i=0;i<cnt;i++)
            arr[i]=temp1.on0[i];
    }
    else
    {
        cnt=temp1.count2;
        for(i=0;i<cnt;i++)
            arr[i]=temp1.on1[i];
    }
    for(i=0;i<cnt;i++)
    {
        for(j=0;j<p[arr[i]].count1;j++)
        {
            flag=0;
            for(l=0;l<k;l++)
            {
```

Scanned by CamScanner

```
                if(arr2[l]==p[arr[i]].on0[j])
                {
                        flag=1;
                        break;
                }
        }
        if(flag==0)
                arr2[k++]=p[arr[i]].on0[j];
    }
}
for(i=0;i<k;i++)
{
    for(j=i+1;j<k-1;j++)
    {
        if(arr2[i]>arr2[j])
        {
                t=arr2[j];
                arr2[j]=arr2[i];
                arr2[i]=t;
        }
    }
}
for(i=0;i<k;i++)
    temp3.on0[i]=arr2[i];
k=0;
for(i=0;i<cnt;i++)
{
    for(j=0;j<p[arr[i]].count2;j++)
    {
        flag=0;
        for(l=0;l<k;l++)
        {
                if(arr2[l]==p[arr[i]].on1[j])
                {
                        flag=1;
                        break;
                }
        }
```

Scanned by CamScanner

```
                if(flag==0)
                        arr2[k++]=p[arr[i]].on1[j];
            }
        }
        for(i=0;i<k;i++)
        {
            for(j=i+1;j<k-1;j++)
            {
                if(arr2[i]>arr2[j])
                {
                    t=arr2[j];
                    arr2[j]=arr2[i];
                    arr2[i]=t;
                }
            }
        }
        temp3.count2=k;
        for(i=0;i<k;i++)
            temp3.on1[i]=arr2[i];
    }
    void incent()
    {
        count++;
    }
};
int checkprev(int count,int arr[5])
{
    int i,j,flag;
    for(i=0;i<change;i++)
    {
        flag=1;
        if(ns[i].c==count)
        {
            flag=0;
            for(j=0;j<count;j++)
            {
                if(arr[j]!=ns[i].prev[j])
                {
```

Scanned by CamScanner

```
                flag=1;
                break;
            }
        }
    }
    if(flag==0)
        return(ns[i].new1);
}
return(-1);
}
int main()
{
    int i,m,j,n,flag,valid1,valid2,there;
    clrscr();
    table nfa,dfa;
    nfa.getdata();
    nfa.putdata();
    n=nfa.returncnt();
    dfa.init();
    for(i=0;i<n;i++)
    {
        valid1=nfa.check(i,0);
        valid2=nfa.check(i,1);
        temp1=nfa.getstate(i);
        if(valid1==1&&valid2==1)
            dfa.addstate(temp1);
        else
        {
            if(valid1==1)
            {
                temp2.count1=1;
                temp2.on0[0]=temp1.on0[0];
            }
            else
            {
                there=checkprev(temp1.count1,temp1.on0);
                if(there!=-1)
                {
```

Scanned by CamScanner

```
            temp2.count1=1;
            temp2.on0[0]=there;
        }
        else
        {
            flag=1;
            temp2.count1=1;
            temp2.on0[0]=n;
            for(j=0;j<temp1.count1;j++)
            {
                ns[change].prev[j]=temp1.on0[j];
            }
            nfa.newtrans(0);
            ns[change++].new1=n;
            n++;
        }
    }
    if(valid2)
    {
        temp2.count2=1;
        temp2.on1[0]=temp1.on1[0];
    }
    else
    {
        there=checkprev(temp1.count2,temp1.on1);
        if(there!=-1)
        {
            temp2.count2=1;
            temp2.on1[0]=there;
        }
        else
        {
            flag=1;
            temp2.count2=1;
            temp2.on1[0]=n;
            for(j=0;j<temp1.count2;j++)
            {
                ns[change].prev[j]=temp1.on1[j];
```

Scanned by CamScanner

```
                }
                nfa.newtrans(l);
                ns[change++].new1=n;
                n++;
            }
        }
        if(flag==1)
            nfa.addstate(temp3);
        dfa.addstate(temp2);
    }
}
nfa.putdata();
dfa.putdata();
getch();
return 0;
}
```

Output

```
Enter the number of states :4
for state 0
Enter number of new states and their values on 0
1 0
Enter number of new states and their values on 1
1 1
for state 1
Enter number of new states and their values on 0
1 0
Enter number of new states and their values on 1
1 2
for state 2
Enter number of new states and their values on 0
2 1 3
Enter number of new states and their values on 1
1 2
for state 3
Enter number of new states and their values on 0
1 0
Enter number of new states and their values on 1
1 2
```

Scanned by CamScanner

state\input	0	1
p0	p0,	p1,
p1	p0,	p2,
p2	p1,p3,	p2,
p3	p0,	p2,

state\input	0	1
p0	p0,	p1,
p1	p0,	p2,
p2	p4,	p2,
p3	p1,	p2,
p4	p0,	p2,

❑❑❑

Scanned by CamScanner

Semester VI - Computer Engineering

Tech-Max Publications

Office address : B/5, First Floor, Maniratna Complex,
Aranyeshwar Corner, Pune - 411009. Maharashtra State, India.
Tel. : 91-20-24217965, 91-20-24225065
E-mail : info@techmaxbooks.com

facebook.com/techmax.publications

Books are available at :

KRISHNA BOOKS COLLECTIONS

Ground Floor, Krishna Niwas Building, Behind BEST
Niwas Building, Near Napoo Hall, Chandavarkar Road,
Matunga East, Mumbai 400019.
Mobile No. : 9820741455 / 9833082745 / 9833082761

ISBN : 978-93-88723-55-8

Price ₹ 325/-

Tech-Max Publications
www.techmaxbooks.com

For Library Orders Contact
Ved Book Distributors
Mr. Sachin Waingade
88975 71421 / 92208 77214
E-mail
swaingade.techmax@gmail.com /
info@techmaxbooks.com

Scanned by CamScanner