T. Y. B. Sc.
COMPUTER SCIENCE
SEMESTER-VI

NEW SYLLABUS
CBCS PATTERN

COMPILER CONSTRUCTION

Dr. Ms. MANISHA BHARAMBE

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

SPPU New Syllabus

A Book Of

COMPILER CONSTRUCTION

For T.Y.B.Sc. Computer Science : Semester – VI

[Course Code CS 366 : Credits – 2]

CBCS Pattern

As Per New Syllabus

Dr. Ms. Manisha Bharambe

M.Sc. (Comp. Sci.), M.Phil. Ph.D. (Comp. Sci.)

Vice Principal, Associate Professor, Department of Computer Science

MES's Abasaheb Garware College

Karve Road, Pune

Price ₹ 300.00

NIRALI PRAKASHAN

ADVANCEMENT OF KNOWLEDGE

N5951

COMPILER CONSTRUCTION **ISBN 978-93-5451-309-1**

First Edition : January 2022

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan
(For orders outside Pune)
S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan
(For orders within Pune)
119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Tel : (020) 2445 2044; Mobile : 9657703145
Email : niralilocal@pragationline.com

MUMBAI

Nirali Prakashan
Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

DELHI

Nirali Prakashan
Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU

Nirali Prakashan
Maitri Ground Floor, Jaya Apartments,
No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003
Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR

Nirali Prakashan
Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR

Nirali Prakashan
438/2, Bhosale Plaza, Ground Floor,
Khasbag, Opp. Balgopal Talim,
Kolhapur 416 012 Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON

Nirali Prakashan
34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR

Nirali Prakashan
R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks

Preface ...

I take an opportunity to present this Text Book on **"Compiler Construction"** to the students of Third Year B.Sc. (Computer Science) Semester-VI as per the New Syllabus, June 2021.

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts. The book covers theory of Introduction to Compilers, Lexical Analysis (Scanner), Syntax Analysis (Parser) Syntax Directed Definition, Code Generation and Optimization.

A special word of thank to Shri. Dineshbhai Furia, and Mr. Jignesh Furia for showing full faith in me to write this text book. I also thank to Mr. Amar Salunkhe and Mr. Rahul Thorat of M/s Nirali Prakashan for their excellent co-operation.

I also thank Mrs. Yojana Despande, Mr. Ravindra Walodare, Mr. Sachin Shinde, Mr. Ashok Bodke, Mr. Moshin Sayyed and Mr. Nitin Thorat.

Although every care has been taken to check mistakes and misprints, any errors, omission and suggestions from teachers and students for the improvement of this text book shall be most welcome.

Author

Syllabus ...

1. **Introduction** **(4 Lectures)**
   - Definition of Compiler, Aspects of Compilation
   - The Structure of Compiler
   - Phases of Compiler:
     - Lexical Analysis
     - Syntax Analysis
     - Semantic Analysis
     - Intermediate Code Generation
     - Code Optimization
     - Code Generation
   - Error Handling
   - Introduction to One Pass and Multipass Compilers, Cross Compiler, Bootstrapping
2. **Lexical Analysis (Scanner)** **(4 Lectures)**
   - Review of Finite Automata as a Lexical Analyzer, Applications of Regular Expressions and Finite Automata (Lexical Analyzer, Searching using RE), Input Buffering, Recognition of Tokens
   - LEX: A Lexical Analyzer Generator (Simple Lex Program)
3. **Syntax Analysis (Parser)** **(14 Lectures)**
   - Definition, Types of Parsers
   - Top-Down Parser:
     - Top-Down Parsing with Backtracking: Method and Problems
     - Drawbacks of Top-Down Parsing with Backtracking
     - Elimination of Left Recursion (Direct and Indirect)
     - Need for Left Factoring and Examples
   - Recursive Descent Parsing:
     - Definition
     - Implementation of Recursive Descent Parser Using Recursive Procedures
   - Predictive [LL(1)] Parser (Definition, Model)
     - Implementation of Predictive Parser [LL(1)]
     - FIRST and FOLLOW
     - Construction of LL(1) Parsing Table
     - Parsing of a String using LL(1) Table
   - Bottom - Up Parsers
   - Operator Precedence Parser - Basic Concepts
   - Operator Precedence Relations form Associativity and Precedence
     - Operator Precedence Grammar
     - Algorithm for LEADING and TRAILING (with Examples)
     - Algorithm for Operator Precedence Parsing (with Examples)
     - Precedence Functions

- Shift Reduce Parser:
  - Reduction, Handle, Handle Pruning
  - Stack Implementation of Shift Reduce Parser (with Examples)
- LR Parser:
  - Model,
  - Types [SLR (1), Canonical LR, LALR] - Method and Examples
- YACC:
  - Program Sections
  - Simple YACC Program for Expression Evaluation

4. System Directed Definition (7 Lectures)

- Syntax Directed Definitions (SDD)
- Inherited and Synthesized Attributes
- Evaluating an SDD at the Nodes of a Parse Tree, Example
- Evaluation Orders for SDD's
- Dependency Graph
- Ordering the Evaluation of Attributes
- S - Attributed Definition
- L - Attributed Definition
- Application of SDT
- Construction of Syntax Trees
- The Structure of a Type
- Translation Schemes:
  - Definition
  - Postfix Translation Scheme

5. Code Generation and Optimization (7 Lectures)

- Compilation of Expression:
  - Concepts of Operand Descriptors and Register Descriptors with Example.
  - Intermediate Code for Expressions - Postfix Notations, Triples, Quadruples and Expression Trees
- Code Optimization:
  - Optimizing Transformations: Compile Time Evaluation, Elimination of Common Sub Expressions, Dead Code Elimination, Frequency Reduction, Strength Reduction
- Three Address Code
- DAG for Three Address Code
- The Value - Number Method for Constructing DAG's
- Definition of Basic Block, Basic Blocks and Flow Graphs
- Directed Acyclic Graph (DAG) representation of Basic Block
- Issues in Design of Code Generator

Contents ...

1. Introduction	**1.1 – 1.36**
2. Lexical Analysis (Scanner)	**2.1 – 2.38**
3. Syntax Analysis (Parser)	**3.1 – 3.142**
4. System Directed Definition	**4.1 – 4.34**
5. Code Generation and Optimization	**5.1 – 5.44**

CHAPTER 1

Introduction

Objectives...

- To study the Concept of Compiler
- To understand Structure and Phases of Compiler

1.0 INTRODUCTION

- Complier construction is truly an engineering science. With the science, we can methodically almost routinely design and implement fast, reliable, and power compliers.
- The name 'compiler' is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g. assembly language, object code or machine code) to create an executable program.
- A compiler is a translator which translates a program (the source program) written in one language to an equivalent program (the target program) written in another language (see Fig. 1.1).
- We call the languages in which the source and target programs are written the source and target languages, respectively.
- Typically, the source language is a high-level language in which humans can program comfortably (such as Java or C++), whereas the target language is the language the computer hardware can directly handle (machine language) or a symbolic form of it (assembly language).

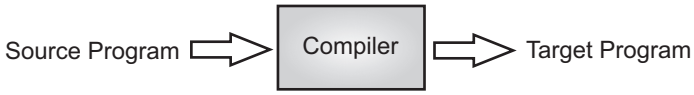


Fig. 1.1: Compiler

1.1 OVERVIEW OF COMPILER

- A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers.

- To understand importance of a compiler, let us take a typical language processing system as shown in Fig. 1.2.
- In addition to a compiler, many other programs are required to create a target program.
- The Fig. 1.2 shows the typical compilation. The preprocessor expands macros into source language statements. The output of compilation process i.e. target code may requires further processing before it can be run.
- The Fig. 1.2 shows that the compiler creates an assembly code which is translated into machine code by assembler.
- The linker is used to link some library routines and then the program actually runs. A loader loads all of them into memory and then the program is executed.

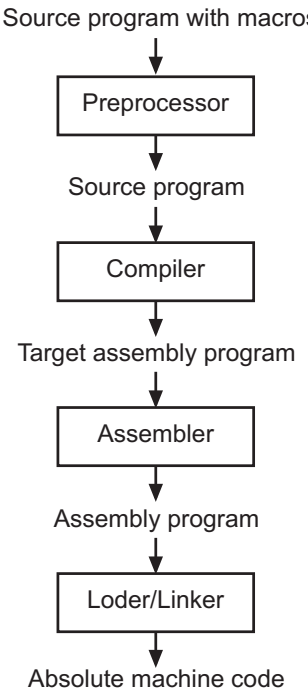


Fig. 1.2: A Language Processing System

1.1.1 Definition of Compiler

- A compiler is a program that reads a program written in one language - the source language and translates it into an equivalent program in another language - the target language.
- Hence, compiler is a translator which translate High Level Language (HLL) program into target language which is Low Level Language (LLL) or Machine Level Language (MLL).

- Fig. 1.3 shows compiler concept. The user can process the input and produced the output if the target program is executable machine language program as shown in Fig. 1.4.

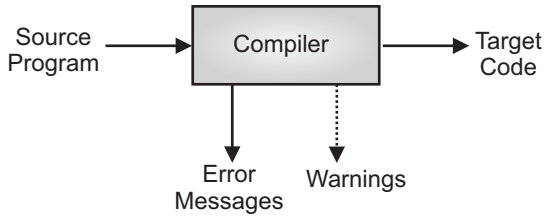


Fig. 1.3: Concept of Compiler

Fig. 1.4: Running the Target Code

- Compilers on UNIX platform convert HLL to LLL whereas compilers on DOS platform usually convert HLL to MLL (Machine Level Language).
- The process of compilation is much more complicated than assembler due to various features supported by HLL which are:
  1. **HLL Programs:** These programs are machine independent i.e. programs are portable whereas LLL programs are machine dependent.
  2. **Data Types:** HLL provides various data types float, double, string, etc. It is task of compiler to convert these data type into basic data types supported by machine e.g. byte, word, etc.
  3. **Data Structures:** HLL provides various data structure like arrays, records, files, etc. These data structures should be mapped to data structure supported by machine.
  4. **Control Structures:** HLL provides various control structures like for, while, repeat-until, etc.
  5. **Scope Rules:** Block structured language (like C, Pascal) allows nested definition of functions. HLL required additional data structure to store the information about variables.
  6. HLL provides runtime support e.g., **recursion and dynamic allocation**.

Advantages of Compiler:

- Source code is not included by compiler, therefore compiled code is more secure than interpreted code.
- Compiler produces an executable file and therefore the program can be run without need of the source code.

- The object program can be used whenever required without the need to of recompilation.

Disadvantages:

- When an error is found, the whole program (source code) has to be re-compiled again and again.
- Object code needs to be produced before a final executable file, this can be a slow process.

1.1.2 Concept of Interpreter

- An interpreter is a computer program that directly executes program instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.
- An interpreter accepts a source code as input and immediately executes it. An interpreter performs analysis of each statement in the source code to find its meaning and performs those specified operations, using the operating system and hardware platform on which it is based (see Fig. 1.5).
- An interpreter is a translation program that converts each high-level program statement into the corresponding machine code.
- Programming language like Python, BASIC and PERL use interpreters.

Fig. 1.5: Concept of Interpreter

- Interpreter execute source program statement by statement. The interpreter is a translator which takes source program as the input and produced output in MLL.
- An interpreter translates only one statement of the program (source code) at a time. It reads only one statement of program at a time, translates it and executes it.
- Then it reads the next statement of the program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed successfully.

Advantages of Interpreter:

1. If an error is found then there is no need to retranslate the whole program like compiler.
2. Debugging (check errors) is easier since the interpreter stops when it finds an error.
3. Easier to create multi-platform (run on different operating system) code, as each different platform would have an interpreter to run the same source code.

Disadvantages of Interpreter:

1. Source code is required for the program to be executed and this source code can be read by any other programmer so it is not a secured.
2. Interpreters are generally slower than compiled programs because interpreter translates one line at a time.

1.1.3 Aspects of Compilation [April 17]

- A compiler bridges the semantic gap between a programming language domain and an execution domain.
- The following are the two aspects of compilation:
  1. Compiler generated code which implements meaning of a source program in the execution domain.
  2. Compilation process diagnosis the wrong semantics of source programming language (PL) or source program.
- To implement these aspects, we discuss the following programming language features:
  1. Data type,
  2. Data structures,
  3. Scope rules,
  4. Control structure.

1.1.3.1 Data Type

- A data type is the specification of:
  1. Values for variables of the type (e.g., `int i, j`).
  2. Operations on the values of the type (e.g., `i = j + 10`).

  Here, operations of the type normally include an assignment operation and a set of data used for manipulation of operations.
- Semantics are checked by compiler to ensure that data type is valid data type or not. For semantic checking the following tasks are involved:

  Type Checking: Checking the validity of an operation for the types of its operands. For example, x = x + y; if x is integer and y is float, it checks the types of x and y.

  If the types of variables are not same, then compiler generates code to perform conversion of values, whenever necessary and conversion is performed according to programming language rules. Sometimes user has to explicitly perform type casting or sometimes implicit conversion is performed.

  Compiler semantics tasks are appropriate instruction sequences of the target machine to implement the operation of a type.

For example,

```
int i, j
                } declaration in 'C'
float x, y
y = 10;
x = y + i;
```

- In first statement, since y is float type, the compiler generates code to convert value '10' to the floating point representation.
- In second statement, the addition cannot perform straight way, since y and i types are different. Compiler must generate the code to convert i into float and then addition is perform as floating point operation.
- Checking the legality of each operation and determined the need for type conversion operations, the compiler must generate type specific code to implement the operation.

1.1.3.2 Data Structures

- Data structures like arrays, stacks, queues, records, lists, tables are declared in the HLL.
- The programmer can access parts of the data structures in terms of units meaningful to the programmers.
- The compiler allocates the storage (memory) to various elements of the data structure.

  For example, `int A[10];`

  In this example array A has allocated 10 memory locations to store 10 elements.
- Records and tables can contain heterogeneous data types, so storage mapping is more complicated in such structures.

For example,

```
struct employee
{
  char name [10];
  char sex;
  int id;
}
record [10];
```

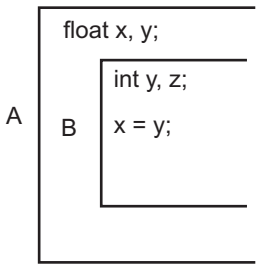
Here, record is array of structure employee, if the statement program is

```
record [i] . id = j
```

- Then there are two kinds of mapping involve. The first one is array reference i.e. record [i] and second is used to access field id within an element of record.
- The compiler must first decide how to represent different values of the type employee and then develop an appropriate mapping between values.

1.1.3.3 Scope Rules

- The scope of a program entity (for example, data item) is the point of the program where the entity is accessible.
- In most of the language the scope of the data items is restricted to the program block in which the data item is declared.
- For example, Variable x of block A is accessible in block A and in the enclosed block B. However, variable y of block A is not accessible in block B since y is re-declared in block B. Thus, statement x = y uses y of block B.



- The scope rules of a language determine which declaration of a name applies when the name appears in the text of a program.
- An occurrence of a name in a procedure is said to be local to the procedure if it is in the scope of a declaration within the procedure, otherwise, the occurrence is called nonlocal.
- During compilation, a symbol table is used to find the declarations that applies to an occurrence of a name.

1.1.3.4 Control Structure

- The control structure of a language is a collection of language facilities for sequencing, altering the flow of control during the execution of programs.
- It includes features like conditional or non-conditional transfer, conditional execution, iteration control and procedure calls.
- The compiler must check that a source program should written accordingly the rules of control structures.

  For example,

1. `for (i = 0; i < j; i++)`
   `{ ---------- }`
2. `if condition then statement`
   `else statement`
3. `do`
   `{ ---------- }`
   `while condition`

1.2 STRUCTURE OF COMPILER

- Compilation or translation is the process translation of source code into target code by a compiler. There are two parts/tasks of compilation i.e., analysis and synthesis.

- The analysis of the input source program and the synthesis of the executable target code as shown in Fig. 1.6.

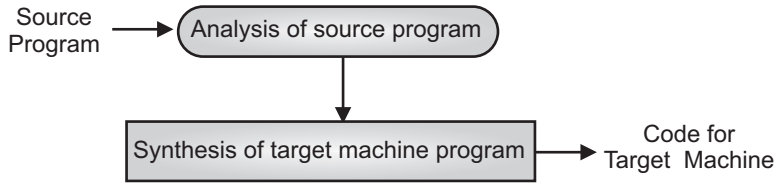


Fig. 1.6: The Analysis and Synthesis Views of Compilation

- The process is known as the analysis and synthesis model of compilation as shown in Fig. 1.6. The structure of a compiler is composed of mapping of analysis and synthesis parts.
  - The **analysis part** breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
  - The **synthesis part** constructs the desired target program from the intermediate representation.
- The fundamental language-processing model for compilation consists of a two-step processing of a source program. These two steps are given below:
  1. Analysis of source program.
  2. Synthesis of the target program.
- The Analysis part known as the front-end of the compiler, which reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.

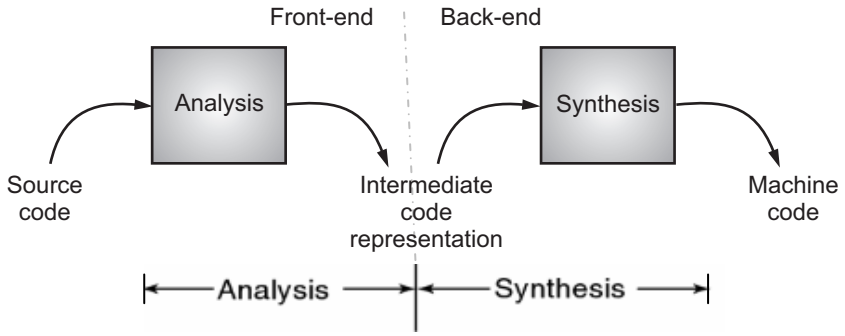


Fig. 1.7: Compilation Process of Compiler/Structure of Compiler

- The Analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.
- The analysis part collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.
- The Analysis part/phase consists of three phases i.e., Lexical Analysis, Syntax Analysis and Semantic Analysis.

- The Synthesis part known as the back-end of the compiler, which generates the target program with the help of intermediate source code representation and symbol table.
- The synthesis phase consists of two phases i.e., Code Optimization and Code Generation.
- A compiler can have many phases and passes. A pass refers to the traversal of a compiler through the entire program.
- A phase of a compiler is a distinguishable stage/step, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

1.3 PHASES OF COMPILER [April 16, Oct. 17, 18]

- A compiler operates in phases. Each individual unique step in compilation process is called as phase. The compilation process is a sequence of various phases.
- A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.
- Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.
- The structure of compiler comprises various phases as shown in Fig. 1.8.

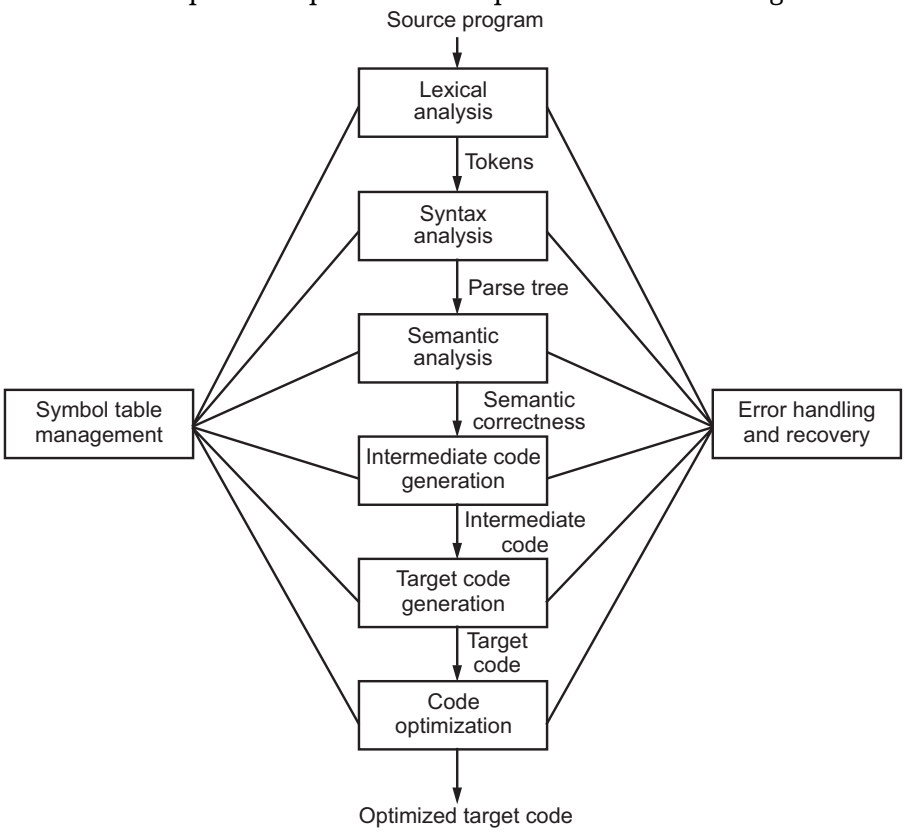


Fig. 1.8: Phases of a Compiler

- The design/structure of compiler can be decomposed into following phases:

1. Lexical Analysis Phase:

- Lexical analysis (also known as scanning) is the first phase of a compiler. Lexical analyzer or scanner reads the source program in the form of character stream and groups the logically related characters together that are known as lexemes.
- Lexical analysis phase scans the source code as a stream of characters and converts it into meaningful lexemes.
- For each lexeme, a token is generated by the lexical analyzer. A stream of tokens is generated as the output of the lexical analysis phase, which acts as an input for the syntax analysis phase.
- Tokens can be of different types, namely, keywords, identifiers, constants, punctuation symbols, operator symbols, etc.

2. Syntax Analysis Phase:

- The second phase of compiler is called the syntax analysis or parsing. Syntax analysis (Parser) processes output of the scanner, detects syntactic constructs and types of statements, generates a parse tree.
- The syntax analysis phase takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).
- In token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

3. Semantic Analysis Phase:

- Semantic analysis (mapper) processes the parse trees, detects the "meaning" of statements and generates an intermediate code.
- Semantic analysis phase checks whether the parse tree constructed follows the rules of language.
- For example, assignment of values is between compatible data types, and adding string to an integer.
- Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc.
- The semantic analyzer produces an annotated syntax tree as an output.

4. Intermediate Code Generation Phase:

- After semantic analysis phase the compiler generates an intermediate code of the source code for the target machine.
- In intermediate code generation phase, the parse tree representation of the source code is converted into low-level or machine-like intermediate representation.
- The intermediate code should be easy to generate and easy to translate into machine language.

- The intermediate code generation phase of compiler, represents a program for some abstract machine. It is in between the high-level language and the machine language.
- This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

5. Code Optimization Phase:

- The next phase of compiler does code optimization of the intermediate code. Code optimization phase, which is an optional phase, performs the optimization of the intermediate code.
- Optimization means making the code shorter and less complex, so that it can execute faster and takes lesser space.
- Optimization can be assumed as something that removes unnecessary code lines and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

6. Code Generation Phase:

- Code generation phase, translates the intermediate code representation of the source program into the target language program.
- In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.
- The code generator phase of compiler, translates the intermediate code into a sequence of (generally) re-locatable machine code.
- Sequence of instructions of machine code performs the task/action as the intermediate code would do.

- All of the above phases involve the two tasks namely, Symbol table management and Error handling.

1. Symbol Table Management:

- A symbol table is a data structure maintained throughout all the phases of a compiler.
- A symbol table is used by the compiler to record and collect information about source program constructs like variable names and all of its attributes, which provide information about the storage space occupied by a variable (name, type and scope of the variables).
- All the identifier's or variable’s names along with their types are stored in the symbol table.
- The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.

2. **Error Handling**
   - Each phase of compiler can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
   - Error handler is invoked whenever any fault occurs in the compilation process of source program.

- Let us see the phases of compiler in detail.

1.3.1 Lexical Analysis

- Lexical analysis or scanning is the first phase of compiler. The program which does lexical analysis is called scanner.
- A scanner is also called as lexical analyzer or linear analysis. The lexical analyzer is the interface between the source program and the complier.
- The scanner scans the input program character by character and groups the character into the lexical units called tokens.
- In simple words, a token is a sequence of characters that represent lexical unit. Each token has type and value.
- The token name is an abstract symbol that represents a kind of lexical unit and the optional attribute value is commonly referred to as token value.
- Each token represents a sequence of characters that can be treated as a single entity. Tokens can be identifiers, keywords, constants, operators, and punctuation symbols such as commas and parenthesis.
- The stream of tokens is called as descriptors. Token is made up of sequence of characters and is treated as one logical entity.
- Tokens are symbolic names for the entities that make up the text of the program e.g. if is used for the keyword if and id is used for any identifier. Those make up the output of the lexical analysis.
- Fig. 1.9 shows use of scanner.

Fig. 1.9: Use of Scanner

- Table 1.1 shows token, toke type and its value.

Table 1.1

Sr. No.	Token	Token type	Value
1.	Max, a, b	Identifier	Pointer to symbol table
2.	20, 56	Number (constant)	Value of number
3.	for, else	Reserve word	Number is given to each reserve word
4.	+, *, >	Operator	ASCII value of operator
5.	;, $, #, ·	Special character	ASCII value of character

- For example, `a := b + c`. In this statement tokens are – identifier, operator, identifier, operator and identifier.

Basic Task of Scanner:

1. Basic task of scanner is to group the characters from input statement into tokens (lexical units) and determines its type and value.
2. Scanner scans character by character. It should know where to stop when token is found. It will stop when ♭, special characters i.e. delimiters are occurs.
3. The lexical units of a sentence are generally isolated by looking for a set of special symbols known as delimiters. So with the help of delimiters scanner finds where the first token over.

   For example, `sum : = sum + 1`. In this statement, : is a delimiter which marks the end of identifier sum.
4. Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes (an instance of a token i.e., group of characters forming a token) which produces token as output.

Auxiliary Task of Scanner:

1. **To know the format of the language:** Scanner must know the format of statement, namely, the identification of various fields on the card, statement end, continuation on another card, e.g. In cobol some statements start with 8^{th} column and others with 12^{th} column. The format of a statement as coded by the programmer, has to be approximately processed by the scanner.
2. **Removing the comments from program:** This is auxiliary task necessary for coding rules. Comment can occur anywhere in program. So syntax analysis is difficult because every time parser (next phase) has to check whether statement is comment or not. Hence, parser is complex. To simplify parser or syntax analyser scanner removes the comment and tokens are not generated for these comments, so further overheads of parser are reduced.
3. **Buffering and LOOKAHEAD:** For a language like Fortran, only delimiters are not sufficient to form a token due to certain rules of the language. Before compilation, program is given to preprocessor which removes the blank, so job of scanner becomes more difficult. For example, consider the FORTRAN statement.

   DO `10 i = 1, 5`

   When spaces are removed, '=' appears to be the first delimiter in the statement, and **DO10i** the first lexical unit in it. However, this is not so. Appearance of ',' after 1 makes this statement a valid DO statement and invalid assignment statement. Hence *DO* is a reserve word, *10* is a label and *i* is the identifier. There are three lexical units in the statement. This means that the lexical analysis should be performed only after statement category is known. (In this ',' says the category of

statement is a DO statement). To know the category of statement, scanner makes the use of buffer and a look-ahead pointer. Scanner stores the complete statement in buffer.

```
DO  10  I   =   1,  10
↑                   ↑____ look-ahead pointer
```

Beginning of token

If look ahead pointer finds the comma, then DO statement is a valid loop else assignment statement.

4. **Error indication and symbol table management :** Errors like invalid identifier etc. are indicated. Various tables are built in lexical analysis phase, each table containing all instances of particular syntax category. For example, a table of variables (symtab), table of constants etc.

Use of Finite Automata (FA) in Scanning:

- To know the token is valid token or not the scanner should know rules for all types of tokens. To represent rules for tokens, we can make the use of regular expression.
- Finite Automata (FA) is a state machine that takes a string of symbols as input and changes its state accordingly. A finite automaton is a recognizer for regular expressions.
- Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. Any finite sequence of alphabets (characters) is called a string.
- The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.
- Example,
  1. d^+ is regular expression for number (integer) where d stands for digit.
  2. Similarly, regular expression for identifier will be $l \cdot (l + d)^*$ where,
     $l \rightarrow$ letter and $d \rightarrow$ digit. [Here, '*' stands for zero or more occurrences of symbol, '+' stands for one or more occurrences of symbol].
  3. Regular expression for reserve word 'begin' can be written as B.E.G.I.N.
- Regular Expressions (REs) are written in such a way for each token. This regular expression can be converted into DFA (Deterministic Finite Automata).
- This DFA is converted into State Transition Table (STT) which is used by the scanner to find a token.
- For example, consider regular expression of identifier $l \cdot (l + d)^*$ or letter (letter + digit)*.
- A deterministic finite automaton of identifier is shown in Fig. 1.10.

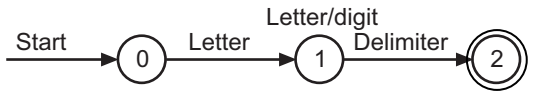


Fig. 1.10: DFA

- In this DFA, state 0 is the start state and state 2 is final state indicated with double circle. If character other than letter or digit then state transition is from state (1) to state (2) and state (2) recognize the token is identifier.
- Thus, looking at final state we can tell which token type it is. This DFA is converted into STT which is as follows:

	Inputs		
State	**Letter**	**Digit**	**Other**
0	1		
1	1	1	2 (Accept)
2			

- All blank entire's indicates error state. This type of transition table can be easily put into a program and a driver can be drive further.
- Utility like Lex (a program that generates lexical analyzer) in UNIX take input as regular expression and generates a transition table for a given regular expression.
- Also a driver is present to use this table. Writing complex on UNIX is comparatively easy.

1.3.2 Syntax Analysis (Parsing)

- Syntax analysis is the second phase of compiler which is also called as parsing or hierarchical analysis.
- In parsing the syntax analysis process the string of descriptor synthesized by the lexical analyzer to determine the syntactic structure of an input statement.
- The syntax analysis phase takes words/tokens from the lexical analyzer and checks the syntactic correctness of the input program.
- Output of the parsing step is representation of the syntactic structure of a statement. The conventional representation is in the form of syntax tree.
- Parsing takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).
- Syntax analyzer groups tokens together into syntactic structures, such as expression, statement and procedure, outputs a syntax tree in which its leaves are the tokens and every non-leaf node represents a syntax class type.
- The syntax analyzer also determines the structure of the source string, which may be represented by syntax trees.

- It basically involves grouping of the statements into grammatical phrases that are used by the compiler to generate the output finally. The grammatical phrases of the source program are usually represented by a parse tree.
- Parser is a program usually part of complier, converts the tokens produced by lexical analyzer into a tree like representation called parse tree.
- A parse tree is a structural representation of the input being parsed. A parse tree is a graphical representation a derivation.
- A derivation basically a sequence of production rules, in order to get the input string. Derivation is used to find whether the string belongs to a given grammar.
- A syntax analyzer takes the input from a lexical analyzer in the form of token streams. It analyses the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.
- For example, `a = b + i` can be represented in syntax tree form as shown in Fig. 1.11.

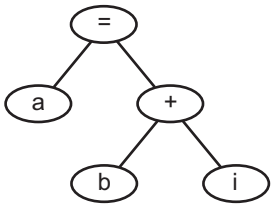


Fig. 1.11: Syntax Tree/Parse Tree

1.3.3 Semantic Analysis

- Semantic analysis is the third phase of compiler which checks whether the parse tree (or syntax tree) follows the rules of language.
- Processing performed by the semantic analysis step can be classified into:
  1. Processing of declarative statements.
  2. Processing of imperative statements.
- During semantic processing of declarative statements, items of information are added to the various lexical tables.
- For examples, type, length, dimensionality of variables and also accessing information like whether procedure parameter or global variables etc.
- While processing executable statements (imperative statements), information from lexical tables is used to determine semantic validity of a construct.
- Semantic analysis, analyze the syntax tree to identify external evaluations, and then apply rules of validity to the elemental evaluations.
- In this phase, actual analysis is done to determine meaning of statement. The meaning can be determined only if statement is syntactically correct.
- For example, Fig. 1.12 shows the syntax tree of statement a = b + i with data types.

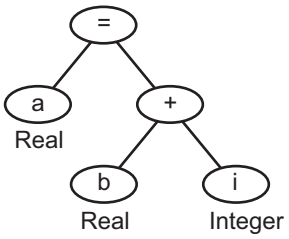


Fig. 1.12: Syntax Tree of Statement a = b + i

- In above example, the statement is semantically wrong (data type mismatch). So different steps which will take place are:
  1. Convert i to real.
  2. Add to b to get result T (say).

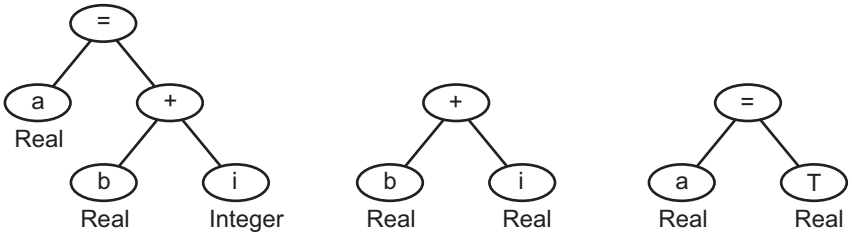


Fig. 1.13

- Semantic analyzer (Mapper) converts or maps syntax trees for each construct into a sequence of intermediate language statements. Some compilers have the ability to do such conversion automatically.

1.3.4 Intermediate Code Generation

- Lexical analysis, syntax analysis and semantic analysis are front-end phases. At the end of these phases intermediate code is generated.
- In the intermediate code generation, compiler generates the source code into the intermediate code.
- Intermediate code is generated between the high-level language and the machine language.
- The codes are easily optimized and also easily translated into the target program. It is easier to produce this code.

Properties of Intermediate Code:

1. It should be easy to produce.
2. It should be easy to translate into target program.

- One of the intermediate code which is used in many compilers is three address code. This code has almost three operands and it consists of sequence of instructions.

  For example, `a := b + c`

- Three address code is,

```
tempi := b + c
temp2 := temp 1
a := temp 2
```

1.3.5 Code Optimization

- Code optimization is aimed at improving the execution efficiency of a program. It is back-end phase of compiler which depends only on target code.
- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- Code optimization is used to improve the intermediate code so that the output of the program could run faster and take less space.
- Efficiency is achieved by:
  1. Rearranging compilation in a program so as to gain an improvement in execution speed without changing meaning of program.
  2. By exploiting peculiarities of the target machine by using appropriate code generation strategies.
- The Fig. 1.14 shows the optimizing compiler.
- The code optimization depends upon the intermediate representation of the source code. To improve the efficiency of program an optimizing transformation is needed.

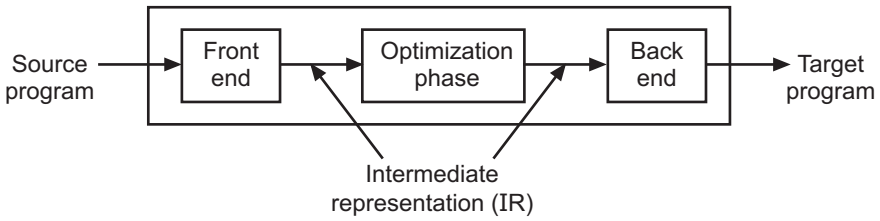


Fig. 1.14: Optimization in Compiler

Advantages of Code Optimization:

1. The optimized program occupied 25 per cent less storage and execute three times faster than un-optimized program.
2. Reduces cost of execution.

Disadvantage of Code Optimization:

1. The 40% extra compilation time is needed.

1.3.6 Code Generation

- The code generation is the last phase of compiler which converts the intermediate code into sequence of machine instructions. These instructions are normally in assembly language.
- Code generation takes the optimized intermediate code as input and maps it to the target machine language.
- Code generation is carried by code generator which takes optimized representation of the intermediate code as input and maps it to the target language machine.
- Code generator translates the intermediate code into the machine code of the specified computer.

  For example, `a:= b + c` is converted as,

```
load b
add c
store a
```

- The code generator utilizes registers more efficiently is good generator. This phase is dependent on machine and operating system for which the code is to be generated.

1.3.7 Symbol Table Management

- Symbol table is a data structure used to store the information about the symbol i.e. identifiers or names that occurs during compilation.
- Symbol table stores the identifiers with their types and addresses and also some additional information. It also stores the scope of identifier that is in which block it belongs.
- When the identifier is detected by lexical analyzer, its entry is made into symbol table with its attributes.
- A symbol table is a data structure used by a compiler to keep track of scope/binding information about names.
- This information is used in the source program to identify the various program elements, like variables, constants, procedures and the labels of statements.
- The symbol table is searched every time a name is encountered in the source text.
- For example, `int i, j`; let i and j are variables in block A.

Symbol	Address	Block
i	200	A
j	202	A

- Hence, symbol table is a data structure which is used for storing identifier's record.

1.3.8 Error Detection, Handling and Reporting

- One of the most important functions of a compiler is the detection and reporting of errors in the source program. Programs submitted to a compiler often have errors of various kinds.
- A good compiler, therefore, should report as many errors as is reasonably possible. In other words, the compiler must act as an effective mechanism to deliver appropriate comments to the user.
- The error messages should allow the programmer to determine exactly where the errors have occurred.
- Errors can be encountered at any phase of the compiler during compilation of the source program for several reasons such as:
    - In lexical analysis phase, errors can occur due to misspelled tokens, unrecognized characters, etc. These errors are mostly the typing errors.
    - In syntax analysis phase, errors can occur due to the syntactic violation of the language.
    - In intermediate code generation phase, errors can occur due to incompatibility of operands type for an operator.
    - In code optimization phase, errors can occur during the control flow analysis due to some unreachable statements.
    - In code generation phase, errors can occurs due to the incompatibility with the computer architecture during the generation of machine code. For example, a constant created by compiler may be too large to fit in the word of the target machine.
    - In symbol table, errors can occur during the bookkeeping routine, due to the multiple declaration of an identifier with ambiguous attributes.
- After detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- A compiler that stops when it finds the first error is not as helpful as it could be. The syntax and semantic errors are detectable by the compiler.
- Error detection and reporting of errors are important functions of the compiler. Whenever an error is encountered during the compilation of the source program, an error handler is invoked.
- Error handler generates a suitable error reporting message regarding the error encountered.
- The error reporting message allows the programmer to find out the exact location the error.

- The following example shows how the statement is translated by a compiler. Consider statement, `a := b – c * 40`
- The translation of this statement is shown in the Fig. 1.15.

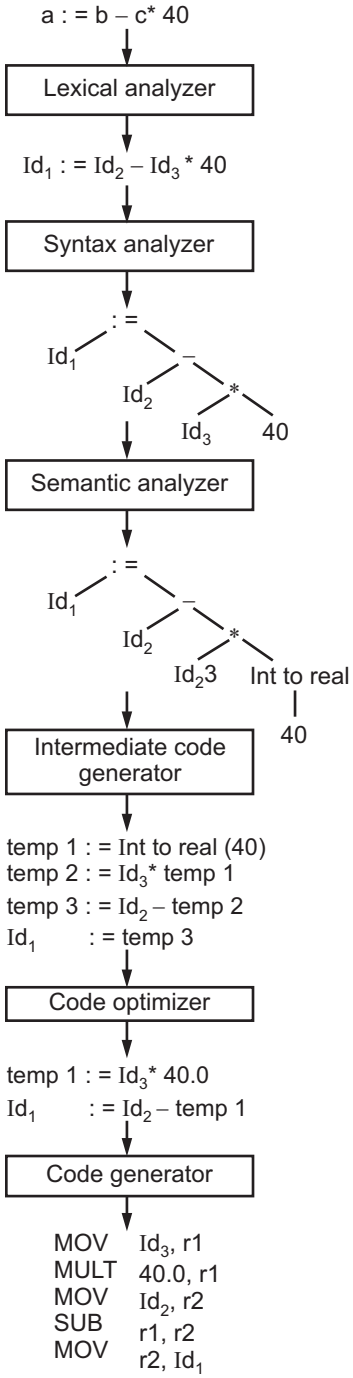


Fig. 1.15: Translation of a Statement by Compiler

1.4 PASSES OF COMPILER

- A pass means several phases of compilers are grouped together and read. A compiler pass refers to the traversal of a compiler through the entire source program.
- In an implementation of a compiler, the activities of one or more phases are combined into a single module known as a pass.
- A pass reads the source program or the output of the previous pass, make the specified transformations.
- The number of passes and the grouping of phases into passes are usually dictated by particular language.
- The compiler phases are grouped into a front-end (or analysis) and a back-end (or synthesis) as explained below:
  1. The front-end part of compiler does the analysis of input code. The front end analyzes the source code to build an internal representation of the program. The front-end called as analysis part which breaks up the source program into constituent pieces and creates an intermediate representation of the source program. It consists of lexical analysis, syntax analysis, semantic analysis and intermediate code generation phases. These phases depend on source language and are independent of the target language.
  2. The back-end part of compiler does the synthesis of the output or target code. The back-end called synthesis part which constructs the desired target program from the intermediate representation. It includes code optimization and code generation phase along with necessary error handling. The back end phases depend on the target language and independent on the source language.
- Compiler pass are two types, single/one pass compiler and multi-pass compiler.
  1. When all the phases of a complier are grouped together into a single pass, then that compiler is known as **single-pass complier** or **one-pass compiler**.
  2. When different phases of a complier are grouped together into two or more passes, then that compiler is known as **multi-pass complier**.

1.4.1 One-pass Compiler [April 16, Oct. 17]

- A single-pass compiler is a compiler that passes through the source code of each compilation unit only once.
- In other words, a single/one-pass compiler does not "look back" at code it has previously processed.
- A one-pass compiler is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code.

- The one-pass compiler converts the program into one or more intermediate representations in steps between source code and machine code and which reprocesses the entire compilation unit in each sequential pass.
- This refers to the logical functioning of the compiler, not to the actual reading of the source file once only.
- For instance, the source file could be read once into temporary storage but that copy could then be scanned many times.
- One-pass compilers are smaller and faster than multi-pass compilers. One-pass compilers are easy to implement and suffers from high storage requirements.
- Entire program is kept into memory because one phase may need information in a different order than a previous phase produces it.
- The internal form of a program may be considerably longer than either the source program or the target program. Back-patching is used to store the symbol's addresses.
- One-pass compiler is used to traverse the program only once. A single pass compiler is also called a narrow compiler.
- The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
- Fig. 1.16 shows one-pass compiler.

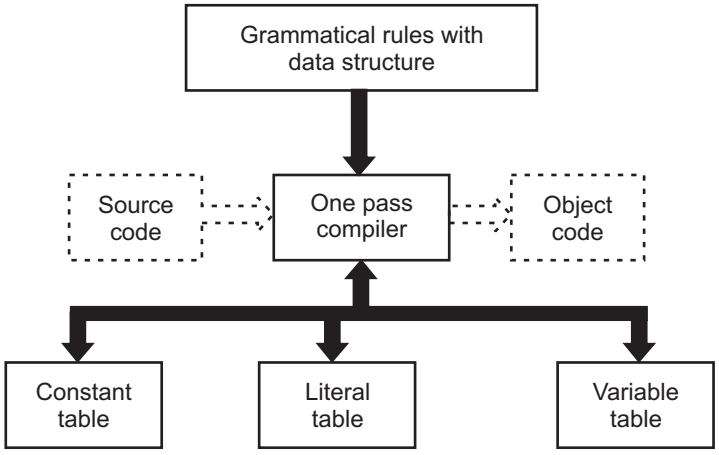


Fig.1.16: One-pass Compiler

Advantages:

1. A single-pass compiler is faster than the multi-pass complier.
2. One-pass compiler uses few passes for compilation.
3. Compilation process in single-pass compiler is less time consuming than multi-pass compiler.

Disadvantages:

1. A single-pass compiler takes more space than the multi-pass compiler.
2. In a single-pass compiler, the complicated optimizations required for high quality code generation are not possible.
3. To count the exact number of passes for an optimizing compiler is a difficult task.
4. One-pass compilers are unable to generate as efficient programs, due to the limited scope of available information.
5. Some programming languages simply cannot be compiled in a single pass, as a result of their design.

1.4.2 Two-pass Compiler [Oct. 17]

- A two-pass compiler is an compiler which goes through assembly language twice and generate object code.
- First pass is called as Pass-I. It performs tasks like Lexical analysis, Syntax analysis, Semantic analysis and intermediate code generation.
- Second pass is called as Pass-II. It performs tasks like Storage allocation, Code optimization and Code generation.
- It suffers from high storage requirements, little less than one-pass.
  1. Backpatching is not required.
  2. Execution time required is more than one-pass.
- Fig. 1.17 shows two-pass compiler.

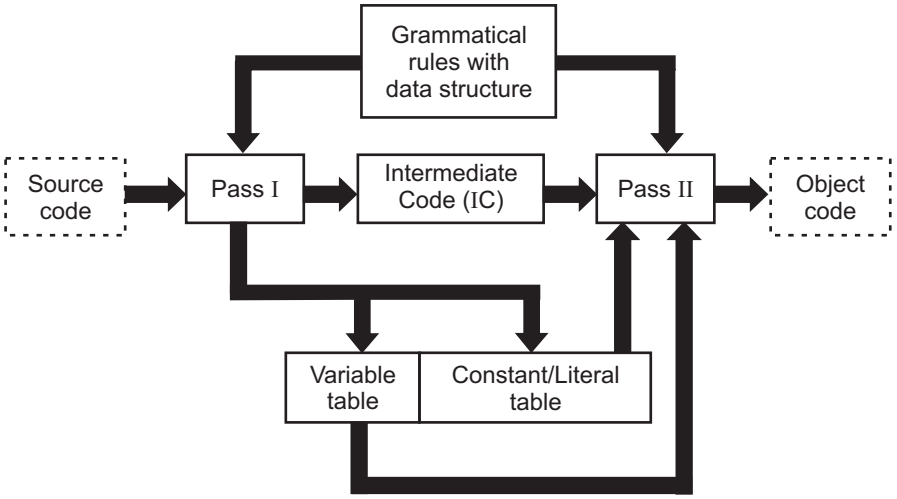


Fig. 1.17: Two-pass Compiler

1.4.3 Multi-pass Compiler

- A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program several times.

- First pass, Second pass, Third pass is called as Pass-I, Pass-II and Pass-III respectfully. As given below:
  - In the Pass I, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
  - In the Pass II, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
  - In the Pass III, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.
- Above pass is going on, until the target output is produced.
- A multi-pass compiler is called a wide compiler. Fig. 1.18 shows multi-pass compiler.

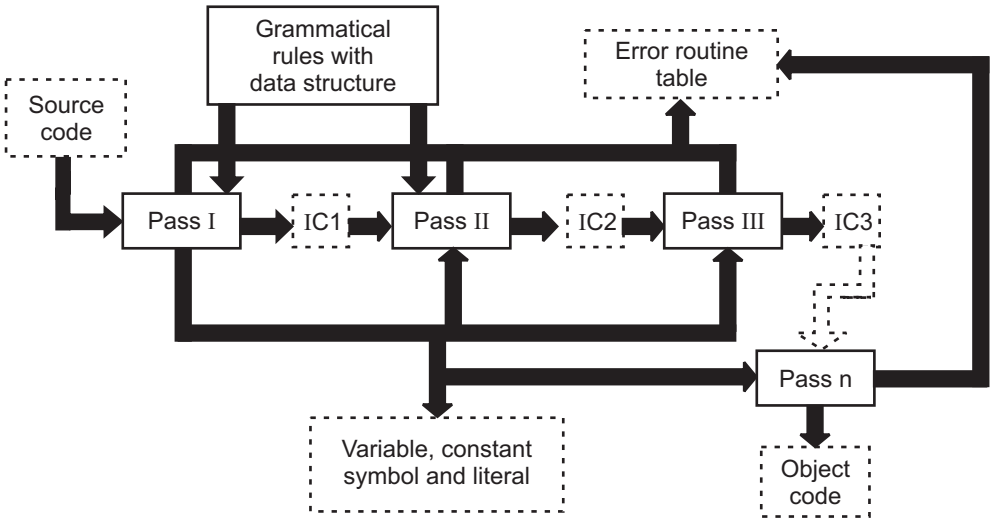


Fig. 1.18: Multi-pass Compiler

Advantages of Multi-pass Compiler:

1. A multi-pass compiler requires lesser memory space than single-pass compiler.
2. The wider scope thus available to these compliers allows better code generation.

Disadvantages of Multi-pass Compiler:

1. In multi-pass compiler each pass reads and writes an intermediate file, which makes the complication process time consuming.
2. The multi-pass compilers are slower than single-pass compiler.
3. The time required for compilation increases with the increase in the number of passes in a complier.

Difference between Single-Pass Compiler and Multi-Pass Compiler:

Sr. No	Single-Pass Compiler	Multi-Pass Compiler
1.	A one-pass complier is that passes through the source code of each compilation unit only once.	A multi-pass compiler is a type of complier that processes the source code of a program several times.
2.	A one-pass complier does not "look back" at code it previously processed.	Each pass takes the result of the previous pass as the input and creates an intermediate output.
3.	In a single-pass compiler all the phases of compiler design are grouped in one pass.	In a multi-pass complier the different phases of a complier design are grouped into multiple passes.
4.	It is less efficient code optimization and code generation.	Better code optimization and code generation.
5.	It is also called as narrow compiler.	It is also called as wide compiler.
6.	Single-pass compiler requires large memory for compilation.	Multi-pass compiler requires small memory for compilation.
7.	They are faster speed in compilation process.	They are slower speed in compilation process. As more number of passes means more execution time.
8.	One pass compiler reads the code only once and then translates/compile it so there is no modification of code.	A multi-pass compiler modifies the program into one or more intermediate representations.
9.	Pascal and C languages compilers are the examples of single-pass compiler.	C++ and Modula-2 languages compilers are the examples of multi-pass compiler.

1.5 CROSS COMPILER [April 18, 19]

- A complier may run on one machine and produce object code for another machine. Such a complier is often called a Cross Complier.
- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
- For example, a compiler that runs on a PC but generates code that runs on Android smartphone is a cross compiler.
- **Definition:** A compiler which may run on one machine and produce the target code for another machine is known as cross compiler.
- A cross compiler can be represented with the help of a T diagram (Tombstone diagrams).

- Fig. 1.19 shows the representation of a T diagram.
- The cross compiler is used to implement the compiler, which is characterized by three languages namely, Source language, Object/Target language and Implementation language in which it is written.
- We can represent the cross compiler by a T diagram which shows three symbols S, T and I where, S is the source language in which the source program is written, T is the target language in which compiler generates/produces its output (or target program) and I is the implementation language in which compiler is written.
- In short we can write as T diagram as $S_I T$.

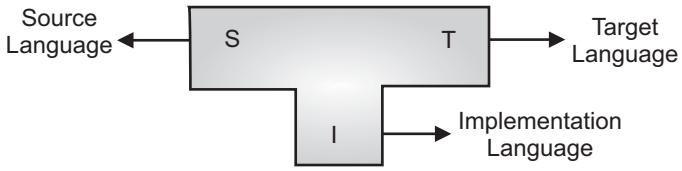


Fig. 1.19: T Diagram Representation in Cross Compiler

- A cross compiler is a type of compiler that can create executable code for different machines other than the machine it runs on.
- A cross compiler can create an executable code for a platform other than the one on which the compiler is running.
- For example, a compiler that runs on Windows platform also generates a code that runs on Linux platform is a cross compiler.

1.6 BOOTSTRAPPING [Oct. 16, April 18, 19]

- Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program.
- Bootstrapping is an approach for making a self-compiling compiler that is a compiler written in the source programming language that it determine to compile.
- A bootstrap compiler can compile the compiler and thus we can use this compiled compiler to compile everything else and the future versions of itself.
- Writing a compiler is a complex process. For Bootstrapping purposes, a compiler deals with three languages, the source language S which compiler compiles, the target language or object language T which compiler generate output and the implementation language I in which compiler is written in.
- We represent these three languages using a T diagram (because of its T like shape) as shown in Fig. 1.20.

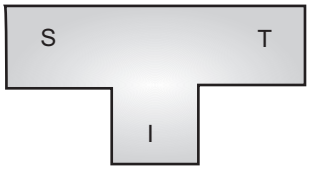


Fig. 1.20

- In UNIX environment, compilers are normally written in C language. Even C compiler is also written in C language.
- If the compiler is written in same language, then how it is compiled? Answer is, we use Bootstrapping.
- The process by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is called as Bootstrapping.

Uses of Bootstrapping:

1. Bootstrapping allows new features to be combined with a programming language and its compiler.
2. Bootstrapping allows languages and compilers to be transferred between processors with different instruction sets
3. It also allows new optimizations to be added to compilers.

- Bootstrapping is the technique for producing a self-compiling compiler i.e., a compiler written in the source programming language that it intends to compile.
- Using the facilities offered by a language to compile itself is the essence of Bootstrapping.
- We can represent S, O and C_L as shown in Fig. 1.21 (a).

$$S \longrightarrow \boxed{C_L} \longrightarrow O$$

Fig. 1.21 (a)

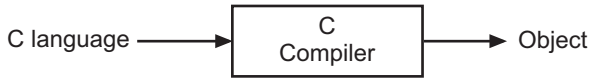


Fig. 1.21 (b): Bootstrapping Process

- These three may be the different languages or source program and compiler language can be same.
- Suppose compiler for new language L can be written in same language L. For example, suppose we want a compiler for new language L, which is available on different machines, say M and N. Consider a language L will work on machine M.
- The source language for compiler is L, target language is M. We want to develop compiler C_M^{LM}.
- So first we will write a small compiler C_M^{SM} for machine M, which translates a subset S of language L into the machine or assembly code of M. As we have C_M^{SM}, we can easily write a compiler C_M^{LM} as shown in Fig. 1.22.

$$C_S^{LM} \rightarrow C_M^{SM} \rightarrow C_M^{LM}$$

Fig. 1.22: Bootstrapping a Compiler

- Now, consider machine N and we want to develop another compiler for L to run on N. Source language is S and target language is N. So we want to convert C_S^{LM} into compiler C_L^{LN} to implement full language L and using C_L^{LN} we produce C_N^{LN} which is compiler for L on N.
- This process is shown in Fig. 1.23.

$$C_L^{LN} \rightarrow C_M^{LM} \rightarrow C_M^{LN}$$

$$C_L^{LN} \rightarrow C_M^{LN} \rightarrow C_N^{LN}$$

Fig. 1.23: Bootstrapping a Compiler to another Machine

- Bootstrapping is an important concept for building a new compiler. Suppose we want to create a cross compiler for the new source language S that generates a target code in language T and the implementation language of this compiler is A.
- We can represent this compiler as, C_A^{ST} (see Fig. 1.24 (a)). Further, suppose we already have a compiler written for language A with both target and implementation language as M.
- This compiler can be represented as, C_A^{AM} (see Fig. 1.24 (b)). Now, if we run C_A^{ST} with the help of C_M^{AM}, then we get a compiler C_M^{ST} (see Fig 1.24 (c)). This compiler compiles a source program written in language S and generates the target code in T, which runs on machine M i.e., the implementation language for this compiler is M.

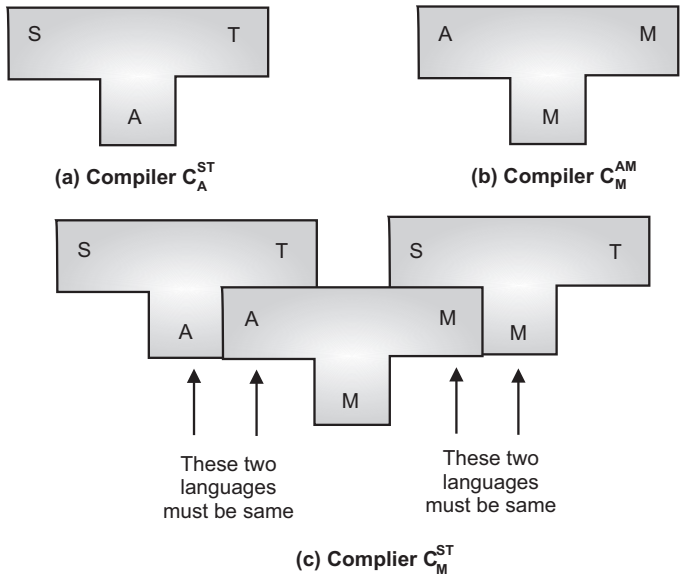


Fig. 1.24: Bootstrapping

Example:

- In the Fig. 1.25 there are three T diagrams.
  - First T diagram contains the P-compiler written in PASCAL. It converts COBOL language into object code.
  - Second T diagram contain the machine language compiler which converts P-code into machine language.
  - Third T diagram contains machine language compiler which converts PASCAL program into object code.

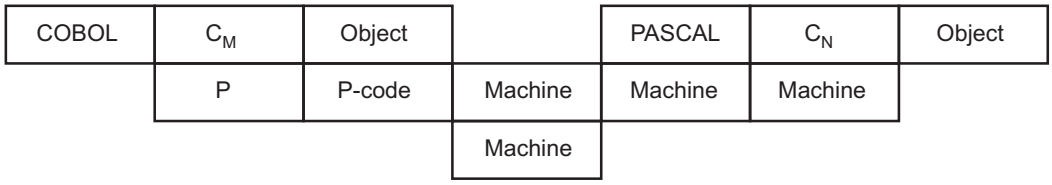


Fig. 1.25: Third T diagram

Advantages of Bootstrapping:

1. In Bootstrapping a compiler has to be written in the language it compiles.
2. Using bootstrapping techniques, an optimizing compiler can optimize itself.
3. In Bootstrapping compiler developers only need to know the language being complied.

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which is a translator (translates source code to object/target code)?
   (a) Compiler (b) Assembler
   (c) Interpreter (d) None of the mentioned
2. Which is a computer program that directly executes instructions written in a programming without requiring them previously to have been compiled?
   (a) Compiler (b) Assembler
   (c) Interpreter (d) None of the mentioned
3. Which is the process translation of source code into target code by a compiler?
   (a) interpretation (b) compilation
   (c) Both (a) and (b) (d) None of the mentioned
4. Aspects of compilation includes,
   (a) Data structures (b) scope rules
   (c) Data types and Control structures (d) All of the mentioned
5. The structure of a compiler is composed of mapping of two parts namely,
   (a) analysis (front-end of the compiler) (b) synthesis (back-end of the compiler)
   (c) Both (a) and (b) (d) None of the mentioned

6. Which refers to the traversal of a compiler through the entire source program?
   (a) token (b) lexeme
   (c) pass (d) None of the mentioned
7. Which is the final phase of compiler?
   (a) lexical analysis (b) syntax analysis
   (c) code optimization (d) code generation
8. Lexical analysis also knows as,
   (a) parsing (b) scanning
   (c) Interpreter (d) None of the mentioned
9. Syntax analysis also knows as,
   (a) parsing (b) scanning
   (c) Interpreter (d) None of the mentioned
10. Compiler translates the source code to,
    (a) executable code (b) machine code
    (c) Both (a) and (b) (d) None of the mentioned
11. Compiler should report the presence of _____ in the source program, in translation process.
    (a) classes (b) errors
    (c) objects (d) None of the mentioned
12. Which data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.?
    (a) Compiler table (b) Symbol table
    (c) Compilation table (d) None of the mentioned
13. The lexical analysis scans the input program character by character and groups the character into the lexical units called,
    (a) tokens (b) scanners
    (c) parsers (d) None of the mentioned
14. How many numbers of tokens in the statement printf("k= %d", k);
    (a) 11 (b) 10
    (c) 4 (d) 6
15. A process, a string of tokens can be generated by,
    (a) parsing (b) scanning
    (c) analyzing (d) translating
16. Following which is not a phase of compiler.
    (a) syntax (b) testing
    (c) lexical (d) semantic

17. Select a machine independent phase of the compiler,
    (a) syntax analysis (b) intermediate code generation
    (c) lexical analysis (d) All of the mentioned
18. By whom is the symbol table created?
    (a) Compiler (b) Assembler
    (c) Interpreter (d) loader
19. Semantic analyzer is used for,
    (a) Generating source code
    (b) Maintaining symbol table
    (c) Generating object code
    (d) None of the mentioned
20. A compiler for a high-level language that runs on one machine and produces code for a different machine is called,
    (a) cross compiler (b) single-pass compiler
    (c) multipass compiler (d) optimizing compiler
21. Which compiler is a compiler that passes through the source code of each compilation unit only once?
    (a) cross compiler (b) single-pass compiler
    (c) multi-pass compiler (d) optimizing compiler
22. Which is an approach for making a self-compiling compiler that is a compiler written in the source programming language that it determine to compile
    (a) Bootstrapping (b) compiling
    (c) translating (d) optimizing
23. A compiler is characterized by three languages as,
    (a) its source language (S) (b) its object language (T)
    (c) language in which it is written (I) (d) All of the mentioned

Answers

1. (a)	2. (c)	3. (b)	4. (d)	5. (c)	6. (c)	7. (d)	8. (b)	9. (a)	10. (c)
11. (b)	12. (b)	13. (a)	14. (d)	15. (a)	16. (b)	17. (d)	18. (a)	19. (c)	20. (a)
21. (b)	22. (a)	23. (d)							

Q. II Fill in the Blanks:

1. A _____ is a program that converts/translates high-level language (source code) to lower level language (machine/object code) without changing the meaning of the program.
2. Compiler _____ includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

3. An _____ accepts a source code as input and immediately executes it.
4. A _____ compiler is fast since all the compiler code is loaded in the memory at once.
5. A _____ of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage.
6. The _____ process is a sequence of various phases and each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.
7. _____ analysis is the first phase of compiler which scans the source code as a stream of characters and converts it into meaningful lexemes.
8. A pass refers to the _____ of a compiler through the entire program.
9. Lexical _____ represents these lexemes in the form of tokens.
10. _____ analysis checks whether the parse tree constructed follows the rules of language.
11. Each phase of compiler can encounter _____ and these errors cab be handled by error handler.
12. The _____ part (back-end) of compiler structure generates the target program with the help of intermediate source code representation and symbol table.
13. A compiler can run on one machine and produce target code for another machine. Such a compiler is known as a _____ -compiler.
14. _____ is an approach for making a self-compiling compiler that is a compiler written in the source programming language that it determine to compile.
15. A cross compiler can be represented with the help of a _____.
16. _____analysis takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).

Answers

1. compiler	2. structure	3. interpreter	4. single-pass
5. phase	6. compilation	7. Lexical	8. traversal
9. analyzer	10. Semantic	11. errors	12. synthesis
13. cross	14. Bootstrapping	15. T-diagram	16. Syntax

Q. III State True or False:

1. A compiler is a computer program that translates computer code written in one programming language (the source language) into another programming language (the target language).
2. A cross compiler is a compiler that runs on one machine and produces object code for another machine. For example, a compiler that runs on a Windows but generates code that runs on Android is a cross compiler.

3. The analysis (front-end) part of compiler structure, of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.
4. After lexical analysis phase the compiler generates an intermediate code of the source code for the target machine.
5. In code generation phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.
6. Bootstrapping is the technique for producing a self-compiling compiler i.e., a compiler written in the source programming language that it intends to compile.
7. Each individual unique step in compilation process is called as pass.
8. An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.
9. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
10. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.
11. Syntax analysis is the second phase of compiler which is also called as scanning.
12. Multi-pass compiler scans the input source once and makes the first modified form, then scans the first-produced form and produces a second modified form, etc., until the object form is produced.
13. The structure of a compiler is composed of mapping of analysis and synthesis parts.
14. Code generation checks whether the parse tree constructed follows the rules of language.

Answers

1. (T)	2. (T)	3. (T)	4. (F)	5. (T)	6. (T)	7. (F)	8. (T)	9. (T)	10. (T)
11. (T)	12. (F)	13. (T)	14. (F)						

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. Define compiler.
2. Define interpreter.
3. List phases of compiler.
4. Define compilation.
5. Enlist types of compilers.
6. Compare compiler and interpreter. Ant two points.
7. Define one-pass compiler.
8. What is multi-pass compiler?
9. Define the term pass.

10. "Symbol table is not required in all phases of compilers". State true or false
11. What is scanner?
12. What is the need of code optimization?
13. What is phase in compiler?
14. Define bootstrapping.
15. 'Compiler is a translator". State true or false.
16. What is the purpose of lexical analysis phase in compiler?
17. Define cross compiler.
18. Which phase in compiler is used for parsing?
19. What is the purpose of code optimization?
20. Define error in compiler.
21. List advantages of interpreter.

(B) Long Answer Questions:

1. What is compiler? Explain with diagram. Also state its advantages and disadvantages.
2. What is interpreter? How it works? How it is differ from compiler?
3. Write a short note on: Aspect of compilation.
4. Describe data types and data structures in compiler.
5. With the help of example describe scope rules.
6. Explain structure of compiler diagrammatically.
7. With the help of diagram describe phase of compiler.
8. What is lexical analysis? Explain in detail.
9. What semantic analysis? Describe in detail.
10. What is symbol table? Hot manage it? Describe in detail.
11. What is multi-pass compiler? Explain diagrammatically with its advantages and disadvantages.
12. How the FA in used in scanner? Explain in detail.
13. What is parsing? How to use it in compiler?
14. What is parse tree? Explain with example.
15. Explain the term intermediate code generation in detail.
16. What is code optimization? Explain with diagram.
17. What is error? How to handle it in compiler? Explain with example.
18. What is one-pass compiler? Explain diagrammatically with its advantages and disadvantages.
19. What is cross compiler? How to represent it? Explain in detail.
20. What is bootstrapping? How to use it? Describe with example.
21. Differentiate between one-pass, multi-pass and cross compilers.

UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Which one is faster in terms of execution, one pass compiler or two pass compiler? **[1 M]**

Ans. Refer to Sections 1.4.1.

2. Explain in detail with the help of a diagram the phases of a compiler. **[5 M]**

Ans. Refer to Section 1.3.

October 2016

1. State True or False, "Bootstrapping is a useful in implementation of cross complier". **[1 M]**

Ans. Refer to Section 1.6.

April 2017

1. List the two aspects of compilation. **[1 M]**

Ans. Refer to Section 1.1.3.

October 2017

1. State True or False, "Target code is generated in the analysis phase of the compiler". **[1 M]**

Ans. Refer to Section 1.3.

2. Differentiate between one-pass compiler and two-pass compiler. **[5 M]**

Ans. Refer to Sections 1.4.1 and 1.4.2.

April 2018

1. What is a cross compiler? **[1 M]**

Ans. Refer to Section 1.5.

October 2018

1. List the phases of compiler. **[1 M]**

Ans. Refer to Section 1.3.

2. Define the term bootstrapping. **[1 M]**

Ans. Refer to Section 1.6.

April 2019

1. Define cross compiler. **[1 M]**

Ans. Refer to Section 1.5.

CHAPTER 2

Lexical Analysis (Scanner)

Objectives...

- To understand Concept of Lexcial Analysis
- To Design the Lexical Analyzer
- To understand Lex

2.0 INTRODUCTION

- Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.
- Lexical analysis is the act of breaking down source text into a set of words called tokens. Each token is found by matching sequential characters to patterns.
- A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.
- The scanner or Lexical Analyzer (LA) performs the task of reading a source text as a file of characters and dividing them up into tokens.
- All the tokens are defined with regular grammar and the lexical analyzer identities strings as tokens and sends them to syntax analyzer for parsing.
- A typical lexical analyzer or scanner is shown in Fig. 2.1.

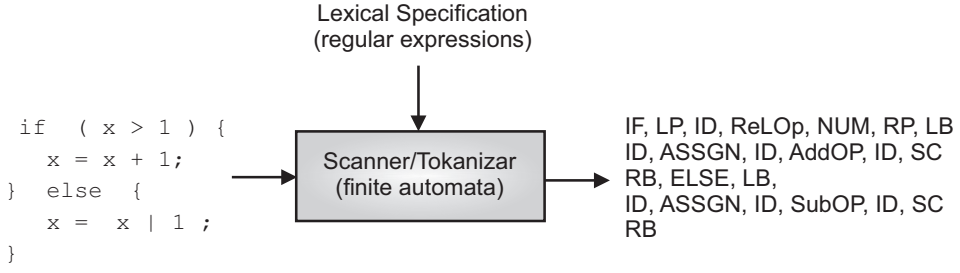


Fig. 2.1: A Lexical Analyzer

- A lexical analyzer consists of an implementation of the finite-state automation. Finite Automata (FA) is a state machine that takes a string of symbols as input and changes its state accordingly.

2.1 BASIC CONCEPTS IN SCANNER [April 16, 17]

- Lexical analysis is the first phase of the compiler also known as a scanner. It converts the high level input program into a sequence of tokens.
- The main task accomplished by lexical analysis is to identify the set of valid words of a language that occurs in an input stream.
- The program which does lexical analysis is called scanner, (see Fig. 2.2).

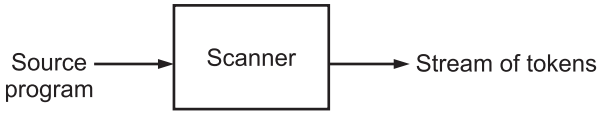


Fig. 2.2: Working of Scanner

- The scanner scans the input program character by character and groups the character into the lexical units called tokens.
- Lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning).
- A program that performs lexical analysis may be termed a lexer, tokenizer or scanner although scanner is also a term for the first stage of a lexer.

2.1.1 Role of Lexical Analyzer [April 16]

- Lexical analyzer acts as an interface between the input source program to be compiled and the later stages of the compiler.
- Lexical analysis means to separate words from the statements from the source code and using rules and regulations i.e. pattern, it converts words in to tokens.
- These words are called as lexeme and the program which performs this task is called as Lexical analyzer.

Need of Lexical Analyzer:

1. Lexical analyzer can perform functions like deleting comments, extra blank spaces and blank lines, keeping track of line numbers.
2. It builds different tables like table of operators, constant table, symbol table etc.
3. Using lexical analyzer we can simplify the design of syntax analyzer.
4. Having lexical analyzer and syntax analyzer is in same pass, which avoids the unnecessary storage of intermediate code.
5. The lexical analyzer may keep track of the number of new line characters, so a line number is associated with each error message by lexical analyzer.

- The main task of lexical analyzer is to read the input program and breaking it up into a sequence of tokens. These tokens are used by the syntax analyzer.
- The interaction between lexical analyzer and parser or syntax analyzer is shown in Fig. 2.3.

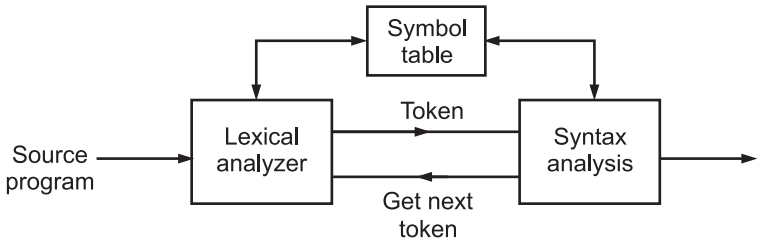


Fig. 2.3: Lexical Analyzer and Parser Interaction

- Lexical analyzer sends a sequence of tokens to the parser and parser sends acknowledge "get next token to lexical analyzer. Lexical analyzer is called by the parser whenever it needs a new token.
- Sometimes, lexical analyzers are divided into following two processes:
  1. **Scanning:** It is a simple process that does not tokenize the input, such as deletion of comments and compaction of consecutive white space characters into one.
  2. **Lexical Analysis:** It is a complex process, which gives output of the scanner as a token.
- There are number of reasons the analysis phase and parsing phase are separated. These reasons are explained below:
  1. **Efficiency:** Separating lexical analysis from syntax analysis increases design efficiency. Compiler efficiency is improved. In lexical analysis, buffering technique is used for reading the input, which speeds up the compiler significantly.
  2. **Simplicity:** By having LA as a separate phase, the complier design is simplified. By separating lexical and syntax analysis phase, logic becomes too simple e.g., if comments and whitespaces (blank, tab, newline) are removed in lexical analysis phase, the parsing becomes so easy and parser design is not complex.
  3. **Portability:** Portability is enhanced. Due to input/output and character set variations, lexical analyzers are not always machine independent. We can take care of input alphabet peculiarities at this level.

2.1.2 Tokens, Patterns and Lexemes [April 16, Oct. 16]

- When talking about lexical analysis, most often we use the terms lexeme, token and pattern.

1. Token: [Oct. 16]

- Token is a sequence of characters that represents a unit of information in the source program. A token is a pair which contains a token name and an optional attribute value.
- Token is a group of characters with logical meaning. Token is a logical building block of the language.
- There is a set of strings in the input for which the same token produced is output. The tokens are input to the parser.

- Examples of tokens include keyword, identifier, operators, constant, punctuations symbols and so on.

2. Pattern:

- A pattern is a rule that describes the character that can be grouped into tokens. It is expressed as a regular expression.
- Input stream of characters are matched with patterns and tokens are identified. A pattern is a set of strings described by a rule that the token may have.
- A rule that defines a set of input strings for which the same token is produced as output is known as pattern.
- Regular expressions play an important role for specifying patterns. Let us see an example, pattern/rule for id is the it should start with a letter followed by any number of letters of digits. This is given by the following regular expression:

  ```
  [A – Za – z][A – Za – z 0 – 9]*
  ```

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

3. Lexeme: **[April 16, 18]**

- A lexeme is a sequence of characters in the source program that matches the pattern for a token.
- Lexeme is an instance of a token. It is the actual text/character stream that matches with the pattern and is recognized as a token.
- For example, in the 'C' language the statement, a = a + 1; in which, token is identifier for lexeme a and pattern is rule for identifier.
- Take another example in 'C' language the statement, `printf("sum = %d\n", sum);` in which, sum is lexemes matching the pattern for token id and "sum = %d\n" is a lexeme matching literal. The printf() is lexeme matching the pattern keyword.
- Consider the following statements:

  1. `const pi = 3.14;`
  2. `int x;`
  3. `if a < b ? a : b;`
  4. `a = "lexical analysis".`

- Lexical analyzer scans the above statements and construct in the Table 2.1.

Table 2.1: Examples of tokens

Line No.	Lexeme	Pattern	Token
1.	const	const	keyword
	pi	[a-zA-z] [a-z-z0-9]*	identifier
	=	= | < | > | < = | > = | ! =	relational operator
	3.14	[0-9] [0-9]*	number

2.	int x	int [a-zA-z] [a-zA-Z0-9]*	keyword identifier
3.	if a, b <, ?, :	if [a-zA-z] [a-zA-Z0-9]* = | < | > | < = | > = | ! = ? | :	keyword (if) identifier operator
4.	a = "lexical analysis"	[a-zA-z] [a-zA-z0-9]* = |< | > | < = | > = | ! = "any characters between"	identifier operator literal

- A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.
- When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.
- For example, the pattern num matches both strings 0 and 1, but it is essential for the code generator to know what string was actually matched.

2.1.3 Lexical Errors

- During the lexical analysis phase Lexical error can be detected. Lexical error is a sequence of characters that does not match the pattern of any token.
- Scanner scans the input and finds lexeme. If the lexeme not matches with any pattern then lexical analyzer detects error.
- In simple words, a character sequence that cannot be scanned into any valid token is a lexical error.
- Misspelling of identifiers, keyword, or operators are considered as lexical errors. The error recovery technique is used by lexical analyzer.
- The following are the error recovery actions performed by lexical analyzer:
  1. Detecting one character from the remaining input.
  2. Inserting a missing character.
  3. Replacing incorrect character by a correct character.
  4. Transposing two adjacent characters.

2.1.4 Input Buffering [April 16, Oct. 16]

- The lexical analyzer scans the input character by character and groups them into tokens.
- Moreover, it needs to look ahead several characters beyond the next token to determine the next token itself. So, an input buffer is needed by the lexical analyzer to read its input.

- In a case of large source program, significant amount of time is required to process the characters during the compilation. To reduce the amount of overhead needed to process a single character from input character stream, specialized buffering techniques have been developed.
- The lexical analyzer scans the characters of the source program one at a time to discover tokens; however, many characters beyond the next token may have to be examined before the next token itself can be determined.
- For this reason two pointers are used, one pointer to mark the beginning of the token being discovered, and the other, a look ahead pointer, to scan ahead of the beginning point, until the token is discovered.
- Fig. 2.4 shows the how to scan the input tokens using look ahead pointer.

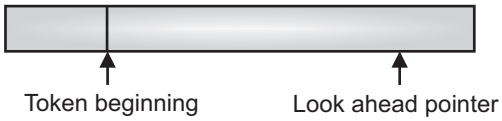


Fig. 2.4: Scanning the Input

- The two pointers to the input buffer i.e., token beginning pointer and look ahead pointer as shown in Fig. 2.4.
- Token Beginning Pointer points to the beginning of the string to be read. Look Ahead Pointer moves ahead to search for the end of the token.
- For example, the Fig. 2.5 shows the lexeme Begin pointer is at character t and forward pointer is at character a.
- The forward pointer is scanned until the lexeme total is found. Once, it is found, both these pointers point to *, which is the next lexeme to be discovered.

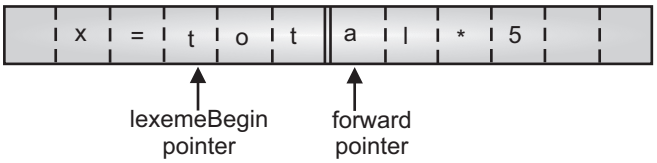


Fig. 2.5: Example of Input Buffer

- A buffer can be divided into two halves. If the look Ahead pointer moves towards halfway in First Half, the second half is filled with new characters to be read.
- If the look Ahead pointer moves towards the right end of the buffer of the second half, the first half will be filled with new characters, and it goes on.
- Fig. 2.6 shows the buffer divided into two halves of n-characters, where n is number of characters on one disk block e.g., 1024.

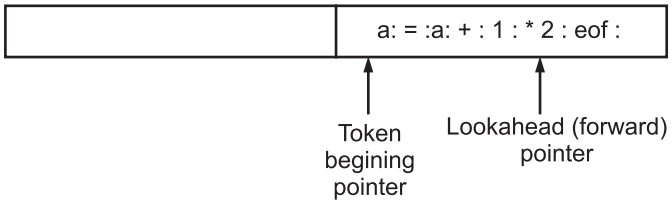


Fig. 2.6: Input Buffer with Two Halves

- The string of the characters between the two pointers is the current token. Initially both pointers points to the first character of the next token to be found.
- Once, the token is found, a look ahead pointer is set to the characters at its right end and beginning pointer is set to the first character of the next token. White spaces (blanks, tab, newline) are not tokens.
- If the look ahead pointer moves beyond the buffer halfway mark, then other half is filled with the next characters from the source file.
- Since, the look ahead pointer move from left half and then again to right half, it is possibility that we may loose characters that not yet been grouped into tokens.
- Every time when left half is full, right half is loaded. If forward pointer move at the end of right half again left half is loaded.
- We can make buffer larger if we choose another buffering scheme. We use sentinels buffering so that characters are not loose when we move left half to right half and vice versa.
- The code of input buffering is as follows:

```
if lookahead pointer is at end of left half
then
   load the right half
   and increment lookahead pointer
else
if lookahead pointer is at end of right half
then
   load the left half and move lookahead pointer to the beginning of left half
else
   increment lookahead pointer
```

- These buffering techniques, makes the reading process easy and also reduces the amount of overhead required to process.

2.1.5 Sentinels [Oct. 16]

- In sentinels we use special character that is not the part of source program. This character is at the end of each half. So every time look ahead pointer checks this character and then the other half is loaded.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- The advantage is that we will not loose characters for which token is not yet formed, while moving from one half to other half.

- Normally, eof is a special character used. But here additional test is required to check whether lookahead pointer points to an eof.
- The Fig. 2.7 shows the sentinels buffering scheme.

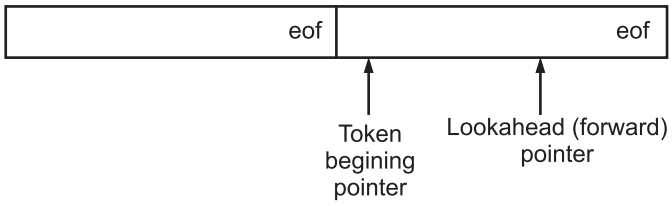


Fig. 2.7: Buffering with Sentinels

- In order to optimize the number of tests to one for each advance of forward pointer, sentinels are used with buffer, (see Fig. 2.8).
- The idea is to extend each buffer half to hold a sentinel at the end. The 'eof' is usually preferred as it will also indicate end of source program.
- The sentinel is a special character eof (end of file) that cannot be a part of source program.

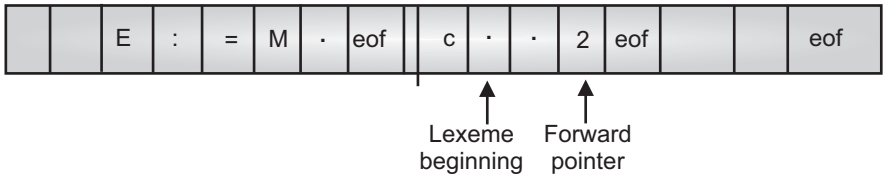


Fig. 2.8: Example of Sentinels

2.2 REVIEW OF FINITE AUTOMATA AS A LEXICAL ANALYZER

- A finite automation is formerly defined as a 5-tuple (Q, ∑, δ, q0, F) where,

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

  ∑ is input alphabet.

  q_0 is initial state.

  F is a set of final states and F ⊆ Q.

  δ is a transition function or mapping function $Q_x \sum \rightarrow Q$ using this the next state can be determined depending on the current input.
- Finite State Machines (FSMs) can be used to simplify lexical analysis. A finite automaton is a recognizer because it merely recognizes whether the input strings are in the language or not.
- A recognizer for a language is a program that takes as input a string x and answer is 'yes', if x is a sentence of the language and 'no' otherwise.
- We compile a regular expression into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA).

- Transition diagram show the actions that take place when a lexical analyzer is called by the parser to get the next token.
- We use a transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
- The design of lexical analyzers depends more or less on automata theory. In fact, a lexical analyzer is a finite automaton design.
- Lexical analysis is a part of compiler and it is designed using finite automaton, which is known as Lexical analyzer.
- Lexical analyzer is used to recognize the validity of input program, whether the input program is grammatically constructed or not.
- For example, suppose we wish to design a lexical analyzer for identifiers; an identifier is defined to be a letter followed by any number of letters or digits, as follows,

```
identifier = {{letter} {letter, digit}}
```

- It is easy to see that the DFA (Deterministic Finite Automata) in Fig. 2.9 will accept the above defined identifier. The corresponding transition table for the DFA is given as given below:

State/symbol	Letter	Digit
A	B	C
B	B	B
C	C	C

Initial state : A
Final state : B

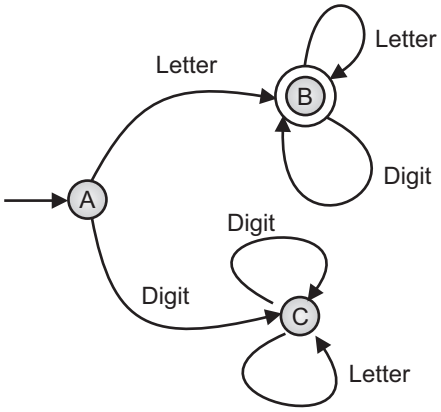


Fig. 2.9: DFA that Accepts Identifier

2.3 APPLICATIONS OF REGULAR EXPRESSIONS AND FINITE AUTOMATA (FA)

- An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

- There are three general approaches to design lexical analyzer:
  1. Write a lexical analyzer in a conventional system programming language (using regular expressions).
  2. Write a lexical analyzer in assembly language.
  3. Write a lexical analyzer using lexical analyzer generator (LEX utility on LINUX).
- Lexical analyzer represents each token in terms of regular expression and regular expression is represented by transition diagram.
- The method of design of lexical analysis is finite automata or transition diagram. We will discuss how lexical analyzer recognizes tokens and its type and value.
- Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
- Transition diagrams are used to keep track of the information about characters that are seen as the forward pointer scans the input.
- Finite Automata (FA) is a state machine that takes a string of symbols as input and changes its state accordingly. FA is a recognizer for regular expressions.
- Regular Expressions (REs) have the capability to express finite languages by defining a pattern for finite strings of symbols.
- The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.
- Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings.
- Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition.
- Regular languages are easy to understand and have efficient implementation. Let us consider a programming language fragment for 'C' language.

  Statement | if cond statement
  | if cond statement else statement
  | ε

  Cond | id relop id
  | id relop num
  | num relop id
  | id
- This is simple if - else statement in 'C' language where relop is any relational operator used for condition in if statement.
- Here, if - else statement is represented in CFG, where LHS symbols are not considered as tokens. Hence the tokens are if, else, relop, id and num.

- Now, for each token we represent regular expressions as follows:

  if → if

  else → else

  relop → < |<=| > |>=| = |< >|

  id → letter (letter | digit)*

  num → (digit)+

  where '*' is zero or more occurrences and '+' is one or more occurrences.

- We assume that tokens are separated by white spaces (blanks, tabs or newlines) so we can have regular expression.

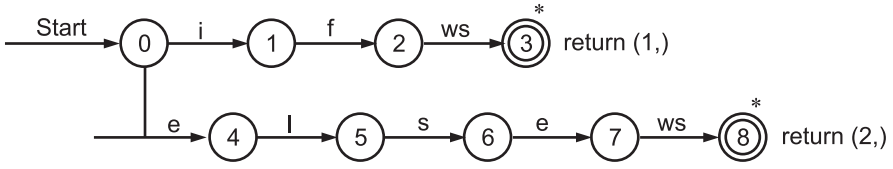
  ωs → (blank | tab | \n)+

- If the match for ws is found, the lexical analyzer does not return a token to the parser. A lexical analyzer recognizes tokens in the input buffer and gives the attribute values shown in Fig. 2.10.

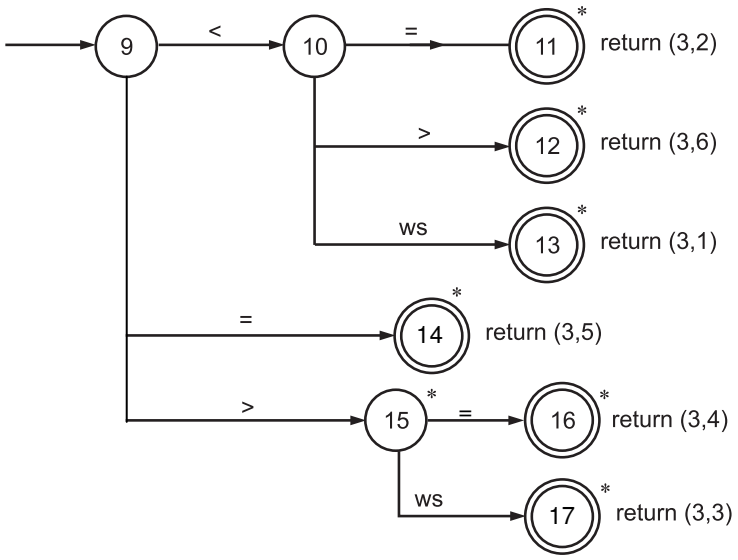
Token	Code	Attribute value
if	1	–
else	2	–
<	3	1
<=	3	2
>	3	3
>=	3	4
=	3	5
<>	3	6
id	4	pointer to symbol table
num	5	pointer to symbol table

Fig. 2.10: Reorganization of Tokens Analysis

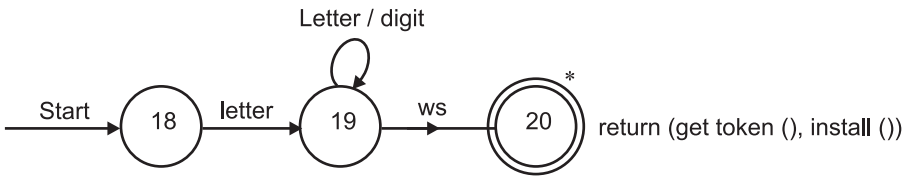
- Design of lexical analyzer can be explained by finite automation. For each token, we draw a transition diagram. A sequence of transition diagrams can be converted into a program or pseudo code.
- The Fig. 2.11 shows the transition diagrams for tokens shown in Fig. 2.10.



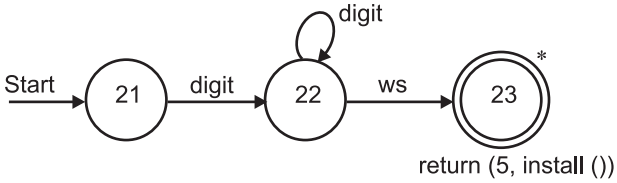
(a) Transition Diagram for Keywords



(b) Transition Diagram for Relational Operator



(c) Transition Diagram for Identifiers



(d) Transition Diagram for num

Fig. 2.11

- For writing a code for each state in the transition diagram of Fig. 2.11, we use the following functions as standard functions:
  1. **getchar():** It returns the input character pointed by lookahead pointer and advances look ahead pointer to next character.
  2. **install():** If token is not in the symbol table then we can add it using install-id() and get the attribute value which is pointer to symbol table.
  3. **error():** Error message can be given.
  4. **gettoken():** It is used to obtain the token. If the lexeme is a keyword, the corresponding token is returned; otherwise the token id is returned.

5. **retract:** Used to retract look ahead pointer one character, which token is found, next character is delimeter and since delimeter is not the part of token, we are retract procedure.

- To convert transition diagrams into a program, we can write a code for each state. To obtain the next character from input buffer we use getchar() function. Then finds the state which has no edge that is after blank or new line (ws).
- It is denoted by double circle, and here the token is found. If all transition diagrams have been tried without success, then error correction routine is called. We use '*' to indicate states on which retract is called.
- The pseudo code is as follows:

Pseudo-code for keyword:

```
state 0 :  c = getchar();
           if c='i' goto state 1;
           else
           if c='e' goto state 4;
           else
           error();
state 1 :  c = getchar();
           if c='f' goto state 2;
           else
           error();
state 2 :  c = getchar()}
           if delimeter(c)
           goto state 3;
           else
           error();
state 3 :  retract();
           return (if,install());
state 4 :  c = getchar();
           if c='l' goto state 5
           else
           error( ;
state 5 :  c = getchar();
           if c='s' goto state 6
           else
           error();
```

```
state 6 : c = getchar();
          if c='e' goto state 7
          else
          error();
state 7 : c = getchar();
          if
          delimeter(c) goto state 8;
          else
          error();
state 8 : retract();
          return (else,install());
```

- Similarly we can write pseudo code for identifier as follows :

```
state 18 : c := getchar();
           if letter (c) goto state 19;
           else
           error();
state 19 : c := getchar();
           if letter (c) or digit (c) then goto state 19;
           else if delimiter (c) goto state 20;
           else
           error ();
state 20 : retract()
           return(id, install ());
```

- Hence, after reorganization of tokens the scanner represents by FA and it can be then implemented by writing code for each state. In UNIX, lex or flex is the utility available which is used to generate lexical analyzer automatically.

Example 1: Create pseudo-code for identifier of C language.

Solution: Find out regular expression for identifier.

```
id → l (l | d)*
```

i.e the pattern for id is

```
[a– zA – z] [a – zA – z0 – 9]*
```

We design the transition diagram for identifier as follows:

Fig. 2.12: Transition Diagram

Pseudo-code:

```
state 0 :  c = getchar();
           if letter (c) goto state 1
           else
           error();
state 1 :  c = gertchar();
           if letter (c) or digit (c) goto state 1
           else
           error()
state 2 :  retract (c);
           return (id, install ());
```

Example 2: Create psedo-code for number of C language.

Solution: Number is (digit) (digit)*

The finite automata is,

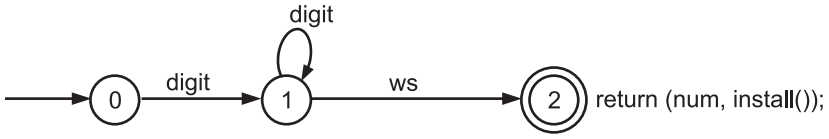


Fig. 2.13: The Finite Automata

Pseudo-code:

```
state 0 :  c = getchar()
           if digit (c) goto state 1
           else
           error()
state 1 :  c = getchar();
           if digit (c) goto state 1
           else
           error();
state 2 :  retract();
           return (num, install ());
```

Example 3: Write pseudo-code for hex-decimal number of C language.

Solution: The hex number start with OX or Ox and it contains 0 – 9, A – F, a - f characters.

The pattern is `O [xX] [0 – 9 A – F a – f]*`

The FA is,

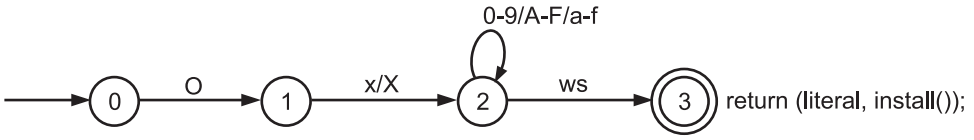


Fig. 2.14: Finite Automata Diagram

Pseudo-code :

```
state 0 :  c = getchar();
           if c = 'O' goto state 1
           else
           error();
state 1 :  c = getchar();
           if c = 'x' or c = 'X' goto state 2
           else
           error();
state 2 :  c = getchar();
           if ((c > 0 and c < 9)) or (c > 'A' and c < 'F') goto state 3
           else
           error();
state 3 :  retract();
           return (literal, install ());
```

- Regular expressions play an important role in the study of finite automation and its application. The oldest applications of regular expressions were in specifying the component of a compiler called a lexical analyzer.
- Regular expressions are extensively used in the design of lexical analyzer. Regular expression is used to represent the language (lexeme) of finite automata (lexical analyzer).
- A regular expression is a compact notation that is used to represent the patterns corresponding to a token.
- Regular languages, which are defined by regular expressions are used extensively for matching patterns within text (as in Word processing or Internet searches) and for lexical analysis in computer language compilers.
- Regular expressions and finite automata are powerful tools for encoding text patterns and searching for these patterns in textual data.
- For example, hashtags in social media messages and posts (e.g. `#OccupyWall Street`) can be found using the notation `(^|\s) # ([A-Za-z0-9_]+)`, where `(^|\s) #` denotes

beginning of a line or a blank space followed by a # and `([A-Za-z0-9_]+)` denotes a sequence of one or more alphanumeric characters and the underscore symbol.

2.3.1 Input Buffering in Lexical Analysis

- Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- The lexical analyzer scans the characters of the source program one at a time to discover tokens.
- Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined.
- For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer.
- Input buffering is defined as a temporary area of memory into which each record of data is read when the Input statement executes. There are efficiency issues concerned with the buffering of input.
- A two-buffer input scheme that is useful when looking ahead on the input is necessary to identify tokens. Then we introduce some useful techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
- The two techniques for input buffering are Buffer pairs and Sentinels. buffer pairs are used to hold the input data. In buffer pairs a buffer is divided into two N-character halves. Each buffer is of the same size N and N is usually the number of characters on one block (e..g., 1024 or 4096 bytes).
- The sentinel is a special character that cannot be part of the source program, (eof character is used as sentinel).
- For detail information of input buffering and sentinel refer to Sections 2.1.4 and 2.1.5.

2.3.2 Recognition of Tokens

- Tokens can be recognized by Finite Automata (FA). In this section, we will elaborate how to construct recognizers that can identify the tokens occurring in an input stream.
- These recognizers are most widely known as Finite Automata. As the name suggests, it is a machine with a finite number of states in which the machine can perform.
- Transition diagram is notations for representing FA. Transition diagram is a directed labeled graph in which it contains nodes and edges. Nodes represent the states and edges represent the transition of a state.
- Fig. 2.15 shows finite automaton that could be part of a lexical analyzer. The job of this automaton is to recognize the keyword then.
- It thus needs five states, each of which represents a different position in the word then that has been reached so far.

- These positions correspond to the prefixes of the word, ranging from the empty string (i.e., nothing of the word has been seen so far) to the complete word.

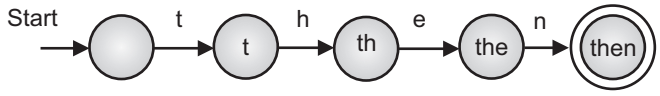


Fig. 2.15: A Finite Automation Modeling Recognition of Then

- In Fig. 2.15, the five states are named by the prefix of then seen so far. Inputs correspond to letters.
- We may imagine that the lexical analyzer examines one character of the program that it is compiling at a time and the next character to be examined is the input to the automaton.
- The start state corresponds to the empty string and each state has a transition on the next letter of then to the state that corresponds to the next-larger prefix.
- The state named then is entered, when the input has spelled the word then. Since it is the job of this automaton to recognize when then has been seen, we could consider that state the lone accepting state.
- Lexical analysis can be performed with pattern matching through the use of regular expressions. Therefore, a lexical analyzer can be defined and represented as a DFA.
- Recognition of tokens implies implementing a regular expression recognizer. This entails implementation of a DFA.
- Fig. 2.16 (a) shows steps in token recognition.
- The identified token is associated with a pattern which can be further specified using regular expressions.
- From the regular expression, we construct an DFA. Then the DFA is turned into code that is used to recognize the token.

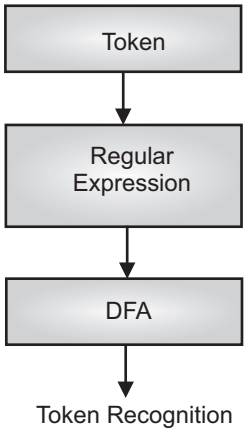


Fig. 2.16 (a): Steps in Token Recognition

- Fig. 2.16 (b) shows an example of transition diagram. Suppose that we want to build a lexical analyzer for the recognizing identifier, > =, >, integer const.
- The corresponding DFA that recognizes the tokens is shown in the Fig. 2.16 (b).

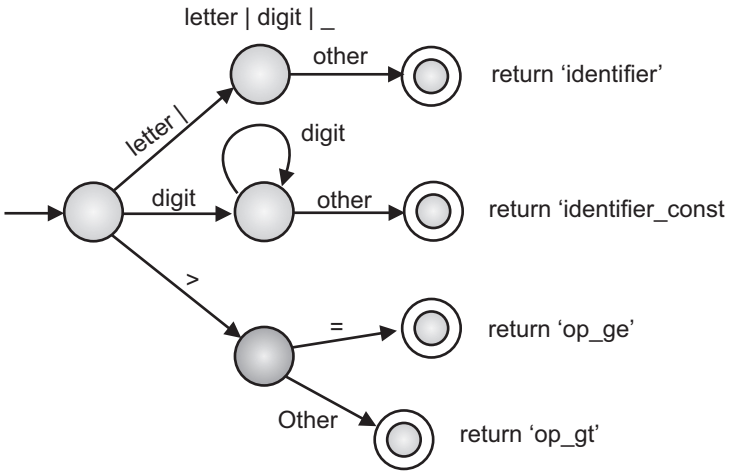


Fig. 2.16 (b): DFA that Recognizes Tokens id, integer_const, etc.

2.4 LEX: LEXICAL ANALYZER GENERATOR [Oct. 16, April 17, 18, 19]

- Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). The purpose of a lex program is to read an input stream and recognize tokens.
- Use a lexical-analyzer generator, such as the Lex compiler, to produce the lexical analyzer from a regular-expression based specification. In this case, the generator provides routines for reading and buffering the input.
- Lex is better known as Lexical Analyzer Generator in Unix OS. It is a command that generates lexical analysis program created from regular expressions and C language statements contained in specified source files.
- Lexical Analyzer Generator introduce a tool called Lex, which allows one to specify a lexical analyzer by specifying regular expressions to describe pattern for tokens. Lex tool itself is a lex compiler.
- A lex compiler or simply lex is a tool for automatically generating a lexical analyzer for a language. It is an integrated utility of the UNIX operating system. The input notation for the lex is referred to as the lex language.
- Lex is a program used to construct lexical analyzer for a variety of languages. The input of lex is in lex language.
- Lex is generally available on UNIX or LINUX. It generates a scanner written in 'C' language.
- Lex is one program of many that is used for generating lexical analyzers based on regular expressions.

- A lexical analyzer is a program that processes strings and returns tokens that are recognized within the input string.
- The token identifies the recognized substring and associates attributes with it. Lex uses regular expressions to specify recognizable tokens of a language.
- The Fig. 2.17 shows the lexical analyzer creation on lex.

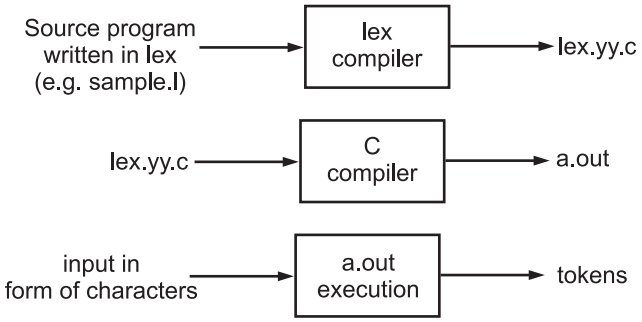


Fig. 2.17: Lexical analyzer on lex

- The source program is written in lex language having extension ·*l*. Then this program is run on lex compiler which always generate C program lex·yy·c which we link with lex library - *ll*.
- This program is having representation of transition diagram for every regular expression present in the program. Then len·yy·c is compiled on 'C' compiler which generate output file a·out.
- The about is used for transforming input stream into a sequence of tokens. The compilation process on LINUX is as follows :

```
$ lex sample·l
$ cc lex·yy·c - ll   (where – ll is optional)
$ ·\a·out
```

Lex Program Specification: **[Oct. 16]**

- A lex program consists of three sections and each section is separated by %%.

```
definition or declaration
%%
translation rules
%%
procedures written in 'C' language
```

- The three sections of lex program are explained below:
  1. **The declaration section** consists of declaration are header files, variables and constants. It also contains regular definitions. We surround the C code with special delimeters "%{" & "%}", Lex copies the data between "%{" & "%}" directly to the 'C' file.

2. **The rule section** consists of the rules written in regular expression forms with corresponding action. This action part is written in 'C' language. In other words, each rule is made up of two parts: a **pattern** and an **action** separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. These patterns are regular expressions written in UNIX-style. C code lines anywhere else are copied to an unspecified place in the generated 'C' file. If the C code is more than one statement then it must be enclosed within braces { }.

   When a lex scanner runs, it matches the input against the pattern in the rule section. When it finds a match then it execute C code associated with that pattern.
3. **The procedure section** is in C code and it is copied by lex to C file. For large program, it is better to have supporting code in a separate source file. If we change lex file then it will not be affected.

- The following table shows the regular expressions used in lex.

Table 2.2

Regular Expressions	Matches
• (dot)	any single character except the newline character "\n".
*	zero or more occurrences of expression
+	one or more occurrences of expression
?	zero or one occurrences of regular expression. e.g. (– (digit) ?)
\	Any escap character in 'C' language.
[a - z]	range of characters from 'a' to 'z'. range is indicated by '-' (hyphen).
[0 - 9]	range of digits from 0 to 9.
$	matches end of line, as it is last character of regular expression.
^	matches beginning of line, as it is first character of regular expression.
[012]	zero or one or two.
|	"or" e.g. a | b either a or b.
( )	regular expressions are grouped (a | b).

Example 4: A lex program to recognize verbs had, have and has.

Solution:

```
/* Lex program to recognize verbs */
% {
  #include<stdio.h>
% }
%%
```

```
(has/have/had)          { printf ("%s : is a verb \n", yytext);}
[a – zA – z]+           { printf ("%s : is not a verb \n", yytext);}
%%
main();
{
  yylex();
}
```

- Let us save this program as sample·l and compile, as follows :

```
$ lex sample·l
$ cc lex·yy·c - ll
$ ·/a·out
  had
  had is a verb
$ ·/a·out
  abc
  abc is not a verb
```

- Some Lex Library functions are discribed below: **[Oct. 17, 18, April 18]**

1. **yylex():** This function is used to start or resume scanning. The next call in program1 to yylex() will continue from the point where it left off. All codes in rule section are copied into yylex().
2. **yytext():** Whenever a lexer matches a token, the text of the token is stored in the null terminated string yytext. (work just like pointers in C). Whenever the new token is matched, the contents of yytext are replaced by new token.
3. **yywrap():** The purpose of yywrap() function to additional processing in order to "wrap" things up before terminating. **[April 19]**

```
yywrap()
{
  return (1);
}
```

When yylex() reaches the end of its input file, it calls yywrap( ), which returns a value of 0 or 1. If the value is 1, indicates that no further input is available. By default it always return 1.

4. **yyerror( ):** The yyerror( ) function is called which reports the error to the user.

```
yyerror()
  {
    printf ("error (any)");
  }
```

- This routine finds the error in the input. All versions of yacc also provide a simple error reporting routine.

Note:

- When we use a lex scanner and a yacc parser together, the parser is the high level routine.
- It calls the lexer yylex( ) whenever it needs a token from the input and lex returns a token value to the parser.
- In this lex-yacc communication, the lexer and parser have to agree what the token codes are. Hence, using a preprocessor #define we define integer for each token in lex program.
- For example: `#define id 257`

  `#define num 258`

  yacc can write a C header file containing all of the token definitions in y·tab·h.
- To compile and execute lex program on UNIX :

  `$ lex programme.l`

  `$ cc lex.yy.c - ll`

  (*ll* is lex library)
- We can also compile without - *ll* option.

  `$ ./a.out` (to run the program)

Example 5: Find out regular expression in LEX for language containing the strings starting with and ends with d over {a, d}.

Solution: a [ad]* d

Example 6: Find regular expression for hex-decimal number in C language.

Solution: [– +] ? 0 [xX] [0 – 9a – fA – F]$^{+}$

Example 7: Find regular expression for language ending with 1 over {0, 1}.

Solution: [0 1] [0 | 1]*1

Example 8: Write a regular expression for floating point number in C language.

Solution: [– +] ? [0 – 9]$^{+}$ (\ · [0 – 9]$^{+}$) ? (E (+ \–) ? [0 – 9]$^{+}$) ?

Example 9: A lex program to recognize the token of Example 5.

Solution:

```
/* Lex specification to recognize tokens if, else, <, >, <=, >=, =, <>, id, num */
% {
#define if 255
#define else 256
```

```
#define relop 257
#define id 258
#define num 259
% }
%%
[iI] [fF]                {return (if);}
[eE] [lL] [sS] [eE]      {return (else);}
"<"                      {return (relop);}
"<="                     {return (relop);}
">="                     {return (relop);}
">"                      {return (relop);}
"="                      {return (relop);}
"<>"                     {return (relop);}
[A-Za-z] [A-Za-z0-9]*    {return (id);}
[0-9]+                   {return (num);}
%%
main( )
{
  yylex( );
}
```

Example 10: A lex program to count number of vowels and number of words per line and total number of lines ending with '·'.

Solution:

```
% { # include <stdio·h>
    int lcnt = 0, vcnt = 0, wno = 0;
% }
% %
[ · ]+ {lcnt ++; wno ++; printf ("lcnt = % d vowel = % d
       word = % d \n", lcnt, vcnt, wno);
       vcnt = 0; wno = 0;}
[ ]+      {wno ++;}
[aeiouAEIOU]  {vcnt ++;}
· ;
% %
    main()
```

```
{  printf ("Enter input : \n");
   yylex();
   printf (" \n Total lines = % d", lcnt);
}
```

Example 11: A lex program to find sum of N numbers.

Solution:

```
% {
   # include<stdio·h>
   int sum, x, a [50];
% }
% %
[0 – 9]+  { x=atoi(yy text);
            sum=0; printf ("\n");
            for (i=0; i<x; i++)
            { printf ("\enter number");
              scanf ("%d", &a[i]);
              sum + = a[i];
            }
            printf ("sum is /n=", sum);
            return (0);
          }
% %
main()
{
  printf ("enter how many numbers");
  yylex();
}
  yyerror()
{
  printf ("error");
}
  yywrap()
  {
    return (1);
  }
```

Now compile this program as:

```
$ lex sum·l
$ cc lex·yy·c - ll
$ ·/a·out
  enter how many numbers
3
  enter number 10
  enter number 20
  enter number 30
  → sum is 60
```

We can also change the a·out file name that is we can use our own executable file.

```
$ lex sum·l
$ cc lex·yy·c - 0 sumout - ll
$ ·/sumout
    enter how many numbers
    2
    enter number 5
    enter number 10
    sum is 15.
```

Example 12: A lex program to find factorial of a given number.

Solution:

```
% {
   # include<stdio·h>
   int i, fact, n;
% }
% %
[0 – 9]+  { n = atoi (yytext);
          for (i=1; i<=n; i++)
          fact = fact * i;
          printf ("Factorial do no. % d is %d\n", n, fact);
          return (0)
% %
   main()
   { printf ("\n Enter number");
     yylex();
   }
```

Output:

```
$ lex fact · l
$ cc · lex · yy · c –ll
$ ·/a·out
Enter number
5
  Factorial of number 5 is 120
  enter number
  4
  factorial of number 4 is 24.
```

Example 13: A lex program to find sum of first n numbers.

Solution:

```
% {
   # include<stdio·h>
     int i, x, sum = 0;
% }
% %
[0 – 9]+  {
          x=atoi (yytext);
          for (i=0; i<=x; i++)
        {
          sum=sum+i;
          printf ("%d", i);
        }
          prinft ("The sum of first %d numbers is %d /n"; x, sum);
          return (0);
        }
  % %
    main()
    {
      printf ("\enter number \n");
      yylex();
    }
```

Output:

```
$ lex sum · l
```

```
$ cc lex · yy · c – 0 sumofn –ll
$ ·/sumofn
  Enter number
    5
    1 2 3 4 5
  The sum of first 5 numbers is 15
$ ·/ sumofn
  enter number
    10
    1  2  3   4  5  6  7  8  9  10
The sum of first 10 numbers is 55.
```

Example 14: A lex program which find out total words, lines and characters from input end with $.

Solution:

```
% {
   # include<stdio·h>
     int c=0, w=0, l=0;
% }
% %
[\t]+  {
       w++; c+=strlen (yytext);}
[·\n] {
       l++, w++}
[$]  {
       printf ("\n\n\t total characters : % d \n
       \t Total words : \t%d\n Total lines : %d
       \t \n", c, w, l);
       return (0);
     }
% %
   main()
  {
    yylex();
  }
```

Output :

```
$ lex cwl . l
$ cc lex.yy.c – 0 cwl –ll
$ ./cwl
  lexical analysis is the first phase of compiler
  parser is the second phase $
  Total characters : 72
  Total words : 13
  Total lines : 2
```

Example 15: A lex program to find total vowels from the input.

Solution:

```
% {
     # include<stdio.h>
     int w=0, vc=0;
%  }
% %
[\t]+  {w++;]
[a e i o u A E I O U]  {vc ++;}
[.]+{
     printf ("\n\n Total vowels %d is", vc;}
. ;
% %
     main()
     {
       printf ("enter input \n");
       yylex();
     }
```

Output :

```
Enter input
Compiler is easy course
Total vowels is 9.
```

Example 16: A lex program to display occurrences of word *Computer* in text.

Solution:

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

```
        int c=0; w=0;
% }
%%
    [\t]+  {
           w++; c+=strlen (yytext);}
    [·\n] {
           printf ("\n\n\t total number of occurrences = % d \n", c);
    [cC] computer [c++};
% %
    main()
    {
      yylex();
    }
```

Output :

```
$ lex count·l
$ cc lex·yy·c -ll
$ · /a·out
    Computer and computer
    Total number of occurrences = 2
```

Example 17: A lex program to find area of square.

Solution:

```
% {
     # include<stdio·h>
     int a;
%  }
% %
[0 – 9]+  {
          s = atoi (yytext);
          a = s * s
          printf ("\n area of square is \n%d", a);
          return (0);
         }
% %
  main()
  {
```

```
    printf ("enter side \n");
    yylex();
  }
```

Output:

```
$ lex area . l
$ cc lex.yy.c
$ ./a.out
Enter side 2
Area of square is 4.
```

Example 18: A lex program that identifies tokens like id, if and for.

Solution:

```
%{
  include <stdio.h>
  %}
  %%
  [a-zA-z] [a-zA-z0-9]*     {return id;}
  [iI]  [fF]       {return if;}
  [Ff] [Oo] [Rr]           {return for;}
  %%
  main()
  {
    printf ("\n Enter word");
    yylex();
  }
```

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which is the process of converting a sequence of characters from source program into a sequence of tokens?
   (a) Lexical analysis (b) Syntax analysis
   (c) Semantic analysis (d) None of the mentioned
2. A program which performs lexical analysis is termed as,
   (a) tokenizer (b) lexical analyzer (lexer)
   (c) scanner (d) All of the mentioned

3. Which is a state machine that takes a string of symbols as input and changes its state accordingly?
   (a) Finite automata (b) Finite compilation
   (c) Finite translation (d) None of the mentioned
4. Which is a sequence of characters that matches the pattern for a token i.e., instance of a token?
   (a) String (b) Pattern
   (c) Lexeme (d) grammar
5. Which is sequence of characters that can be treated as a unit of information in the source program?
   (a) String (b) Pattern
   (c) Lexeme (d) Token
6. Which expressions have the capability to express finite languages by defining a pattern for finite strings of symbols?
   (a) Grammar (b) Regular
   (c) Language (d) None of the mentioned
7. A character sequence that cannot be scanned into any valid token is,
   (a) a lexical pattern (b) a lexical token
   (c) a lexical error (d) None of the mentioned
8. Which is a special character (eof) that cannot be part of the source program?
   (a) Sentinel (b) Buffer pair
   (c) Token (d) Pattern
9. The role of Lexical analyzer includes,
   (a) Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
   (b) Enters the identified token into the symbol table.
   (c) Displays error message with its occurrence by specifying the line number.
   (d) All of the mentioned
10. The process of forming tokens from an input stream of characters is called as,
    (a) Characterization (b) Tokenization
    (c) translocation (d) None of the mentioned
11. When expression sum=3+2 is tokenized then what is the token category of 3?
    (a) Identifier (b) Integer Literal
    (c) Assignment (d) Addition Operator

12. Which grammar defines Lexical Syntax?
    (a) Context Free Grammar (CFG) (b) Syntactic Grammar
    (c) Regular Grammar (d) Lexical Grammar
13. Which is used for reorganization of tokens?
    (a) Finite automata (b) Lexical Grammar
    (c) Regular Grammar (d) None of the mentioned
14. Which is a tool used for generating/creating lexical analyzers based on regular expressions?
    (a) FLex (b) TLex
    (c) Lex (d) None of the mentioned
15. Which function is used to start or resume scanning?
    (a) yyerror() (b) yytext()
    (c) yywrap() (d) yylex()
16. What is the output of a lexical analyzer?
    (a) Machine code (b) Parse tree
    (c) Stream of tokens (d) Intermediate code
17. What goes over the characters of the lexeme to produce value?
    (a) Parser (b) Scanner
    (c) Generator (d) none of the mentioned
18. If the input file is Myfile.c then output file of Lex is,
    (a) Myfile.yy.c (b) Myfile.lex
    (c) Myfile.e (d) Myfile.obj
19. In lex, following which character matches end of line, as it is the last character of regular expression.
    (a) # (b) ^
    (c) ? (d) $
20. Lex program transform the lex file into which program.
    (a) y.tab.c (b) lex.yy.c
    (c) yywrap (d) a.out

Answers

1. (a)	2. (d)	3. (a)	4. (c)	5. (d)	6. (b)	7. (c)	8. (a)	9. (d)	10. (b)
11. (b)	12. (d)	13. (a)	14. (c)	15. (d)	16. (c)	17. (b)	18. (a)	19. (d)	20. (b)

Q. II Fill in the Blanks:

1. _____ analysis (or lexing or tokenization) is the process of converting a sequence of characters (such as in a computer program) into a sequence of tokens (strings with an assigned and thus identified meaning).
2. A _____that performs lexical analysis may be termed a lexer or tokenizer or scanner.
3. Because of large amount of time consumption in moving characters, specialized _____ techniques have been developed to reduce the amount of overhead required to process an input character.
4. _____ is a valid sequence of characters which are given by lexeme.
5. The sequence of tokens produced by lexical _____ helps the parser in analyzing the syntax of programming languages.
6. A lexical _____ is caused by the appearance of some illegal character or misspelling of identifiers and keywords,, mostly at the beginning of a token.
7. _____ describes a rule that must be matched by sequence of characters (lexemes) to form a token.
8. Lex tool itself is a lex _____.
9. The _____ function is called which reports the error to the user.
10. The grammar defined by regular expressions is known as regular _____.
11. Regular _____ is an important notation for specifying patterns.
12. Finite automata is a _____ for regular expressions.
13. _____ stands for lexical-analyzer generator.
14. Lexical analyzer acts as an _____ between the input source program to be compiled and the later stages of the compiler.
15. An automaton with a _____ number of states is called a Finite Automaton (FA)

Answers

1. Lexical	2. program	3. buffering	4. Token
5. analyzer	6. error	7. Pattern	8. compiler
9. yyerror()	10. grammar	11. expression	12. recognizer
13. Lex	14. interface	15. finite	

Q. III State True or False:

1. Lexical analysis is the process of producing tokens from the source program.
2. Tokens can be defined by regular expressions or grammar rules.
3. To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme for this reason a two-buffer (buffer-pair) scheme is introduced to handle large look heads safely.

4. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.
5. If the lexical analyzer finds a token valid, it generates an error.
6. Lexical errors can be handled by the actions Deleting one character from the remaining input, Inserting a missing character into the remaining input and Replacing a character by another character.
7. The language defined by regular grammar is known as regular language.
8. The yylex() function is used to start or resume scanning.
9. The sentinel is a special character i.e., eof that can be part of the source program.
10. Lexical Analysis can be implemented with the Deterministic finite Automata.
11. The output is a sequence of tokens that is sent to the parser for syntax analysis.
12. A sequence of input characters that comprises a single token is called a lexeme.
13. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as lexemes.
14. The purpose of a lex program is to read an input stream and recognize tokens.
15. Lexical analyzer represents each token in terms of regular expression.

Answers

1. (T)	2. (F)	3. (T)	4. (T)	5. (F)	6. (T)	7. (T)	8. (T)	9. (F)	10. (T)
11. (T)	12. (T)	13. (F)	14. (T)	15. (T)					

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is the purpose of lexical analysis?
2. Define lexing.
3. Define tokenization.
4. Define regular expression.
5. What finite automata?
6. List any two lex library function.
7. Define token recognition.
8. Define input buffering.
9. “Lex is a compiler”. Comment.
10. What is token?
11. What is the output of Lex program?
12. Lex is a scanner provided by Linux operating system. State true/ false.
13. Define pattern.
14. 'Lexical analyzer keeps the track of line number', state true or false.

15. What is lexical analyzer?
16. What is lexeme?
17. Write the purpose of lex library functions yylex() and yyerror().
18. What is lexical error?
19. Define the term sentinel.
20. What is Lex?

(B) Long Answer Questions:

1. What is lexical analysis? Explain with diagram.
2. Differentiate between tokens and lexemes.
3. With the help of diagram describe lex.
4. What is meant by lex libraries? Describe any two library functions with example.
5. What is input buffering? What is its purpose? Explin with example.
6. What the applications of regular expressions and finite automata.
7. With the help of diagram describe how to recognition of tokens.
8. Write a short note on: Finite Automata (FA) as a lexical analyzer?
9. What is sentinel? Describe with example.
10. Write a lex program which find out factors of a given number.
11. Write a lex program to find the area of circle.
12. Write a lex program to find factorial of a given number.
13. What is lex? Write steps of execution of lex program. Explain any three lex library functions in brief.
14. Write a lex program to count total number of vowels and total number of consonants from the input string.
15. Write a lex program to return tshe tokens identifier and number.

UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Define the term 'lexeme'. **[1 M]**

Ans. Refer to Section 2.1.2, Point (3).

2. What is the output of Lexical Analyzer? **[1 M]**

Ans. Refer to Section 3.1.

3. Write a LEX program to find factorial of a given number. **[5 M]**

Ans.
```
% {
  # include <stdio.h>
     int i, fact, n;
  % }
```

```
% %
[0-9]+ {  n = atoi (yytext);
   for (i = 1; i < = n; 1++)
   fact = fact * i;
   printf ("The factorial is % d", fact);
   return (0);
% %  }
   main()
   {
      print ("Enter the number");
      yylex();
   }
```

4. Write a short note on Input Buffering with the help of a diagram. **[3 M]**

Ans. Refer to Section 2.1.4.

October 2016

1. State the three sections of a lex program. **[1 M]**

Ans. Refer to Section 2.4.

2. Define the term Token. **[1 M]**

Ans. Refer to 2.1.2, Point (1).

3. What is the use of lookahead pointer? **[1 M]**

Ans. Refer to Section 2.1.5.

2. Write a lex program to find the area of a circle. **[5 M]**

Ans. Refer to Section 2.4.

April 2017

1. What is the output of Lexical Analysis? **[1 M]**

Ans. Refer to Section 2.1.

2. Write a LEX Program to find the area of a circle. Write the steps to execute the LEX program. **[5 M]**

Ans. Refer to Section 2.4.

October 2017

1. List any two LEX Library functions. **[1 M]**

Ans. Refer to Section 2.4.

2. Write a LEX program of find sum of first n numbers. **[5 M]**

Ans. Refer to Section 2.4.

April 2018

1. List any two LEX library functions. **[1 M]**

Ans. Refer to Section 2.4.

2. Define the term lexeme. **[1 M]**

Ans. Refer to Section 2.1.2, Point (3).

October 2018

1. What is the output of LEX program? **[1 M]**

Ans. Refer to Section 2.4.

2. State any two functions of Lex library. **[1 M]**

Ans. Refer to Section 2.4.

3. Write a LEX Program for calculating the cube of a given number. **[5 M]**

Ans. Refer to Section 2.4.

April 2019

1. State True or False. The yywrap() lex library function by default always return 1. **[1 M]**

Ans. Refer to Section 2.4.

2. Give the name of the file which is obtained after compilation of the lex program by the Lex compiler. **[1 M]**

Ans. Refer to Section 2.4.

3. Write a LEX Program which identifies the tokens like id, if, for and while. **[5 M]**

Ans. Refer to Section 2.4.

CHAPTER 3

Syntax Analysis (Parser)

Objectives...

- To learn Concept of Syntax Analysis (Parsing)
- To understand Parser and it's Types
- To study YACC

3.0 INTRODUCTION

- Syntax analysis is the second phase of a compiler. Syntax is the grammatical structure of a language or program.
- Syntax analysis phase is also known as parsing.
- Parsing is the process of determining whether a string of tokens can be generated by a grammar.
- Syntax analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.
- The errors like missing commas, brackets; invalid variable are reported by compiler in syntax analysis or parsing phase.
- The input to parsing phase is token from lexical analyzer and output is parse tree.
- In general, syntax analysis means to check the syntax of the input statements with the help of stream of tokens from lexical analysis and produce parse tree to the semantic analysis.
- The program which performs syntax analysis is called as syntax analyzer or parser.
- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The process of constructing a derivation from a specific input sentence is called parsing.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code and outputs a parse tree.

3.1 PARSERS

- The program performing syntax analysis is known as parser. The syntax analyzer (parser) plays an important role in the compiler design.
- The main objective of the parser is to check the input tokens to analyze its grammatical correctness.
- Parser is one of the components in a complier, which determines whether if a string of tokens can be generated by a grammar.
- Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of compiler for further processing.
- Syntax analysis or parsing is a major component of the front-end of a compiler. Parsing is the process of determining if a string of tokens can be generated by a grammar.
- A parser scans an input token stream, from left to right and groups the tokens in order to check the syntax of the input string by identifying a derivation by using the production rules of the grammar.
- The syntax analyzer receives valid tokens from the scanner and checks them for grammar and produces valid syntactical constructs. The syntax analyzer is also called as parser.

Fig. 3.1: Syntax Analyzer

- The syntax analyzer (parser) performs the following main tasks/functions:
  1. The function of the syntax analyzer is to check that the tokens output by the lexical analyzer occur in patterns permitted by the syntax of the expressions.
  2. It should report any syntax error in the program.
  3. It should also recover from the errors so that it can continue to process the rest of the input.
- A syntax analyzer creates the syntactic structure (generally a parse tree) of the given program.
- It groups the tokens appearing in the input in order to identify larger structures in the program.
- This process is done to verify that the string can be generated by the grammar for the source language.

3.1.1 Definition of Parsing

- Parsing takes input from the lexical analysis and builds a parse tree, which will be used in future steps to develop the machine code.
- To determine the syntactic structure of an input from lexical analysis is called as parsing.
- The goals of parsing are to check the validity of a source string and to determine its syntactic structure.
- Parsing (also known as syntax analysis) can be defined as a process of analyzing a text which contains a sequence of tokens, to determine its grammatical structure with respect to a given grammar. **OR**
- Parsing is the process of determining the syntactic structure (generally a parse tree or syntax tree or derivation tree) input token streams from lexical analysis to the grammatically rules.
- Parsing takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). A parse tree depicts the steps in parsing.
- In parsing phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.
- The Fig. 3.2 shows the parsing in compilation process. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

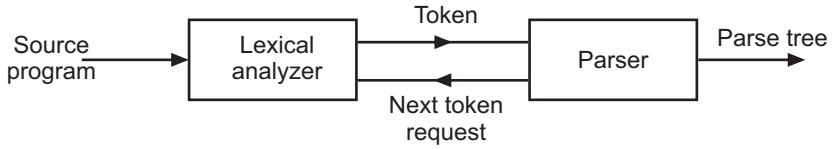


Fig. 3.2: Parsing

- Syntax analysis or parsing means to check the tokens present in source program are grouped in the syntactically correct format or not. Each language has its own syntax. To define the syntax of a language, we make the use of grammars.
- Fig. 3.3 shows process of parsing.
- The first stage is the token generation or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions.
- The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression.
- The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action.

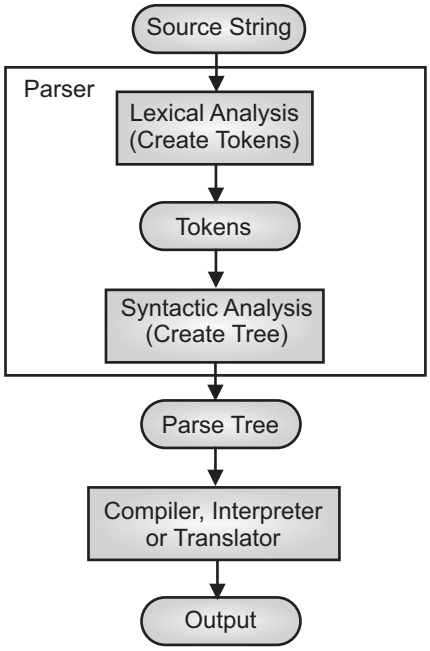


Fig. 3.3: Parsing Process

- Consider the sentence: "The Cat ate the Mouse".
- The syntax of the English language for the above statement is described by using the symbols S: Sentence, V: Verb, O: Object, A: Article, N: Noun, SP: Subject phrase, VP: Verb phrase, NP: Noun phrase.

  Here, N → Mouse, Cat
  V → ate
  A → the
- The complete derivation or generation of the sentence is given as follows:

  S ⇒ SP VP
  ⇒ A N VP
  ⇒ the N VP
  ⇒ the Cat VP
  ⇒ the Cat V O
  ⇒ the Cat ate O
  ⇒ the Cat ate NP
  ⇒ the Cat ate A N
  ⇒ the Cat ate the N
  ⇒ the Cat ate the Mouse

  Rules

S	→	SP VP
SP	→	A N
VP	→	V O
O	→	NP
NP	=	A N
- We are trying to generate the sentence from the set of productions or rules. Left hand side is replaced with one of the rule or production rule.

- The words that cannot be replaced by anything are called terminals and the words that must be replaced by other things are called non-terminals.

  $\therefore$ Non-terminals $\rightarrow$ string of non-terminals + terminals.
- The grammatical rules are often called productions. To know the syntax of language is correct or not, grammar is used, called Context Free Grammar (CFG) which is invented by the Noam Chomsky in 1956. It is also called Type-2 grammar.
- Following terminologies are required in syntax analysis phase:

1. **Alphabet:** A set of characters allowed by the language. The individual members of these set are called terminals. An alphabet is a finite collection of symbols denoted by $\sum$. A symbol is an abstract entity. It cannot be formerly defined as, points in geometry.
2. **String:** Group of characters from the alphabet. Any finite sequence of alphabets is called a string.
3. **Production Rules**: They are the rules to be applied for splitting the sentence into appropriate syntactic form for analysis. Syntax analyzers follow production rules defined by means of context-free grammar.

   These rules are in the following **form**:

   Non-terminals $\rightarrow$ string of terminals and non-terminals.

   This form is called BNF (Backus Noun Form) since on left hand side only one non-terminal symbol is present.
4. **Grammar (CFG):** CFG is collection of an alphabet of letters called terminals from which we are going to make the string that will be the words of a language. A CFG consists of terminals, non-terminals, start symbol and production rules.

   Terminal is a token from the alphabet of strings in the language. A set of symbols are called non-terminals. Start symbol refers to starting of the sentence rules (mostly its starts with capital letter S).

   A finite set of productions in the BNF form:

   Grammar: G = (NT, T, P, S)

   where, NT $\rightarrow$ finite set of non-terminals

   T $\rightarrow$ finite set of terminals

   P $\rightarrow$ finite set of production rules and S is a start symbol.

   Example of grammar is,

   $S \rightarrow E$

   $E \rightarrow E + E$ is same as $S \rightarrow E$

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

   $E \rightarrow id$

   Here, NT = {S, E} T = {+, * id}

   Above four production rules and S is start symbol.

5. **Sentence:** A string of terminals is called as sentence. For example: id + id ∗ id.
6. **Sentential form:** It consists of non-terminals and terminals. For example: E ∗ E where E is NT and ∗ is T.
7. **Derivation:** A derivation is basically a sequence of production rules, in order to get the input string. It is replacement of a non-terminal by the R.H.S. of production rule. Every non-terminal symbol in sentential form is replaced by terminals or non-terminals using production rules. This process is continued until we get a sentence.

   If the leftmost non-terminal in sentential form is replaced by the R.H.S. of production rule and then the derivation is obtained, it is called as left derivation.

   If the rightmost non-terminal symbol in sentential form is replaced by R.H.S. of production rule, then the derivation is called as right derivation.

   For example:

   $S \Rightarrow E$
   $\Rightarrow E + E$
   $\Rightarrow id + E$
   ↑ left derivation

   $S \Rightarrow E$
   $\Rightarrow E + E$
   $\Rightarrow E + id$
   ↑ right derivation
8. **Reduction:** Replacement of a set of terminals or non-terminals (or sentential form), which matches R.H.S. of a production rule by a non-terminal on L.H.S. is called as reduction.

   For example: id + id ∗ id

   $E + id * id \quad \leftarrow$ we reduce id to E (since $E \rightarrow id$)

   (Top-down parsing uses derivation whereas bottom-up parsing uses reduction).
9. **Syntax tree or Parse tree or Derivation tree:** It is a graphical representation of an expression which indicates the sequence in which production rules can be applied on starting symbols so as to derive the string. A parse tree is a representation for the sentence into a structure. Parse tree is used to define the hierarchical structure of the token streams. A parse tree is a diagram which exhibits the syntactic structure of the expressions.

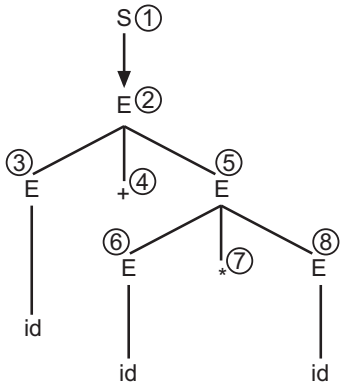


Fig. 3.4: Parse Tree

10. **Ambiguous Grammar:** A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

    Consider a grammar G whose production rules are,

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

    Consider a sentence id + id * id. Now, if we want to generate parse tree for this statement starting with S, then we can generate it with two methods.

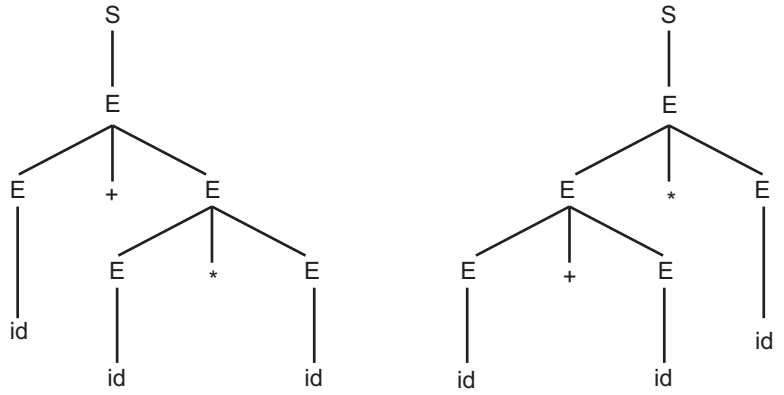


Fig. 3.5: Rightmost and Leftmost Parse Tree

When same input sentence has more than one parse trees then the grammar is said to be ambiguous.

In general, a CFG is called ambiguous if for at least one word in the language that it generates, there are two possible derivations of the word that correspond to different syntax tree.

In the above grammar, ambiguity is present because the operators +, &, * have been given the same priority.

Ambiguity can be removed by giving precedence to the operators. The above grammar can be modified as follows:

$$S \rightarrow E$$
$$E \rightarrow E + V \mid V$$
$$V \rightarrow V * T \mid T$$
$$T \rightarrow id$$

which is unambiguous grammar.

3.1.2 Types of Parsers

- Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types i.e., top-down parsing and bottom-up parsing.

- In the top-down parsing, we attempt to construct the derivation of the input string or to build a parse tree starting from the top (root) to the bottom (leaves).
- In the bottom-up parsing, we build the parse tree, starting from the bottom (leaves) and work towards the top.
- In both cases, input to the parser is scanned from left to right, one symbol at a time. Fig. 3.6 shows the parsing methods.

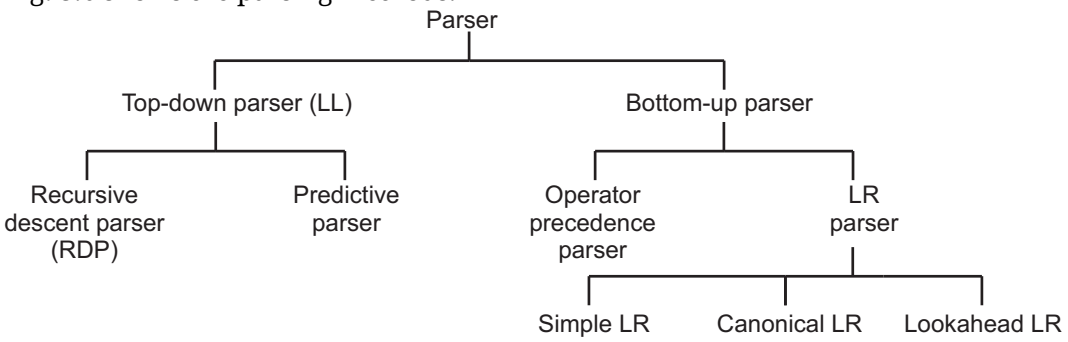


Fig. 3.6: Types of Parsing

- There are two major classes/types of parsers namely, top-down and bottom-up. These names refer to the way in which the parse tree is formed.
- Top-down parsers build a parse tree starting at the root and work down to the leaves. Bottom-up parsers build a parse tree starting at the leaves and work up to the root.

3.2 TOP-DOWN PARSING

- In top-down parsing, we start at the top of the parse tree (with the start symbol) and we end up at the bottom of the parse tree.
- In other words, the top-down parsers begin with the root and grow the tree toward the leaves.
- We use derivation in top-down parsing i.e. input string is derived from starting symbol 'S' and by applying production rules. The idea of top-down parsing is to build the derivation tree from the root.
- In top-down parsing, the derivation has the format $S \overset{*}{\Rightarrow} \alpha$ where α is input string which can be derived from S in 'n' numbers of steps where $n \geq 0$.
- Top-down parsing is also called LL (1) parsing or Left-to-Left parsing because in this method, we always select leftmost non-terminal for expansion.
- Due to this, syntax tree goes on increasing on L.H.S. i.e. it scans the input from left to right and generates leftmost parsing. Here '1' stands for look ahead.
- Consider the example of string a + b * c being parsed according to the grammar.

S	→	E	Here, start symbol is S
E	→	T + E | T	NT = {S, E, T, V}
T	→	V * T | V	Ts = {+, *, id}
V	→	id	

Sentence is a + b * c $\Rightarrow$ id + id * id

Prediction	Sentential Form Used
E $\Rightarrow$ T + E	T + E
T $\Rightarrow$ V	V + E
V $\Rightarrow$ id	id + E
E $\Rightarrow$ T	id + T
T $\Rightarrow$ V * T	id + V * T
V $\rightarrow$ id	id + id * T
T $\rightarrow$ V	id + id * V
V $\rightarrow$ id	id + id * id

Fig. 3.7: Top-down Parsing using Derivation

- Derivation according to a selected alternative is called as prediction. All possible sentences derivable from grammar can be according to top-down parse.
- If we have more than one production with same L.H.S. in case of LL (1), then backtracking is required.
- Backtracking means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.
- Whenever, any derivation is produced during processing, it can be matched with the input string.
- A successful match implies that further predictions should be made to continue the parse. An unsuccessful match implies that some previous predictions have gone wrong.
- At this stage, it is necessary to reject the previous predictions so that the new predictions can be made.

Example, CFG is,

$S \rightarrow aAb$

$A \rightarrow ab \mid b$

Input string ω = abb

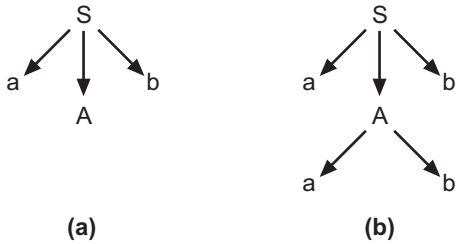


Fig. 3.8: Derivation of w using Top-down Parsing

- After applying first unique production on S, we now match second input symbol and advance input pointer to 'b' of string abb. Now consider b, which is third input symbol, i.e. advance pointer to 'b'.

- But derivation is A → ab. Since 'a' does not match, we report failure and go back to A to try for other alternative. Here, reset input pointer to position 2 and then we obtain.

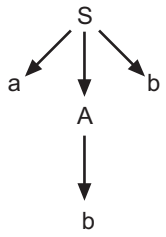


Fig. 3.9

- Here, we have parse tree for w and we halt. Using backtracking, parsing is completed successfully.

3.2.1 Top-Down Parsing with Backtracking

- Backtracking means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production.
- The backtracking technique may process the input string more than once to determine the right production.
- Consider the grammer,

  S →rXd

  X →oa | ea

  For input string "raed", Backtracking will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The production of S (S →rXd) matches with it.
- So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X →oa).
- It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X →ea).
- Now the parser matches all the input letters in an ordered manner. The string is accepted.

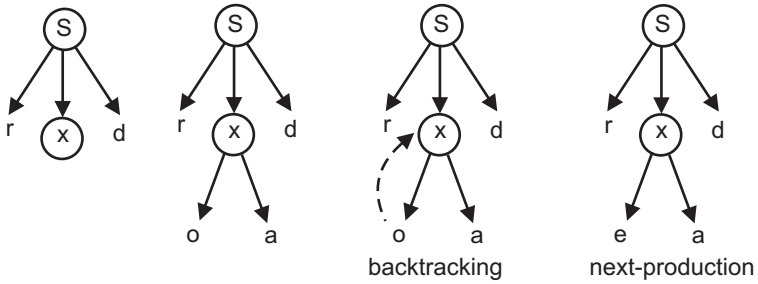


Fig. 3.10

- Backtracking is required in the next example and we shall suggest a way of keeping track of the input when backtracking takes place.

For example, consider the grammar:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string = cad. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled S.

An input pointer points to c, the first symbol of w. We then use the first production for S to expand the tree and obtain the tree of Fig. 3.11 (a).

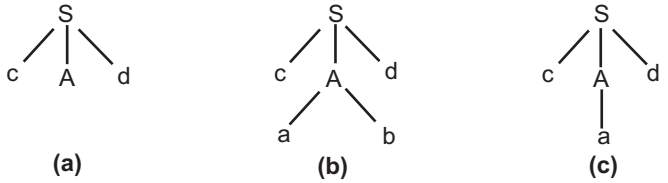


Fig. 3.11: Steps in top-down parse

The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w and consider the next leaf, labeled A.

We can then expand A using the first alternative for A to obtain the tree of Fig. 3.11 (b).

We now have a match for the second input symbol so we advance the input pointer to d, the third input symbol and compare d against the next leaf, labeled b.

Since b does not match d, we report failure and go back to A to see whether there is another alternative for A that we have not tried but that might produce a match.

In going back to A, we must reset the input pointer to position 2, the position it had when we first came to A, which means that the procedure for A must store the input pointer in a local variable.

We now try the second alternative for A to obtain the tree of Fig. 3.11 (c). The leaf a matches the second symbol of ω and the leaf d matches the third symbol.

Since, we have produced a parse tree for ω, we halt and announce successful completion of parsing.

- A backtracking parser is a non-deterministic recognizer of the language generated by the grammar.
- The simplest way of top-down parsing is to use backtracking. A parser takes the grammar and constructs the parse tree by selecting the production as per the guidance initiated by left to right scanning of the input string.
- For example, if the input string is s= bcd and the given grammar has productions,

  $S \rightarrow bX$

  $X \rightarrow d \mid cX$
- For the construction of the parse tree for the string bcd, we start with root leveled with start symbol S.

- We have only one option for S as bX and also its fist symbol (terminal b) is matched with the first symbol of string bcd.
- Now the replacement of X must be done in such a way that the second leaf node in the derivation tree should be 'e'.
- If X is replaced with ‘d’ then we will have to back track because there will be no matching in second symbol between input string and yield of the parse tree.
- Therefore the non-terminal X is replaced with cX. Finally the non-terminal X will be replaced with 'd' so that the yield of a parse tree is similar to input string.
- The construction of a parse tree is given in Fig. 3.12.

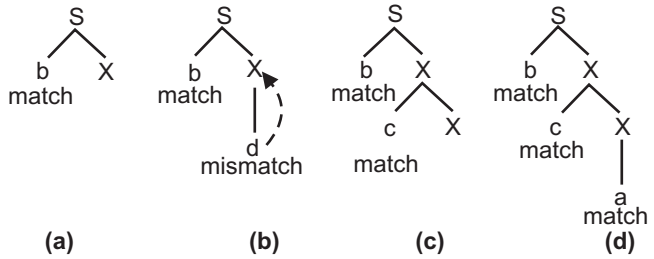


Fig. 3.12: Steps of Construction of a Parse Tree

3.2.2 Drawbacks in Top-Down Parsing with Backtracking

- Backtracking slows down parsing even if no semantic actions are performed during parsing.
- If we make a series of expansions and subsequently discover a mismatch, we may have to undo the semantic effects of erroneous expansions like, say the removal of entries from symbol table.
- Undoing the semantic effects is a substantial overhead for a compiler. This is one of the major disadvantages of Backtracking a top-down parser.
- In top-down backtracking parser, the order in which alternates of production rules are tried can also affect the language considered. This makes backtracking parsers unsuitable for production compilers.
- Another disadvantage of the backtracking top-down parser is the difficulty in error reporting.
- In top-down backtracking parser, it is difficult to pinpoint where the error has occurred and consequently the compiler cannot emit informative error messages.
- Precise error indication is not possible in top-down parsing. Whenever a mismatched is encountered, the parser performs the standard actions of backtracking to make another prediction.
- When no more predictions are possible, the input string is declared erroneous. At this stage, it is not possible to pinpoint the error precisely.

- Left-recursion is another problem in top-down parsing. The grammar which is left recursive is not suitable in top-down parsing. For example, the grammar is,

$$S \rightarrow E$$
$$E \rightarrow E + V \mid V$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow id$$

Now, if we try to derive an input string id + id * id, the predictions made would be,

$$E \Rightarrow E + V$$
$$E \Rightarrow E + V$$
$$E \Rightarrow E + V$$
$$:$$

Since, the left-to-left nature of the parse would push the parser into an infinite loop of prediction to make top-down parsing feasible.

It is necessary to rewrite a grammar so as to eliminate left-recursion.

Hence, two main problems occurring in top-down parsing are left-recursion and backtracking.

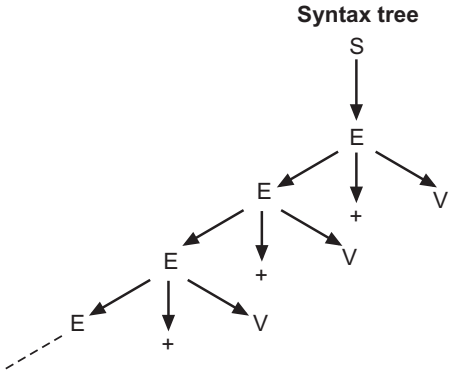


Fig. 3.13

- Elimination of backtracking in top down parsing would have several advantages parsing would become more efficient and it would be possible to perform semantic actions and precise error reporting during parsing.
- Backtracking can be avoided by transforming the grammar in such a way that at each step the choice of production that can lead us to solution can be easily identified.
- In other words, at each step, we can 'predict' which of the productions can lead us to the complete derivation of the input string, if one exists.
- The idea behind a top-down predictive parser is that the current non-terminal being processed combined with the next input symbol can guide the parser to take the correct production rule eventually leading to the match of complete input string.
- The predictive parser is a type of top-down parser that does not require backtracking in order to derive various input strings.
- This is possible because the grammar for the language is transformed such that backtracking is not needed.
- What kind of transformations do we make to the grammar rules to suit a predictive parser? There are two types of transformations done to the grammar in order to suit a predictive parser. They are Elimination of left recursion and Left factoring.

3.2.3 Elimination of Left-Recursion (Direct and Indirect)

- The left recursive grammar having the production in the form $A \rightarrow A\alpha$, where A is non-terminal and α is a string of terminals and/or non-terminals.
- We have already discussed that left recursive grammars are not suitable for top-down parsing. The above production is called immediate *left*-recursion.
- Let us see how this left-recursion is removed. If the production is $A \rightarrow A\alpha/\beta$, this left recursive production is changed to non-left recursive production without changing the language of the grammar as:

$$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A'/\in \end{array}}$$

- If we consider group of A-productions which are having immediate left-recursion as:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$$

- Then we convert them into non-left recursive productions as,

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A'$$
$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \in$$

- Now consider the grammar of expression:

$$\boxed{\begin{array}{l} E \rightarrow E + V | V \\ V \rightarrow V * T | T \\ T \rightarrow (E) | id \end{array}}$$

Grammar 3.1

- Now, we eliminate immediate left-recursion which is present in 2 production.

$$E \rightarrow E + V | V$$

  Here, A is E, α is + V and β = V.
- By applying above rule, we get,

$$E \rightarrow VE'$$
$$E' \rightarrow + VE' | \in$$

- Similarly, for V-production we get,

$$V \rightarrow TV'$$
$$V' \rightarrow * TV' | \in$$

- So the grammar becomes:

$$\boxed{\begin{array}{lll} E & \rightarrow & VE' \\ E' & \rightarrow & + VE' | \in \\ V & \rightarrow & TV' \\ V' & \rightarrow & * TV' | \in \\ T & \rightarrow & (E) | id \end{array}}$$

Grammar 3.2

- Now, if we are not having left-recursion immediately but left-recursion is present in production of the form $A \overset{+}{\Rightarrow} A\alpha$ (derivations of two or more steps).
- Then we first convert into in immediate left-recursion form by replacing non-terminals to its deviations and then apply the rule.
- A context-free grammar is called left recursive if a non-terminal 'A' as a leftmost symbol of the rightmost derivation of A-production (called direct left-recursive) or through some other non-terminal definitions (called indirect/hidden left-recursive).

Direct Left Recursion:

- Let the grammar is A -> Aα/β, where α consists of a terminal and non-terminals but β does not start with A.
- At the time of derivation if we start from A and replace A by the production $A \rightarrow A\ \alpha$, then for all the time A will be the leftmost non-terminal symbol.

Indirect Left Recursion:

- Indirect left recursion in its simplest form could be defined as:

  $A \rightarrow B\alpha \mid C$

  $B \rightarrow A\beta \mid D,$

  possibly giving the derivation $A \Rightarrow B\alpha \Rightarrow A\beta\alpha \Rightarrow$.
- More generally, for the non-terminals A_0 A_1, ..., A_n, indirect left recursion can be defined as being of the form:

  $A_0 \rightarrow A_1\alpha \mid ....$

  $A_1 \rightarrow A_2\alpha_2 \mid ....$

  ....

  $A_n \rightarrow A_0\alpha_{n+1} \mid ....$

  where $\alpha_1, \alpha_2, ..., \alpha_n$, are sequences of non-terminals and terminals.

Example 1: If the productions of grammar are:

$$A \rightarrow BC \mid a$$
$$B \rightarrow AB \mid b$$
$$C \rightarrow a$$

Solution: Here, there is no immediate left-recursion but it is present as:

$$A \Rightarrow BC \Rightarrow ABC$$
left-recursion

Hence A productions are

$$A \rightarrow ABC \mid bC$$

We obtain,

$$A \rightarrow ABC \mid bC \mid a$$

Here, α = BC and β = a & bC

$$\left.\begin{array}{l} A \rightarrow aA' \mid bCA' \\ A' \rightarrow BCA' \mid \in \end{array}\right\} \text{ removing left-recursion}$$

$\therefore$ Grammar becomes (without left-recursion):

$$\begin{aligned} A &\rightarrow aA' \mid bCA' \\ A' &\rightarrow BCA' \mid \in \\ B &\rightarrow AB \mid b \\ C &\rightarrow a \end{aligned}$$

Example 2: Eliminate left-recursion from following grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \in$

Solution: Here, the non-terminal S is left recursive because:

$S \Rightarrow Aa \Rightarrow Sda$

It is not immediate left-recursion.

To eliminate left-recursion, we substitute S-productions in A-productions and we obtain,

$A \rightarrow Ac \mid Aad \mid bd \mid \in$

Now eliminating immediate left-recursion, we get following A-productions.

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \in$

So the grammar becomes:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \in$

Example 3: Eliminate left-recursion from the following grammar:

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Solution: L-production is having immediate left-recursion. After eliminating left-recursion we get,

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow , SL' \mid \in$

3.2.4 Need for Left Factoring

- Even though after eliminating left-recursion, some grammars are not suitable for top-down parsing. For one non-terminal there are two or more productions.

- A useful method for manipulating grammars into a form suitable for top-down or predictive parsing is left-factoring.
- Left factoring is a process of factoring out the common prefixes of alternatives. That is, left factoring is a grammar transformation or manipulation which is useful for producing a grammar suitable for recursive-decent parsing or predictive parsing.
- Left factoring is needed to avoiding backtracking problem. To left factor a grammar, we collect all productions that have the same Left-Hand-Side (LHS) non-terminal and begin with the same terminal symbols on the Right-Hand-Side (RHS).
- We combine the common strings into a single production and then append a new non-terminal symbol to the end of this new production.
- Finally, we create a new set of productions using this new non-terminal for each of the suffixes to the common production.

Definition:

- If $A \rightarrow \alpha\beta \mid \alpha\gamma$ are two A-productions (where α is non-empty string) then with left-factored, original productions become:

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta \mid \gamma$

Example 4: Find out left-factoring grammar for following grammar:

$S \rightarrow aAbB \mid aAb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

Solution: Applying the left-factoring rule for S-production. (Here, α = aAb, β = B & $\gamma = \in$).

The grammar becomes:

$S \rightarrow aAbS'$

$S' \rightarrow B \mid \in$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid a$

Now, apply left-factoring rule for A-production and B-production. The left-factored grammar becomes:

$S \rightarrow aAbS'$

$S' \rightarrow B \mid \in$

$A \rightarrow aA'$

$A' \rightarrow A \mid \in$

$B \rightarrow bB'$

$B' \rightarrow B \mid \in$

Example 5: Consider the following grammar:

rexp → rterm + rexp | rterm

rterm → rfactor rterm | rfactor

rfactor → rfactor * | rprimary

rprimary → a|b

(a) Left factor this grammar.

(b) Does left-factoring make the grammar suitable for top-down parsing.

(c) Eliminate left-recursion from original grammar.

(d) Is the resulting grammar is suitable for top-down parsing.

Solution: First remove the left-recursion, we obtain,

rexp → rterm + rexp | rterm

rterm → rfactor rterm | rfactor

rfactor → rprimary rfactor'

rfactor' → * rfactor' | ∈

rprimary → a|b

This grammar is not suitable for top-down parsing since there are more than one alternative to derive a non-terminal, e.g. for rexp, parse will not understand which derivation among, rterm + rexp or rterm takes place. Hence, left factored the grammar we obtain,

rexp → rterm rexp'

rexp' → + rexp |∈

rterm → rfactor rterm'

rterm' → rterm |∈

rfactor → rprimary rfactor'

rfactor' → * rfactor' |∈

rprimary → a|b

This grammar is suitable for top-down parsing.

Note: Left-factoring avoids a backtracking.

3.3 RECURSIVE DESCENT PARSING [April 16, 17, 18, 19, Oct. 16, 17, 18]

- Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
- Recursive descent parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.
- Recursive descent parser is a top-down parser.

3.3.1 Definition

- A parse that uses a set of recursive procedures to recognize its input without backtracking is called as Recursive Descent Parsing (RDP).
- A recursive descent parsing program consists of a set of producers, one for each non-terminal execution starts with a start symbol procedure.
- The execution ends or halts when procedure body scans the entire input string.
- General recursive-descent may require, backtracking, means it may require repeated scans over the input.
- A typical procedure for a non-terminal in a top-down parser is as follows:

```
void S()
{ choose an s-production
    S → X1 X2 X3 ------ Xn;
  for (i = 1 to n)
  {   if (xi is a nonterminal)
    call procedure Xi();
    else if (Xi is current input symbol)
      advance the input to next symbol
    else
      error
  }
}
```

3.3.2 Implementation of Recursive Descent Parser using Recursive Procedures

- Top-down parsing is often implemented as a set of recursive procedures, one for each non-terminal in the grammar and is then called recursive descent parsing.
- A Recursive Descend (RD) parser is a variant of top-down parsing without backtracking. It uses a set of recursive procedures to perform parsing.
- The advantages of recursive descent parsing are its simplicity and generality. It can be implemented in any language supporting recursive procedures.
- The term "recursive descent" refers to a parsing method which is conveniently implemented using recursive procedures calling each other.
- Following are the steps to construct Recursive Descent parser:
  1. Remove left-recursion of the grammar.
  2. Find out left-factoring grammar if required.
  3. Apply procedure of above program code (Section 3.3.1) for all non-terminal symbol.

Example 6: Construct Recursive Descent Parser for the following:

$E \rightarrow VE'$

$E' \rightarrow +VE' | \in$

$V \rightarrow TV'$

$V' \rightarrow *TV' | \in$

$T \rightarrow (E) \mid id$

Solution: The grammar is without left-recursion and left-factored. This grammar is suitable for non-backtracking recursive descent parser. We can write procedures, which are mutually recursive to recognize arithmetic expression for this grammar. The procedures of this parser are as follows:

```
E( ) /* procedure for production E  →  VE' */
{
    V( );
    EPRIME ( );
};
procedure EPRIME ( ) /* procedure for production E' →  +VE'  */
{
    if input symbol = '+' then
    {
        ADVANCE ( );
        V ( );
        EPRIME ( );
    };
};
V ( ) /*  procedure for V →  TV' */
{
    T ( );
    VPRIME( );
};
VPRIME ( ) /* procedure for V' →   * TV' */-
{
    if input symbol = '*'
    {
    ADVANCE ( );
    T ( );
```

```
    VPRIME ( );
        };
};
T ( ) /* procedure for T →  (E)|id */
{
    if input_symbol = 'id'
        ADVANCE ( );
else
    if input_symbol = '('
    {
        ADVANCE ( );
        E ( );
        if input_symbol = ')'
            ADVANCE ( )
            else ERROR ( )
    }
      else ERROR ( )
};
```

Fig. 3.14: Recursive Descent Parsing for Grammar 3.6

Example 7: Write recursive descent parser (RDP) for the CFG given below:

$$S \rightarrow aBAab|aBb$$
$$A \rightarrow Aa|b$$
$$B \rightarrow bB|b$$

Solution: Here, A-production have left-recursion.

After removing left-recursion grammar becomes.

$$S \rightarrow aBAab|aBb$$
$$A \rightarrow bA'$$
$$A' \rightarrow aA'|\in$$
$$B \rightarrow bB|b$$

Now, for the grammar we need to apply left-factoring for S-productions. Thus, we get grammar as,

$$S \rightarrow aBS'$$
$$S' \rightarrow Aab|b$$
$$A \rightarrow bA'$$
$$A' \rightarrow aA'|\in$$
$$B \rightarrow bB|b$$

Now, the set of procedures/functions given below gives RDP.

```
S()  /* function */
{
  if input_symbol = 'a'
  {
    ADVANCE();
    B( );
    SPRIME();
  }
}
SPRIME() /* function S'( ) */
{
  if input_symbol = 'b'
  {
    ADVANCE( );
  }
  else
    }
      A();
      if input_symbol = 'a'
      {
        ADVANCE();
        if input_symbol = 'b'
        {
          ADVANCE();
        }
      }
  }
}
A( ) /* function A() */
{
  if input_symbol = 'b'
  {
    ADVANCE();
    APRIME();
```

```
    }
}
    APRIME()           /* function A'( ) */
    {
      if input_symbol = 'a'
      {
        ADVANCE();
        APRIME();
      }
        else
          error()
    }
B()    /*  function B() */
{
    if input_symbol = 'b'
    {
      ADVANCE();
      B();
    }
      else
        if input_symbol = 'b'
        {
          ADVANCE();
        }
    }
```

Example 8: Construct recursive descent parser for the following grammar:

$S \rightarrow aA \mid AB$

$A \rightarrow BA \mid a$

$B \rightarrow SA \mid b$

Solution: This grammar has no left-recursion and no left factored.

Recursive Descent Parser is as follows:

```
S( )      /* Procedure for S */
{ if (inputsymbol='a')
  { ADVANCE( )
    A( );
```

```
  }
  else
      { A( );
      B( );
      }
}
A( )  /* procedure for A */
{ if (inputsymbol='a')
   ADVANCE( )
  else
   { B( );
   A( );
   }
}
B( )  /* procedure for B */
{ if(inputsymbol='b')
   ADVANCE( )
else
   { S( );
   A( );
   }
}
```

Example 9: Construct recursive descent parser for the following grammar:

$$A \rightarrow 0A0 | A1 | AA | 1$$

Solution: Eliminating left-recursion the grammar is,

$A \rightarrow 0A0A' | 1A'$ where α_1 is 1, α_2 is A, β_1 is 0A0, β_2 is 1.

$A' \rightarrow 1A' | AA' | \in$

The Recursive Descent Parser is as follows:

```
A{ }  /* procedure for A */
{ if inputsymbol='0'
   { ADVANCE( );
     A( );
     if inputsymbol = '0'
```

```
        {   ADVANCE( );
         APRIME( );
        }
      }
    else
      { if inputsymbol='1'
        {   ADVANCE( );
         APRIME( );
        }
      else
         error( );
  }
    APRIME( ) /* procedure for A' */
    { if inputsymbol = '1'
      { ADVANCE( );
        APRIME( );
      }
    else
      { A( );
        APRIME( ) ;
      }
  }
```

Example 10: Construct recursive descent parser for the following grammar:

$S \rightarrow Aab \mid aBb$

$A \rightarrow Aa \mid b$

$B \rightarrow bB \mid b$

Solution: Eliminate left-recursion from A-productions

$S \rightarrow Aab \mid aBb$

$A \rightarrow bA'$

$A' \rightarrow aA' \mid \in$

$B \rightarrow bB \mid b$

Find left-factoring from B-productions, we get

$S \rightarrow Aab \mid aBb$

$A \rightarrow bA'$

$A' \rightarrow aA' | \in$

$B \rightarrow bB'$

$B' \rightarrow B | \in$

This grammar is now suitable for Recursive Descent Parsering. Recursive Descent Parser is as follows:

```
S( )   /* procedure for S */
{ if inputsymbol='a'
  { ADVANCE( )
    B( );
    if inputsymbol='b'
      {
      ADVANCE( );
      }
      else
    { A( )
      if inputsymbol='a'
      {   ADVNACE( );
       if inputsymbol = 'b'
       {  ADVANCE( );
       }
       else
          error( );
       }
      else
       error( );
  }
A ( )  /* procedure for A */
  {
    if inputsymbol = 'b'
    { ADVANCE ( );
      APRIME ( );
    }
      else
       error ( )
```

```
}
APRIME( )  /* procedure for A' */
{ if inputsymbol='a'
  { ADVANCE ( );
    APRIME( );
  }
  else
  error
}
B( )    /* procedure for B*/
{
  if inputsymbol='b'
  { ADVANCE( );
    BPRIME( );
  }
  else
    error( )
}
  BPRIME    /* procedure for B' */
{
  B( );
}
```

Example 11: Construct recursive descent parser for the following grammar:

$S \rightarrow iStSeSf|iStSf|0$

Solution: Eliminate left-factoring we get,

$S \rightarrow iStSS'|0$

$S' \rightarrow esf|f$

Now Recursive Descent Parser is as follows:

```
S( )
{
  if inputsymbol='i'
  { ADVANCE( );
    S( );
```

```
        if inputsymbol='t'
         {  ADVANCE( );
            S( );
            SPRIME( );
         }
         else
            error( );
      }
      else
        if inputsymbol='0'
        {   ADVANCE( );
        }
        else
         error( );
    }
    SPRIME( )
    { if inputsymbol='e'
      { ADVANCE( );
        S( );
        if inputsymbol='f'
         {  ADVNACE( );
         }
        else
         error( );
      }
      else
        if inputsymbol='f'
        {   ADVANCE( );
        }
        else
         error( );
      }
  }
```

3.4 PREDICTIVE PARSER

- By eliminating left-recursion from the grammar and left-factoring the resulting grammar, we can obtain a grammar that can be parsed by recursive descent parser that needs no backtracking is a predictive parser.
- The efficient method of implementing recursive descent parsing is predictive parser. Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string.
- Predictive parsers are top-down parsers. A predictive parser is a recursive descent parser that does not require backtracking.
- Predictive parsing is a special form of recursive-descent parsing in which the look-ahead symbol unambiguously determines the procedure selected for each non-terminal.
- The sequence of procedures called in processing the input implicitly defines a parse tree for the input.
- The efficient method of implementing recursive-descent parsing is predictive parser.

Predictive Parsing Algorithm:

1. Let 'a' be the first input symbol of input string w and X be the top stack symbol.
   where X is non-terminal.
2. while (X ≠ $)/* stack is not empty */
   { if (X = a) POP the stack
      Let, a be the next symbol of w_i,
      else if (X is a terminal) error( );
      else if (M[X, a] is an error entry) error( );
      else if (M[x, a] = x → y_1 y_2 ---- y_n)
      { output the production X → y_1 y_2 ---- y_n;
      POP the stack
      push y_n, y_{n-1}, ---- y_1, onto the stack, with y_1 on top.
     }
     X be the top of the stack
   }

- The predictive parser using stack is shown in Fig. 3.14 for the input string id + id * id using predictive parsing algorithm.
- Consider grammar of Example 6.

$$E \rightarrow VE'$$
$$E' \rightarrow +VE' \mid \in$$

$V \rightarrow TV'$

$V' \rightarrow * TV' \mid \in$

$T \rightarrow (E) \mid id$

Stack	Input	Output
$ E	id + id * id $	
$ E'V (here V is TOS: Top of stack)	id + id * id $	E → VE'
$ E'V'T	id + id * id $	V → TV'
$ E'V' id	id + id * id $	T → id
$ E'V'	+ id * id $	(pop, if TOS = input symbol)
$ E'	+ id * id $	V' → ∈
$ E'V +	+ id * id $	E' → + VE'
$ E'V	id * id $	pop
$ E'V'T	id * id $	V → TV'
$ E'V' id	id * id $	T → id
$ E'V'	* id $	pop
$ E'V'T *	* id $	V' → * TV'
$ E'V'T	id $	pop
$ E'V' id	id $	T → id
$ E'V'	$	pop
$ E'	$	V' → ∈
$	$	E' → ∈

Fig. 3.14: Predictive Parsing using Stack

- Here, all derivations are leftmost derivations, here the input is scan from left to right. Hence the TOS is always the left symbol of the rightmost sentential form of the production rule.

3.4.1 LL Parser

- An LL parser is a top-down parser for a subset of the Context-Free Grammars (CFGs).
- An LL parser parses the input from left to right and constructs a leftmost derivation of the sentence. The class of grammars which are parsable in this way is known as the LL grammars.
- An LL parser is called an LL(k) parser if it uses k tokens (or input strings) of look ahead when parsing a sentence.
- If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking, then it is called an LL(k) grammar.

- Of these grammars, LL(1) grammars, although fairly restrictive are very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions.
- A CFG whose parsing table has no multiply defined entries is called an LL(1) grammar. Here, the "1" signifies the fact that the LL parser uses one input symbol of look ahead to decide its next move.
- An LL parser (Left-to-right, Leftmost derivation) is a top-down parser for a restricted context-free language. It parses the input from Left to right, performing Leftmost derivation of the sentence.
- An LL parser is called an LL(k) parser if it uses k tokens of lookahead when parsing a sentence. A grammar is called an LL(k) grammar if an LL(k) parser can be constructed from it.
- A formal language is called an LL(k) language if it has an LL(k) grammar. The set of LL(k) languages is properly contained in that of LL(k+1) languages, for each $k \geq 0$.

3.4.2 LL(1) Parser [April 16, 17, 18, Oct. 16, 17, 18, 19]

- An LL parser parses the input from left to right and constructs a leftmost derivation of the sentence are called LL(1) parser.
- In LL(1) stands for Left-to-right parse, Leftmost derivation, 1-symbol lookahead.
- The LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads.
- Generally k = 1, so LL(k) may also be written as LL(1).

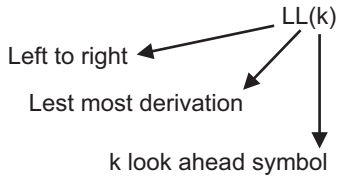


Fig. 3.15

- **Definition:** Predictive parser can be constructed for a class of LL(1) grammar. The first 'L' stands for scanning the input from left to right and second 'L' stands for producing leftmost derivation and '1' stands for using one input symbol of look-a-head at each step to make parsing action decisions.
- The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive descent is also known as LL(1) Parser.

3.4.3 Parser Model for LL(1)

- Fig. 3.16 shows structure/model of LL(1) parser. Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree.

- Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed.
- The parser refers to the parsing table to take any decision on the input and stack element combination.
- The predictive parsers have the following components:
  1. **Input string** which is to be parsed.
  2. **Stack** consists of sequence of grammar symbols i.e. non-terminals and terminals of the grammar.
  3. **Predictive parsing table** which is 2D array [non-terminals, terminals]. It is a tabular implementation of the recursive descent parsing, where a stack is maintained by the parser rather than the language in which parser is written.
  4. An output stream.
- The Fig. 3.16 shows the model of predictive parser.

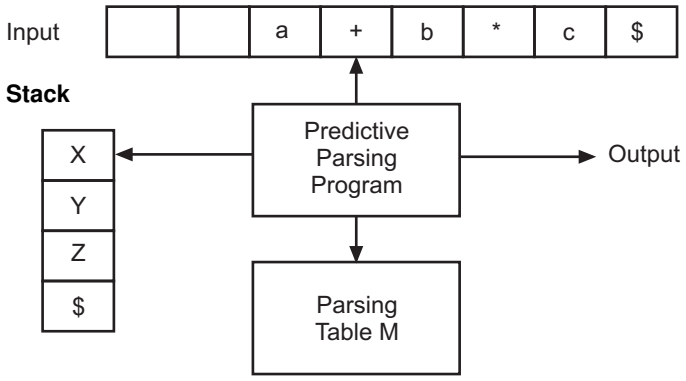


Fig. 3.16: Model of Table-driven Predictive Parser

- To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.
- Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input.

3.4.4 Implementation of Predictive Parser LL(1)

- The LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser.
- Here, we discuss a parser that determines that given string can be generated from a given grammar (or parsing table) or not.
- Let given grammar is G = (V, T, S, P)

  where V-variable symbol set, T-terminal symbol set, S- start symbol, P- production set.

LL(1) Parser algorithm:

Input: 1. stack = S //stack initially contains only S.

2. input string = w$

   where S is the start symbol of grammar, w is given string, and $ is used for the end of string.

3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output: determines that given string can be produced by given grammar(parsing table) or not, if not then it produces an error.

Steps:

```
while(stack is not empty)
{
  // initially it is S
A = top symbol of stack;
// initially it is first symbol in string, it can be $ also
r = next input symbol of given string;

if (A∈T or A==$)

{
  if(A==r)
  {
    pop A from stack;
    remove r from input;
  }
  else
    ERROR();
  }

  else if (A∈V)

  {
    if(PT[A,r]= A →B1B2....Bk)
    {
       pop A from stack;
```

```
            // B1 on top of stack at final of this step
            push Bk,Bk-1......B1 on stack
        }
        else if (PT[A,r] = error())
            error();
    }
}
```

3.4.5 Construction of Parse Table and LL(1) Parse Table [Oct. 17]

- Parse table is a two-dimensional array where each row is leveled with non terminal symbol and each column is marked with terminal or special symbol $. Each cell holds a production rule.
- Now, we construct the predictive parsing table. For the construction of a predictive parsing table, we need two functions associated with a grammar G.
- These functions are FIRST and FOLLOW, which are required to write the entries of parsing table.

1. FIRST: [Oct. 17]

- FIRST is computed for all grammar symbols that is for non-terminals and terminals. Following rules are applied to find FIRST (X).
  - (i) If X is terminal symbol, then FIRST (X) = {X} i.e. the FIRST of terminal is terminal itself.
  - (ii) If $X \rightarrow \in$ is a production, then FIRST (X) = {$\in$}.
  - (iii) If X is non-terminal symbol and it is having production whose leftmost symbol of RHS is terminal then the terminal symbol is in FIRST (X). i.e. $X \rightarrow a\alpha$ where $a \in T$ then FIRST (X) = {a}.
  - (iv) If X is non-terminal symbol having production in the form $X \rightarrow$ AB ... Z. Here, the RHS contains sequence of non-terminals.

    Now, if production $A \rightarrow a$, then FIRST (X) = {a}.

    If $A \rightarrow \in$ then FIRST (X) = FIRST (B).

    If $B \rightarrow \in$ then FIRST (X) = FIRST (C) and so on.

    If AB ... Z $\overset{*}{\Rightarrow} \in$ then FIRST (X) = {$\in$}.

2. FOLLOW:

- Follow is computed only for non-terminals.
- Following rules are applied to find Follow (A):
  - (i) If S is start symbol, then add $ to FOLLOW (S).
  - (ii) If there is a production $A \rightarrow \alpha B\beta$, $\beta \neq \in$, then everything in FIRST (β) is in FOLLOW (B) except $\in$.

(iii) If there are productions, $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST (B) contains $\in$ (i.e. $\beta \overset{*}{\Rightarrow} \in$) then everything in FOLLOW (A) is in FOLLOW (B).

Example 12: Consider Grammar 3.2:

$$E \rightarrow VE'$$
$$E' \rightarrow +VE' \mid \in$$
$$V \rightarrow TV'$$
$$V' \rightarrow *TV' \mid \in$$
$$T \rightarrow (E) \mid id$$

Solution: To compute FIRST:

FIRST (E) = FIRST (V) = FIRST (T) = {(, id}
FIRST (E') = {+, $\in$}
FIRST (V') = {*, $\in$}
FIRST (+) = {+}
FIRST (*) = {*}
FIRST (id) = {id}
FIRST (( ) = {(}
FIRST ( )) = {)}

Now, we compute FOLLOW as follows:

To compute FOLLOW (E).

Search 'E' on RHS of productions we obtain,

$T \rightarrow (E)$ This is in the form
$A \rightarrow \alpha B \beta$ where, B = E, α = (, β =)

Apply rule 2 and rule 1 we get,

FOLLOW (E) = {), $}
FOLLOW (E') = ?

Consider two productions where E' is on RHS.

$E \rightarrow VE'$
$E' \rightarrow +VE'$

apply rule 3. We obtain.

FOLLOW (E') = FOLLOW (E) = {), $}

Now, compute FOLLOW (V).

$E \rightarrow VE'$
$E \rightarrow +VE'$
} rule 2 and rule 3 both are applicable

$\therefore$ FOLLOW (V) = {+,), $}

Similarly, we compute

FOLLOW (V') = FOLLOW (V) = {+,), $}

FOLLOW (T) = {+, ∗,), $}

Therefore, Follow of above grammar is,

FOLLOW (E) = FOLLOW (E') = {), $}

FOLLOW (V) = FOLLOW (V') = {+,), $}

FOLLOW (T) = {+, ∗,), $}

Example 13: Construct FIRST and FOLLOW for the following grammar:

$S \rightarrow iCtSS' \mid a$

$S' \rightarrow eS \mid \in$

$C \rightarrow b$

Solution:

FIRST (i) = {i}

FIRST (t) = {t}

FIRST (e) = {e}

FIRST (a) = {a}

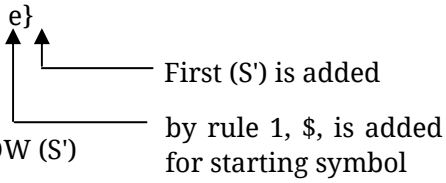
FIRST (b) = {b}

FIRST (S) = {i, a}

FIRST (S') = {e, ∈}

FIRST (C) = {b}

FOLLOW (S) = {$, e}



To compute FOLLOW (S')

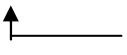
Consider $S \rightarrow iCtSS'$

FOLLOW (S') = FOLLOW (S)

$\therefore$ FOLLOW (S') = {$, e)

To compute FOLLOW (C)

Consider $S \rightarrow iCtSS'$



FOLLOW (C) = FIRST t = t

Hence,

FOLLOW (S) = {$, e}
FOLLOW (S') = {$, e}
FOLLOW (C) = {t}

Example 14: Find FIRST and FOLLOW of the following grammar.

$S \rightarrow BC \mid AB$

$A \rightarrow aAa \mid \in$

$B \rightarrow bAa$

$C \rightarrow \in$

Solution: To compute FIRST:

FIRST (a) = {a}

FIRST (b) = {b}

FIRST (S) = {b, a}

FIRST (A) = {a, $\in$}

FIRST (B) = {b}

FIRST (C) = {$\in$}

To compute FOLLOW:

FOLLOW (S) = {$}

To find FOLLOW (A)

Consider, (1) $S \rightarrow AB$

FOLLOW (A) = FIRST (B) = {b}

Consider, (2) $A \rightarrow aA$

FOLLOW (A) = first (a) = {a}

Consider (3) $B \rightarrow bAa$

FOLLOW (A) = first (a) = {a}

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

To find FOLLOW (B),

Consider, $S \rightarrow BC$

FOLLOW (A) = FOLLOW (S) ($\because C \Rightarrow \in$)

Consider, $S \rightarrow AB$

FOLLOW (B) = FOLLOW (S)

$\therefore$ FOLLOW (B) = {$}

FOLLOW (C) = FOLLOW (S) = {$}

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

FOLLOW (A) = {a, b}

FOLLOW (B) = {$}

FOLLOW (C) = {$}

Example 15: Find the first and follow sets of the following grammar:

$$S \rightarrow \$\#$$
$$E \rightarrow E - T \mid T$$
$$T \rightarrow F \uparrow T \mid F$$
$$F \rightarrow (E) \mid id$$

Solution:

FIRST (#) = {#} FIRST (↑) = {↑}

FIRST (( ) = {(} FIRST (–) = {–}

FIRST (id) = {id) FIRST ( )) = {)}

FIRST (S) = FIRST (E) = FIRST (T)

= FIRST (F) = {(, id}

FOLLOW (S) = {$}

FOLLOW (E) = {#, –,)}

FOLLOW (T) = {#, –,)}

FOLLOW (F) = {#, –,), ↑}

3.4.6 Construction of LL(1) Parse Table

- In LL(1) parsing the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree.
- And finally, the 1 represents the number of look-ahead, which means how many symbols are we going to see when you want to make a decision.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

- **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
- **Follow():** What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

- First(α) and for each terminal in First(α), make entry A –> α in the table.
- If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.
- If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.

For example, consider the Grammar:

E --> TE'

E' --> +TE' | ε

T --> FT'

T' --> *FT' | ε

F --> id | (E)

*ε denotes epsilon.

Find their First and Follow sets:

	First	Follow
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/ε	{ +, ε }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
T’ –> *FT’/ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

- Now, the LL(1) Parsing Table is:

	id	+	*	(	)	$
E	E –> TE’			E –> TE’		
E’		E’ –> +TE’			E’ –> ε	E’ –> ε
T	T –> FT’			T –> FT’		
T’		T’ –> ε	T’ –> *FT’		T’ –> ε	T’ –> ε
F	F –> id			F –> (E)		

- As we can see that all the null productions are put under the Follow set of that symbol and all the remaining productions are lie under the First of that symbol.

3.4.7 Parsing of a String using LL(1) Parse Table

- The LL(1) is a top-down parser. For a given input string and starting from the start symbol of the given grammar, top-down parsers tries to create a parse tree using depth-first methods.
- Hence, top-down parsers can be realized as searching methods for leftmost derivations for input strings.
- The LL(1) is predictive in the sense that it is able to decide the rule to apply based only on the currently processed non-terminal and following input symbol. This is only possibly if the grammar has a specific form called LL(1).
- The LL(1) parser uses a stack and a parse table called LL(1) table. The stack stores the productions that the parser is to apply.
- The LL(1) table columns correspond to terminals of the grammar plus an extra column for the $ symbol (the marker for the input symbol end).

- Rows of LL(1)-table correspond to non-terminals of the LL(1) grammar. Each cell of the LL(1) table is either empty or including a single grammar production.
- Therefore the table includes the parser actions to be taken based on the top value of the stack and the current input symbol.

For example, consider the Grammar,

S --> A | a

A --> a

Find their First and Follow sets:

	First	Follow
S –> A/a	{ a }	{ $ }
A –>a	{ a }	{ $ }

Parsing Table:

	a	$
S	S –> A, S –> a	
A	A –> a	

- Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

3.4.8 Construction of Predictive Parsing Table

- Now, for the above grammar we can construct predictive parsing table. It is constructed in the following manner: Let us consider parsing table as T.
  1. We consider each production of grammar.
  2. If the terminal symbol is the first symbol of the RHS then add that production in the parsing table. For example: If the production is $A \rightarrow a\alpha$ where $a \in T$ then add $A \rightarrow a\alpha$ is parsing table for the entry T [A, a].
  3. If the non-terminal symbol is the first symbol of the RHS then for all FIRST of that non-terminal, the production entry is added in table for LHS non-terminal.

     For example, $A \rightarrow BC$ and FIRST (B) = {b} then at T [A, b], entry $A \rightarrow BC$ is added.
  4. If $\in$ is in FIRST (X) for production $A \rightarrow X$, then find the Follow (A) and add $A \rightarrow X$ to T [A, b] where b is a terminal in FOLLOW (A). Here if $\in$ is FIRST (X) and $ is in FOLLOW (A) then add $A \rightarrow X$ to T [A, $].
  5. The undefine entries of T are error.

Example 16: Consider the following grammar and construct a parsing table as follows:

Grammar is,

$$E \rightarrow VE'$$
$$E' \rightarrow +VE' \mid \in$$
$$V \rightarrow TV'$$
$$V' \rightarrow TV' \mid \in$$
$$T \rightarrow (E) \mid id$$

Solution: 1. Start with first production:

$$E \rightarrow VE'$$

$$\text{FIRST}(V) = \{(, id\}$$

$\therefore$ At T [E, ( ] and T [E, id] make entry $E \rightarrow VE'$.

2. $E' \rightarrow + VE'$

T [E', +] make entry $E' \rightarrow + VE'$.

3. $E' \rightarrow \in$

$$\text{Follow}(E') = \{), \$ \}$$

T [E',)] and T [E', $] make entry $E' \rightarrow \in$.

Similarly, we can fill parsing table entries for all productions. The predictive parsing table of Grammar 3.2 is shown in Fig. 3.17.

Non-Terminals	Input symbols					
	+	*	(	)	id	$
E			$E \rightarrow VE'$		$E \rightarrow VE'$	
E'	$E' \rightarrow + VE'$			$E' \rightarrow \in$		$E' \rightarrow \in$
V			$V \rightarrow TV'$		$V \rightarrow TV'$	
V'	$V' \rightarrow \in$	$V' \rightarrow * TV'$		$V' \rightarrow \in$		$V' \rightarrow \in$
T			$T \rightarrow (E)$		$T \rightarrow id$	

Fig. 3.17: Predictive Parsing Table

3.4.9 LL(1) Grammars (April 16)

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1). Hence, the above grammar is LL(1) grammar.
- The LL(1) grammars are suitable for top-down parsing. The LL(1) grammar is not left-recursive and not ambiguous.
- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha | \beta$ are two distinct productions of G, the following conditions hold.
  1. Both α and β should not start or derive with terminal a.
  2. Atmost one of α and β can derive the empty string.
  3. If $\beta \overset{*}{\Rightarrow} \in$, then α does not derive any string starting with a terminal in FOLLOW (A). If $\alpha \overset{*}{\Rightarrow} \in$, then β does not derive any string starting with a terminal in FOLLOW (A).

Example 17: Construct predictive parsing table and check whether the following grammar is LL (1)?

$$S \rightarrow AB$$

$$A \rightarrow BS | b | \in$$

$$B \rightarrow AS | a$$

Solution: The above grammar is left-recursive since we have recursive in B-production (not immediate).

$$B \rightarrow AS|a \text{ (Substitute A-production in B-productions)}$$
$$B \rightarrow BSS|bS|S|a, \text{ After removing left-recursion}$$
$$B \rightarrow bSB'|SB'|aB'$$
$$B' \rightarrow SSB'|\in$$

Hence, grammar without left-recursion is,

$$S \rightarrow AB$$
$$A \rightarrow BS|b|\in$$
$$B \rightarrow bSB'|SB'|aB'$$
$$B' \rightarrow SSB'|\in$$

Now, calculate FIRST and FOLLOW sets,

$$\text{FIRST (S)} = \{b, a, \in\}$$
$$\text{FIRST (A)} = \{b, a, \in\}$$
$$\text{FIRST (B)} = \{b, a, \in\}$$
$$\text{FIRST (B')} = \{b, a, \in\}$$
$$\text{FOLLOW (S)} = \{\$, a, b\}$$
$$\text{FOLLOW (A)} = \{\$, a, b\}$$
$$\text{FOLLOW (B)} = \{\$, a, b\}$$
$$\text{FOLLOW (B')} = (\$, a, b\}$$

Now, we construct predictive parsing table.

Non-terminal	Input Symbols		
	a	**B**	**$**
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow BS$ $A \rightarrow \in$	$A \rightarrow BS$ $A \rightarrow b$, $A \rightarrow \in$	$A \rightarrow BS$ $A \rightarrow \in$
B	$B \rightarrow aB'$ $B \rightarrow SB'$	$B \rightarrow bSB'$ $B \rightarrow SB'$	$B \rightarrow SB'$
B'	$B' \rightarrow SSB'$ $B' \rightarrow \in$	$B' \rightarrow SSB$ $B' \rightarrow \in$	$B' \rightarrow SSB'$ $B' \rightarrow \in$

Fig. 3.18: Predictive Parsing Table

Since, parsing table has multiply defined entries, the given grammar is not LL(1).

Example 18: Check whether following grammar is LL(1) or not?

$S \rightarrow BC \mid AB$

$A \rightarrow aAa \mid \in$

$B \rightarrow bAa$

$C \rightarrow \in$

Solution: Find FIRST and FOLLOW of the grammar.

FIRST of the grammar:

FIRST (a) = {a}

FIRST (b) = {b}

FIRST (S) = {a, b}

FIRST (A) = {a, ∈}

FIRST (B) = {b}

FIRST (C) = {∈}

FOLLOW of the grammar:

FOLLOW (S) = {$}

FOLLOW (A) = {a, b}

FOLLOW (B) = {$}

FOLLOW (C) = {$}

Now construct the parsing table:

Non-Terminal	Terminal		
	a	B	$
S	S → AB	S → BC	
A	A → aAa A → ∈	A → ∈	
B		B → aAa	
C			C → ∈

Above grammar is not LL(1) grammar because there is multiple entry in cell M [A, a] of the predictive parsing table.

Example 19: Check following grammar is LL(1) or not ?

S → AB

A → BS | b | ∈

Solution: (i) No production rule from B ∴ B is useless

(ii) We can eliminate,

S → AB and A → BS

(iii) Now S is useless ∴ Grammar is useless,

(iv) It is not LL(1).

Example 20: Check following grammar is LL (1) or not ?

A → AcB | cD | D

B → bB | id

D → DaB | BbB | B

Solution: This grammar is not LL (1) because grammar is having left-recursion in A-production and D-production. Eliminating left-recursion we get,

$$A \rightarrow cDA' \mid DA'$$
$$A' \rightarrow cBA' \mid \in$$
$$B \rightarrow bB \mid id$$
$$D \rightarrow BbBD' \mid BD'$$
$$D' \rightarrow aBD' \mid \in$$

Since, in D-production both productions are having same start symbol.

$\therefore$ Apply left-factoring

$$\left.\begin{array}{l} D \rightarrow BD'' \\ D'' \rightarrow bBD' \mid D' \end{array}\right\} \text{ D-production after left-factoring}$$

Hence grammar is,

$$A \rightarrow cDA' \mid DA'$$
$$A' \rightarrow cBA' \mid \in$$
$$B \rightarrow bB \mid id$$
$$D \rightarrow BD''$$
$$D'' \rightarrow bBD' \mid D'$$
$$D' \rightarrow aBD' \mid \in$$

Now, compute FIRST and FOLLOW:

FIRST (a) = {a}
FIRST (c) = {c}
FIRST (b) = {b}
FIRST (id) = {id}
FIRST (A) = {c, b, id}
FIRST (A') = {c, $\in$}
FIRST (B) = {b, id}
FIRST (D) = {b, id}
FIRST (D') = {a, $\in$}
FIRST (D'') = {b, a, $\in$}
FOLLOW (A) = {$)
FOLLOW (B) = {c, b, a, $}
FOLLOW (A') = {$}
FOLLOW (D) = {c, $}
FOLLOW (D') = {c, $}
FOLLOW (D'') = {c, $}

Now construct predictive parsing table.

Non-terminal	a	b	C	id	$
A		$A \rightarrow DA'$	$A \rightarrow cDA'$	$A \rightarrow DA'$	$A' \rightarrow \in$
A'			$A' \rightarrow cBA'$		$A' \rightarrow \in$
B		$B \rightarrow bB$		$B \rightarrow id$	
D		$D \rightarrow BD''$		$D \rightarrow BD''$	
D'	$D' \rightarrow aBD'$		$D' \rightarrow \in$		$D' \rightarrow \in$
D''	$D'' \rightarrow D'$	$D'' \rightarrow bBD'$			$D'' \rightarrow D'$

There is no multiply entry in the table.

Therefore grammar is LL (1).

Example 21: Check whether following grammar is LL (1).

$S \rightarrow abAB \mid Abc$

$A \rightarrow Ba \mid \in$

$B \rightarrow bA \mid Aa \mid \in$

Solution: Grammar contains indirect left-recursion in B-production after substituting A-production is B-production.

$S \rightarrow abAB \mid Abc$

$A \rightarrow Ba \mid \in$

$B \rightarrow bBa \mid ba \mid Baa \mid a \mid \in$

After eliminating left-recursion we get,

$S \rightarrow abAB \mid Abc$

$A \rightarrow Ba \mid \in$

$B \rightarrow bBaB' \mid baB' \mid aB' \mid B'$

$B' \rightarrow aaB' \mid \in$

Now, compute FIRST and FOLLOW:

FIRST (S) = {a, ∈, b} FOLLOW (S) = {$}

FIRST (A) = {a, b, ∈} FOLLOW (A) = {a, b}

FIRST (B) = {a, b, ∈} FOLLOW (B) = {a, $}

FIRST (B') = {a, ∈} FOLLOW (B') = {a, $}

Now construct parsing table,

Non-terminal	a	B	$
S	$S \rightarrow abAB$	$S \rightarrow Abc$	$S \rightarrow Abc$
A	$A \rightarrow Ba$ $A \rightarrow \in$	$A \rightarrow \in$	$A \rightarrow \in$
B	$B \rightarrow aB'$ $B \rightarrow B'$	$B \rightarrow aBaB'$ $B \rightarrow baB'$	
B'	$B' \rightarrow aaB'$ $B' \rightarrow \in$		$B' \rightarrow \in$

There are multiple entries in the table, hence above grammar is not LL (1).

Example 22: Check following grammar is LL (1) or not ?

$S \rightarrow a \mid \wedge \mid (R)$

$T \rightarrow S, T \mid S$

$R \rightarrow T$

Solution: Apply left-factoring for T-production, we get

$S \rightarrow a \mid \wedge \mid (R)$

$T \rightarrow ST'$

$T' \rightarrow , T \mid \in$

$R \rightarrow T$

Compute FIRST and FOLLOW

FIRST (S) = {a, ^, (} FOLLOW (S) = {$, ,}

FIRST (T) = {a, ^, ( ) FOLLOW (T) = {) }

FIRST (T') = {, , ∈} FOLLOW (T') = {) }

FIRST (R) = {a, ^, (} FOLLOW (R) = {) }

Now construct parsing table,

Non-terminals	Terminals					
	a	^	(	)	,	$
S	$S \rightarrow a$	$S \rightarrow \wedge$	$S \rightarrow (R)$			
T	$T \rightarrow ST'$	$S \rightarrow ST'$	$T \rightarrow ST'$			
T'					$T' \rightarrow , T$	$T' \rightarrow \in$
R	$R \rightarrow T$	$R \rightarrow T$	$R \rightarrow T$			

- The above grammar is LL (1) because there is no multiple entry in the table.

3.5 BOTTOM-UP PARSING [Oct. 17, 18, April 19]

- Bottom-up parsing constructs a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- Bottom-up parsers begin with the leaves and grow the tree toward the root. At each step, a bottom-up parser identifies a contiguous substring of the parse tree's upper fringe that matches the right-hand side of some production; it then builds a node for the rule's left-hand side and connects it into the tree.
- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.
- Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.
- In bottom-up parsing, we start with input string and using steps of reduction reach to start symbol or distinguished symbol of a grammar.
- It uses sequence of reductions and syntax tree for an input string is developed through a sequence of reductions. If the input string is reduced to the start-symbol, then the string is valid string of a language.
- The idea of bottom-up parsing is to build the derivation tree from the leaves to the root. The parser performs the following actions:
  - It checks the sentential form (terminals and/or non-terminals) of the input string which matches with RHS of production rule. If match is found then sentential form is reduced with a LHS non-terminal of that production. This process is done until no reduction is possible.
  - The bottom-up parsing is also called LR (left-to-right) or shift-reduce parser LR means the grammars scans the input from left to right and generate right most derivation.

Example 23: Consider the grammar:

$$S \rightarrow S - B \mid B$$
$$B \rightarrow B * A \mid A \qquad \textbf{(Grammar 3.3)}$$
$$A \rightarrow id$$

Solution: Consider input string to be parse is $x - y * z$ i.e., $id - id * id$.

The above string is parse using bottom-up parsing as follows:

	Production Used
$id - id * id$	$A \rightarrow id$
$\underline{A} - A * A$	$B \rightarrow A$
$B - A * A$	$S \rightarrow B$
$S - B * A$	$B \rightarrow A$
$S - B$	$B \rightarrow B * A$
S	$S \rightarrow S - B$

We get starting symbol S. Hence, the string is valid string of above grammar.

- Bottom-up parsers construct parse trees starting from the leaves and work up to the root.
- The Fig. 3.19 shows the bottom-up parser can be divided into two types namely, Operator precedence parser and LR parser.
- LR parser further divided into:
  1. Simple LR i.e. SLR(1).
  2. Cannonical LR
  3. Lookahead LR (LALR).

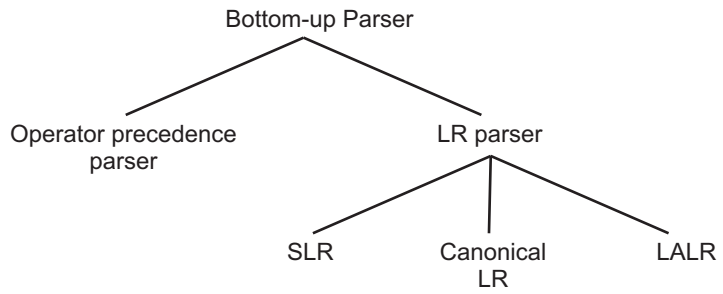


Fig. 3.19: Types of Bottom-up Parsing

3.6 OPERATOR PRECEDENCE PARSER [April 17, 19, Oct. 18]

- An operator precedence parser is a bottom-up parser that interprets an operator-precedence grammar.
- An operator precedence grammar is a context-free grammar that has the property that no production has either an empty right hand side (null productions) or two adjacent non-terminals in its right-hand side.
- For small class of grammars, we can easily construct shift-reduce parsers are operator precedence parsers. Normally, for the grammar having expressions, operator precedence parsers are used.
- Suppose if the grammar have expression with many operators like +, –, ∗, /, ↑. The expression consists of many operators is solved by giving precedence and associativity to the operator. Such grammar of expression is called operator grammar.
- The technique is used to parse operator grammar is called operator precedence parsers.
- The operator precedence parsing uses three precedence relations: < (less than), = (equal to), and > (greater than) between two terminals.
- For example, if we consider two terminals a and b. The expression is (a + b) (a ∗ b) since '∗' have higher precedence than '+' we have relation ∗ > + or + < ∗. The expression a ∗ b is solved first.

Definition:

- The operator grammar in which only one precedence relations <·, = or ·> holds between two terminals, and the grammar is not having ∈-production is called an operator-precedence grammar.

  Using Operator-Precedence Relations: Consider the operator grammar of expression is:

  $S \rightarrow S + S \mid S * S \mid a$

  where, a is any identifier.
- We can derive the string a + a * a. Let $ is end marker at both side of the string and the precedence relations are shown in Fig. 3.20.

RHS / LHS	a	+	*	$
a		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Fig. 3.20: Operator Precedence Relation

- We can use stack for operator-precedence parsing as follows:
  1. Let $ is TOS and input string w$ has pointer which scans the input ahead.
  2. If TOS < pointer symbol or TOS = pointer symbol then push input symbol to the stack, (pointer symbol is the symbol pointed by input pointer of input string) and advance the pointer to next symbol.
  3. If TOS > pointer symbol then pop the TOS until TOS < terminal symbol which is most recently popped. Here the string between <· & ·> is handle which is solved first.
  4. Else error.
- Consider above grammar and used precedence relations from Fig. 3.21. Let us use operator precedence parsing to find the handle (right-sentential form) for the string a + a * a.

Stack	Input	Action
$	a + a * a $	push (·˙· TOS < a)
$ a	+ a * a $	TOS > '+' reduce i.e. pop
$ S	+ a * a $	push
$ S +	a * a $	push

$ S + a	* a $	TOS > '*' pop or reduce
$ S + S	* a $	push
$ S + S *	a $	push
$ S + S * a	$	TOS > '$' reduce
$ S + S * S	$	reduce (·˙· * > $)
$ S + S	$	reduce (·˙· + > $)
$ S	$	accept

Fig. 3.21: Operator Precedence Parsing

- Here, the parser finds the right end of the handle. So first S * S is solved and parse tree is generated.

3.6.1 Operator Precedence Relations from Associativity and Precedence

- For the binary operator we can use precedence's associativity. Precedence between operators of a grammar is termed operator precedence.
- Consider language consists of arithmetic expressions. We can define operator precedence relations by considering some rules as follows:

Precedence	Associativity
↑ (highest)	right
*, /	left
+, –	left
unary	non-associative

1. The highest precedence operator have ·> relation with lowest precedence operator. e.g. * ·> +.
2. For equal precedence operator, associativity is considered.

(a) For example, $ a + b + c $ here LHS '+' have higher
$ < + > + > $ precedence than RHS '+'.

We solve expression within <· and ·> first.

Since, the '+' is left associative LHS '+' is greater precedence than RHS '+'.

(b) If operator is right associative then RHS operator have higher precedence than LHS.

Example, $ a ↑ b ↑ c $

$ <· ↑ <· ↑ ·> $ is solved first.

3. The identifier is always higher precedence. Whenever id is found it is reduced with corresponding non-terminal (say S). Also consider the following rules:

(a) The end marker $ has always least precedence.

(b) (< (

(<· id

) <·)

Example, $ ((b + c) * a) $

$ <· (<· (<· + ·>) ·> * ·> $

- There are following two methods for determining what precedence relations should hold between a pair of terminals:
  1. Use the conventional associativity and precedence of operator.
  2. Selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.
- This parser relies on the following three precedence relations ⋗, ≐, ⋖.

a ⋖ b This means a "yields precedence to" b.

a ⋗ b This means a "takes precedence over" b.

a ≐ b This means a "has precedence as" b.

Example 24: Consider the grammar with all these operators:

$$S \rightarrow S + S \mid S - S \mid S * S \mid S/S \mid (S) \mid \text{id}$$

Solution: The operator precedence relations are shown in Fig. 3.22.

RHS / LHS	id	+	–	*	/	(	)	$
id		·>	·>	·>	·>		·>	·>
+	<·	·>	·>	<·	<·	<·	·>	·>
–	<·	·>	·>	<·	<·	<·	·>	·>
*	<·	·>	·>	·>	·>	<·	·>	·>
/	<·	·>	·>	·>	·>	<·	·>	·>
(	<·	<·	<·	<·	<·	<·	≐	
)		·>	·>	·>	·>		·>	·>
$	<·	<·	<·	<·	<·	<·		

Fig. 3.22: Operator Precedence Relations

Note: Handling the unary operator is difficult in precedence parsing. Lexical analyzer should distinguish whether operator is unary or not, before parsing.

3.6.2 Operator Precedence Grammars

- We have already discussed the definition of operator precedence grammars. In Fig. 3.22 we made assumption of the precedence relations.
- Now, we will discuss how to compute precedence relations in operator precedence grammar which is $\in$-tree.
- A grammar that is generated to define the mathematical operators is called operator grammar with some restrictions on grammar.
- An operator precedence grammar is a context-free grammar that has the property that no production has either an empty right hand side (null productions) or two adjacent non-terminals in its right-hand side.
- For each two terminal symbols a and b we define:
  1. $a = b$,
     if the production is in form say
     $S \rightarrow aSb \mid ab$ where 'a' appears immediate to left of 'b' or 'a' and 'b' are separated by one non-terminal.
  2. $a \lessdot b$,
     if say $S \rightarrow aA$
     $A \rightarrow bB$
- The non-terminal A is immediate to right of 'a' and it derives the production in which the first terminal is 'b'.
  3. $a > b$,
     if say $S \rightarrow Ab$
     $A \rightarrow Ba$
- The non-terminal A is immediate to the left of 'b' derives the string whose last symbol is 'a'.

Example 25: Consider the grammar:

$S \rightarrow S - S \mid S * S \mid (S) \mid id$

Solution: Consider the first production and find its derivations.

(a) $S \Rightarrow S - S$ (use derivation for RHS 'S')
$\Rightarrow S - S - S$ (by definition 2)

The relation is $- \lessdot -$

(b) $S \Rightarrow \underline{S} - S$
$\Rightarrow S - S - S$ (by definition 3)

The relation is $- \gtrdot -$.

Since, two precedence relations holds between operators. The above grammar is not operator precedence grammar.

The above grammar is made unambiguous and operator precedence as follows:

$$S \rightarrow S - B \mid B$$
$$B \rightarrow B * A \mid A$$
$$A \rightarrow (S) \mid id$$

This grammar is operator precedence grammar because there is only one precedence holds between the pairs of terminals.

From the above discussion, we are noticed that we can construct precedence relations using some methodology. This methodology is, we have two sets LEADING (A) and TRAILING (A) for each non-terminal $*$ A.

3.6.3 LEADING and TRAILING [Oct. 16]

- LEADING and TRAILING are functions specific to generating an operator-precedence parser, which is only applicable if we have an operator precedence grammar.

Definition:

1. (a) LEADING (A) = $\{a \mid A \overset{+}{\Rightarrow} \alpha a \beta$ where α may be $\in$ or any non-terminal$\}$
   (b) If a is in LEADING (B) and B is the first non-terminal derived from (i.e. $A \rightarrow B\alpha$) then a is in LEADING (A).
2. TRAILING (A) = $\{a \mid A \overset{+}{\Rightarrow} \alpha a \beta$, where β may be $\in$ or single non-terminal.

- From the above definition we observe that Leading is same as function First which we discussed in Section 3.2.5.1, but not exactly.

3.6.3.1 Algorithm for LEADING

- We consider input is context free grammar and finding the leading which is Boolean array L [A, a] in which we make the entry for it is true if a is in LEADING (A). We get output as the leading symbols for each non-terminal.

Procedure for Computation of Leading:

1. Make entry L [A, a] is false initially for each non-terminal 'A' and terminal 'a'.
2. For each production of the form,

```
A → aα  OR  A → Baα
{
   make L [A, a]:= true,
   push (A, a) onto stack;
}
3. do
     for each production of the form A → Bα
     }
       make L [A, a]:= true;
```

```
        push (A, a) onto stack;
      }
        pop  top pair (B, a) from stack;
    while stack is not empty.
```

Computation of LEADING:

- Leading is defined for every non-terminal.
- Terminals that can be the first terminal in a string derived from that non-terminal.
- LEADING(A)={ a| A $\overset{+}{\Rightarrow}$ αaβ }, where β is ε or any non-terminal, $\overset{+}{\Rightarrow}$ indicates derivation in one or more steps, A is a non-terminal.

3.6.3.2 Algorithm for TRAILING

- In this section we will study algorithm for TRAILING.
- **Input:** CFG
- **Output:** Set of trailing symbols in Boolean array T [A, a].
- **Procedure for Computation of Trailing:**

```
1. For each non-terminal A and terminal a {T (A, a):= false}.
2. For each production of the form A → aα, A → αaB
   {
    T (A, a):= true;
    push T (A, a) onto stack;
3. do
    {
      for each production of the form A → αB
      {
         T (A, a):= true
         push T (A, a) onto stack
      }
      pop top pair (B, a) from stack;
    }
      while stack not empty.
```

Computation of TRAILING:

- Trailing is defined for every non-terminal.
- Terminals that can be the last terminal in a string derived from that non-terminal.
- TRAILING(A)={ a| A $\overset{+}{\Rightarrow}$ αaβ },where β is ε or any non-terminal, $\overset{+}{\Rightarrow}$ indicates derivation in one or more steps, A is a non-terminal.

Example 26: Find leading and Trailing symbols for the following grammar:

$$S \rightarrow S - B \mid B$$
$$B \rightarrow B * A \mid A \qquad \textbf{(Grammar 3.4)}$$
$$A \rightarrow (S) \mid id$$

Solution:

```
LEADING (S) = {-, *, (, id}
LEADING (B) = {*, ( , id}
LEADING (A) = {(, id}
TRAILING (S) = {-, *, ), id}
TRAILING (B) = {*, ), id}
TRAILING (A) = {), id}
```

Example 27: Find leading and trailing symbols of the following grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Solution:

```
LEADING (E) = {+, *, (, id}
LEADING (T) = {*, (, id}
LEADING (F) = {(, id}
LEADING (E) = {+, *, id, )}
TRAILING (T) = {*, id, )}
TRAILING (F) = { ), id}
```

Example 28: Compute LEADING and TRAILING for the following grammar.

$$S \rightarrow (T) \mid a \mid \wedge$$
$$T \rightarrow T, S \mid s \qquad \textbf{(Oct. 16)}$$

Solution:

```
LEADING (S) = { (, a, ^}
LEADING (T) = { (, a, ^ }
TRAILING (S) = { ), (,, a, ^}
TRAILING (T) = { ), , (,, a, ^}
```

Example 29: Find out following grammar is operator precedence grammar or not.

$$S \rightarrow a \mid \wedge \mid (R)$$
$$T \rightarrow S, T \mid S$$
$$R \rightarrow T$$

Solution:

```
LEADING (S) = {a, ^, ( }
LEADING (T) = {,, a, ^ ( }
        (R) = {,, a, ^, ( )
TRAILING (S) = {a, ^, )}
```

(T) = {,, a, ^,)}

(R) = {, , a, ^,)}

Operator precedence relation table is as follow:

	a	^	(	)	,	$
a				·>	·>	·>
^		<·		·>	·>	·>
(	<·	<·	<·	=		
)				·>	·>	·>
,	<·	<·	<·			
$	<·	<·	<·			

In above table, there is no more than one precedence relation between two terminals. So above grammar is operator precedence grammar.

Example 30: Find out following grammar is operator precedence or not ?

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

Solution: (1) Consider the first production and find its derivation.

$$E \Rightarrow E + E \Rightarrow E + E + E$$

By definition 2: the non-terminal E is immediate right of '+' and it derive the production in which the first terminal is '+' (i.e. $E \rightarrow E + E$). $\therefore$ + <· +.

(2) Now by definition 3: The non-terminal E is immediate left of '+' and derive the string whose last terminal system is +.

$$E \Rightarrow E + E$$
$$\Rightarrow E + E + E$$

$\therefore$ + ·> +

Since two precedence relations hold between operators. The above grammar is not operator precedence grammar.

Example 31: Find out the following grammar is operator precedence grammar or not?

$$S \rightarrow aAb +$$
$$A \rightarrow (B \mid a$$
$$B \rightarrow A)$$

Solution: The derivation is

$$S \Rightarrow aAb +$$
$$\Rightarrow a (Bb +$$
$$\Rightarrow a (A) b +$$
$$\Rightarrow a (a) b +$$

The derived string is not proper expression. So we cannot find operator precedence relation. Therefore, grammar is not operator precedence grammar.

3.6.4 Operator Precedence Parsing Algorithm

- Operator precedence parser constructed for operator precedence grammar.
- **Input:** Operator precedence grammar.
- **Output:** Shift-reduce parsing using stack.
- **Procedure:** Let stack bottom marker is $ and input string be $a_1\ a_2 \ldots a_n$ $.

1. if only $ on the stack and $ in the input then accept and break.
2. let a be the TOS of stack and b be the current input symbol.
3. if a < b or a = b then shift b onto the stack else if a > b then

   { pop the stack }

   until the TOS terminal is related by < to

   the terminal most recently popped else error.

- Consider, the string **id– id** for the grammar.

$$S \rightarrow S - B \mid B$$
$$B \rightarrow B * A \mid A$$
$$A \rightarrow id$$

Stack	Input	Precedence Relation
$	id – id $	$ <· id
$ id	– id $	id ·> – (pop or reduce)
$ A	– id $	$ <· –
$ A –	id $	– <· id
$ A – id	$	id ·> $ (pop) or reduce
$ A – A	$	

- The handle indicated by the precedence relations is A – A. But there is no production which matches to it for reduction. So we ignore non-terminals.
- Shift-reduce precedence parsing only uses terminals. Keep only terminals on the stack and make reductions.

Disadvantages:

1. We cannot find the correct input or which non-terminal is present between terminals.
2. We can accept input which was not a sentence of the operator precedence grammar. Any sentence will be parsed.

3.6.5 Precedence Functions

- The table of precedence relations need not be stored by compilers which users operator-precedence parsers.
- Operator precedence parsers use precedence functions that map terminal symbols to integers and so the precedence relations between the symbols are implemented by numerical comparison.
- The parsing table can be encoded by two precedence functions f and g that map terminal symbols to integers. We select f and g such that:
  1. f(a) < g(b) whenever a is precedence to b.
  2. f(a) = g(b) whenever a and b having precedence.
  3. f(a) > g(b) whenever a takes precedence over b.
- The table is encoded by two precedence functions f and g, which map terminal symbols to integer.
- Consider two symbols a and b (terminals) such that:

f(a)	<	g(b)	for	a <· b
f(a)	=	g(b)	for	a = b
f(a)	>	g(b)	for	a ·> b

- To find precedence relation, f(a) and g(b) is compared.
- Method for finding precedence function table from operator precedence relation table is given below:
  1. The functions f_a and g_a are created for all terminal a of the grammar including $.
  2. Create a graph for all f_a and g_a as nodes and find the transition as follows:

     Take input as the precedence relation table.

     For each a and b (terminals from table).

     - a <· b then draw edge from the g_b to f_a.
     - a ·> b, then draw an edge from the group of f_a to g_b.
     - a = b no edge.
  3. If the graph constructed by step 2, has a cycle, then no precedence function exist. If there is no cycle, then the numerical value of f_a is the length of the longest path beginning at the group of f_a and the numeric value of g_a is the length of the longest path beginning at the group of g_a (number of nodes on the path).

Example 32: Consider the following precedence relation table:

	id	–	*	$
id		·>	·>	·>
–	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Draw the graph according to the above rules.

Solution:

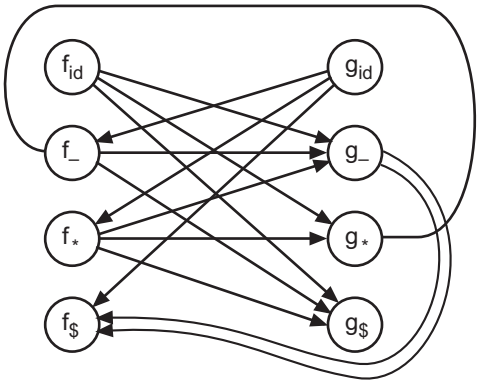


Fig. 3.23: Graph of Precedence Functions

Here, e.g. from fid the longest path we get is 4 (number of nodes present) which is $\text{fid} \rightarrow g* \rightarrow f\_ \rightarrow g_{\$}$.

Similarly, we can find the longest path for all modes and we get precedence function table as follows:

	Id	**–**	*****	**$**
f	4	2	4	0
g	5	1	3	0

Example 33: Consider the following grammar and find precedence functions.

E = E + T | T

T = T * F | F

F → (E) | id

Solution: Find first precedence relation table.

RHS / LHS	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	

Compute the graph of precedence function.

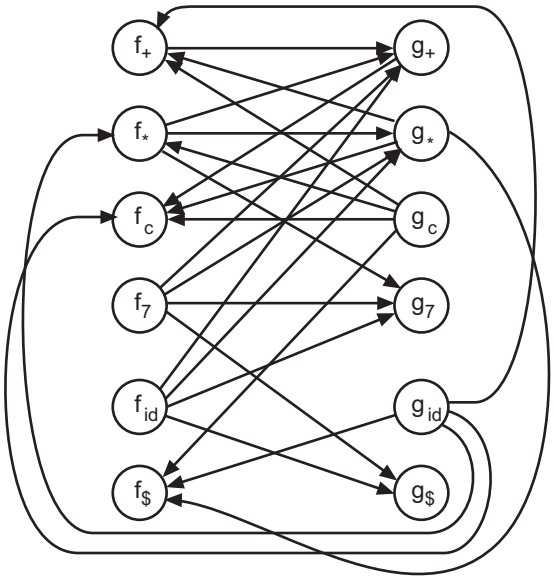


Fig. 3.24: The Graph of Precedence Function

We find the longest path for all nodes. We get precedence function table as follows:

	+	*	(	)	id	$
f	2	4	0	4	4	0
g	1	3	5	0	5	0

3.7 SHIFT REDUCE PARSING [April 18, Oct. 18]

- The most common bottom-up parsers are the shift-reduce parsers.
- The shift-reduce parsers examine the input tokens and either shift (push) them onto a stack or reduce elements at the top of the stack, replacing a right-hand side by a left-hand side.
- Shift-reduce parser can be implemented using stack. The stack consists of grammar symbols (non-terminals/terminals).

- Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.
  - **Shift Step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
  - **Reduce Step:** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.
- A shift-reduce parser uses a stack to hold the grammar symbols while awaiting reduction. During the operation of the parser, symbols from the input are shifted onto the stack.
- If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the non-terminal occurring on the LHS of the rule.
- This shift reduce process continues until the parser terminates, reporting either success or failure.
- The parser has input buffer, which consists of the string which is to be parsed. Initially stack contains $ at the bottom of the stack. Let input string ends with $.
- Initially stack is $ and input is $\omega$$.
- The parser shift zero or more input symbols onto the stack until a handle is on the top of the stack. If the handle is found at TOS then it reduces to LHS of the appropriate production rule. This shift or reduce process is continue until the stack has the start symbol and the input is empty.
- Here, the string is successfully parse and parser halts. While parsing the parser can detect the error and parsing is not successful. The shift-reduce parsing using stack is shown in Fig. 3.25.
- Now, consider the above Grammar 3.3.

Production Number:

1. $S \rightarrow S - B$
2. $S \rightarrow B$
3. $B \rightarrow B * A$
4. $B \rightarrow A$
5. $A \rightarrow id$

Now, we parse the string, id – id ∗ id using shift reduce parser,

Stack	Input	Action
$	id – id ∗ id $	shift
$ id	– id ∗ id $	reduce by A → id or (r5)
$A	– id ∗ id $	reduce by B → A or (r4) shift
$ B	– id ∗ id $	reduce by S → B or (r2)
$ S	– id ∗ id $	shift
$ S –	id ∗ id · $	shift
$ S – id	∗ id $	reduce by A → id or (r5)
$ S – A	* id $	reduce by B → A or (r4)
$ S – B	* id $	shift
$ S – B ∗	id $	shift
$ S – B ∗ id	$	reduce by A → id or (r5)
$ S – B ∗ A	$	reduce by B → B ∗ A or (r3)
$ S – B	$	reduce by S → S – B or (r1)
$ S	$	accept

Fig. 3.25: Shift-reduce Parser for the Input String id – id ∗ id

- This shift-reduce parser have the four possible actions as follows:
  1. **shift:** The next input symbol is shifted onto the top of the stack.
  2. **reduce:** From the rightmost to left, the handle at the top of the stack is found and it is reduced with non-terminal which matches with the handle.

     This reduce by r # n action is written at reduction, where n is production number.
  3. **accept:** Accept action means parser complete the parsing of input string successfully.
  4. **error:** If syntax error is occurred during parsing, then parser calls the error recovery routine.

3.7.1 Reduction, Handle and Handle Pruning [Oct. 18, April 19]

- Shift-reduce parser is a type of bottom-up parser. It generates the parse tree from leaves to the root.

Reduction:

- In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.
- To perform reduction, the parser must know the right end of the handle which is at the top of the stack.

- Then the left end of the handle within the stack is located and the non-terminal to replace the handle is decided.

Example 34: Perform the bottom-up parsing for the given string on the Grammar, i.e., shows the reduction for string abbcde on the following Grammar

$S \rightarrow a\ A\ B\ e$

$A \rightarrow A\ b\ c \mid b$

$B \rightarrow d$

Solution :

$S \Rightarrow^{rm} a\ A\ \underline{B}\ e$

$\Rightarrow^{rm} a\ \underline{A}\ d\ e$

$\Rightarrow^{rm} a\ \underline{A}\ b\ c\ d\ e$

$\Rightarrow^{rm} a\ b\ b\ c\ d\ e$

↑ Bottom-up Parsing (Reduction)

- It can reduce the string abbcde to the starting symbol S by applying the rightmost derivation in reverse at each step.

Handle: **[April 19]**

- Each replacement of the Right side of production by the left side in the process above is known as "Reduction" and each replacement is called "Handle."
- A handle of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.
- While reduction, we find the match. The sentential form (string) which matches the RHS of production rule while reduction, then that string is called "handle".
- Here, we are using right most derivation in reverse direction. i.e. we start with a string of terminals which we want to parse and derive the start symbol in reverse.

$S \overset{*}{\Rightarrow} \omega$ (derivation)

Example 35: Consider the Grammar,

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Perform Rightmost Derivation string $id_1 + id_2 * id_3$. Find Handles at each step.

Solution :

$S \Rightarrow^{rm} E + \underline{E}$

$\Rightarrow^{rm} E + E * \underline{E}$

$\Rightarrow^{rm} E + \underline{E} * id_3$

$\Rightarrow^{rm} \underline{E} + id_2 * id_3$

$\Rightarrow^{rm} id_1 + id_2 * id_3$

↑ Bottom-Up Parsing

Handles at each step:

Right Sentienal Form	HANDLE	Production Used
$id_1 + id_2 * id_3$	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		

Handle Pruning: **[Oct. 18]**

- We use this in reverse way that we find the handle and handle is reduced with LHS. If we reach to S then parsing is successful. This process is called handle pruning.
- The Process of discovering a handle and reducing it to the appropriate left hand side is called handle pruning. Handle pruning forms the basis for a bottom-up parsing.
- A rightmost derivation in reverse can be obtained by handle pruning.
- For example, consider the Grammar 3.3. To reduce the string id – id ∗ id we get the sequence of reduction is as shown in Fig. 3.26.

Right Sentential Form	Handle
id – id ∗ id	id
A – id ∗ id	id
A – A ∗ id	id
A – A ∗ A	A
B – A ∗ A	B
S – A ∗ A	A
S – B ∗ A	B ∗ A
S – B	S – B
S	

Fig. 3.26: Reductions using Bottom-up Parsing

- Bottom-up parsing is also called as shift-reduce parsing. It uses stack and action on input. This action can be either shift or reduce.

Example 36: Consider the following Grammar,

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

Check whether input string "ccdd" is accepted or not accepted using Shift-Reduce parsing.

Solution:

$S \Rightarrow^{rm} C\underline{C}$

$\Rightarrow^{rm} \underline{C}d$

$\Rightarrow^{rm} c\ \underline{C}\ d$

$\Rightarrow^{rm} cc\ \underline{C}\ d$

$\Rightarrow^{rm} cc\ d\ d$

Bottom-up Parsing

Stack	Input String	Action
$	ccdd$	Shift
$ c	cdd$	Shift
$ cc	dd$	Shift
$ ccd	d$	Reduce by C → id
$ ccC	d$	Reduce by C → cC
$cC	d$	Reduce by C → cC
$C	d$	Shift
$Cd	$	Reduce by C → d
$CC	$	Reduce by S → CC
$S	$	Accept

3.7.2 Stack Implementation of Shift-Reduce Parser

- Any string of Grammar can be parsed by using stack implementation, as in shift-reduce parsing.
- Shift-reduce parser is a type of bottom-up parser. It uses a stack to hold grammar symbols.
- A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack. When a handle occurs on the top of the stack, it implements reduction.
- The shift-reduce parser consists of input buffer, stack and parse table. **Input buffer** consists of strings, with each cell containing only one input symbol.
- **Stack** contains the grammar symbols, that are inserted using shift operation and they are reduced using reduce operation after obtaining handle from the collection of buffer symbols.
- **Parse table** consists of two parts, goto and action, which are constructed using terminal, non-terminals and complier items.
- There are the various steps of shift-reduce parsing which are as follows:
  1. It uses a stack and an input buffer.
  2. Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.

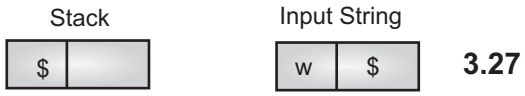


Fig. 3.27

3. **Shift:** Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.
4. **Reduce:** Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.
5. **Accept:** Step 3 and Step 4 will be repeated until it has identified an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains $.

Fig. 3.28

6. **Error:** Signal discovery of a syntax error that has appeared and calls an error recovery routine.

- For example, consider the grammar

  `S → aAcBe`

  `A → Ab|b`

  `B → d`

  and the string is abbcde.
- It can reduce this string to S. It can scan string abbcde looking for the substring that matches the right side of some production. The substrings b and d qualify.
- Let us select the left-most b and replace it with A, the left side of the production A →b, and obtain the string aAbcde.
- It can identify that Ab, b, and d each connect the right side of some production. Suppose this time it can select to restore the substring Ab by A, the left side of the production A →Ab and it can achieve aAcde.
- Thus replacing d by B, the left side of the production B →d, and can achieve aAcBe. It can replace this string by S.
- Each replacement of the right-side of a production by the left side in the process above is known as reduction.

Drawbacks of Shift Reduce Parsing:

1. **Shift|Reduce Conflict:** Sometimes, the SR parser cannot determine whether to shift or to reduce.
2. **Reduce|Reduce Conflict:** Sometimes, the Parser cannot determine which of Production should be used for reduction.

Example 37: To stack implementation of shift-reduce parsing is done, consider the grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$ and input string as $id_1 + id_2 \rightarrow id_3\ id_1 + id_2 * id_3$.

Stack	Input String	Action
$	$id_1 + id_2 * id_3$$	Shift
$ id_1	$+id_2 * id_3$$	Reduce by $E \rightarrow id$
$ E	$+id_2 * id_3$$	Shift
$ E +	$id_2 * id_3$$	Shift
$ E + id_2	$* id_3$$	Reduce by $E \rightarrow id$
$E + E	$* id_3$$	Shift
$E + E *	$id_3$$	Shift
$E + E * id_3	$	Reduce by $E \rightarrow id$
$E + E * E	$	Reduce by $E \rightarrow E * E$
$E + E	$	Reduce by $E \rightarrow E + E$
$E	$	Accept

Let us illustrate the above stack implementation.

$\rightarrow$ Let the grammar be,

$S \rightarrow AA$

$A \rightarrow \alpha A$

$A \rightarrow b$

Let the input string 'ω' be abab$

ω = abab$

Stack	Input String	Action
$	abab$	Shift
$a	bab$	Shift
$ab	ab$	Reduce ($A \rightarrow b$)
$aA	ab$	Reduce ($A \rightarrow aA$)
$A	ab$	Shift
$Aa	b$	Shift
$Aab	$	Reduce ($A \rightarrow b$)
$AaA	$	Reduce ($A \rightarrow aA$)
$AA	$	Reduce ($S \rightarrow AA$)
$S	$	Accept

Conflicts during Shift-reduce Parsing:

- For some CFG (context free grammars) shift-reduce parsing is not used. The parser cannot decide whether action is shift or reduce. This is called **s**hift-reduce conflicts.
- Sometimes parser cannot decide which reduction have to be made among many reductions. This is called reduce-reduce conflicts.
- If the grammar is having conflict while parsing, then such a grammar is not LR grammar. This conflict example we will discuss in later sections.
- An ambiguous grammar can never be LR. Grammars used in compiling process usually fall in the LR (1) class, where one is lookahead symbol.

Example 38: Consider grammar of if-then-else statement.

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

Suppose we have the current stack configuration in shift-reduce parser is **stack.**

stack	input
… if expr then stmt	else … $

Here, we cannot tell if expr then stmt is a handle. The parser either reduce "if expr then stmt" into stmt or shift "else" to the stack depending upon what follows the else on the input.

Hence, in the above grammar shift-reduce conflict occurs during parsing. The above grammar is ambiguous. It is possible to resolve the conflict and grammar can be made unambiguous.

Example 39: The another conflict is reduce-reduce conflict. Consider an example in which lexical analyzer returns token name **id** for all names, regardless of their type. The statement a (i, j) appear as a token stream id (id, id) to the parser.

Some productions are as follows:

1. `stmt → id (param_list)`
2. `param_list → param_list, param`
3. `param → id`
4. `param_list → param`
5. `expr → id, expr`
6. `expr → id`

If the current stack configuration is,

stack	input
… id (id	, id) …

Here, id is on top of the stack and id must be reduced, but by which production? Either by production (3) or production (6). Here, reduce-reduce conflict occurs. The solution is, we can change the grammar production and parse unambiguous grammar.

3.8 LR PARSERS [April 16, 18, Oct. 18]

- LR parsing was invented by Donald Knuth in 1965. LR parsing is bottom-up technique LR means the input scans from left to right and generate rightmost derivation in reverse.
- Let K is the number of input symbols of lookahead that are used in making parsing decisions. We use LR (K) parsing technique. By default K = 1.
- The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.
- The LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.
- An LR parser (Left-to-right, Rightmost derivation in reverse) reads input text from left to right without backing up (this is true for most parsers), and produces a rightmost derivation in reverse: it does a bottom-up parse

Advantages of LR Parsing:

1. This method is non-backtracking shift-reduce parsing method.
2. All class of grammars can be parsing using LR method. It is superset of predictive parser.
3. Syntax errors are detected immediately while scanning the input.
4. Left-recursion is not the problem in LR parsing. Grammar may be left recursive.
5. LR parser is almost used in compilers of all programming languages.
6. The class of grammars that can be parsed using LR methods is a superset of the class of grammars that can be parsed with LL method or predictive parser.

- The techniques for constructing LR parsing table are:
  1. **SLR (Simple LR Parser):** It works on smallest class of grammar. SLR constructs parsing tables which helps to perform parsing of input strings. SLR Parsing can be done if context-free Grammar will be given.
  2. **Canonical Look Ahead LR Parser (CLR)**: It defines canonical lookahead. CLR parsing uses the canonical collection of LR(1) items to construct the CLR (1) parsing table. CLR(1) parsing table makes more number of states as compared to the SLR(1) parsing. In the CLR(1), it can locate the reduce node only in the lookahead symbols.
  3. **Lookahead LR (LALR):** It works on intermediate size of grammar. LALR is intermediate in power between SLR and CLR parser. It is the compaction of CLR Parser, and hence tables obtained in this will be smaller than CLR Parsing Table.

3.8.1 Model for LR Parser

- Fig. 3.29 shows model of LR Parser. LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions namely, ACTION and GOTO.
  1. The **ACTION** function takes as arguments a state i and a terminal a (or $, the input end marker).
  2. The **GOTO** function takes a state and grammar symbol as arguments and produces a state.
- The driver program is same for all LR parsers. Only the parsing table changes from one parser to another.
- The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser shifts a state. Each state summarizes the information contained in the stack.
- The stack stores the chronology of grammar with a $ at the bottom of the stack. The string which has to be parsed stays in the input buffer, which is used to indicate the end of input, followed by a $ Symbol.
- The stack holds a sequence of states, so, s1, · ·· , Sm, where Sm is on the top.

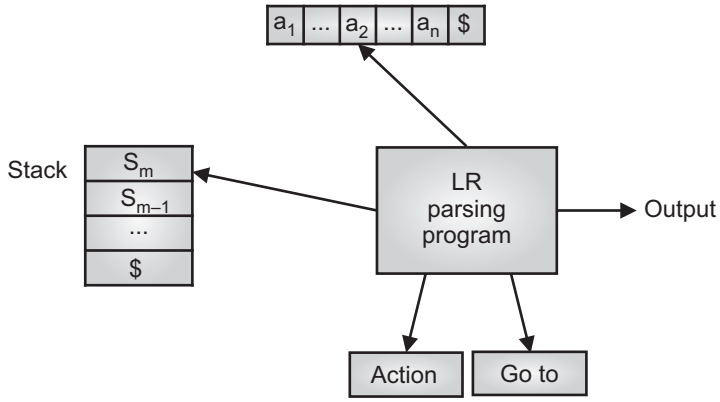


Fig. 3.29: Model of LR Parser

3.8.2 Types of LR Parser

[April 16, 17, 18, 19, Oct. 16, 17]

- Fig. 3.30 shows types of LR parser.

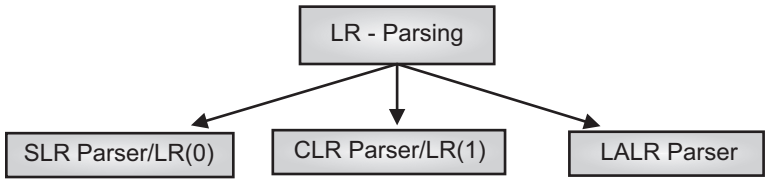


Fig. 3.30: Types of LR Parser

- Let us see different parsers in detail.

SLR Parser: [April 16, 17, 18, 19, Oct. 16, 17, 18]

- SLR represents "Simple LR Parser". It is very easy and cost-effective to execute.
- The SLR parsing action and goto function from the deterministic finite automaton that recognizes viable prefixes.
- It will not make specifically defined parsing action tables for all grammars but does succeed on several grammars for programming languages.
- Given a grammar G. It augment G to make G', and from G' it can construct C, the canonical collection of a set of items for G'.
- It can construct ACTION the parsing action function, and GOTO, the goto function, from C using the following simple LR Parsing table construction technique. It needed us to understand FOLLOW (A) for each non-terminal A of a grammar.
- The SLR(1) is a grammar having an SLR parsing table is said to be SLR(1).

Working of SLR Parser:

- SLR Parsing can be done if context-free Grammar will be given. In LR (0), 0 means there is no Look Ahead symbol.
- Fig. 3.31 shows working of SLR parser.

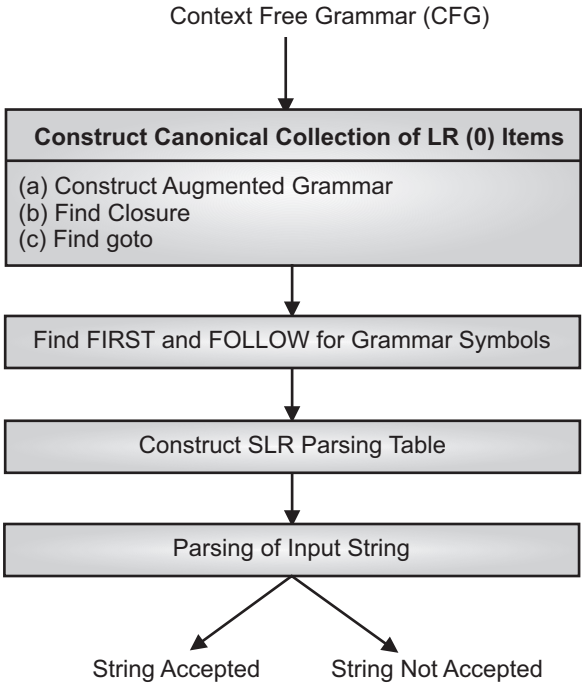


Fig. 3.31

- The LR(0) item for Grammar G consists of a production in which symbol dot (.) is inserted at some position in R.H.S of production.

- For example, for the production S →ABC, the generated LR (0) items will be,

  S → · ABC

  S → A · BC

  S → AB · C

  S → ABC ·

  Production S → ε generates only one item, i.e., S →·. Canonical LR(0) collection helps to construct LR parser called Simple LR (SLR) parser.
- To create Canonical LR(0) collection for Grammar, following things are required:
  - Augmented Grammar.
  - Closure Function.
  - goto Function.

CLR Parser:

- CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR(1) items to construct the CLR(1) parsing table.
- CLR(1) parsing table make more number of states as compared to the SLR(1) parsing. In the CLR(1), it can locate the reduce node only in the lookahead symbols.
- Fig. 3.32 shows working of CLR parser.

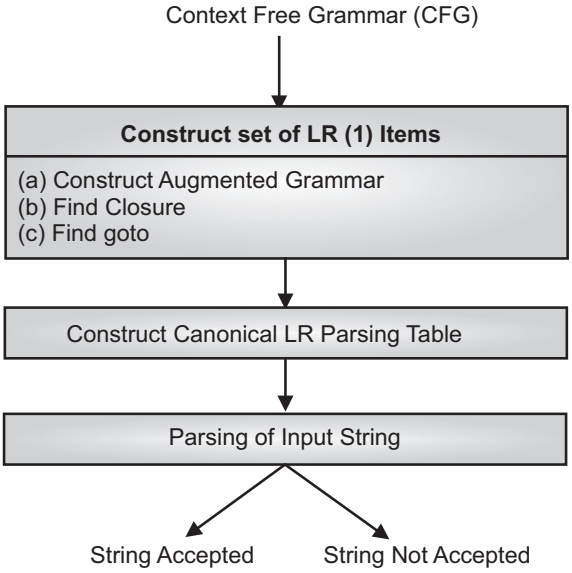


Fig. 3.32

- Construction of LR(1) collection of items for Grammar, it requires following three things:
  - Augmented Grammar.
  - Closure Function.
  - goto Function.

LALR Parser: [April 16, 17, Oct. 16, 17]

- LALR Parser is Look Ahead LR Parser. It is intermediate in power between SLR and CLR parser.
- It is the compaction of CLR parser, and hence tables obtained in this will be smaller than CLR parsing table.
- Fig. 3.33 shows working of LALR parser.

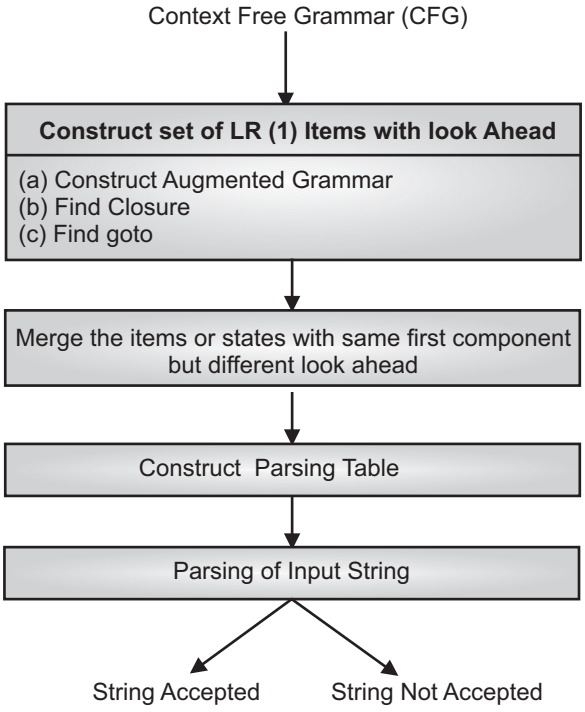


Fig. 3.33

- For constructing the LALR(1) parsing table, the canonical collection of LR(1) items is used.
- In the LALR(1) parsing, the LR(1) items with the equal productions but have several look ahead are grouped to form an individual set of items.
- It is frequently the similar as CLR(1) parsing except for the one difference that is the parsing table.
- The overall structure of all these LR Parsers is the same. There are some common factors such as size, class of context-free grammar, which they support, and cost in terms of time and space in which they differ.
- Let us see the comparison between SLR, CLR, and LALR Parser.

Sr. No.	SLR Parser	LALR Parser	CLR Parser
1.	SLR parser is very easy and cheap to implement.	LALR parser is also easy and cheap to implement.	CLR parser is expensive and difficult to implement.
2.	SLR parser is the smallest in size.	LALR and SLR have the same size.	CLR parser is the largest.
3.	Error detection is not immediate in SLR parser.	Error detection is not immediate in LALR parser.	Error detection can be done immediately in CLR parser.
4.	SLR parser fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
5.	SLR parser requires less time and space complexity.	LALR parser requires more time and space complexity.	CLR parser also requires more time and space complexity.

3.8.3 Constructing SLR Parsing Table

- The LR parsing tables are a two-dimensional array in which each entry represents an ACTION or GOTO entry.
- A programming language grammar having a large number of productions has a large number of states or items, i.e.,I_0, I_1 ... ... I_n.
- A grammar for which an SLR parser can be constructed is said to be an SLR grammar. SLR parsers are the parsers which use SLR parsing tables.
- For constructing SLR parsing table, we start with LR(0) items set i.e. with K = 0, which we call the canonical LR(0) collection, it provides the basis for constructing SLR parsers.
- An LR(0) item of grammar G is a production of G with a dot at some position of the right side. Thus, the production $X \rightarrow ABC$ yields the four items:

$$X \rightarrow \cdot ABC$$
$$X \rightarrow A \cdot BC$$
$$X \rightarrow AB \cdot C$$
$$X \rightarrow ABC \cdot$$

- The production $A \rightarrow \in$ generates only one item, $A \rightarrow \cdot$. An item can be represented by a pair of integers, the first giving the number of the production and the second the position of the dot.

- To construct the canonical LR(0) collection for a grammar, we need to define an augmented grammar for G and two functions CLOSURE and GOTO.

Augmented Grammar:

- If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and production $S' \rightarrow S'$.
- The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, the acceptance occurs when and only when the parser is about to reduce by $S' \rightarrow S$.

The Closure Operation:

- If I is a set of items for a grammar G, then the set of items CLOSURE (I) is constructed from I by the rules:
  1. Every item in I is in CLOSURE (I)
  2. If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE (I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to I, if it is not already there, we apply this rule until no more new items can be added to closure (I).
- Example, consider the augmented expression grammar of Grammar 3.3.

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow S - B \mid B \\ B &\rightarrow B * A \mid A \\ A &\rightarrow (S) \mid id \end{aligned}$$

(Grammar 3.5)

If I is in set of one item {[$S' \rightarrow \cdot S$]}, then closure (I) contains the items:

$$\begin{aligned} S' &\rightarrow \cdot S \\ S &\rightarrow \cdot S - B \\ S &\rightarrow \cdot B \\ B &\rightarrow \cdot B * A \\ B &\rightarrow \cdot A \\ A &\rightarrow \cdot (S) \\ A &\rightarrow \cdot id \end{aligned}$$

- We want to find the closure of $S' \rightarrow \cdot S$. Since, after dot we have non-terminal immediately, so we add all the productions of that non-terminal. Hence we have added S productions.
- In S-productions again we get dot following non-terminal B. Hence, add B-productions and so on. We get the items in closure(I).

The GOTO Operator:

- If I is the set of items and X is grammar symbol. We define GOTO (I, X) as $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X\beta]$ is in I.

- For example: If the set of items I is {[S' → S·] [S → S · – B] then GOTO (I, –) is the following rule.

$$S \rightarrow S - \cdot B \qquad \text{(dot is shifted by 1 position)}$$

- Now, after dot immediate non-terminal B is present, so we again use closure procedure.

$$S \rightarrow S - \cdot B$$
$$B \rightarrow \cdot B * A$$
$$B \rightarrow \cdot A$$
$$A \rightarrow \cdot (S)$$
$$A \rightarrow \cdot id$$

Construction of LR (0) Items:

Input: Augmented grammar G'.

Output: LR (0) items [canonical collection of LR (0)].

Procedure:

1. A = {closure ([S' → · S])} where, S' is start symbol.
2. For such item I in A and each grammar symbol X such that goto (I, X) is not in C, add goto (I, X) to C until no more sets of items can be added to C.

Example 40: Consider the grammar:

$$S \rightarrow S - B \mid B$$
$$B \rightarrow B * A \mid A \qquad \textbf{(Grammar 3.6)}$$
$$A \rightarrow (S) \mid id$$

Find the canonical sets of LR (0) items.

Solution: Make the grammar augmented first.

$$S' \rightarrow S$$
$$S \rightarrow S - B$$
$$S \rightarrow B$$
$$B \rightarrow B * A$$
$$B \rightarrow A$$
$$A \rightarrow (S)$$
$$A \rightarrow id$$

Now, find LR (0) sets of items:

$$I_{0:}\ S' \rightarrow \cdot S$$
$$S \rightarrow \cdot S - B$$
$$S \rightarrow \cdot B$$
$$B \rightarrow \cdot B * A$$

B → · A
A → · (S)
A → · id

Goto (I_0, S)

I_1: S' → S ·
S → S · – B

Goto (I_0, B)

I_2: S → B ·
B → B · * A

Goto (I_0, A)

I_3: B → A ·

Goto (I_0, C)

I_4: A → (· S)
S → · S – B
S → · B
B → · B * A
B → · A
A → · (S)
A → · id

Goto (I_0, id)

I_5: A → id ·

Goto (I_1, –)

I_6: S → S – · B
B → · B * A
B → · A
A → · (S)
A → · id

Goto (I_2, *)

I_7: B → B * · A
A → · (S)
A → · id

Goto (I_4, S)

I_8: A → (S ·)
S → S · – B

Goto (I_4, B) = I_2
Goto (I_4, A) = I_3
Goto (I_4, C) = I_4 } repeated
Goto (I_4, id) = I_5

Goto (I_6, B)

$I_{9:}$ S → S – B ·
B → B · * A

Goto (I_6, A) = I_3
Goto (I_6, C) = I_4 } repeated
Goto (I_6, id) = I_5

Goto (I_7, A)

$I_{10:}$ B → B * A ·

Goto (I_8,))

$I_{11:}$ A → (S) ·

No more items can be added.

The I_0 to I_{11} are the canonical LR (0) collections we can draw DFA for above LR (0) items, is shown in Fig. 3.34.

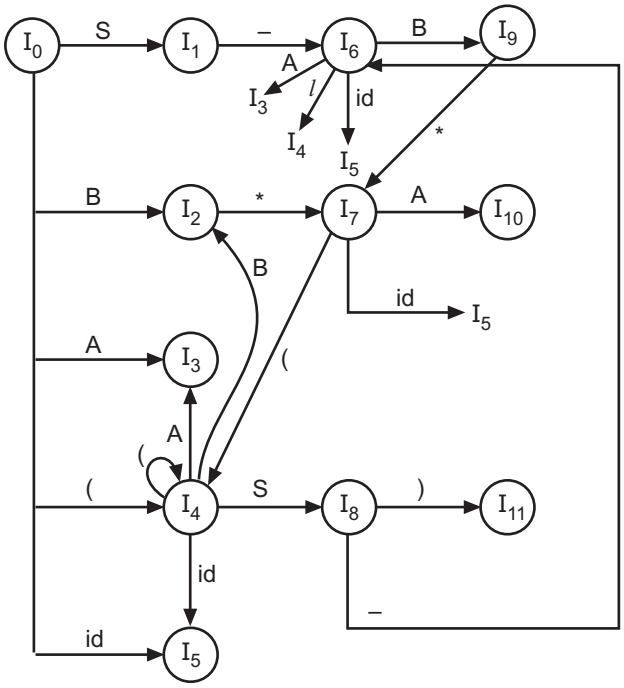


Fig. 3.34: DFA

SLR Parsing Tables:

- After constructing LR(0) items we will construct SLR parsing table. Table consists of action and goto part for which we can use DFA.
- We required FOLLOW set of all non-terminals of the Grammar 3.6.

Algorithm to Construct SLR Parsing Table:

1. Let $A = \{I_0, I_1, \ldots I_n\}$ be the collection of set of LR (0) items of grammar.
2. The states of parsing table are from 0 to n, for each I_0 to I_n.

   The actions of table are as follows:

   (a) If $[A \to \alpha \cdot a\ \beta]$ is in I_i and goto $(I_i, a) = I_j$ then set action [i, a] to "shift j". Here, 'a' must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set action [i, a] to "reduce $A \to \alpha$" for all a in FOLLOW (A); here A may not be S'.

   (c) If $[S' \to S \cdot]$ is in I_i, then set action [i, $] to "accept".

   (d) The goto transitions for state i are constructed for all non-terminals A using the rule: If goto $(I_i, A) = I_j$, then goto [i, A] = j.

   (e) All entries which are not defined are made "error".

Note: The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

If we get any conflict actions by the above rules, then the grammar is not SLR(1).

Now, construct the SLR table from Grammar 3.6.

FOLLOW (S) = {$, –,)}
FOLLOW (B) = {*, –,), $}
FOLLOW (A) = {*, –,), $}

1. Consider I_0 item:

$$
\begin{aligned}
S' &\to \cdot S \\
S &\to \cdot S - B \\
S &\to \cdot B \\
B &\to \cdot B * A \\
B &\to \cdot A \\
A &\to \cdot (S) \\
A &\to \cdot id
\end{aligned}
$$

First find terminal symbol immediately after dot and put shift action in table for it.

$A \to \cdot (S)$ action is [0, (] = shift 4 or S4

and $A \to \cdot id$ action is [0, id] = shift 5 or S5.

Now, find dot at last which is not present.

2. Consider I_1 item:

$$S' \rightarrow S \cdot$$
$$S \rightarrow S \cdot - B$$

rule (c) is applicable for $S' \rightarrow S \cdot$.

We get,

action [1, $] = accept

and for $S \rightarrow S \cdot - B$ rule (a) is applicable and we get,

action [1, –] = shift 6 or S6

3. Consider I_2 item:

$$S \rightarrow B \cdot$$

find FOLLOW (S), we get,

action [2, $] = action [2, –] = action [2,)] = reduce by $S \rightarrow B$ i.e. r_2

$$B \rightarrow B \cdot * A$$

action [2, *] = shift 7 i.e. S7.

- Similarly, we find actions for all set of items of LR (0) and then parsing table entries are made, which are shown in Fig. 3.35.

State	Action						Goto		
	id	–	*	(	)	$	S	B	A
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 3.35: SLR Parsing Table for Expression Grammar

So the grammar is SLR(1).

Remember:

"Every SLR(1) grammar is unambiguous", always true but "every unambiguous grammar is SLR(1)", not always true. Some unambiguous grammars are not SLR(1).

Example 41: Check whether following grammar is SLR (1) or not.

$$S \rightarrow L = R$$
$$S \rightarrow R$$
$$L \rightarrow *R$$
$$L \rightarrow id$$
$$R \rightarrow L$$

(Grammar 3.7)

Solution: First we find the set of LR (0) items as follows:

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot *R$
$L \rightarrow \cdot id$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot *R$
$L \rightarrow \cdot id$

I_5: $L \rightarrow id \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot *R$
$L \rightarrow \cdot id$

I_7: $L \rightarrow *R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R$

Fig. 3.36: Canonical LR(0) for Grammar 3.7

FOLLOW (S) = {$} FIRST (S) = {*, id}
FOLLOW (L) = {=, $} FIRST (L) = {*, id}
FOLLOW (R) = {$, =} FIRST (R) = {*, id}

The SLR parsing table is shown in Fig. 3.37.

States	Action				Goto		
	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				accept			
2	S6/r5			r5			
3				r2			
4		S4	S5			8	7
5	r4			r4			
6							9
7	r3			r3			
8	r5			r4			
9				r1			

Fig. 3.37: The SLR Parsing Table

Since there is shift-reduce conflict at entry action [2, =] in the table. The above grammar is not SLR(1).

Example 42: Construct SLR parsing table for following grammar.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Solution: $E' \rightarrow E$ (Augumented grammar)

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

The LR (0) items are as follows:

I_0: $E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$ } E-productions are added

$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$ } T-productions are added

$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$ } F-productions are added

Goto (I_0, E)

I_1: $E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T$

Goto (I_0, T)

I_2: $E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

Goto (I_0, F)

I_3: $T \rightarrow F \cdot$

Goto (I_0, ( )

I_4: $F \rightarrow (\cdot E)$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

Goto (I_0, id)

I_5: $F \rightarrow id \cdot$

Goto (I_1, +)

I_6: $E \rightarrow E + \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

Goto (I_2, *)

I_7: $T \rightarrow T * \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

Goto (I_4, E)

I_8: $F \rightarrow (E \cdot)$

$E \rightarrow E \cdot + T$

Goto (I_4, T) = I_2

Goto (I_4, F) = I_3

Goto (I_4, ( ) = I_4

Goto (I_4, id) = I_5

} repeated

Now, Goto(I_6, T) = I_9

I_9: E → E + T ·

T → T · * F

Goto (I_6, F) = I_3

Goto (I_6, ( ) = I_4

Goto (I_6, id) = I_5

} repeated

Now, Goto (I_7, F) = I_{10}

I_{10}:T → T * F ·

Goto (I_7, ( ) = I_4

Goto (I_7, id) = I_5

Now, Goto (I_8,)) = I_{11}

I_{11}:F → (E) ·

Goto (I_8, +) = I_6

Goto (I_9, *) = I_7

Now, compute FOLLOW of non-terminals.

FOLLOW (S) = {$, + ,)}

FOLLOW (T) = {$, +, *,)}

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

Now construct the parsing table,

State	Action						Goto		
	id	–	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 3.38: SLR Parsing Table for Expression Grammar

Consider the string id * id + id and show the moves of LR parsers to parse the string using stack.

Stack Symbols	Input	Action
$	id * id + id $	shift
id	* id + id $	reduce by F → id
F	* id + id $	reduce by T → F
T	* id + id $	shift
T *	id + id $	shift
T * id	+ id $	reduce by F → id
T * F	+ id $	reduce by T → T * F
T	+ id $	reduce by E → T
E	+ id $	shift
E +	id $	shift
E + id	$	reduce by F → id
E + F	$	reduce by T → F
E + T	$	reduce by E → E + T
E	$	accept

Fig. 3.39: Moves of LR Parser for id * id + id

Example 43: Check following grammar is SLR(1) grammar or not?

$$S \rightarrow A \mid B$$
$$A \rightarrow aA \mid b$$
$$B \rightarrow dB \mid b$$

Solution: Augmented grammar is,

$$S' \rightarrow S$$
1. $S \rightarrow A$
2. $S \rightarrow B$
3. $A \rightarrow aA$
4. $A \rightarrow b$
5. $B \rightarrow dB$
6. $B \rightarrow b$

Now, find LR (0) set of items:

I_0: $S' \rightarrow \cdot S$

$S \rightarrow \cdot A$

$S \rightarrow \cdot B$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

$B \rightarrow \cdot dB$

$B \rightarrow \cdot b$

Goto (I_0, S)

I_1: $S' \rightarrow S \cdot$

Goto (I_0, A)

I_2: $S \rightarrow A$

Goto (I_0, B)

I_3: $S \rightarrow B$

Goto (I_0, a)

I_4: $A \rightarrow a \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

Goto (I_0, b)

I_5: $A \rightarrow b$

$B \rightarrow b$

Goto (I_0, d)

I_6: $B \rightarrow d \cdot B$

$B \rightarrow \cdot dB$

$B \rightarrow \cdot b$

Goto (I_4, A)

I_7: $A \rightarrow aA \cdot$

Goto (I_4, a) = I_4

Goto (I_4, b)

I_8: $A \rightarrow b \cdot$

Goto (I_6, B)

I_9: $B \rightarrow dB \cdot$

Goto (I_6, d) = I_6

Goto (I_6, b)

I_{10}: $B \rightarrow b \cdot$

No more items can be added.

FOLLOW (S) = FOLLOW (A) = FOLLOW (B) = {$}

Now, construct SLR (1) table,

State	Action				Goto		
	a	b	d	$	S	A	B
0	S4	S8	S6		1	2	3
1				accept			
2				r1			
3				r2			
4	S4	S8				7	
5				r3/r5			
6		S10	S6				9
7				r3			
8				r3			
9				r4			
10				r5			

The above grammar is not SLR (1) because there are multiple entries in cell action [5, $]. This is called reduce-reduce conflict.

Example 44: Check following grammar is SLR (1) or not.

$$S \rightarrow bAB \mid aA$$
$$A \rightarrow Ab \mid b$$
$$b \rightarrow aB \mid a$$

Solution: Augmented grammar is,

0. $S' \rightarrow S$
1. $S \rightarrow bAB$
2. $S \rightarrow aA$
3. $A \rightarrow Ab$
4. $A \rightarrow b$
5. $B \rightarrow aB$
6. $B \rightarrow a$

Now, find LR (0) sets of items:

I_0 : Closure ($S' \rightarrow \cdot S$)

$\{S' \rightarrow \cdot S$
$S \rightarrow \cdot bAB$
$S \rightarrow \cdot aA\}$

Goto (I_0, S)

I_1: $S' \rightarrow S \cdot$

Goto (I_0, b)

I_2: $S \rightarrow b \cdot AB$

A → · Ab

A → · b

Goto (I_0, a)

I_3: S → a · A

A → · Ab

A → · b

I_4: Goto (I_2, A)

I_4: S → bA · B

A → A · b

B → · aB

B → · a

Goto (I_2, b)

I_5: A → b ·

Goto (I_3, A)

I_6: S → aA ·

A → A · b

Goto (I_3, b) = I_5 (repeated)

Goto (I_4, B)

I_7: S → bAB ·

Goto (I_4, b)

I_9: A → Ab ·

Goto (I_4, a)

I_8: B → a · B

B → a ·

B → · aB

B → · a

Goto (I_6, b) ≅ I_8

Goto (I_9, B)

I_{10}: B → aB ·

Goto (I_9, a) ≅ ± a

No more items can be added.

	FIRST	FOLLOW
S'	{b, a}	{$}
S	{b, a}	{$}
A	{b}	{a, b, $}
B	{a}	{$}

Now construct SLR parse table.

State	Action			Goto		
	a	b	$	S	A	B
0	S3	S2		1		
1			accept			
2		S5			4	
3		S5			6	
4	S9	S8				7
5	r4	r4	r4			
6		S8	r2			
7			r1			
8	r3		r3			10
9	S9	r3	r6			
10			r5			

Above grammar is SLR (1) grammar because no multiple entries in the table.

Example 45: Check following grammar is SLR or not.

$S \rightarrow A \mid D$

$A \rightarrow B; C$

$B \rightarrow bd \mid B; d$

$C \rightarrow ae \mid a; C$

Solution: Augmented grammar is,

$S' \rightarrow S$

1. $S \rightarrow A$
2. $A \rightarrow B; C$
3. $B \rightarrow bd$
4. $B \rightarrow B; d$
5. $C \rightarrow ae$
6. $C \rightarrow a; C$

Here, D is useless

$\therefore$ Remove production $S \rightarrow D$.

Now, find set of LR (0) items:

I_0: $S' \rightarrow \cdot S$

$S \rightarrow \cdot A$

$A \rightarrow \cdot B; C$

$B \rightarrow \cdot bd$

$B \rightarrow \cdot B; d$

Goto (I_0, S)

I_1: $S' \rightarrow S \cdot$

Goto (I_0, A)

I_2: S → A ·

Goto (I_0, B)

I_3: A → B · ; C

B → B · ; d

Goto (I_0, b)

I_4: B → b · d

Goto (I_3, ;)

I_5: A → B ; · C

C → · ae

C → · a ; C

B → B ; · d

Goto (I_4, d)

I_6: B → bd ·

Goto (I_5, C)

I_7: A → B ; C ·

Goto (I_5, a) = I_8

I_8: C → a · e

C → a · ; C

Goto (I_8, d)

I_9: B → B ; d ·

Goto (I_8, e)

I_{10}: C → ae ·

Goto (I_8, ;)

I_{11}: C → a ; · C

C → · ae

Goto (I_{11}, C)

I_{12}: C → a ; C ·

Goto (I_{11}, a)

I_{13}: C → a · e

Goto (I_{13}, e) = I_{10}

No more items can be added.

Now, compute FOLLOW.

FOLLOW (S) = FOLLOW (A) = FOLLOW (C) = {$}

FOLLOW (B) = {;}

Now, construct SLR table.

State	Action						Goto			
	a	b	d	e	;	$	S	A	B	C
0		S4					1	2	3	
1						accept				
2						r1				
3					S5					
4			S6							
5	S8		S9							7
6					r3					
7						r2				
8				S4	S11					
9					r4					
10						r5				
11	S13									
12						r6				
13				S10						

The above grammar is SLR (1).

Parse the string bd; d; a; ae using LR parsing method by stack.

Stack	Input	Action
$	bd ; d ; a ; ae$	shift (S4)
$ b	d ; d ; a ; ae$	shift (S6)
$ bd	; d ; a ; ae$	(r3) reduce by B→ bd
$ B	; d ; a; ae$	shift (S5)
$ B ;	d ; a ; ae$	shift (S9)
$ B ; d	; a ; ae$	reduce by B → B · d (r4)
$ B	; a ; ae$	shift (S5)
$ B ;	a ; ae$	shift (S8)
$ B ; a	; ae$	shift(S11)
$ B ; a ;	ae$	shift (S13)
$ B ; a ; a	e$	shift (S10)
$ B ; a ; ae	$	reduce by C → ae
$ B ; a ; C	$	reduce by C → a; C
$ B ; C	$	reduce by A → B; C
$ A	$	reduce by S → A
$ S	$	accept

Example 46: Check whether the following grammar is SLR(1) or not? **[April 16]**

$S \rightarrow A \text{ and } A \mid A \text{ or } A$

$A \rightarrow id \mid (S)$

Solution: The Augmented grammar is,

$S' \rightarrow S$

$S \rightarrow A \text{ and } A$ ... (1)

$S \rightarrow A \text{ or } A$ ... (2)

$A \rightarrow id$ ... (3)

$A \rightarrow (S)$ ... (4)

I_0: $S' \rightarrow \cdot S'$

$S \rightarrow \cdot A \text{ and } A'$

$S \rightarrow \cdot A \text{ or } A$

$A \rightarrow \cdot id$

$A \rightarrow \cdot (S)$

I_1: Goto (I_0, S)

$S' \rightarrow S \cdot$

I_2: Goto (I_0, A)

$S \rightarrow A \cdot \text{and } A'$

$S \rightarrow A \cdot \text{or } A'$

I_3: Goto (I_0, id)

$A \rightarrow id \cdot$

I_4: Goto (I_0, ( )

$A \rightarrow (\cdot S)$

$S \rightarrow \cdot A \text{ and } A'$

$S \rightarrow \cdot A \text{ or } A'$

$A \rightarrow id$

$A \rightarrow \cdot (S)$

I_5: Goto (I_2, and)

$S \rightarrow A \text{ and } \cdot A$

$A \rightarrow \cdot id$

$A \rightarrow \cdot (S)$

I_6: Goto (I_2, or)

$S \rightarrow A \text{ or } \cdot A$

$A \rightarrow \cdot id$

$A \rightarrow \cdot (S)$

$I_{7:}$ Goto (I_4, S)

$A \rightarrow (S \cdot)$

Goto (I_4, A) = I_2

Goto (I_4, id) = I_3

Goto (I_4, C) = I_4

$I_{8:}$ Goto (I_5, A)

$S \rightarrow A \text{ and } A \cdot$

Goto (I_5, id) = I_3

Goto (I_5, C) = I_4

$I_{9:}$ Goto (I_6, A)

$S \rightarrow A \text{ or } A \cdot$

Goto (I_6, ( ) = I_4

$I_{10:}$ Goto (I_7,))

$A \rightarrow (S) \cdot$

First S = {id, (} Follow (S) = {$,) }

First A = {id, (} Follow (A) = {and, or $,)}

State	and	Action	id	(	)	$	Goto S	A
0			S_3	S_4			1	2
1						Accept		
2	S_5	S_6						
3	r_3	r_3			r_3	r_3		
4			S_3	S_4			7	
5			S_3	S_4				8
6			S_3	S_4				9
7					S_{10}			
8						r_1		
9						r_2		
10	r_4	r_4			r_4	r_4		

$\therefore$ There are no multiple entries. The grammar is SLR(1).

Example 47: Show that the following grammar is SLR(1) but not LL(1)?

$S \rightarrow SA \mid A$

$A \rightarrow a$

Solution: 1. To check grammar is SLR (1).

Augmented grammar is:

$S' \rightarrow S$

1. $S \rightarrow SA$
2. $S \rightarrow A$
3. $A \rightarrow a$

Now, compute LR (0) set of items.

I_0: $S' \rightarrow \cdot S$

$S \rightarrow \cdot SA$

$S \rightarrow \cdot A$

$A \rightarrow \cdot a$

Goto (I_0, S)

I_1: $S' \rightarrow S \cdot$

$S \rightarrow S \cdot A$

$A \rightarrow \cdot a$

Goto (I_0, A)

I_2: $S \rightarrow A \cdot$

Goto (I_0, a)

I_3: $A \rightarrow a \cdot$

Goto (I_1, A)

I_4: $S \rightarrow SA \cdot$

Goto (I_1, a) = I_3

SLR Table

State	Action		Goto	
	a	$	S	A
0	S3		1	2
1	S3	accept		4
2	r2	r2		
3	r3	r3		
4	r1	r1		

There is no multiply entires so the grammar is SLR (1).

FOLLOW (S) = {$, a} = FOLLOW (A)

2. To check grammar is not LL (1):

The grammar is left-recursive. After eliminating left-recursion we get,

$S \rightarrow AS'$

$$S' \rightarrow AS' \mid \in$$
$$A \rightarrow a$$

The FIRST and FOLLOW are as follows:

FIRST (S) = {a}
FIRST (A) = {a}
FIRST (S') = {a, ∈}
FOLLOW (S) = {$}
FOLLOW (S') = {$}
FOLLOW(A) = {$}

Construction of LL (1) Parsing table:

	a	$
S	$S \rightarrow AS'$	
A	$A \rightarrow a$	
S'	$S' \rightarrow AS'$ $S' \rightarrow \in$	$S' \rightarrow \in$

The table has multiply define entries.

Therefore, grammar is not LL (1).

3.8.4 Canonical LR Parse Table

- The canonical LR parser makes full use of the lookahead symbols. It is more powerful than SLR(1).
- This parser uses a large set of items, called the LR(1) items.
- Steps in construction of canonical LR parsing table:
  1. Construct sets of LR(1) items.
     Draw DFA for all goto.
  2. Construct canonical LR parsing table.
- **Construction of sets of canonical LR(1) items:** Two functions closure and goto are used.
  1. **closure (I):**

```
{
   for each item [A → α·Bβ, a] in I,
   and each production B → ·γ in G'
   and each terminal b in FIRST (·a)
   do the following,
   if [B → · γ, b] is not in I,
```

```
        then add [B → · γ, b] in I
        until no more items can be added to I;'
    }
```

2. goto (I, X):

```
    {
        If [A → α · Xβ, a] is in set of items,
        add [A → αX·β, a] in I.
    }
```

3. To find LR(1) items:

```
Let A = {closure ([S' → · S, $])}
```

for each set of items in I, and for each grammar symbol X, add goto (I, X) to A if it was not present until no more items can be added to A.

Example 48: Consider the following grammar:

$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

Solution: First make it augmented grammar as,

$$S' \rightarrow S$$
$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

Now, find LR(1) set as follows:

Start with $[S' \rightarrow \cdot S, \$]$

$$I_{0:}\ S' \rightarrow \cdot S\ ,\ \$$$

Since, after dot S is non-terminal so use procedure for closure to add items.

Here, $A = S'$, $B = S$, $\alpha = \in$ and $\beta = \in$, $a = \$$.

for $[A \rightarrow \alpha \cdot B\beta, a]$ and first (βa) = first $(\in \$) = \{\$\}$.

So we add the item

$$S \rightarrow \cdot AA, \$$$

Now, again after dot non-terminal A so again apply precedence closure.

For production $A \rightarrow \cdot AA, \$$

Here, $A = S$, $\alpha = \in$, $B = A$, $\beta = A$ and $a = \$$. (apply $A \rightarrow \alpha \cdot B\beta$)

$$\text{first } (\beta a) = \text{first } (A) = \{a, b\}$$

So we add the items for A as to follows with lookaheads a and b.

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

Now, the I_0 set is,

$$I_{0:}\ S' \rightarrow \cdot S, \$$$
$$S \rightarrow \cdot AA, \$$$
$$A \rightarrow \cdot aA, a|b$$
$$A \rightarrow \cdot b, a|b$$

Now find I_1, Goto (I_0, S)

$$\boxed{I_1 : S' \rightarrow S \cdot, \$}$$

Goto (I_0, A)

$$I_{2:}\ S \rightarrow A \cdot A, \$$$

(while using goto function lookahead will not change)

Now, we have to add A-productions using closure rule

$$A \rightarrow \alpha \cdot B\beta, \$$$
$$\alpha = A,\quad B = A,\quad \beta = \in \text{ and } a = \$$$

$\therefore$ first (βa) = \$

we add,

$$A \rightarrow \cdot aA, \$$$
$$A \rightarrow \cdot b, \$$$

The I_2 is

$$I_{2:}\ S \rightarrow A \cdot A, \$$$
$$A \rightarrow \cdot aA, \$$$
$$A \rightarrow \cdot b, \$$$

Similarly, we proceed further,

Goto (I_0, a)

$$I_{3:}\ A \rightarrow a \cdot A,\ a|b$$
$$A \rightarrow \cdot aA,\ a|b$$
$$A \rightarrow \cdot b, a|b$$

Goto (I_0, b)

$$I_{4:}\ A \rightarrow b \cdot, a|b$$

Goto (I_2, A)

$$I_{5:}\ S \rightarrow AA \cdot, \$$$

Goto (I_2, a)

$$I_{6:}\ A \rightarrow a \cdot A, \$$$
$$A \rightarrow \cdot aA, \$$$
$$A \rightarrow \cdot b, \$$$

Goto (I_2, b)

$I_{7:}$ A → b ·, $

Goto (I_3, A)

$I_{8:}$ A → aA ·, a | b

Goto (I_6, A)

I_9: A → aA ·, $

Goto (I_6, a) = I_6

Goto (I_6, b) = I_7

No more items can be added.

We get I_0 to I_9 are the set of LR(1) items. The DFA using goto is shown in Fig. 3.40.

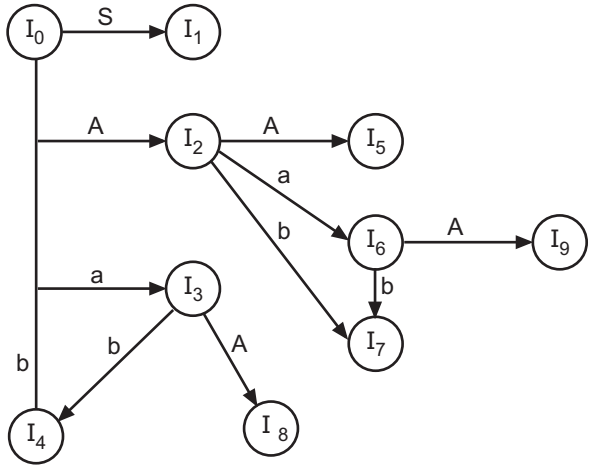


Fig. 3.40: DFA for goto Function

- **Algorithm for Construction of Canonical LR Parsing Table:**
  1. Construct A = {I_0, I_1, ..., I_n}, the collection of sets of LR(1) items for augmented grammar G'.
  2. State of parsers are i = 0, 1, ... n for I_0, I_1, ..., I_n. The parsing action for state i is determined as follows:
     - If [A → α·aβ, b] is in I_i, and goto (I_i, 0) = I_j, then action [i, a] = shift j, where a is terminal.
     - If [A → α ·, a] is in I_i, A ≠ S' then action [i, a] → "reduce (A → α)".
     - If [S' → S ·, $] is in I_i, then action [i, $] = accept.
     - If goto (I_i, A) = I_j then goto [i, A] = j.
     - All undefined entries are error.
- If any conflict occurs in parsing table by applying above rules then grammar is not LR(1). There should not be any multiply-define entries.

- Every SLR(1) grammar is an LR(1) and SLR parser may have less states than LR(1) parser for the same grammar because LR parsing uses lookaheads.
- The LR(1) parsing table is shown in Fig. 3.41 for grammar of Example 47.

State	Action			Goto	
	a	b	$	S	A
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 3.41: Canonical Parsing Table

3.8.5 Constructing LALR Parsing Table

- LALR or lookahead LR is more powerful than SLR.
- The number of states in SLR and LALR are the same. In this method, we combine the set of items whose core part is same and lookahead is different from LR(1) set of parsers.
- From Fig. 3.42 we find the following items having same core, so we take their union as follows:

$$
\begin{aligned}
I_{36:}\ & A \rightarrow a \cdot A, && a|b|\$ \\
& A \rightarrow \cdot aA, && a|b|\$ \\
& A \rightarrow \cdot b, && a|b|\$ \\
I_{47:}\ & A \rightarrow b \cdot, && a|b|\$ \\
I_{89:}\ & A \rightarrow aA \cdot, && a|b|\$
\end{aligned}
$$

- **Algorithm for LALR Table Construction:**
  1. Construct A = {I_0, I_1, ..., I_n}, the collection of sets of LR(1) items.
  2. Find all the sets having same core and replace these sets by their union.
  3. The resulting set of LR(1) items are A' = {J_0, J_1, ..., J_m} (say). The parsing actions for state i are constructed from J_i is same as LR(1) parsing table.

4. The goto table.
   The J is the union of LR(1) item set means $J = I_1 \cup I_2 \cup ... \cup I_k$ then goto (I_1, X), goto (I_2, X), ... goto (I_k, X) are the same as I_1, I_2, ..., I_k. Let B be the union of all sets of items having the same core as goto (I_i, X). Then goto (J, X) = B.
5. If the parsing table consists of any conflict then grammar is not LALR(1).

- The LALR parsing table for example is shown in Fig. 3.42.

State	Action			Goto	
	a	b	$	S	A
0	s36	s47		1	2
1			acc		
2	s36	s47			
36	s36	s47			5
47	r3	r3	r3		89
5			r1		
89	r2	r2	r2		

Fig. 3.42: LALR Parsing Table

Example 49: Consider the following grammar:

$$S' \rightarrow S$$
$$S \rightarrow aAd \mid bBd \mid aBe \mid bAe \qquad \textbf{(Grammar 3.8)}$$
$$A \rightarrow c$$
$$B \rightarrow c$$

Find the grammar is LALR(1) or not?

Solution:

$S' \rightarrow S$

1. $S \rightarrow aAd$
2. $S \rightarrow bBd$
3. $S \rightarrow aBe$
4. $S \rightarrow bAe$
5. $A \rightarrow c$
6. $B \rightarrow c$

I_0: $S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot aAd, \$$
$S \rightarrow \cdot bBd, \$$
$S \rightarrow \cdot aBe, \$$
$S \rightarrow \cdot bAe, \$$

Goto (I_0, S)

I_1: S' → S ·, \$

Goto (I_0, a)

I_2: S → a · Ad, \$
S → a · Be, \$
A → · c, d
B → · c, e

Goto (I_0, b)

I_3: S → b · Bd, \$
S → b · Ae, \$
B → · c, d
A → · c, e

Goto (I_2, A)

I_4: S → aA · d, \$

Goto (I_2, B)

I_5: S → aB · e, \$

Goto (I_2, C)

I_6: A → c ·, d

Goto (I_3, B)

I_7: S → bB · d, \$

Goto (I_3, A)

I_8: S → bA · e, \$

Goto (I_3, c)

I_9: B → c ·, d
A → c ·, e

Goto (I_4, d)

I_{10}: S → aAd ·, \$

Goto (I_5, e)

I_{11}: S → aBe ·, \$

Goto(I_7, d)

I_{12} : S → bBd ·, \$

Goto (I_8, e)

I_{13}: S → bAe ·, \$

The LR(1) parsing table is shown in Fig. 3.43.

State	Action						Goto		
	a	b	c	d	e	$	S	A	B
0	S2	S3					1		
1						acc		4	5
2			S6					8	7
3			S9						
4				S10					
5					S11				
6				r5	r6				
7				S12					
8					S13				
9				r6	r5				
10					r1				
11					r3				
12					r2				
13					r4				

Fig. 3.43: The LR(1) Parsing Table

There are no multiply-define entries.

$\therefore$ The grammar is LR(1).

To find LALR(1) Parsing Table: Merge similar core items. There are two items I_6 and I_9 which merge together and taking union of them, we get

$$I_{69}: A \rightarrow c \cdot, e \mid d$$
$$B \rightarrow c \cdot, e \mid d$$

LALR(1) table is shown in Fig. 3.44 for Grammar 3.8.

State	Action						Goto		
	a	b	c	d	e	$	S	A	B
0	S2	S3					1		
1						acc		4	5
2			S6					8	7
3			S9						
4				S10					
5					S11				
69				r5/r6	r6/r5				

7				S12					
8					S13				
10						r1			
11						r3			
12						r2			
13						r4			

Fig. 3.44: LALR(1) Table for Grammar 3.8

There is reduce-reduce conflict.

So grammar is not LALR.

Example 50: Check following grammar is LR (1) grammar or not?

$$S \rightarrow bA \mid a$$
$$A \rightarrow CB \mid CBe$$
$$B \rightarrow dB$$

Solution: In the above grammar C is useless symbol. After eliminating productions of C symbol we get,

$$S \rightarrow bA \mid a$$
$$B \rightarrow dB$$

Again A and B are useless. Therefore grammar is $S \rightarrow a$,

Augumented grammar is,

$$S' \rightarrow S$$
$$S \rightarrow a$$

Now find LR (1) items:

I_0: $S' \rightarrow \cdot S, \$$

$S \rightarrow \cdot a, \$$

Goto (I_0, 5)

I_1: $S' \rightarrow S \cdot, \$$

Goto (I_0, a)

I_2: $S \rightarrow a \cdot, \$$

Canonical LR parsing table is as follows:

State	Action		Goto
	a	$	S
0	S2		1
1		accept	
2		r1	

Example 51: Check following grammar is canonical LR (1) or not ?

$$S \rightarrow E = E \mid a$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * a \mid a$$

[April 16]

Solution: Augmented grammar is

$$S' \rightarrow S$$
1. $S \rightarrow E = E$
2. $S \rightarrow a$
3. $E \rightarrow E + T$
4. $E \rightarrow T$
5. $T \rightarrow T * a$
6. $T \rightarrow a$

Now, compute set of LR (1) items:

I_0: $S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot E = E, \$$
$S \rightarrow \cdot a, \$$
$E \rightarrow \cdot E + T, + \mid =$
$E \rightarrow \cdot T, + \mid =$
$T \rightarrow \cdot T * a, + \mid = \mid *$
$T \rightarrow \cdot a, + \mid = \mid *$

Goto (I_0, S)

I_1: $S' \rightarrow S \cdot, \$$

Goto (I_0, E)

I_2: $S \rightarrow E \cdot = E, \$$
$E \rightarrow E \cdot + T, + \mid =$

Goto (I_0, a)

I_3: $S \rightarrow a \cdot, \$$
$T \rightarrow a \cdot, + \mid = \mid *$

Goto (I_0, T) = I_4

I_4: $E \rightarrow T \cdot, + \mid =$
$T \rightarrow T \cdot * a, + \mid = \mid *$

Goto (I_2, =)

I_5: $S \rightarrow E = \cdot E, \$$
$E \rightarrow \cdot E + T, + \mid \$$
$E \rightarrow \cdot T, + \mid \$$
$T \rightarrow \cdot T * a, \$ \mid + \mid *$
$T \rightarrow \cdot a, \$ \mid + \mid *$

Goto (I_2, +)

I_6: E → E + · T, + 1 =
T → · T * a, + | = | *
T → · a, + | = | *

Goto (I_4, *) = I_7

I_7: T → T * · a, + | = | *

Goto (I_5, E)

I_8: S → E = E ·, $
E → E · + T, + | $

Goto (I_5, T)

I_9: E → T ·, + | $
T → T · * a, + | $ | *

Goto (I_5, a)

I_{10}: T → a ·,$ | * | +

Goto (I_6, T)

I_{11}: E → E + T ·, + | =
T → T · * a, + | = | *

Goto (I_6, a)

I_{12}: T → a ·, + | = | *

Goto (I_7, a)

I_{13}: T → T * a ·, + | = | *

Goto (I_8, +)

I_{14}: E → E + · T, + | $
T → · T * a, + | $ | *
T → · a, + | $ | *

Goto (I_9, *)

I_{15}: T → T * · a, + | $ | *

Goto (I_{14}, T)

I_{16}: E → E + T *, + | $
T → T · * a, $ | + | *

Goto (I_{14}, a)

I_{17}: T → a ·, + | $ | *

Goto (I_{15}, a)

I_{18}: T → T * a ·, + |$| *

No more items can be added in the set,

FOLLOW (S) = {$}

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

(T) = {*, +, =, $}

Now, construct the parsing table of LR(1) as follows:

State	Action					Goto		
	=	a	+	*	$	S	E	T
0	.	S3				1	2	4
1					accept			
2	S5		S6					
3	r6	r6		r6	r2			
4	r4		r4	S7				
5		S10					8	9
6								11
7		S13						
8			S14		r1			
9			r4	S15	r4			
10			r6	r6	r6			
11	r3		r3	S7				
12	r6		r6	r6				
13	r5		r5	r5				
14		S17						16
15		S18						
16		r3		S15	r3			
17		r6		r6	r6			
18		r5		r5	r5			

There is no multiply entries in the table. Therefore, grammar LR (1) grammar.

Example 52: Check following grammar is LR (1) grammar or not?

$S \rightarrow AB$

$A \rightarrow BSB \mid BB \mid b$

$B \rightarrow aAb \mid a$

Solution: To find LR (1) set: first grammar is augmented.

$S' \rightarrow S$

1. $S \rightarrow AB$
2. $A \rightarrow BSB$

3. A → BB
4. A → b
5. B → aAb
6. B → a

FOLLOW (S) = {$) FIRST (S) = FIRST (A) {b, a}

FOLLOW (A) = {a, b} FIRST (B) = {a}

(B) = {$, a, b}

Now find out LR (1) items:

$I_{0:}$ S' → · S, $

S → · AB, $

A → · BSB, a

A → · BB, a

A → · b, a

B → · aAb, a | b

B → · a, a | b

Goto (I_0, S) = I

I_1: S' → S ·, $

Goto (I_0, A)

I_2: S → A · B, $

B → · aAb, $

B → · a, $

Goto (I_0, B)

I_3: A → B · SB, a

A → B · B, a

S → · AB, a

A → · BSB, a

A → · BB, a

A → · b, a

B → · aAb, a | b

B → · a, a | b

Goto (I_0, b)

I_4: A → b ·, a

Goto (I_0, a)

$I_{5:}$ B → a · Ab, a | b

B → a ·, a | b

A → · BSB, b
A → · BB, b
A → · b, b
B → · aAb, a | b
B → · a, a | b

Goto (I_2, B)

I_6: S → AB ·, $

Goto (I_2, a)

I_7: B → a · Ab, $
B → a ·, $
A → · BSB, b
A → · BB, b
A → · b, b
B → · aAb, a | b
B → · a, a | b

Goto (I_3, S)

I_8: A → BS · B, a
B → · aAb, a
B → · a, a

Goto (I_3, B)

I_9: A → BB ·, a
A → B · SB, a
A → B · B, a
S → · AB, a
A → · BSB, a
A → · BB, a
A → · b, a
B → · aAb, a | b
B → · a, a | b

Goto (I_3, A)

I_{10}: S → A · B, a
B → · aAb, a
B → · a, a

Goto (I_3, b) = I_4

Goto (I_3, a) = I_5

Goto (I_5, A)

I_{11}: B → aA · b, a | b

Goto (I_5, B)

I_{12}: A → B · SB, b

A → B · B, b

S → · AB, a

A → · BSB, a

A → · BB, a

A → · b, a

B → · aAb, a | b

B → · a, a | b

Goto (I_5, a)

I_{13}: B → a ·, a | b

B → a · Ab, a | b

A → · BSB, a | b

A → · BB, a | b

A → · b, a | b

Goto (I_5, b)

I_{14}: A → b ·, b

Goto (I_7, A)

I_{15}: B → aA · b, $

Goto (I_7, B) = I_{12}

Goto (I_7, b) = I_{13}

Goto (I_7, a) = I_5

Goto (I_8, B)

I_{16}: A → BSB ·, a

Goto (I_8, a)

I_{17}: B → a · Ab, a

B → a ·, a

A → · BSB, b

A → · BB, b

A → · b, b

B → · aAb, a | b

B → · a, a | b

Goto (I_9, S) = I_8

Goto (I_9, B) = I_9

Goto (I_9, A) = I_{10}

Goto (I_9, b) = I_4

Goto ($I_{10,\ B)}$

I_{18}: S → AB ·, a

Goto (I_{10}, a) = I_{15}

Goto (I_{11}, b)

I_{19}: B → aAb ·, a | b

Goto (I_{12}, S)

$I_{20:}$ A → BS · B, b

B → · aAb, b

B → · a, b

Goto (I_{12}, B)

I_{21}: A → BB ·, b

A → B · SB, a

A → B · B, a

S → · AB, a

A → · BSB, a

A → · BB, a

A → · b, a

B → · aAb, a | b

B → · a, a | b

Goto (I_{12}, a) = I_{17}

Goto (I_{12}, b) = I_4

Goto (I_{15}, b)

I_{22}: B → aAb ·, $

Goto (I_{17}, A)

I_{23}: B → aA · b, a

Goto (I_{17}, B) = I_{12}

Goto (I_{17}, a) = I_5

Goto (I_{16}, A)

I_{24}: B → aA · b, a | b

Goto (I_{16}, B)

I_{25}: A → B · SB, a | b

A → B · B, a | b

S → · AB, a | b
A → · BSB, a | b
A → · BB, a | b
B → · aAb, a | b
B → · a, a | b

Goto (I_{19}, B)

I_{26}: A → BSB ·, b

Goto (I_{19}, a)

I_{27}: B → a · Ab, b
B → a ·, b
A → · BSB, b
A → · BB, b
A → · b, b
B → · aAb, a | b
B → · a, a | b

Goto (I_{20}, S) = I_8
Goto (I_{20}, B) = I_9
Goto(I_{20}, A) = I_{10}
Goto (I_{20}, b) = I_4
Goto (I_{20}, a) = I_5

Goto (I_{24}, b)

I_{28}: B → aAb ·, a | b

No more items can be added.

So construct LR(1) parsing table,

State	Action			Goto		
	a	b	$	S	A	B
0	S5	S4		1	2	3
1			accept			
2	S7					6
3		S4		8	10	9
4	r4					
5	S13/r6	S14/r6				
6			r1			
7	S15	S13	r6		13	12
8	S15					

9	S5/r6	S4		8	10	9
10	S15					16
11		S19				
12	S17	S4		20		21
13						
14	r2					
15	S15/r6	S22				
16	r2					
17	S5					12
18	r1					
19	r5	r5				26
20	S5	S4		8	10	9
21		r3/S4				
22			r5			
23	S28					
24		S28				
25						
26		r2				
27	S13	S14				
28	r5	r5				

The above grammar is not LR(1) grammar.

Example 53: Check following grammar is LALR(1) grammar or not?

$$S \rightarrow bABc$$
$$A \rightarrow Ad ; \mid \in$$
$$B \rightarrow B ; C \mid c$$
$$C \rightarrow a$$

Solution: The agumented grammar is,

$$S' \rightarrow S$$

1. $S \rightarrow bABc$
2. $A \rightarrow Ad;$
3. $A \rightarrow \in$
4. $B \rightarrow B; C$
5. $B \rightarrow C$
6. $C \rightarrow a$

Now find LR (1) items,

I_0: S' → · S, $

S → · bABc, $

Goto (I_0, S)

I_1: S' → S ·, $

Goto (I_0, b)

I_2: S → b · ABc, $

A → · Ad; , c

A → · , c

Goto (I_2, A)

I_3: S → bA · Bc, $

B → · B; C, c

B → · c, c

A → A · d; , c

Goto (I_3, B)

I_4: S → bAB · c, $

B → B · ; C, c

Goto (I_3, c)

I_5: B → c ·, c

Goto (I_3, d)

I_6: A → Ad · ; , c

Goto (I_4, c)

I_7: S → bABc ·, $

Goto (I_4, ;)

I_8: B → B; · C, c

C → · a, c

Goto (I_6, ;)

I_9: A → Ad; ·, c

Goto (I_8, C)

I_{10}: B → B; C ·, c

Goto (I_8, a)

I_{11}: C → a ·, c

No more items can be added. Now, construct LR (1) table.

State	Action						Goto		
	a	b	c	d	;	$	S	A	B
0		S2					1		
1						accept			
2			r3					3	
3			S5	S6					4
4			S7		S8				
5			r5						
6					S9				
7						r1			
8	S11								10
9			r2						
10			r4						
11			r6						

The above grammar is LR (1) grammar, since there is no multiply entries.

In this, merge is not possible. Therefore, above grammar is LALR(1).

Example 54: Consider the following grammar,

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L, S \mid s$$

Find the operator precedence relation and parse the string (a, (a, a)) using precedence relation.

Solution:

	(	)	a	,
(	<	=	<·	·>
)	·	·>	<·	<·
a	·>	·>		·>
,	<·	·>	<·	

The string is (a, (a, a)).

Stack	Input	Precedence Relation
$	(a, (a, a))	
$ (	a, (a, a))	push (·̇· C < a)
$ (a	, (a, a))	pop (·> a > ,)

$ (S	, (a, a))	push
$ (S,	(a, a))	push
$ (S, (	a, a))	push
$ (S, (a	, a))	pop (·>a > ,)
$ (S, (S	, a))	pop
$ (S, (L	, a))	push
$ (S, (L,	a))	push
$ (S, (L, a	))	pop (·>a >))
$ (S, (L, S	))	push
$ (S, (L, S)	)	pop
$ (S , (L)	)	push
$ (S, S)	$	pop
$ (L, S)	$	pop
$ (L)	$	pop
$ S	$	accept

3.9 PARSER GENERATOR (YACC) [April 16]

- YACC stands for "Yet Another Compiler – Compiler". YACC assists in the next phase of the compiler.
- YACC creates a parser which will be output in a form suitable for inclusion in the next phase.
- YACC is a utility on UNIX used for parser generator. YACC program transforms the yacc file (e.g. y1·y) into a C program called y·tab·c.
- The program y·tab·c is a representation of an LALR parser written in C, along with other C routines.
- By compiling y·tab·c along with the ly library that contains the LR parsing program using the command

  $ cc y·tab·c – ly
- We can obtain the desired object program a·out that performs the translation specified by the original yacc program.
- A YACC source program has three parts:

```
declarations
% %
translation rules
% %
supporting c-routines
```

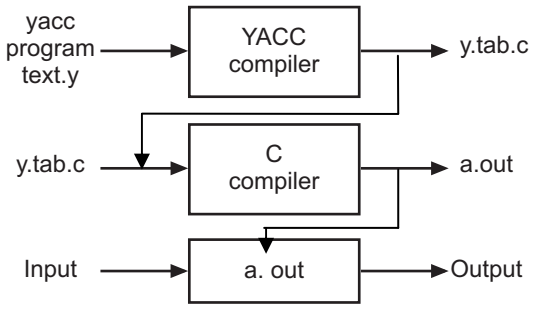


Fig. 3.45: Process of Translation Program

The Declaration Section:

- It includes declaration of tokens along with its values used on the power stack which declared in lex program. The normal C code declaration part written within % {and %}.
- Then we declared tokens and start symbol as follows:

  % start start symbol

  % token name value
- We can use tokens without declaration by enclosing them in single quote as '=', '+'.
- We can define more than one token as % token token1 value1 token2 value2 token3 value3 .....
- We end above section using %% and then define the rule of grammar. These rules are production rules of grammar and associated with semantic action. We define production rules of grammar as:

  left hand side → production1 | production2 | production3 |
- This will define yacc specification as:

```
left hand side   : production 1   {semantic action1}
                 | production 2   {semantic action 2}
                 :
                 :
                 | production n   {semantic action n}
                 ;
```

  Instead of '→ here: (colon) is used.
- In YACC production if we write single character enclosed within single quote then it is treated as terminal.
- The string letters and digit which are not enclosed within quotes are treated as non-terminal symbol.
- One or more production rules are separated by vertical bar and each production rule ends with semicolon. Always first left-hand side is taken to be start symbol.

- For example, We define grammar for expression is,

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

  yacc rule are:

```
% %
E  :  E '+' E
   |  E * E
       :
   |  id;
```

- Every symbol in a **yacc** parser has a value which gives information about a particular instance of a symbol.
- Whenever, a parser reduces a rule, it executes user 'C' code for that rule i.e. rule's action. The action is written at the end of the rule before semicolon or vertical bar.
- The action code can refer to the values of the right hand side symbols as $1, $2, ... and can set the value of the LHS by setting $$. In above grammar we can write action as follows:

```
E  :  E '+' E          {($$ = $1 + $3);}
   |  E '*' E{($$ = $1 * $3);}
        :
   |  id               {$$ = $1;}
```

  Here, first E has value $1, then operator has value $2, and then second E has value $3.

To compile and execute yacc program:

```
$ lex lexprogram · l
$ cc  lex·yy·c
$ yacc – d yaccprogram · y   (makes y·tab·c and y·tab·h)
$ cc  y·tab·c – ly
$  ·\a·out.
```

OR

```
$ lex  lexprogram·l
$ yacc – d yaccprogram·y
$ cc – o  outfile y·tab·c lex·yy·c  – ly – ll   (compile and link c files)
$ ·\ outfile.
```

- **yyerror( ):** In YACC specification error routine may be added whenever necessary.

  For example: `yyerror()`

```
{
  printf ("expression is not valid");
}
```

For expression grammar $E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid - E \mid (E) \mid id$ we can write YACC specification as below:

```
E  :  E '+' E              {$$ = $1 + $3;}
   |  E '–' E              {$$ = $1 – $3;}
   |  E '*' E{$$ = $1 * $3;}
   |  E '/' E
      {   if ($3= =0)
             yyerror ("divide by zero");
          else
             $$ = $1/$3;
          }
   |  '–' E                {$$ = – $2;}
   |  '(' E ')'            {$$ = $2;}
   |  id                   {$$ = $1;}
;
```

- This grammar is ambiguous grammar. If we compile this, then yacc shows errors such as shift/reduce conflict.
- The YACC cannot find that which action to be taken either shift or reduce a rule. To resolve this, we assign precedences of the operators and also associativity for operators.

Using YACC with Ambiguous Grammars:

- Consider the above ambiguous grammar for our discussion:

$$S \rightarrow E$$
$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid - E \mid (E) \mid id$$

- YACC will resolve all parsing action conflicts using two rules:
  1. Rewriting the grammar by removing conflict. For example, here left-recursion is resolve by using rules which we have discussed earlier, so we are adding more non-terminal symbols and production rules. So that reduce/reduce conflict may be removed.
  2. A shift-reduce conflict can be removed by giving a precedence and associativity to each production involved in a conflict and also for each terminal symbol which is involved in a conflict.
- We can specify precedences explicitly. Find the operators which are left associative or right associative and use keyword left and right before them respectively. This is defined in declaration section as follows:

```
%   left '+'  '–'
%   left '*'  '/'
```

% non-assoc UMINUS

- Declaring in this way, tells the yacc that "+" and "–" are left associative and at the lowest precedence level. "∗' and "/" are also left associative but precedence is higher than + and – operators. For unary minus, a pseudo-token UMINUS is used.
- Suppose if we add production rule in the grammar is **E ↑ E** then, we declare precedence of it as,

  % right '^'

  which is after ∗ and / line, since ^ has highest precedence.
- Whenever the yacc finds shift/reduce conflict due to an ambiguous grammar, it consults the table of precedences and conflict is resolved.
- As we know "–" has low precedence, but we want unary minus to have higher precedence than multiplication or division.
- So we use **% prec UMINUS** in the yacc rule specification which tells the yacc to use precedence for UMINUS (unary minus) not for minus.

  Example for YACC specification for grammar of expression:

```
% {
      #include "lex·yy·c"
% }
% start S
% token id 255
% left '–' '+'
% left '*' '/'
% nonassoc UMINUS
%%
  S :  S                {printf ("%d'n", $1);}
         ;
  E :  E '+' E             {$$ = $1 + $3;}
    |  E '–' E           {$$ = $1 – $3;}
    |  E '*' E           ($$ = $1 * $3;}
    |  E '/' E
          {
            if ($3==0)
            yyerror ("divide by zero");
            else
               $$ = $1/$3
          }
```

```
| '–' E  % prec UMINUS   {$$ = – $2;}
| '(' E ')'             {$$ = $2;}
| id                    {$$ = $1;}
  ;
%%.
```

Example 55: Write lex and yacc specifications for the following grammar:

program → START assign STOP

assign → assign assign | ID = NUM | ID = ID

Solution: lex specification:

```
% {
    # include "y · tab  · h"
    # include <stdio·h>
    # define S  255
    # define sp 256
    # define eq 257
    # define no 258
    # define id 259
% }
% %
[0 – 9]+                        {return (no);}
[ = ]                           {return (eq);}
[Ss] [Tt] [Aa] [Rr] [Tt]        {return (S);}
[Ss] [Tt] [Oo] [Pp]             {return (SP);]
[A – Za–z] [0–9a – zA – Z]* {return (id);}
% %
/ * yacc specification * /
% {
    # include "lex · yy · c"
    # include <stdio·h>
% }
%  start s;
% token s 255 sp 256 eq 257 no 258 id 259;
% %
S   :  sta  asg  sto            {printf ("valid");}
    ;
```

```
asg :  a   asg
    |  a
    ;
a   :  ID   EQ  NUM
    |  ID  EQ    ID
    ;
ID  :  id
    ;
NUM :  no
       ;
EQ  :  eq
    ;
sta :  s
    ;
sto :  sp
    ;
% %
    main()
    {
        printf ("Enter input \n");
        yyparse ( );
    }
    yyerror()
    {
        printf ("ERROR")
    }
```

Example 56: Write yacc specification for grammar for while-do statement in Pascal.

S → while cond do begin stats end

Cond → id op idnum

Stats → id: = idnum

Idnum → id | num

Solution :

```
/*  YACC specification */
% {
    # include "lex · yy · c"
```

```
        # include < stdio · h >
    % }
    % start S
    % token ybegin, yend, ywhile ydo yassign semi
    % token id num op
    % %
         S  :  ywhile cond ydo ybegin stats yend {printf ("valid");}
            ;
      cond  :  id op idnum
            ;
     stats  :  stat stats
            ;
      stat  :  id yassign idnum semi
            ;
     idnum  :  id
           |   num
            ;
% %
   main()
   {
      printf ("Enter statement");
      yyparse ( );
   }
   yyerror()
   {
      printf ("Invalid statement")
   }
```

Difference between Recursive Descent Parser and Predictive Parser:

Sr. No.	Recursive Decent Parser	Predictive Parser
1.	Left recursive grammars are not suitable.	Left recursive grammars are not suitable.
2.	It accepts LL (1) grammar.	It accepts LL (1) grammar.
3.	It uses recursive procedures.	It uses parser table.
4.	Parser requires more space in memory since it is recursive.	This parser requires less space in memory.

5.	Precise error indication is not possible.	It detects the errors using parse table.
6.	First and follow functions are not required.	FIRST and FOLLOW functions are required.

Difference between Top-down and Bottom-up Parsing:

Sr. No.	Top-down Parsing	Bottom-up Parsing
1.	Top-down parser uses derivation.	It uses reduction.
2.	Parsing start from starting symbol and derive the input string.	Parsing start from input string and reduce to the starting symbol.
3.	Left-recursion and backtracking are two problems occurring in this parsing.	No left-recursion and backtracking problems.
4.	Ambiguous grammars are not suitable for top-down parsing.	It accepts ambiguous grammar.
5.	Precise error indication is not possible.	Precise error indication is possible.
6.	This uses LL(1) grammar.	This uses LR grammar.
7.	It scans the input from left-to-right and generates leftmost derivation.	It scans input from left-to-right and generates rightmost deviation.
8.	Top-down parsers are • Recursive descent parser • Predictive parser.	Bottom-up parsers are • Operator precedence parser. • LR parser (SLR, LR (1), LALR)
9.	No conflict occur.	Shift-reduce and reduce-reduce conflict occurs.

Difference between LL Parser and LR Parser:

Sr. No.	LL Parser	LR Parser
1.	The first L in the LL parser is for scanning the input from left to right, and the second L is for the leftmost derivation.	The L in LR parser is for the left to right, and R stands for rightmost derivation in the reverse order.
2.	LL follows the leftmost derivation.	LR follows rightmost derivation in reverse order.
3.	An LL parser amplifies non – terminals.	Terminals are condensed in LR parser.

4.	LL parser constructs a parse tree in a top-down manner.	LR parser constructs a parse tree in a bottom-up manner.
5.	It ends whenever the stack in use becomes empty.	It starts with an empty stack.
6.	It starts with the start symbol.	It ends with the start symbol.
7.	It is easier to write.	It is difficult to write.
8.	Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.
9.	Reads the terminals when it pops out of the stacks.	Reads the terminals while it pushes them into the stack.
10.	Builds the parse tree top-down.	Builds the parse tree bottom-up.
11.	Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
12.	Examples: LL(0), LL(1)	Examples: SLR(1), CLR(1) etc.

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which is the process of determining whether a string of tokens can be generated by a grammar?

   (a) lexical analysis (b) syntax analysis

   (c) semantic analysis (d) None of the mentioned

2. Parsing takes the token produced by lexical analysis as input and generates, a parse tree (or syntax tree).

   (a) a parse tree (b) a syntax tree

   (c) Both (a) and (b) (d) None of the mentioned

3. The program which performs syntax analysis is called as,

   (a) parser or syntax analyzer (b) lexical analyzer

   (c) semantic analyzer (d) None of the mentioned

4. What does a syntactic analyzer do?

   (a) maintain symbol table (b) collect data structure

   (c) create syntax/parse tree (d) None of the mentioned

5. Which is basically a sequence of production rules, in order to get the input string?

   (a) Parse tree (b) Derivation

   (c) Rotation (d) None of the mentioned

6. Types of parsing includes,
   (a) The process of constructing the parse tree which starts from the root and goes down to the leaf is Top-Down Parsing.
   (b) The process of constructing the parse tree which starts from the leaf and goes down to the root is bottom-up parsing.
   (c) Both (a) and (b)
   (d) None of the mentioned
7. Recursive Descent Parsing (RDP) technique is a,
   (a) top-down parsing without backtracking
   (b) top down parsing with backtracking
   (c) right-to-left parsing
   (d) left-to-right parsing
8. Which parser has the capability to predict which production is to be used to replace the input string.
   (a) top-down parser (b) bottom-up parser
   (c) Predictive parser (d) None of the mentioned
9. The input grammar to the parsing phase of the compiler is of type,
   (a) regular (b) context free
   (c) context sensitive (d) None of the mentioned
10. YACC program transform yacc file into which program,
   (a) yywrap.c (b) y.yacc.c
   (c) yylex.c (d) y.tab.c
11. An LALR(1) parser for a grammar can have shift-reduce (S-R) conflicts if and only if,
   (a) The SLR(1) parser for G has S-R conflicts
   (b) The LR(1) parser for G has S-R conflicts
   (c) The LR(0) parser for G has S-R conflicts
   (d) The LALR(1) parser for G has reduce-reduce conflicts
12. A bottom up parser generates,
   (a) Rightmost derivation (b) Leftmost derivation
   (c) Rightmost derivation in reverse (d) Leftmost derivation in reverse
13. Shift reduce parsers are,
   (a) bottom-up parser (b) top-down parser
   (c) May be top down or bottom up (d) nor top down not bottom up

14. Which parser is a top-down parser for a restricted context-free language and which parses the input from Left to right, performing Leftmost derivation of the sentence.
    (a) LR parser (Left-to-right, Rightmost derivation in reverse)
    (b) LL (Left-to-right, Leftmost derivation) parser
    (c) Predictive parser
    (d) None of the mentioned
15. Which is a non-recursive, shift-reduce, bottom-up parser?
    (a) Shift-reduce parsing (b) LL parser
    (c) Predictive parser (d) LR parser
16. If A –>β is a production then reducing βto A by the given production is called handle pruning i.e., removing the children of A from the parse tree. A rightmost derivation in reverse can be obtained by,
    (a) handle pruning (b) predictive pruning
    (c) shift-reduce pruning (d) None of the mentioned
17. Which parser is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, without backtracking?
    (a) canonical LR or LR(1) parser (b) LALR (Look-Ahead LR) parser
    (c) SLR (Simple LR) parser (d) Performance Testing
18. YACC stands for,
    (a) Yet Another Computer – Compiler (b) Yet Another Compiler – Compiler
    (c) Yet Another Compilation – Compiler (d) None of the mentioned

Answers

1. (b)	2. (c)	3. (a)	4. (c)	5. (b)	6. (c)	7. (a)	8. (c)	9. (b)	10. (d)
11. (b)	12. (c)	13. (a)	14. (b)	15. (d)	16. (a)	17. (c)	18. (b)		

Q. II Fill in the Blanks:

1. Syntax analysis checks the _____ structure of the given input, i.e. whether the given input is in the correct syntax or not.
2. _____ analyzers follow production rules defined by means of context-free grammar.
3. The program which performs syntax analysis is called as _____.
4. CLR parsing uses the _____ collection of LR (1) items to construct the CLR (1) parsing table.
5. If the sentential form of an input is scanned and replaced from left to right, it is called _____ derivation.
6. A syntax analyzer takes the input from a lexical analyzer in the form of _____ streams.
7. A parse tree depicts associativity and precedence of _____.

8. _____ parsers begin with the root and grow the tree toward the leaves.
9. _____ factoring is a grammar transformation which is useful for producing a grammar suitable for recursive-decent parsing or predictive parsing.
10. If one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production using _____.
11. A canonical LR parser or LR(1) parser is an LR(k) parser for k=1, i.e. with a _____ lookahead terminal.
12. A parse tree is a _____ depiction of a derivation.
13. An LL parser accepts _____ grammar.
14. If an operand has operators on both sides (left and right), the side on which the operator takes this operand is decided by the _____ of those operators.
15. First and Follow sets can provide the _____ position of any terminal in the derivation.
16. _____ descent parsing suffers from backtracking.
17. The parse tree is constructed by using the _____ grammar of the language and the input string.
18. _____ parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions Action and Goto
19. In _____ parser, the input string will be reduced to the starting symbol.
20. The sentential form (string) which matches the RHS of production rule while reduction, then that string is called _____.
21. Recursive descent is a _____ parsing technique that constructs the parse tree from the top and the input is read from left to right.
22. A form of recursive-descent parsing that does _____ require any back-tracking is known as predictive parsing.
23. The _____ can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.
24. A CFG is said to _____ grammar if there exists more than one derivation tree for the given input string i.e., more than one LeftMost Derivation Tree (LMDT) or RightMost Derivation Tree (RMDT).
25. An operator _____ grammar is a context-free grammar that has the property that no production has either an empty right hand side (null productions) or two adjacent non-terminals in its right-hand side.
26. Predictive parsing uses a _____ and a parsing _____ to parse the input and generate a parse tree.
27. _____ is a utility on UNIX used for parser generator.
28. The process of discovering a handle and reducing it to the appropriate left hand side is called handle _____.

Answers

1. syntactical	2. Syntax	3. parser	4. canonical
5. left-most	6. token	7. operators	8. Top-down
9. Left	10. Backtracking	11. single	12. graphical
13. LL	14. associativity	15. actual	16. Recursive
17. pre-defined	18. LR	19. shift-reduce	20. handle
21. top-down	22. not	23. reduction	24. ambiguous
25. precedence	26. stack, table	27. YACC	28. pruning

Q. III State True or False:

1. Syntax analysis is also known as parsing.
2. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.
3. If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
4. Leading and Trailing are functions specific to generating an operator-precedence parser, which is only applicable if you we an operator precedence grammar.
5. An operator precedence parser is a bottom-up parser that interprets an operator-precedence grammar.
6. YACC assists in the previous phase of the compiler.
7. In left factoring technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.
8. Bottom-up parsers begin with the leaves and grow the tree toward the root.
9. The LR parser is a shift-reduce, top-down parser.
10. If two different operators share a common operand, the precedence of operators decides which will take the operand.
11. A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol.
12. Left-recursive grammar is considered to be a problematic situation for top-down parsers.
13. The LL (1) grammars are suitable for bottom-up parsing.
14. LR parsing tables are a two-dimensional array in which each entry represents an Action or Goto entry.
15. Handle pruning forms the basis for a bottom-up parsing.
16. LALR parser, parse a text according to a set of production rules specified by a formal grammar.
17. LR parsers are also known as LR(k) parsers.

18. A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Answers

1. (T)	2. (T)	3. (T)	4. (T)	5. (T)	6. (F)	7. (T)	8. (T)	9. (F)	10. (T)
11. (T)	12. (T)	13. (F)	14. (T)	15. (T)	16. (T)	17. (T)	18. (T)		

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What syntax analysis?
2. "Top-down parsing can be implemented using shift-reduce method". State true or false.
3. What do LL and LR stand for?
4. List types of parsers.
5. What is the purpose of parser?
6. Define alphabet.
7. What is grammar?
8. Define reduction.
9. What is derivation?
10. Define parse tree.
11. What is meant by ambiguous grammar?
12. Define top-down parser.
13. What does second 'L' stand for in LL(1) Parser?
14. Define backtracking.
15. Give need for left factoring.
16. Define recursive-descent parsing.
17. YACC is a LR parser. Justify.
18. Define predictive parser.
19. Define parse table.
20. What is LL(1) grammar?
21. Define bottom-up parsing.
22. Define Canonical LR parser.
23. Define operator precedence.
24. What is meant by operator precedence grammar?
25. What is leading and trailing.
26. Define precedence function.
27. Construct LR(1) items for the following production $S \rightarrow a$.

28. Define shift-reduce parsing.
29. “YACC is parser”. State true or false.
30. Define handle.
31. Define handle pruning.
32. List types of LR parser.
33. Differentiate between SLR and LALR parsers.
34. What is the purpose of YACC?
35. Which type of conflict is not possible in LR parser?
36. Define SLR parser.
37. 'Every SLR (1) grammar is LR (1) grammar.' Comment.

(B) Long Answer Questions:

1. What parsing? With the help of diagram describe process of parsing.
2. What is parser? What are the tasks are performed by it? Explain with example.
3. With the help of example describe derivation tree?
4. What is the purpose of ambiguous grammar? Describe with example.
5. Describe bottom-up parser with example.
6. Explain top-down parser with example.
7. Write a short note on: Top-down parsing with backtracking with its drawbacks.
8. What is elimination of left recursion? Describe with example.
9. Describe Recursive Descent Parsing (RDP) in detail.
10. How to construct a predictive parsing table? Explain with example.
11. How to implement RDP using recursive procedures?
12. What is predictive parser? Explain in detail.
13. What is LL parser? Explain with its model.
14. How to implement LL(1) parser? Explain with example.
15. What is parsing table? How to construct it? Describe with example.
16. What is LL(1) grammar? Describe in detail.
17. What is bottom-up parser? What are its types? Explain two of them in short.
18. What is operator precedence parser? Explain with example.
19. What is operator precedence grammar? Describe with example.
20. What is LEADING ad TRAILING? Define them? Also explain with example.
21. Give algorithm for LEADING ad TRAILING.
22. Write a short note on: Operator precedence parsing algorithm.
23. What is precedence function? What is its use in parsing? Explain with example.

24. With the help of steps describe shift-reduce parser. Also state its purpose, advantages and disadvantages.
25. Give relationship between reduction, handle and handle pruning in detail.
26. With the help of example describe stack implementation of shift-reduce parser.
27. Describe LR parser with its advantages.
28. What is SLR parsing? How it works? Explain diagrammatically.
29. With the help of diagram describe model for LR parser.
30. What is SLR parser? How to work it? Explain diagrammatically.
31. What is CLR parser? How to work it? Explain diagrammatically.
32. Differentiates between SLR, CLR and LALR parsers.
33. What is YACC? Describe in detail.
34. Differentiate between recursive descent parser and predictive parser:
35. Differentiate between top-down and bottom-up parsing:
36. Differentiate between LL parser and LR parser.
37. Construct parsing table and check whether the following grammar is LL(1):

    $S \rightarrow BC \mid AB$
    $A \rightarrow aAa \mid b$
    $B \rightarrow bAa \mid \in$
    $C \rightarrow \in$

38. Check whether the following grammars are SLR (1):

    (i) $S \rightarrow AS \mid a$
        $A \rightarrow SA \mid b$

    (ii) $S \rightarrow Aa \mid bAc \mid dc \mid bda$
        $A \rightarrow d$

    (iii) $S \rightarrow bAB \mid aA$
        $A \rightarrow Ab \mid b$
        $B \rightarrow aB \mid a$

    (iv) $S \rightarrow$ A and A | A or A
        $A \rightarrow id \mid (S)$

    (v) $S \rightarrow L = R \mid R$
        $L = * R \mid id$
        $R \rightarrow L$

39. Check whether the following grammars are LL (1) :

    (i) $S \rightarrow ScB \mid cA$
        $B \rightarrow bB \mid \in$
        $A \rightarrow AaB \mid B$

(ii) $S \rightarrow S\ \#$
$S \rightarrow aA \mid b \mid cB \mid d$
$A \rightarrow aA \mid b$
$B \rightarrow cB \mid d$

(iii) $S \rightarrow aAbCC \mid Aba$
$A \rightarrow BaA \mid Cb$
$B \rightarrow bA \mid \in$
$C \rightarrow aBb \mid \in$

(iv) $S \rightarrow abAB \mid AbC$
$A \rightarrow Ba \mid \in$
$B \rightarrow bA \mid Aa \mid \in$

40. Check whether the following grammars are LALR :

(i) $S' \rightarrow S$
$S \rightarrow L = R \mid R$
$L \rightarrow * R \mid id$
$R \rightarrow L$

(ii) $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow F * \mid (E) \mid a \mid b \mid \#$

(iii) $S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
$A \rightarrow C$
$B \rightarrow C$

(iv) $S \rightarrow CC$
$C \rightarrow aC \mid b$

41. Check whether the following grammars are LR (1) :

(i) $S \rightarrow A * B \mid A + B$
$A \rightarrow aA \mid a$
$B \rightarrow Ab \mid b$

(ii) $S \rightarrow Aa \mid bAc \mid Bc \mid bBa$
$A \rightarrow d$
$B \rightarrow c$

(iii) $S \rightarrow 0\ A\ 2$
$A \rightarrow 1\ A\ 1 \mid 1$

42. Construct recursive descent parser for the following grammars :

(i) $S \rightarrow Ab$
$A \rightarrow a \mid B \mid \in$
$B \rightarrow b$

(ii) $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

(iii) $S \rightarrow aAb \mid Sa \mid a$
$A \rightarrow Ab \mid b$

(iv) $S \rightarrow Aab \mid aBb$
$A \rightarrow Aa \mid b$
$B \rightarrow bB \mid b$

43. What is left-factoring? How it is used in recursive descent parser?
44. Write a YACC program for simple calculator which performs operations like 23 ** 2 + 15 – 3/5 + 1.
45. Construct an SLR parsing table for the grammar:
$E \rightarrow E \text{ sub } R \mid E \text{ sup } E \mid \{E\} \mid \in$
$R \rightarrow E \text{ sup } E \mid E$
46. Check following grammar is LALR (1) grammar or not.
$E \rightarrow E + T \mid T$
$T \rightarrow TF \mid F$
$F \rightarrow F * \mid (E) \mid a \mid b \mid \in$
47. Construct recursive descent parser for the following grammar.
(i) $S \rightarrow abSa \mid aaAb \mid b$
$A \rightarrow baAb \mid b$
(ii) $S \rightarrow Aab \mid aBb$
$A \rightarrow Aa \mid b$
$B \rightarrow bB \mid b$
48. Check the following grammars are SLR (1).
(i) $S \rightarrow 0A2$
$A \rightarrow 1A1 \mid 1$
(ii) $S \rightarrow L = R$
$S \rightarrow R$
$L \rightarrow * R$
$L \rightarrow id$
$R \rightarrow L$
(iii) $S \rightarrow A \mid B$
$A \rightarrow aA \mid b$
$B \rightarrow dB \mid b$
49. Explain different types of conflicts occurs in LR parser.

50. Construct LR (0) items for the following production, A → XYZ.
51. Check the following grammar's are LR (1) or not.
    (i) S → E = E | a
        E → E + T | T
        T → T * a | a
    (ii) S → bA | a
        A → CB | cBe
        B → dB
52. Consider the following grammar and parse the string using shift-reduce parsing. Input string is a and * b,
    S → S + C | B
    B → B & C | C
    C → * c | a | b
53. Consider following grammar,
    S → (L) | a
    L → L, s | a
    Find operator precedence relation and parse the (a, (a, a)) using precedence relation.
54. Check whether the following grammar is SLR (1) or not.
    S → A | B
    A → aA | b
    B → dB | b
55. Check whether given grammar is LL (1) or not.
    S → ScB/cA
    A → AaB/B
    B → bB/∈
56. Check whether grammar is SLR (1) or not.
    S → P
    P → bRAe
    A → AmE/E
    R → Rdm/∈
    E → P/q
57. Check whether following grammar is operator precedence or NOT (draw precedence relation table).
    S → S + P/P
    P → P * Q/Q
    Q → (S)/q

58. Construct shift-reduce parser for the CFG:

    S → aAd/aSd

    A → bAcc/bcccc

    string : a 'abbccccccdd'

59. The following grammar is operator precedence grammar. Comment and justify.

    S → SRA | a | b

    A → c | d · R → e|F

60. Check whether the given grammar is LL (1) or not.

    S → a | ^ | (R)

    R → T

    T → S · T | A

61. Check whether the following grammar is SLR or not.

    S → A and A | A or A

    A → (S) | id

62. Check whether the given grammar is LR(1) or not :

    S → aAd

    A → bAb|b

63. Construct Operator Precedence Table for following grammar:

    S → L = R | R

    L → * R | id

    R → L

64. Compute FIRST and FOLLOW of the following grammar :

    S → a ABbCD | ∈

    A → Amd | ∈

    B → SAn | hn | ∈

    C → Sf | Cg

    D → aBD | ∈

65. Construct a Recursive Descent Parser for the following CFG :

    S → aA | bB

    A → aA | b

    B → dB | b

66. Write YACC specification to solve the expression using grammar, E → E + E|id.

67. Construct RDP for the grammar, A → OAO | A 1 | AA | 1.

68. Check whether the given grammar is SLR (1) or not.

    S → Aa|bAc|dc|bda

    A → d

69. Check whether the given grammar is LL (1) or not.

    S → AaAb | BbBa

    A → E

70. Consider the following grammar and input string. Parse the string using shift-reduce parser. Show the contents of stack, input and action at each stage.

    S → TL;

    T → int | float

    L → L, id | id

    Input string → int id, id;

71. Construct a Recursive Descent Parser for the following CFG:

    S → abSa | aaAb | b

    A → b

72. Test whether the given grammar is LL(1) or not and construct a predictive parsing for it.

    S → BD | AB

    A → aAa | b

    B → aAa | ∈ (epsilon)

    D → ∈ (epsilon)

73. Check whether the following grammar is SLR(1) or not.

    S → xAy | xBy | xAz

    A → qS | q

    B → q

74. Consider the following grammar:

    S → A * B | * A

    A → # B | B #

    B → * A | #

    Parse the string # * # # * # using shift-reduce Parser.

75. Check whether the following grammar is LALR or not.

    E → AA

    A → aA | d

76. Find FIRST and FOLLOW to the following grammar:

    E → aA|(E)

    A → + E | * E | e

77. Consider the following grammar:

    S → (L) | a

    L → L, S | S

    Find the operator precedence relation and parse the string (a, (a, a)) using precedence relation.

UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. State True or False: Shift-Shift conflict does not occur in LR parser. **[1 M]**

Ans. Refer to Section 3.8.

2. Write a Recursive Descent Parser (RDP) for the following grammar:

S → Aab | aBb

A → Aa | b

B → bB | b **[5 M]**

Ans. Refer to Section 3.3.

3. Check whether the given grammar is LL(1) or not:

S → BD | AB

A → aAa | b

B → bAa | ε (epsilon)

D → ε (epsilon) **[5 M]**

Ans. Refer to Section 3.4.2.

4. Check whether the given grammar is LALR(1) or not:

S → E = E | a

S → E + T | T

T → T * a | a **[6 M]**

Ans. Refer to Section 3.8.2.

5. Check whether the given grammar is SLR(1) or not :

S → A and A | A or A

A → id | (S) **[6 M]**

Ans. Refer to Section 3.8.2.

6. Give the full form of YACC. **[1 M]**

Ans. Refer to Section 3.9.

October 2016

1. Compute LEADING and TRAILING symbols of the following grammar:

S → (T) | α | ^

T → T, S | S **[1 M]**

Ans. Refer to Section 3.6.3.

2. Check whether the given grammar is LL(1) or not :

S → S # | aA | b | eB | d

A → aA | b

B → eB | d **[5 M]**

Ans. Refer to Section 3.4.2.

3. Write a Recursive Descent Parser (RDP) for the following grammar:

A → aAa | Ab | AA | b **[5 M]**

Ans. Refer to Section 3.3.

4. Check whether the given grammar is SLR(1) or not:

S → bAB | aA

A → Ab | b

B → aB | a **[6 M]**

Ans. Refer to Section 3.8.2.

5. Check whether the given grammar is LALR(1) or not:

E → E + T | T

T → T * F | F

F → F* | (E) | a | b | # **[6 M]**

Ans. Refer to Section 3.8.2.

April 2017

1. Construct LR(0) set f items for the production A → ε. **[1 M]**

Ans. Refer to Section 3.8.2.

2. Write a Recursive Decent Parser (RDP) for the following grammar:

S → aAb | bA

A → Ad | b **[5 M]**

Ans. Refer to Section 3.3.

3. Check whether the given grammar is LL(1) or not:

A → AaB | x

B → BCb | Cy

C → Ce | ε → (epsilon) **[5 M]**

Ans. Refer to Section 3.4.2.

4. Check whether the given grammar is SLR(1) or not:

S → aAB

A → bA | e

B → bB | a **[6 M]**

Ans. Refer to Section 3.8.2.

5. Construct LALR(1) parsing table and check whether the given grammar is LALR(1) or not:

S → AaB | B

A → bB | d

B → A|e **[6 M]**

Ans. Refer to Section 3.8.2.

6. Find out the following grammar is operator precedence grammar or not:

S → a | ^ | (R)

T → S, T | S

R → T

Ans. Refer to Section 3.6.

October 2017

1. Computer FIRST for the following productions:

S → AB | Ad

A → aA| ∈ (epsilon)

B → bB | ∈ (epsilon) **[1 M]**

Ans. Refer to Section 3.4.5.

2. Write a Recursive Descent Parser (RDP for the following grammar:

S → aA*b* | Sa

A → A*b* | *b* **[5 M]**

Ans. Refer to Section 3.3.

3. Check whether the given grammar is LL(1) or not:

S → SeB | eA

A → AaB | B

B → bB | ∈ (epsilon). **[5 M]**

Ans. Refer to Section 3.4.2.

4. Check whether the given grammar is SLR(1) or not:

S → aAb | aBb | aAd

A → dS | d

B → e **[6 M]**

Ans. Refer to Section 3.8.2.

5. Check whether the given grammar is LALR(1) or not:

S → <L> | a

L → L, S | S **[6 M]**

Ans. Refer to Section 3.8.2.

6. Which one is the most powerful parser in Bottom up parsers? **[1 M]**

Ans. Refer to Section 3.5.

April 2018

1. List the different types of conflicts that occur in LR parser. **[1 M]**

Ans. Refer to Section 3.8.

2. Check whether the given grammar is LL(1) or not:

S → A
A → aA | Ad
B → bBC|f
C → g **[5 M]**

Ans. Refer to Section 3.4.2.

3. Consider the following grammar and parse the input string "211221" using shift-reduce parser. Show the contents of stack, input and action taken at each stage:

S → 1B | 2A
A → 1 | 1S | 1AA
B → 2 | 2S | 1BB **[5 M]**

Ans. Refer to Section 3.7.

4. Check whether the given grammar is LR(1) or not:

S → A | ab
A → aAb | B
B → a **[6 M]**

Ans. Refer to Section 3.8.

5. Write a Recursive Descent Parser (RDP) for the following grammar:

S → AB
A → Aa | a
B → Bb|b **[4 M]**

Ans. Refer to Section 3.3.

6. Check whether the given grammar is SLR(1) or not:

N → V=E|E
E → V
V → a | *E **[6 M]**

Ans. Refer to Section 3.8.2.

October 2018

1. Which parser is most powerful parser in Bottom-up parser? **[1 M]**

Ans. Refer to Section 3.5.

2. Name the conflict which is not possible in LR parser. **[1 M]**

Ans. Refer to Section 3.8.

3. What is handle pruning? **[1 M]**

Ans. Refer to Section 3.7.1.

4. Define the term token. **[1 M]**

Ans. Refer to Section 3.1.

5. Check whether the given grammar is LL(1) or not:

$S \rightarrow aAB \mid$

$A \rightarrow Aa \mid b$

$B \rightarrow bB \mid \varepsilon$ **[5 M]**

Ans. Refer to Section 3.4.2.

6. What is an operator grammar? Consider the following grammar:

$S \rightarrow (T) \mid a \mid \wedge$

$T \rightarrow T, S$ **[5 M]**

Ans. Refer to Section 3.6.

7. Consider the following grammar and parse the input string "abbb#" using shift-reduce parser. Show the contents of stack, input and action taken at each stage:

$S \rightarrow R\#$

$R \rightarrow P \mid Rb \mid b$

$P \rightarrow a$ **[5 M]**

Ans. Refer to Section 3.7.

8. Check whether the given grammar is LR(1) or not:

$S \rightarrow Aa \mid bAc$

$T \rightarrow d$ **[6 M]**

Ans. Refer to Section 3.4.2.

9. Write a Recursive Descent Parser (RDP) for the following grammar:

$S \rightarrow Aa \mid bAc$

$A \rightarrow aA \mid b$

$B \rightarrow dB \mid b$ **[4 M]**

Ans. Refer to Section 3.3.

10. Check whether the given grammar is SLR(1) or not:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow a$ **[6 M]**

Ans. Refer to Section 3.8.2.

April 2019

1. State True or False. Bottom-up parsing uses the process of derivation. **[1 M]**

Ans. Refer to Section 3.5.

2. Define the term handle. **[1 M]**

Ans. Refer to Section 3.7.1.

3. Check whether the given grammar is LL(1) or not.

$S \rightarrow aAB \mid c$

$A \rightarrow Ba \mid d$

$B \rightarrow bA \mid Ab \mid \varepsilon$ **[5 M]**

Ans. Refer to Section 3.4.2.

4. Write a Recursive Descent Parser (RDP) for the following grammar.

$S \rightarrow aA \mid SbB$

$A \rightarrow aA \mid bB$

$B \rightarrow b$ **[5 M]**

Ans. Refer to Section 3.3.

5. Consider the following operator grammar, $S \rightarrow S+S \mid S*S \mid d$.

(i) Construct the operator precedence relation table.

(ii) Construct the precedence function table. **[5 M]**

Ans. Refer to Section 3.6.

6. Check whether the given grammar is SLR(1) or not.

$S \rightarrow bAB \mid aA$

$A \rightarrow Ab \mid b$

$B \rightarrow aB \mid a$ **[6 M]**

Ans. Refer to Section 3.8.2.

7. Check whether the given grammar is canonical LR(1) or not.

$S \rightarrow bcS \mid SbA \mid a$

$A \rightarrow d$ **[6 M]**

Ans. Refer to Section 3.4.2.

CHAPTER 4

Syntax Directed Definition

Objectives...

- To learn Basic Concepts of Syntax Directed Definition (SDD)
- To understand the Syntax Directed Translation (SDT)
- To understand Applications of SDT

4.0 INTRODUCTION

- The output of Lexical analysis phase is token. Parser checks the syntax of the language.
- Besides specifying the syntax of a language a context-free grammar can be used to help guide the translation of program.
- A Syntax Directed Translation (SDT) specifies the translation of a construct in terms of attributes associated with its syntactic components.
- The SDT is a commonly used notation for specifying attributes and semantic rules along with the context-free grammar.
- A Syntax Directed Definition (SSD) is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol.
- In this chapter, we introduce a grammar oriented compiling technique known as syntax-directed translation also introduce semantic of the languages that is type checking.
- For simplify, we consider the syntax-directed translation of infix expressions to postfix form and build syntax trees for programming constructs.

4.1 SYNTAX DIRECTED TRANSLATION (SDT) [April 16, 17]

- Syntax-directed translation fundamentally works by adding actions to the productions in a context-free grammar, resulting in a Syntax-Directed Definition (SDD).
- Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser.

- The main idea behind syntax-directed translation is that the semantics or the meaning of the program is closely tied to its syntax.
- Most of the modern compiled languages exhibit this property. Syntax-directed translation involves:
  1. Identifying attributes of the grammar symbols in the context-free grammar.
  2. Specifying semantic rules or attributes equations relating the attributes and associates them with the productions.
  3. Evaluating semantic rules to cause valuable side-effects like insertion of information into the symbol table, semantic checking, issuing of an error message, generation of intermediate code, and so on.
- An attribute is any property of a symbol. For example, the data type of a variable is an attribute.
- A CFG in which a program fragment called output action (semantic action or semantic rule) is associated with each production is known as Syntax Directed Translation (SDT).
- In short, SDD given as follows:

  Attributes + CFG + Semantic rules = Syntax Directed Definition (SDD).
- The principle of Syntax Directed Translation (SDT) states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse tree, (see Fig. 4.1).

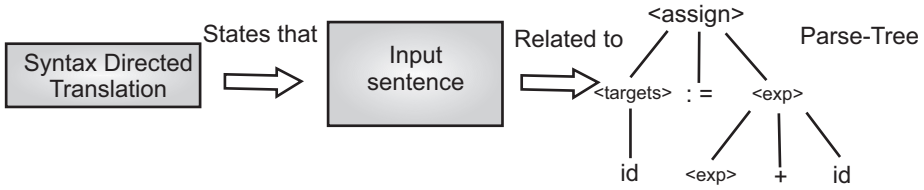


Fig. 4.1: Concept of SDT

- Syntax-directed translation is done by attaching rules to productions in a grammar. Consider an expression represented by production rule,

  $\texttt{expr} \rightarrow \texttt{expr}_1 \texttt{ + term}$

  Here, expr is the sum of two sub-expressions $expr_1$ and term.
- We translate this into pseudo-code as:

```
translate expr1;
translate term;
handle +;
```

- Then build a syntax-tree for expr. and then compute the values of attributes at the node of the tree by visiting the nodes of the tree.
- The above production is written as:

Production	Semantic rule
$expr \rightarrow expr_1 + term$	$expr \cdot code = expr_1 \cdot code \mid\mid term \cdot code \mid\mid '+'$

- The semantic rule specifies that the string expr.code is formed by concatenating $expr_1$.code, term.code and character '+'.
  - In syntax-Directed translation grammar, symbols are associated with attributes to associate information with the programming language constructs that they represent.
  - Values of these attributes are evaluated by the semantic rules associated with the production rules.
  - Evaluation of these semantic rules:
    - generate intermediate codes.
    - generates error messages.
    - put information into the symbol table.
    - perform type checking.
    - perform almost any activities above.
  - An attribute may hold almost any thing.
    - a string, a number, a memory location, a complex record, table references.
- Sometimes, translation can be done during parsing. Therefore, we study a class of syntax directed translations called "L-attribute translations" (L for left-to-right) in which translation can be performed during passing.
- A smaller class called "S-attributed translations" (S for synthesized, which can performed with bottom-up parse.)

Form of a Syntax-Directed Definition (SDD):

- In a syntax-directed definition, each grammar production $A \rightarrow a$ is associated with a set of semantic rules of the form $b = f(C_1, C_2, ..., C_k)$, where f is a function and either.
  - b is a synthesized attribute of A and $C_1, C_2, ..., C_k$, are attributes belonging to the grammar symbols of the production, or
  - b is an inherited attribute of one of the grammar symbols on the right side of the production and $C_1, C_2, ... C_k$, are attributes belonging to the grammar symbols of the production.
- In either case, we say that attribute and depends on the attributes $C_1, C_2, ... C_k$. An attribute grammar is a syntax directed definition in which the functions in semantic rules cannot have side effects.
- Syntax-Directed Translation (SDT) is an extension of Context-Free Grammar (CFG) which acts as a notational framework for the generation of intermediate code.
- A parse tree showing the values of attributes at each node is called an annotated parse tree. **[April 16, 17, 19]**
- The process of computing the attributes values at the nodes is called annotating or decorating of the parse tree.

- An annotated parse tree for the input string 3 * 5 + 4 n is shown in Fig. 4.2.

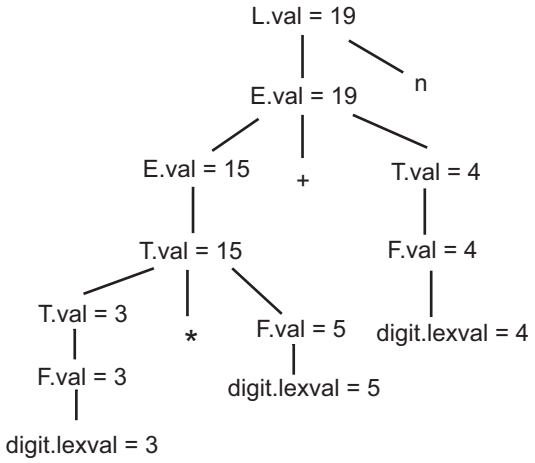


Fig. 4.2

4.2 SYNTAX DIRECTED DEFINITIONS (SDD) [April 18, Oct. 16]

- A context-free grammar in which the productions are shown along with its associated semantic rules is called as a syntax-directed definition.
- While grammars and automata describe the structure or syntax of strings in the language, something more is needed to describe the meaning or semantics of those strings.
- One approach for defining semantics is to extend a CFG with additional semantic rules that describe operations or actions to take at certain points within each production.
- Additional values computed by a semantic rule may be stored as attributes, which are parameters associated with each non-terminal or token, and then reused in other semantic rules.
- The resulting combination of a CFG with additional semantic rules is called a Syntax-Directed Definition (SDD).
- A SDD is a context-free grammar together with attributes and rules, where attributes are associated with grammar symbols and rules are associated with production.
- If S is a symbol and a is one of its attributes then we write S.a which is value of a at a particular node of tree labeled S.
- When we associate semantic rules with productions, we use two notations i.e., Syntax-Directed Definitions and Translation Schemes.

1. Syntax Directed Definitions (SDD):

- give high level specifications for translations.
- hide many implementation details such as order of evaluation of semantic actions.

- we associate a production rule with a set of semantic actions and we do not say when they will be evaluated.

2. Translation Schemes:

- indicate the order of evaluation of semantic actions associated with a production rule.
- in other words translation schemes give or little bit information about implementation details.

Attribute Grammar: **[April 16]**

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) is appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

  Example,

```
E → E + T { E.value = E.value + T.value }
```

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.
- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Based on the way the attributes get their values, they can be broadly divided into two categories namely, synthesized attributes and inherited attributes.

4.2.1 Inherited and Synthesized Attributes [Oct. 16, 17, April 18]

- A SDD is a generalization of a CFG in which each grammar symbol is associated with a set of attributes.
- There are two types of set of attributes for a grammar symbol namely, Synthesized attributes and Inherited attributes.

1. Synthesized Attribute: **[Oct. 16]**

- A synthesized attributes for a non-terminal X at a parse-tree node N is defined by a semantic rule associated with the production at N.
- A synthesized attribute of node N is attribute values at the children of N and N itself.
- For example, for production $E \rightarrow E_1 + T$, the semantic rules, $E.val \rightarrow E_1 \cdot val + T \cdot val$. Symbol E is associated with a synthesized attribute val.

2. Inherited Attribute: [Oct. 17]

- An inherited attribute for a non-terminal Y at a parse tree node N is defined by a semantic rule associated with the production at the parent of N.
- An inherited attribute of node N is attribute values of N's parent, N itself and N's siblings.
- Terminals can have synthesized attributes, but terminals can not have inherited attributes.
- For terminal symbols there are no semantic rules for computing the value of an attribute; they have lexical values supplied by lexical analyzer.

Example 1: Consider an example of grammar of expression. Fig. 4.3 shows the SDD of the grammar.

$L \rightarrow E$

$E \rightarrow E_1 + T \mid T$

$T \rightarrow T_1 * F \mid F$

$F \rightarrow (E) \mid digit$

Production	Semantics Rules
$L \rightarrow E$ return	Print (E.val) or L.val = E·val
$E \rightarrow E_1 + T$	$E{\cdot}val = E_1{\cdot}val + T{\cdot}val$
$E \rightarrow T$	$E{\cdot}val = T{\cdot}val$
$T \rightarrow T_1 * F$	$T{\cdot}val = T_1{\cdot}val * F{\cdot}val$
$T \rightarrow F$	$T{\cdot}val = F{\cdot}val$
$F \rightarrow (E)$	$F{\cdot}val = E{\cdot}val$
$F \rightarrow digit$	$F{\cdot}val = digit{\cdot}lexval$

Fig. 4.3: SDD of the Grammar

- Symbols E, T and F are associated with synthesized attribute val.
- The token digit has a synthesized attribute lexval (it is assumed that it is evaluated by the lexical analyzer).
- The production $L \rightarrow E$ return, where return is a endmarker, sets L.val to E.val and produces the result of the entire expression.
- For production $E \rightarrow E_1 + T$, the val for the E is the sum of the values of E_1 and T (all its children).
- For production $E \rightarrow T$, the val. for the E is same as val at the child for T.

S-attributed SDD: [Oct. 18]

- An SDD that involves only synthesized attributes is called S-attributed. The above (Fig. 4.3) SDD is S-attributed.

- A S-attributed SDD can be implemented in conjunction with LR parser. A SDD without side effects is also called an attribute grammar.
- Here, in rule L·val = E·val, the value of E·val is assign to L-value i.e. address. Hence, there is no side effect.

Differentiation between Synthesized Attributes and Inherited Attributes:

Sr. No.	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The value of the translation of the non-terminal on the left side of the production as a function of translation of non-terminal on the right-hand side (RHS) is called a synthesized attribute (translation).	Translation of a non-terminal on right-hand side of a production is determined in terms of a non-terminal on the left-hand which side is called an inherited attribute.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	Synthesized attributes pass on information up the parse tree.	Inherited attributes pass on information down the parse tree.
5.	Synthesized attributes can be contained by both the terminals and non-terminals.	Inherited attributes can't be contained by both but it is only contained by non-terminals.
6.	S-attributes are also called reference attributes (call by reference).	Inherited attributes are called value attributes (call by value).
7.	**Example:** $E \rightarrow E_1 + E_2$ (E.val = E_1.val + E_2.val) Semantic action enclosed in parentheses states that translation in Left-Hand Side (LHS) of production is determined by adding together translations associated on RHS of production.	**Example:** $A \rightarrow XYZ$ (Y.val = 2*A.val) States that the translation in RHS is determined by translation associated with LHS of production.

4.2.2 Evaluating an SDD at the Nodes of a Parse Tree

[Oct. 16, April 17, 18, 19]

- An SDD is a context-free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules are associated with productions.
- To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree.
- Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.
- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.
- In this section, we would like to discuss how all the attributes at each node of the pause tree are evaluated.
- A parse-tree, with values of its attributes at each node is called **annotated parse tree**.
- **To evaluate an SDD:**
  - Construct annotated parse tree.
  - Attributes at a node of a parse tree are evaluated.
  - The evaluation order is depends upon whether the attribute is synthesised attribute or inherited attribute.
  - In synthesize attribute, we can evaluate in bottom-up order (preorder traversal) i.e. children first.
  - In inherited attribute, we can evaluate in top down order i.e. parent first. (preorder traversal).
  - The SDD having both synthesized and inherited attributes, there is no guarantee of any order to evaluate an attributes at nodes.

Example 2: Consider non-terminals X and Y with synthesized and inherited attributes X.s and Y.i respectively.

Production	Semantics Rules
$X \rightarrow Y$	$X \cdot s \rightarrow Y \cdot i$ $Y \cdot i \rightarrow X \cdot s + 1$

- It is not possible to evaluate either $X \cdot s$ at node N or $Y \cdot i$ at the child of N without first evaluating the other.
- This is circular dependency of $X \cdot s$ and $Y \cdot i$, which is shown in Fig. 4.4.

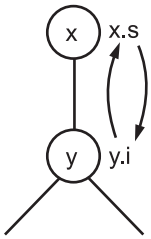


Fig. 4.4: Circular Dependency of Attributes

- However, circular dependency attributes can be evaluated by some useful subclasses of SDD.

Example 3: Consider the grammar of rules of Fig. 4.5 and construct annotated parse tree for the input string 5 + 3 * 4 return.

This is synthesized attribute SDD.

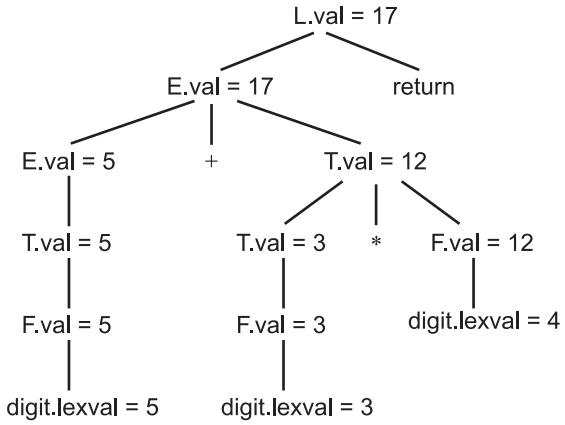


Fig. 4.5: Annotated Parse Tree for 5 + 3 * 4

Here, each non-terminal attribute **val** is computed in a bottom-up order i.e. children attributes are computed first and then apply rule to the parent node. Consider the node for production $F \rightarrow T * F$. The value of attribute T.val is defined by,

Production	Semantic Rules
$T \rightarrow T_1 * F$	$T \cdot val = T_1 \cdot val * F \cdot val$

Example 4: Construct annotated parse tree for input 3 * 5 * 2.

Solution: This is both inherited attribute and synthesized attribute SDD. This uses top-down parser.

$\therefore$ For this input string we use non-left recursive grammar for expression.

Grammar is, $E \rightarrow T\in'$

$E' \rightarrow + T\in' \mid \in$

$T \rightarrow FT'$

$$T' \rightarrow * FT' \mid \in$$

$$F \rightarrow \text{digit}$$

The parse tree for input string $4 * 5 * 3$ is,

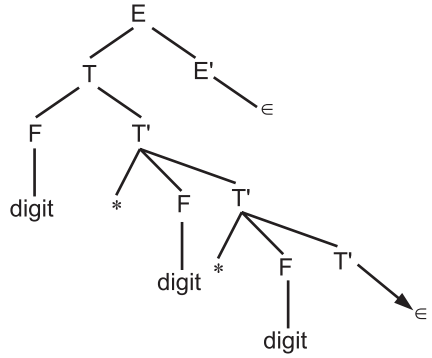


Fig. 4.6: Parse Tree

The Fig. 4.7 shows the SDD based on above grammar suitable for top-down parsing.

Production	Semantic Rules
1. $E \rightarrow TE'$	$E' \cdot inh = T \cdot val$ $E \cdot val = E' \cdot syn$
2. $E' \rightarrow + TE_1'$	$E_1' \cdot inh = E' \cdot inh + T \cdot val$ $E' \cdot syn = E_1' \cdot syn$
3. $E' \rightarrow \in$	$E' \cdot syn = E' \cdot inh$
4. $T \rightarrow FT'$	$T' \cdot inh = F \cdot val$ $T \cdot val = T' \cdot syn$
5. $T' \rightarrow * FT_1'$	$T_1' \cdot inh = T' \cdot inh \times F \cdot val$ $T' \cdot syn = T_1' \cdot syn$
6. $T' \rightarrow \in$	$T' \cdot syn = T' \cdot inh$
7. $F \rightarrow \text{digit}$	$F \cdot val = \text{digit} \cdot lexval$

Fig. 4.7: SDD with Inherited Attributes

- The non-terminals E, T and F have a synthesized attribute val and terminal digit has a synthesized attribute lexval.
- The non-terminals E' and T' has two attributes an inherited attribute inh and a synthesized attribute syn.

- Consider rule $T' \rightarrow * FT_1'$, the head T' inherits the left operand of * in the production. For example: In a * b * c the root of the subtree for * b * c inherits a. Then root of subtree for * c inherits a * b value and so on. Fig. 4.8 shows the annotated parse tree for 4 * 5 * 3.

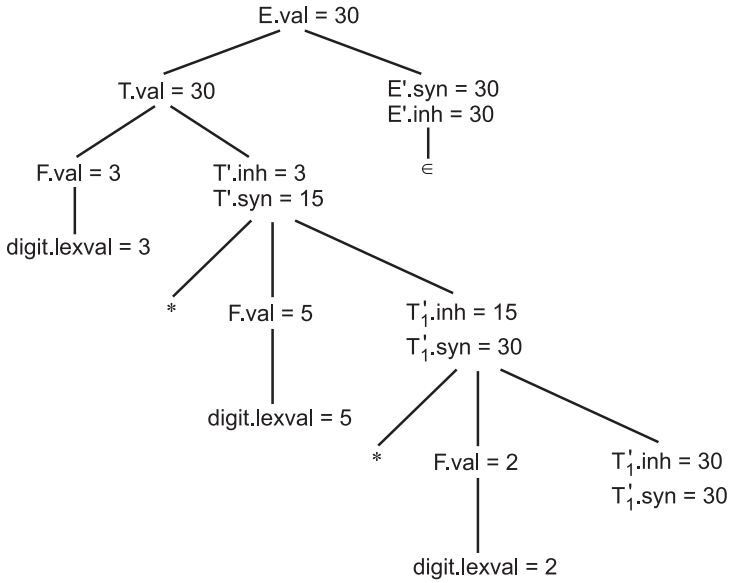


Fig. 4.8: Annotated Parse tree for 3 * 5 * 2

Consider left subtree of 3 * 5.

- Here, F·val = 3 since lexical analyzer supplied the value 3. The second child of T is the inherited attribute T'·inh is defined by semantic rule T'·inh = F·val ∴ T'·inh = 3.
- For production $T' \rightarrow * FT_1'$. The inherited attribute T_1'·inh is defined by semantic rule.

$$T_1'\cdot inh = T_1' inch \times F\cdot val$$

∴ $$T_1'\cdot inh = 3 * 5 = 15$$

- For production $E' \rightarrow \in$. The semantic rule is E'·syn = E'·inh = 30.

Example 5: Consider the grammar:

$$D \rightarrow TL$$

$$T \rightarrow int \mid real$$

$$L \rightarrow L, id \mid id$$

Construct SDD and annotated parse tree for this inherited attribute grammar with following semantic rules.

Production	Semantic rules
1. $D \rightarrow TL$	L·inh = T·val
2. $T \rightarrow int$	T·val = int
3. $T \rightarrow real$	T·val = real
4. $L \rightarrow L_1$, id	L_1·inh = L·inh
	L·val = L_1·inh, id·val
5. $L \rightarrow id$	L·val = id·val

Fig. 4.9: SDD with Inherited Attribute L · inh

Solution: The Fig. 4.10 shows the annotated parse tree for input string real id_1, id_2, id_3. For example, real a, b, c. This is an example for type declaration of any language. The value of L·inh at the three L-nodes gives value of the identifiers id_1, id_2 and id_3. There values are determined by computing the value of the attribute T·val at the left child of the root and then evaluated L·inh top-down at the three L-nodes in the right subtree of the root.

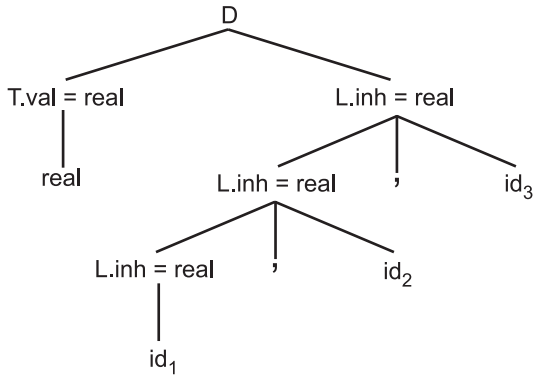


Fig. 4.10: Annotated Parse Tree with Inherited Attribute for Input String Real id_1, id_2, id_3

4.3 EVALUATION ORDERS FOR SDD's

- Dependency graphs are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.
- There are two classes of SDD's, which we will discuss in this section.
  1. S-attributed, and
  2. L-attributed.

4.3.1 Dependency Graph [April 16, 17]

- The inter-dependencies among the inherited and synthesized attributes at the nodes in a parse tree can be shown by a directed graph called a dependency graph.

- A dependency graph depicts the flow of information among the attribute in-stances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.
- It is useful and customary to depict the data flow in a node for a given production rule by a simple diagram called a dependency graph.
- A dependency graph represents the flow of information between the attribute instances in a parse tree.
- The inter-dependencies among the attributes at the nodes in a parse tree can be shown by a dependency graph.
- It is used to depict the inter-dependencies among the inherited and synthesized attributes at the nodes in a parse tree.

Construction of Dependency Graph:

1. Put each semantic rule into the form b: f (c_1, c_2, … c_k), by introducing a dummy synthesized attribute b.
2. The graph has a node for each attribute associated with a grammar symbol and an edge to the node for b from node for c if attribute b depends on attribute c.
3. For each attribute a of the grammar symbol at n construct a node in the dependency graph for a.
4. For each semantic rule b: f (c_1, c_2 … c_k) construct an edge from node for c_i to the node for b $\forall 1 < i \leq k$.

- For example, suppose $A \rightarrow XY$ is a production with semantic rule $A \cdot a := f(X \cdot x, Y \cdot y)$. This rule defines a synthesized attribute $A \cdot a$ that depends on the attribute $X \cdot x$ and $Y \cdot y$.
- For this production, we have 3 nodes in the dependency graph and edges are:
  1. $A \cdot a$ from $X \cdot x$ ($\because A \cdot a$ depends on $X \cdot x$)
  2. $A \cdot a$ from $Y \cdot y$ ($\because A \cdot a$ depends on $Y \cdot y$).
- If we consider inherited attribute, then the production $A \rightarrow XY$ has a semantic rule

$$X \cdot i = g(A \cdot a, Y \cdot y)$$

then the edges in the dependency graph are:

- $X \cdot i$ from $A \cdot a$
- $X \cdot i$ from $Y \cdot y$

} Here $X \cdot i$ depends on $A \cdot a$ and $Y \cdot y$

Example 6: Consider $E \rightarrow E + E$; the dependency graph is shown in Fig. 4.11.

Production	Semantic Rules
$E \rightarrow E_1 + E_2$	$E \cdot val = E_1 \cdot val + E_2 \cdot val$ (Synthesized attribute)

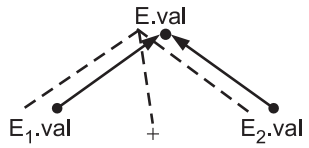


Fig. 4.11: The Dependency Graph

Example 7: Construct dependency graph for input 5 + 3 * 4 for grammar of Fig. 4.12.

Solution:

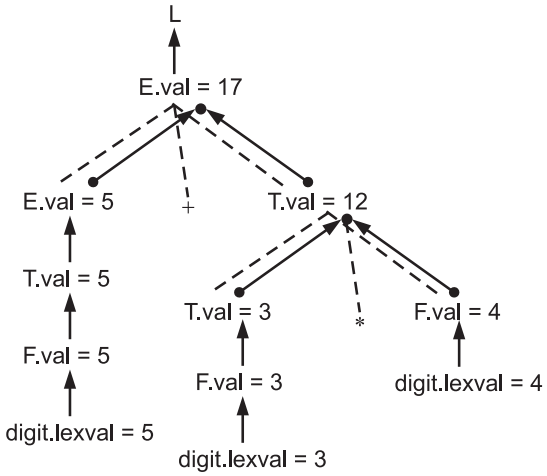


Fig. 4.12: Dependency Graph for Input 5 + 3 * 4

Example 8: Consider the grammar:

$$D \rightarrow TL$$
$$T \rightarrow \text{int} \mid \text{real}$$
$$L \rightarrow L, \text{id} \mid \text{id}$$

Construct dependency graph.

Solution:

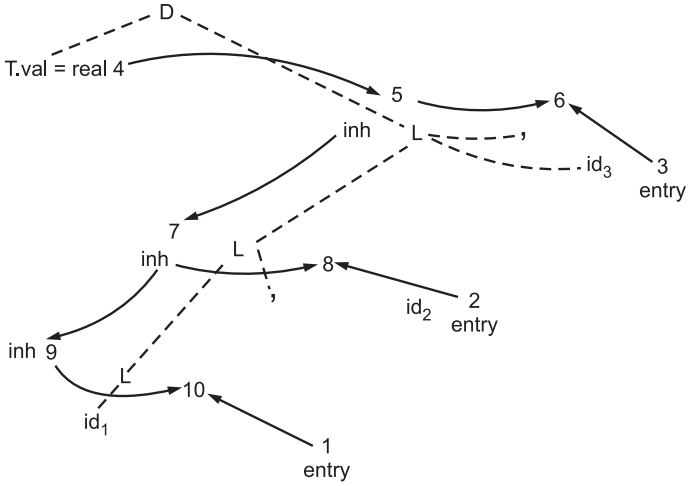


Fig. 4.13: Dependency Graph

- The number shows the node of the dependency graph.
- There is an edge to node 5 for L·inh from node 4 for T·val because inherited attribute L·inh depends on T·val; by semantic rule.

  $L \cdot inh = T \cdot val$ for production $D \rightarrow TL$
- Consider the production $L \rightarrow L, id$

  The semantic rules are

  $L_1 \cdot inh = L \cdot inh$

  $L \cdot val = L_1 \cdot inh, id \cdot val$

  The node 7 has two downward edges to node 8 and 9 because $L_1 \cdot inh$ depends on L·inh and each id is associated with L.
- The nodes 6, 8 and 10 are constructed for id_1, id_2 and id_3.

Example 9: Construct dependency graph for the productions:

$T \rightarrow FT'$

$T' \rightarrow * FT'$

$T' \rightarrow \in$

$F \rightarrow digit.$

Solution:

Production	Semantic rules
1. $T \rightarrow FT'$	$T' \cdot inh = F \cdot val$ $T \cdot val = T' \cdot syn$
2. $T' \rightarrow * FT'$	$T_1' \cdot inh = T' \cdot inh \times F \cdot val$ $T' \cdot syn = T_1' \cdot syn$
3. $T \rightarrow \in$	$T' \cdot syn = T' \cdot inh$
4. $F \rightarrow digit$	$F \cdot val = digit \cdot lexval$

The dependency graph is shown in Fig. 4.14 for input string digit * digit.

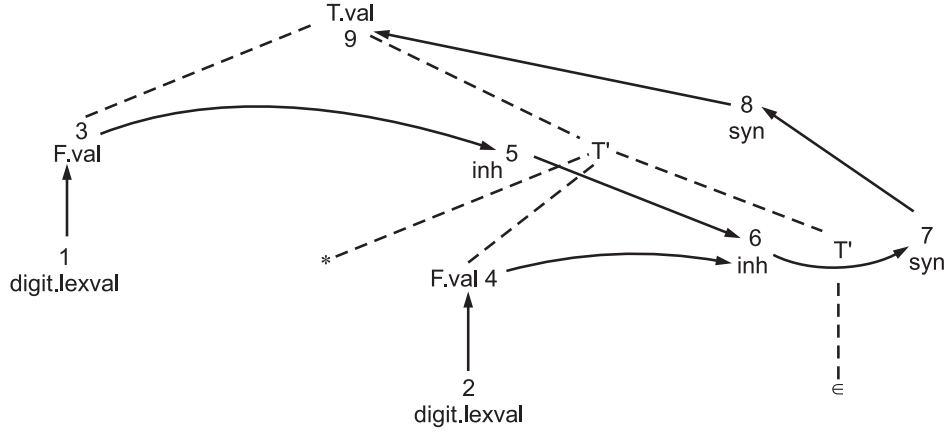


Fig. 4.14: The Dependency Graph for Input String Digit * Digit

4.3.2 Ordering the Evaluation of Attributes

- The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. Several methods are used for evaluating semantic rules.

1. **Parse-tree Methods:**

- This is called topological sort of the graph.
- If the dependency graph has an edge from node X to node Y, then the attribute corresponding to X must be evaluated before attribute of Y.
- These methods will fail to find an evaluation order only if the dependency graph as a cycle.
- If there is no cycle, then there is always one topological sort.

2. **Rule-based Methods:**

- The semantic rules associated with productions are analysed by special tool during compiler construction time.

3. **Oblivious Methods:**

- An evaluation order is chosen without considering the semantic rules and order of evaluation is forced by parsing method.
- Rule-based and oblivious methods are more efficient in terms of compile time and space because they need not construct dependency graph at compile-time while parse-tree methods constructs dependency graph at compile time.
- For example: The dependency graph of Fig. 4.14 is constructed using topological sort, it has no cycles and the topological order is 1 3 5 2 4 6 7 8 9.

4.3.3 S-Attributed Definition

- The SDD is S-attributed if every attribute is synthesized. A syntax-directed translation is called S-attributed if all its attributes are synthesized.
- For S-attributed SDD, the attributes are evaluated in bottom-up order of the nodes of the parse tree.
- The attributes are evaluated by using postorder traversal of the parse tree.
- Since, bottom-up parsing uses postorder traversal, S-attributed definitions can be implemented during bottom-up parsing or LR parsing.
- Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed.
- The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack.

4.3.4 L-Attributed Definition

- L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- The idea behind L-Attributed Definitions, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed").
- The classes of syntax-directed definitions whose attributes can always be evaluated in depth-first order are called L-Attributed Definitions.
- The L in the L-Attributed Definitions stands for left because attribute information appears to flow from left to right.)

Example 10: Consider the grammar of Example 8. The SDD is L-attributed because following two productions are using inherited attributes and remaining attributes are synthesized. The inherited attribute productions are:

Production	Semantic Rule
1. $T \rightarrow FT'$	$T'.inh = F.val$
2. $T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$

Here, in the first production rule, inherited attribute $T' \cdot inh$ is define using only F·val and F appears as the left of T' in the production rule. In the second production rule, $T''_1.inh$ is define using T'·inh attribute and F·val·, F appears as the left of T'_1 in the production rule.

Example 11: Consider the grammar with production

$$A \rightarrow XY.$$

Solution:

Production	Semantic rule
$A \rightarrow XY$	$A.syn = X.val$ $X.inh = f(Y.val, A.syn)$

The first semantic rule A·syn = X·val, is either S-attributed or L-attributed SDD. It defines synthesized attribute A·syn in terms of attribute an child. (X contain in production body).

The second rule defines an inherited attribute X·inh, so SDD cannot be S-attributed.

Also SDD cannot be L-attributed because the attribute Y·val is used to help X·inh and Y is the right of X in the production body.

Any SDD containing the above production is neither S-attributed nor L-attributed.

- **L-Attributed Definition:** A SDD is L-attributed if each inherited attribute of X_i, $1 \leq i \leq n$, on the right side of $A \rightarrow X_1, X_2, \ldots, X_n$, depends only on,
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{i-1}$ to the left of X_i in the production rule and
  2. the inherited attributes of A.

  Note: Every S-attributed definition is L-attributed because rule 1 and 2 apply only to inherited attributes.

Example 12: Consider the following SDD.

Production	Semantic rules
1. $A \rightarrow LM$	$L \cdot inh = I(A \cdot inh)$ $M \cdot inh = m(L \cdot syn)$ $A \cdot syn = f(M \cdot syn)$
2. $A \rightarrow QR$	$R \cdot inh = r(A \cdot inh)$ $Q \cdot inh = q(R \cdot syn)$ $A \cdot syn = f(Q \cdot syn)$

In this SDD the inherited attribute $Q \cdot inh$ of the grammar symbol Q depends on its right side. Therefore, this SDD is not L-attributed.

4.3.5 Semantic Rules with Controlled Side Effects

- Translations can involve side effects such as:
  1. Printing a result with desk calculator (grammar of expression) instead of saving it in synthesized attribute.
  2. Code generator might enter the type of an identifier into a symbol table.
  3. Updating global variable.
- Side effects in SDD's can be controlled in one of the following ways:
  1. Permit side effects when attribute evaluation based on any topological sort of the dependency graph.
  2. The translation is produced in any allowable order.
- Consider grammar of expression or desk calculator with production rule:

Production	Semantic rule
$L \rightarrow E$ return	print (E·val)

Here, instead of semantic rule L·val = E·val, the desk calculator print the result and not saving result in synthesized attribute.

Note: Semantic rules that are executed with side effects are treated as the definition of dummy synthesized attributes associated with the head of the production. (Here L is the head).

Example 13: Consider the following SDD:

Production	Semantic Rules
1. $D \rightarrow TL$	L·inh = T·type
2. $T \rightarrow int$	T·type = integer
3. $T \rightarrow real$	T·type = real
4. $L \rightarrow L_1, id$	L_1·inh = L·inh addtype (id·entry, L·inh)
5. $L \rightarrow id$	addtype (id·entry, L·inh)

This SDD is for type declaration.

For example, int a, b, c

real x, y.

- T·type is a attribute of T which is the type in the declaration D.
- L has inherited attribute, to pass the declare type down the list of identifier.
- Production 2 and 3 are having synthesized attribute T·type for integer or real value.
- This type is passed to L·inh of production 1.
- In production 4, the value of L_1·inh is computed by copying the value of L·inh from the parent of that node.
- The function addtype is called with 2 arguments id·entry, which is lexical value point to the symbol table and L·inh is the type assign to every identifier.

The dependency graph is shown in Fig. 4.15 for input real id_1, id_2, id_3.

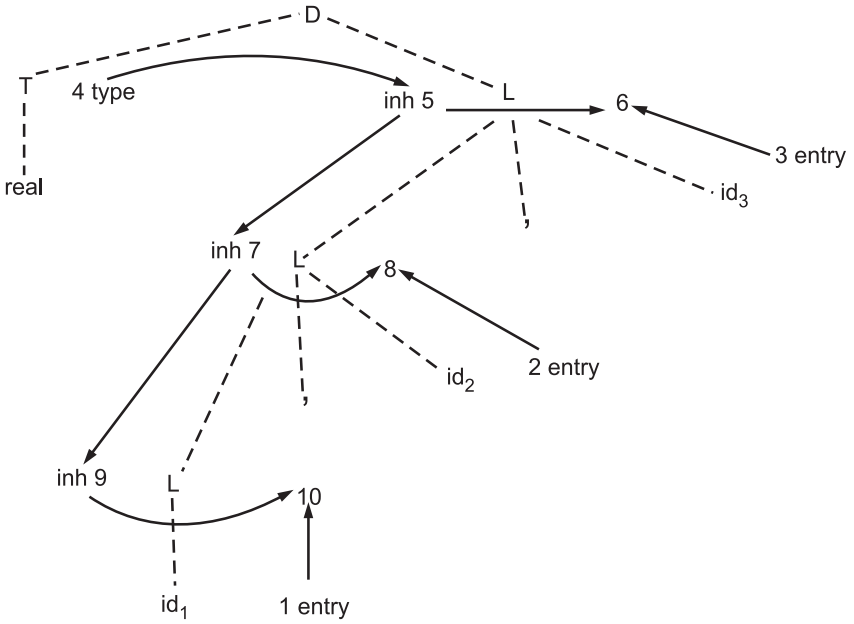


Fig. 4.15: Dependency for Real id_1, id_2, id_3

- In this SDD, notes 6, 8 and 10 are the dummy attributes represents the function addtype to evaluate a type.
- For each identifier on the list, the type is entered into the symbol table entry for the identifier. So entering the type for one identifier does not affect the symbol table entry for any other identifier.

  So entries are updated in any order, which controls the side effects.

Example 14: Construct annotated parse tree for the expressions, int a, b, c for the SDD.

Solution:

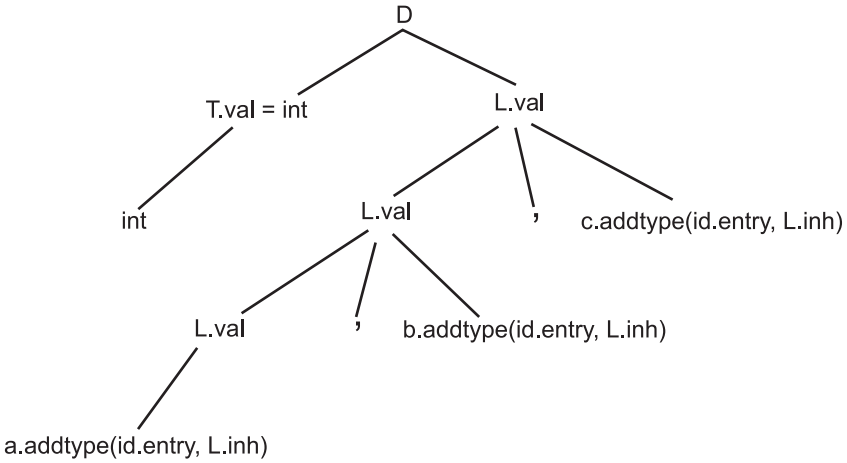


Fig. 4.16: Annotated Parse Tree

4.4 APPLICATIONS OF SDT

- The main application of syntax-directed translation is construction of syntax trees. The use of syntax trees as an intermediate representation in many compilers.
- For any nesting loop, the tree representation is easier. Parsing becomes easy so C compiler usually construct syntax tree for declarations.

4.4.1 Construction of Syntax Trees

- Syntax tree is nothing but a condensed form of a parse tree in which the operator and keyword modes of the parse tree are moved to their parents and a chain of single production is replaced by a single link.
- **Definition:** A syntax tree is a condensed form of parse tree which is useful for representing language constructs.
- For example, Syntax tree for $E \rightarrow E+ E$ is,

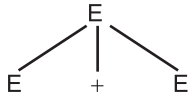


Fig. 4.17: Syntax Tree

- In syntax tree, operators and keywords do not appear as leaves, they are interior nodes.
- Syntax-directed translation can be based on syntax trees. The syntax tree for expression, 3 + 4 * 5 is shown in Fig. 4.18.

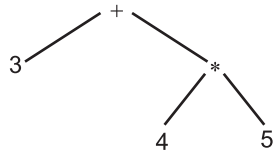


Fig. 4.18: Syntax Trees for Expressions

- The construction of syntax tree for an expression is similar to the translation of an expression into postfix form.
- By creating a node for each operator and operand, construct the subtrees. Each node in a syntax tree is a record of several fields.
- If the node is operator, then the fields are: operator and pointers to the node for the operands. The operator is the label of the node.
- During translation syntax tree nodes may have additional values of attributes attached to the node.
- The following are the functions used to create the nodes of binary tree for expression:
  1. **New code (op, left, right):** It creates an operator node with label op and two pointers left and right.
  2. **New leaf (id, entry):** It creates an identifier node with label id and a field containing entry, a pointer to symbol-table for identifier.
  3. **New leaf (num, val):** It creates a number node with label num and a field containing val value of the number.

Example 15: Consider a sequence of following function calls to create a syntax tree for expression a + 5 – b.

Solution: Let P_1, P_2, …, P_5 are pointers to nodes and entrya and entryb are pointers to the symbol-table for identifiers a and b respectively.

1. P_1 = New leaf (id, entrya);
2. P_2 = New leaf (num, 4);
3. P_3 = New node ('+', P_1, P_2);
4. P_4 = New leaf (id, entryb);
5. P_5 = New node ('–', P_3, P_4).

} Steps in construction of syntax tree

The tree is constructed as bottom-up. The function calls first and second constructs a leaves for a and 5 and the pointers are saved using P_1 and P_2. The call new node ('+', P_1, P_2) construct interior node with a and 5 are the children.

The S-attributed definition of Fig. 4.19. Constructs a syntax free for simple expression containing operators + and –.

Production	Semantic rules
1. $E \rightarrow E_1 + T$	E·node = new Node ('+', E_1·node, T·node)
2. $E \rightarrow E_1 - T$	E·node = new Node ('–', E_1·node, T·node)
3. $E \rightarrow T$	E·node = T · node
4. $T \rightarrow id$	T·node = new leaf (id, id·entry)
5. $T \rightarrow num$	T·node = new leaf (num, num·val)

Fig. 4.19: SDD for Constructing Syntax Tree for an Expression

- The production $E \rightarrow E_1 + T$, creates a node with + for operator and two children E_1·node and T·node.
- The production $E + E_1 - T$ creates a node with – for operator and two children $\in_1$·node and T·node.
- The production $E \rightarrow T$, no node is created.
- For production $T \rightarrow id$, the single terminal present at right, so we construct leaf to create a leaf node for identifier.
- The id·entry points the symbol table.
- Similarly leaf node is created for number.
- The lexical value num·val is value of constant.
- Consider an expression a + 5 – b. Now, the syntax tree for expression a + 5 – b is constructed as shown in Fig. 4.20.
- The parse tree is shown dotted. The nodes labeled by the non-terminals E and T use the synthesized attribute node. The rules are evaluated during bottom-up parse.

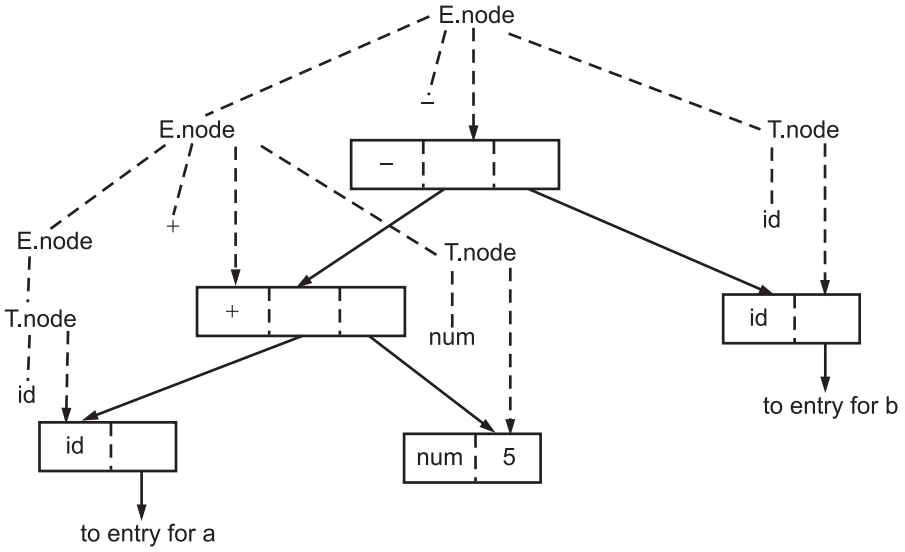


Fig. 4.20: Construction of the Syntax Tree for Expression a + 5 – b

Example 16: Consider L-attributed definition of Fig. 4.21, which uses top-down parsing. The grammar is non-left-recursive. Consider the expression a + 5 – b and construct the syntax tree using definition of Fig. 4.21.

Production	Semantic Rules
1. $E \rightarrow TE'$	E·nod = E' · syn E'·inh = T · node
2. $E' \rightarrow - TE_1'$	E_1'·inh = new Node ('–', E'·inh, T·node E'·syn = E_1'·syn
3. $E' \rightarrow + TE_1'$	E_1'·inh = new Node ('+', E'·inh, T·node) E'·syn = E_1'·syn
4. $E' \rightarrow E$	E'·syn = E'·inh
5. $T \rightarrow id$	T·node = new leaf (id, id·entry)
6. $T \rightarrow num$	T·nod = new leaf (num, num·val)

Fig. 4.21: SDD Expression during Top-down Parsing

Construct dependency graph and syntax tree.

Solution: The syntax tree design for top-down parsing is same as in Fig. 4.20. It uses same steps for construction of syntax tree, even though the structure of the parse tree differs.

The dependency graph is shown in Fig. 4.22.

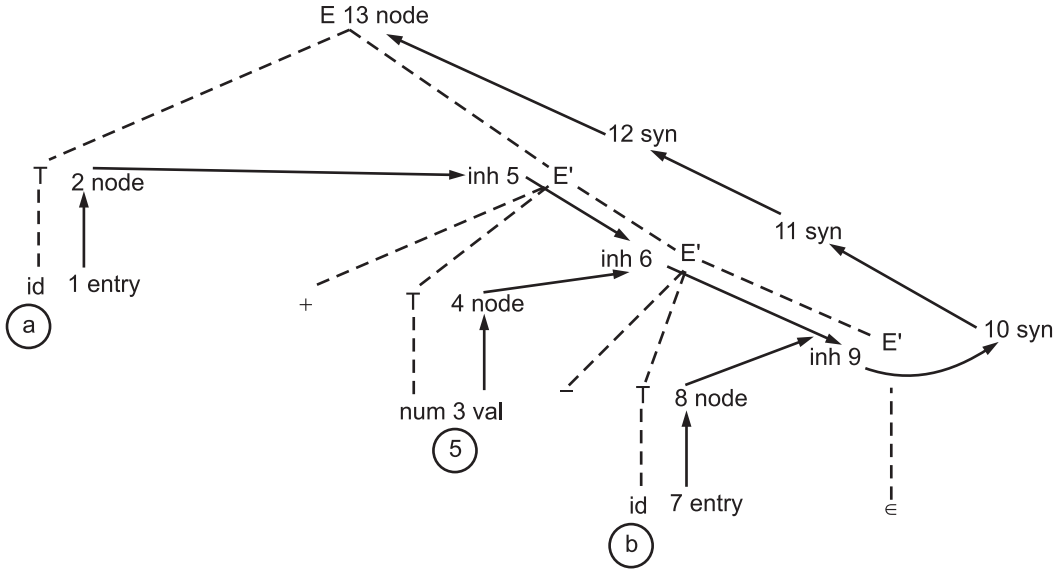


Fig. 4.22: Dependency Graph for a + 5 – b with SDD of Fig. 4.21

4.4.2 The Structure of a Type

- The attributes of parse three are used to carry information from one part of parse tree another. Sometimes, due to the design of the language, the structure differs.
- The structure array is differs with either basic type or array type.
- Let us consider the example of declaration of 2-D array in 'C' language.

Example 17: The array declaration in 'C' is int [2] [4].

This is express as array of 2 arrays of 4 integers. The type expression becomes.

array (2, array (4, integer))

The parse tree is shown in Fig. 4.23. The operator array has 2 parameters.

1. a number
2. a type (if type is a leaf then its parent node array returns two children number and type).

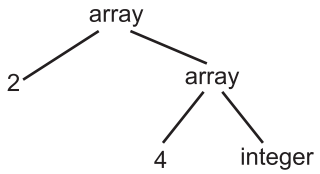


Fig. 4.23: Tree for int [2] [4]

SDD for this type expression is shown in Fig. 4.24.

Production	Semantic rules
1. $D \rightarrow AB$	D·type = B·type B·a = A·type
2. $A \rightarrow$ int	A·type = integer
3. $A \rightarrow$ float	A·type = float
4. $B \rightarrow$ [num] B_1	B·type = array (num·val, B_1·type) B_1·a = B·a
5. $B \rightarrow \in$	B·type = B·a

Fig. 4.24: SDD for Array Type

- D generates basic data type or an array type.
- A generates either int type or float type.
- If derivation is,

  $D \Rightarrow AB \Rightarrow$ int $B \Rightarrow$ int

  $D \Rightarrow AB \Rightarrow$ float $B \Rightarrow$ float.

  then D generates basic type·(B derives $\in$).

- If derivation is,

$$D \Rightarrow AB \Rightarrow \text{int [num] } B_1$$
$$\Rightarrow \text{int [num] [numb] } B_1$$
$$\Rightarrow \text{int [num] [num]}$$

  then D generates array type.
- The non-terminals D and A have a synthesized attribute type. The non-terminal B has two attributes: inherited attribute a and synthesized attribute type. Inherited attribute is used to pass the attribute value a down the tree.
- The non-terminal B inherit type from A.
- The Fig. 4.25 shows the annotated parse tree for input string int [2] [4].

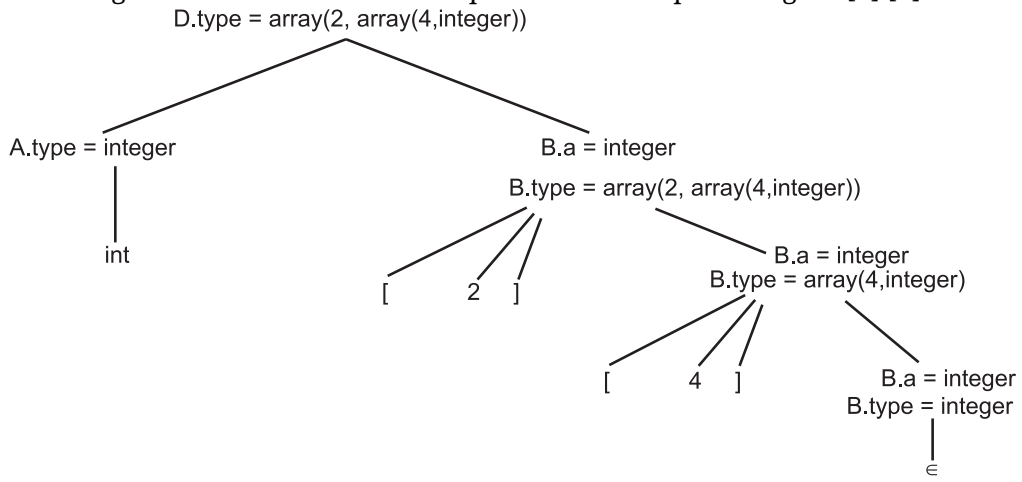


Fig. 4.25: Annotated Parse Tree of Array Type

4.5 SDT SCHEMES

- We have seen how to use syntax directed definition to translate into trees. All these SDD's are implemented using Syntax Directed Translation (SDT) schemes.

4.5.1 Definition

- SDT is context free grammar with program fragments embedded within production bodies; where program fragment is called semantic actions.
- A translator for an arbitrary SDD can be difficult to build. However, there are large classes of SDD for which translators are constructed.
- There are two classes of SDD's to construct translators:
  1. S-attributed (LR-parsable)
  2. L-attributed (LL-parsable).
- In this section we discuss such one class, the S-attributed definition. Syntheiszed attributes can be evaluated by bottom-up parser.
- The parser keeps the values of S-attributes associated with the grammar symbols on its stack, which we will discuss in this section.

4.5.2 Postfix Translation Schemes

- We consider the S-attributed SDD and bottom-up parser, which is the simplest implementation.
- SDT's with all actions at the right ends of the production bodies are called postfix SDT's.

Example 18: Consider the SDD for desk calculator shown in Fig. 4.26.

Productions	Semantic Actions
$L \rightarrow E$ retuns	{ print (E·val);}
$E \rightarrow E_1 + T$	{E·val = $\in_1$·val + T·val;}
$E \rightarrow T$	{$\in$·val = T·val;}
$T \rightarrow T_1 * F$	{T·val = T_1·val $\times$ F·val;}
$T \rightarrow F$	{T·val = F·val;}
$F \rightarrow (E)$	{F·val = E·val}
$F \rightarrow$ digit	{F·val = digit·lexval;}

Fig. 4.26: Postfix SDT for Expression

The postfix SDT of desk calculation has only one change. The first production prints the value. The annotated parse tree for expression (5 + 3 * 4) is shown in Fig. 4.27.

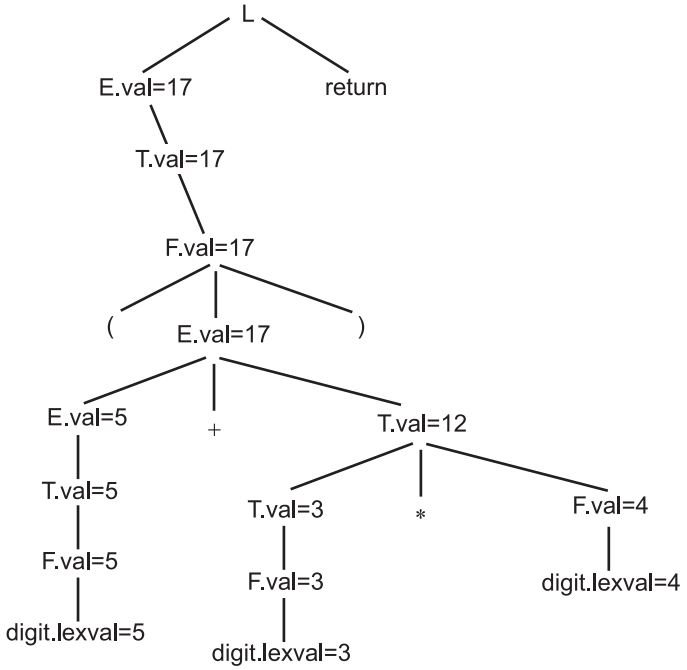


Fig. 4.27: Annotated Parse Tree for (5 + 3 * 4)

These actions can be correctly performed along with reduction steps of the parser.

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which refers to a method of compiler implementation where the source language translation is completely driven by the parser.
   (a) syntax-directed translation (b) compiler-directed translation
   (c) code-directed translation (d) None of the mentioned
2. Syntax directed translation can be based on,
   (a) syntax tree (b) parse tree
   (c) syntax tree as well as parse tree (d) None of the mentioned
3. Which is refers are useful tool for determining an evaluation order for the attribute instances in given parse tree?
   (a) wait-for-graph (b) dependency graph
   (c) sparse-tee graph (d) None of the mentioned
4. Which specifies the values of attributes by associating semantic rules with the grammar productions?
   (a) syntax-directed translation (b) compiler-directed translation
   (c) code-directed definition (d) Syntax Directed Definition (SDD)
5. If an attribute is an attribute of the non-terminal on the left-hand side of a production called an,
   (a) Inherited attribute (b) Synthesized attribute
   (c) Both (a) and (b) (d) None of the mentioned
6. An attribute of a non-terminal on the right-hand side of a production is called an,
   (a) Inherited (I attribute) (b) Synthesized (S attribute)
   (c) Both (a) and (b) (d) None of the mentioned
7. Which is a special form of context-free grammar where some additional information (attributes) is appended to one or more of its non-terminals in order to provide context-sensitive information?
   (a) Regular Grammar (b) Context-Free Grammar
   (c) Operator Grammar (d) Attribute Grammar
8. A parse-tree, with values of its attributes at each node is called as,
   (a) annotated parse tree (b) annotated compiler tree
   (c) annotated dependency graph tree (d) None of the mentioned
9. The methods for ordering the evaluation of attributes includes,
   (a) Oblivious method (b) Parse-tree (topological sort) method
   (c) Rule-based method (d) All of the mentioned

10. SDT's with all actions at the right ends of the production bodies are called as,
    (a) prefix SDT (b) postfix SDT
    (c) parserfix SDT (d) None of the mentioned
11. Which of the following error is expected to recognize by semantic analyzer?
    (a) Type mismatch (b) Undeclared variable
    (c) Reserved identifier misuse. (d) All of the mentioned
12. In a bottom-up evaluation of a syntax directed definition, inherited attributes can be,
    (a) always be evaluated
    (b) evaluated only if the definition is L-attributed
    (c) be evaluated only if the definition has synthesized attributes
    (d) never be evaluated
13. What is true about Syntax Directed Definitions?
    (a) Syntax Directed Definitions + Semantic rules = CFG
    (b) Syntax Directed Definitions + CFG = Semantic rules
    (c) CFG + Semantic rules = Syntax Directed Definitions
    (d) None of the mentioned

Answers

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

Q. II Fill in the Blanks:

1. Syntax Directed Definition (SDD) is a _____ grammar with attributes and rules together which are associated with grammar symbols and productions respectively.
2. _____ is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
3. _____ of a language provide meaning to its constructs, like tokens and syntax structure.
4. Syntax directed definition that involves only _____ attributes is called S-attributed.
5. Syntax trees are _____ top-down and left to right.
6. Attributes of _____ definitions may either be synthesized or inherited.
7. If an SDT uses only synthesized attributes, it is called as S-attributed _____. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).
8. Synthesized attributes represent information that is _____ passed up the parse tree.

9. Syntax Directed Translation (SDT) are _____ rules to the grammar that facilitate semantic analysis.
10. Semantic analyzer receives AST (Abstract Syntax Tree) from its _____ stage (syntax analysis).

Answers

1. context-free	2. Attribute grammar	3. Semantics	4. synthesized
5. parsed	6. L-attributed	7. SDT	8. being
9. augmented	10. previous		

Q. III State True or False:

1. The main idea behind SDD is that the semantics or the meaning of the program is closely tied to its syntax.
2. The attributes of an S-attributed SDD can be evaluated in bottom up order of nodes of the parse tree.
3. The syntax directed definition in which the edges of dependency graph for the attributes in production body, can go from left to right and not from right to left is called L-attributed definitions.
4. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.
5. The inherited attribute can take value either from its child or from its siblings.
6. Attribute grammar is a medium to provide semantics to the context-free grammar Semantics help interpret symbols, their types, and their relations with each other.
7. L-attributed SDT is the form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes.
8. Some of the semantics errors that the semantic analyzer is expected to recognize include Accessing an out of scope variable, Type mismatch, Undeclared variable, Reserved identifier misuse and so on.
9. A dependency graph represents the flow of information between the attribute instances in a parse tree.
10. Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Answers

1. (T)	2. (T)	3. (T)	4. (T)	5. (F)	6. (T)	7. (T)	8. (T)	9. (F)	10. (T)

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. Define SDD.
2. What is SDT?

3. "S-attributed uses top-up parser". State true or false.
4. State two differences between annotated parse tree and dependency graph.
5. Define annotated parse tree.
6. What is dependency graph?
7. Define L-attributed grammar.
8. State two steps to implement SDT's.
9. What are the two classes of SDD's?
10. S-attributed uses which type of parser
11. What is an L-attributed definition?
12. When do we use the inherited attributes?
13. List any two applications of SDT.
14. Enlist SDT schemes.
15. For each parse-tree node (say X) the dependency graph has a node for each attribute associated with that node. State True/False.

(B) Long Answer Questions:

1. What is SDD? Which two notations used by SDD?
2. What is attributed grammar? Define it. Explain with example.
3. What is inherited attribute? Explain with example.
4. What is annotated parse tree? How to evaluate it? Explain with example.
5. Explain dependency graph with example.
6. Write a short note on: Ordering the evaluation of attributes.
7. With the help of example describe L- and S-attributed definitions.
8. Explain side effects in SDD's in detail.
9. What is syntax tree? How to construct it? Describe with example.
10. Write a short note on: Structure of a type.
11. Describe SDT schemes in detail.
12. Explain Postfix Translation Schemes with the help of example.
13. For the input expression (4 + 6) * (2 + 4), design SDD and draw annotated tree using the following grammar:

    $L \rightarrow E$

    $E \rightarrow E_1 + T \mid T$

    $T \rightarrow T_1 * F \mid F$

    $F \rightarrow (E) \mid digit$
14. Explain the ways to control side effects in SDD's.

15. Consider the following SDT:

Production	Semantic Rule
$E \rightarrow E + T$	$E_1.val = E_2.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * P$	$T_1.val = T_2.val * P.val * P.num$
$T \rightarrow P$	T.val = P.val * P.num
$P \rightarrow (E)$	P.val = E.val
$P \rightarrow 0$	P.num = 1 P.val = 2
$P \rightarrow 1$	P.num = 2 P.val = 1

Solve the following:

(i) What is E.val for string 1 * 1 + 1?

(ii) What is E.val for string 1 * 0?

16. Construct the dependency graph for the following attribute grammar and find the values of S.v. Assume that the value of S.u = 3.

Productions	Semantic Rule
$S \rightarrow ABC$	B.u = S.u A.u = B.v + C.v S.v = A.v
$A \rightarrow \in$	A.v = 2 * A.u
$B \rightarrow \in$	B.v = B.u
$C \rightarrow \in$	C.v = 1

17. Consider the following SDD and find the dependency graph for the expression 4 + 5 + 10.

Production	Semantic Rule
$A \rightarrow PQ$	Q.inh = P.val A.val = Q.syn
$Q \rightarrow PQ$	$Q_1.inh = Q.inh + P.val$ $Q.inh = Q_1.syn$
$Q \rightarrow \in$	Q.syn = Q.inh
$P \rightarrow digit$	P.val = digit.lexval

18. State the type of SDD for the following grammar:

Production	Semantic Rule
A → BC	A.S = B.b B.i = F (C.c, A.s)

19. Show the syntax tree representation and directed acyclic graph representation for the expression, N = N + 20.
20. Consider the following grammar,

    E → E + T | T

    T → num, num | num

    Give a syntax directed definition to determine the type of each sub-expression.
21. Give the examples of SDD which is neither S-attributed nor L-attributed.

UNIVERSITY QUESTIONS AND ANSWERS

April 2016

1. Define the term Attribute grammar. **[1 M]**

Ans. Refer to Section 4.2.

2. For the input expression (5 + (26 – 3)) * 2, design SDD and draw and annotated tree using the following grammar:

E → E + T | E – T | T

T → T * F | F

F → (E) | id **[5 M]**

Ans. Refer to Section 4.2.2.

3. What is the use of dependency graph? **[1 M]**

Ans. Refer to Section 4.3.1.

October 2016

1. Define synthesized attribute. **[1 M]**

Ans. Refer to Section 4.2.1, Point (1).

2. Consider the following grammar for unsigned numbers:

number → number digit | digit

A → 0 | 1| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Design SDD and draw an annotated tree for the number 927. **[5 M]**

Ans. Refer to Section 4.2.2.

April 2017

2. Define annotated parse tree. For the input expression 3*5+4n, draw an annotated parse tree using the following SDD:

Production Rule			Semantic Rule
S	→	E	S.val = E.val
E	→	E1 + T	E.val = E1.val + T.val
E	→	T	E.val = T.val
T	→	T1 * F	T.val = T1.val × F.val
T	→	F	T.val = F.val
F	→	digit	F.val = digit.lexval

[4 M]

Ans. Refer to Section 4.2.2.

3. State one difference between annotated parse tree and dependency graph. **[1 M]**

Ans. Refer to Sections 4.1 and 4.3.1.

October 2017

1. Define parse tree. **[1 M]**

Ans. Refer to Section 4.1.

2. Define the term inherited attribute. **[1 M]**

Ans. Refer to Section 4.2.1, Point (2).

3. For the input expression 10 * 9 * 8 * (7 + 6), design SDD and draw annotated tree using the following grammar:

L → E

E → E_1 + T | T

T → T_1 * F | F

F → (E) digit

[5 M]

Ans. Refer to Section 4.2.2.

April 2018

1. Terminals can have synthesized attributes, but not inherited attributes. State true or false. **[1 M]**

Ans. Refer to Section 4.2.1.

2. Define SDD. (Syntax Directed Definitions). **[1 M]**

Ans. Refer to Section 4.2.

3. Consider the following SDD and find the dependency graph for the expression, 7 * 5:

Production Rules			Semantic Rules
S	→	AB	B.inh = A.val
			S.val = B.syn
B	→	*AB_1	B_1.inh = B.inh * A.val
			B.syn = Bl.syn

B	→	C	B.syn = B.inh
A	→	digit	A.val = digit.lexval

[5 M]

Ans. Refer to Section 4.2.2.

4. Write the steps to construct syntax tree using the semantic rules. Construct a syntax tree for the following SDD:

Production Rules			Semantic Rules
E	→	E_1 + T	E.node = new Node ('+", E_1.node, T.node)
E	→	E_1 – T	E.node = new Node ('–', E_1.node, T.node)
E	→	T	E.node = T.node
T	→	(E)	T.node = E.node
T	→	id	T.node = new Leaf (id, id.entry)
T	→	num	T.node = new Leaf (num, num.val)

[4 M]

Ans. Refer to Section 4.2.2.

October 2018

1. State True or False. An SDD is S-attributed if every attribute is synthesized. **[1 M]**

Ans. Refer to Section 4.2.1.

2. Write a short not on SDD (Syntax Directed Definitions). **[4 M]**

Ans. Refer to Section 4.2.

April 2019

1. Define Annotated parse tree. **[1 M]**

Ans. Refer to Section 4.1.

2. Consider the following grammar and SDD of the grammar as given below:

D → TL

T → int | float

L → L,id | id

The SDD is as follows:

Production	Semantic Rules
1. D → TL	L.inh = T.val
2. T → int	L.inh = T.val
3. T → float	T.val = float
4. L → L_1,id	L_1.inh = L.inh L.val = L_1.inh.id.val
5. L → id	L.val = id.val

Construct a dependency graph for the input string int id_1. id_2. id_3 **[5 M]**

Ans. Refer to Section 4.2.2.

CHAPTER 5

Code Generation and Optimization

Objectives...

- To understand Code Generation
- To learn DAG and Issues in Code Generator
- To study Basic Blocks and Flow Graphs

5.0 INTRODUCTION

- Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.
- The code generator takes an intermediate representation of the source program as input and produces an equivalent target program as output.
- Optimization means making the code shorter/small and less complex, so that it can execute faster and takes lesser memory space.
- In process of translation of source language into target language compiler constructs a sequence of Intermediate Representation (IR).
- The Fig. 5.1 shows the back end of compilation phases where intermediate code is generated and code optimization takes place.

Fig. 5.1: The Back-end of Compilation Phases

- The design of intermediate representation varies from compiler to compiler. For example, for C++ compiler, it uses C language as a intermediate form.

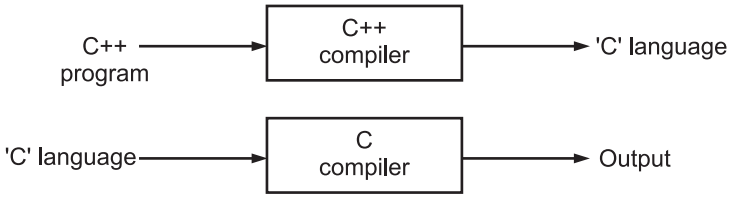


Fig. 5.2: Intermediate Representation

- Compiler needs to produce efficient target programs, so optimizer maps the IR into IR from which more efficient code can be generated.
- In this chapter, we will discuss code optimization and code generation phases of the compiler.

5.1 COMPILATION OF EXPRESSIONS

- The steps performed in code generation for expression are as follows:

Step 1: Determine the precedence and associativity for the operators in an expression and find the order in which expressions are evaluated.

Example,

(a) a + b * c

This expression evaluates b * c first and then the result is added.

(b) a + b – c

This expression is evaluated from left to right.

(c) a ↑ b ↑ c

This expression is evaluated from right to left.

Step 2: Select the instructions to be used in the target code generation.

Example,

a + b is written as,

ADD A, B

Step 3: Use the registers to store the partial result:

Example,

a + b × c is written as

MOVER AREG, B

MULT AREG, C

MOVER AREG, temp

- In the second step, the choice of an instruction to be used in the target code depends on:
  - The type of length of each operand
  - The addressability of each operand

    (where the operand is stored and how it is accessed.
- To maintain the type, length and addressability information for each operand, the operand descriptors are used.
- While evaluating an expression, the partial result of the subexpression is computed. These results are stored in CPU registers for computations.

- If the number of results exceeds than the number of available registers, then the results are moved to memory.
- The following issues can be handled by using register descriptor:
  1. How to move the results between memory and CPU registers.
  2. How to know which partial result is contained in a register.
- Register descriptor is used to maintain the register status.
- In this section we will discuss operand descriptors and register descriptors in detail.

5.1.1 Operand Descriptors [April 19]

- Operand descriptors consist of:
  1. **Attributes:** It contains the subfields type, length and other information of operand.
  2. **Addressability:** It specifies the location of the operand and also specifies how to operand can be accessed. It has two subfields.
     (i) **Addressability Code:**
     M : Operand is in memory.
     R : Operand is in register.
     AR : Address in register.
     AM : Address in memory.
     (ii) **Address:** It is address of CPU register or memory word.
- Each operand specifies the operand descriptor i.e. for id, constants and partial results. We assume that all operand descriptors are stored in an array called operand descriptor e.g. descriptor #i is the descriptor in array operand_descriptor [i].

Example 1: Consider the expression x * y. The code generated is as follows:

```
MOVER AREG,  x
MULT  AREG,  y
```

Assume x and y are integers of size 1 memory word. Three operand descriptors are used during code generation are as follows:

1.	(int, 1)	M, addr(x)	Descriptor for x
2.	(int, 1)	M, addr(y)	Descriptor for y
3.	(int, 1)	R, addr (AREG)	Descriptor for x * y

The code generated in YACC (Yet Another Compiler - Compiler) during parsing is shown in Fig. 5.3.

```
%%
E: E + T{$ $ = codegen ('+', $ 1, $ 3)}
  | T   { $ $ = $ 1}
```

```
    ;
T: T * F{$ $ = codegen ('*', $ 1, $ 3)}
  | F   { $ $ = $ 1}
F: id   { $ $ = getreg ($ 1)}
   ;
% %
  getreg (operand)
  {
    i = i + 1;
    operand_descr [i] = ((type), (addressability_ code, address))
    return i;
  }
```

Fig. 5.3: Code Generator

The getreg returns the location L to hold the value of id in the assignment statement. When an operator is reduced by the parser, the function 'codegen' is called with that operand and descriptors of its operands as parameters.

Example 2: Consider the expression a * b + c. The code generated for this expression is:

```
MOVER   AREG,   A
MULT    AREG,   B
MOVER   AREG,   TEMP
ADD     AREG,   C
```

Five operand descriptors are used during code generation. Assume a, b and c are integers of size 1 memory word.

The operand descriptors are as follows:

1.	(int, 1)	M, addr(a)	Descriptor for a
2.	(int, 1)	M, addr(b)	Descriptor for b
3.	(int, 1)	R, addr (AREG)	Descriptor for a * b
4.	(int, 1)	M, addr (C)	Descriptor for c
5.	(int, 1)	R, addr (AREG)	Descriptor for temp + c

5.1.2 Register Descriptors

- Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register.
- Whenever, a new register is required during code generation, this descriptor is consulted for register availability.

- The register descriptor consist of two fields:
  1. **Status:** 'free' or 'occupied' to indicate register is free or occupied respectively.
  2. **Operand descriptor #:** If status = occupied, this field contains the descriptor # for the operand which is in the register.
- The array Register_descriptor is used to store register descriptors. For each CPU registers one register descriptor is assigned.

Example 3: Consider the statement x * y

The register descriptor for AREG after generating code for x * y in Example 1 of Sec. 5.1.1 is,

Occupied	# 3

This implies that register AREG contains the operand descriptor #3.

Example 4: Consider the expression a * b + c * d. Let us see how the partial results are saved. When all registers are occupied, the next operator is evaluated by copying the contents of registers into the memory and free registers.

The steps in code generation for a * b + c * d:

Step 1: Build descriptor #1 for identifier a.

Step 2: Build descriptor #2 for identifier b.

Step 3: Generate code,

```
MOVER AREG, A
MULT AREG, B
BUILD descriptor #3
```

Step 4: Build descriptor #4 for identifier c.

Step 5: Build descriptor #5 for identifier d.

Step 6: Generate code MOVEM AREG, TEMP1,

```
MOVER AREG, c
MULT AREG, d
buid descriptor #6
```

Step 7: Generate code,

```
ADD    AREG      TEMP1
```

- The partial result saved last is always used before the results saved earlier e.g. value of c * d is used before the value of a * b.
- LIFO data structure is used i.e. stack. temp1 is used i.e. stack. temp1 is used to store the result of a * b. The register temp1 is reused for storing the further partial results.
- Fig. 5.4 (a) shows the operand and register descriptors after step 5.
- Fig. 5.4 (b) shows the operand and register descriptors after step 7.

	Operand Descriptors	
1	(int, 1)	M, addr(a)
2	(int, 1)	M, addr(b)
3	(int, 1)	R, addr(AREG)
4	(int, 1)	M, addr(c)
5	(int, 1)	M, addr(d)

Register Descriptor

Occupied	#3

(a)

	Operand Descriptors	
1	(int, 1)	M, addr(a)
2	(int, 1)	M, addr(b)
3	(int, 1)	R, addr(temp[1])
4	(int, 1)	M, addr(c)
5	(int, 1)	M, addr(d)
6	(int, 1)	R, addr(AREG)

Register Descriptor

Occupied	#6

(b)

Fig. 5.4: Operand and Register Descriptors

- Here, Register descriptor #3 is used to indicate that the value of the operand described by operand descriptor #3 (after a * b) has been moved to memory location temp1. The register descriptor #6 which describe the partial result, c * d; since the operand descriptor #6, it is defined as a partial result.

Example 5: Consider a statement, a * b + c * d * (e + f) + c * d. Show the contents of operand descriptors and register descriptor after complete code generation.

Solution: The code generated is:

```
MOVER   AREG,   A
MULT    AREG,   B
MOVEM   AREG,   TEMP1
MOVER   AREG,   C
MULT    AREG,   D
MOVEM   AREG,   TEMP2
MOVER   AREG,   E
ADD     AREG,   F
MULT    AREG,   TEMP2
ADD     AREG,   TEMP1
MOVEM   AREG,   TEMP1
MOVER   AREG,   C
MULT    AREG,   D
ADD     AREG,   TEMP1
```

	Operand Descriptors	
1.	(int, 1)	M, addr (a)
2.	(int, 1)	M, addr (b)
3.	(int, 1)	M, addr (temp[1])
4.	(int, 1)	M, addr (c)
5.	(int, 1)	M, addr (d)
6.	(int, 1)	M, addr (temp[2])
7.	(int, 1)	M, addr (e)
8.	(int, 1)	M, addr (f)
9.	(int, 1)	M, addr (temp[2])
10.	(int, 1)	M, addr (temp[1])
11.	(int, 1)	M, addr (temp[1])
12.	(int, 1)	M, addr (c)
13.	(int, 1)	M, addr (d)
14.	(int, 1)	M, addr (temp[1])

Register	Descriptor
Occupied	# 14

Result is stored after code generator at statement 14

5.2 INTERMEDIATE CODES FOR EXPRESSIONS

- Some kind of intermediate code is generated to represent the results of syntax analysis are as follows:
  1. Postfix strings,
  2. Triples and quadruples, and
  3. Expression trees.
- The input to the code generation consists of an intermediate optimized code that can be a sequence of quadruples, a sequence of triples or a postfix string.

5.2.1 Postfix Strings

- In the postfix notation an operator appears to the right of its operands. This is also called Polish notation.
- In postfix polish notation, each operator is written immediately after its operands. The set of polish expressions with operators + and * can be generated by the grammar.

<operand> → <operand><operand> +

<operand> → <operand><operand> *

<operand> → var

where, var denotes any variable.

- Consider grammar of expressions:

  E → E + E | E * E | id

  Operator precedence matrix is as shown below:

LHS Symbol \ RHS Symbol	id	+	*
id	≐	⋗	⋗
+	⋖	⋗	⋖
*	⋖	⋗	⋗

Procedure: To parse a sentence by using operator precedence symbol:

Step 1: Put ⋖ at the left end of the input and put ⋗ at the right end of input.

Step 2: Remove all non-terminals from input.

Step 3: Put precedence operator between every 2 terminals.

Step 4: Reduce the innermost sentence enclosed between ⋖ and ⋗.

Step 5: Continue till no more terminals are present.

- To implement this procedure for compilation of expression a stack is used. Symbols are shifted till ⋗ is found, and are removed from stack till ⋖ is found and reduce the string. This is continued till the entire sentence is parsed. Here, we will discuss how expression string is parse and converted into postfix notation.
- Consider source string in infix form is,

  |– a + b * c + d * e ↑ f –|

  [2] [1] [5] [4] [3] ... … (5.1)

  ⋖ ⋖ ⋗ ⋖ ⋖ ⋗

- The number appearing over the operators indicate their evaluation order. The stack is used in the following ways to convert expression string in postfix form:
  1. Initialize stack and reset stack pointer.
  2. Read start string.
  3. Enter symbol into stack (operands) until the new operator has lower precedences than TOS operator. Here, we have encountered handle and evaluation takes place.

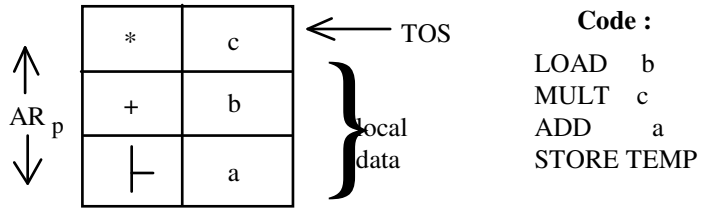


Fig. 5.5: Stack of Partial Implementation of AR_p

- This partial result is stored in TEMP i.e. temporaries in AR_p (activation recorde of postfix expression) and perform other evaluation until no more terminals are present.
- The postfix form of intermediate code is,

  |– a b c * + d e f ↑ * + –|

  [1] [2] [3] [4] [5]

- The postfix string is a popular intermediate code in non-optimizing compilers due to simple code generation and use.

5.2.2 Triples and Quadruples [Oct. 16, 17, April 17, 19]

- The instruction of the Triples presentation is divided into three fields - op, arg1 and arg2. The fields arg1 and arg2 for the arguments of op (operator) are either pointers to the symbol table or pointers into the triple structure. Since three fields are used, the intermediate code format is known as triples.
- A quadruple is a record structure with four fields which we call op, arg 1, arg2 and result. The op field contains an internal code for the operator. The three-address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result.
- Statements with unary operators like x = -y or x = y do not use arg2. Operators like param (Parameter Operator) use neither arg2 nor result. Conditional and unconditional jumps put the target label in result.
- The contents of fields arg1, arg2 and result are normally pointers to the symbol table entries for the names represented by these fields.

1. Triples: [Oct. 16, 17, April 19]

- The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.
- A triple is a representation of an elementary operation in the following tabular form:

Operator	Operand 1	Operand 2

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. So, we use only three field to represent a statement.

- Since, three fields are used, the intermediate code format is known as triples. Every triple has its own number.
- Each operand of a triple is either a variable or constant or the result of some evaluation represented by another triple.
- If the result is used as operand, then the next evaluation if the result is same, the operand field contains that triple's number.

Example 6: Consider the expression a + (b * c) + d ↑ e.

The postfix form is abc * + de ↑

The Fig. 5.6 shows the triples for the above expression.

	Operator	Operand 1	Operand 2
1	*	b	c
2	+	(1)	a
3	↑	e	f
4	*	d	(3)
5	+	(2)	(4)

Fig. 5.6: Triples for String ⊢a + b · c + d * c ↑ f ⊣

Example 7: Consider the expression, a = b * – c + b * – c.

The triple * representation is,

Triple No.	Operator	Operand 1	Operand 2
1	uminus	c	–
2	*	b	1
3	uminus	c	–
4	*	b	3
5	+	2	4
6	assign	a	5

Example 8: Show the triple representation of x = y[i].

The triple representation is,

	Operator	Operand 1	Operand 2
1	= [ ]	y	i
2	assign	x	1

A hash organization can be used for the table of triples. Triples are useful in code optimization.

2. Indirect Triples:

- To eliminate the common sub-expressions which are used more than one place in a program, indirect triples is useful. Indirect triples is useful in optimizing compilers.
- A program statement is represented as a list of triple numbers. While processing a new expression, the occurrences of identical expressions are detected and the triple number is searched from the triple table for that expression and then statement table is formed.
- The indirect triples representation saves the memory (storage economy). It is also used in certain forms of optimization called common subexpression elimination.

Example 9: Fig. 5.7 shows the indirect triples representation for program segment,

z = a + b * c + d * e ↑ f;

y = x + b * c

Use of b * c in both statements is reflected by the fact that triple number 1 appears in the list of triples for both statements.

	Operator	Operand 1	Operand 2
1.	*	b	c
2.	+	(1)	a
3.	↑	e	f
4.	*	d	(3)
5.	+	(2)	(4)
6.	+	x	(1)

(a) Triple's Table

Stmt. No.	Triple Nos.
1	1, 2, 3, 4, 6
2	1, 6

(b) Statement Table

Fig. 5.7: Indirect Triples

Example 10: Consider the expression, a = b * – c + b * – c.

Indirect triples representation is:

	Operator	Operand 1	Operand 2
1.	unimus	c	–
2.	*	b	1
3.	+	2	2
4.	assign	a	3

(a) Triple's Table

Stmt. No.	Triple Nos.
1	1, 2, 3, 4

(b) Statement Table

Fig. 5.8

3. Quadruples: [Oct. 16, 17, April 17, 19]

- A quadruple is a record structure with four fields:

Operator	Operand 1	Operand 2	Result name

- Here, result name is the result of the evaluation. It can be used as the operand of another quadruple.
- When an expression is to be moved from one part of a program to another part during program optimization, then triples are not suitable because triple numbers would change. Hence, quadruples are more convenient than using a number to designate a subexpression.

Example 11: Fig. 5.9 shows the quadruple of expression String 5.1.

	Operator	Operand 1	Operand 2	Result name
1.	*	b	c	t_1
2.	+	t_1	a	t_2
3.	↑	e	f	t_3
4.	*	d	t_3	t_4
5.	+	t_2	t_t	t_5

Fig. 5.9: Quadruples

Example 12: Consider the expression,

```
a = b * – c + b * – c
```

The quadruples representation is:

	Operator	Operand 1	Operand 2	Result name
1.	uminus	c		t_1
2.	*	b	t_1	t_2
3.	uminus	c		t_1
4.	★	b	t_3	t_4
5.	+	t_2	t_4	t_5
6.	:=	t_5		a

Fig. 5.10: Representation of Quadruples

- The contents of fields operand1, operand2 and result are normally pointers to the symbol table entries for the names represented by these fields.
- If so, temporary names must be entered into the symbol table as they are created and location is easily accessed via the symbol table.

Comparison of Quadruples and Triplet:

(i) Quadruples are more useful in an optimizing compiler, where statements are often moved around. Triplets are not much more suitable in optimizing compiler.

(ii) When an expression is to be moved from one part of a program to another part during program optimization, triple numbers would change. In quadruples, if we move a statement computing x, the statement using x requires no change.

(iii) Indirect triples look very much like quadruples as far as their utility is concerned. They use same amount of space and they are equally efficient for recording of code.

(iv) Indirect triples can save some space compared with quadruples if the same temporary value is used more than once.

4. Expression Tree: **[April 17]**

- Expression tree is an important form of optimization which saves number of machine instruction while code generation.
- Expression tree as name suggests is nothing but expressions arranged in a tree-like structure in which internal node corresponds to the operator and each leaf node corresponds to the operand.
- For example: Expression is (A + B) / (C – D). The expression tree is shown in Fig. 5.11.

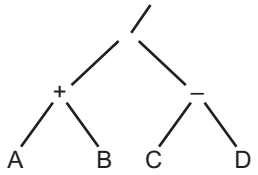


Fig. 5.11: The Expression Tree

- The code generated is as follows:

Left-right Code		Right-to-left Code	
MOVER	AREG, A	MOVER	AREG, C
ADD	AREG, B	SUB	AREG, D
MOVEM	AREG, TEMP 1	MOVEM	AREG, TEMP 1
MOVER	AREG, C	MOVER	AREG, A
SUB	AREG, D	ADD	AREG, B
MOVEM	AREG, TEMP 2	DIV	AREG, TEMP1
MOVER	AREG, TEMP1		
DIV	AREG, TEMP 2		

- Instead of always generating code from left-to-right, the right subtree is evaluated before the left-subtree, if the register requirements of both subtree of an operation are identical.

5.3 CODE OPTIMIZATION [April 16, 18, 19]

- For efficient execution of the program, code optimization is required. Code optimization is back-end phase of compiler which depends only on target code.

- The goals of optimization are the reduction of execution time and the improvement in memory usage.
- Efficiency is achieved by:
  1. Eliminating the redundancies in a program.
  2. Rearranging or rewriting computations in a program.
  3. Using appropriate code generation strategies.
- The Fig. 5.12 shows the optimizing compiler. The code optimization depends upon the intermediate representation of the source code.
- The front end generates the intermediate code which consists of triplet and quadruples.
- To improve the efficiency of program, an optimizing transformation is needed. The Transformed Intermediate Code (IR) is input to the code generation phase.

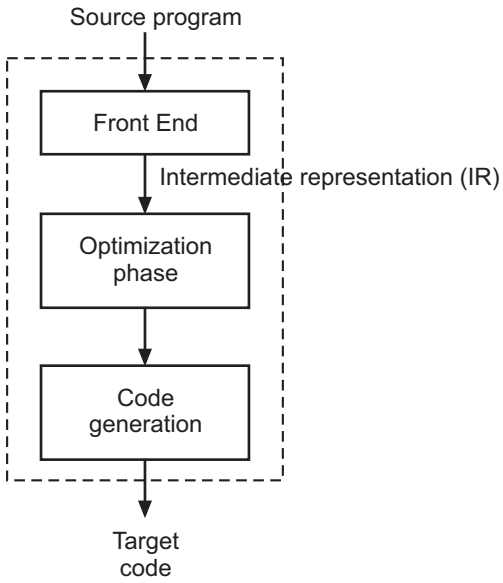


Fig. 5.12: Optimizing Compiler Schematic

- The optimization techniques are independent of both the programming language and the target machine.

Advantages of Code Optimization:

1. The optimized program occupied 25 per cent less storage and execute three times faster than un-optimized program.
2. Reduces cost of execution.

Disadvantage:

1. The 40% extra compilation time is needed.

5.3.1 Optimizing Transformations

- Compiler optimization is generally implemented using a sequence of optimizing transformations, algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources or executes faster.
- Optimizing transformations are classified into local and global transformation.
  1. **Local transformation** is used for small segments of a program.
  2. **Global transformation** is used for small segments of a program. Global transformation is used for larger segments consisting of loops or function bodies.
- There are a number of ways in which a compiler can improve a program optimization without changing the function it computes.
- Code optimization may require various transformation of the source program. The following are the commonly used optimizing transformations:
  1. Compile time evaluation,
  2. Elimination of common sub-expressions,
  3. Dead code elimination,
  4. Frequency reduction, and
  5. Strength reduction.

5.3.1.1 Compile Time Evaluation

- During compilation itself if certain actions are performed then execution efficiency can be improved.
- There are some statements in the program, which can be executed at the compilation time and hence it reduces the execution time.
- Constant folding is one of the type. The value of the operation or constant can be replaced at the time of compilation. Thus X = 20/2 = can be replaced by X = 10 at compile time itself.
- Example, expression 2 * 3.14 can be replaced by 6.28.

5.3.1.2 Elimination of Common Sub-expressions

- An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.

  For example,

  a = x * y t = x * y

  .....$\Rightarrow$ a = t

```
.....  .....
b = x * y + 10  .....
                b = t + 10
```

- Here, subexpressions contain two occurrences of x * y. The second occurrence of x * y can be eliminated because the first occurrence of x * y is always evaluated before the second occurrence, during execution. The first result of x * y is saved in t and this value is used in assignment of b.
- For example,

t_1:= 4 * i		t_1 += 4 * i
x:= a[t_1]		x:= a[t_1]
t_2:= 4 * i		t_2:= t_1
t_3:= 4 * j	$\Rightarrow$	t_3: 4 * j
t_4:= a[t_3]		t_4:= a[t_3]
t_5:= 4 * j		t_5:= t_3
a[t_5]:= x		a[t_5]:= x

- Here, the assignments to t_2 and t_5 have the common subexpressions 4 * i and 4 * j respectively, they have been eliminated by using t_1 instead of t_2 and t_3 instead of t_5.

Implementation:

1. Expression which results in the same value are identified.
2. These expressions are easily identified using triples and quadruples.
3. Equivalence of expression is determined by considering whether their operands have the same values in all occurrences.
4. If subexpressions have same value then the expression can be eliminated.

   Use of algebraic equivalence improves the effectiveness of optimization but increases the cost of optimization.

5.3.1.3 Dead Code Elimination [April 18]

- Dead code is the code which can be omitted from a program without affecting the results.
- Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.
- Dead is nothing but the useless code, statements that compute values that never get used.
- Example, consider the following code segment:

```
{.....
….
   a=10;
```

```
if (a==10)
{
x++;
  printf("%d", x);
}
else
{
y++;
  printf("in dead code")
}
}
```

- The segment contains dead code if the value assign to a is not used in the program, no matter how control flows affecting executing this assignment.

5.3.1.4 Frequency Reduction

- Code from the high execution frequency region of the program can be moved to low execution frequency region. This reduces the execution time.
- Consider the following code segment:

```
for (i = 0; i < 10; i ++)
  {
    a: = a + i * b;
    b: = b + i;
    c: = x + 10;
    d: = d * c + b;
  }
```

- In this loop, value of 'c' remains same as every time loop is executed. So c can be calculated outside the loop and the code is rewritten as

```
c: = x + 10,
  for (i = 1; i < 10; i ++)
  {
    a: = a + i * b;
    b: = b + i;
    d: = d * c + b;
  }
```

5.3.1.5 Strength Reduction [Oct. 16]

- High strength operation can be replaced by low strength operation to reduce the execution time. Consider the following code segment:

```
for (i = 1; i < 100; i ++)
  {
    :
    x: = i * 10;
    :
  }
```

- It can be written by replacing * by + as follows:

```
temp: = 10;
  for (i = 1; i < 100; i ++)
  {
    :
    x: = temp;
    :
    temp: = temp + 10;
    :
  }
```

5.3.2 Local and Global Optimizations

- Optimization of a program is structured into following two phases:
  1. **Local optimization:** The optimizing transformations are applied over small segments of a program consisting of a few statements.
  2. **Global optimization:** The optimizing transformations are applied over a program unit i.e. over a function or a procedure.
- Local optimization is performed within a block segment in sequential nature. The certain optimizations, e.g. loop optimization are beyond the scope of local optimization. Global optimization scope is outside the block.
- Consider optimization such as elimination of common subexpression.
- Let the program segment contains n occurrences of expession x * y. In local optimization this subexpression x * y is eliminated within the block sequentially, say using quadruples. Suppose x * y is occur three times in block then optimization is done three times. The forth occurrence is not eliminated because it belongs to a different block. So after local optimization, global optimization only needs to consider elimination of the first occurrence of x * y – other occurrences of x * y are either not

redundant, or would have been already eliminated locally. Consider the global optimization subexpression elimination.

- If some expression x * y occurs in block A, and also in block B, then the subexpression in block B is eliminated with two conditions:
  1. Block B is executed only after block A.
  2. No assignment of x or y have been executed after the last evaluation of x * y in block A.
- The optimization is done by saving the value of x * y in a temporary location in all blocks; which satisfies condition 1.

5.4 THREE ADDRESS CODE

- Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree.
- That syntax tree then can be converted into a linear representation, e.g., postfix notation.
- The three address code is a sequence of instructions of the following form:

  a = b op c

  where, a, b, c are names, constants or compiler generated temporaries and op is an operator.
- Three-address code is a linearized representation of a syntax tree in which explicit names correspond to the interior nodes of the graph.
- The reason for the term 'three-address code' is the each statement usually contains three addresses, two for the operands and one for the result.
- Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.
- For example, a = b + c * d;
- The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

  r1 = c * d;

  r2 = b + r1;

  r3 = r2 + r1;

  a = r3

  r being used as registers in the target program (r1, r2, r3 are compiler generated temporary names.)
- A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms namely, quadruples and triples. Three address code is a linearized representation of a syntax tree or a DAG.

5.4.1 DAG for Expressions [Oct. 16, 17, 18, April 17, 18, 19]

- A Directed Acyclic Graph (DAG) for an expression identifies the common sub-expressions in the expression.
- DAG is constructed similar to syntax tree. A DAG has a node for every sub-expression of an expression; an interior node represents an operator and its children represent its operands.
- The difference between syntax tree and DAG is a node N in a DAG has more than one parent if N represents a common sub-expression.
- In syntax tree for each expression, the tree is drawn and for common expression the sub-trees are duplicated as many times that common sub-expression occurs.
- DAG generates more efficient intermediate code. Example, consider an expression x + x * (y – z) + (y – z) * a.
- The DAG representation of this expression is shown in Fig. 5.13.

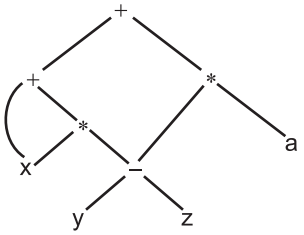


Fig. 5.13: DAG for Expression x + x ∗ (y – z) + (y – z) ∗ a

- In this expression x appears twice, x has two parents because x is common to the two sub-expressions x and x ∗ (y – z).
- The common sub-expression (y – z) occurs twice, it represent by the same node (–), which has two parents.
- The syntax directed definition of expression is shown in Fig. 5.4 used to construct a DAG or a syntax tree.
- The syntax tree for expressions x – y + a or x ∗ y + a also be constructed from Fig. 5.14, where the functions **Leaf** and **Node** created a fresh node each time they were called; even if common sub-expression is present.
- DAG is created if before creating a new node, **Leaf** and **Node** functions first check whether an identical node already exists. If exist, **Node** returns the existing node, otherwise it creates a new node.
- The sequence of steps used to construct the DAG in Fig. 5.13 is shown in Fig. 5.14, provided **Node** and **Leaf** creats new nodes only when necessary. It returns pointers to existing nodes with the correct label and children whenever possible.
- In Fig. 5.14 entry-x, entry-y, entry-z and entry-a points to the symbol table entries for identifiers x, y, z and a respectively.

1. P_1: = Leaf (id, entry_x);
2. P_2: = Leaf (id, entry_x) = P_1
3. P_3: = Leaf (id, entry_y);
4. P_4: = Leaf (id, entry_z);
5. P_5: = Node ('–', P_3, P_4);
6. P_6: = Node ('∗', P_2, P_3);
7. P_7: = Node ('+', P_1, P_6);
8. P_8: = Leaf (id, entry_y) = P_3
9. P_9: = Leaf (id, entry_z) = P_4
10. P_{10}: = Node ('–', P_3, P_4) = P_5
11. P_{11} = Leaf (id, entry_a)
12. P_{12} = Node ('∗', P_5, P_{11})
13. P_{13} = Node ('+', P_7, P_{12}).

Fig. 5.14: Steps for constructing DAG of Fig. 5.13

- Consider call for P_2, when P_2 is call, the node constructed by the previous call P_1 is returned.

  $\therefore \quad P_1 = P_2$

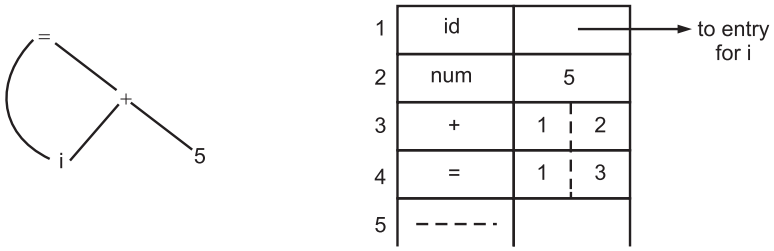
- Similarly, P_8 and P_9 calls returned P_3 and P_4 calls respectively. So node returned on line 10 must be the same one constructed by the call of P_5.

5.4.2 The Value-number Method for Constructing DAG's

- In many applications, nodes are implemented as records stored in an array as shown in Fig. 5.15.
- Each record is one node, which has a label field that determines an operation code. The node can be referred by its index or position in the array. The integer index of a node is called a value number.
- Consider an assignment statement,

  I = i + 5

- The DAG representation is shown in Fig. 5.15.



(a) DAG **(b) Array Representation**

Fig. 5.15: Nodes of a DAG for i = i + 5 allocated in an Array

- The node label = has value numbers 4 and its left and right children's have value numbers 1 and 3 respectively.
- The node label + has value number 3 and its left and right children have value numbers 1 and 2 respectively.
- Thus, value numbers help to construct expression DAG's efficiently. Each node is referred by its value number.
- The signature of an interior node is a triple (op, *l*, r) where op is label, *l* = left child and r = right child value number.
- Let us discuss how value-numbers are useful for constructing the nodes of a DAG.

Constructing Nodes of a DAG:

- Search the array for a node N with label (op, *l*, r). If there is such a node, return the value number of N, otherwise create a new node M (op, *l*, r) and return its value number.
- How to determine the node is already in the array. The most efficient approach is use of **hash table**. The nodes are put into buckets. Each bucket will have only a few nodes.
- The hash function h computes the number of a bucket from the value of op, *l* and r. It will always return the same bucket number for node (op, *l*, r).
- The bucket index h (op, *l*, r) is computed and if N is not in the bucket h (op, *l*, r) then new node M is created and added to this bucket.
- The buckets can be implemented as link lists as shown in Fig. 5.16.

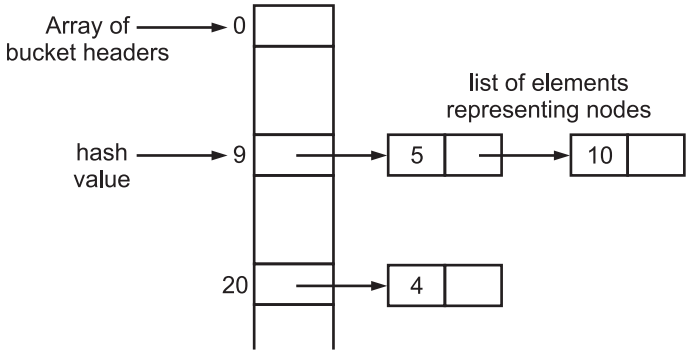
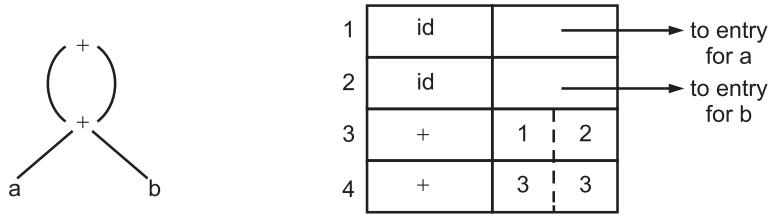


Fig. 5.16: Hash Table for Searching Buckets

- Each cell in a linked list represents a node. The bucket headers, consisting of pointers to the first cell in a list, are stored in an array.
- The node (op, *l*, r) is searched in the list and search is successful if node (op, *l*, r) whose header at index h (op, *l*, r) of the array is found.

Example 13: Construct the DAG and identify the value numbers for the subexpressions of the following expression (a + b) + (a + b).

Solution:



(a) DAG **(b) Array Representation**

Fig. 5.17

5.5 DEFINITION OF BASIC BLOCKS AND FLOW GRAPHS

- Basic blocks are important concepts from both code generation and optimization.
- In this section, we deal with flow graphs and show how flow graph representation of intermediate code is helpful in code generation.

5.5.1 Basic Blocks [Oct. 17]

- Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code.
- These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.
- A basic block is a sequence of consecutive statements like IF-THEN-ELSE, SWITCH-CASE, DO-WHILE, FOR, and REPEAT-UNTIL, etc. in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

```
w = 0;
x = x + y;
y = 0;
if (x > z)
{
   y = x;
   x++;
}
else
{
   y = z;
   z++;
}
w = x + z;
```

Source Code

```
w = 0 ;
x = x + y ;
y = 0 ;
if ( x >  z )
```

```
y = x ;
x++ ;
```

```
y = z ;
z ++ ;
```

```
w = x + z ;
```

Basic Blocks

Fig. 5.18

- Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block.
- If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Definition of Basic Block:

- A basic block is a sequence of consecutive statement in which flow of control enters at the beginning and leaves at the end without halting or branching except at the last instruction.

  Algorithm: Partitioning tree-address instructions into basic blocks.

  Method:

  1. We determine the set of **leaders**, the first statements of basic blocks.

     The rules for finding the leaders are:

     (i) The first three-addresses instruction in the IR is a leader.

     (ii) Any instruction that is the target of conditional and unconditional goto or jump is a leader.

     (iii) Any instruction that immediately follows a goto or conditional goto (jump) statement is a leader.

  2. The basic block consists of a leader and the statements before the next leader.

Example 14: Consider the fragment of code shown in Fig. 5.19, which computes dot product of two vectors of size 10.

```
{
prod=0;
   i=1;
do
{
   prod=prod+a[i]*b[i];
   i=i+1;
}
   while i ≤ 10
}
```

Fig. 5.19

The three-address statements are,

```
1.  product = 0
2.  i = 1
3.  t1 = 4 * i    //  (4 * i) bytes is the offset address block b
4.  t2 = a [t1]
```

```
5.  t3 = 4 * i
6.  t4 = b [t3]
7.  t5 = t2 + t4
8.  t6 = prod + t5
9.  prod = t6
10. t7  =  i + 1
11. i = t7
12. if i ≤ 10 goto 3
```

Fig. 5.20: Three Address Code for Fig. 5.19

Here, there are two basic blocks. Statements 1 and 2 are in 1st basic block. The second basic block has leader statement no. 3 and it contains statement 3 to 12. Since from statement 3 loop starts and statement 12 has a jump statement to statement 3. Therefore, 3 is a leader .

So in Fig. 5.20 leaders are statements 1 and 3. Many transformations can be applied to the basic block to improve the quality of code.

Transformations on Basic Blocks:

- Transformations on basic blocks are:
  1. Common sub-expression elimination.
  2. Dead-code elimination.
  3. Renaming of temporary variables.
  4. Interchange of statements.
- Let us see above transformations on basic blocks in detail:

1. Common Sub-expression Elimination: Consider the basic block.

(i)	a = b + c		a = b + c
(ii)	c = a + d	Transformation →	c = a + d
(iii)	x = b + c		x = b + c
(iv)	y = a + d		y = c

- The statements second and fourth compute the same expression i.e. a + d ⇒ b + c + d (value of a is replaced). So we transform basic block as shown above.
- The statements first and third compute different results as third statement uses value of c i.e. expression is b + a + d and first statement is b + c. So they are not similar.

2. Dead Code Elimination:

- Dead code is one or more than one code statements. The dead code plays no role in any program operation and therefore it can simply be eliminated.

- If we remove any root node from DAG which is never subsequently used (dead), then repeated application of such transformation will remove all nodes from the DAG that corresponds to dead code.

3. Renaming Temporary Variables:

- Suppose t = a + b where t is temporary. We change to x = a + b where x is new temporary variable. Then the value of basic block is not changed.

4. Interchange of Statements:

- We can interchange the sequence of two statements without affecting the value of basic blocks.

$t_1 = a + b$

$t_2 = x + y$

5.5.2 Flow Graphs

- Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks.
- A graph representation of three address statement, called a flow graph.
- The flow of control between the set of basic blocks is represented by using flow graph.
  - The nodes of flow graphs are the basic blocks.
  - The block whose first statement is a leader that block is initial block.
  - There is an edge from block A to block B if first instruction of block B immediately follows the last instruction of block A.
- Fig. 5.21 shows an example of flow graph with basic blocks.

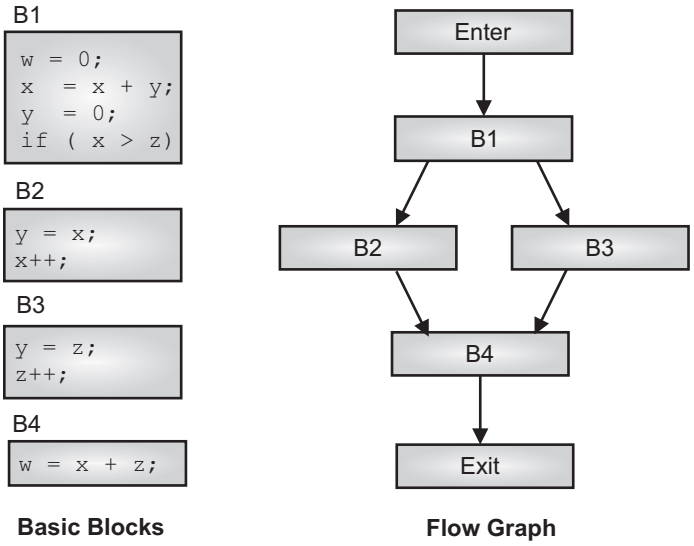


Fig. 5.21

- There are two ways to represent such edge:
  1. There is conditional and unconditional jump from the last statement of A to first statement of B.
  2. Block B immediately follows A in the order of the program and A does not ends with an unconditional jump.
- In other words, A is predecessor of B and B is successor of A.

Example 15: Construct the flow graph of Fig. 5.20.

Solution: There are 2 basic blocks. The flow graph is shown in Fig. 5.22.

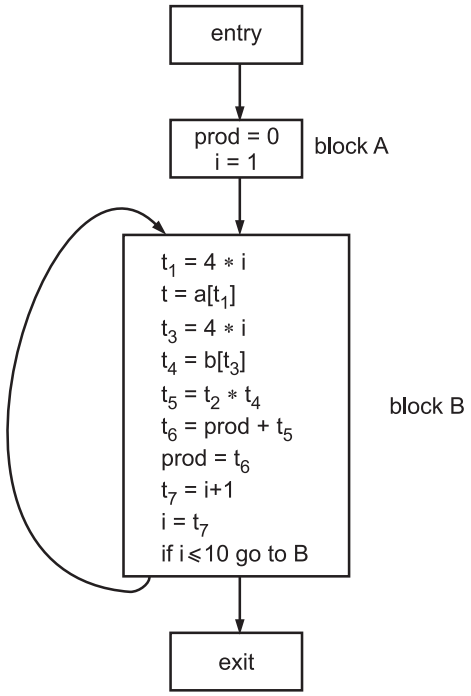


Fig. 5.22: Flow Graph

Here we add 2 nodes **entry** and **exit**, that do not corresponds to executable instructions. Many times the basic block represents intermediate code in the form of quadruples. If quadruples are moved during code optimization then it causes the problem of quadruple number reference in jump statements at the end of basic block. Thus, we prefer to make jumps from point to blocks rather than quadruples as shown in Fig. 5.22.

Example 16: Construct the three-address code and flow graph of the following code segment:

```
void quicksort (m, n)
int m, n
{
    int i, j, v, x;
```

```
    if (n ≤ m) return;
    i = m – 1;
    j = n; v = a [n];
    while (1)
    {
      do i = i + 1; while (a [i] < v);
      do j = j – 1; while (a [j] > v);
      if (i > = j) break;
      x = a [i]; a [i] = a [j]; a [j] = x;   // swap
}
      x = a [i]; a [i] = a [n]; a [n] = x;
      quicksort (m, i); quicksort (i + 1, n);
```

Solution:

(1)	i = m – 1	(16)	$t_7 = 4 * i$
(2)	j = n	(17)	$t_8 = 4 * j$
(3)	$t_1 = 4 * n$	(18)	$t_9 = a [t_8]$
(4)	$v = a [t_1]$	(19)	$a [t_7] = t_9$
(5)	i = i + 1	(20)	$t_{10} = 4 * j$
(6)	$t_2 = 4 * i$	(21)	$a [t_{10}] = x$
(7)	$t_3 = a [t_2]$	(22)	goto (5)
(8)	if $t_3 < v$ goto (5)	(23)	$t_{11} = 4 * i$
(9)	j = j – 1	(24)	$x = a [t_{11}]$
(10)	$t_4 = 4 * j$	(25)	$t_{12} = 4 * i$
(11)	$t_5 = a [t_4]$	(26)	$t_{13} = 4 * n$
(12)	if $t_5 > v$ goto (9)	(27)	$t_{14} = a [t_{13}]$
(13)	if i > = j goto (23)	(28)	$a [t_{12}] = t_{14}$
(14)	$t_6 = 4 * i$	(29)	$t_{15} = 4 * n$
(15)	$x = a [t_6]$	(30)	$a [t_{15}] = x$

Fig. 5.23: Three-address Code

There are 6 basic blocks. Block B_1 start with first instruction. The statement 5 is the leader of block B_2 from which loop start. The statement 9 is the leader of block B_3 where next do loop start. The statement 12 is leader of block B_4 which is conditional jump. The statement 14 is leader of block B_5 where loop ends. The statement 23 is leader of block B_6 and it is jump from block B_4.

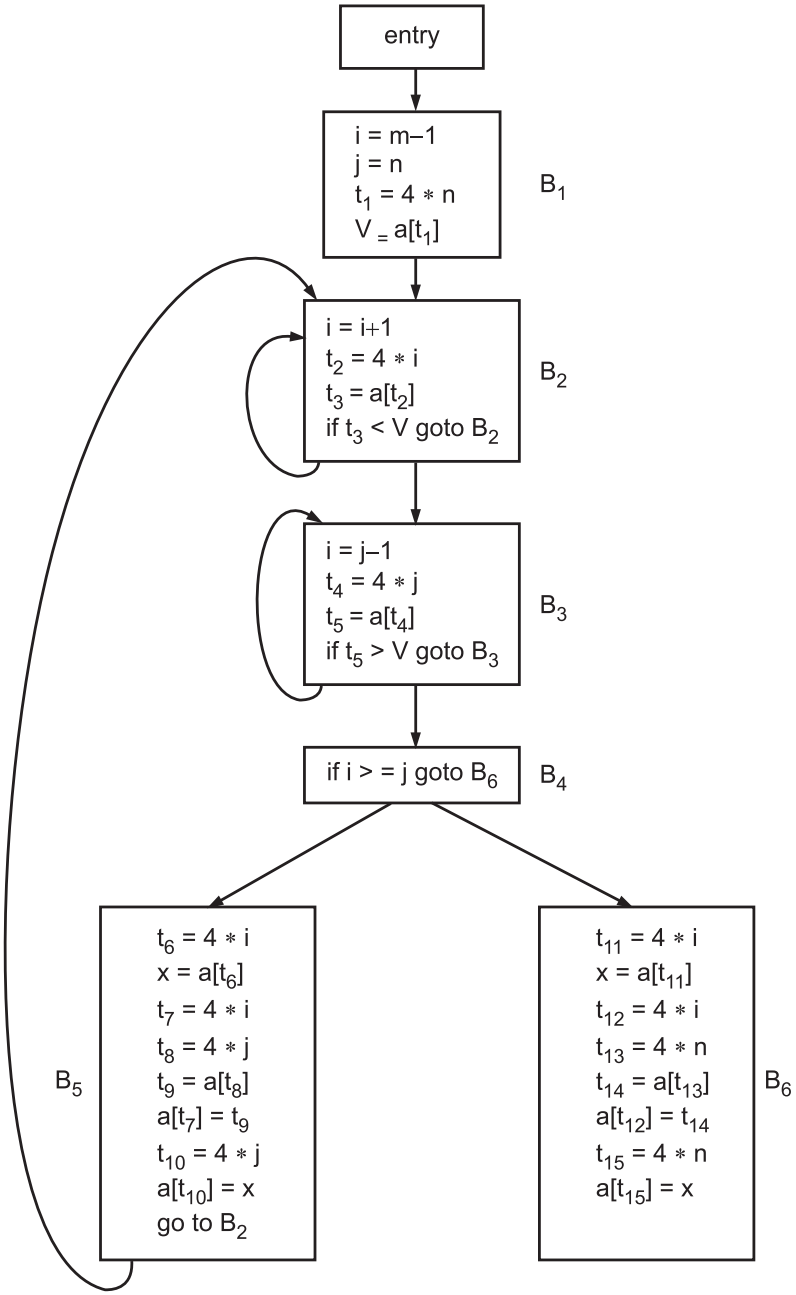


Fig. 5.24: Flow Graph

5.6 DAG REPRESENTATION OF BASIC BLOCK

- For better code optimization basic block is transformed into Directed Acyclic Graph (DAG).

- DAG construction for a basic block with following labels on nodes:
  1. Leaves are labelled by identifiers, variables names or constants. Most leaves has r-value. The leaves represents initial values of names.
  2. Interior initial values of names.
  3. Interior nodes are labelled by an operator.
  4. The sequence of identifiers can also be the label of nodes.
- **Algorithm to Construct a DAG:** Three address statements can be one of the following form:

  Case 1: a = b op c

  e.g. a = b + c

  Case 2: a = b

  e.g. a = b

  Case 3: a = op b

  e.g. a = + 1
- Now, construct a DAG with these 3 statements.
  1. If node b is undefined, create a leaf labelled b and node (b) the new node. If node c is undefined, create leaf labelled c and node (c) is new node.
  2. In case (1) if node labelled op is present then its left child is node (b) and right child is node (c). If op is not present then create a new node, whose left and right children are node (b) and node (c) respectively.

     In case (2) new node (a) is assigned to node (b).

     In case (3) if op is present, then only one child added is node (b).
  3. If case (2) exist, then identifier a is attached to the list of identifier for node b and set node (a) to b.

Example 17: Construct DAG for the block

a = b + c

b = a – d

c = b + c

d = a – d

Solution: Using common sub-expression elimination the block transforms as,

a = b + c

b = a – d

c = b + c

d = b

A DAG is,

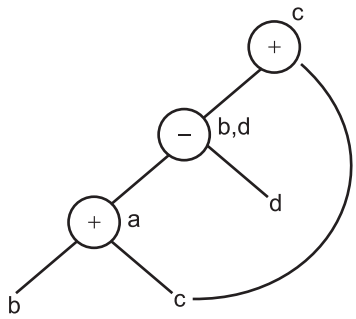


Fig. 5.25: DAG for Basic Block

Example 18: Construct DAG for the block.

1. $t_1 = 4 * i$
2. $t_2 = a\ [t_1]$
3. $t_3 = 4 * i$
4. $t_4 = b\ [t_3]$
5. $t_5 = t_2 * t_4$
6. $t_6 = prod + t_5$
7. $prod = t_6$
8. $t_7 = i + 1$
9. $i = t_7$
10. if $i \leq 10$ goto (1)

Solution: DAG is shown in Fig. 5.26.

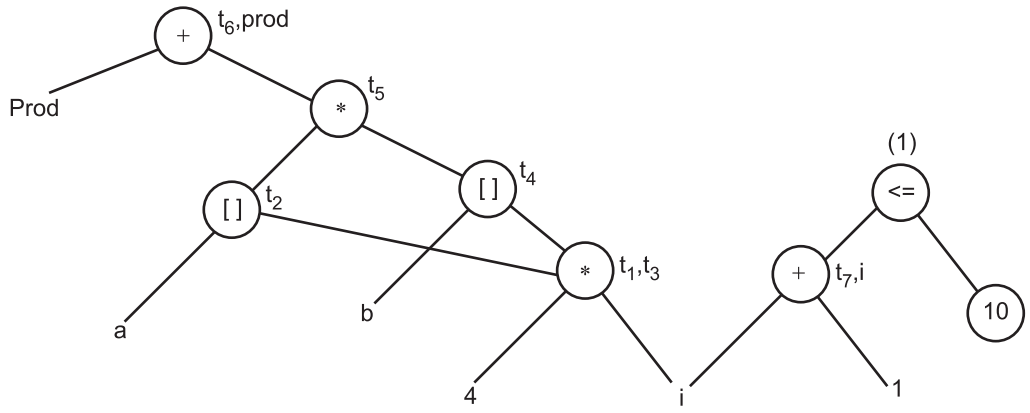


Fig. 5.26: Construction of DAG

Steps of constructing a DAG:

1. For first statement $t_1 = 4 * i$, we create leaves 4 and i.
2. We create a node labelled * and attach identifier t_1 to it.
3. For second statement $t_2 = a\ [t_1]$, we create a new leaf labelled **a** and find previously created node (t_1). Then create a new node labelled [ ] which has two children a and t_1.
4. For third statement, $t_3 = 4 * i$, node 4 and node i were already exist, so operator * will not create a new node but append t_3 on the identifier list of t_1.

5. For fourth statement t_4 = b [t_3], we create new node b (leaf) and t_3 is already exist. So create a new node labelled [ ] which has two children b and t_3.
6. For fifth statement $t_5 = t_2 * t_4$, create a new node for operator * labelled t_5 whose children are t_2 and t_4.
7. For sixth statement t_6 = prod + t_5, create a new node (for **(+)** lablled t_6 whose children are prod and t_5.
8. For seventh statement t_6 = prod, add identifier prod to the list of t_6.
9. For eighth statement $t_7 = i + 1$, add new leaf labelled **1** and new node + labelled t_7.
10. For i = t_7, node i is appended to t_7 identifier list.
11. For last statement i is left child and 10 is right child of operator < = node.

Example 19: Construct a DAG for block:

a = b + c
b = b – c
c = c + d
x = b + c

Solution:

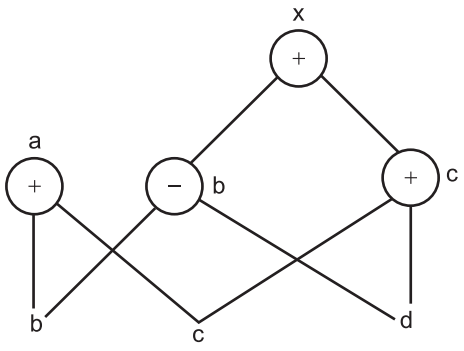


Fig. 5.27: Output of Example 19

Example 20: Construct a DAG for block:

b = a [i]
a [j] = d
e = a [i]

Solution:

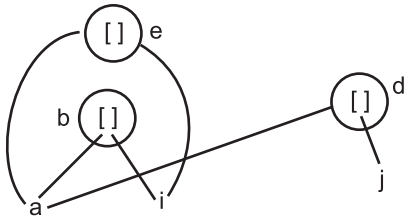


Fig. 5.28: Output of Example 20

5.7 ISSUES IN THE DESIGN OF A CODE GENERATOR

- The main purpose of code generator is to produce a correct code. In this section we will discuss how the code is generated using minimal requirements of registers.
- The code generation depends on the target language and operating system issues such as memory management instruction selection, register allocation and evaluation order.
- Let us examine all these issues:

1. Input to the Code Generator:

- The input to the code generator is the intermediate code produced by the front end along with the symbol table information.
- Many forms of Intermediate Representation (IR) are three-address code such as quadruples, triples, indirect triples, linear representation such as postfix notation and graphical representation such as syntax trees and DAG's.
- These are the input to the code generator and these inputs are free from syntactic and semantic errors.

2. The Target Program:

- The output of the code generator is the target program. Many forms of target codes are:
  (i) absolute machine language.
  (ii) relocatable machine language.
  (iii) assembly language.
- Absolute machine language program can be placed in a fixed location in memory and immediately executed.
- Relocatable machine language program allows subprogram to be compiled separately. If the target machine does not handle relocation automatically, then relocatable loaders and used.
- Assembly language program as a output makes the process of code generation somewhat easier. It generates symbolic instruction code and readability is improved.

3. Memory Management:

- Name-to-address binding of data objects in run-time memory is done by co-operatively by the front end and code generator phase.
- The name in the three-address statement refers to the symbol-table entry during code generation. The symbol table provides the information of relative address of a name in a data area.

4. Instruction Selection:

- To generate efficient code instruction selection is important. For each type of three-address statement, target code is generated.

Example 21: Consider three-address statement a = b + c where a, b and c are statically allocated. The code sequence is:

```
MOV    b,  R0              // load b into register R0
ADD    c,  R0              // add c to register R0
MOV    R0, a               //  store R0 into a
```

The above code sequence generates a poor code because code generated is statement-by-statement.

Example 22: Consider three-address statement,

```
a = b + c
d = a + e
```

The code generated is,

```
1.    MOV   b,  R0
2.    ADD   c,  R0
3.    MOV   R0, a     } operation is repeated
4.    MOV   a,  R0
5.    ADD   e,  R0
6.    MOV   R0, d
```

Here, instructions 3 and 4 are redundant. The code above is poor code generation.

In order to design good code sequences following points have to be consider:

1. We need to know the instruction cost i.e. deciding which machine-code sequence is best.
2. Quality of code should be improved. Quality of code is determined by its speed and size.

For example, the increment statement a = a + 1 can be implemented more efficiently by using single instruction.

```
INC    a      // increment a
instead of   MOV a, R0
ADD    #1,    R0
MOV    R0,    a
```

5. Register Allocation:

- Only few registers are available to perform the operation on any machine. So efficient utilization of registers is important in generating good code.
- The use of registers is divided into two subproblems.

(i) Register Allocation: During register allocation, we select set of variables and allocate to registers.

(ii) Register Assignment: During subsequent register assignment we select registers and allocate a variable.

Example 23: Consider statement: (a + b)/(c – d).

Solution: The codes generated are shown in Fig. 5.29. The Fig. 5.29 (a) shows the code generated when we evaluate expression from left-to-right and Fig. 5.29 (b) shows the code generated when we evaluate expression from right-to-left. The register requirements for both are not identical. Fig. 5.29 (a) requires two registers to store temporary results T_1, T_2. Fig. 5.29 (a) requires only one register to store result T_1.

LOAD	A	LOAD	C
ADD	B	SUB	D
STORE	T_1	STORE	T_1
LOAD	C	LOAD	A
SUB	D	ADD	B
STORE	T_2	DIV	T_1
LOAD	T_1		
DIV	T_2		
(a) Left-to-Right		**(b) Right-to-Left**	

Fig. 5.29

6. Evaluation Order:

- Efficiency of target code also depends on evaluation order. Generate code such that to hold intermediate result less numbers of registers are required. So requirement of registers should be minimal.
- Example 23 shows the order of evaluation is also important to speed up the execution.

PRACTICE QUESTIONS

Q. I Multiple Choice Questions:

1. Which compiler phase gets the intermediate code as input and produces optimized intermediate code as output?
   (a) code generation (b) code optimization
   (c) lexical analysis (d) syntax analysis
2. Which final phase of a compiler and gets input from code optimization phase and produces the target code or object code as result?
   (a) code generation (b) code optimization
   (c) lexical analysis (d) syntax analysis
3. Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the,
   (a) token (b) error
   (c) code (d) tree

4. Which graph depicts how the program control is being passed among the blocks?
   (a) dependency (b) parse
   (c) directed acyclic (d) control flow
5. Which plays no role in any program operation and therefore it can simply be eliminated?
   (a) dead code elimination (b) sleep code elimination
   (c) live code elimination (d) None of the mentioned
6. Which has to track both the registers (for availability) and addresses (location of values) while generating the code.
   (a) syntax generator (b) semantic generator
   (c) code generator (d) None of the mentioned
7. An operand descriptors consists of,
   (a) Attributes contain the subfields type, length and other information of operand.
   (b) Addressability specifies the location of the operand and also specifies how to operand can be accessed.
   (c) Address is address of CPU register or memory location.
   (d) All of the mentioned
8. In which expressions arranged in a tree-like structure in which internal node corresponds to the operator and each leaf node corresponds to the operand.
   (a) Expression tree (b) Syntax tree
   (c) Parse tree (d) None of the mentioned
9. In which notation the expression places the operator at the right end such as as xy^+.
   (a) Polish (b) Postfix
   (c) Infix (d) Both (a) and (b)
10. Intermediate code generation produces intermediate representations for the source program which are of the following forms,
    (a) Postfix notation (b) Three address code
    (c) Expression tree (d) All of the mentioned
11. Which have three fields (operator, operand1, operand2) to implement the three address code?
    (a) quadruples (b) triples
    (c) postfix (d) None of the mentioned
12. Compiler optimization is generally implemented using a sequence of optimizing transformations such as,
    (a) Local (b) Global
    (c) Both (a) and (c) (d) None of the mentioned

13. Which is a type of intermediate code that can contain at most three operands? For example, let 'x = y + z' be an entered statement then it will be written in three-address code as, '+ y, z, x' which means that add y to z and store the result in x.

    (a) Three-address code (b) Quadruples-address code

    (c) triples-address code (d) None of the mentioned

Answers

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

Q. II Fill in the Blanks:

1. _____ generation phase translates the intermediate code representation of the source program into the target language program.
2. Code _____ translates the intermediate code into the machine code of the specified computer.
3. The code which is used for the conversion of source code into machine code is termed as _____ code and lies in the middle of source code and machine code.
4. _____ acyclic graph is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too.
5. Their _____ of operations can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
6. Optimization is a program _____ technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.
7. In _____ -independent optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.
8. _____ descriptor is used to inform the code generator about the availability of registers.
9. A sequence of three address statements (a statement involving no more than three references (two for operands and one for result)) is known as _____ -address code.
10. The intermediate code generator, which is usually the front end of a compiler, is used to _____ intermediate code.
11. _____ optimization is performed within a block segment in sequential nature. The certain optimizations, e.g. loop optimization are beyond the scope of local optimization.
12. Basic _____ play an important role in identifying variables, which are being used more than once in a single basic block.

13. In _____ each instruction in triples presentation has three fields namely, operator, operand1 and operand2 and the results of respective sub-expressions are denoted by the position of expression.
14. There are expressions that consume more CPU cycles, time, and memory and these expressions should be replaced or reduced (strength reduction) with cheaper expressions without compromising the output of _____.
15. Optimization can be done by removing unnecessary _____ lines so that it takes low memory and less execution time.

Answers

1. Code	2. generator	3. intermediate	4. Directed
5. strength	6. transformation	7. machine	8. Register
9. three	10. generate	11. Local	12. blocks
13. Triples	14. expression	15. code	

Q. III State True or False:

1. Code generator takes optimized code as an input and generates the target code for the machine.
2. Basic blocks in a program can be represented by means of control flow graphs.
3. Code optimization phase attempts to improve the intermediate code, so that faster running machine code will result with less memory space.
4. Operand descriptor keeps track of values stored in each register.
5. Optimization means making the code shorter and less complex, so that it can execute faster and takes lesser space.
6. Code generation takes the optimized intermediate code as input and maps it to the target machine language.
7. Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture.
8. Generation of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.
9. In basic blocks when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.
10. A three-address code has at most three address locations to calculate the expression.
11. Dead code is one or more than one code statements, which are either never executed or unreachable and/or if executed, their output is never used.
12. In Quadruples each instruction in quadruples presentation is divided into four fields namely, operator, operand1, operand2 and result.

Answers

1. (T)	2. (T)	3. (T)	4. (F)	5. (T)	6. (F)	7. (T)	8. (F)	9. (T)	10. (T)
11. (T)	12. (T)								

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is the purpose of code generation phase of compiler.
2. Give the function of code optimization.
3. Define triples.
4. Which intermediate code representations of expression are suitable for optimizing compilers?
5. Define postfix string.
6. Define flow graph.
7. Why basic block is transformed into DAG? Give reason.
8. Define the term 'basic block'.
9. Define DAG.
10. List code optimization techniques.
11. Give the DAG representation for the following basic block :

    x = a[i]

    a[j] = y
12. List types of descriptors.
13. What is the use of register descriptor?
14. Define quadruple.
15. State any two advantages of code optimization.
16. Define dead code.
17. Define optimization.
18. What is frequency reduction.
19. Define basic block.

(B) Long Answer Questions:

1. What is code generation? Explain in detail.
2. What is code optimization? Explain in detail.
3. Write a short note on: Compilations of expressions.
4. What are register and operand descriptors? Explain with example. Also differentiate them.
5. Describe intermediate code for expressions in detail.
6. What is three address code? Describe with example.

7. What is triple? Give its syntax. Also explain with example.
8. What is quadruple? How to represent it? Explain with example.
9. Differentiate between triple and quadruple.
10. What is meant by optimizing transformations? Enlist its types.
11. What is dead code? How to eliminate it? Explain with example?
12. What is strength reduction? Describe in detail.
13. What are the phases of optimization? Explain in detail.
14. What is DAG? With help of example explain it.
15. Write a short note on: Value-number method for constructing DAG's.
16. What is basic block? Explain with example.
17. What is meant by transformations on basic blocks? Explain in detail.
18. What is flow graph? What is its purpose? Explain with example.
19. With the help of example describe how to construct DAG using basic block.
20. What are the issues are in design of a code generator? Explain two of them in derail.
21. What is expression tree? How tto create it? Explain wit hexample.
22. Sketch the expression tree for (A + B)/(C – D).
23. Construct the DAG for basic block :

    d = b * c

    e = a + b

    b = b * c

    a = e – d
24. Construct the DAG for basic block,

    a[i] = b

    * x = c

    d = a [j]

    e = * x

    *x = a [i]
25. Construct DAG for following 3-address code :

    b = 10 + a

    x = b [i]

    b [j] = y
26. Construct DAG of the expressions, x = 2 * y + sin (2 * x), z = x/2.
27. Construct DAG and identify the value numbers of the expression a + b + (a + b).
28. Construct triple and indirect triple for the strings, a + b * c + d * e ↑ f & x + b * c.

29. Show indirect triples representation for the program segment:

    z:=a + b * c + d * e ↑ f :

    y:=x + b * c;

30. Generate DAG for the following basic block:

    a = b + c

    b = a – d

    c = b + c

    d = a – b

31. Construct DAG for the following expressions:

    (i) b * (a + c) + (a + c) * d

    (ii) (2) y + (y + x) / (x – z) * (x – z).

32. Construct indirect triples and quadruples for the following:

    (i) a * b + e ↑ f/d – c

    (ii) y + a * b.

33. Generate DAG for the following basic block:

    t_0 : = b + c

    t_1 : = t_0 * d

    a : = t_1

    t_2 : = f * a

    e : = t_2

    t_3 : = b + c

    t_4 : = t_3 * e

    f : = t_4

    f_5 : = b + c

    t_6 : = t_5/d

    g : = t_6

34. Consider following expressions/code segment:

    (i) gamma = m + n * (n + q) ^ t;

    (ii) alpha = (n + q) * (m – n) ^ (m + n);

    Show the entries in triple's table.

35. Construct DAG for the following expressions :

    (i) ((x + y) * x + (2/(x + y))) * ((x + y) * x)

    (ii) 2 + 3 * 4 + (3 * 4)/5.

36. Construct DAG for the following expression :
    (i) 2 * (3 + 4) + (3 + 4) * 2
    (ii) b + (b + a) / (b – c) * (a – c).
37. Construct triple and indirect triple for the strings, a + b * c + d * e ↑ f & x + b * c.
38. Construct indirect triples and quadruples for the expression, a = b * – c + b * – c.
39. Construct DAG for the following Basic block:

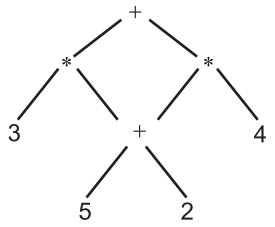
    a = b + c
    b = a + c
    d = b + e

UNIVERSITY QUESTIONS AND ANSWERS

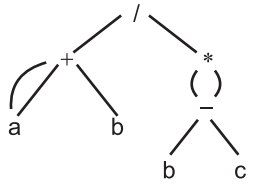
April 2016

1. Construct Directed Acyclic Graph (DAG) for the following expressions:
(i) 3 * (5 + 2) + (5 + 2) * 4
(ii) a + (a + b)/(b – c) * (b – c) **[5 M]**

Ans. (i)



(ii)



2. Write a short note on code optimization techniques. **[4 M]**

Ans. Refer to Section 5.3.

October 2016

1. What is meant by strength reduction? **[1 M]**

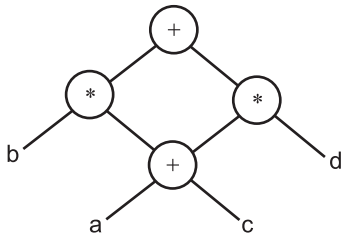
Ans. Refer to Section 5.3.1.5.

2. Construct triples and quadruples for the expression, a + b * c + d * e ↑ f. **[4 M]**

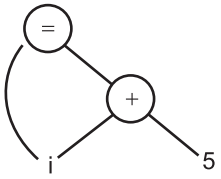
Ans. Refer to Section 5.2.2, Points (1) and (3).

3. Construct DAG for the following expressions:
(i) b * (a + c) + (a + c) * d
(ii) i = i + 5. **[4 M]**

Ans. (i) b * (a + c) + (a + c) * d



(ii) i = i + 5.



April 2017

1. Give an expression tree for (a*b) + ((c*d) * (e+f)) + (c*d). **[1 M]**

Ans. Refer to Section 5.2.2, Point (4).

2. Define Directed Acyclic Graph (DAG). Construct DAG for the following expressions:

(i) b * (a + c) + (a + c) * d

(ii) y + (y + x)/(x – z) * (x – z). **[5 M]**

Ans. Refer to Section 5.4.1.

3. What is the use of quadruples over triplets? Give quadruples representation for the expression, a+b*c+d*e↑f. **[4 M]**

Ans. Refer to Section 5.2.2.

October 2017

1. What is the use of Directed Acyclic Graph (DAG)? **[1 M]**

Ans. Refer to Section 5.4.1.

2. Define the term Basic block. **[1 M]**

Ans. Refer to Section 5.5.1.

3. Construct Directed Acyclic Graph (DAG) for the following expression:

(i) (a + a * (b – c) + (b – c) * d)

(ii) (a + b) + (a + b) **[5 M]**

Ans. Refer to Section 5.4.1.

4. Construct triples and quadruples for the following expression:

(a + b) * (m – n) ↑ (m + n). **[4 M]**

Ans. Refer to Section 5.2.2, Points (1) and (3).

April 2018

1. Define the term dead code. **[1 M]**

Ans. Refer to Section 5.3.1.3.

2. Define Directed Acyclic Graph (DAG). Construct DAG for the following expressions:

(i) (a + a * (b – c)) + ((b – c) * d)

(ii) ((a + b) * (c – d))/f * (a + b). **[5 M]**

Ans. Refer to Section 5.4.1.

3. Explain in detail any two code optimization techniques with appropriate examples. **[5 M]**

Ans. Refer to Section 5.3.

October 2018

1. Define Directed Acyclic Graph (DAG). Construct DAG for the following expressions:

(i) (1 + 1*(3–2) + (3–2)*4)

(ii) r + s * t + (s * t)/u. **[5 M]**

Ans. Refer to Section 5.4.1.

April 2019

1. Define operand descriptors. **[1 M]**

Ans. Refer to Section 5.1.1.

2. List the techniques used in code optimization. **[1 M]**

Ans. Refer to Section 5.3.

3. Consider the expression a=b*(-c)+b*(-c).

Give.

(i) Triple representation

(ii) Quadruple representation. **[4 M]**

Ans. Refer to Section 5.2.2, Points (1) and (3).

4. Define Directed Acyclic Graph (DAG). Construct DAG for the following expression (a+a*(c-b)+(c-b)*d). **[4 M]**

Ans. Refer to Section 5.4.1.

Notes :

Notes :