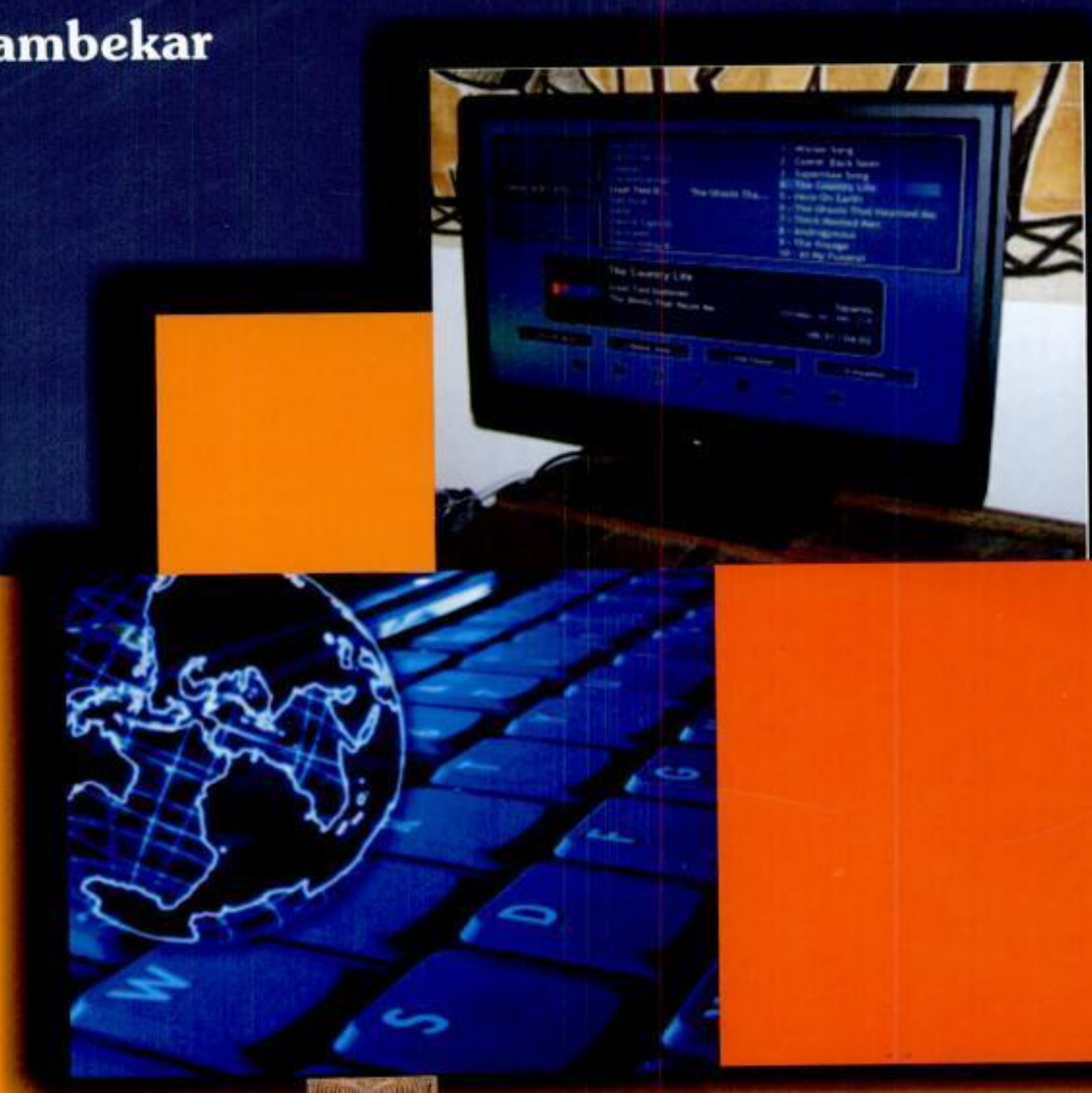


First Edition : 2009

Compiler Design

A. A. Puntambekar



Technical Publications PuneTM

Copyrighted material



Compiler Design

ISBN 9788184313444

All rights reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :

Technical Publications Pune[®]

#1, Amit Residency, 412, Shaniwar Peth, Pune - 411 030, India.

Printer :

Alert DTPrinters
Sr.no. 10/3, Sinhgad Road,
Pune - 411 041

Table of Contents

Chapter-1 Overview of Compilation	(1 - 1) to (1 -20)
1.1 Introduction.....	1 - 1
1.2 Why to Write Compiler ?.....	1 - 1
1.2.1 Compiler : Analysis - Synthesis Model	1 - 2
1.2.2 Execution of Program	1 - 2
1.3 Compilation Process in Brief	1 - 4
1.3.1 The Phases of Compiler	1 - 4
1.4 Cousins of the Compiler	1 - 9
1.5 Concept of Pass.....	1 - 11
1.6 Types of Compiler.....	1 - 12
1.7 Bootstrapping of Compiler	1 - 13
1.8 Interpreter	1 - 14
1.9 Comparison of Compilers and Interpreters	1 - 15
1.10 Compiler Construction Tools.....	1 - 15
Solved Exercise.....	1 - 17
Review Questions	1 - 20
Chapter-2 Lexical Analysis	(2 - 1) to (2 - 38)
2.1 Introduction.....	2 - 1
2.2 Role of Lexical Analyzer.....	2 - 1
2.2.1 Tokens Patterns Lexemes.....	2 - 2
2.3 Input Buffering	2 - 4
2.4 Specification of Tokens	2 - 6
2.4.1 Strings and Language	2 - 7
2.4.2 Operations on Language	2 - 7
2.4.3 Regular Expression for Common Programming Language Features	2 - 8

2.4.4 Notations used for Representing Regular Expressions	2 - 9
2.4.5 Non Regular Language	2 - 9
2.5 Recognition of Tokens	2 - 10
2.6 Use of Regular Expression in Lexical Analysis	2 - 12
2.7 Lex-Lexical Analyzer Generator	2 - 17
2.8 LEX Specification and Features	2 - 24
2.9 Design of Lexical Analyzer Generator	2 - 28
2.9.1 Pattern Matching based on NFA's	2 - 28
Solved Exercise	2 - 31
Review Questions	2 - 37

Chapter-3 Top Down Parsing (3 - 1) to (3 - 50)

3.1 Introduction	3 - 1
3.2 Concept of Syntax Analysis	3 - 1
3.2.1 Basic Issues in Parsing	3 - 2
3.2.2 Role of Parser	3 - 3
3.3 Context Free Grammar	3 - 4
3.3.1 Derivation and Parse Trees	3 - 5
3.3.2 Ambiguous Grammar	3 - 10
3.4 Basic Parsing Techniques	3 - 12
3.5 Top down Parser	3 - 12
3.5.1 Problems with Top down Parsing	3 - 14
3.6 Recursive Descent Parser	3 - 19
3.7 Predictive LL(1) Parser	3 - 22
3.7.1 Construction of Predictive LL(1) Parser	3 - 23
Solved Exercise	3 - 33
Review Questions	3 - 50

Chapter-4 Bottom Up Parsing (4 - 1) to (4 - 86)

4.1 Introduction	4 - 1
4.2 Concept of Bottom Up Parser	4 - 1
4.3 Shift Reduce Parser	4 - 5
4.4 Operator Precedence Parser	4 - 8

4.5 LR Parser	4 - 10
4.5.1 SLR Parser	4 - 12
4.5.2 LR(k) Parser	4 - 34
4.5.3 LALR Parser	4 - 43
4.6 Comparison of LR Parsers	4 - 55
4.7 Handling Ambiguous Grammar	4 - 56
4.8 Error Recovery in LR Parser	4 - 63
4.9 Yacc/Automatic Parser Generator	4 - 66
4.9.1 YACC Specification	4 - 67
Solved Exercise	4 - 73
Review Questions	4 - 85

Chapter-5 Semantic Analysis (5 - 1) to (5 - 12)

5.1 Introduction	5 - 1
5.2 Need of Semantic Analysis	5 - 1
5.3 Type Analysis and Type Checking	5 - 2
5.3.1 Type Expression	5 - 2
5.4 What is Intermediate Code?	5 - 4
5.4.1 Benefits of Intermediate Code Generation	5 - 4
5.5 Intermediate Forms of Source Programs	5 - 5
5.5.1 Types of Three Address Statements	5 - 7
5.6 Implementation of Three Address Code	5 - 8
Solved Exercise	5 - 9
Review Questions	5 - 11

Chapter-6 Syntax Directed Translation (6 - 1) to (6 - 32)

6.1 Introduction	6 - 1
6.2 Syntax Directed Definition (SDD)	6 - 1
6.2.1 Construction of Syntax Trees	6 - 10
6.2.1.1 Construction for Syntax Tree for Expression	6 - 10
6.2.1.2 Directed Acyclic Graph for Expression	6 - 12
6.3 Bottom-Up Evaluation of S-Attributed Definitions	6 - 15
6.3.1 Synthesized Attributes on the Parser Stack	6 - 15
6.4 L-attributed Definitions	6 - 19
6.4.1 L-attributed Definition	6 - 19

6.4.2 Translation Scheme	6 - 20
6.4.2.1 Guideline for Designing the Translation Scheme	6 - 22
6.5 Top-Down Translation	6 - 22
6.5.1 Construction of Syntax Tree for the Translation Scheme	6 - 24
6.6 Bottom Up Evaluation of Inherited Attributes	6 - 26
Solved Exercise.....	6 - 28
Review Questions	6 - 32

Chapter-7 Generation of Three Address Code (7 - 1) to (7 - 36)

7.1 Introduction.....	7 - 1
7.2 Conversion of Popular Programming Language Constructs into Intermediate Code Form	7 - 1
7.3 Declarations	7 - 2
7.4 Assignment Statements	7 - 2
7.4.1 Type Conversion	7 - 5
7.5 Arrays	7 - 7
7.6 Boolean Expressions	7 - 13
7.6.1 Numerical Representation	7 - 13
7.6.2 Flow of Control Statements	7 - 15
7.7 Case Statements	7 - 17
7.8 Procedure Calls	7 - 19
7.9 Backpatching.....	7 - 20
7.9.1 Backpatching using Boolean Expressions	7 - 20
7.9.2 Backpatching using Flow of Control Statements	7 - 26
7.10 Intermediate Code Generation using YACC	7 - 28
Solved Exercise.....	7 - 32
Review Questions	7 - 36

Chapter-8 Symbol Tables (8 - 1) to (8 -28)

8.1 Introduction.....	8 - 1
8.2 Symbol Table Format	8 - 1
8.2.1 Entries in Symbol Table	8 - 2
8.2.2 How to Store Names in Symbol Table ?	8 - 3
8.3 Organization of Block Structured Languages	8 - 4

8.3.1 Symbol Table Organization using Linear List	8 - 4
8.3.2 Symbol Table Organization using Self Organizing List	8 - 5
8.3.3 Hashing	8 - 5
8.3.4 Tree Structure Representation of Scope Information	8 - 7
8.4 Block Structure and Non Block Structure Storage Allocation	8 - 7
8.4.1 Activation Record	8 - 7
8.4.2 Local Data	8 - 10
8.4.3 Access to Non Local Names	8 - 11
8.4.3.1 Static Scope or Lexical Scope	8 - 12
8.4.3.2 Lexical Scope for Nested Procedure	8 - 14
8.4.4 Dynamic Scoping (Storage Allocation for Non Block Structured Languages)	8 - 18
8.5 Variable Length Data	8 - 20
8.6 Parameter Passing	8 - 21
8.7 Run Time Stack and Heap Storage Allocation.....	8 - 22
8.7.1 Source Language Issue	8 - 23
8.7.2 Sub Division of Run Time Memory	8 - 23
8.7.3 Storage Allocation Strategies	8 - 25
8.7.3.1 Static Allocation	8 - 25
8.7.3.2 Stack Allocation	8 - 26
8.7.3.3 Heap Allocation	8 - 26
Review Questions	8 - 27

Chapter-9 Code Optimization (9 - 1) to (9 - 16)

9.1 Consideration for Optimization.....	9 - 1
9.2 Classification of Code Optimization	9 - 2
9.3 Scope of Operation	9 - 3
9.4 Local Optimization	9 - 3
9.4.1 Common Subexpression Elimination	9 - 3
9.4.2 Copy Propagation	9 - 4
9.4.3 Dead Code Elimination	9 - 5
9.4.4 Constant Folding	9 - 5
9.5 Loop Optimization	9 - 6

9.6 DAG Representation	9 - 9
9.6.1 DAG Based Local Optimization	9 - 11
Solved Exercise	9 - 13
Review Questions	9 - 16

Chapter-10 Data Flow Analysis (10 - 1) to (10 - 50)

10.1 Introduction	10 - 1
10.2 Basic Block	10 - 1
10.2.1 Some Terminologies used in Basic Blocks	10 - 2
10.2.2 Algorithm for Partitioning into Blocks	10 - 2
10.2.3 Flow Graph	10 - 3
10.2.4 Basic Terminologies used in Loops	10 - 5
10.3 Global Optimization	10 - 9
10.4 Data Flow Analysis	10 - 10
10.4.1 Data Flow Properties	10 - 10
10.5 Data Flow Equations	10 - 14
10.6 Iterative Data Flow Analysis	10 - 15
10.6.1 Reaching Definition	10 - 16
10.6.2 Live Variable Analysis	10 - 20
10.7 Redundant Common Subexpression Elimination	10 - 28
10.8 Copy Propagation	10 - 30
10.9 Induction Variable	10 - 31
Solved Exercise	10 - 33

Chapter-11 Object Code Generation (11 - 1) to (11 - 36)

11.1 Introduction	11 - 1
11.2 Code Generation	11 - 1
11.3 Object Code Forms	11 - 2
11.4 Issues in Code Generation	11 - 3
11.5 Target Machine Description	11 - 6
11.5.1 Cost of the Instruction	11 - 7
11.6 Machine Dependant Code	
Optimization (Peep Hole Optimization)	11 - 8
11.6.1 Characteristics of Peephole Optimization	11 - 9

11.7 Register Allocation and Assignment.....	11 - 10
11.7.1 Global Register Allocation	11 - 11
11.7.2 Usage Count.	11 - 12
11.7.3 Register Assignment for Outer Loop	11 - 13
11.7.4 Graph Coloring for Register Assignment	11 - 13
11.8 Simple Code Generation Algorithm	11 - 14
11.9 DAG for Register Allocation	11 - 17
11.9.1 Rearranging Order	11 - 18
11.9.2 Heuristic Ordering	11 - 19
11.9.3 Labeling Algorithm	11 - 21
11.10 Generic Code Generation Algorithm	11 - 25
Solved Exercise	11 - 29

References	(R - 1)
-------------------	----------------

Compiler Design Lab	(L - 1) to (L - 42)
----------------------------	----------------------------

Overview of Compilation

1.1 Introduction

Compilers are basically translators. Designing a compiler for some language is a complex and time consuming process. Since new tools for writing the compilers are available this process has now become a sophisticated activity. While studying the subject compiler it is necessary to understand what is compiler and how the process of compilation can be carried out. In this chapter we will get introduced with the basic concepts of compiler. We will see how the source program is compiled with the help of various phases of compiler. Lastly we will get introduced with various compiler construction tools.

1.2 Why to Write Compiler ?

In this section we will discuss two things : "*what is compiler ?*" And "*why to write compiler ?*" Let us start with "*what is compiler?*"

Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).

During this process of translation if some errors are encountered then compiler displays them as error messages. The basic model of compiler can be represented as follows

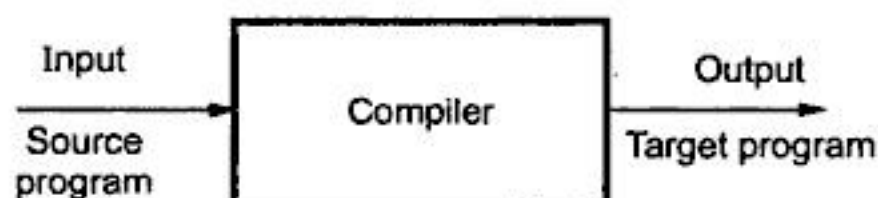


Fig. 1.1 Compiler

The compiler takes a source program as higher level languages such as C, PASCAL, FORTRAN and converts it into low level language or a machine level language such as assembly language.

1.2.1 Compiler : Analysis - Synthesis Model

The compilation can be done in two parts : analysis and synthesis. In **analysis part** the source program is read and broken down into constituent pieces. The syntax and the meaning of the source string is determined and then an intermediate code is created from the input source program. In **synthesis part** this intermediate form of the source language is taken and converted into an equivalent target program. During this process if certain code has to be optimized for efficient execution then the required code is optimized. The analysis and synthesis model is as shown in Fig.1.2.

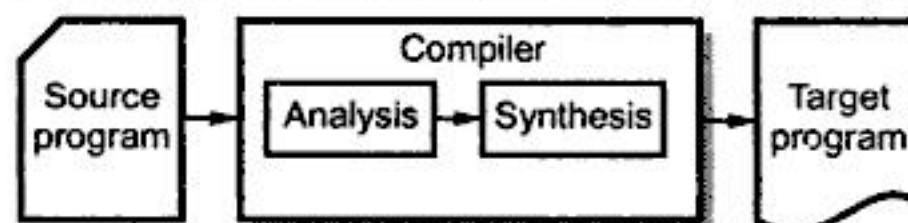


Fig. 1.2 Analysis and synthesis model

The analysis part is carried out in three sub parts

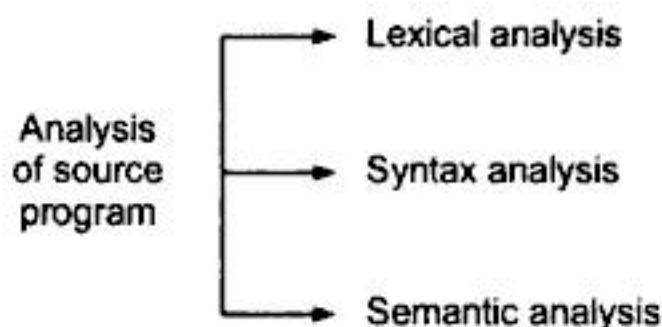


Fig. 1.3

1. Lexical Analysis – In this step the source program is read and then it is broken into stream of strings. Such strings are called **tokens**. Hence tokens are nothing but the collection of characters having some meaning.

2. Syntax Analysis – In this step the tokens are arranged in hierarchical structure that ultimately helps in finding the syntax of the source string.

3. Semantic Analysis – In this step the **meaning** of the source string is determined.

In all these analysis steps the meaning of the every source string should be unique. Hence actions in lexical, syntax and semantic analysis are uniquely defined for a given language. After carrying out the synthesis phase the program gets executed.

1.2.2 Execution of Program

To create an executable form of your source program only a compiler program is not sufficient. You may require several other programs to create an executable target program. After a synthesis phase a target code gets generated by the compiler. This target program generated by the compiler is processed further before it can be run which is as shown in the Fig. 1.4.

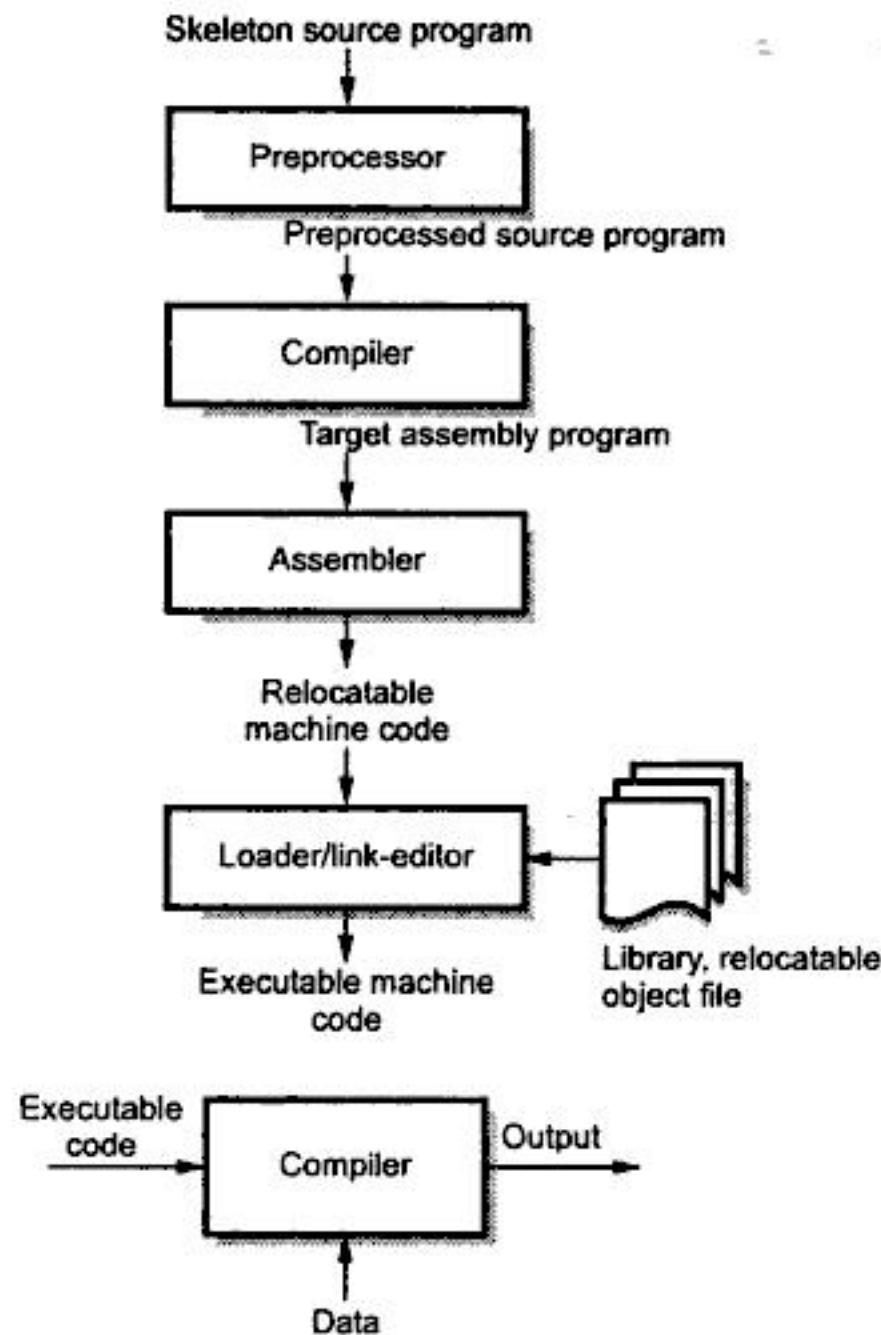


Fig. 1.4 Process of execution of program

The compiler takes a source program written in high level language as an input and converts it into a target assembly language. The assembler then takes this target assembly code as input and produces a relocatable machine code as an output. Then a program loader is called for performing the task of **loading and link editing**. The task of loader is to perform the relocation of an object code. **Relocation** of an object code means allocation of load time addresses which exist in the memory and placement of load time addresses and data in memory at proper locations. The link editor links the object modules and prepares a single module from several files of relocatable object modules to **resolve the mutual references**. These files may be library files and these library files may be referred by any program.

Properties of compiler

When a compiler is built it should possess following properties

1. The compiler itself must be bug-free.
2. It must generate correct machine code.
3. The generated machine code must run fast.
4. The compiler itself must run fast (compilation time must be proportional to program size).

5. The compiler must be portable (i.e. modular, supporting separate compilation).
6. It must give good diagnostics and error messages.
7. The generated code must work well with existing debuggers.
8. It must have consistent optimization.

1.3 Compilation Process in Brief

In this section we will focus on *"How a source program gets compiled?"*

1.3.1 The Phases of Compiler

As we have discussed earlier the process of compilation is carried out in two parts : **analysis and synthesis**. Again the analysis is carried out in three phases : lexical analysis, syntax analysis and semantic analysis. And the synthesis is carried out with the help of intermediate code generation, code generation and code optimization. Let us discuss these phases one by one.

1. Lexical Analysis

The lexical analysis is also called **scanning**. It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group of strings called token. A token is a sequence of characters having a collective meaning. For example if some assignment statement in your source code is as follows,

total = count + rate *10

Then in lexical analysis phase this statement is broken up into series of tokens as follows

1. The identifier **total**
2. The **assignment** symbol
3. The identifier **count**
4. The **plus** sign
5. The identifier **rate**
6. The **multiplication** sign
7. The constant number **10**

The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

2. Syntax Analysis

The syntax analysis is also called **parsing**. In this phase the tokens generated by the lexical analyser are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the tokens together. The hierarchical structure generated in this phase is called **parse tree** or **syntax tree**. For the expression $\text{total} = \text{count} + \text{rate} * 10$ the parse tree can be generated as follows.

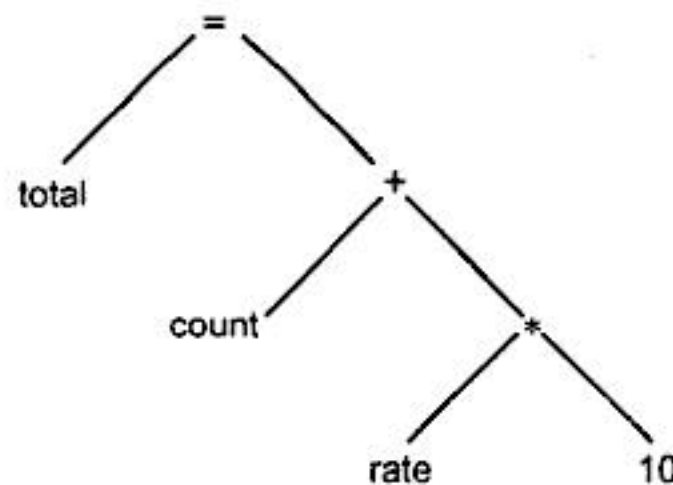


Fig. 1.5 Parse tree for $total = count + rate * 10$

In the statement ' $total = count + rate * 10$ ' first of all $rate * 10$ will be considered because in arithmetic expression the multiplication operation should be performed before the addition. And then the addition operation will be considered. For building such type of syntax tree the production rules are to be designed. The rules are usually expressed by context free grammar. For the above statement the production rules are –

- (1) $E \leftarrow \text{identifier}$
- (2) $E \leftarrow \text{number}$
- (3) $E \leftarrow E_1 + E_2$
- (4) $E \leftarrow E_1 * E_2$
- (5) $E \leftarrow (E)$

where E stands for an expression.

- By rule (1) count and rate are expressions and
- by rule(2) 10 is also an expression.
- By rule (4) we get $rate * 10$ as expression.
- And finally $count + rate * 10$ is an expression.

3. Semantic Analysis

Once the syntax is checked in the syntax analyser phase the next phase i.e. the semantic analysis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if ...else statements or performing arithmetic operations of the expressions that are type compatible, or checking the scope of operation.

For example,

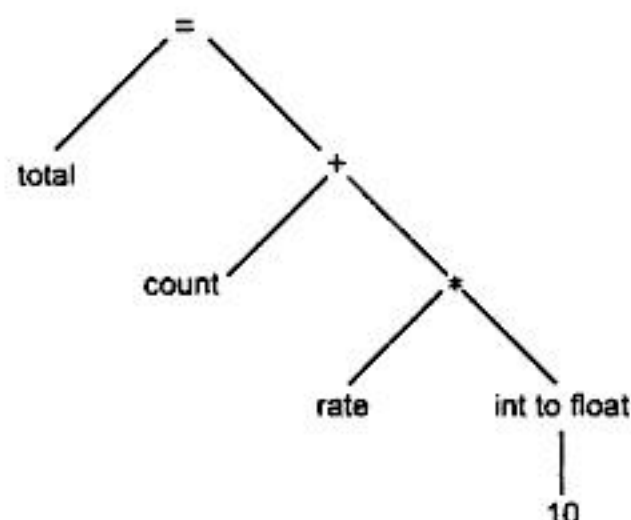


Fig. 1.6 Semantic analysis

Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

4. Intermediate code generation

The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as *three address code, quadruple, triple, posix*. Here we will consider an intermediate code in three address code form. This is like an assembly language. The three address code consists of instructions each of which has at the most three operands. For example,

t1 := int to float (10)

t2 := rate × t1

t3 := count + t2

total := t3

There are certain **properties** which should be possessed by the three address code and those are,

1. Each three address instruction has at the most one operator in addition to the assignment. Thus the compiler has to decide the order of the operations devised by the three address code.
2. The compiler must generate a temporary name to hold the value computed by each instruction.
3. Some three address instructions may have fewer than three operands for example first and last instruction of above given three address code. i.e.

t1 := int to float (10)

total := t3

5. Code optimization

The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory. Thus by optimizing the code the overall running time of the target program can be improved.

6. Code generation

In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions.

MOV rate, R1

MUL #10.0, R1

MOV count, R2

ADD R2, R1

MOV R1, total

Example - Show how an input $a = b + c * 60$ get processed in compiler. Show the output at each stage of compiler. Also show the contents of symbol table.

Symbol table management

- To support these phases of compiler a symbol table is maintained. The task of symbol table is to store identifiers (variables) used in the program.
- The symbol table also stores information about **attributes** of each identifier. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it.
- The symbol table also stores information about the **subroutines** used in the program. In case of subroutine, the symbol table stores the name of the subroutine, number of arguments passed to it, type of these arguments, the method of passing these arguments(may be call by value or call by reference) return type if any.
- Basically symbol table is a data structure used to store the **information about identifiers**.
- The symbol table allows us to find the record for each identifier quickly and to **store or retrieve data** from that record **efficiently**.
- During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attributes of an identifier and therefore the attributes are entered by remaining phases of compiler.
- Various phases can use the symbol table in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what type of identifiers are. Then during code generation typically information about how much storage is allocated to identifier is seen.

Error detection and handling

To err is human. As programs are written by human beings therefore they can not be free from errors. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form of **message**. When the input characters from the input do not form the token, the lexical analyzer detects it as error. Large number of errors can be detected in syntax analysis phase. Such errors are popularly called as **syntax errors**. During semantic analysis; type mismatch kind of error is usually detected.

Front end and Back End

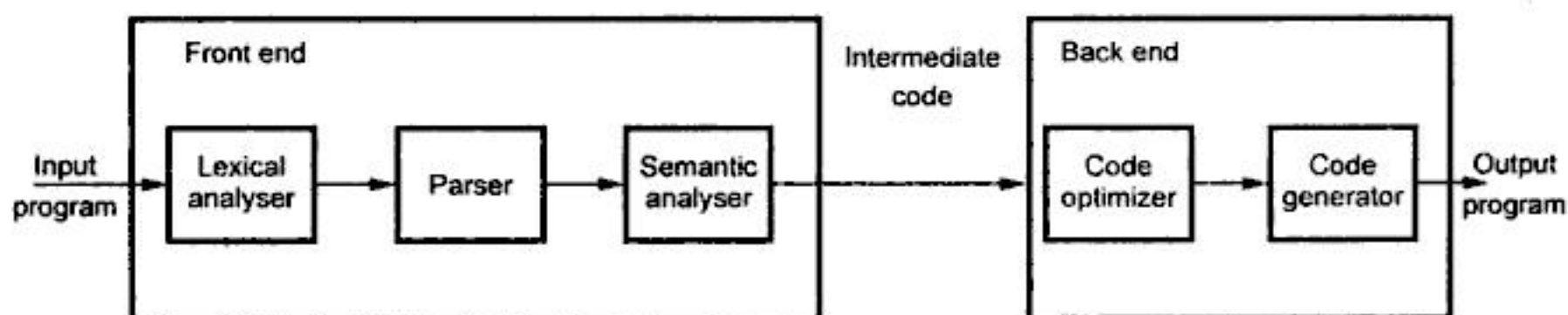


Fig. 1.7 Front end and back end model of compiler

Different phases of compiler can be grouped together to form a **front end** and **back end**. The front end consists of those phases that are primarily dependant on the source language and independent on the target language. The front end consists of analysis part. Typically it includes lexical analysis, syntax analysis, and semantic analysis. Some amount of code optimization can also be done at front end. The back end consists of those phases that are totally dependent upon the target language and independent on the source language. It includes code generation and code optimization. The front end back end model of the compiler is very much advantageous because of following reasons.

1. By keeping the same front end and attaching different back ends one can produce a compiler for same source language on different machines.
2. By keeping different front ends and same back end one can compile several different languages on the same machine.

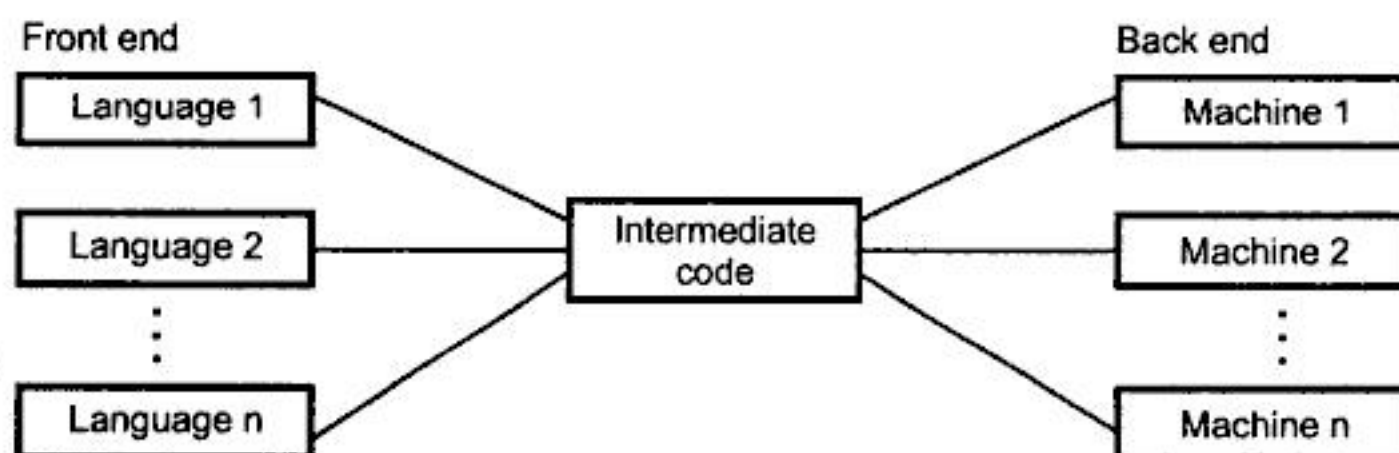


Fig. 1.8 Role of front end and back end

1.4 Cousins of the Compiler

Sometimes the output of preprocessor may be given as input to the compiler.

Cousins of compiler means the context in which the compiler typically operates. Such contexts are basically the programs such as preprocessor, assemblers, loaders and link editors.

Let us discuss them in detail.

1. Preprocessors – The output of preprocessors may be given as the input to compilers. The tasks performed by the preprocessors are given as below

Preprocessors allow user to use macros in the program. Macro means some set of instructions which can be used repeatedly in the program. Thus macro preprocessing task is be done by preprocessors.

Preprocessor also allows user to include the header files which may be required by the program.

For example :

```
#include<stdio.h>
```

By this statement the header file *stdio.h* can be included and user can make use of the functions defined in this header file. This task of preprocessor is called **file inclusion**.

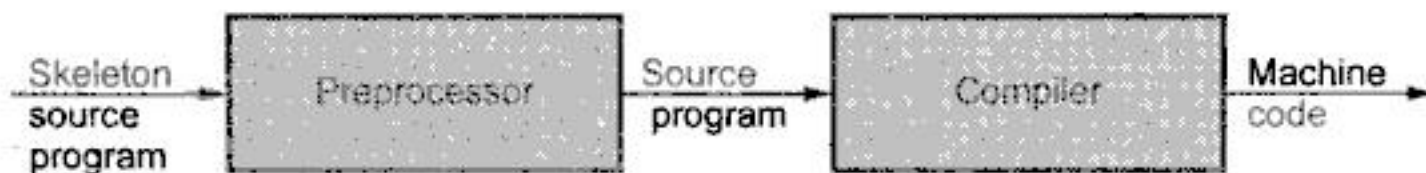


Fig. 1.9 Role of preprocessor

Macro Preprocessor – Macro is a small set of instructions. Whenever in a program macroname is identified then that name is replaced by macro definition (i.e. set of instruction defining the corresponding macro).

For a macro there are two kinds of statements **macro definition** and **macro use**. The macro definition is given by keyword like “*define*” or “*macro*” followed by the name of the macro.

For example :

```
# define PI 3.14
```

Whenever the “PI” is encountered in a program it is replaced by a value 3.14.

2. Assemblers – Some compilers produce the assembly code as output which is given to the assemblers as an input. The assembler is a kind of translator which takes the assembly program as input and produces the machine code as output.

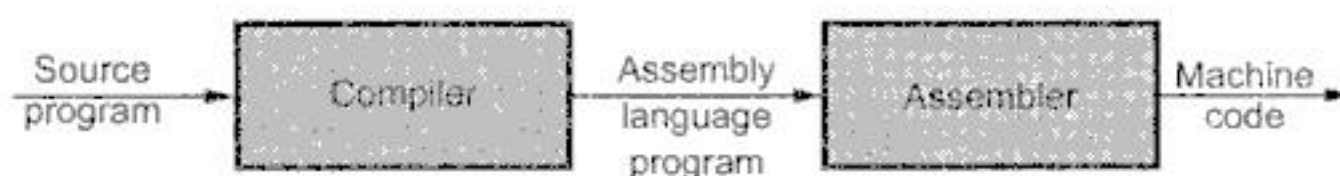


Fig. 1.10 Role of assembler

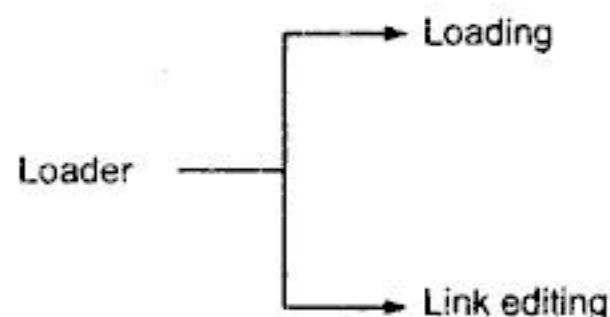
An assembly code is a mnemonic version of machine code. The typical assembly instructions are as given below

```
MOV  a, R1
MUL  #5, R1
ADD  #7, R1
MOV  R1, b
```

The assembler converts these instructions in the **binary language** which can be understood by the machine. Such a binary code is often called as **machine code**. This machine code is a *relocatable* machine code that can be passed directly to the loader/linker for execution purpose.

The assembler converts the assembly program to low level machine language using two passes. A pass means one complete scan of the input program. The end of second pass is the relocatable machine code.

3. Loaders and Link Editors – Loader is a program which performs two functions: loading and link editing. Loading is a process in which the relocatable machine code is read and the relocatable addresses are altered. Then that code with altered instructions and data is placed in the memory at proper location. The job of link editor is to make a single program from several files of **relocatable machine code**. If code in one file refers the location in another file then such a reference is called **external reference**. The link editor resolves such external references also.



1.5 Concept of Pass

One complete scan of the source language is called **pass**. It includes reading an input file and writing to an output file. Many phases can be grouped one pass. It is difficult to compile the source program into a single pass, because the program may have some **forward references**. It is desirable to have relatively few passes, because it takes time to read and write intermediate file. On the other hand if we group several phases into one pass we may be forced to keep the entire program in the memory. Therefore memory requirement may be large.

In the first pass the source program is scanned completely and the generated output will be an easy to use form which is equivalent to the source program along with the additional information about the storage allocation. It is possible to leave a blank slots for missing information and fill the slot when the information becomes available. Hence there may be a requirement of more than one pass in the compilation process. A typical arrangement for optimizing compiler is one pass for scanning and parsing, one pass for semantic analysis and third pass for code generation and target code optimization. C and Pascal permit one pass compilation, Modula-2 requires two passes.

1.6 Types of Compiler

In this section we will discuss various types of compilers.

Incremental compiler :

Incremental compiler is a compiler which performs the recompilation of only modified source rather than compiling the whole source program.

The basic features of incremental compiler are,

1. It tracks the dependancies between output and the source program.
2. It produces the same result as full recompile.
3. It performs less task than the recompilation.
4. The process of incremental compilation is effective for maintainance.

Cross compiler :

Basically there exists three types of languages

1. **Source language** i.e. the application program.
2. **Target language** in which machine code is written.
3. The **Implementation language** in which a compiler is written.

There may be a case that all these three languages are different. In other words there may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called **cross compiler**. Thus by using cross compilation technique platform independency can be achieved.

To represent cross compiler T diagram is drawn as follows

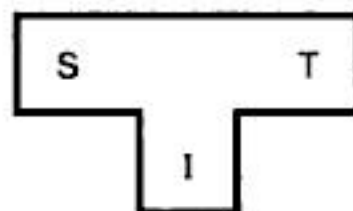


Fig. 1.11 (a) T diagram with S as source, T as target and I as implementation language

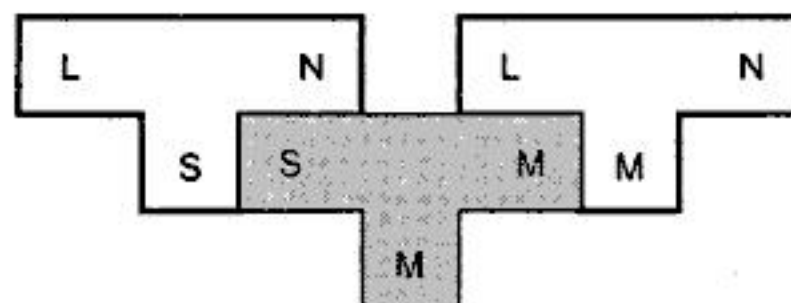


Fig. 1.11 (b) Cross compiler

For source language L the target language N gets generated which runs on machine M.

1.7 Bootstrapping of Compiler

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle far more complicated program. This complicated program can further handle even more complicated program and so on.

Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages. To clearly understand the **bootstrapping** technique consider a following scenario.

Suppose we want to write a cross compiler for new language X . The implementation language of this compiler is say Y and the target code being generated is in language Z . That is, we create X_YZ . Now if existing compiler Y [i.e. compiler written in language Y] runs on machine and generates code for M then it is denoted as Y_MM . Now if we run X_YZ using Y_MM then we get a compiler X_MZ . That means a compiler for source language X that generates a target code in language Z and which runs on machine M .

Following diagram illustrates the above scenario.

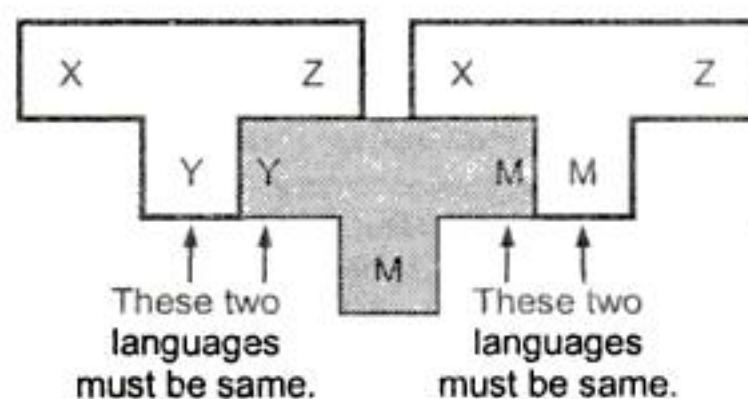
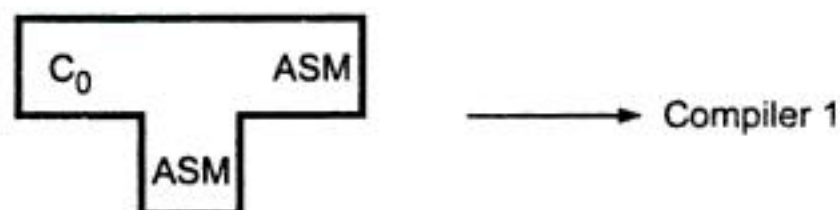


Fig. 1.12

Example - We can create compilers of many different forms. Now we will generate compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

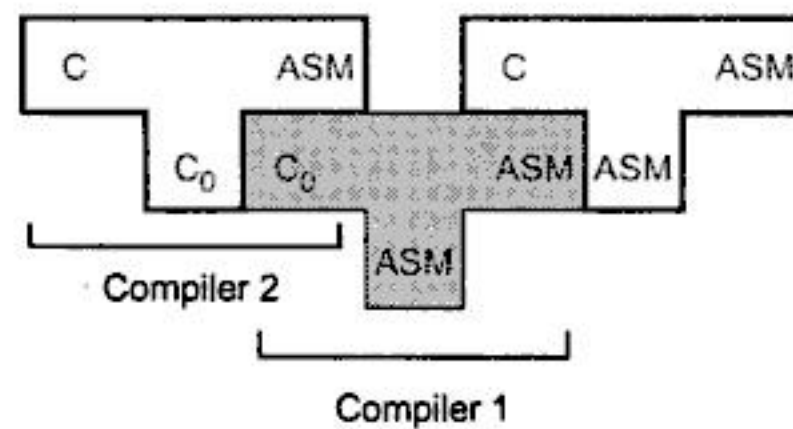
Step 1 : First we write a compiler for a small subset of C in assembly language.



Step 2 : Then using this small subset of C i.e. C_0 , for the source language C the compiler is written.



Step 3 : Finally we compile the second compiler. Using compiler 1 the compiler 2 is compiled.



Step 4 : Thus we get a compiler written in ASM which compiles C and generates code in ASM.

1.8 Interpreter

An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input. It does not produce the object code rather each time the program needs execution. The model for interpreter is as shown in following figure

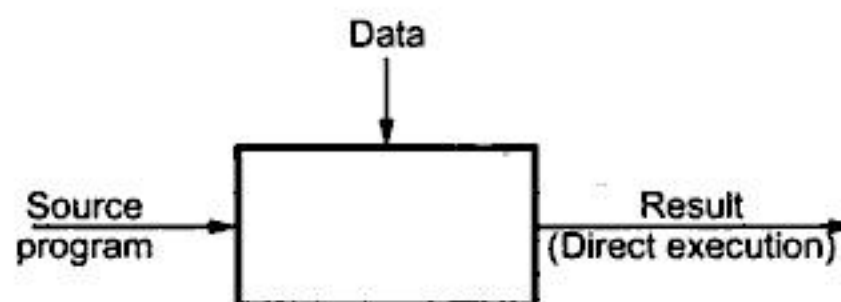


Fig. 1.13 Interpreter

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

Advantages :

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a variable may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it Machine independent.

Disadvantages :

- The execution of the program is slower.
- Memory consumption is more.

1.9 Comparison of Compilers and Interpreters

In interpreter the analysis phase is same as compiler i.e. lexical, syntactic and semantic analysis is also performed in interpreter. In compiler the object code needs to be generated whereas in interpreter the object code need not be generated. During the process of interpretation the interpreter exists along with the source program. Binding and type checking is dynamic in case of interpreter.

The interpretation is less efficient than compilation. This is because the source program has to be interpreted every time it is to be executed and every time it requires analysis of source program. The interpreter can be made **portal** as they do not produce the object code which is not possible in case of compiler. Interpreters **save time in assembling and linking**. Self modifying program can be interpreted but can not be compiled. Design of interpreter is **simpler** than the compiler. Interpreters give us **improved debugging environment** as it can check the errors like out of bound array indexing at run time. Interpreters are most useful in the environment where dynamic program modification is required.

1.10 Compiler Construction Tools

Writing a compiler is tedious and time consuming task. There are some specialized tools for helping in implementation of various phases of compilers. These tools are called compiler construction tools. These tools are also called as compiler-compiler, compiler-generators, or translator writing system. Various compiler construction tools are given as below

1. Scanner generator – These generators generate lexical analysers. The specification given to these generators are in the form of regular expressions.

The UNIX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.

2. Parser generators – These produce the syntax analyzer. The specification given to these generators is given in the form of context free grammar. Typically UNIX has a tool called YACC which is a parser generator.

3. Syntax-directed translation engines - In this tool the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.

4. Automatic code generator – These generators take an intermediate code as input and converts each rule of intermediate language into equivalent machine language. The template matching technique is used. The intermediate code statements are replaced templates that represent the corresponding sequence of machine instructions.

5. Data flow engines – The data flow analysis is required to perform good code optimization. The data flow engines are basically useful in code optimization.

Summary

- Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).
- The compilation can be done in two parts: analysis and synthesis.
- Various phases of compiler are
 - Lexical analysis
 - Syntax analysis
 - Semantic analysis
 - Intermediate code generation
 - Code optimization
 - Code generation
- Along with these phases symbol table management and error detection and handling is done.
- The compiler has a front end and back end skeleton.
- Cousins of compiler are
 - Preprocessor
 - Assembler
 - Loader

- Link Editor
- One complete scan of the source language is called **pass**. It includes reading an input file and writing to an output file.
- An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input.
- Various compiler construction tools are scanner generator, parser generator like LEX and YACC, syntax directed translation engine, automatic code generator, data flow engines.

In this chapter we have discussed the fundamental concepts of compiler. Now we will discuss every phase of compiler in detail in further chapters.

Solved Exercise

Q.1 : *Mention some of the cousins of the compiler.*

Ans. : Cousins of compiler means the context in which the compiler typically operates. Such contexts are basically the programs such as preprocessor, assemblers, loaders and link editors.

1. **Preprocessors** - The preprocessors allow user to use Macros in the program. Preprocessors also allow user to include the header files which may be required by the program.
2. **Assemblers** - Some compilers produce the assembly code as output which is given to the assemblers as input. The assembler takes assembly program as input and produces the machine code as output.
3. **Loaders and link editors** - Loader perform the task of loading and link editing. It is a process in which the relocatable machine code is read and the relocatable addresses are altered. The link editor resolve the external references.

Q.2 : *What are compiler construction tools ?*

Ans. : Writing a compiler is tedious and time consuming task. Therefore there are some specialized tools for implementing various phases of compiler. These tools are called compiler construction tools.

Various compiler construction tools are

- i) Scanner generator - For example : LEX
- ii) Parser generator - For example : YACC
- iii) Syntax directed translation engines.
- iv) Automatic code generator.
- v) Data flow engines.

Q.3 : What is preprocessor ?

Ans. : Preprocessors are kind of programs which take the input source program and produce the output which becomes input to compiler. Various functions performed by preprocessors are

- i) Macro preprocessing ii) File inclusion iii) Language extensions.

Q.4 : What is cross compiler ? Give an example.

Ans. : There may be a compiler which runs on one machine and produces the target code for another machine. Such a compiler is called cross compiler. Thus by using cross compilation technique platform independancy can be achieved.

For example :

For the first version of EQN compiler, the compiler is written in C and the command are generated for TROFF, which is as shown in Fig. 1.14.

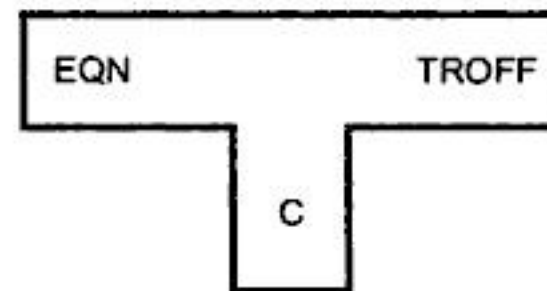


Fig. 1.14

The cross compiler for EQN can be obtained by running it on PDP-11 through C compiler, which produces output for PDP-11 as shown below -

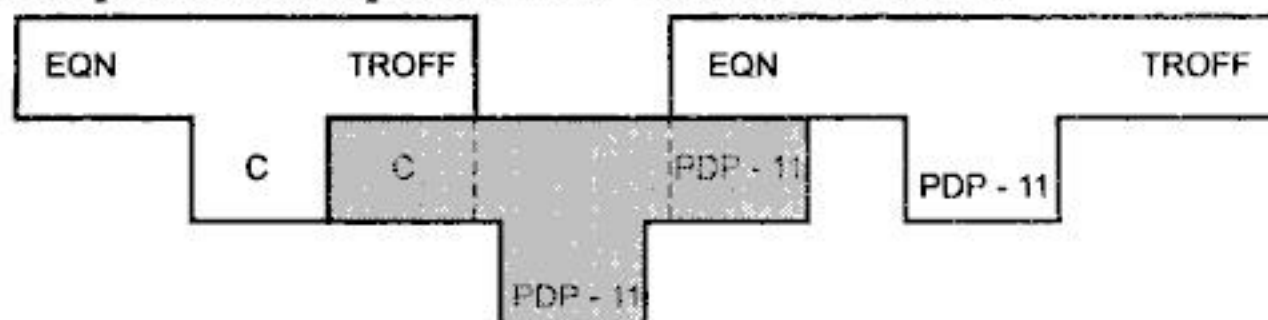


Fig. 1.15

Q.5 : What is an incremental compiler ? Enlist the basic features of incremental compiler.

Ans. : Incremental compiler is such a compilation scheme in which only modified source text gets recompiled and merged with previously compiled code to form a new target code. Thus incremental compiler avoids recompilation of the whole source code on some modification. Rather only modified portion of source program gets recompiled.

Various features of incremental compiler are

- i) During program development process modifications to source program can cause recompilation of whole source text. This overhead is reduced in incremental compiler.
- ii) Run time errors can be patched up just like compile time errors by keeping program modification as it is.
- iii) The compilation process is faster.

- iv) Handling batch programs become very flexible using incremental compiler.
- v) In incremental compiler program structure table is maintained for memory allocation of target code. When a new statement is compiled then new entry for it is created in program structure table. Thus memory allocation required for incremental compiler need not be contiguous.

Q.6 : With the help of a block schematic explain how "compilers - compilers" can reduce the effort in implementing new compiler.

Ans. : There are various tools that help in implementation of various phases of compiler. Such tools are called as compiler compiler.

These tools are -

- i) Scanner generator : These generators generate lexical analyser. The specification given to these generators is in the form of regular expressions. For example LEX.
- ii) Parser generator : These produce syntax analyzer. The specification is given to that generator in the form of context free grammar. For example YACC.
- iii) Syntax directed translation engine : In this tool parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.
- iv) Data flow engines : The data flow engines are required to perform code optimization.
- v) Automatic code generator : It takes as an input the intermediate code and produces the machine code as an output. The block schematic is as shown below -

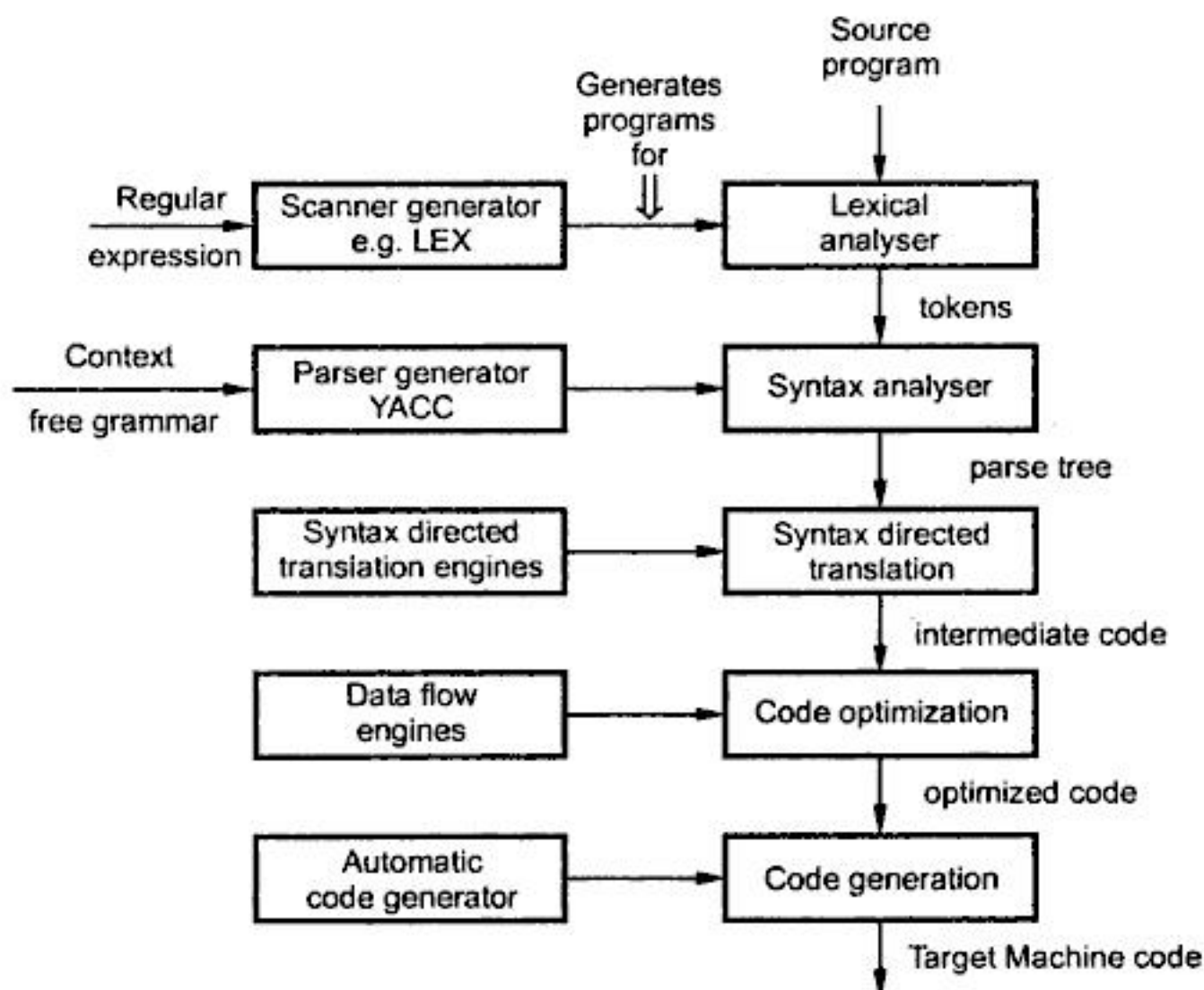


Fig. 1.16

Q.7 : Define pass of compiler. Which are the factors that decide number of passes for a compiler ?

Ans. : Several phases of compiler are grouped into one pass. Thus pass of compiler is simply collection of various phases.

For example : Lexical analysis, syntax analysis and intermediate code generation is grouped together and forms a single pass.

Various factors affecting number of passes in compiler are -

1. Forward reference.
2. Storage limitations.
3. Optimization.

The compiler can require more than one passes to complete subsequent information.

Review Questions

1. What is compiler?
2. Explain analysis - synthesis model of compiler?
3. With a neat block diagram, explain various phases of compiler.
4. What are the advantages of front-end and back-end of compiler ?
5. Explain the concept of compiler-compiler.
6. What is concept of Bootstrapping.
7. What is interpreter ? Differentiate between interpreter and compiler.
8. Enlist the properties of compiler.
9. Explain, why it is better to have two passes in compiler than having one pass ?
10. What is the use of symbol table in the process of compilation.



Lexical Analysis

2.1 Introduction

The process of compilation starts with the first phase called *lexical analysis*. In this phase the input is scanned completely in order to identify the tokens. The token structure can be recognized with the help of some diagrams. These diagrams are popularly known as **finite automata**. And to construct such finite automata regular expressions are used. These diagrams can be translated into a program for identifying tokens. In this chapter we will see what the role of lexical analyzer in compilation process is. We will also discuss the method of identifying tokens from source program. Finally we will learn a tool, LEX which automates the construction of lexical analyzer.

2.2 Role of Lexical Analyzer

Lexical analyzer is the first phase of compiler. The lexical analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens. Each token is a single logical cohesive unit such as identifier, keywords, operators and punctuation marks. Then the parser to determine the syntax of the source program can use these tokens. The role of lexical analyzer in the process of compilation is as shown below –

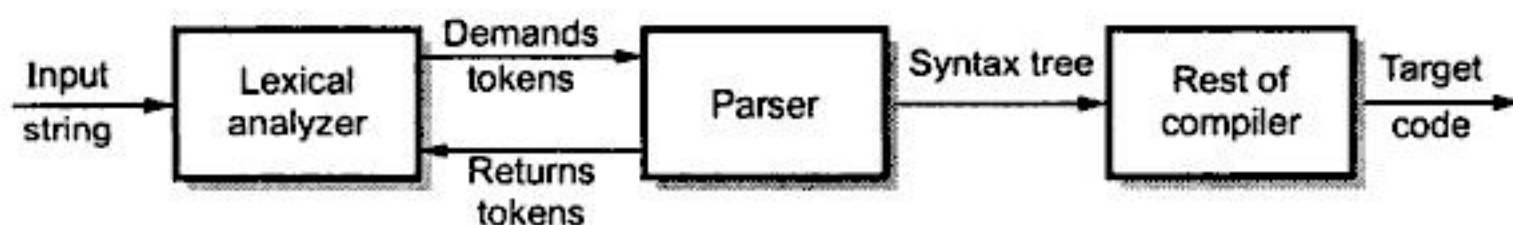


Fig. 2.1 Role of lexical analyzer

As the lexical analyzer scans the source program to recognize the tokens it is also called as scanner. Apart from token identification lexical analyzer also performs following functions.

Functions of lexical analyzer

1. It produces stream of tokens.
2. It eliminates blank and comments.
3. It generates symbol table which stores the information about identifiers, constants encountered in the input.
4. It keeps track of line numbers.
5. It reports the error encountered while generating the tokens.

The lexical analyzer works in two phases in first phase it performs scan and in the second phase it does lexical analysis; means it generates the series of tokens.

2.2.1 Tokens Patterns Lexemes

Let us learn some terminologies, which are frequently used when we talk about the activity of lexical analysis.

Tokens : It describes the class or category of input string. For example identifiers, keywords, constants are called tokens.

Patterns : Set of rules that describe the token.

Lexemes : Sequence of characters in the source program that are matched with the pattern of the token. For example int, i, num, ans, choice;

Let us take one example of programming statement to clearly understand these terms –

if (a<b)

The token gets generated as



Here "if", "(", "a", "<", "b", ")" are all lexemes.

- "if" is a keyword
- "(" is opening parenthesis
- "a" is identifier
- "<" is an operator and so on.

Now let us understand "*what is identifier?*"

1. Identifier is a collection of letters.
2. Identifier is collection of alphanumeric characters and identifier beginning character should be necessarily a letter.

Rules for being valid identifiers

1. The name of the identifier should not begin with letter or any special character.

For example : *1index*, *\$amount*, *_count* are invalid identifiers but *index1* is valid one.

2. The only allowed special character in identifier name is underscore.
3. There should not be any space in the identifier name.

For example

`int total amount`
is invalid identifier.

4. The name of identifier must not be a keyword.

`int switch`
is invalid identifier

Why this phase is called scanner?

When we want to compile a given source program, we submit that program to compiler. A compiler scans the source program and produces sequence of tokens therefore lexical analysis is also called as scanner. For example –The piece of source code is as given below

```
int MAX (int a, int b)
{
    if (a>b)
    return a;
    else
    return b;
}
```

Lexeme	Token
int	Keyword
MAX	Identifier
(Operator
int	Keyword
a	Identifier
,	Operator

int	Keyword
b	Identifier
)	Operator
{	Operator
if	Keyword

Note that the blank and new line characters can be ignored. These streams of tokens will be given to syntax analyser.

2.3 Input Buffering

The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers begin pointer and forward pointer *bptr* and *fptr* to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string as shown below –

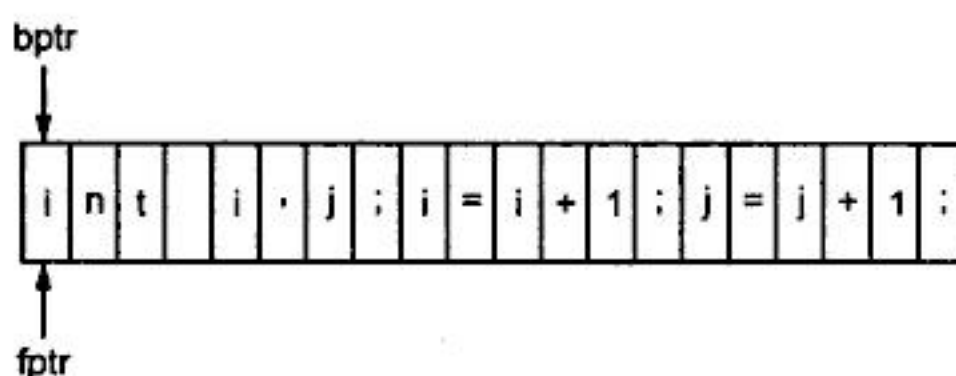


Fig. 2.2 Initial configuration

The *bptr* remains at the beginning of the string to be read and the *fptr* moves ahead to search for end of lexeme. As soon as the blank space is encountered it indicates end of lexeme. In above example as soon as *fptr* encounters a blank space the lexeme "int" is identified.

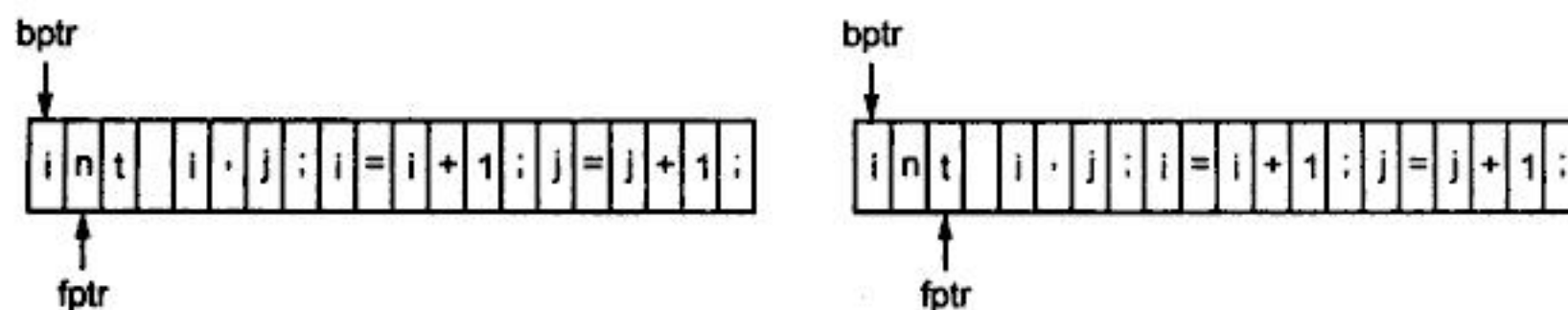


Fig. 2.3 Input buffering

Then *fptr* will be at white space. When *fptr* encounters white space it ignore and moves ahead. Then both the *bptr* and *fptr* is set at next token *i*.

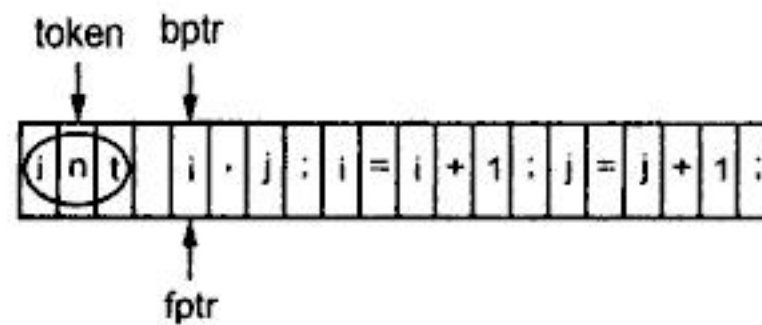


Fig. 2.4 Input buffering

The input character is thus read from secondary storage. But reading in this way from secondary storage is costly. Hence buffering technique is used. A block of data is first read into a buffer, and then scanned by lexical analyzer.

There are two methods used in this context: one buffer scheme and two buffer scheme.

1. One buffer scheme

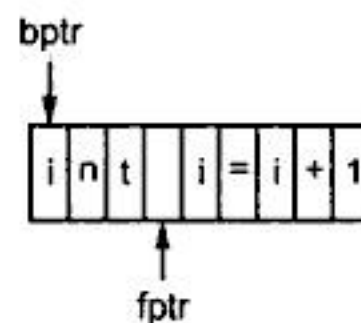


Fig. 2.5 One buffer scheme storing input string

In this one buffer scheme, only one buffer is used to store the input string. But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first part of lexeme.

2. Two buffer scheme

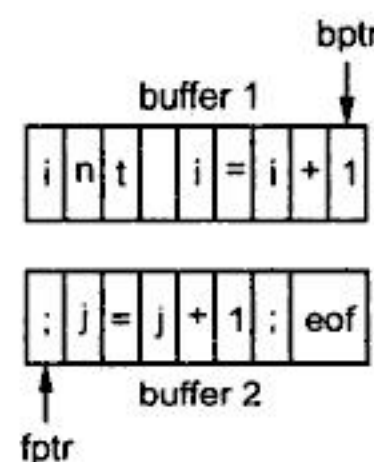


Fig. 2.6 Two buffer scheme storing input string

- To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. In this method, the first buffer and second buffer are scanned alternately. When end of current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then the input can not be scanned completely.

- Initially both the *bptr* and *fp* are pointing to the first character of first buffer. Then the *fp* moves towards right in search of end of lexeme. As soon as blank character is recognized, the string between *bptr* and *fp* is identified as corresponding token.
- To identify the boundary of first buffer end of buffer character should be placed at the end of first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. When *fp* encounters first *eof*, then one can recognize end of first buffer and hence filling up of second buffer is started. In the same way when second *eof* is obtained then it indicates end of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.
- This *eof* character introduced at the end is called *sentinel* which is used to identify the end of buffer.

Code for input buffering

```
if (fp==eof(buff1)) /*encounters end of first buffer*/
{
    /*Refill buffer2*/
    fp++;
}
else if (fp==eof(buff2)) /*encounters end of second buffer*/
{
    /*Refill buffer1*/
    fp++;
}
else if (fp==eof(input))
    return; /*terminate scanning*/
else
    fp++; /*still remaining input has to be scanned */
```

2.4 Specification of Tokens

To specify tokens regular expressions are used. When a pattern is matched by some regular expression then token can be recognized. Let us understand the fundamental concepts of language.

2.4.1 Strings and Language

String is a collection of finite number of alphabets or letters. The strings are synonymously called as words.

- The length of a string is denoted by $| S |$.
- The empty string can be denoted by ϵ .
- The empty set of strings is denoted by Φ .

Following terms are commonly used in strings.

Term	Meaning
Prefix of string	A string obtained by removing zero or more tail symbols, For example for string Hindustan the prefix could be 'Hindu'.
Suffix of string	A string obtained by removing zero or more leading symbols, For example for string Hindustan the suffix could be 'stan'.
Substring	A string obtained by removing prefix and suffix of a given string is called substring. For example for string Hindustan the string 'indu' can be a substring.
Sequence of string	Any string formed by removing zero or more not necessarily the contiguous symbols is called sequence of string For example Hisan can be sequence of string.

2.4.2 Operations on Language

As we have seen that the language is a collection of strings. There are various operations which can be performed on the language.

Operation	Description
Union of two languages L1 and L2.	$L1 \cup L2 = \{\text{set of strings in } L1 \text{ and strings in } L2\}$.
Concatenation of two languages L1 and L2.	$L1.L2 = \{\text{set of strings in } L1 \text{ followed by set of strings in } L2\}$.

Kleen closure of L.	$L^* = \bigcup_{i=0}^{\infty} L^i$ <p>L^* denotes zero or more concatenations of L</p>
Positive closure of L.	$L^+ = \bigcup_{i=1}^{\infty} L^i$ <p>L^+ denotes one or more concatenations of L</p>

For example, Let L be the set of alphabets such as $L = \{A, B, C \dots Z, a, b, c \dots z\}$ and D be the set of digits such as $D = \{0, 1, 2 \dots 9\}$ then by performing various operations as discussed above new languages can be generated as follows –

- $L \cup D$ is a set of letters and digits
- LD is a set of strings consisting of letters followed by digits.
- L^5 is a set of strings having length of 5 each.
- L^* is a set of strings having all the strings including ϵ .
- L^+ is a set of strings except ϵ .

2.4.3 Regular Expression for Common Programming Language Features

Regular are mathematical symbolisms which describe the set of strings of specific language. It provides convenient and useful notation for representing tokens. Here are some rules that describe define the regular expressions over the input set denoted by Σ .

1. ϵ is a regular expression that denotes the set containing empty string.
2. If R_1 and R_2 is regular expression then $R = R_1 + R_2$ (same can also be represented as $R = R_1 | R_2$) is also regular expression which represents union operation.
3. If R_1 and R_2 is regular expression the $R = R_1.R_2$ is also a regular expression which represents concatenation operation.
4. If R_1 is a regular expression then $R = R_1^*$ is also a regular expression which represents kleen closure.

A language denoted by regular expressions is said to be a regular set or a regular language. Let us see some examples of regular expressions.

➡ **Example 2.1 :** Write a Regular Expression (R.E.) for a language containing the strings of length two over $\Sigma = \{0, 1\}$.

Solution : R.E. = $(0+1)(0+1)$

⇒ **Example 2.2 :** Write a regular expression for a language containing strings which end with "abb" over $\Sigma = \{a, b\}$

Solution : R.E. = $(a+b)^*abb$

⇒ **Example 2.3 :** Write a regular expression for a recognizing identifier.

Solution : For denoting identifier we will consider a set of letters and digits because identifier is a combination of letters or letter and digits but having first character as letter always. Hence R.E. can be denoted as,

$$\text{R.E} = \text{letter}(\text{letter}+\text{digit})^*$$

Where letter = (A, B,...Z, a, b,...z) and digit = (0,1,2,...9).

2.4.4 Notations used for Representing Regular Expressions

Regular expressions are tiny units, which are useful for representing the set of strings belonging to some specific language. Let us see notations used for writing the regular expressions.

1. One or more instances – To represent one or more instances + sign is used. If r is a regular expression then r^+ denotes one or more occurrences of r . For example set of strings in which there are one or more occurrences of 'a' over the input set $\{a\}$ then the regular expression can be written as a^+ . It basically denotes the set of $\{a, aa, aaa, aaaa, \dots\}$.
2. Zero or more instances – To represent zero or more instances * sign is used. If r is a regular expression then r^* denotes zero or more occurrences of r . For example, set of strings in which there are zero or more occurrences of 'a' over the input set $\{a\}$ then the regular expression can be written as a^* . It basically denotes the set of $\{\epsilon, a, aa, aaa, \dots\}$.
3. Character classes – A class of symbols can be denoted by $[]$. For example $[012]$ means 0 or 1 or 2. Similarly a complete class of small letters from a to z can be represented by a regular expression $[a-z]$. The hyphen indicates the range. We can also write a regular expression for representing any word of small letters as $[a-z]^*$.

2.4.5 Non Regular Language

A language which can not be described by a regular expression is called a non regular language and the set denoted by such language is called non regular set.

There are some languages which can not be described by regular expressions. For example we cannot write a regular expression to check whether the given language is

palindrome or not. Similarly we cannot write a regular expression to check whether the string is having balanced parenthesis.

Thus regular expressions can be used to denote only fixed number of repetitions. Unspecific information cannot be represented by regular expression.

2.5 Recognition of Tokens

For a programming language there are various types of tokens such as identifier, keywords, constants, and operators and so on. The token is usually represented by a pair token type and token value.

Token type	Token value
------------	-------------

Fig. 2.7 Token representation

The token type tells us the category of token and token value gives us the information regarding token. The token value is also called **token attribute**. During lexical analysis process the symbol table is maintained. The token value can be a pointer to symbol table in case of identifier and constants. The lexical analyzer reads the input program and generates a symbol table for tokens.

For example –

We will consider some encoding of tokens as follows.

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(8	1

)	8	2
+	9	1
-	9	2
=	10	-

Consider, a program code as

```
if (a<10)
    i=i+2;
else
    i=i-2;
```

Our lexical analyzer will generate following token stream.

1,(8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2,(5,107), 10, (5,107), (9,2), (6,110).

The corresponding symbol table for identifiers and constants will be,

Location Counter	Type	Value
100	identifier	a
:	:	:
105	constant	10
:	:	:
107	identifier	i
:	:	:
110	constant	2

In above example scanner scans the input string and recognizes "if" as a keyword and returns token type as 1 since in given encoding code 1 indicates keyword "if" and hence 1 is at the beginning of token stream. Next is a pair (8,1) where 8 indicates parenthesis and 1 indicates opening parenthesis '('. Then we scan the input 'a' it recognizes it as identifier and searches the symbol table to check whether the same entry is present. If not it inserts the information about this identifier in symbol table and returns 100. If the same identifier or variable is already present in symbol table then lexical analyzer does not insert it into the table instead it returns the location where it is present.

2.6 Use of Regular Expression in Lexical Analysis

Lexical analysis is a process of recognizing tokens from input source program. Now the question is how does lexical analyzer recognize tokens, from given source program ? Well, the lexical analyzer stores the input in a buffer. It builds the regular expressions for corresponding tokens. From these regular expressions, finite automata is built. When lexeme matches with the pattern generated by finite automata, the specific token gets recognized. The block schematic for this process is as shown below-

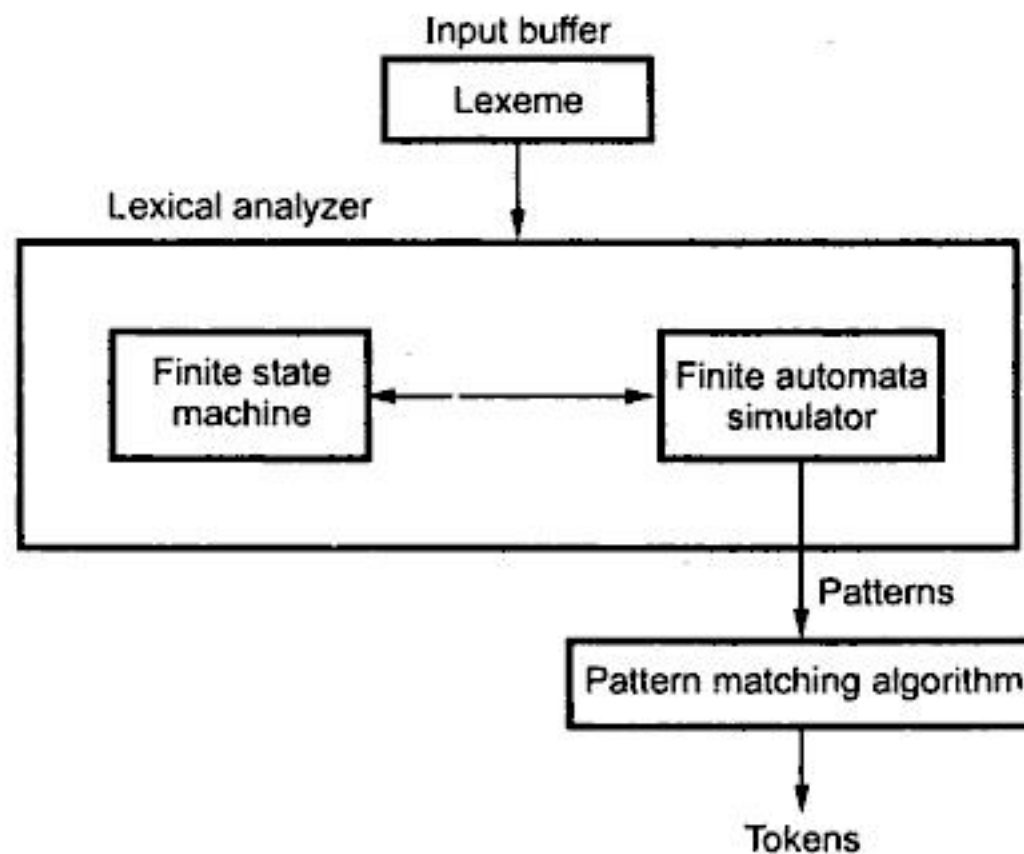


Fig. 2.8 Block schematic of lexical analyzer

While constructing the lexical analyzer we first design the regular expression for recognizing the corresponding token. A diagram resembling the flowchart is built. Such a diagram is called **transition diagram**. The transition diagram elaborates the actions to be taken while recognizing the token. The lexeme is stored in an input buffer. The forward pointer scans the input character-by-character moving from left to right. The transition diagram is used to keep track of the information about characters that are seen as the forward pointer scans the input. Positions in a transition diagram are called **states** and those are drawn by circles and the edges in the diagram represent the **transitions** from one state to another.

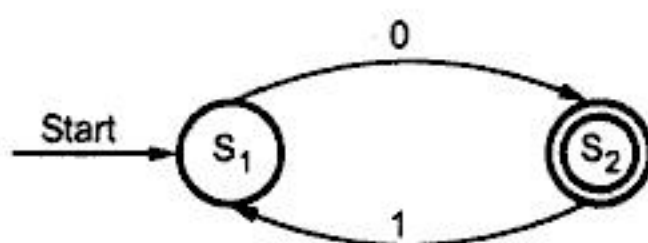


Fig. 2.9 Transition diagram

There is a special state called **start state**, which denotes the starting of transition diagram. From starting state we start recognizing the tokens. After recognizing the token we should reach to the final state. In Fig. 2.9 S_1 is a start state and S_2 is a final state.

Let us take some examples to understand the concept of transition table.

The transition table is a tabular representation of transition diagram. We will first design the transition diagram and then we will build the transition table.

State \ Input	0	1
$\rightarrow S_1$	S_2	-
$\odot S_2$	-	S_1

Fig. 2.10 Transition table

The finite state machine (FSM) can be defined as -

A collection of 5 - tuples $(Q, \Sigma, \delta, q_0, F)$ where

Q is finite set of states, which is non empty.

Σ is input alphabet, indicates input set.

q_0 in Q is the initial state.

F is a set of final states.

δ is a transition function or a function defined for going to next state.

The finite automata can be represented as

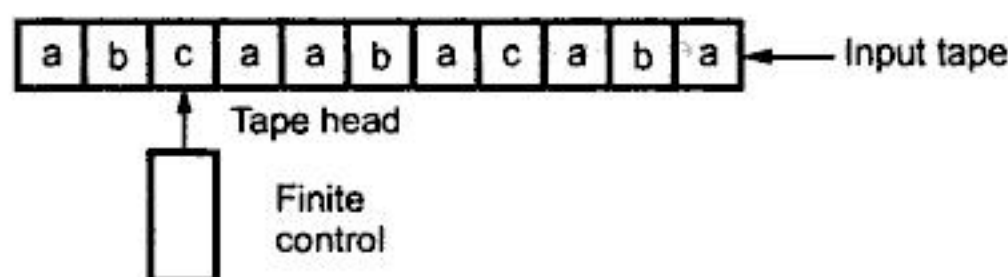


Fig. 2.11 Finite automata

The finite automata is a mathematical representation of finite state machine. The machine has input tape in which input is placed occupying one character in each cell. The tape head reads the symbol from the input tape.

The finite control is always one of internal states which decides what will be the next state after reading the input by the tape header. For example suppose current state is q_1 and suppose now the tape head is pointing to c , it is a finite control which decides what will be the next state at input c . To understand how FA recognizes the particular language, we will see few examples.

► **Example 2.4 :** Design a FA which accepts the only input 101 over the input set $Z = \{0, 1\}$

Solution :

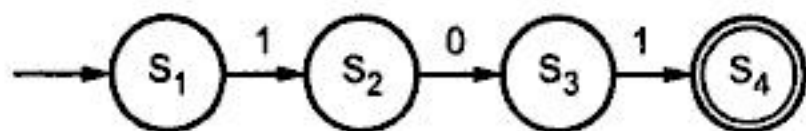


Fig. 2.12 For Ex. 2.4

Note that in the problem statement it is mentioned as only input 101 will be accepted. Hence in the solution we have simply shown the transitions, for input 101. There is no other path shown for other input.

► **Example 2.5 :** Design FA which accepts odd number of 1's and any number of 0's.

Solution : In the problem statement, it is indicated that there will be a state which is meant for odd number of 1's and that will be the final state. There is no condition on number of 0's.

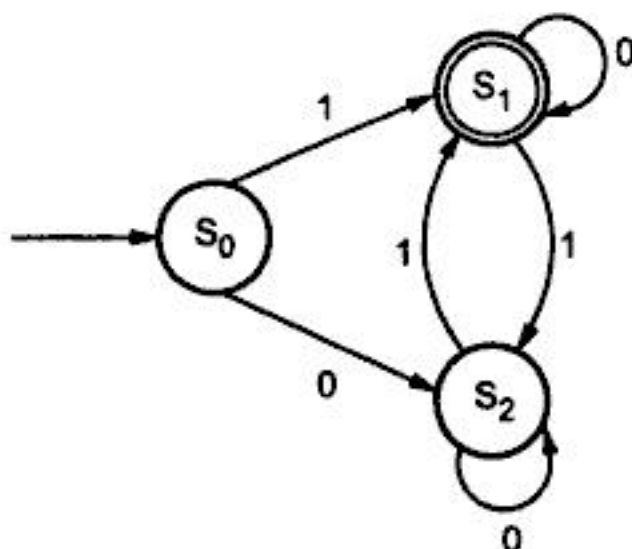


Fig. 2.13 For Ex. 2.5

At the start if we read input 1 then we will go to state S_1 which is a final state as we have read odd number of 1's. There can be any number of zeros at any state and therefore the self loop is applied to state S_2 as well as to state S_1 . For example if the input is 10101101, in this string there are any number of zeros but odd number of ones. The machine will derive this input as follows -

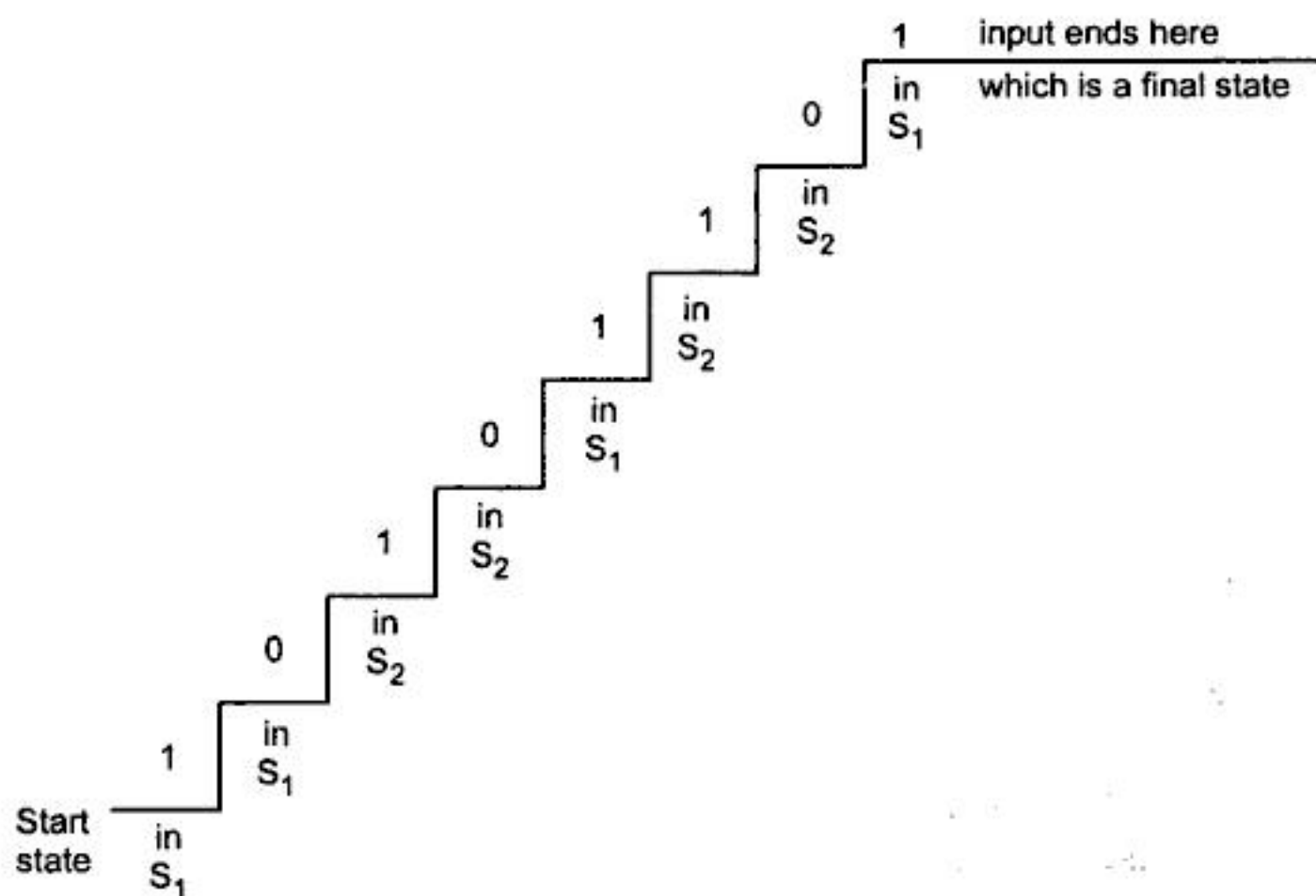


Fig. 2.14 Ladder diagram of processing the input

➡ **Example 2.6 :** Design a transition diagram for the language consisting of all the strings containing at least one pair of consecutive a's over the set {a,b}.

Solution :

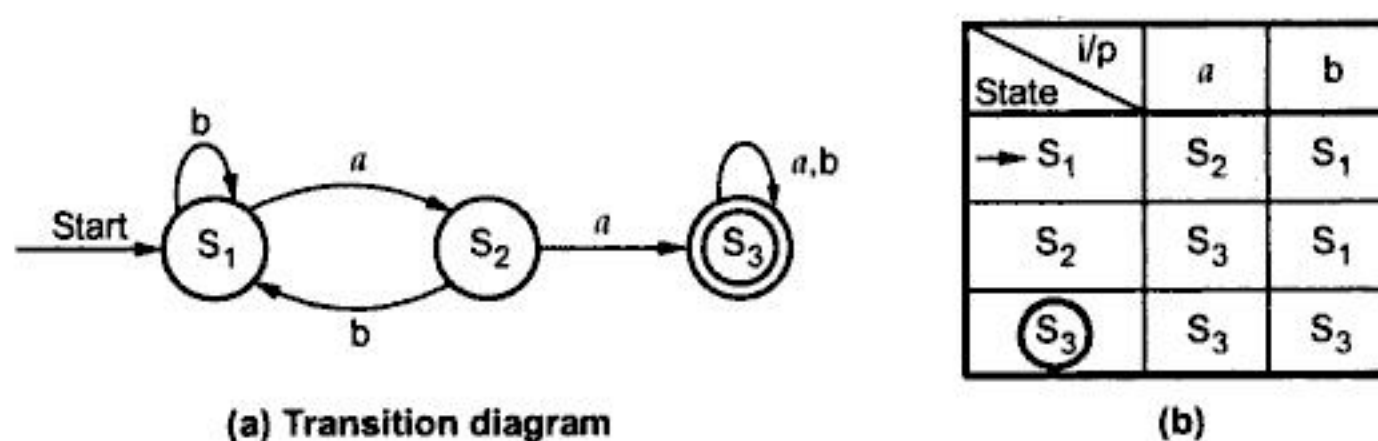


Fig. 2.15

➡ **Example 2.7 :** Design a transition diagram for the language consisting of all the strings which accept exactly one 0 or one 1 over {0,1}.

Solution :

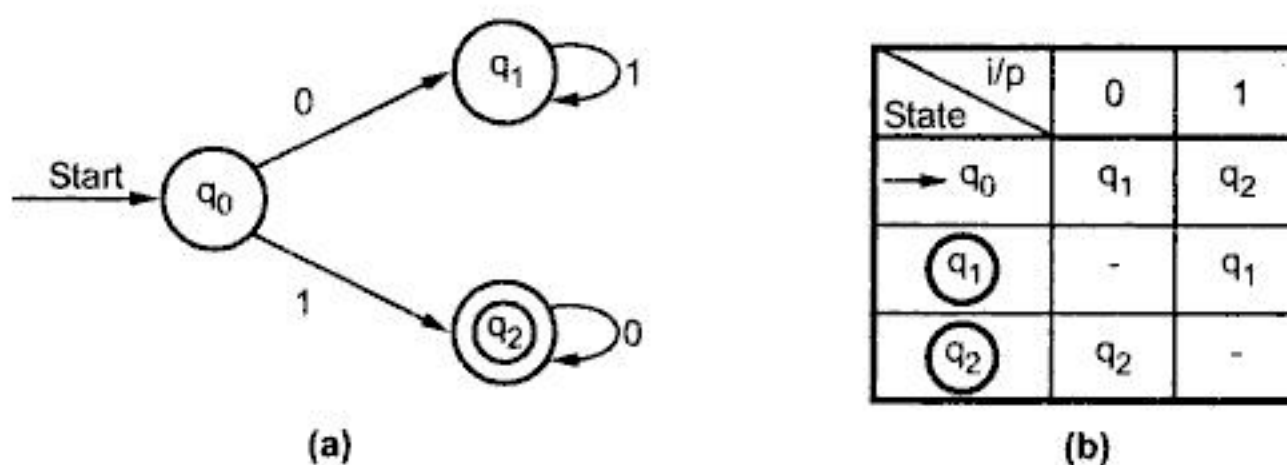


Fig. 2.16

We will now see some transition diagrams, which can be drawn for recognizing the tokens for a input language.

The regular expression for **identifier** will be

r.e. = letter (letter | digit)*

The regular expression for **constant** will be

r.e. = digit⁺ | (digit)⁺ (.) (digit)⁺ | (digit)⁺ (.) (digit)⁺ E(+ | -) (digit)⁺

The functions enter_id() and enter_num() in following transition diagrams are for entering identifier and constant values into symbol table. The token 'id' and 'num' indicate the corresponding entry in the symbol table is for identifier and constant, respectively.

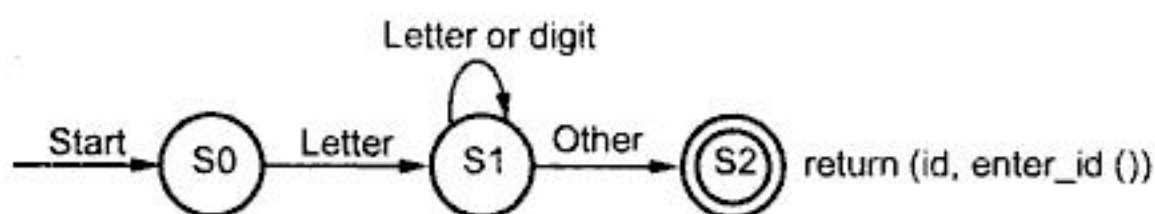


Fig. 2.17 Transition diagram for identifier

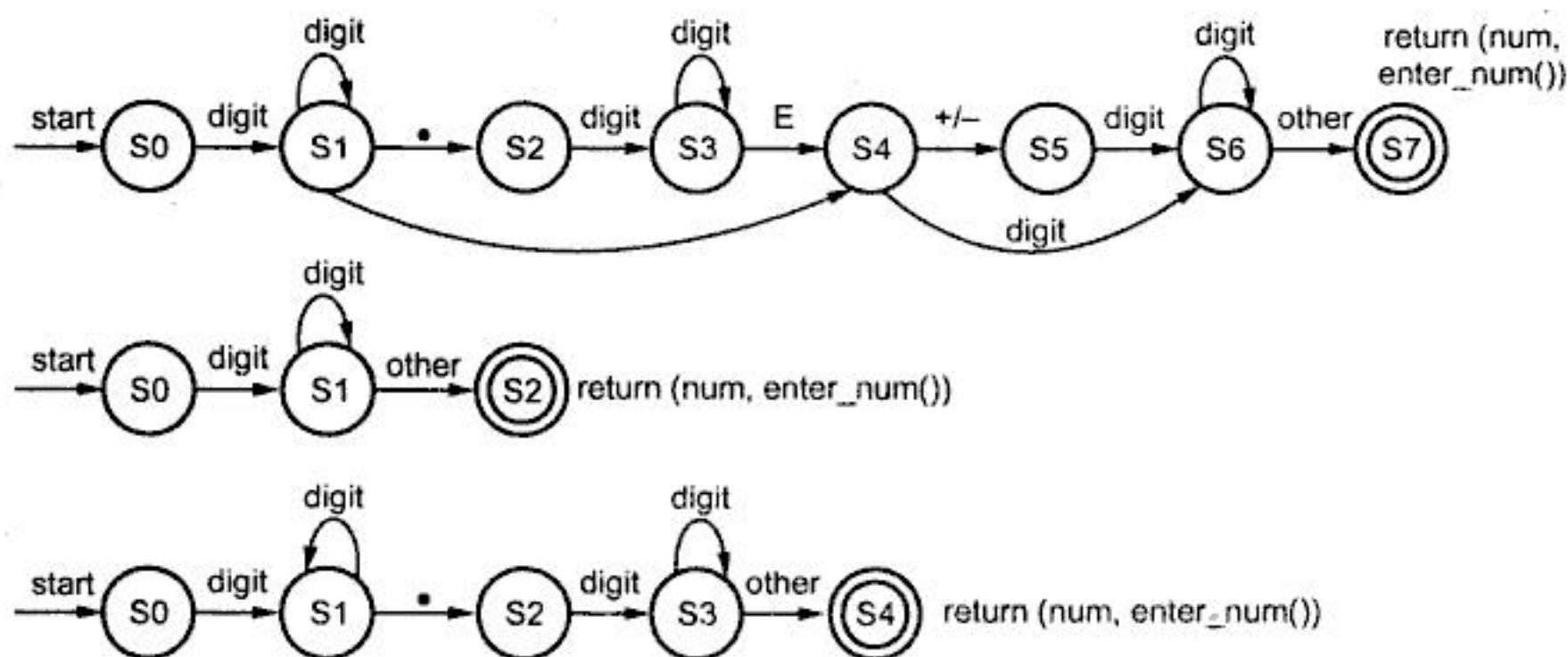


Fig. 2.18 Transition diagram for constant

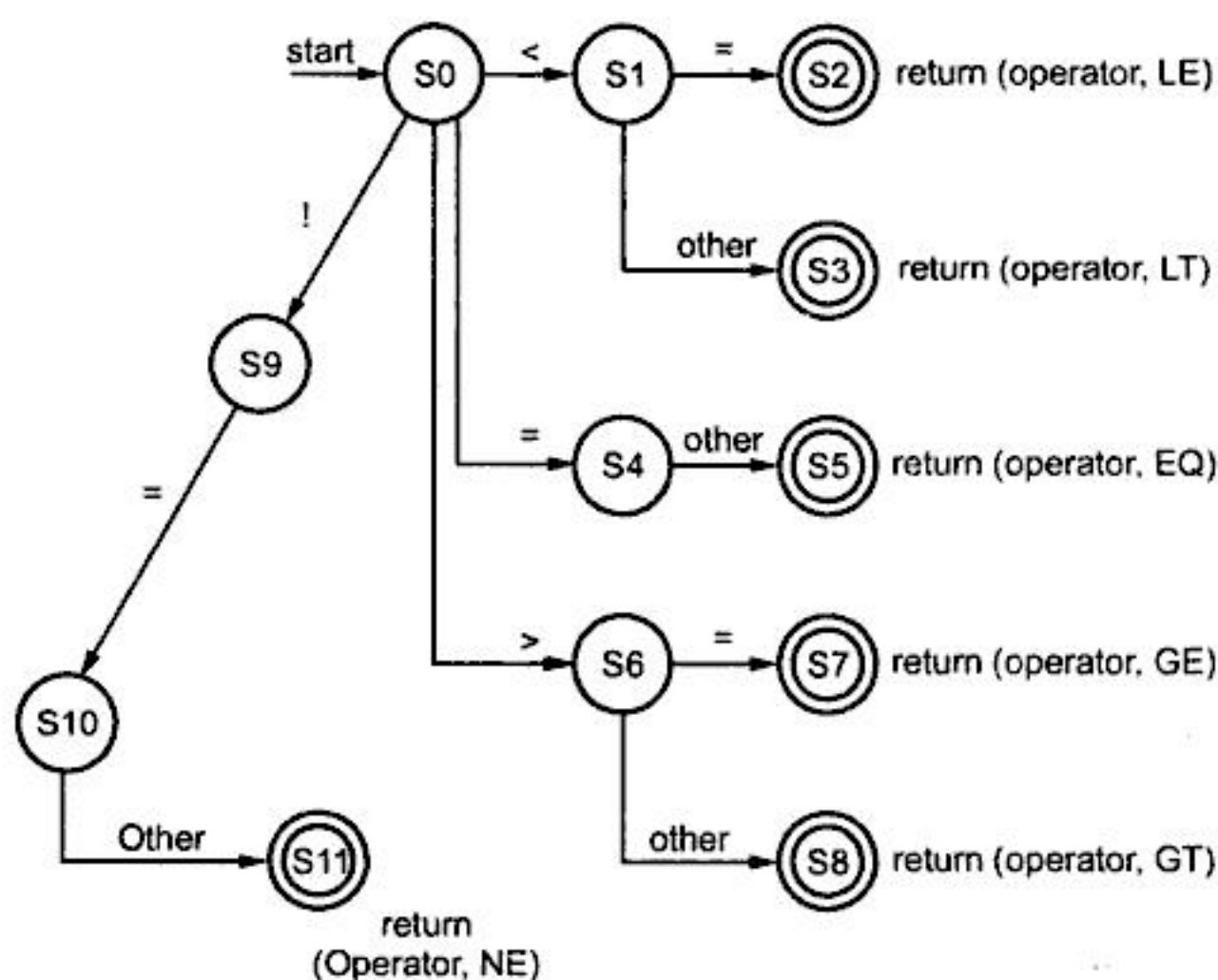


Fig. 2.19 Transition diagram for relational operator

2.7 Lex-Lexical Analyzer Generator

For efficient design of compiler, various tools have been built for constructing lexical analyzers using the special purpose notations called regular expressions.

The regular expressions are used in recognizing the tokens. Now we will discuss a special language that specifies the tokens using regular expressions. A tool called LEX gives this specification. Basically LEX is a Unix utility which generates the lexical analyzer. While discussing LEX we will learn how to write the specification file. In LEX tool, designing the regular expressions for corresponding tokens is a logical task.

Having a file name with extension.l (often pronounced as dot L) the lex specification file can be created. For example : the specification file name can be x.l . This x.l is then given to LEX compiler to produce lex.yy.c. This lex.yy.c is a C program which is actually a lexical analyzer program. As we know that the specification file stores the regular expressions for the tokens, the lex.yy.c file consists of the tabular representation of the transition diagrams constructed for the regular expressions of specification file say x.l. The lexemes can be recognized with help of tabular transitions diagrams and some standard routines. In the specification file of LEX actions are associated with each regular expression. These actions are simply pieces of C code. These pieces of C code are directly carried over to the lex.yy.c. Finally the C compiler compiles this generated lex.yy.c and produces an object program a.out. When some input stream is given to a.out then sequence of tokens get generated. The above described scenario is modeled below –

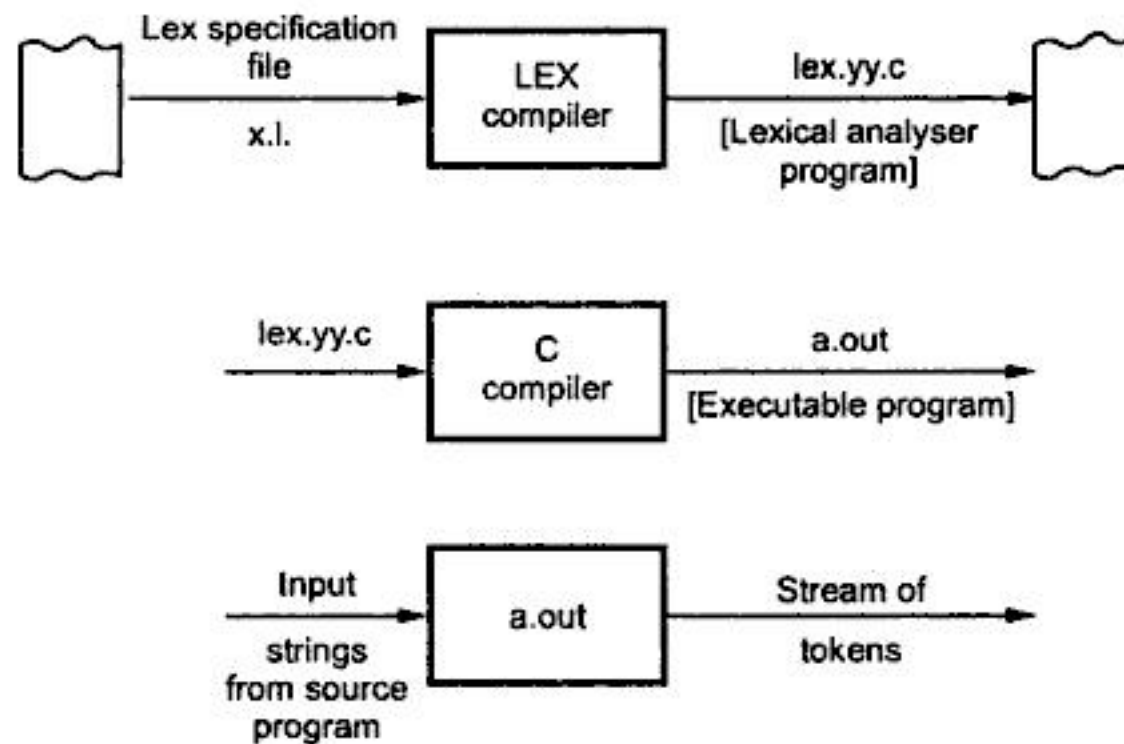


Fig. 2.20 Generation of lexical analyser using LEX

Now the question arises how do we write the specification file? Well, the LEX program consists of three parts - : 1. Declaration section 2. Rule section and 3. Procedure section

```

%{
Declaration section
%}
%%
Rule section
%%
Auxiliary procedure section
  
```

- In the declaration section declaration of variables constants can be done. Some regular definitions can also be written in this section. The regular definitions are basically components of regular expressions appearing in the rule section.
- The rule section consists of regular expressions with associated actions. These translation rules can be given in the form as -

```

R1 {action1}
R2 {action2}
.
.
.
Rn {actionn}
  
```

Where each R_i is a regular expression and each $action_i$ is a program fragment describing what action is to be taken for corresponding regular expression. These actions can be specified by piece of C code.

- And third section is a auxiliary procedure section in which all the required procedures are defined. Sometimes these procedures are required by the actions in the rule section.
- The lexical analyzer or scanner works in co-ordination with parser. When activated by the parser, the lexical analyzer begins reading its remaining input, character by character at a time. When the string is matched with one of the regular expressions R_i then corresponding action_i will get executed and this action_i returns the control to the parser.
- The repeated search for the lexeme can be made in order to return all the tokens in the source string. The lexical analyzer ignores white spaces and comments in this process. Let us learn some LEX programming with the help of some examples –

```
%{  
%}  
%%  
"Rama" |  
"Seeta" |  
"Geeta" |  
"Neeta" |      printf("\n Noun");  
"Sings" |  
"dances" |  
"eats"         printf ("\n Verb");  
%%  
main ()  
{  
    yylex();  
}  
int yywrap()  
{  
    return 1;  
}
```

This is a simple program that recognizes noun and verb from the string clearly. There are 3 sections in above program.

- The section starting and ending with % { and %} respectively is a definition section.
- The section starting with %% is called rule section. This section is closed by %% within %% consists of regular expression and actions. Rule 1 gives the definition of noun and second rule gives the definition of verb.

- The third section consists of two functions the main function and the yywrap function. In main function call to yylex routine is given. This function is defined in lex.yy.c program. First we will compile our above program (x.l) using lex compiler. And then the lex compiler will generate a ready C program named lex.yy.c. This lex.yy.c makes use of regular expression and corresponding actions defined in x.l. Hence our above program x.l is called lex specification file.

When we compile lex.yy.c file we get a output file named a.out. (This is a default output file on LINUX platform). On execution of a.out we can give the input string.

Following commands are used to run the lex program x.l.

```
$ lex x.l
$ cc lex.yy.c
$ ./a.out
```

After entering these commands a blank space for entering input gets available. There we can give some valid input.

```
Rama eats
Noun
verb
Seeta sings
Noun
verb
```

Then press either control + c or control d to come out of the output.

vi Editor commands

While writing LEX program on LINUX popularly a vi editor is used. Here are some commonly used vi commands.

- To start vi editor : At the command prompt we can give following command

```
$ vi filename
```

For example \$ vi x.l

Then file with x.l will be created and opened in vi editor.

There are two modes in which vi editor works. The insert mode and command mode. To write something in the file we should make insert mode on. For that we have to give

```
ESC i
```

Now we can type the desired text in the file.

To exit vi and save changes -

```
ESC :zz
```

or ESC : wq.

To exit vi without saving changes

ESC : q!

Following commands are used to handle the text in the file.

Command with ESC	Purpose
r	replaces character under cursor with next character.
R	Keep replacing character until ESC is pressed.
a	append after cursor.
A	append at end of line.
x	deletes character under cursor.
dd	deletes line under cursor.
dw	deletes word under cursor.
db	deletes word before cursor.
yy	copies line which may be put by 'p' command.
P	puts back before cursor.
p	puts back after cursor.

- To move at particular line number then give following command

ESC : #

Here # is the desired line number.

➡ **Example 2.8 :** Write a LEX program for recognizing the tokens such as identifiers, keywords. [Here we have given program name as lexprog. l]

Solution :

```
%{
%}
identifier[a-zA-Z][a-zA-Z0-9]*
%%

/*For recognizing preprocessor directives*/
#.* {printf("\n%s is a PREPROCESSOR DIRECTIVE", yytext);}

/*For recognizing keywords*/
```



```

int|
float|
char|
double|
while|
for|
do|
break|
continue|
void|
switch|
case|
long|
struct|
const|
typedef|
return|
else|
goto                                {printf("n\t%s is a KEYWORD",yytext);}
if[\\t]*\\(                          {printf("n\tif is a KEYWORD\\n\\t(");}
    /*For recognizing single line comments*/
    \\\\*.*\\*\\|
    \\\\..*                          {printf("n\\n\\t%s is a COMMENT\\n", yytext);}
    /*Function call or definition*/
    {identifier}\\((printf("n\\n FUNCTION CALL/DEFINITION
    \\n\\t%S",yytext);}
    /*Block start and end*/
    \\{                               {printf("n BLOCK BEGINS");}
    \\}                               {printf("n BLOCK ENDS");}
    /*For variables*/
    {identifier}(\\[[0-9]*\\))*{printf("n\\t% is an IDENTIFIER",yytext);}
    /*For a string*/
    \".*\"                            {printf("n\\t%s is a STRING",yytext);}

```

```

    /*For recognizing constants */
[0-9]+          {printf("\n\t%s is a NUMBER",yytext);}
\)(\;)?         {printf("\n\t");ECHO;printf("\n");}
\((            ECHO;

    /*For operators*/
=                {printf("\n\t%s is an ASSIGNMENT
                    OPERATOR",yytext);}

\<=|
\>=|
\<|
==|
\>              {printf("\n\t%s is a RELATIONAL
                    OPERATOR",yytext);}

.|\\n           ;

    /*End of Rules section*/
%%
int main(int argc, char**argv)
{
    if(argc>1)
    {
        FILE *file;
        file=fopen(argv[1],"r");
        If(!file)
        {
            printf("Could not open %s\n", argv[1]);
            exit(0);
        }
        yyin = file;
    }
    yylex();
    printf("\n\n");
    return 0;
}

```


Area.C

```
#include<stdio.h>
#include<stdio.h>
double area_of_circle(double r);
int main(int argc,char*argv[])
{
if(argc<2) /*This is a Single line comment */
{
printf("Usage:%s radius\n", argv[0]);
exit(1);
}
else
{
double radius = atof(argv[1]);
double area = area_of_circle(radius);
printf("Area of circle with radius %f = %f\n",radius,area);
}
return 0;//Returning 0
}
```

The program Area.C can be taken as input test program. This program can be scanned by generated lex.yy.C (We have now generated lex.yy.C using LEX tool). Hence the output obtained using following commands.

```
[root@localhost]# lex lexprog.l
[root@localhost]# cc lex.yy.c
[root@localhost]# ./a.out Area.c
```

2.8 LEX Specification and Features

To generate the lexical analyzer the automation tool like LEX is the efficient way. The specification file for LEX is based on the pattern-action structure as shown below :

```
R1 {action1}
R2 {action2}
.
.
.
Rn {actionn}
```

Where each R_i is a regular expression and each $action_i$ is a program fragment describing what action is to be taken for corresponding regular expression. Each action is a piece of program code that is to be executed whenever lexeme is matched by R_i .

Regular expression	Meaning
*	Matches with zero or more occurrences of preceding expression. For example 1^* occurrence of 1 for any number of times.
.	Matches any single character other than new line character.
[]	A character class which matches any character within the bracket. For example $[a-z]$ matches with any alphabet in lower case.
()	Group of regular expressions together put into a new regular expression.
" "	The string written in quotes matches literally. For example "Hanumaan" matches with the string Hanumaan.
\$	Matches with the end of line as last character.
+	Matches with one or more occurrences of preceding expression. For example $[0-9]^+$ any number but not empty string.
?	Matches zero or one occurrence of preceding regular expression. For example $[+]?[0-9]^+$ a number with unary operator.
^	Matches the beginning of a line as first character.
[^]	Used as for negation. For example $[^verb]$ means except verb match with anything else.
\	Used as escape metacharacter. For example $\backslash n$ is a newline character. $\backslash \#$ prints the # literally
	To represent the or i.e. another alternative. For example $a b$ means match with either a or b.

LEX Actions

There are various LEX actions that can be used for ease of programming using LEX tool.

1. **BEGIN** - It indicates the start state. The lexical analyzer starts at state 0.
2. **ECHO** - It emits the input as it is.
3. **yytext** - When lexer matches or recognizes the token from input token then the lexeme is stored in a null terminated string called yytext.

As soon as new token is found the contents of `yytext` are replaced by new token.

4. `yylex()` - This is an important function. As soon as call to `yylex()` is encountered scanner starts scanning the source program.
5. `yywrap()` - The function `yywrap()` is called when scanner encounters end of file. If `yywrap()` returns 0 then scanner continues scanning. When `yywrap()` returns 1 that means end of file is encountered.
6. `yyin` - It is the standard input file that stores input source program.
7. `yylen` - When a lexer recognizes token then the lexeme is stored in a null terminated string called `yytext`. And `yylen` stores the length or number of characters in the input string. The value in `yylen` is same as `strlen()` functions.
8. How to write `main()` in LEX ?

The `main ()` can be written as

```
void main
{
    yylex();
}
```

or it can be written with command line parameters such as `argc` and `argv`. If command line parameters are passed to the `main` then while executing the program we give input source file name.

For example - if the source program which is to be scanned is `x.c` then

```
#!/a.out x.c
```

will scan the input source `x.c`.

- 9) Where to write C code -

We can write a valid 'C' code between

```
%{ %}
```

we can write any C function in a subroutine section.

We can write valid C code in the action part for corresponding regular expression.

We have to construct recognizer that looks for the lexemes stored in the input buffer. It works using these two rules -

1. If more than one pattern matches then recognizer has to choose the longest lexeme matched.
2. If there are two or more patterns that match the longest lexeme, the first listed matching pattern is chosen.

Lexical analysis is essentially a process of recognizing different tokens from the source program. This process of recognition can be accomplished by building a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

2.9 Design of Lexical Analyzer Generator

To design the lexical analyzer generator first design the patterns of regular expressions for the tokens and then from these patterns it is easy to design a non-deterministic finite automata. But it is always easy to simulate the behavior of a DFA with a program ; hence we have to convert the NFA drawn from the patterns to DFA.

The design of lexical analyzer generator is based on two approaches –

- Pattern matching using NFA.
- Using DFA for lexical analyzer.

Let us discuss these approaches one by one.

2.9.1 Pattern Matching based on NFA's

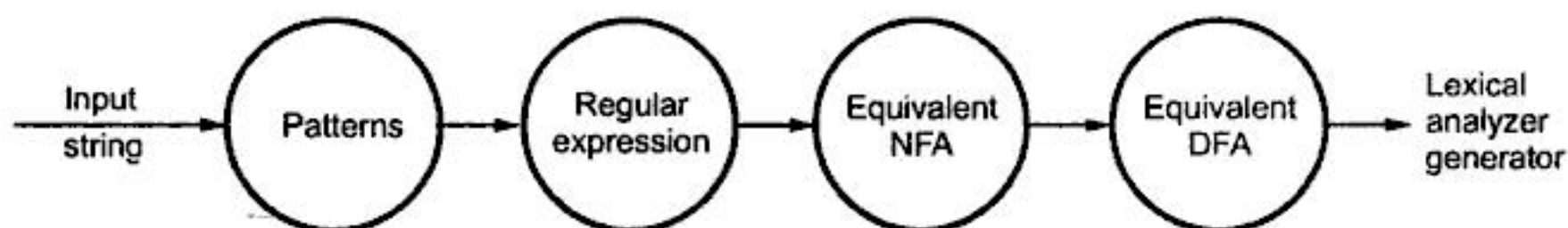


Fig. 2.23 Building Lexical analyzer generator

To design Lexical analyzer generator, the pattern of regular expressions are designed first. These patterns are for recognizing various tokens from input string. From these patterns it is easy to design a non deterministic finite automata. But it is always easy to simulate the behavior of DFA with a program. Hence we convert the NFA drawn from these patterns to DFA.

- In this method the transition table for each NFA can be drawn and each NFA is designed based on the pattern mentioned in the specification file of LEX. Hence there can be n number of NFA's for n number of patterns. A start state is taken and with transitions all these NFAs can be connected together as shown below :

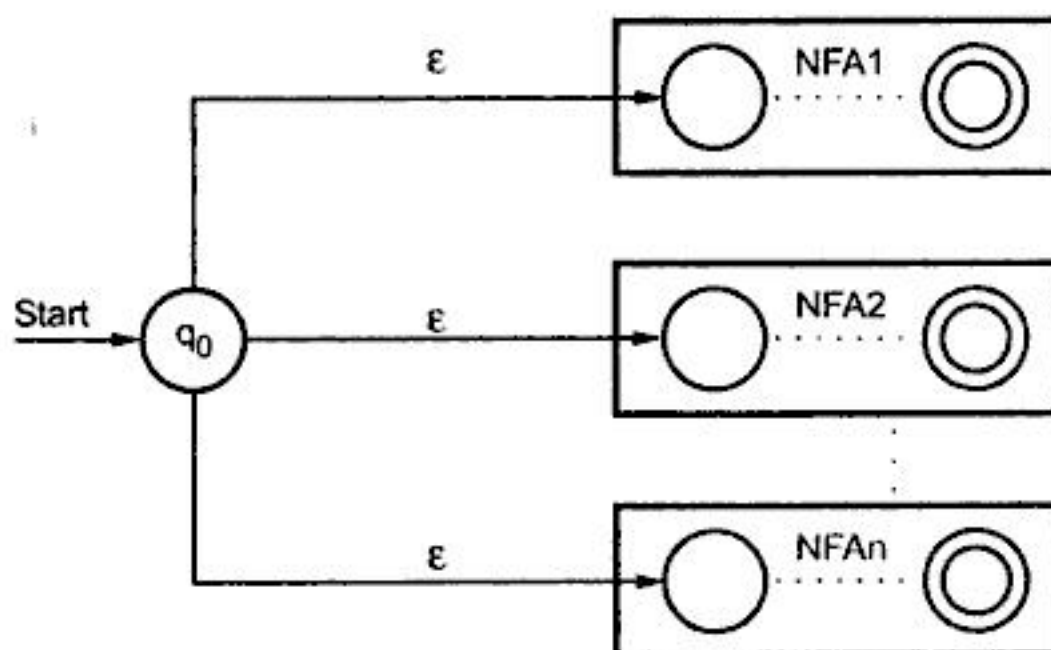


Fig. 2.24 Composite NFA

- This combined NFA recognizes the longest prefix of the input matched by the pattern of regular expression R_i . While simulating NFA even though we get accepting states we should proceed until it reaches the termination.
- For a pattern matching process there could be **multiple accept states** in the current set of states. We simply record the current input position and the pattern for corresponding regular expression R_i and we continue making transitions until termination is found.
- As soon as we reach to the termination, we retract the forward pointer to position at the last match occurred. Thus this pattern matching process **identifies the token** from the input string.

For example

0	{ }	Pattern 1
011	{ }	Pattern 2
0 ⁺ 1 ⁺	{ }	Pattern 3

Now to identify the string 0011 we begin with initial set of states. The first pattern is matched with the first symbol of the input 0011 but as in the next transition we reach to the accept state we simply record the state and current input pointer. Now we choose the second pattern and move on the transition diagram of 011 but the second symbol of the input does not match with pattern 011. Hence we record the set of states and try for the third pattern we proceed for the transition diagram 0⁺1⁺ when the input terminates we reach to the accept state. The token can be identified from the source input string.

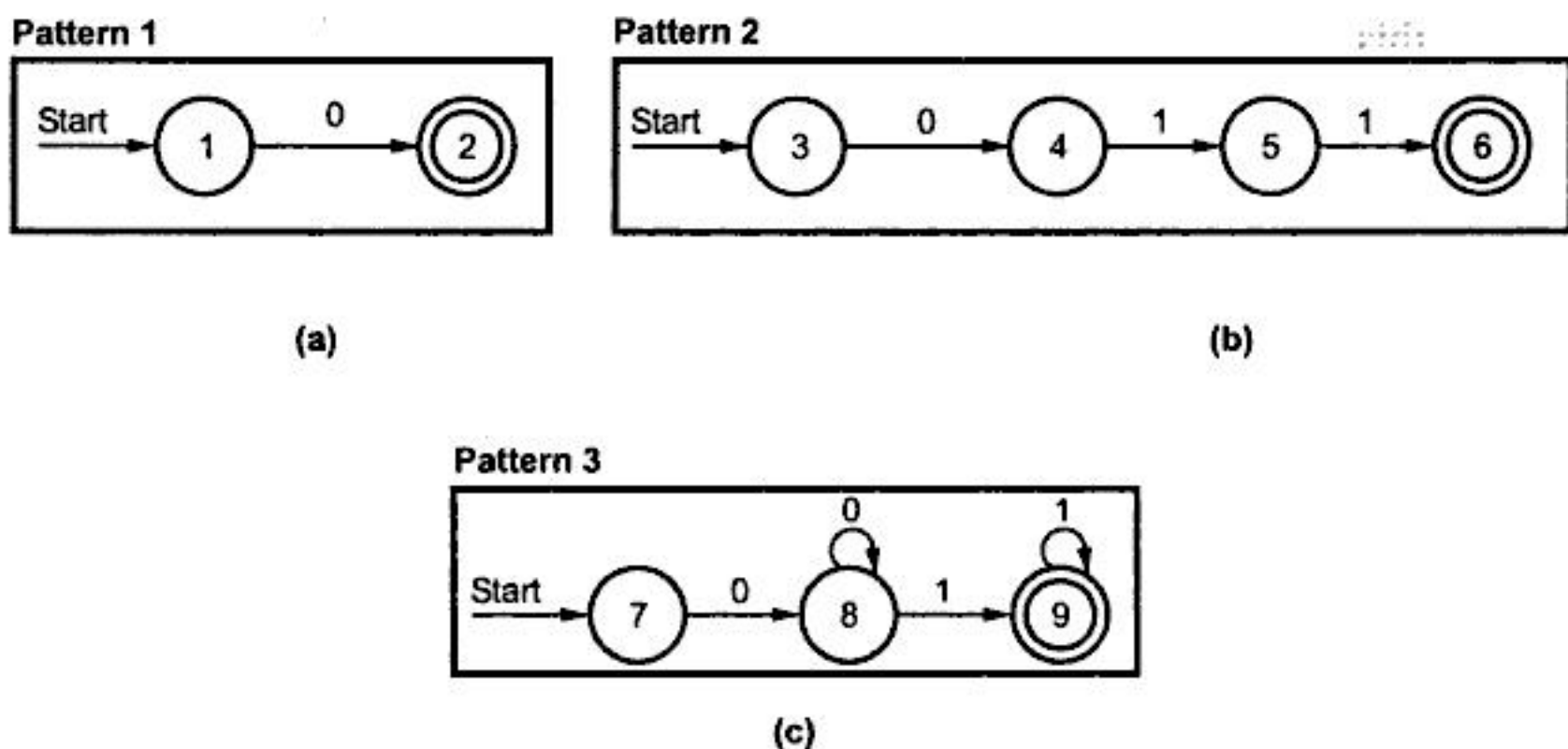


Fig. 2.25

The composite NFA for given pattern is

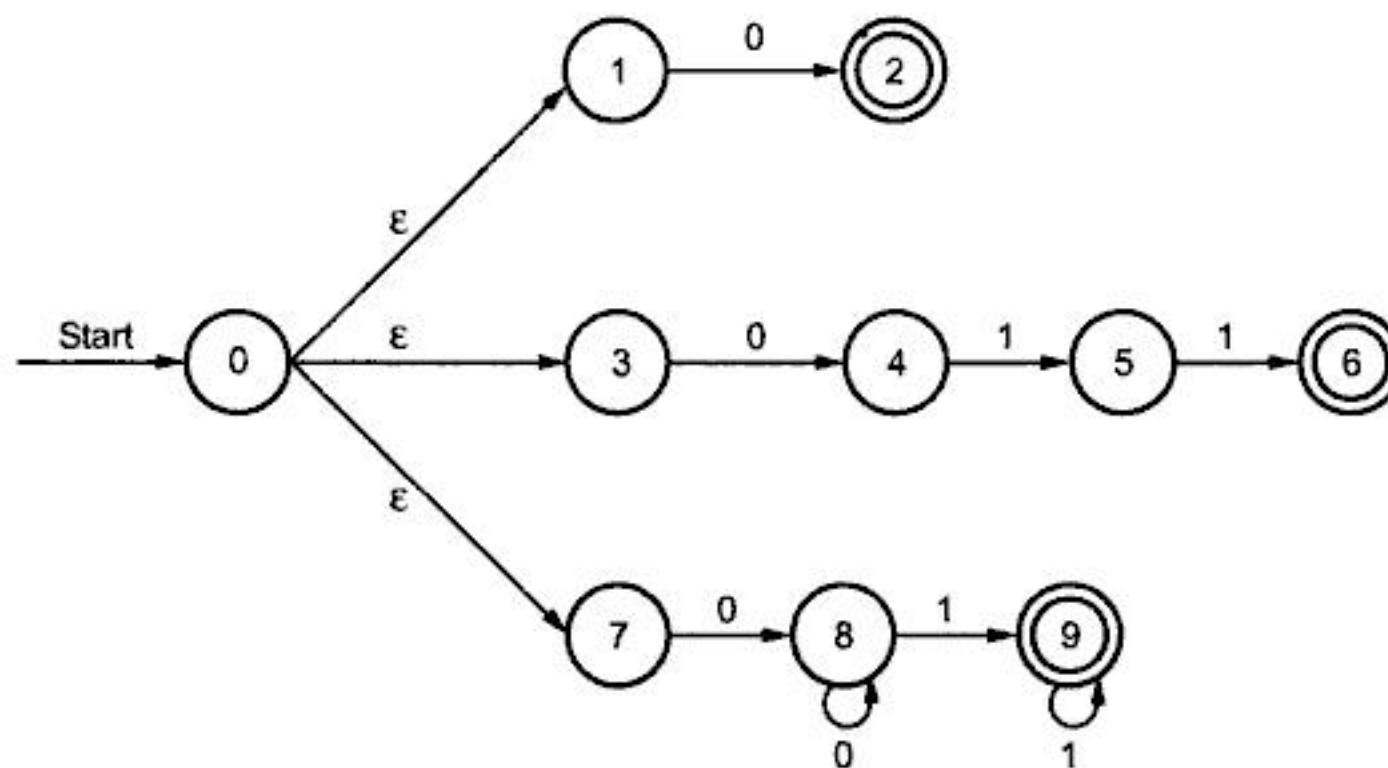


Fig. 2.26

To summarize this one can say that LEX tries to match the input string with the help of patterns one by one and it tries to match the **longest lexeme**.

Once composite NFA is built this NFA has to be converted to DFA

- In this method the DFA is constructed from the NFAs drawn from patterns of regular expressions. Then we try to match the input string with each of the DFA. The accepting states are recorded and we go on moving on the transitions until the input string terminates.
- If there are two patterns matching with the input string then the pattern which is mentioned first in the LEX specification file is always considered. For example consider the LEX specification file it consists of

0	{ }
011	{ }
0 ⁺ 1 ⁺	{ }

Now we have already built a NFA in earlier discussion for the above pattern. Let us design the DFA from NFA.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

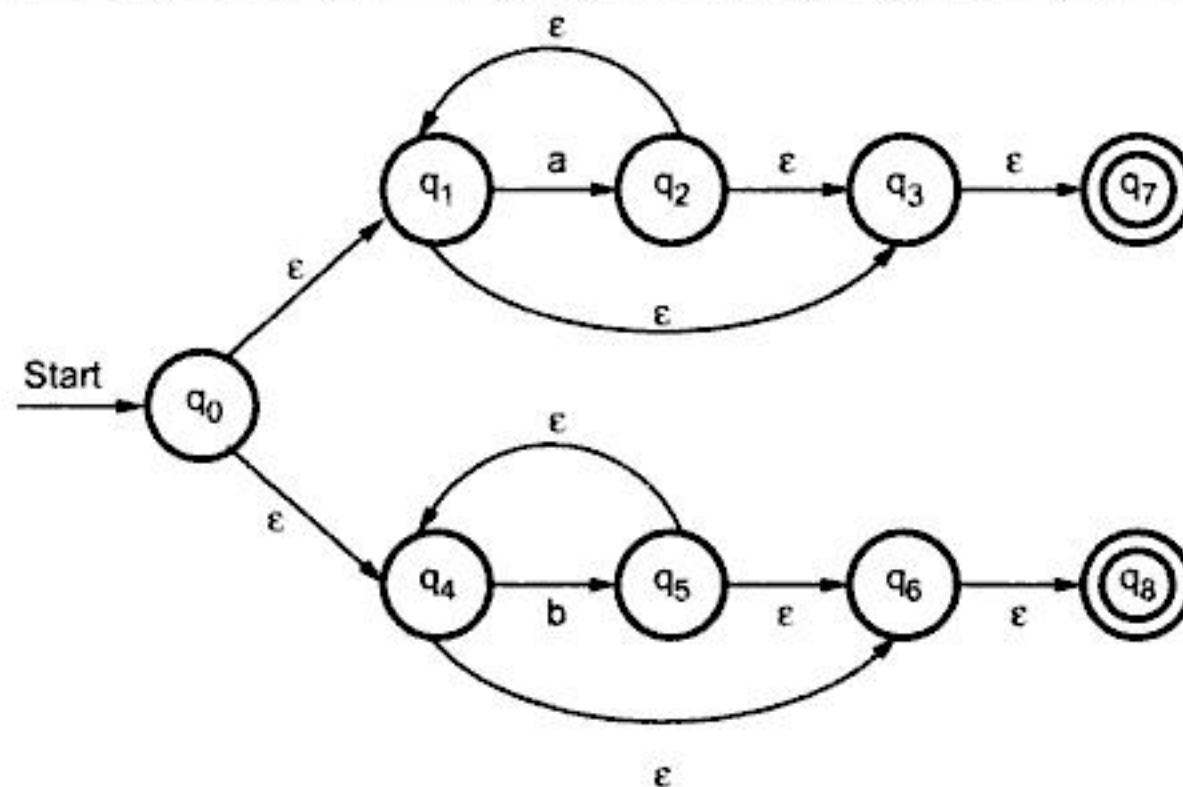


Fig. 2.28

Q.2 : What are the issues of the lexical analyzer?

Ans. : Various issues in lexical analysis are

- 1) The lexical analysis and syntax analysis are separated out for simplifying the task of one or other phases. This separation reduces the burden on parsing phase.
- 2) Compiler efficiency gets increased if the lexical analyzer is separated. This is because a large amount of time is spent on reading the source file. And it would be inefficient if each phase has to read the source file. Lexical analyzer uses buffering techniques for efficient scan of source file. The tokens obtained can be stored in input buffer to increase performance of compiler.
- 3) Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyzer. The representation of non standard symbol can be isolated in lexical analyzer. For example in Pascal, ↑ can be isolated in lexical analyzer.

Q.3 : What is the need for separating the analysis phase into lexical analysis and parsing ?

Ans. : Following are the reasons for separating the analysis phase into separate lexical analysis and parsing -

- i) Separation of lexical analysis from syntax analysis allows the simplified design. For example handling white spaces and comments in parsing is more complicated. Instead, if white spaces and comments have already been removed from lexical analyzer then it can lead to cleaner design of analysis phase.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Q.9 : Describe precisely and concisely the language denoted by following regular expressions.

- i) $((a+b)(a+b)^*)^*aa(a+b)^*$
- ii) $(a+b)^*a(a+b)(a+b)(a+b)$
- iii) $(aa+bb)^*((ab+ba)(aa+bb)^*(ab+ba)(aa+bb)^*)^*$
- iv) $(b+ab)^*(a+b)^*$

Ans. :

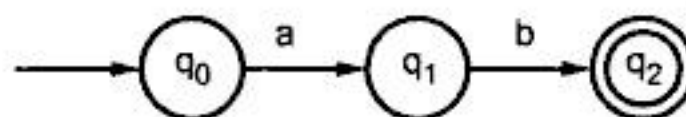
- i) This is a language containing all the strings containing a substring "aa".
- ii) This is a language containing all the strings containing 'a' as forth symbol from right end.
- iii) This is a language in which all the strings contains even number a's and even number of b's.
- iv) The language in which strings do not contain "aa" as a substring.

Q.10 : Give DFAS accepting the following regular languages over $\Sigma = \{a, b\}$.

- i) All strings which contain the string ab as a substring.
- ii) All the strings that do not contain ab as a substring.

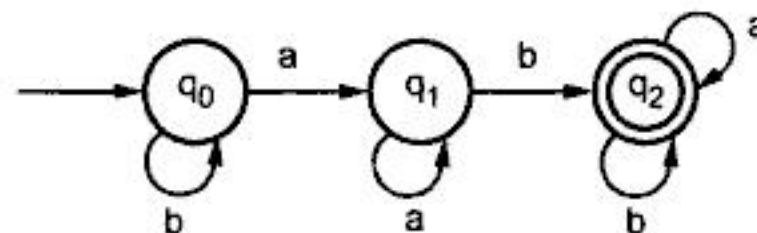
Ans. : i) The string "ab" should be the substring.

Step 1 :



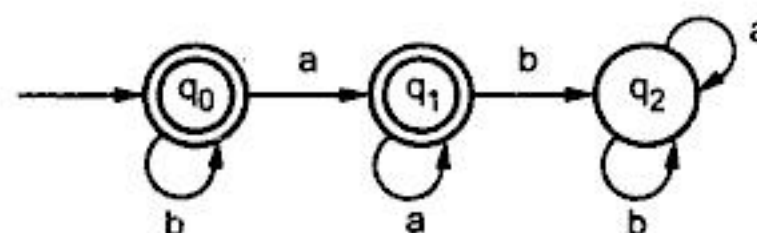
Then we can we can draw remaining possible transitions at q_0 , q_1 and q_2 states.

Step 2 :



is the required DFA.

- ii) In this language there should not be "ab" as a substring. So we will simply compliment DFA obtained in answer (i) [That is make non final states as final and final states as non-final.]





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

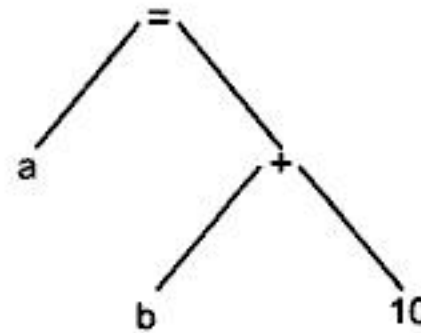


Fig. 3.1 Parse Tree for $a = b + 10$

The parse tree drawn above is for some programming statement. It shows how the statement gets parsed according to their syntactic specification.

3.2.1 Basic Issues in Parsing

- There are two important issues in parsing :
 - i) Specification of syntax
 - ii) Representation of input after parsing.
- A very important issue in parsing is **specification of syntax** in programming language. Specification of syntax means how to write any programming statement. There are certain characteristics of specification of syntax –
 - i) This specification should be **precise and unambiguous**.
 - ii) This specification should be in **detail**, i.e. it should cover all the details of the programming language.
 - iii) This specification should be **complete**.

Such a specification is called “**Context Free Grammar**” (CFG).

- Another important issue in parsing is **representation of the input after parsing**. This is important because all the subsequent phases of compiler take the information from the parse tree being generated. This is important because the information suggested by any input programming statement should not be differed after building the syntax tree for it.
- Lastly the most crucial issue is the parsing algorithm based on which we get the parse tree for the given input. There are two different **approaches** to parsing: **Top down** and **bottom up**. The parsing algorithms are based on these approaches. These algorithms deal with following issues -
 - o How these algorithms work?
 - o Are they efficient in nature?
 - o What are their merits and limitations?
 - o What kind of input they require?



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Instead of choosing the arbitrary non terminal one can choose.

- i) Either leftmost non terminal in a sentential form then it is called leftmost derivation.
- ii) Or rightmost non terminal in a sentential form, then it is called rightmost derivation.

➡ **Example 3.1 :** Let G be a context free grammar for which the production rules are given as below -

$$\begin{aligned} S &\rightarrow aB \mid bA \\ A &\rightarrow a \mid aS \mid bAA \\ B &\rightarrow b \mid bS \mid aBB \end{aligned}$$

Derive the string $aaabbabbba$ using above grammar.

Solution : Leftmost derivation

$$\begin{aligned} S &\rightarrow aB \\ &\quad aaBB \\ &\quad aaaBBB \\ &\quad aaabSBB \\ &\quad aaabbABB \\ &\quad aaabbaBB \\ &\quad aaabbabB \\ &\quad aaabbabbS \\ &\quad aaabbabbbA \\ &\quad aaabbabbba \end{aligned}$$

Rightmost Derivation

$$\begin{aligned} S &\rightarrow aB \\ &\quad aaBB \\ &\quad aaBbS \\ &\quad aaBbbA \\ &\quad aaBbba \\ &\quad aaaBBba \\ &\quad aaaBbbba \\ &\quad aaabSbbba \\ &\quad aaabbAbbba \\ &\quad aaabbabbba \end{aligned}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



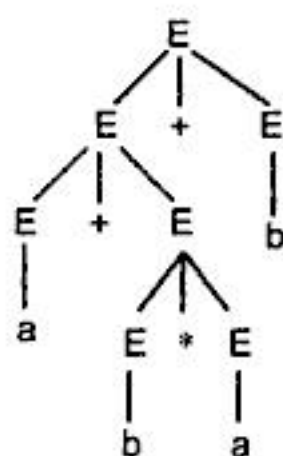
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

➡ **Example 3.3 :** Consider the grammar given below -

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid a \mid b$

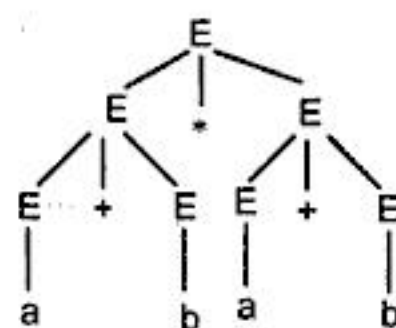
Obtain leftmost and rightmost derivation for the string $a + b * a + b$.

Solution : Leftmost derivation



E
 $E + E$
 $E + E + E$
 $a + E + E$
 $a + E * E + E$
 $a + b * E + E$
 $a + b * a + E$
 $a + b * a + b$

Rightmost derivation



E
 $E * E$
 $E * E + E$
 $E * E + b$
 $E * a + b$
 $E + E * a + b$
 $E + b * a + b$
 $a + b * a + b$

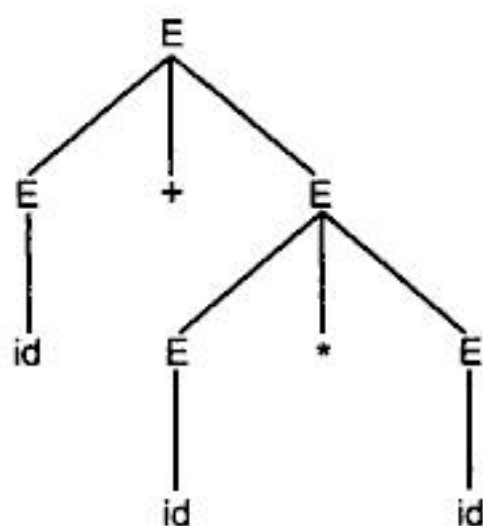
3.3.2 Ambiguous Grammar

A grammar G is said to be ambiguous if it generates more than one parse trees for sentence of language $L(G)$.

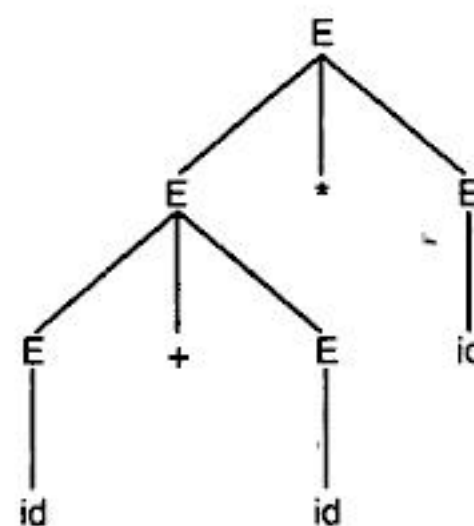
For example :

$E \rightarrow E + E \mid E * E \mid id$

Then for $id + id * id$



(a) Parse tree 1



(b) Parse tree 2

Fig. 3.7 Ambiguous grammar



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Step 3:

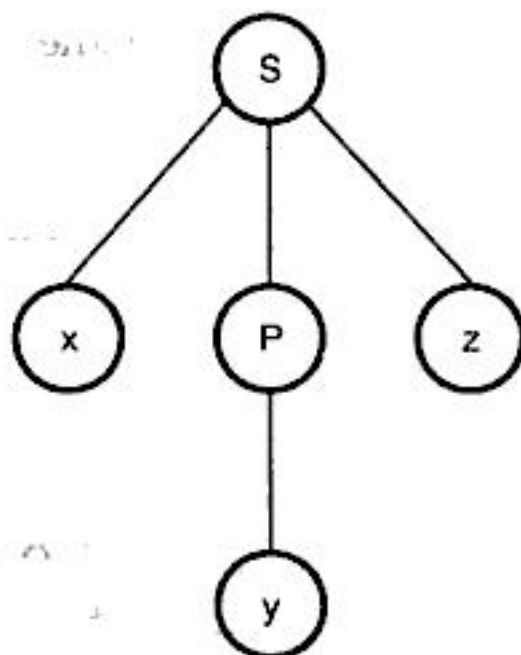


Fig. 3.11 (c)

We halt and declare that the parsing is completed successfully.

In top down parsing selection of proper rule is very important task. And this selection is based on trial and error technique. That means we have to select a particular rule and if it is not producing the correct input string then we need to backtrack and then we have to try another production. This process has to be repeated until we get the correct input string. After trying all the

productions if we found every production unsuitable for the string match then in that case the parse tree can not be built.

3.5.1 Problems with Top down Parsing

There are certain problems in top down parsing. In order to implement the parsing we need to eliminate these problems. Let us discuss these problems and how to remove them.

1) Backtracking

Backtracking is a technique in which for expansion of non terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

For example :

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

Then

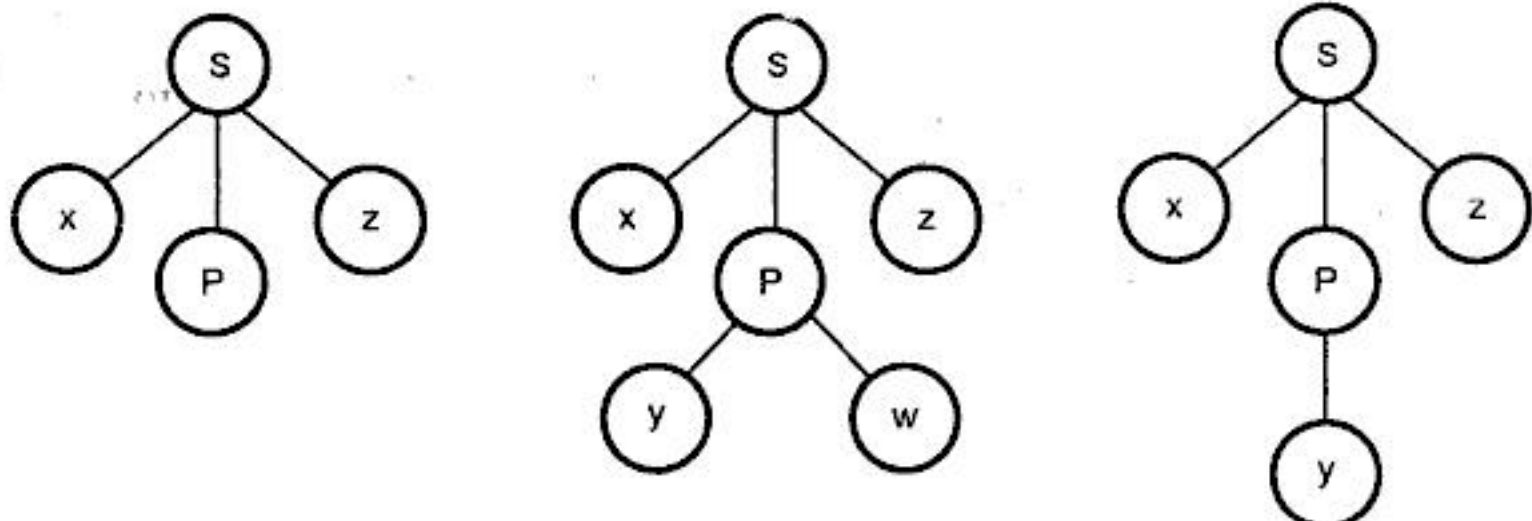


Fig. 3.12



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

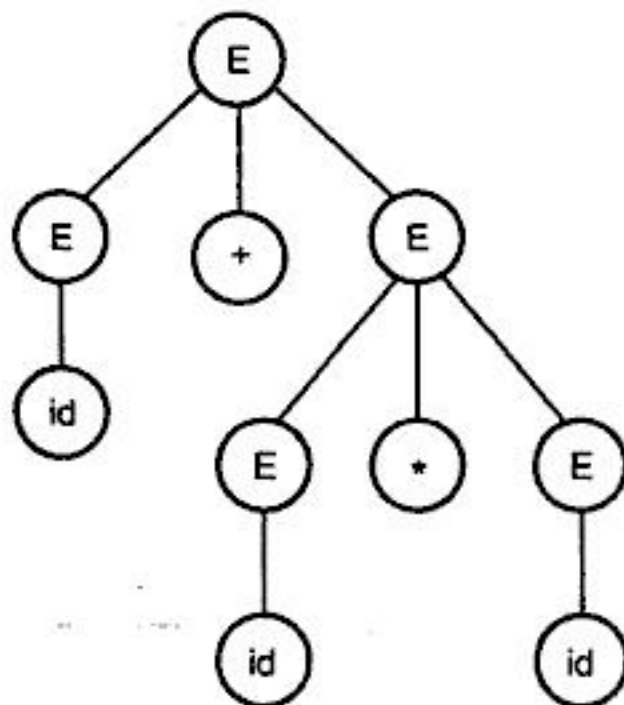
4) Ambiguity

The ambiguous grammar is not desirable in top down parsing. Hence we need to remove the ambiguity from the grammar if it is present.

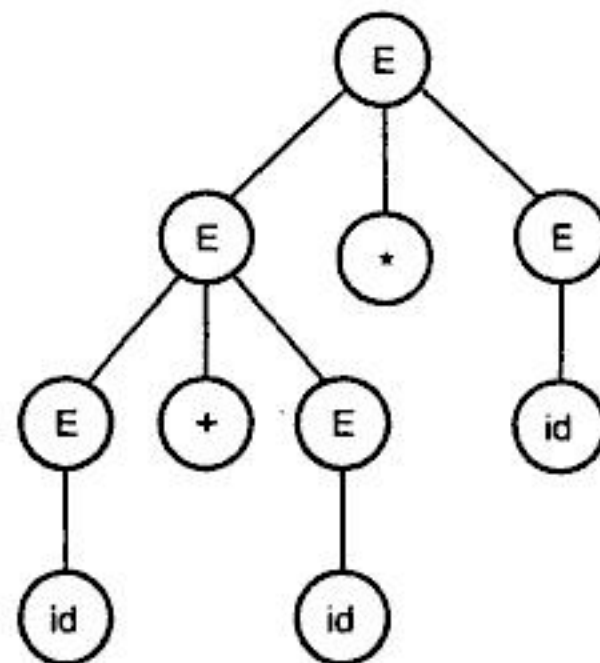
For example :

$E \rightarrow E + E \mid E * E \mid id$

is an ambiguous grammar. We will design the parse tree for $id+id*id$ as follows.



(a) Parse tree 1



(b) Parse tree 2

Fig. 3.15 Ambiguous grammar

For removing the ambiguity we will apply one rule : if the grammar has left associative operator (such as $+$, $-$, $*$, $/$) then induce the left recursion and if the grammar has right associative operator (exponential operator) then induce the right recursion.

The unambiguous grammar is

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow id$

Note one thing that the grammar is unambiguous but it is left recursive and elimination of such left recursion is again a must.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

As lookahead pointer points to \$ we quit by reporting success. Thus the input string can be parsed using recursive descent parser.

Construction of recursive descent parser is easy. However the programming language that we choose for RD parser should support recursion. The internal details are not accessible in this type of parser. For instance; in this parser we cannot access the current leftmost sentential form. Secondly at any instant, we can not access the stack containing recursive calls.

Writing procedures for left recursive grammar ($A \rightarrow \alpha\beta_1 | \alpha\beta_2$) is crucial. And to make such procedure simpler first we have to left factor the grammar. We cannot write recursive descent parsers for all types context free grammar.

3.7 Predictive LL(1) Parser

This top down parsing algorithm is of non recursive type. In this type of parsing a table is built. For LL(1) - the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for input string. And the number 1 in the input symbol means it uses only one input symbol (lookahead) to predict the parsing process.

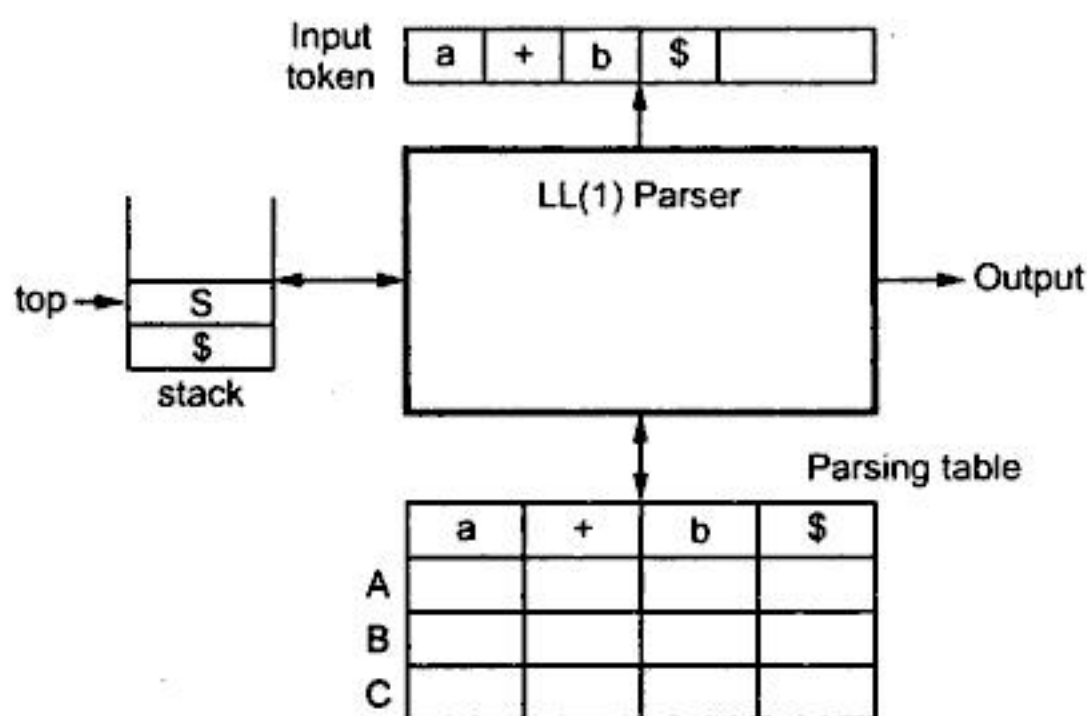


Fig. 3.19 Model for LL(1) Parser

The simple block diagram for LL(1) parser is as given below.

The data structures used by LL(1) are i) input buffer ii) stack iii) parsing table. The LL(1) parser uses input buffer to store the input tokens. The stack is used to hold the **left sentential form**. The symbols in RHS of rule are pushed into the stack in reverse order i.e. from right to left. Thus use of stack makes this algorithm non recursive. The table is basically a two dimensional array. The table has row for non terminal and column for terminals. The table can be represented as $M[A, a]$ where A is a non terminal and a is current input symbol. The parser works as follows –



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Consider $T' \rightarrow *FT'$ by computational rule 3

$$T' \rightarrow *FT'$$

$$A \rightarrow \alpha B \beta$$

$$A = T', \alpha = *, B = F, \beta = T'$$

$$\text{FOLLOW}(A) = \text{FOLLOW}(B)$$

$$\text{FOLLOW}(T') = \text{FOLLOW}(F)$$

$$\text{But } \text{FOLLOW}(T) = \{ +,), \$ \}$$

$$\text{Hence } \text{FOLLOW}(F) = \{ +,), \$ \}$$

$$\text{Finally } \text{FOLLOW}(F) = \{ * \} \cup \{ +,), \$ \}$$

$$\text{FOLLOW}(F) = \{ +, *,), \$ \}$$

To summarize above computation

Symbols	FIRST	FOLLOW
E	{(, id}	{), \$}
E'	{+, ε}	{), \$}
T	{(, id}	{+,), \$}
T'	{*, ε}	{+,), \$}
F	{(, id}	{+, *,), \$}

Algorithm for Predictive parsing table –

The construction of predictive parsing table is an important activity in predictive parsing method. This algorithm requires FIRST and FOLLOW functions

Input : The Context Free Grammar G

Output : Predictive Parsing table M.

Algorithm : For the rule $A \rightarrow \alpha$ of grammar G

1. For each a in $\text{FIRST}(\alpha)$ create entry $M[A, a] = A \rightarrow \alpha$ where a is terminal symbol.
2. For ϵ in $\text{FIRST}(\alpha)$ create entry $M[A, b] = A \rightarrow \alpha$
Where b is the symbols from $\text{FOLLOW}(A)$.
3. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$ then create entry in the table $M[A, \$] = A \rightarrow \alpha$.
4. All the remaining entries in the table M are marked as SYNTAX ERROR



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$\$E' T'$	$* id \$$	
$\$E' T' F*$	$* id \$$	$T \rightarrow * FT'$
$\$E' T' F$	$id \$$	
$\$E' T' id$	$id \$$	$F \rightarrow id$
$\$E' T'$	$\$$	
$\$E'$	$\$$	$T \rightarrow \epsilon$
$\$$	$\$$	$E' \rightarrow \epsilon$

Thus the input string gets parsed.

Thus it is observed that the input is scanned from left to right and we always follow leftmost derivation while parsing the input string. Also at a time only one input symbol is referred to taking the parsing action. Hence the name of this parser is LL(1). The LL(1) Parser is a table driven predictive parser. The left recursion and ambiguous grammar is not allowed for LL(1) parser.

➡ **Example 3.7 :** Show that following grammar :

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

is LL (1).

Solution : Consider the grammar :

$$S \rightarrow AaAb$$

$$S \rightarrow BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

Now we will compute FIRST and FOLLOW functions.

FIRST (S) = {a, b} if we put

$$S \rightarrow AaAb$$

$$S \rightarrow aAb \quad \text{When } A \rightarrow \epsilon$$

also $S \rightarrow BbBa$

$$S \rightarrow bBa \quad \text{When } B \rightarrow \epsilon$$

$$\text{FIRST (A) = FIRST (B) = } \{\epsilon\}$$

$$\text{FOLLOW (S) = } \{\$\}$$

$$\text{FOLLOW (A) = FOLLOW (B) = } \{a, b\}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The predictive parsing table can be constructed as

	a	()	,	\$
S	$S \rightarrow a$	$S \rightarrow (L)$			
L	$L \rightarrow SL'$	$L \rightarrow SL'$			
L'			$L' \rightarrow \epsilon$	$L' \rightarrow ,SL'$	

Q.3 : Construct the behaviour of the parser on the sentence (a, a) using the grammar specified above.

Ans. : As we have constructed a predictive parsing table in Q.2 we will parse the string (a, a) using that table as shown below.

State	Input	Actions
\$ S	(a, a) \$	
\$) L ((a, a) \$	$S \rightarrow (L)$
\$) L	a, a) \$	$L \rightarrow SL'$
\$) L' S	a, a) \$	$S \rightarrow a$
\$) L' a	a, a) \$	
\$) L'	, a) \$	$L' \rightarrow , SL'$
\$) L' S,	, a) \$	
\$) L' S	a) \$	$S \rightarrow a$
\$) L' a	a) \$	
\$) L') \$	$L' \rightarrow \epsilon$
\$)) \$	
\$	\$	Accept

Q.4 : Check whether the following grammar is LL(1) grammar.

$S \rightarrow iEtS/iEtSeS/a$

$E \rightarrow b$

Also define FIRST and FOLLOW procedures.

Ans. : Let,

$S \rightarrow iEtS/iEtSeS/a$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

\$) T') T' S	$\uparrow, (a), a)$	$S \rightarrow \uparrow$
\$) T') T' \uparrow	$\uparrow, (a), a)$	\$
\$) T') T'	, (a), a) \$	$T' \rightarrow ,ST'$
\$) T') T' S,	, (a), a) \$	
\$) T') T' S	(a), a) \$	
\$) T') T') + ((a), a) \$	
\$) T') T') T	a), a) \$	
\$) T') T') T' S	a), a) \$	$S \rightarrow a$
\$) T') T') T' a	a), a) \$	
\$) T') T') T'	, a) \$	$T' \rightarrow \epsilon$
\$) T') T')	, a) \$	
\$) T') T'	, a) \$	$T' \rightarrow ,ST'$
\$) T') T' S,	, a) \$	
\$) T') T' S	a) \$	$S \rightarrow a$
\$) T') T' a	a) \$	
\$) T') T') \$	$T' \rightarrow \epsilon$
\$) T')) \$	
\$) T'	\$	

The input string is read completely but there are some states in stack. Hence this is an invalid string for given grammar.

Q.6 : What is an ambiguous grammar ? Give an example.

Ans. : A grammar G is said to be ambiguous if it generates more than one parse trees for sentence of language L(G).

For example :

$$E \rightarrow E + E \mid E * E \mid id$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

\$ E' T')) / num \$	$E' \rightarrow \epsilon$
\$ E' T'	/ num \$	
\$ E' T' F/	/ num \$	$T' \rightarrow / FT'$
\$ E' T' F	num \$	
\$ E' T' P	num \$	$F \rightarrow P$
\$ E' T' num	num \$	$P \rightarrow \text{num}$
\$ E' T'	\$	
\$ E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

Q.8 : Consider the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

a) What are the terminals, non terminals and start symbol ?

b) Find parse trees for the following sentences :

(i) (a, a)

(ii) (a, (a, a))

(iii) (a, ((a, a), (a,a)))

c) Construct a leftmost derivation for each of the sentences in (b).

d) Construct a rightmost derivation for each of the sentences in (b).

e) What language does the grammar generate ?

Ans. :

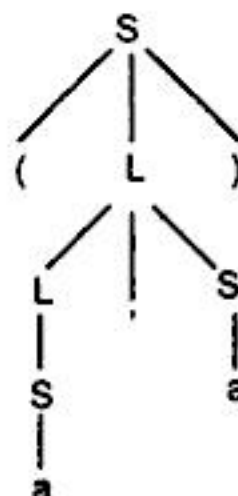
a) The terminals are $T = \{a, (,), ,\}$

The non terminals are $V = \{ L, S \}$

The start symbol is S.

b) The parse tree for

i) (a, a) is





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Q.9 : For the following given grammar G ,

$A \rightarrow Ba \mid bC$

$B \rightarrow d \mid eBf$

$C \rightarrow gC \mid g$

determine which of the following strings are in the above language. For the strings belonging to above grammar construct parse trees.

The strings are : $\{bg, bffd, bggg, edfa, eedffa, faae\}$

Ans. :

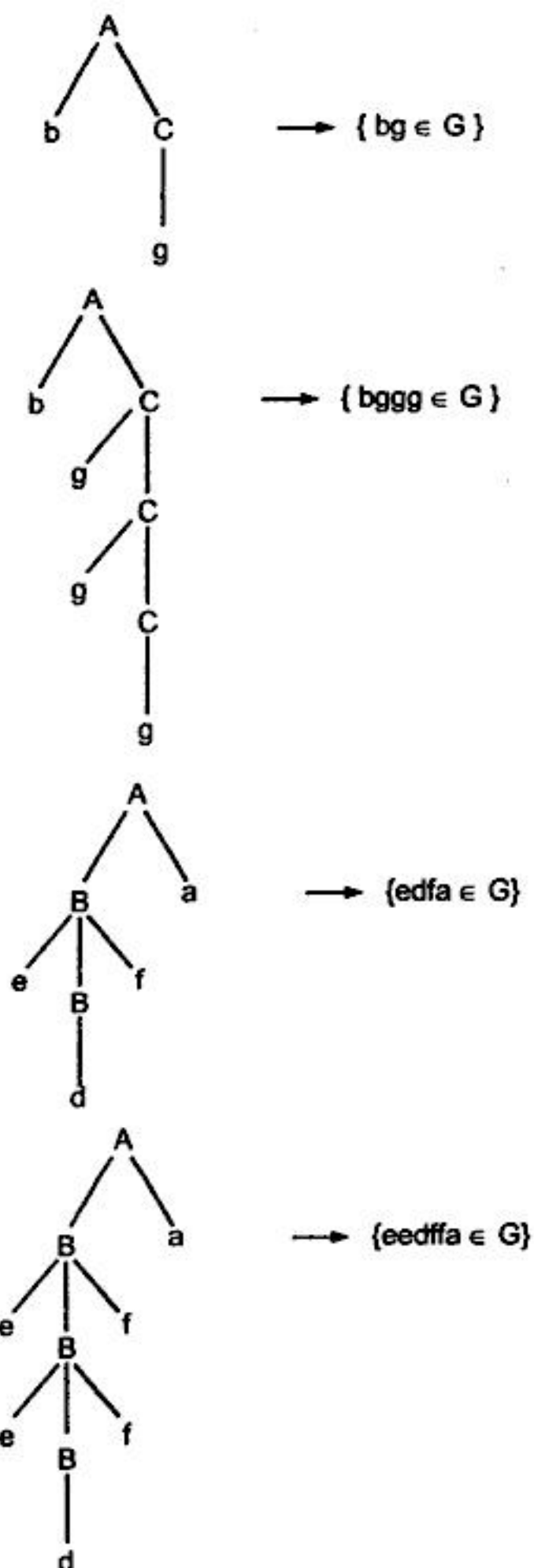


Fig. 3.21



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Bottom Up Parsing

4.1 Introduction

In previous chapter, we have discussed top down parsing technique along with its methods. Top down parsing is an approach in which the parse tree gets build from top to bottom. The top down parsing has its own merits and demerits. In this chapter we will discuss another parsing technique called **bottom up parsing**. This class of parsing technique has various methods. We will discuss various methods from this family of parsing technique such as shift reduce parsing, SLR, LALR and LR parsing. While learning these methods our intention will be to understand "how the input string gets parsed using above mentioned methods ?"

The task of parser is twofold : either it checks the input string completely for its syntax or it generates error messages on syntactically invalid input strings. Hence we will also discuss error recovery in parsing and handling of ambiguous grammar.

Last but not least - we will discuss automatic parser generator : YACC an utility from UNIX. This section is well supported by programming examples.

4.2 Concept of Bottom Up Parser

In bottom up parsing method, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol. The process of parsing halts successfully as soon as we reach to start symbol.

The parse tree is constructed from bottom to up that is from leaves to root. In this process, the input symbols are placed at the leaf nodes after successful parsing. The bottom up parse tree is created starting from leaves, the leaf nodes together are reduced further to internal nodes, these internal nodes are further reduced and eventually a root node is obtained. The internal nodes are created from the list of Terminal and non terminal symbols. This involves -

Non Terminal for internal node = Non terminal \cup terminal

In this process, basically parser tries to identify RHS of production rule and replace it by corresponding LHS. This activity is called reduction. Thus the crucial but



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The bold strings are called handles.

Right sentential form	Handle	Production
id + id + id	id	$E \rightarrow id$
E + id + id	id	$E \rightarrow id$
E + E + id	id	$E \rightarrow id$
E + E + E	E + E	$E \rightarrow E + E$
E + E	E+E	$E \rightarrow E + E$
E		

Thus bottom parser is essentially a process of detecting handles and using them in reduction.

4.3 Shift Reduce Parser

Shift reduce parser attempts to construct parse tree from leaves to root. Thus it works on the same principle of bottom up parser. A shift reduce parser requires following data structures –

1. The input buffer storing the input string.
2. A stack for storing and accessing the LHS and RHS of rules.

The initial configuration of Shift reduce parser is as shown below -



The parser performs following basic operations.

Fig. 4.2 Initial configuration

1. **Shift** : Moving of the symbols from input buffer onto the stack, this action is called shift.
2. **Reduce** : If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means RHS of rule is popped of and LHS is pushed in. This action is called Reduce action.
3. **Accept** : If the stack contains start symbol only and input buffer is empty at the same time then that action is called accept. When accept state is obtained in the process of parsing then it means a successful parsing is done.
4. **Error** : A situation in which parser can not either shift or reduce the symbols, it can not even perform the accept action is called as error.

Let us take some examples to learn the shift-reduce parser.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

These meanings appear similar to the less than, equal to and greater than operators. Now by considering the precedence relation between the arithmetic operators we will construct the operator precedence table. The operators precedences we have considered are id,+,*, $\$$

	id	+	*	$\$$
id		$\cdot >$	$\cdot >$	$\cdot >$
+	$< \cdot$	$\cdot >$	$\cdot >$	$\cdot >$
*	$< \cdot$	$\cdot >$	$\cdot >$	$\cdot >$
$\$$	$< \cdot$	$< \cdot$	$< \cdot$	

Fig. 4.3 Precedence relation table

Now the consider the string

id + id * id

We will insert $\$$ symbols at the start and end of the input string. We will also insert precedence operator by referring the precedence relation table.

$\$ < \cdot \text{id} \cdot > + < \cdot \text{id} \cdot > * < \text{id} \cdot > \$$

We will follow following steps to parse the given string –

- Scan the input from left to right until first $\cdot >$ is encountered
- Scan backwards over $=$ until $< \cdot$ is encountered.
- The handle is a string between $< \cdot$ and $\cdot >$.

The parsing can be done as follows -

$\$ < \cdot \text{id} \cdot > + < \cdot \text{id} \cdot > * < \text{id} \cdot > \$$	Handle id is obtained between $< \cdot \cdot >$ Reduce this by $E \rightarrow \text{id}$
$E + < \cdot \text{id} \cdot > * < \text{id} \cdot > \$$	Handle id is obtained between $< \cdot \cdot >$ Reduce this by $E \rightarrow \text{id}$
$E + E * < \cdot \text{id} \cdot > \$$	Handle id is obtained between $< \cdot \cdot >$ Reduce this by $E \rightarrow \text{id}$
$E + E * E$	Remove all the non terminals.
$++$	Insert $\$$ at the beginning at the end. Also insert the precedence operators.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

For example :

$$S \rightarrow \bullet ABC$$

$$S \rightarrow A \bullet BC$$

$$S \rightarrow AB \bullet C$$

$$S \rightarrow ABC \bullet$$

The production $S \rightarrow \epsilon$ generates only one item $S \rightarrow \bullet$.

- 2) **Augmented grammar** : If a grammar G is having start symbol S then augmented grammar is a new grammar G' in which S' is a new start symbol such that $S' \rightarrow S$. The purpose of this grammar is to indicate the acceptance of input. That is when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.
- 3) **Kernel items** : It is collection of items $S \rightarrow \bullet S$ and all the items whose dots are not at the leftmost end of RHS of the rule.
Non-Kernel items : The collection of all the items in which \bullet are at the left end of RHS of the rule.
- 4) **Functions closure and goto** : These are two important functions required to create collection of canonical set of items.
- 5) **Viable prefix** : It is the set of prefixes in the right sentential form of production $A \rightarrow \alpha$. This set can appear on the stack during shift/reduce action.

Closure operation

For a context free grammar G , if I is the set of items then the function $\text{closure}(I)$ can be constructed using following rules.

1. Consider I is a set of canonical items and initially every item I is added to $\text{closure}(I)$.
2. If rule $A \rightarrow \alpha \bullet B\beta$ is a rule in $\text{closure}(I)$ and there is another rule for B such as $B \rightarrow \gamma$ then

$\text{closure}(I) : A \rightarrow \alpha \bullet B\beta$
 $B \rightarrow \bullet \gamma$

This rule has to be applied until no more new items can be added to $\text{closure}(I)$.

The meaning of rule $A \rightarrow \alpha \bullet B\beta$ is that during derivation of the input string at some point we may require strings derivable from B . A non terminal immediately to the right of \bullet indicates that it has to be expanded shortly.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

I_0 . But again as we can see that the rule $E \rightarrow \bullet T$ which we have added, contains non terminal T immediately right to \bullet . So we have to add T -productions in I_0 , $T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$.

In T -productions after \bullet comes T and F respectively. But since we have already added T productions so we will not add those. But we will add all the F -productions having dots. The $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet id$ will be added. Now we can see that after dot (and id are coming in these two productions. The $($ and id are terminal symbols and are not deriving any rule. Hence our closure function terminates over here. Since there is no rule further we will stop creating I_0 .

Now apply $goto(I_0, E)$

$$\begin{array}{l} E' \rightarrow \bullet E \\ E \rightarrow \bullet E + T \end{array} \xrightarrow[\text{right}]{\text{Shift dot to}} \begin{array}{l} E' \rightarrow E \bullet \\ E \rightarrow E \bullet + T \end{array}$$

Thus I_1 becomes

$$\begin{array}{l} goto(I_0, E) \\ I_1: E' \rightarrow E \bullet \\ E \rightarrow E \bullet + T \end{array}$$

Since in I_1 there is no non terminal after dot we can not apply $closure(I_1)$.

By applying $goto$ on T of I_0

$$\begin{array}{l} goto(I_0, T) \\ I_2: E \rightarrow T \bullet \\ T \rightarrow T \bullet * F \end{array}$$

Since in I_2 there is no non terminal after dot we can not apply $closure(I_2)$.

By applying $goto$ on F of I_0

$$\begin{array}{l} goto(I_0, F) \\ I_3: T \rightarrow F \bullet \end{array}$$

Since after dot in I_3 there is nothing, hence we can not apply $closure(I_3)$.

By applying $goto$ on $($ of I_0 . But after dot E comes hence we will apply closure on E , then on T , then on F .



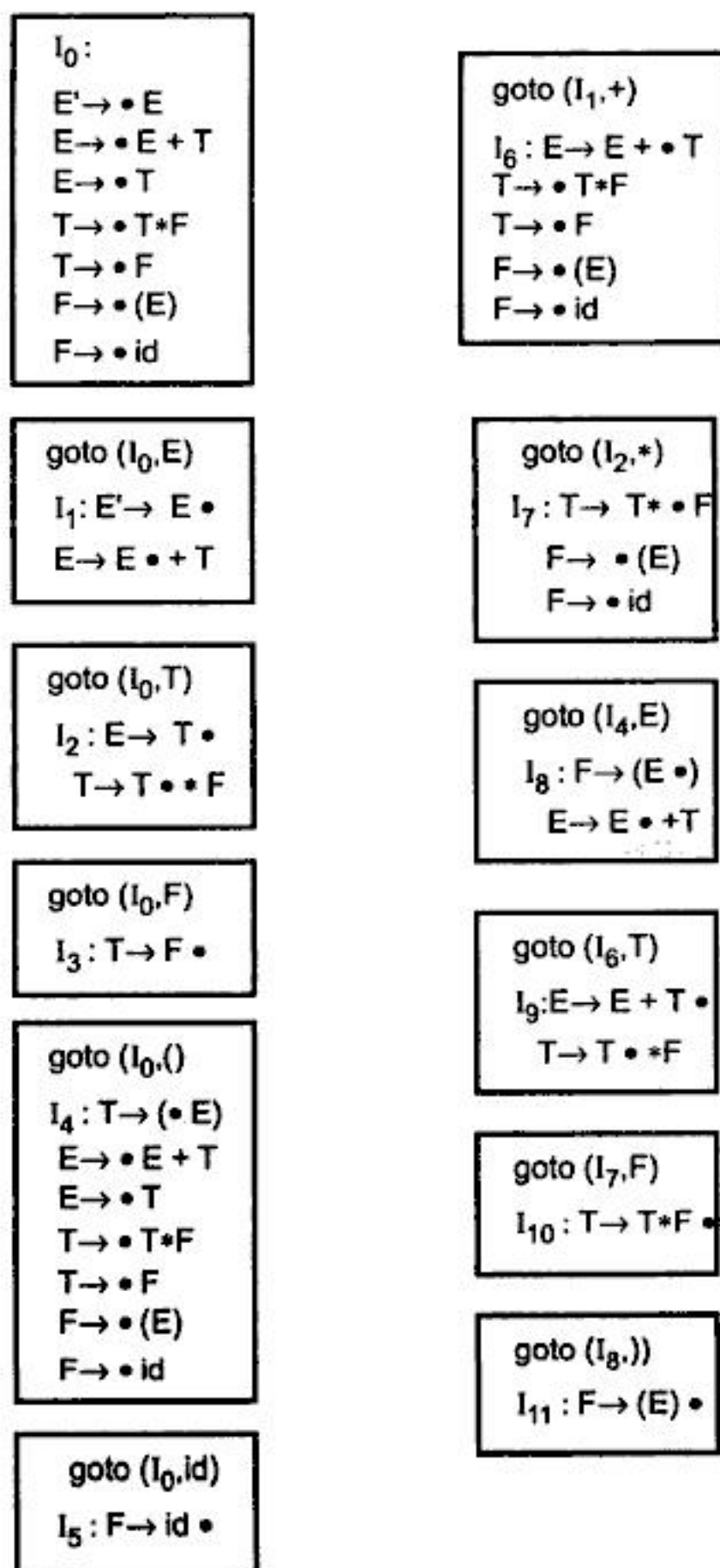
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Consider $F \rightarrow \bullet (E)$

$A \rightarrow \alpha \bullet a\beta$

$A=F, \alpha = \epsilon, a= (, \beta = E)$

$\text{goto}(I_0, () = I_4$

$\therefore \text{action}[0, (] = \text{shift } 4$

Similarly for $F \rightarrow \bullet \text{id}$

Entry in the action table $\text{action}[0, \text{id}] = \text{shift } 5 \because \text{goto}(I_0, \text{id}) = I_5$

Other item in I_0 does not give any action. Hence we will find the actions from I_0 to I_{11} .

State	action						goto		
	id	+	*	()	\$	E	T	F
0	S5			S4					
1		S6							
2			S7						
3									
4	S5			S4					
5									
6	S5			S4					
7	S5			S4					
8		S6			S11				
9			S7						
10									
11									

Thus SLR parsing table is filled up with the shift actions. Now we will fill it up with reduce and accept action. The S in the table indicates "shift" action.

According to the rule 2.c from parsing table algorithm, there is a production $E' \rightarrow E \bullet$ in I_1 . Hence we will add the action "Accept" in $\text{action}[1, \$]$.

Now in state I_2

$E \rightarrow T \bullet$

$A \rightarrow \alpha \bullet$ rule 2b

$A = E, \alpha = T$ For this we want to find FOLLOW (E)

$\text{FOLLOW}(E) = \{+,), \$\}$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

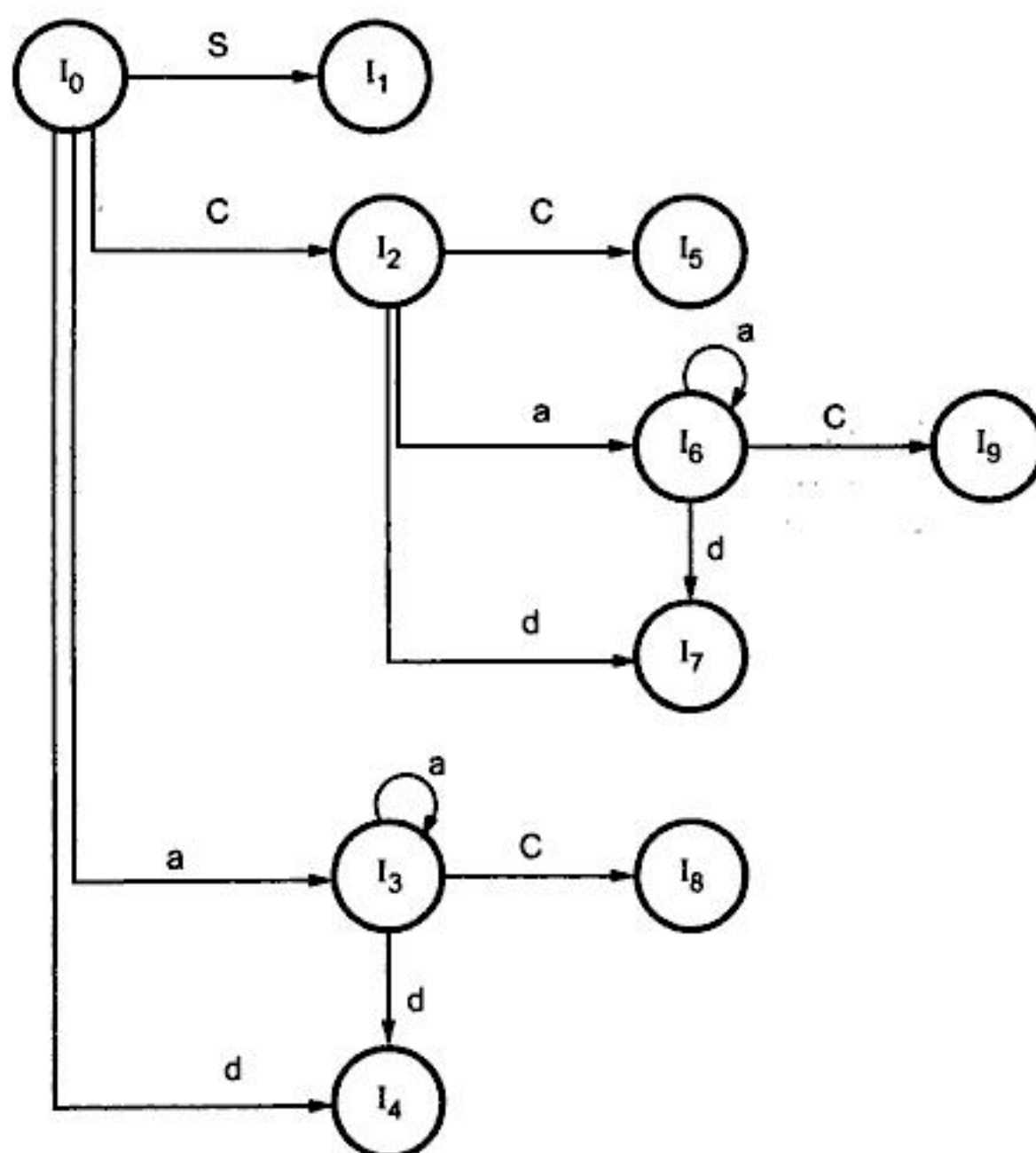


Fig. 4.8 DFA [goto graph]

$C \rightarrow d \bullet, a/d$

$A \rightarrow \alpha \bullet, a$

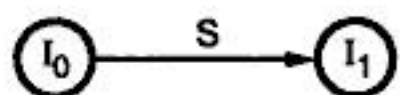
$A=C, \alpha = d, a=a/d$

action[4,a] = reduce by $C \rightarrow d$ i.e. rule 3

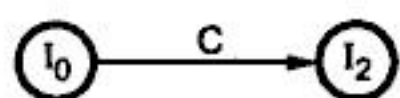
$S' \rightarrow S \bullet, \$$ in I_1

So we will create action[1,\$]=accept.

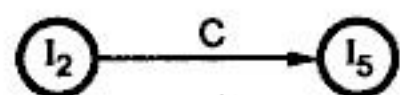
The goto table can be filled by using the goto functions.



goto (I_0, S) = I_1



goto (I_0, C) = I_2



goto (I_2, C) = I_5



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Construction of LALR parsing table –

The algorithm for construction of LALR parsing table is as given below –

Step 1 : Construct the LR(1) set of items.

Step 2 : Merge the two states I_i and I_j if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$

Step 3 : The parsing actions are based on each item I_i . The actions are as given below –

- If $[A \rightarrow \alpha \bullet a \beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$ then create an entry in the action table $\text{action}[I, a] = \text{shift } j$.
- If there is a production $[A \rightarrow \alpha \bullet, a]$ in I_i then in the action table $\text{action}[I_i, a] = \text{reduce by } A \rightarrow \alpha$. Here A should not be S' .
- If there is a production $S' \rightarrow S \bullet, \$$ in I_i then $\text{action}[i, \$] = \text{accept}$.

Step 4 : The goto part of the LR table can be filled as :The goto transitions for state i is considered for non terminals only. If $\text{goto}(I_i, A) = I_j$ then $\text{goto}[I_i, A] = j$.

Step 5 : If the parsing action conflict then the algorithm fails to produce LALR parser and grammar is not LALR(1). All the entries not defined by rule 3 and 4 are considered to be "error".

➡ Example 4.13 :

$S \rightarrow CC$

$C \rightarrow aC$

$C \rightarrow d$

Construct the parsing table for LALR(1) parser.

Solution : First the set LR(1) items can be constructed as follows with merged states.

$I_0:$ $S' \rightarrow \bullet S, \$$ $S \rightarrow \bullet CC, \$$ $C \rightarrow \bullet aC, a/d$ $C \rightarrow \bullet d, a/d$	$I_{36}:$ goto (I_0, a) $C \rightarrow a \bullet C, a/d/\$$ $C \rightarrow \bullet aC, a/d/\$$ $C \rightarrow \bullet d, a/d/\$$
$I_1:$ goto (I_0, S) $S' \rightarrow S \bullet, \$$	$I_{47}:$ goto (I_0, d) $C \rightarrow d \bullet, a/d/\$$
$I_2:$ goto (I_0, C) $S \rightarrow C \bullet C, \$$ $C \rightarrow \bullet aC, \$$ $C \rightarrow \bullet d, \$$	$I_5:$ goto (I_2, C) $S \rightarrow CC \bullet, \$$ $I_{89}:$ goto (I_3, C) $C \rightarrow aC \bullet a/d/\$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$I_6 : \text{goto } (I_3, A)$

$S \rightarrow bA \cdot c, \$$

$I_7 : \text{goto } (I_3, d)$

$S \rightarrow bd \cdot a, \$$

$A \rightarrow d \cdot, c$

$I_8 : \text{goto } (I_4, c)$

$S \rightarrow dc \cdot, \$$

$I_9 : \text{goto } (I_6, c)$

$S \rightarrow bAc \cdot, \$$

$I_{10} : \text{goto } (I_7, a)$

$S \rightarrow bda \cdot, \$$

In above set of canonical items no states are having common production rules. Hence we can not merge these states. The same set of items will be considered for building LALR parsing table.

We will construct LALR parsing table using following rules.

1. If $[A \rightarrow \alpha \cdot a\beta, b]$ is in I_i and $\text{goto } (I_i, a) = I_j$ then action $[i, a] = \text{Shift } j$.
2. If there is a production $[A \rightarrow \alpha \cdot, a]$ in some state I_i then action $[i, a] = \text{reduce by } A \rightarrow \alpha$.
3. If there is a product $S' \rightarrow S \cdot, \$$ in I_i then action $[i, \$] = \text{accept}$.

Action						goto	
	a	b	c	d	\$	S	A
0		S3		S4		1	2
1					Accept		
2	S5						
3				S7			6
4	r5		S8				
5							
6			S9				
7	S10		r5				
8					r3		
9					r2		
10					r4		



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Now using the above set of LR(1) items we will construct LR(1) parsing table as follows -

States	Action					goto		
	a	b	c	d	\$	S	A	B
0		S3		S5		1	2	4
1					ACCEPT			
2	S6							
3				S9			7	8
4			S10					
5	r5		r6					
6					r1			
7			S11					
8	S12							
9	r6		r5					
10					r3			
11					r2			
12					r4			

We can parse the string "bda" using above constructed LR (1) parsing table as :

Stack	Input buffer	Action
\$0	bda\$	Shift 3
\$0b3	da\$	Shift 9
\$0b3d9	a\$	Reduce by B \rightarrow d
\$0b3B8	a\$	Shift 12
\$0b3B8a12	\$	Reduce by S \rightarrow bBa
\$0S1	\$	Accept



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Using precedence and associativity to resolve parsing action conflicts

Consider an ambiguous grammar for arithmetic expression –

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Now we will build the set of LR(0) items for this grammar.

$I_0:$ $E' \rightarrow \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$	$I_5: goto(I_1, *)$ $E \rightarrow E * \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$
$I_1: goto(I_0, E)$ $E' \rightarrow E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$	$I_6: goto(I_2, E)$ $E \rightarrow (E \bullet)$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$
$I_2: goto(I_0, ($ $E' \rightarrow (\bullet E)$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$	$I_7: goto(I_4, E)$ $E \rightarrow E + E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$
$I_3: goto(I_0, id)$ $E \rightarrow id \bullet$	$I_8: goto(I_5, E)$ $E \rightarrow E * E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$
$I_4: goto(I_1, +)$ $E \rightarrow E + \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$	$I_9: goto(I_6,)$ $E \rightarrow (E) \bullet$

Here FOLLOW(E) = {+, *,), \$}.

We have computed FOLLOW (E) as it will be required while processing.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$I_0:$ $S' \rightarrow \bullet S$ $S \rightarrow \bullet iSeS$ $S \rightarrow \bullet iS$ $S \rightarrow \bullet a$	$I_3: goto(I_0, a)$ $S \rightarrow a \bullet$
$I_1: goto(I_0, S)$ $S' \rightarrow S \bullet$	$I_4: goto(I_2, S)$ $S \rightarrow iS \bullet es$ $S \rightarrow iS \bullet$
$I_2: goto(I_0, i)$ $S \rightarrow i \bullet Ses$ $S \rightarrow i \bullet S$ $S \rightarrow \bullet iSeS$ $S \rightarrow \bullet iS$ $S \rightarrow \bullet a$	$I_5: goto(I_4, e)$ $S \rightarrow iSe \bullet S$ $S \rightarrow \bullet iSeS$ $S \rightarrow \bullet iS$ $S \rightarrow \bullet a$
	$I_6: goto(I_5, S)$ $S \rightarrow iSeS \bullet$

The FOLLOW(S)={e,\$}. Now we will build the SLR parse table for above obtained set of items.

State	Action				goto
	i	e	a	\$	
0	S2		S3		1
1				Accept	
2	S2		S3		4
3		r3		r3	
4		S5 or r2		r2	
5	S2		S3		6
6		r1		r2	

Clearly in the above table at action[5,e] there is a shift/reduce conflict. We will now try to resolve it.

Consider the input "iaea\$" for processing –

Stack	Input	Action with conflict resolution
\$0	iaea\$	Shift
\$0i2	iaea\$	Shift
\$0i2i2	aea\$	Shift
\$0i2i2a3	ea\$	Reduce $S \rightarrow a$
\$0i2i2S4	ea\$	From the conflict we have chosen Reduce $S \rightarrow iS$
\$0i2S4	ea\$	Reduce $S \rightarrow iS$
\$0S1	a\$	Error!!



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

From all these situations we conclude some error messages are :

- E1 : These errors are in states I_0, I_2, I_4 and I_5 . This indicates that the operand should appear before operator. Hence the error message will be "missing operand".
- E2 : This error is in) column and from states I_0, I_1, I_2, I_4 and I_5 indicating unbalancing in right parenthesis. Hence error message will be "unbalanced right parenthesis".
- E3 : The operator is expected in this case of error as it is from state I_1 or I_6 . The error message will be "missing operator".
- E4 : This error occurs at state 6 in the \$ column. The state 6 expects) parenthesis at the end of expression. Hence the error message will be "missing right parenthesis".

Thus the modified table with appropriate error messages is as shown below –

State	Action						goto
	id	+	*	()	\$	E
0	S3	E1	E1	S2	E2	E1	1
1	E3	S4	S5	E3	E2	Accept	
2	S3	E1	E1	S2	E2	E1	6
3	r4	r4	r4	r4	r4	r4	
4	S3	E1	E1	S2	E2	E1	7
5	S3	E1	E1	S2	E2	E1	8
6	E3	S4	S5	E3	S9	E4	
7	r1	r1	S5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

While parsing the input, LR parser will refer the parsing table if it detects the error entry then respective error message will be reported. This is how we get syntax errors when we compile our program.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

/*For log no | LOG no (log base 10)*/
log |
LOG {return LOG;}

/*For ln no (Natural log)*/
ln {return nLOG;}

/*For sin angle*/
sin |
SIN {return SINE;}

/*For cos angle*/
cos |
COS {return COS;}

/*For tan angle*/
tan |
TAN {return TAN;}

/*For memory*/
mem {return MEM;}

[ \t] ; /*Ignore white spaces*/

/*End of input*/
\$, {return 0;}

/*Catch the remaining and return a single character
token to the parser*/
\n|. return yytext[0];
%%

```

The YACC program

```

/*Program Name :calci.y */
%{
double memvar;
%}

/*To define possible symbol types*/
%union
{
double dval;
}

/*Tokens used which are returned by lexer*/
%token <dval> NUMBER
%token <dval> MEM
%token LOG SINE nLOG COS TAN

```




You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The routine `yyerror` is used to print the error message when an error is occurred in parsing of input.

Thus we have seen how an input string is parsed and syntactically checked.

Solved Exercise

Q. 1 : What do you mean by handle pruning ?

Ans. : To obtain rightmost derivation in reverse, handle pruning method is used. Consider that W is an input string then the right most derivation in reverse can be

$$S \Rightarrow \underset{\text{rm}}{\gamma_0} \Rightarrow \underset{\text{rm}}{\gamma_1} \Rightarrow \underset{\text{rm}}{\gamma_2} \Rightarrow \dots \underset{\text{rm}}{\gamma_n} = W$$

We locate handle from the input string and that handle can be replaced by LHS rule for corresponding handle. Thus input string can be reduced to start symbol. This method is called handle pruning.

For example :

Input	Handle	Production used
$E + id_2 * id_3$	id_1	$E \rightarrow id_1$
$E + id_2 * id_3$	id_2	$E \rightarrow id_2$
$E + E * id_3$	id_3	$E \rightarrow id_3$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E	—	—

Q. 2 : Define LR(0) items.

Ans. : The LR(0) item for grammar G is a production rule in which symbol \bullet is inserted at some position in RHS of the rule. For example

$$S \rightarrow \bullet A B C$$

$$S \rightarrow A \bullet B C$$

$$S \rightarrow A B \bullet C$$

$$S \rightarrow A B C \bullet$$

The production $S \rightarrow \epsilon$ generates only one item $S \rightarrow \bullet$.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$I_1 :$	goto (I_0, S) $S' \rightarrow S \bullet, \$$	$I_7 :$	goto (I_4, R) $L \rightarrow * R \bullet = \$$
$I_2 :$	goto (I_0, L) $S \rightarrow L \bullet = R, \$$ $R \rightarrow L \bullet, \$$	$I_8 :$	goto (I_4, L) $R \rightarrow L \bullet, = \$$
$I_3 :$	goto (I_0, R) $S \rightarrow R \bullet, \$$	$I_9 :$	goto (I_6, R) $S \rightarrow L = R \bullet, \$$
$I_4 :$	goto ($I_0, *$) $L \rightarrow * \bullet R, = \$$ $R \rightarrow \bullet L, = \$$ $L \rightarrow \bullet * R, = \$$ $L \rightarrow \bullet id, = \$$	$I_{10} :$	goto (I_6, L) $R \rightarrow L \bullet, \$$
$I_5 :$	goto (I_0, id) $L \rightarrow id \bullet, = \$$	$I_{11} :$	goto ($I_6 *$) $L \rightarrow * \bullet R, \$$ $R \rightarrow \bullet L, \$$ $L \rightarrow \bullet * R, \$$ $L \rightarrow \bullet id, \$$
		$I_{12} :$	goto (I_6, id) $L \rightarrow id \bullet, \$$
		$I_{13} :$	goto (I_{11}, R) $L \rightarrow * R \bullet, \$$

From above set of items we have got

i) I_4 and I_{11} give same production but lookheads are different. Hence merge them to form I_{411}

ii) $I_5 = I_{12}$ Hence I_{512}

iii) $I_7 = I_{13}$ Hence I_{713}

iv) $I_8 = I_{10}$ Hence I_{810}

Therefore the set of items for LALR are

$I_0 :$	$S' \rightarrow \bullet S, \$$ $S \rightarrow \bullet L = R, \$$ $S \rightarrow \bullet R, \$$ $L \rightarrow \bullet * R, = \$$ $L \rightarrow \bullet id, = \$$ $R \rightarrow \bullet L, \$$	$I_{713} :$	$L \rightarrow * R \bullet, = \$$
		$I_{810} :$	$R \rightarrow L \bullet, = \$$
		$I_9 :$	$S \rightarrow L = R \bullet, \$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Review Questions

1. Define the term Bottom up parsing.
2. Give different types of bottom up parsing methods.
3. Write short note on SLR parser.
4. Write an algorithm for constructing SLR parsing table.
5. Compare SLR, LALR and LR parser.
6. What is dangling else problem? Discuss the computing of LR(O) items for the same. How a conflict gets resolved during parsing.
7. Write short note on i) YACC ii) YACC specification
8. Write a YACC program to convert infix to postfix form.
9. Explain various conflicts that occur during shift reduce parsing.
10. What is the use of look ahead symbol in LALR parsing.

□□□



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

ii) **Product** – The type expression for the cartesian product is given by $T1 \times T2$ where $T1$ and $T2$ are the two data types. Always \times is assumed to be left associative.

iii) **Struct** – The structure given by keyword struct is also a type expression. The structure is applied as a product of the type of the fields(members) of the structure. Such a product is a type expression in struct.

For example ;

```
struct stud {
    char name[10];
    float marks;
}
```

```
struct stud student[10];
```

Here the type name is stud having the type expression as

$\text{struct } ((\text{name} \times \text{array}(0,1\dots 9,\text{char})) \times (\text{marks float}))$

Thus the type expression for student is given as

$\text{array}(0,1,\dots 9,\text{stud})$

iv) **Pointers** – The type expression for pointer is given as $\text{pointer}(T)$ where T is a data type.

For example

```
float *xyz;
```

then type expression for identifier xyz is given as

$\text{pointer}(\text{float})$

v) **Function** – The type expression for function is given by domain-range. In the sense that the type expression for function is $D \rightarrow R$.

For example

```
int sum(int a,int b)
```

The type expression for sum is

$$\underbrace{\text{int} \times \text{int}}_{\text{Domain}} \rightarrow \underbrace{\text{int}}_{\text{range}}$$

- The type expression can be represented using tree or DAG(Directed Acyclic Graph) Let us take an example for the same

```
int maxi(int a,int b,float *c)
```




You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Here t1 and t2 are the temporary names generated by the compiler. There are at the most three addresses allowed (two for operands and one for result). Hence the name of this representation is three-address code.

5.5.1 Types of Three Address Statements

The form of three address code is very much similar to assembly language. Here are some commonly used three address codes for typical language constructs.

Language construct	Intermediate code form	Meaning
Assignment statement	$x := y \text{ op } z$	Here binary operation is performed using operator 'op'
Assignment statement	$x := \text{op } y$	Here the unary operation is performed. The operator 'op' is an unary operator.
Copy statement	$x := y$	Here the value of y is assigned to x.
Unconditional jump	goto L	The control flow goes to the statement labeled by L
Conditional jump	if x relop y goto L	The relop indicates the relational operators such as <, =, >= . If x relop y is true then it executes goto L statement.
Procedure calls	param x_1 param x_2 ... param x_n call p, n return y	Here the parameters x_1, x_2, \dots, x_n are used as parameters to the procedure p. The return statement indicates the return value y.
Array statements	$x := y[i]$ $x[i] := y$	The value at i^{th} index of array y is assigned to x. The value of identifier y is assigned at the index i of the array x.
Address and pointer assignments	$x := \&y$ $x := *y$ $*x := y$	The value of x will be the address or location of y The y is a pointer whose value is assigned to x The r-value of object pointed by x is set by the l-value of y.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Q. 4 : Construct abstract syntax tree and DAG for the assignment statement
 $x = a * b + c - a * b + d$

Ans. :

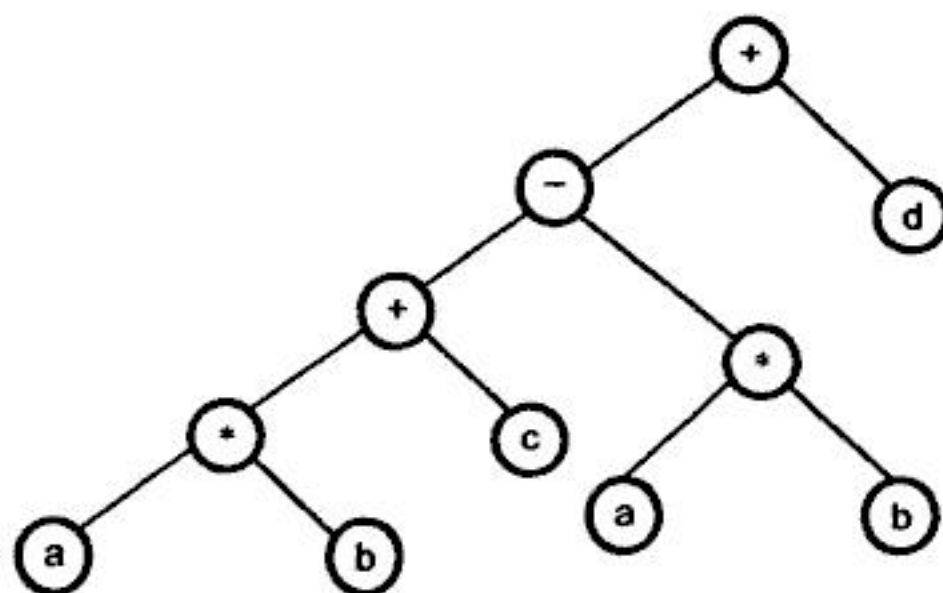


Fig. 5.7 Abstract syntax tree

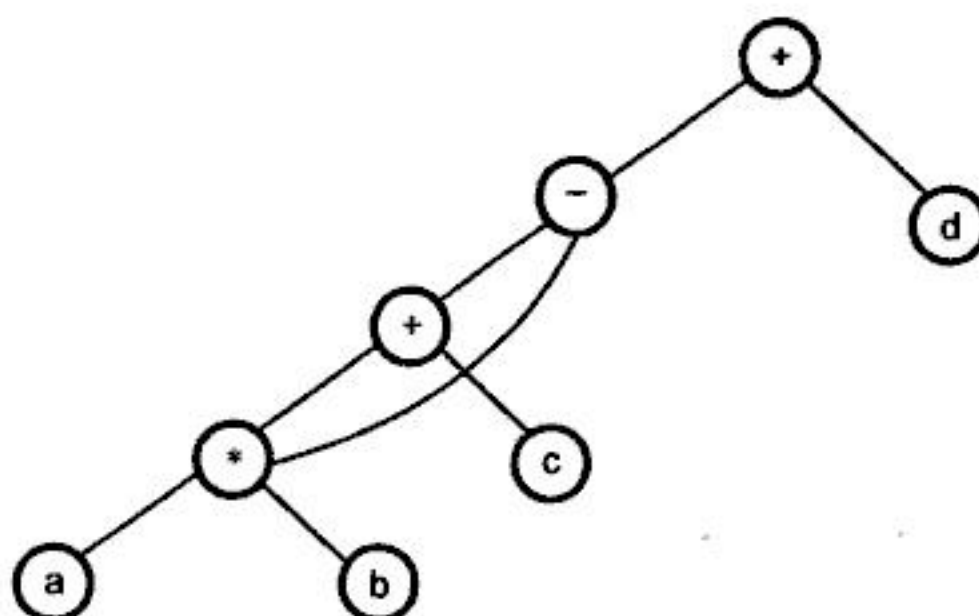


Fig. 5.8 DAG

Review Questions

1. What are the benefits of intermediate code generation ?
2. Explain various forms of intermediate code with some suitable examples.
3. What are different types of three address statements ?
4. Write Quadruple, Triple and three address code for
 $(a+b) * (a+b) - (a+b) * d$

□□□



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The syntax directed definition can be written for the above grammar by using semantic actions for each production.

Production Rule	Semantic actions
$S \rightarrow EN$	$\text{Print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow E_1 - T$	$E.\text{val} = E_1.\text{val} - T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} \times F.\text{val}$
$T \rightarrow T_1 / F$	$T.\text{val} = T_1.\text{val} / F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$
$N \rightarrow ;$	Can be ignored by lexical analyzer as ; is terminating symbol.

For the non-terminals E, T and F the values can be obtained using the attribute "val". Here "val" is a attribute and semantic rule is computing the value of val. (How? that we will discuss shortly!)

The token digit has synthesized attribute *lexval* whose value can be obtained from lexical analyzer. In the rule $S \rightarrow EN$, symbol S is the start symbol. This rule is to print the final answer of the expression.

- In syntax directed definition, terminals have synthesized attributes only.
- Thus there is no definition of terminal. The synthesized attributes are quite often used in syntax directed definition. The syntax directed definition that uses only synthesized attributes is called **s-attributed definition**.
- In a parse tree, at each node the semantic rule is evaluated for annotating (computing) the S-attributed definition. This processing is in **bottom up** fashion i.e. from leaves to root.

Following steps are followed to compute S-attributed definition –

1. Write the Syntax directed definition using the appropriate semantic actions for corresponding production rule of the given grammar.
2. The annotated parse tree is generated and attribute values are computed. The computation is done in bottom up manner.
3. The value obtained at the root node is supposed to be the final output.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The value of L nodes is first obtained from T.type(sibling). The T.type is basically lexical value obtained as int or float or char or double. Then the L nodes give the type of the identifiers a, b and c. The computation of type is done in top-down manner or in preorder traversal. Using function Enter_type the type of identifiers a, b and c is inserted in the symbol table at corresponding id.entry (The id.entry is the address of corresponding identifier in the symbol table).

3. Dependency graph

The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependency graph.

For the rule $X \rightarrow YZ$ the semantic action is given by $X.x := f(Y.y, Z.z)$ then

Synthesized attribute is $X.x$ and $X.x$ depends upon attributes $Y.y$ and $Z.z$.

➡ **Example 6.1 :** Design the dependency graph for the following grammar

$$E \rightarrow E_1 + E_2$$

$$E \rightarrow E_1 * E_2$$

Solution : The semantic rules for the above grammar is as given below-

Production Rule	Semantic Rule
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$
$E \rightarrow E_1 * E_2$	$E.val := E_1.val \times E_2.val$

The dependency graph is as shown in Fig. 6.4.

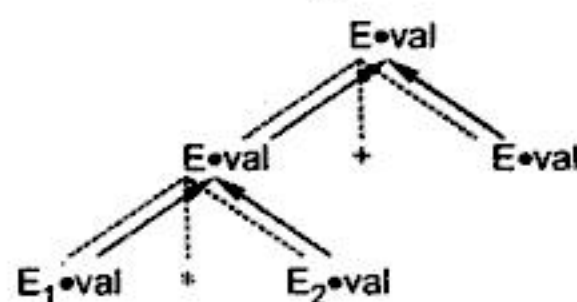


Fig. 6.4 Dependency graph

The synthesized attributes can be represented by $\bullet val$. Hence the synthesized attributes are given by $E \bullet val$, $E_1 \bullet val$ and $E_2 \bullet val$. The dependencies among the nodes is given by solid arrows. The arrows from E_1 and E_2 show that value of E depends upon E_1 and E_2 . We have represented parse tree using dotted lines.

➡ **Example 6.2 :** Design the dependency graph for the following grammar

$$S \rightarrow T \text{ List}$$

$$T \rightarrow \text{int}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Postfix expression $xy*5 - z+$

Symbol	Operation
x	$p_1 = \text{mkleaf}(\text{id}, \text{ptr to entry x})$
y	$p_2 = \text{mkleaf}(\text{id}, \text{ptr to entry y})$
*	$p_3 = \text{mknnode}(*, p_1, p_2)$
5	$p_4 = \text{mkleaf}(\text{num}, 5)$
-	$p_5 = \text{mknnode}(-, p_3, p_4)$
z	$p_6 = \text{mkleaf}(\text{id}, \text{ptr to entry z})$
+	$p_7 = \text{mknnode}(+, p_5, p_6)$

Consider the string $x*y-5+z$ and let us draw the syntax tree.

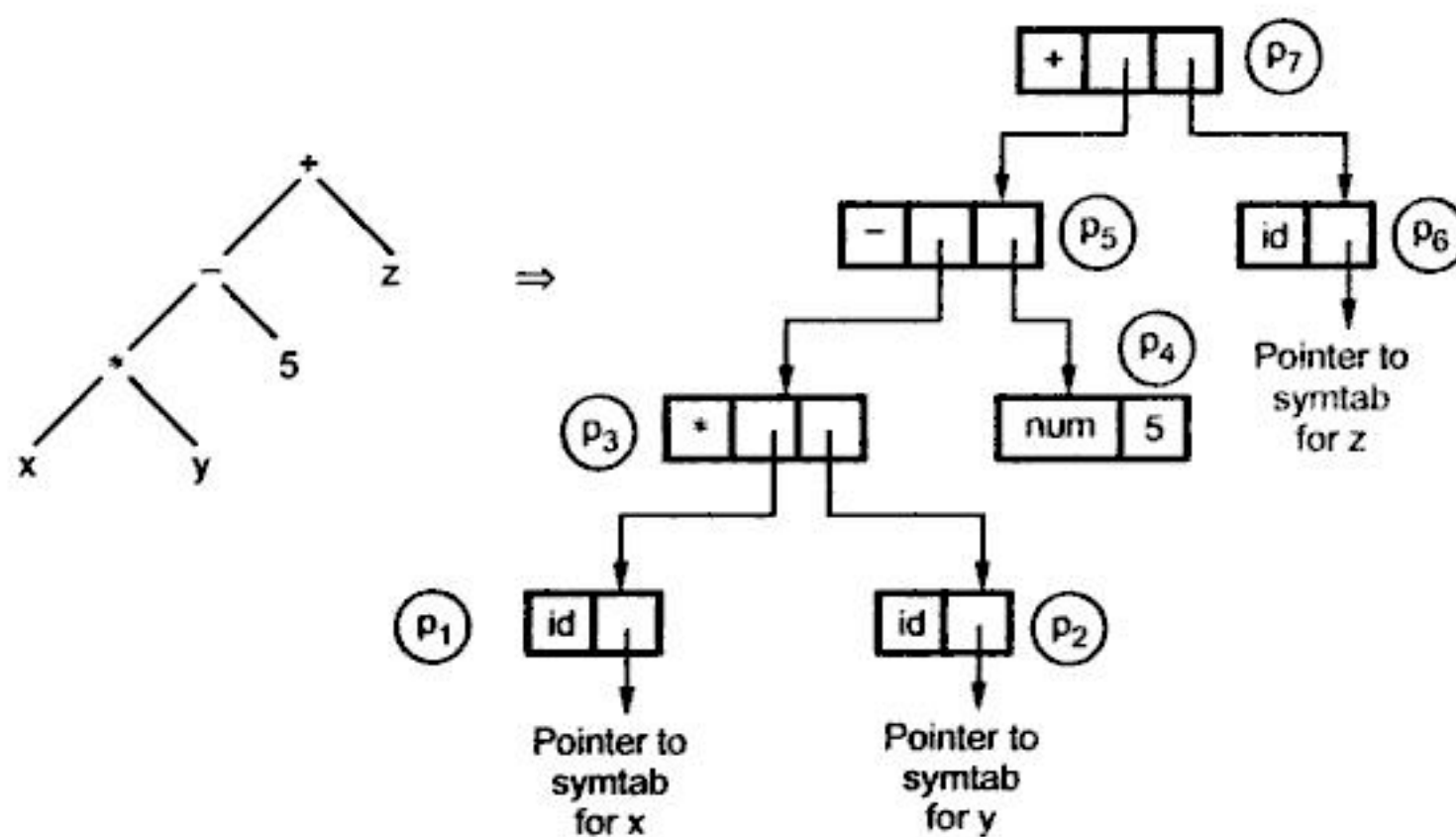


Fig. 6.7 Syntax tree

The syntax directed definition for the above grammar is as given below :

Production Rule	Semantic operation
$E \rightarrow E_1 + T$	$E.\text{nptr} = \text{mknnode}('+', E_1.\text{nptr}, T.\text{nptr})$
$E \rightarrow E_1 - T$	$E.\text{nptr} = \text{mknnode}('-', E_1.\text{nptr}, T.\text{nptr})$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

6.3 Bottom-Up Evaluation of S-Attributed Definitions

We have already discussed how to use syntax directed definitions to specify translations. Now in this section we will discuss how to implement syntax directed translation scheme for the syntax directed definitions. Hence a translator is built. The task of building translator for any arbitrary syntax directed definition is very difficult. However, to accomplish this task there are large classes of syntax directed definitions for which it is easy to construct translators.

- S-attributed definition is one such class of syntax directed definition with synthesized attributes only.
- Synthesized attributes can be evaluated using the bottom up parser.
- The purpose of stack is to keep track of values of the synthesized attributes associated with the grammar symbol on its stack. This stack is commonly known as parser stack.

6.3.1 Synthesized Attributes on the Parser Stack

1. A translator for S-attributed definition is implemented using LR parser generator.
2. A bottom up method is used to parse the input string.
3. A parser stack is used to hold the values of synthesized attribute.

The stack is implemented as a pair of state and value. Each state entry is the pointer to the LR (1) parsing table. There is no need to store the grammar symbol implicitly in the parser stack at the state entry. But for ease of understanding we will refer the state by unique grammar symbol that is been placed in the parser stack. Hence parser stack can be denoted as $stack[i]$. And $stack[i]$ is a combination of $state[i]$ and $value[i]$. For example, For the production rule $X \rightarrow ABC$ the stack can be as shown in Fig. 6.11.

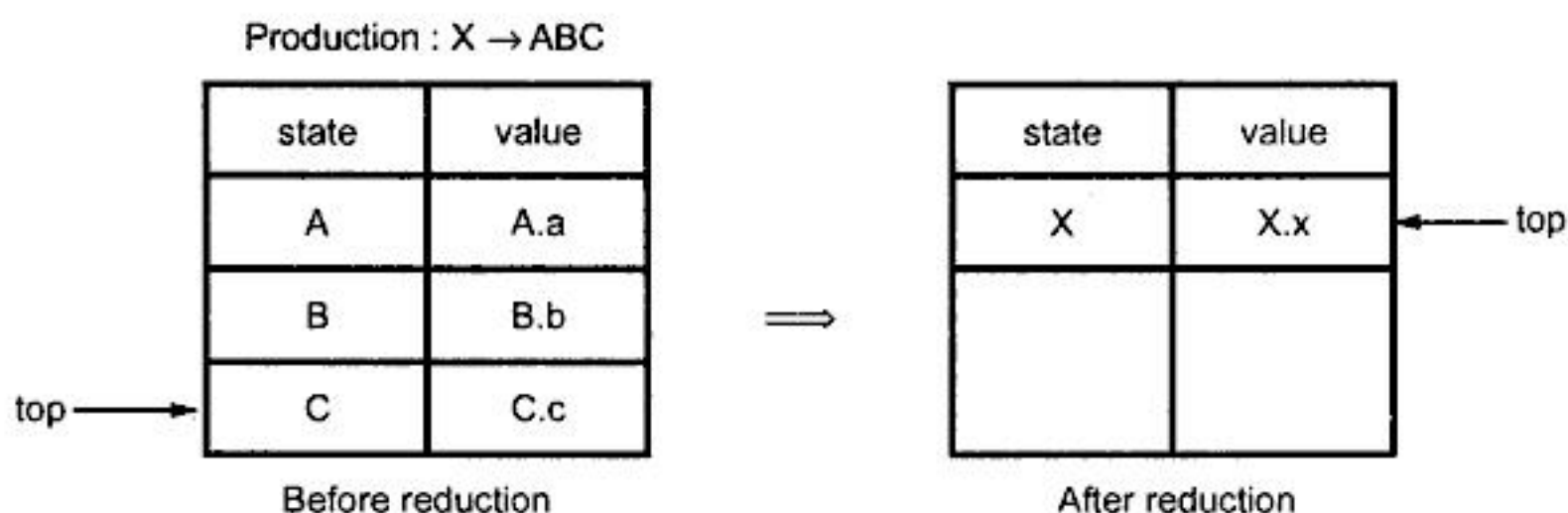


Fig. 6.11 Parser stack



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

6.4 L-attributed Definitions

The S- attributed definitions are for synthesized attributes only where as there is another class of attributes which is L-attribute.

The class of L-attributes can be evaluated in depth first order.

6.4.1 L-attributed Definition

The syntax directed definition can be defined as the L-attributed for the production rule $A \rightarrow X_1X_2...X_n$ where the inherited attribute X_k is such that $1 \leq k \leq n$. The production $A \rightarrow X_1X_2...X_n$ is such that

- 1) It depends upon the attributes of the symbol $X_1, X_2, ..., X_{i-1}$ to the left of X_i
- 2) It also depends upon the inherited attribute A.

Note that because of these two conditions every S-attributed definition is also L-attributed definition.

Let us discuss one example of L-attributed definitions.

➡ **Example 6.7 :** Check whether the given SDD (Syntax Directed Definition) is L-attributed or not.

$A \rightarrow PQ$	$P.in := p(A.in)$ $Q.in := q(P.sy)$ $A.sy := f(Q.sy)$
$A \rightarrow XY$	$Y.in := y(A.in)$ $X.in := x(Y.sy)$ $A.sy := f(X.sy)$

Solution :

The attributes **in** and **sy** represent the inherited and synthesized attributes respectively. The given syntax directed definition is not L-attributed definition.

Production	Semantic action	Class of attribute
$A \rightarrow PQ$	$P.in := p(A.in)$	L-attribute
	$Q.in := q(P.sy)$	L-attribute
	$A.sy := f(Q.sy)$	L-attribute
$A \rightarrow XY$	$Y.in := y(A.in)$	L-attribute
	$X.in := x(Y.sy)$	Not L-attribute
	$A.sy := f(X.sy)$	L-attribute

Because here value of left symbol (X) is dependent upon value of right symbol (i.e. Y)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

➡ **Example 6.9 :** The syntax directed definition for determining the value of arithmetic expression is as given below : Give the translation scheme and draw the annotated tree.

Production	Semantic action
$E \rightarrow E_1 + T$	$\{E.val := E_1.val + T.val\}$
$E \rightarrow E_1 - T$	$\{E.val := E_1.val - T.val\}$
$E \rightarrow T$	$\{E.val := T.val\}$
$T \rightarrow (E)$	$\{T.val := E.val\}$
$T \rightarrow \text{digit}$	$\{T.val := \text{digit.lexval}\}$

Solution : Now first remove the left recursion and rewrite the translation scheme for non left recursive grammar.

Production	Semantic rule
$E \rightarrow T$ P	$\{P.in := T.val\}$ $\{E.val := P.s\}$
$P \rightarrow +T$ P_1	$\{P_1.in := P.in + T.val\}$ $\{P.s := P_1.s\}$
$P \rightarrow -T$ P_1	$\{P_1.in := P.in - T.val\}$ $\{P.s := P_1.s\}$
$P \rightarrow \epsilon$	$\{P.s := P.in\}$
$T \rightarrow ($ E $)$	$\{T.val := E.val\}$
$T \rightarrow \text{digit}$	$\{T.val := \text{digit.lexval}\}$

Now the annotated parse tree can be drawn in Fig. 6.13.

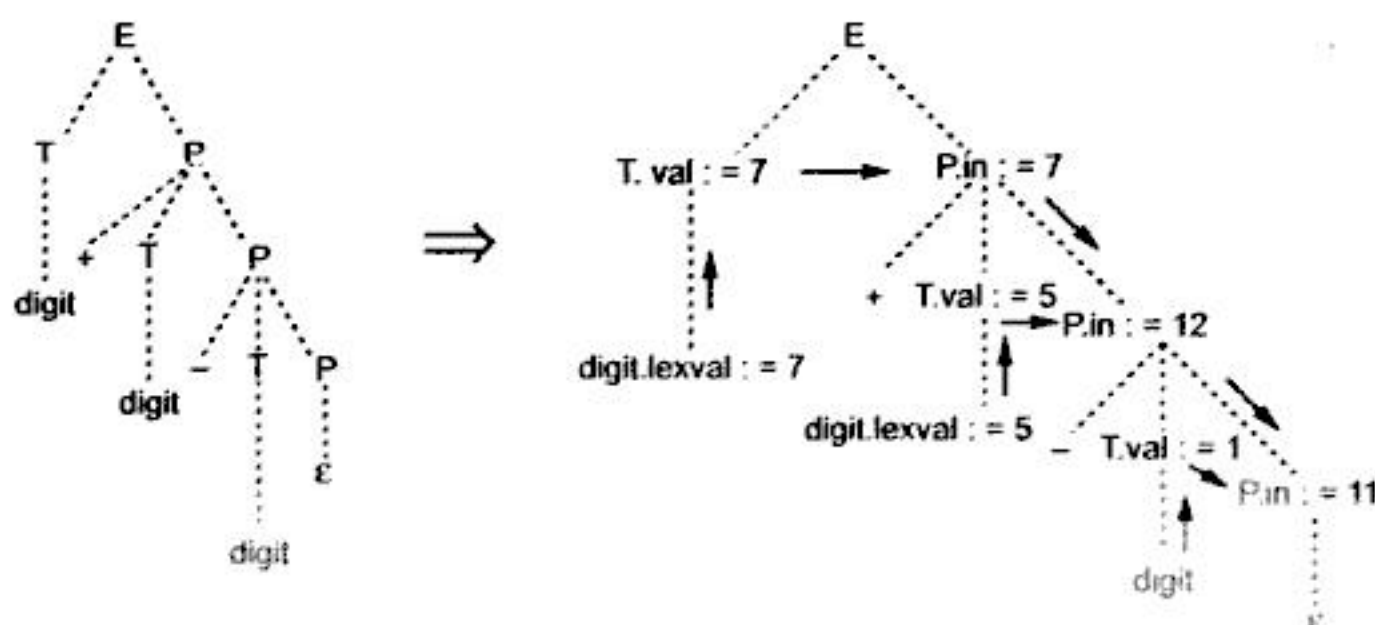


Fig. 6.13 Top down translation



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Following are the observations about the above implementation

- The value of List.in is always decided by T.type.
- The 'T' will always be below the 'List' in the parser stack and each time to decide what the value of List should be; value of T will be consulted.
- The 'T' will always be at the known position in the value array of parser stack.
- The function Enter_type is used to copy the corresponding type to the identifier. The first parameter to function Enter_type will be entry for id and second parameter will be type for that id. Hence it can be written as Enter_type(id.entry, T.type).

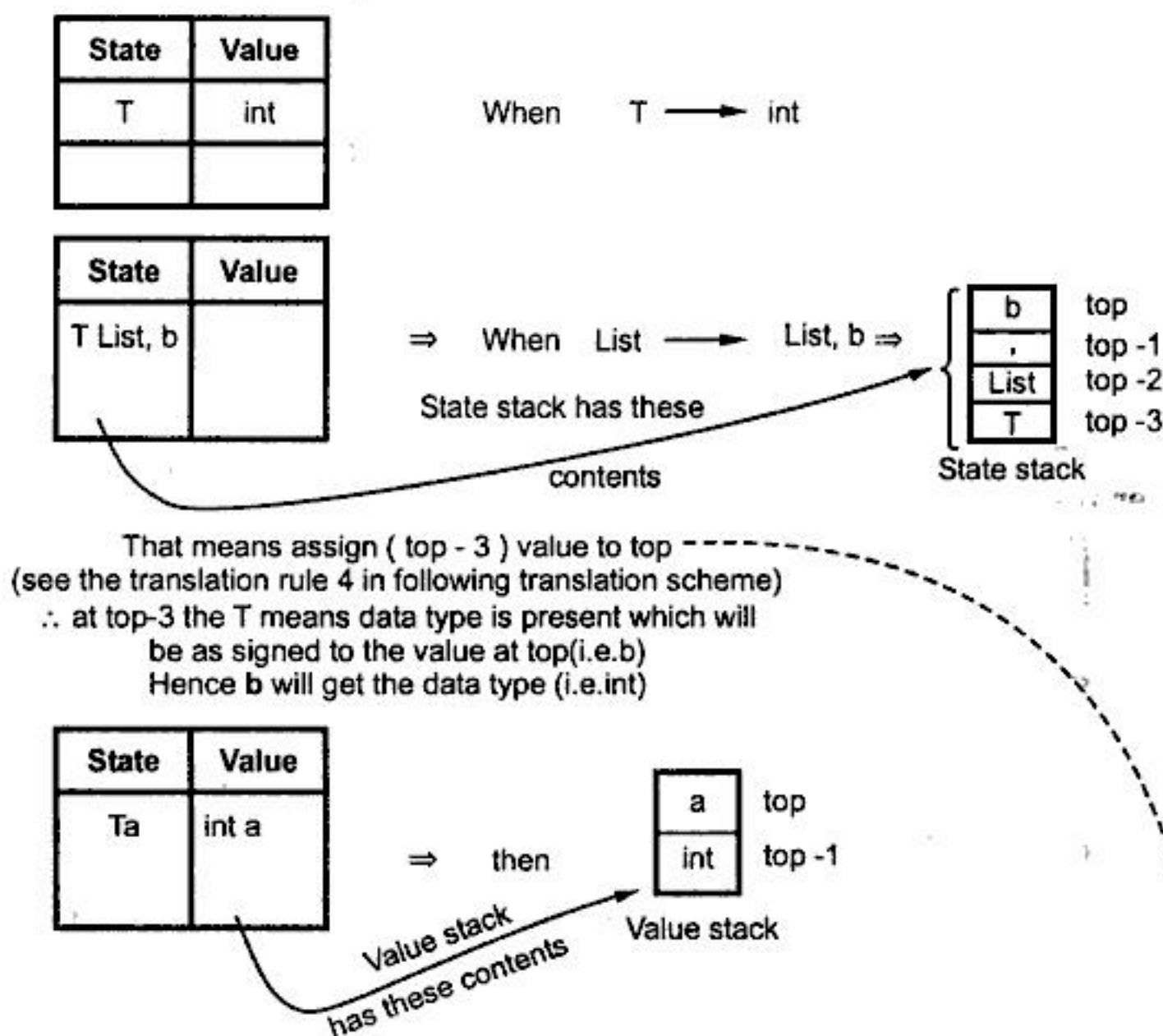


Fig. 6.15 Parser stack contents

The translation scheme can be written as follow :

Production Rule	Code fragment
$S \rightarrow T \text{ List};$	
$T \rightarrow \text{int}$	<code>value[top]:=int</code>
$T \rightarrow \text{float}$	<code>value[top]:=float</code>
$\text{List} \rightarrow \text{List, id}$	<code>Enter_type(value[top],value[top-3])</code>
$\text{List} \rightarrow \text{id}$	<code>Enter_type(value[top],value[top-1])</code>



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Q.5. Consider the following grammar that may be used to specify the syntax for declarative statement.

$D \rightarrow L : T$

$T \rightarrow \text{integer} \mid \text{char}$

$L \rightarrow L, \text{id} \mid \text{id}$

Show the parse tree for the following declarative statement

$A, B5, C3X : \text{Integer}$

Explain the difficulty in writing syntax directed definition to record the type of each variables in the symbol table. Modify the grammar so that one can write at least L - attributed definition.

Ans. : The parse tree for given grammar

$D \rightarrow L : T$

$T \rightarrow \text{Integer} \mid \text{char}$

$L \rightarrow L, \text{id} \mid \text{id}$

is as shown below for $A, B5, C3X : \text{Integer}$

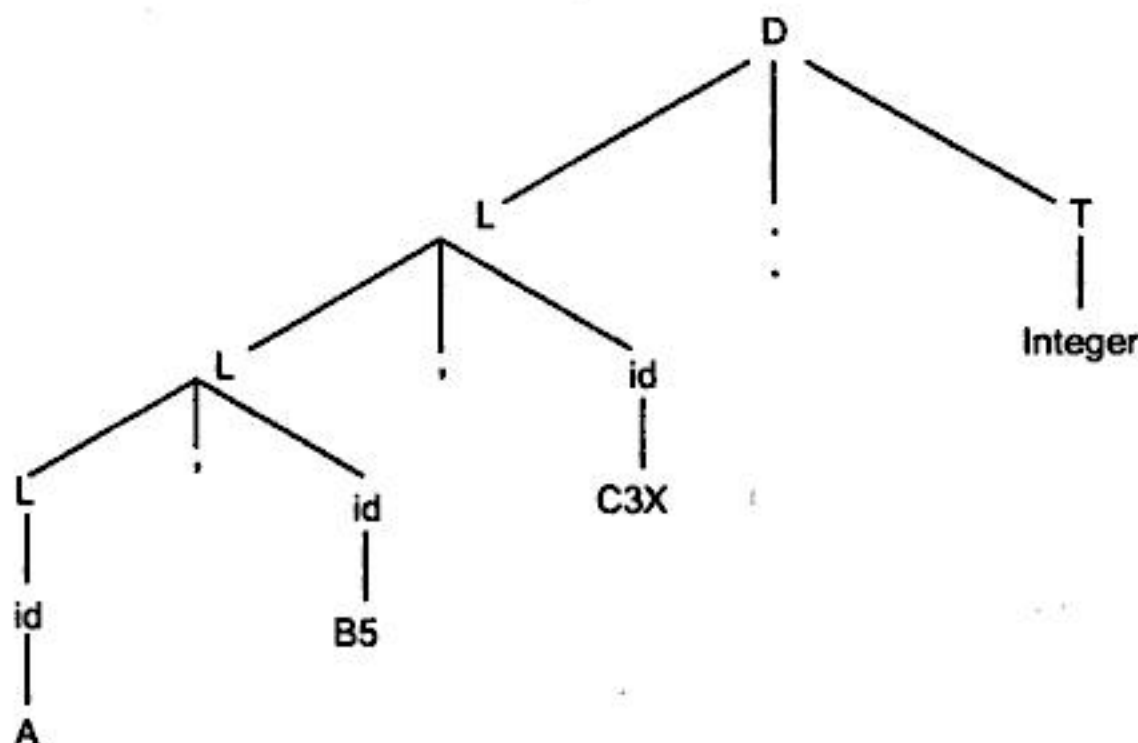


Fig. 6.18 Parse tree

Here we can generate all the identifiers from L but we can not associate the type with identifier using synthesized attributes only. We basically need to inherit type from T for L. From above grammar we get syntax directed definition that is not L - attributed. Hence the translation can not be done during parsing. Therefore we need to modify the grammar so that at least L-attributed definition is possible. The modified grammar is -

$D \rightarrow \text{id}L$

$L \rightarrow , \text{id}L \mid : T$

$T \rightarrow \text{integer} \mid \text{char}$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

➡ **Example 7.1 :** Obtain the translation scheme for obtaining the tree address code for the grammar -

$$S \rightarrow id := E$$

$$E \rightarrow E_1 + E_2$$

$$E \rightarrow E_1 * E_2$$

$$E \rightarrow - E_1$$

$$E \rightarrow (E_1)$$

$$E \rightarrow id$$

Solution : Here we will use the translation scheme which in turn will generate the three address code.

Production Rule	Semantic actions
$S \rightarrow id := E$	<pre> { id_entry:=look_up(id.name); if id_entry ≠ nil then append(id_entry ':=' E.place) else error; /* id not declared*/ }</pre>
$E \rightarrow E_1 + E_2$	<pre> { E.place:=newtemp(); append(E.place ':=' E₁.place '+' E₂.place) }</pre>
$E \rightarrow E_1 * E_2$	<pre> { E.place:=newtemp(); append(E.place ':=' E₁.place '*' E₂.place) }</pre>
$E \rightarrow - E_1$	<pre> { E.place := newtemp(); append(E.place ':=' 'uminus' E₁.place) }</pre>
$E \rightarrow (E_1)$	<pre> { E.place:=E₁.place }</pre>
$E \rightarrow id$	<pre> { id_entry:=look_up(id.name); if id_entry ≠ nil then append(id_entry ':=' E.place) else error; /* id not declared*/ }</pre>

- The look_up returns the entry for id.name in the symbol table if it exists there.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

7.5 Arrays

As we know array is a collection of contiguous storage of elements. For accessing any element of an array what we need is its address. For statically declared arrays it is possible to compute the relative address of each element. Typically there are two representations of arrays

1. Row major representation
2. Column major representation

These representations are as shown below

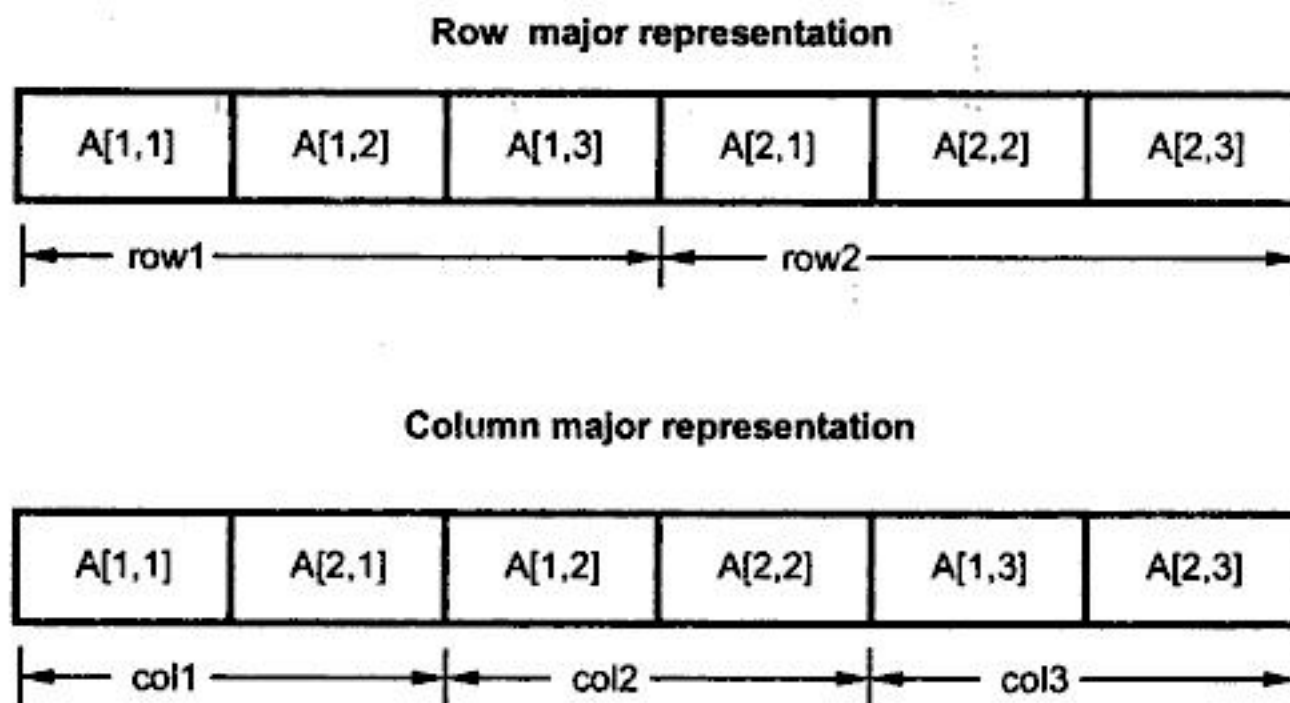


Fig. 7.2 Row major and column major representation

To compute the address of any element ;

Let base is the address at $a[]$ and w is the width of the element (required memory units) then to compute i th address of $a[]$

$$\text{base} + (i - \text{low}) \times W$$

where low is lower bound on subscript. Here $a[\text{low}] = \text{base}$

$$\text{base} + (i - \text{low}) w = \text{base} + i \times w - \text{low} \times w$$

$$= i \times w + (\text{base} - \text{low} \times w)$$

Let $c = \text{base} - \text{low} \times w$ is computed at compile time. Then the relative address of $a[i]$ can be computed as

$$c + (i \times w)$$

Similarly for calculation of relative address of two dimensional array we need to consider i and j subscripts. Considering row major representation we will compute the relative address for $a[i,j]$ using following formula

$$a[i,j] = \text{base} + ((i - \text{low}_1) \times n_2 + (j - \text{low}_2)) \times W$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- To represent the first term (of an expression for $a[i_1, i_2, \dots, i_k]$ i.e. $((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w$) L.offset is used and here L.offset is new temporary. Similarly the second term $base(\dots((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k \times w$ can be represented by L.place. The function $c(List.array)$ is used to obtain the L.place. The function $Size(List.array)$ is used to return the value of width w .
- The $L.offset = NULL$ indicates simple name of id.

In rule 7) the $List.ndim$ indicates number of index expressions (dimensions). Function $Limit(List_1.array, dim)$ returns n_j The n_j means number of elements along j^{th} dimension of array. The $E.place$ denotes the temporary value obtained by computing value from index expression.

- Consider formula $a[i_1, i_2, i_3 \dots i_k] = ((\dots((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w +$
 $base(\dots((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k \times w$

The List for expressions can be produced by

$$(\dots((i_1 n_2 + i_2) n_3 + i_3 \dots) n_m + i_m$$

Using recurrence relation

$$e_1 = i_1$$

$$e_m = e_{m-1} \ n_m + i_m$$

By rule 7) $List_1.place$ corresponds to e_{m-1} and $List.place$ corresponds to e_m

$$\therefore e_m = e_{m-1} \ n_m$$

$$List.place := List_1.place * Limit(List_1.array, dim)$$

- In rule 8) The value of E is finalized. Hence $E.place$ holds the value of expression E and value of E for having dimension $m=1$.

➡ **Example 7.3** : Generate the three address code for the expression $x := A[i, j]$ for an array 10×20 . Assume $low_1 = 1$ and $low_2 = 1$.

Solution : Given that

$$low_1 = 1 \text{ and } low_2 = 1$$

$$n_1 = 10, n_2 = 20$$

$$A[i, j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

$$A[i, j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4)$$

$$A[i, j] = 4 \times (20i + j) + (base - 84)$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In this three address code we have used goto to jump on some specific statement. This method of evaluation is called "short-circuit". The AND has higher precedence over OR hence at location 112 the ANDing is performed and then in the immediate next statement ORing is performed.

7.6.2 Flow of Control Statements

In this section we will discuss the translation of Boolean expression into three address code. The control statements are if-then-else and while-do. The grammar for such statements is as shown below

$S \rightarrow$ if E then S_1
 | if E then S_1 else S_2
 | while E do S_1

While generating three address code -

- To generate new symbolic label the function new_label() is used.
- With the expression E.true and E.false are the labels associated.
- S.code and E.code is for generating three address code.

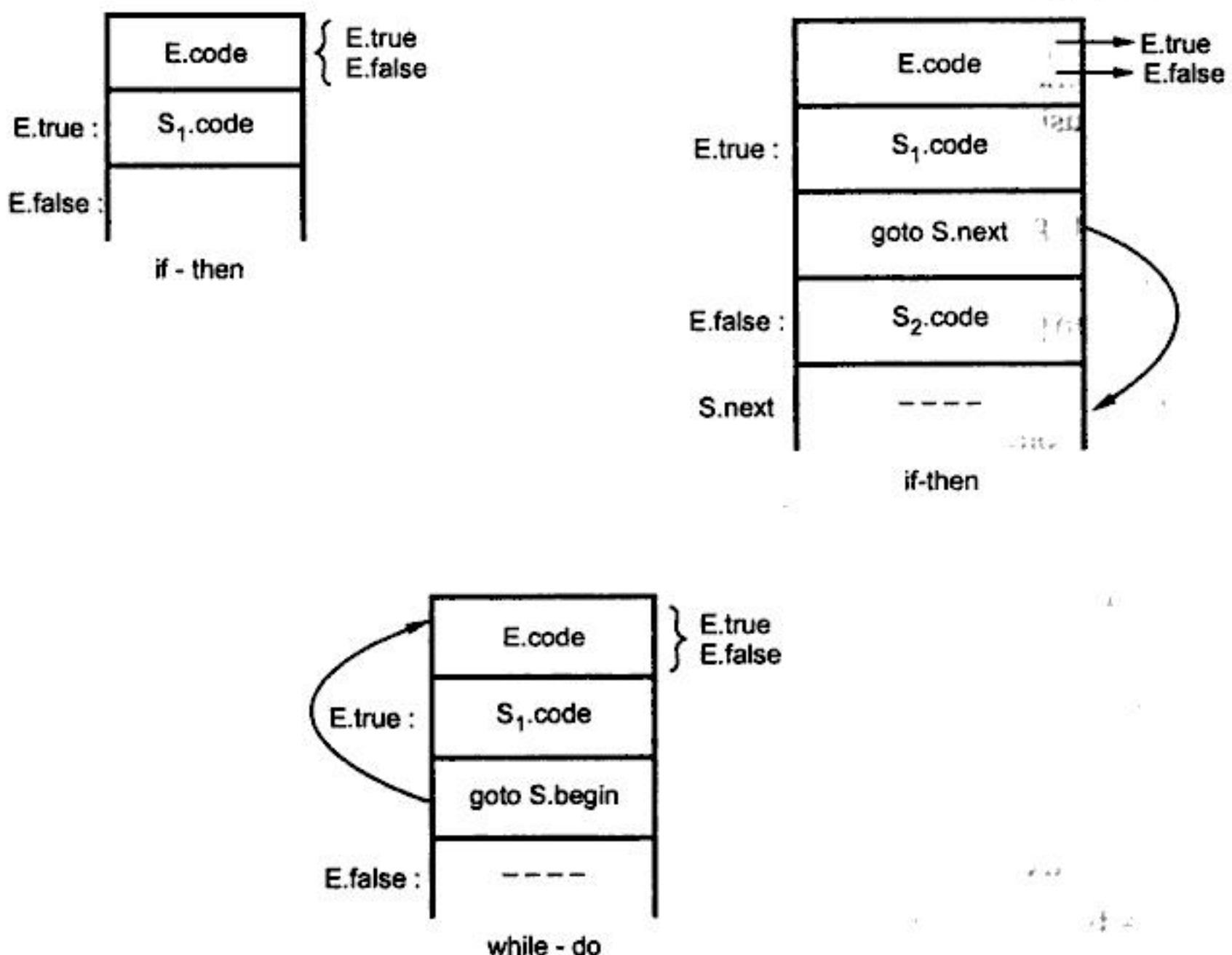


Fig. 7.4



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Solution : The three address code can be

```

100.      if ch = 1 goto L1
101.      if ch = 2 goto L2
102.      L1: t1:=a+b
103.      c:= t1
104.      goto Last
105.      L2: t1:=a-b
106.      c:= t1
107.      goto Last
108.      Last :
```

7.8 Procedure Calls

Procedure or function is an important programming construct which is used to obtain the modularity in the user program.

Consider the grammar for a simple procedure call

$$S \rightarrow \text{call id (L)}$$

$$L \rightarrow L, E$$

$$L \rightarrow E$$

Here the non terminal S denotes the statement and non terminal.

L denotes the list of parameters.

And E denotes the expression it could be id as well.

The translation scheme can be as given below –

Production Rule	Semantic Action
$S \rightarrow \text{call id (L)}$	{ for each item p in queue do append('param' p); append('call' id.place) }
$L \rightarrow L, E$	{ insert E.place in the queue }
$L \rightarrow E$	{ initialize queue and insert E.place in the queue }



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

7.9.2 Backpatching using Flow of Control Statements

The flow of control statements can be handled using backpatching techniques for the following grammar

$$\begin{aligned} S &\rightarrow \text{if } B \text{ then } S \\ S &\rightarrow \text{if } B \text{ then } S \text{ else } S \\ S &\rightarrow \text{while } B \text{ then do } S \\ S &\rightarrow \text{begin } L \text{ end} \\ S &\rightarrow A \\ L &\rightarrow L; S \\ L &\rightarrow S \end{aligned}$$

Where S stands for statement

B for Boolean expression

L stands for statement list

A stands for assignment statement

We will introduce the marker nonterminals to determine the exact point of semantic actions to take place. Hence the grammar with marker nonterminals at appropriate places becomes

$$\begin{aligned} S &\rightarrow \text{if } B \text{ then } M S_1 \\ S &\rightarrow \text{if } B \text{ then } M_1 S_1 N \text{ else } M_2 S_2 \\ S &\rightarrow \text{while } M_1 B \text{ then do } M_2 S \\ S &\rightarrow \text{begin } L \text{ end} \\ S &\rightarrow A \\ L &\rightarrow L_1; M S \\ L &\rightarrow S \\ M &\rightarrow \epsilon \\ N &\rightarrow \epsilon \end{aligned}$$

The translation scheme for the above grammar is as given below



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

THREE ADDRESS CODES

$$B := d / e$$

$$C := c * B$$

$$D := b + B$$

$$E := a = B$$

Quadruples

ID	OPERATOR	OPERAND1	OPERAND2	RESULT
(0)	/	d	e	B
(1)	*	c	B	C
(2)	+	b	B	D
(3)	=	a	B	E

[root@localhost]# ./a.out

Enter The Expression:a=(b)+(c*d)/e;

THREE ADDRESS CODES

$$B := c * d$$

$$C := B / e$$

$$D := b + B$$

$$E := a = B$$

Quadruples

ID	OPERATOR	OPERAND1	OPERAND2	RESULT
(0)	*	c	d	B
(1)	/	B	e	C
(2)	+	b	B	D
(3)	=	a	B	E

[root@localhost]#

Thus we have discussed how an intermediate code gets generated with the help of translation scheme. This intermediate code is of great use for generation of the target machine code. Compiler often makes use of intermediate code for optimizing the code. In the remaining chapters we will discuss the back end of the compiler and learn how exactly the target code gets generated.

Thus we have discussed how an intermediate code gets generated with the help of translation scheme. This intermediate code is of great use for generation of the target machine code. Compiler often makes use of intermediate code for optimizing the code.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Review Questions

1. Write SDT for generating 3 address code for arrays.
2. Give SDT for generating three address code for flow of control statements.
3. Explain the concept of back-patching.
4. Write Lex and YACC program to generate an intermediate code for assignment statements, if-else, for statements.
5. Write Quadruple, Triple and three address code for $(a+b) * (a+b) - (a+b) * d$
6. Translate the following piece of code into three address code

```
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
    {
        C[i][j]=0;
        for(k=0; k<n; k++)
            C[i][j]=C[i][j]+A[i][j]*B[i][j];
    }
}
```
7. Give an intermediate code for procedure call.
8. Translate the following into three address code -

```
int a, b;
a = 3;
b = a + 7;
```





You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- While inserting a new name we should ensure that it should not be already there. If it is already present there then another error occurs i.e. "Multiple defined Name".
- The advantage of list organization is that it takes minimum amount of space.

8.3.2 Symbol Table Organization using Self Organizing List

- This symbol table implementation is using linked list. A link field is added to each record.
- We search the records in the order pointed by the link of link field.
- A pointer "First" is maintained to point to first record of the symbol table.

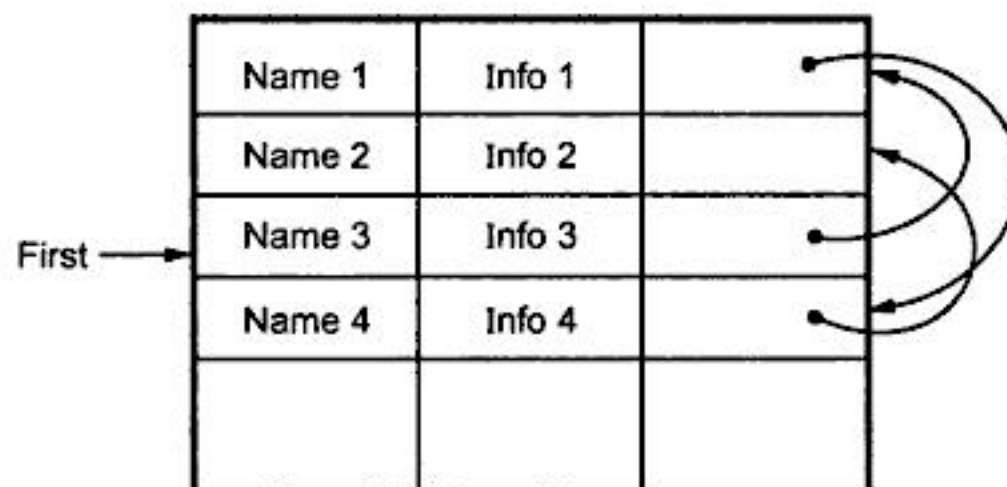


Fig. 8.3 Symbol table

The reference to these names can be Name3, Name1, Name4, Name2.

- When the name is referenced or created it is moved to the front of the list.
- The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

8.3.3 Hashing

- Hashing is an important techniques used to search the records of symbol table. This method is superior to list organization.
- In hashing scheme two tables are maintained a hash table and symbol table.
- The hash table consists of k entries from 0, 1 to $k - 1$. These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such that $h(\text{name})$ will result any integer between 0 to $k-1$. We can search any name by

$$\text{position} = h(\text{name})$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

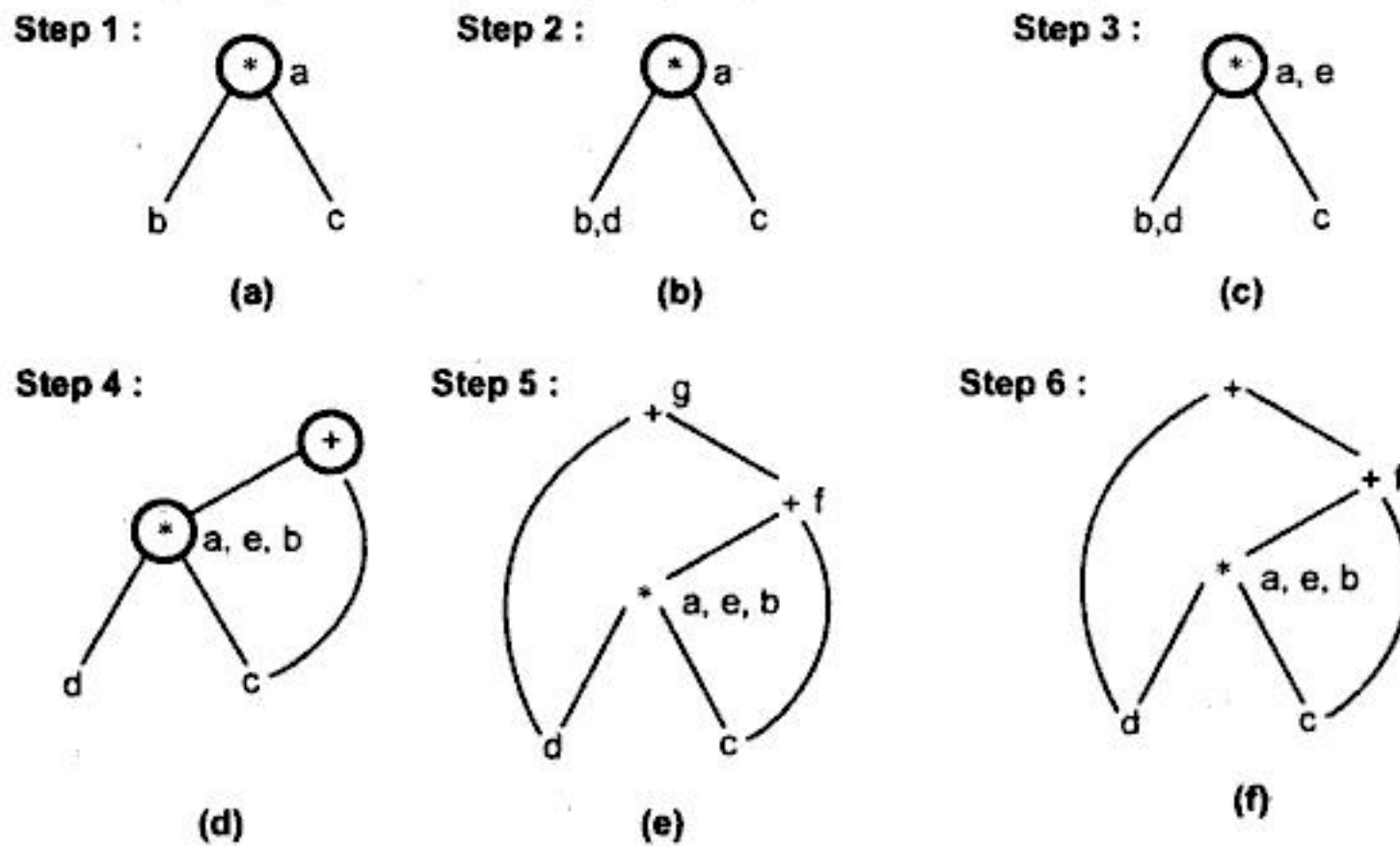


Fig. 9.7 Construction of DAG

The optimized code can be generated by traversing the DAG

The local optimization on the above block can be done

1. A common subexpression $e = d * c$ which is actually $b * c$ (since $d := b$) is eliminated
2. a dead code for $b = e$ is eliminated

The optimized code and basic block is

```

a := b * c
d := b
f := a + c
g := f + d
    
```

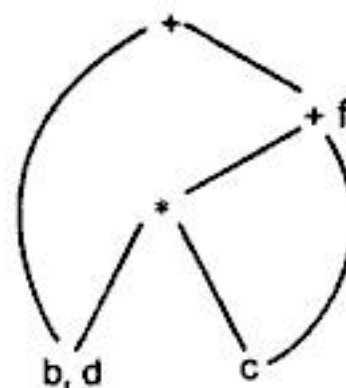


Fig. 9.8 Optimized code and DAG



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Node	At entry	At exit
2	0000	{b+c} = 1000
3	{b+c} = 1000	0000
4	{b+c} = 1000	{a-c} = 0010
5	{a-c} = 0010	{a-c} = 0010
6	0000	{a-c} = 0010
7	{c*d} = 0100	{c*d} = 0101 and {d+e} = 0001 i.e. 0101 && 0001
8	0101	{d+e} = 0001
9	{a-c} and {d+e} = 0011	0011

10.6 Iterative Data Flow Analysis

We can perform data flow analysis by computing data flow equations using either

- Available expression,
- Reaching definition or
- Using live variable analysis

The analysis made by available expression or reaching definition is called forward analysis and the analysis made by using live variable analysis is called backward analysis.

The basic operations that are used in this kind of analysis are

Logical AND or intersection

The intersection operation is implemented as logical AND. The truth table is given for the ease of computation.

A	B	A ^ B
0	0	0
0	1	0

For example
01000 ^ 11001 = 01000



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

After Pass 3 we get,

	in	out
1	{a, c}	a
2	{a, c}	{b, c}
3	{b, c}	b
4	b	a
5	{a, c}	{a, c}

→
→ } similar to pass 2 values
→

$$\text{out [1]} = \text{in [2]}$$

$$\text{out [1]} = \{a, c\}$$

$$\begin{aligned} \text{in [1]} &= \text{use [1]} \cup (\text{out [1]} - \text{def [1]}) \\ &= a \cup (\{a, c\} - a) \\ &= \{a, c\} \end{aligned}$$

$$\text{out [5]} = \text{in [6]} \cup \text{in [2]}$$

$$= c \cup \{a, c\}$$

$$\text{out [5]} = \{a, c\}$$

$$\begin{aligned} \text{in [5]} &= \text{use [5]} \cup (\text{out [5]} - \text{def [5]}) \\ &= a \cup (\{a, c\} - \phi) \\ &= \{a, c\} \end{aligned}$$

After pass 4 we get,

	in	out
1	{a, c}	{a, c}
5	{a, c}	{b, c}

→ Similar to pass 3 value

Now, $\text{out [1]} = \text{in [2]}$

$$\text{out [1]} = \{a, c\}$$

$$\begin{aligned} \text{in [1]} &= \text{use [1]} \cup (\text{out [1]} - \text{def [1]}) \\ &= a \cup (\{a, c\} - a) \end{aligned}$$

$$\text{in [1]} = \{a, c\}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

d) The optimizing compilers should apply following code improving transformations on source language.

- i) Common subexpression elimination.
- ii) Dead-code elimination.
- iii) Code movement.
- iv) Strength reduction.

Q. 5 : Given a piece of code as below optimize it as much as possible.

```

L1      if(a==b) goto L15
L2      r1=load(32)
L3      if(c<d) goto L14
L4      goto L14
L5      r1<<4
L6      goto L10
L7      store(r2,r7)
L8      r5 = r5 * 20
L9      r4 = r4+r4
L10     if(m>n) goto L4
L11     r3=load(20,r9)
L12     if(j!=k) goto L13
L13     r10 = 4*r10
L14     r11 = r11/8
L15     goto L12

```

Ans. : While optimizing the above given code some code may get modified and some dead code will get eliminated.

<pre> L1 if(a==b) goto L13 L2 r1 = load (32) L3 goto L14 L5 r1<<4 L6 if(m>n) goto L14 L6a goto L10 L7 store(r2,r7) L8 r5 = r5*20 L9 r4 = r4+r4 L10 if(m>n) goto L5 L11 r3=load(20,r9) L13 r10=4*r10 L14 r11=r11/8 L15 goto L13 </pre>	<p>→ As on L12 line it is directed to goto L13, we can directly state to jump to L13. Similarly L3 and L4 both lines tell us to jump at L14. Hence L3 is eliminated.</p> <p>→ The line L12 simply directs to jump to L13. Hence it is eliminated</p>
---	--



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$$\begin{aligned}
 \text{in } [6] &= \{x, y, z\} \\
 \text{out } [7] &= \text{in } [2] \rightarrow \text{obtained in pass 2 because we always consider} \\
 &\quad \text{recently computed previous values} \\
 \text{out } [7] &= \{m, x\} \\
 \text{in } [7] &= \text{use } [7] \cup (\text{out } [7] - \text{def } [7]) \\
 &= z \cup (\{m, x\} - m) \\
 &= z \cup \{x\} \rightarrow \text{cancelling } m \\
 \text{in } [7] &= \{x, z\} \\
 \text{out } [8] &= \text{in } [2] \rightarrow \text{value in pass 2} \\
 \text{out } [8] &= \{m, x\} \\
 \text{in } [8] &= \text{use } [8] \cup (\text{out } [8] - \text{def } [8]) \\
 &= y \cup (\{m, x\} - m) \\
 &= y \cup \{x\} \\
 \text{in } [8] &= \{x, y\}
 \end{aligned}$$

After pass 2 we get,

	in	out
1	{x}	{x}
2	{m, x}	{m, x}
3	{m, x}	{m, x}
5	{m, x}	{x}
6	{x, y, z}	{x, y, z}
7	{m, x}	{x, z}
8	{x, y}	{m, x}

Now we will perform computations for pass 3.

$$\begin{aligned}
 \text{out } [1] &= \text{in } [2] \rightarrow \text{obtained in pass 2} \\
 \text{out } [1] &= \{m, x\} \\
 \text{in } [1] &= \text{use } [1] \cup (\text{out } [1] - \text{def } [1]) \\
 &= \phi \cup (\{m, x\} - m)
 \end{aligned}$$



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Let us see an algorithm for elimination of induction variables.

Algorithm : Elimination of induction variables.

Input : A loop L with reaching definition information, loop invariant computation and live variable information.

Output : A flow graph without induction variables.

Method :

1) Find the induction variable i with triple (i, c, d) . Consider a test form -

if $i \text{ relop } x \text{ goto } B$ where i is an induction variable and x is not an induction variable,

$t := c * x$

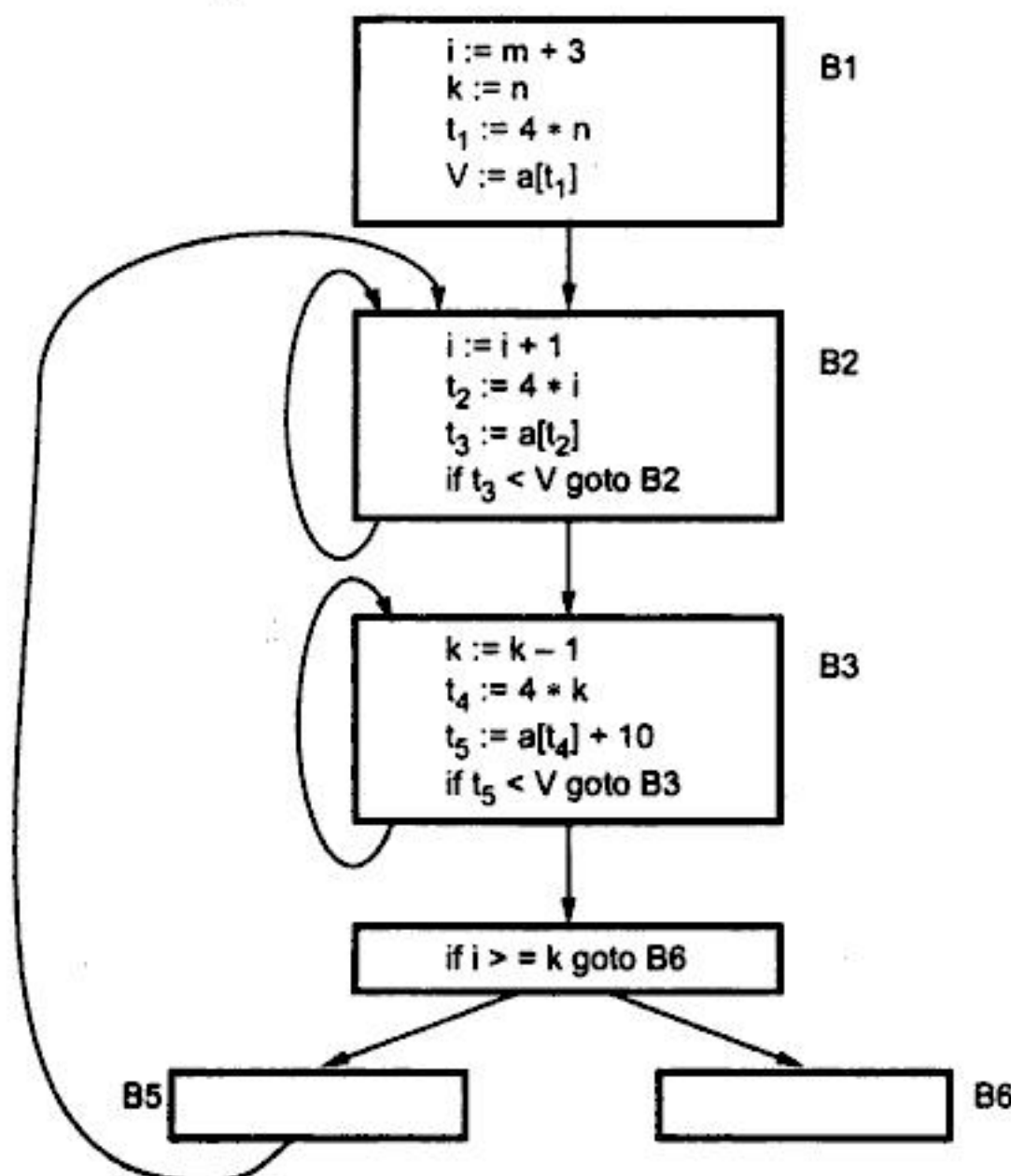
$t := t + d$

if $i \text{ relop } t \text{ goto } B$

where t is a new temporary.

Finally delete all assignments to the eliminated induction variables from the loop L because these induction variables will be useless.

For example :



Here i in B2 and k in B3 are two induction variables because their values get changed at each iteration. We will create new temporary variables r_1 and r_2 to which induction variables i and k are assigned.

Fig. 10.20 (a)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$\text{out [1]} = \text{in [2]}$
 $\text{out [1]} = \{a, c\}$
 $\text{in [1]} = \text{use [1]} \cup (\text{out [1]} - \text{def [1]})$
 $\quad = \phi \cup (\{a, c\} - a)$
 $\text{in [1]} = a$
 $\text{out [2]} = \text{in [3]}$
 $\text{out [2]} = \{b, c\}$
 $\text{in [2]} = \text{use [2]} \cup (\text{out [2]} - \text{def [2]})$
 $\quad = a \cup (\{b, c\} - b)$
 $\quad = a \cup \{c\}$
 $\text{in [2]} = \{a, c\}$
 $\text{out [3]} = \text{in [4]}$
 $\text{out [3]} = \{b\}$
 $\text{in [3]} = \text{use [3]} \cup (\text{out [3]} - \text{def [3]})$
 $\quad = \{b, c\} \cup (b - c)$
 $\text{in [3]} = \{b, c\}$
 $\text{out [4]} = \text{in [5]}$
 $\text{out [4]} = a$
 $\text{in [4]} = \text{use [4]} \cup (\text{out [4]} - \text{def [4]})$
 $\quad = b \cup (a - a) = b$
 $\text{out [5]} = \text{in [6]} \cup \text{in [2]}$
 $\quad = c \cup \{a, c\}$
 $\text{out [5]} = \{a, c\}$
 $\text{in [5]} = \text{use [5]} \cup (\text{out [5]} - \text{def [5]})$
 $\quad = a \cup (\{a, c\} - \phi)$
 $\text{in [5]} = \{a, c\}$

We are getting (pass 1 in [6]) = (pass 2 in [6]) so we will not recompute it. Finally we can summarize pass 3 values as :



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

selection of instructions. If we do not consider the efficiency of target code then the instruction selection becomes a straightforward task.

For each type of three address code the code skeleton can be prepared which ultimately gives the target code for the corresponding construct.

For example $x := a + b$ then the code sequence that can be generated as

MOV a,R0 /* loads the a to register R0*/

ADD b,R0 /* performs the addition of b to R0*/

MOV R0,x /* stores the contents of register R0 to x */

In the above example the code is generated line by line. Such a line by line code generation process generates the poor code because the redundancies can be achieved by subsequent lines and those redundancies can not be considered in the process of line by line code generation.

For example

$x := y + z$

$a := x + t$

The code for the above statements can be generated as follows :

MOV y, R0

ADD z, R0

MOV R0, x

MOV x, R0

ADD t, R0

MOV R0, a

The above generated code is a poor code because MOV R0,a is not used and statement MOV a,R0 is redundant. Hence the efficient code can be

MOV y, R0

ADD z, R0

ADD t, R0

MOV R0, a

The quality of generated code is decided by its speed and size. Simply line by line translation of three address code into target code leads to correct code but it can generate unacceptably non efficient target code.

- **Register allocation**

If the instruction contains register operands then such a use becomes shorter and faster than that of using operands in the memory. Hence while generating a good code efficient utilization of register is an important factor. There are two important activities done while using registers.

1. Register allocation – During register allocation, select appropriate set of variables that will reside in registers.
2. Register assignment – During register assignment, pick up the specific register in which corresponding variable will reside.

Contents

- Overview of Compilation : Phases of compilation - Lexical analysis, Regular grammar and regular expression for common programming language features, Pass and phases of translation, Interpretation, Bootstrapping, Data structures in compilation - LEX lexical analyzer generator.
- Top Down Parsing : Context free grammars, Top down parsing, Backtracking, LL (1), Recursive descent parsing, Predictive parsing, Preprocessing steps required for predictive parsing.
- Bottom up Parsing : Shift reduce parsing, LR and LALR parsing, Error recovery in parsing, Handling ambiguous grammar, YACC - automatic parser generator.
- Semantic Analysis : Intermediate forms of source programs - abstract syntax tree, Polish notation and three address codes, Attributed grammars, Syntax directed translation, Conversion of popular programming languages language constructs into intermediate code forms, Type checker.
- Symbol Tables : Symbol table format, Organization for block structures languages, Hashing, Tree structures representation of scope information. Block structures and non block structure storage allocation : Static, Runtime stack and heap storage allocation, Storage allocation for arrays, strings and records.
- Code Optimization : Consideration for optimization, Scope of optimization, Local optimization, Loop optimization, Frequency reduction, Folding, DAG representation.
- Data Flow Analysis : Flow graph, Data flow equation, Global optimization, Redundant subexpression elimination, Induction variable elements, Live variable analysis, Copy propagation.
- Object Code Generation : Object code forms, Machine dependent code optimization, Register allocation and assignment generic code generation algorithms, DAG for register allocation.



Technical Publications Pune

1, Amit Residency, 412 Shaniwar Peth, Pune - 411030, M.S., India.
Telefax : +91 (020) 24495496/97, Email : technical@vtubooks.com

Visit us at : www.vtubooks.com

First Edition - 2009

Price INR 220/-

ISBN 978-81-8431-344-4



Copyrighted material