VBD

Based on New Syllabus

Based on R.T.M. Nagpur University New Syllabus

COMPILER DESIGN

WITH VIVA VOCE

Ketki C. Gode

SOLVED QUESTION BANK + UNIVERSITY PAPER SOLUTIONS

B.TECH. VI SEMESTER

COMPUTER TECHNOLOGY (CT)

VBD

COMPILER DESIGN

WITH VIVA-VOCE

FOR
B.Tech. VI SEMESTER
Computer Technology
(CT)

By
Ketki C. Gode
M.E.
Assistant Professor

Based on R.T.M. Nagpur University New Syllabus

VBD PUBLICATIONS

Distributed by:
VBD **Publications Pvt. Ltd.**
477, Golchha Marg
Sadar Nagpur - 440001.

Sales Office:
Plot No. 236, Near Ram Coolers,
Besides Ganesh Temple,
Singada Market, Navi Shukrawari,
Mahal, Nagpur - 32
☎: 9373644557, 7620266004
Website : www.vbdpublications.com
E-mail : vbdshirke@gmail.com

Price : Rs. 800/-

For academic session : 2022 - 23

The shelf-life of this book is 6 months.

Books Available At ALL LEADING BOOK SELLERS
OR CONTACT:
VIJAY BOOK DEPOT
GOLCHHA MARG, SADAR, NAGPUR
☎: 0712 - 2520496

For online purchases, please visit WWW.VBDPUBLICATIONS.COM

Published by : ABD Publishers & Printers Pvt. Ltd., Nagpur.

PREFACE

This *VBD* on **Compiler Design** has been prepared for the students of **Sixth Semester Bachelor of Technology (CBCS)- Computer Technology (CT)** as per the New Syllabus prescribed by the RTM-Nagpur University.

The subject matter has been thoroughly dealt with, covering 100 % syllabus.

Each unit begins with its syllabus. The previously asked University questions are fully covered in the order as per the syllabus.

In each unit, sufficient theory is given then followed by Problems based on them.

Every type of problem is included alongwith previously asked.

I am grateful to the publishers and the staff of *VBD* Publications, Nagpur, who have taken considerable pains and shown extreme co-operation during the preparation of this book.

I will appreciate suggestions from teachers and students for the improvement of the book in the future editions.

Author

SYLLABUS

B.Tech. VI Semester Computer Technology (CT)

UNIT - I

Translators, Compilers, Interpreters, Just-in-Time Compilers, Cross Compilers, Bootstrapping, Structure of a typical compiler, Overview of lexical analysis, Syntax analysis, Code optimization and code generation, Design of lexical analyzer.

UNIT - II

Parsers, Shift-Reduce Parser, Top-down parser, Predictive Parsers, Bottom up parsing technique, LR parsing algorithm, Design of SLR, LALR, LR Parsers.

UNIT - III

Syntax directed schemes, Intermediate code, Parse trees, Syntax trees, Three address code, Quadruples, Triples, Indirect Triple, Using syntax directed translation Schemes to translate assignment statements, Boolean expression, If then else structures.

UNIT - IV

Sources of Optimization, Loop Optimization, DAG representation of basic blocks, Global data flow analysis, Dominators, Loop, Invariant computations, Induction variable elimination, Loop unrolling, Loop jamming, Simple code generator, Register allocation and assignment.

CONTENTS

B.Tech. VI Semester Computer Technology (CT)

SOLVED UNIVERSITY QUESTION PAPERS

Note:- Due to Covid - 19 Summer - 2020 to Summer - 2022 exams were conducted online by the respective Colleges.

UNIT - I

UNIT - I

CONTENTS

UNIVERSITY PAPER SOLUTIONS
SINCE SUMMER - 2009

Note to the students :

- The questions given below are from previous Nagpur University examination papers.
- Though these questions are from old university papers, they have been included in the new syllabus.
- Questions which are out of new syllabus are not included here.

SUMMER - 09 (CS)

Q.1. Why design of a compiler is complex than that of an assembler? **3M**

Ans. **P.1-15, Q.18**

Q.2. Give finite automata for a typical identifier of a programming language and also write the complete input required for LEX to produce an analyzer that recognise identifiers. **5M**

Ans. **P.1-30, Q.47**

SUMMER - 09 (CT)

Q.1. Explain why design of a compiler is complex than that of an assembler. **2M**

Ans. **P.1-15, Q.18**

Q.2. What are various compiler writing tools? Explain each in brief. **5M**

Ans. **P.1-14, Q.17**

Q.3. Give finite automata for a typical identifier of a programming language and also write the complete input required for LEX to produce an analyzer that recognizes identifiers. **6M**

Ans. **P.1-30, Q.47**

WINTER - 09 (CS)

Q.1. Compiler is very complex to design than assembler. Justify. **4M**

Ans. **P.1-15, Q.18**

Q.2. Explain each phase of compiler. Also discuss input required and output produced by each phase of compiler with suitable example. **6M**

Ans. **P.1-10, Q.12**

WINTER - 09 (CT)

Q.1. Explain why design of compiler is complex than that of an assembler. **3M**

Ans. P.1-15, Q.18

Q.2. What is LEX? Give general specification of LEX program and its compiling process. **5M**

Ans. P.1-18, 20, Q.27 & 29

Q.3. Explain how bootstrapping can be achieved. **4M**

Ans. P.1-8, Q.10

SUMMER - 10 (CS)

Q.1. What is cross compiler? Explain how bootstrapping is used in design of a compiler. **5M**

Ans. P.1-7, 8, Q.8, 10

Q.2. What are the errors that may encounter in each phase of a compiler? **5M**

Ans. P.1-16, Q.22

Q.3. What is LEX ? Give specification of lex program. **3M**

Ans. P.1-18, 20, Q.27, 29

SUMMER - 10 (CT)

Q.1. Explain the structure of typical two pass compiler. Explain each component in detail. **7M**

Ans. P.1-13, Q.13

Q.2. Explain in detail, various compiler writing tools.. **6M**

Ans. P.1-14, Q.17

Q.3. Explain significance of L and R value in writing assignment statements for a particular programming languages. **4M**

Ans. P.1-27, Q.40

Q.4. Explain, in brief, the format of LEX input file. **3M**

Ans. P.1-20, Q.29

WINTER - 10 (CS)

Q.1. How can you differentiate between a phase and a pass? Discuss the factors which affect number of passes in compiler design. **5M**

Ans. P.1-13, 14, Q.14, 15

Q.2. Explain how bootstrapping can be implemented.

Ans. P.1-8, Q.10

Q.3. Discuss the issues in design of a lexical analyzer.

Ans. P.1-17, Q.23

WINTER - 10 (CT)

Q.1. What is translator? Explain complete compilation process fo level language program.

Ans. P.1-6, 10, Q.4, 12

Q.2. What is the theory and logic behind cross compilation?

Ans. P.1-7, Q.8

SUMMER - 11 (CS)

Q.1. Explain various phases of compiler in detail.

Ans. P.1-10, Q.12

Q.2. What is LEX? Explain structure of LEX and write re expression for signed integer constant.

Ans. P.1-18, 20, 22, Q.27, 29, 31

SUMMER - 11 (CT)

Q.1. Write a LEX program to produce a lexical analyser that recog identifiers.

Ans. P.1-20, Q.29

Q.2. What is cross assembler? What is the use of cross assembler?

Ans. P.1-8, Q.9

WINTER - 11 (CS)

Q.1. What are different phases of compiler? Explain any one phas detail.

Ans. P.1-10, Q.12

Q.2. What are compiler writing tools?

Ans. P.1-14, Q.17

Q.3. What is cross assembler? Explain the application of it.

Ans. P.1-8, Q.9

WINTER - 11 (CT)

Q.1. What is front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages? **6M**

Ans. **P.1-13, Q.13**

Q.2. Explain the difference between a compiler language and Interpreted language. Give examples of each. When would we prefer interpreter over compiler? **7M**

Ans. **P.1-7, Q.7**

SUMMER - 12 (CS)

Q.1. Explain various phases of compiler in detail. **8M**

Ans. **P.1-10, Q.12**

Q.2. What are various errors that may encounter in each phase of a compiler? **5M**

Ans. **P.1-16, Q.22**

Q.3. Explain :

(i) Bootstrapping.

(ii) Cross Assembler. **4M**

Ans. **P.1-8, Q. 10, 9**

SUMMER - 12 (CT)

Q.1. What is the difference between the phases and passes of a compiler? **4M**

Ans. **P.1-14, Q.15**

Q.2. What is front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages? **4M**

Ans. **P.1-13, Q.13**

Q.3. Explain with suitable example parameter passing techniques used in programming languages. **6M**

Ans. **P.1-25, Q.37**

WINTER - 12 (CS)

Q.1. Explain the concept of bootstrapping in compiler design. **6M**

Ans. **P.1-8, Q.10**

Q.2. What do you mean by phases and passes of compiler? Explain phases of compiler in detail. **8M**

Ans. **P.1-5, 10, Q.3, 12**

WINTER - 12 (CT)

Q.1. What is the role of symbol table for each phase of compiler? **6M**

Ans. **P.1-16, Q.21**

Q.2. Write short notes :

(i) Cross-compiler

(ii) Parameter transmission. **7M**

Ans. **P.1-7, 25, Q.8, 37**

SUMMER - 13 (CS)

Q.1. Define cross compiler and bootstrapping. How bootstrapping can be achieved? **4M**

Ans. **P.1-7, 8, Q.8, 10**

Q.2. Which tools are available to write a compiler? Explain any one tool for writing lexical analyzer. **6M**

Ans. **P.1-14, 19, Q. 17, 28**

SUMMER - 13 (CT)

Q.1. How is finite automata useful for lexical analysis? And how many tokens are generated in the following statement?

printf("%d%f%",rollno,per_mark) ; **7M**

Ans. **P.1-30, Q.48**

Q.2. Explain various compiler construction tools in details. **6M**

Ans. **P.1-14, Q.17**

WINTER - 13 (CS)

Q.1. What is cross compiler and bootstrapping? **4M**

Ans. **P.1-7, 8, Q.8, 10**

Q.2. Discuss design issues of lexical analyzer. **4M**

Ans. **P.1-17, Q.23**

Q.3. Why lexical analyzer reads few characters beyond the token in advance before declaring validity of tokens? Explain with example. **5M**

Ans. **P.1-18, Q.25**

WINTER - 13 (CT)

Q.1. Explain analysis - synthesis model of compiler in brief. **5M**

Ans. P.1-9, Q.11

Q.2. Explain with suitable examples, the concept of l-value and r-value of an expression. **4M**

Ans. P.1-27, Q.39

Q.3. Explain with suitable example, what is boot-strapping. **4M**

Ans. P.1-8, Q.10

SUMMER - 14 (CS)

Q.1. Draw and explain general structure of compiler design. **6M**

Ans. P.1-10, Q.12

Q.2. What are compiler writing tools? Explain how to use these tools in detail. **7M**

Ans. P.1-14, Q.17

SUMMER - 14 (CT)

Q.1. What are the various stages in the process of generation of the target code from the source code by the compiler? How these stages are broadly classified? Explain each. **10M**

Ans. P.1-10, Q.12

Q.2. Explain the advantages of analysis synthesis model of compilation. **4M**

Ans. P.1-9, Q.11

WINTER - 14 (CS)

Q.1. Explain the following terms :

(I) Syntax and semantics. (II) Source code and object code. **6M**

Ans. P.1-24, Q.35

Q.2. Explain the output of lexical analysis phase. Give the importance of this output in next phases of compiler. **3M**

Ans. P.1-15, Q.19

Q.3. Write notes on LEX **2M**

Ans. P.1-18, Q.27

Note : Also refer Paper Solutions at the end of the book.

SOLVED QUESTION BANK

[Sequence given as per syllabus]

INTRODUCTION

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code) . The most common reason for wanting to transform source code is to create an executable program.

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 or machine code).

A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone process most programming is, instead, done using a high-level programming language.

This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. Hence compiler is very essential

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. During this process, the compiler will also attempt to spot and report programmer mistakes.

INTRODUCTION TO COMPILER

Q.1. What is compiler?

Ans. Compiler :

- A compiler is a program that translates a high-level language program into functionally equivalent low-level language program.
- So, a compiler is basically a translator whose source language is the high-level language and the target language is a low-level language.
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

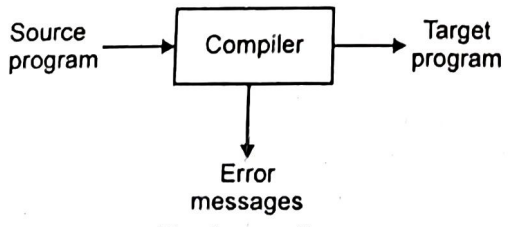


Fig. A compiler

Q.2. What is compilation process? Explain in details.

Ans. Compilation Process :

- Compilation is a process that translates a program in one language (the source language) into an equivalent program in another language (the object or target language).
- Programming language is a problem-oriented language and the target language is a machine language or assembly language (i.e. a machine-oriented language). Thus compilation is a fundamental concept in the production of software.
- It is the link between the (abstract) world of application development and the low-level world of application execution on machines.

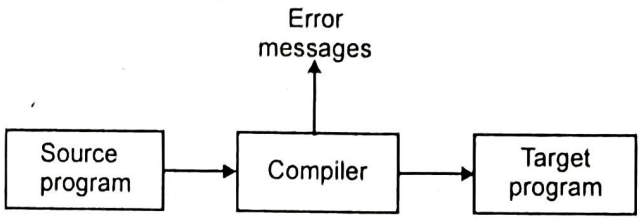


Fig.(a) Compilation process

- Compilation program flowchart can be given as follows :

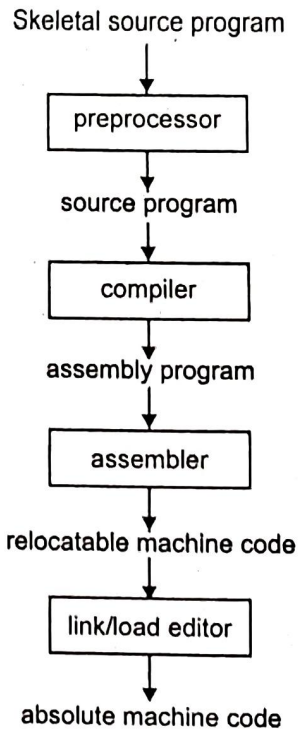


Fig.(b) Compilation program flowchart

Q.3. Define the following :

(i) Tokens (ii) Lexeme

(iii) Patterns

OR Define the following terms :

(i) Phase (ii) Pass

(iii) Pattern (iv) Token

(v) Lexeme

OR What do you mean by phases and phases of compiler?

CS : W-12(4M)

Ans.

(i) Tokens :

- Tokens are symbolic names for the entities that make up the text of the program.
- For example, if for the keyword 'if', and id for any identifier. These make up the output of the lexical analyser.
- Token is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called tokenization.

(ii) Lexeme :

- A lexeme is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token).
- For example, if matches the pattern for if ,and foo123bar matches the pattern for id.

(iii) Patterns :

- A pattern is a rule that specifies when a sequence of characters from the input constitutes a token.
- For example, the sequence if for the token if , and any sequence of alphanumeric starting with a letter for the token id.

(iv) Phase :

- The compiler has to undergo various different individual dependent as well as independent stages of compilation these stages of processing are called as phases of compilation.
- When the source language is large and complex, and high quality output is required, the design may be split into a number of relatively independent phases.
- Having separate phases means development can be parceled up into small parts and given to different people.
- It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

(v) **Pass :**

- Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which made a pass over the source (or some representation of it) performing some of the required analysis and translations.
- The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one-pass compilers generally perform compilations faster than multi-pass compilers.
- Many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

COMPILERS AND TRANSLATORS

Q.4. What is translator? ***CT : W-10(3M)***

Ans. Translator :

- A translator is a program that takes as input a program written in one programming language and produces output program in another language.
- If source language in high level language such as FORTRAN, PL/I, COBOL and object language is a low level such as assembly language or machine language such type of translator is called as compiler.
- Executing a program written is a high level programming language is a two step process.

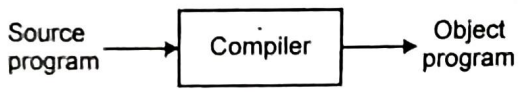


Fig. (1) Compilation

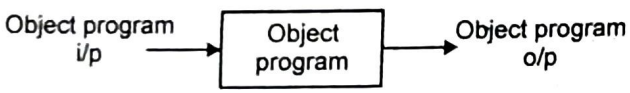


Fig. (2) Execution

- The source program must first be compiled, that is translated into the object program. Then the resulting object program is loaded into memory and executed.

Q.5. State the different types of translators.

Ans. The different types of translators are as follows :

(i) **Interpreter :**

- It translate programming language into simplified langauge called as intermediate code.
- Interpreters are often smaller than compiler and facilitate the implementation of complex programming language constructs.
- The main disadvantage of interpreters is that the execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

(ii) **Assembler :**

- Programmers find difficulty in writing machine language programs. Hence mnemonics (symbol) are developed for each instruction.
- The mnemonics would easily get converted to machine language programs (codes)
- Such language that is understood by processor using symbols is called as assembly language.
- Programs written to translate the assembly language to machine language is called as assemblers.
- The input to an assembler program is called source program, the output is a machine code called as object program.

(iii) **Preprocessor :** It takes program in one high level language and translate it into the equivalent another high level language.

Q.6. Why we need translators?

Ans.

- With the machine language we must communicate directly with computer in terms of bits, registers and primitive machine operations.
- Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is difficult.
- The main disadvantage of machine language coding is that all operations and operands must be specified in a numeric codes.
- So, we need a translators to modify the machine language program into human readable form.

Q.7. What is the difference between a compiler language and Interpreted language. Give examples of each. When would we prefer interpreter over compiler? **CT : W-11(7M)**

Ans.

Sr. No.	Compiler language	Interpreted language
(1)	In compiler language, the program is first converted into machine code and is then directly executed by CPU.	In this the program is first converted into some interpreted code and is then indirectly executed by interpreter program.
(2)	Reads the whole program called source code written in some high level language and translates code of a programming language in machine code also called as object code.	The interpreter takes one statement then translates it and executes and then takes another statement.
(3)	Compiler language program execution is fast since direct machine code is executed.	Interpreted language program execution is slower, because of indirect code execution.
(4)	The compiler translates the entire program and then executes it.	Interpreter stops translating after the first error.
(5)	Code produced is not portable since it produces machine code which is dependent of machine.	Interpreted code is portable since interpreter language program do not produce machine language program.
(6)	Example of source language are C, Pascal, Python, FORTRAN, LISP etc.	Scripting languages like Java script, ASP, JAVA, etc.
(7)	These are inferior compared to interpreted language with reference to program development environment.	These languages are superior to compiler language as it involves testing, debugging and editing a program which supports program development environment.
(8)	It is costlier as compared to interpreted program development environment.	Interpretation of program is cheaper as compared to compiler language.

CROSS COMPILER

Q.8. What is cross compiler? **CS : S-10,W-13(2M)**

OR Define cross compiler. **CS : S-13(2M)**

OR Write short note on cross compiler. **CT : W-12(3M)**

OR What is the theory and logic behind cross compilation? **CT : W-10(4M)**

Ans. Cross compiler :

- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
- A cross compiler is necessary to compile for multiple platforms from one machine.
- A cross compiler could generate an executable for each of them from one main source.
- Basically there exists three types of languages :

(i) Source language i.e. the application program.

(ii) Target language in which machine code is written.

(iii) The implementation language in which a compiler is written.

- There may be a case that all these three languages are different.
- In other words there may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called cross compiler. Thus by using cross compiler techniques platform independency can be achieved.
- For example, we have new language L and want to write cross compiler then S to generate code for machine N ; i.e. we create L_SN . If an existing compiler for S runs on machine M and generate code for M, it is characterized by S_MM .
- If L_SN run through S_MM , we get compiler L_MN i.e. compiler from L to N that runs on M.
- This process is illustrated in following Fig. by putting together T diagrams for these compilers.

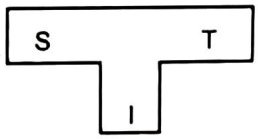


Fig.(a) T diagram with S as source, T as target and I as implementation language

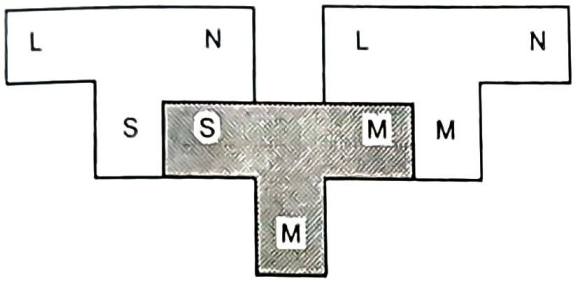


Fig.(b) Cross compiler

- The use of cross compiler makes it possible to overcome the lack of resources by creating an interrelated execution between various components on the system.

Q.9. What is cross assembler? What is the use of cross assembler?

CT : S-11(3M)

OR What is cross assembler? Explain the application of it.

CS : W-11(4M)

OR Explain cross assembler. **CS : S-12(2M)**

Ans. Cross assembler :

- An assembler is a program which creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents.
- An assembler converts assembly language (human readable text) into actual binary processor specific machine code (non human readable binary code).
- An assembler that generates machine language for a different type of computer than the one the assembler is running in.
- It is used to develop programs for computers on a chip or microprocessor used in specialized applications that are either too small or are otherwise incapable of handling the development software.

Use of cross assembler :

(1) Cross assembler are generally used to develop programs which are supposed to run on game consoles, applicances and other specialized small electronics system which are not able to run a development environment.

(2) They can also be used to speed up development for low powered system, for example XASm enables development on a PC based system for a Z80 powered MSX computer.

BOOTSTRAPPING

Q.10. Define bootstrapping. How bootstrapping can be achieved?

CT : W-09(4M), CS : S-13(2M)

OR Explain how bootstrapping is used in design of a compiler.

CS : S-10(3M), W-10(4M), W-12(6M), S-12(2M)

OR What is bootstrapping? **CS : W-13(2M)**

OR Explain with suitable example, what is bootstrapping.

CT : W-13(4M)

Ans. Bootstrapping of compiler :

- Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle far more complicated program.
- This complicated program can further handle even more complicated program and so on.
- Writing a compiler for any high level language is a complicated process.
- It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.
- To clearly understand the bootstrapping technique consider a following scenario :
- Suppose we want to write a cross compiler for new language X.
- The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create X_YZ.
- Now, if existing compiler Y [i.e. compiler written in language Y] runs on machine M and generates code for M then it is denoted as Y_MM. Now if we run X_YZ using Y_MM then we get a compiler X_MZ.
- That means a compiler for source language X that generates a target code in language Z and which runs on machine M.
- Following diagram illustrates the above scenario :

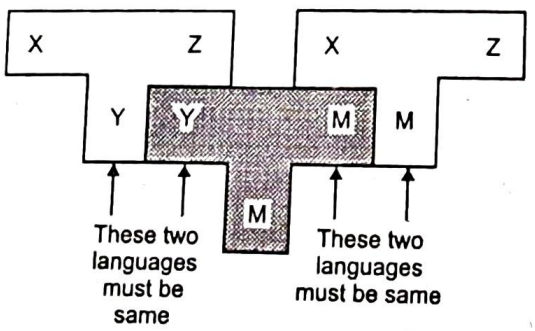
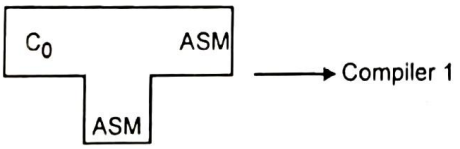


Fig. Bootstrapping process

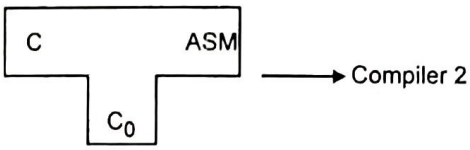
Example :

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

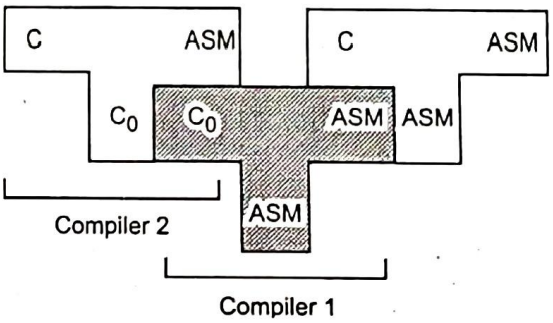
Step 1 : First we write a compiler for a small subset of C in assembly language.



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



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



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

PHASE OF COMPILER DESIGN

Q.11. Explain the analysis and synthesis model of compilation in brief. ***CT : W-13(5M)***

OR Explain the advantages of analysis synthesis model of compilation. ***CT : S-14(4M)***

Ans. The analysis synthesis model of compilation :

- There are two parts of compilation analysis and synthesis.
- 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. Of the two parts, synthesis requires the most specialized techniques.
- During analysis, the operations implied by the source program are determined recorded in a hierarchical structure called a tree.
- Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation.

For example, a syntax tree for an assignment statement is shown in following Fig.(a).

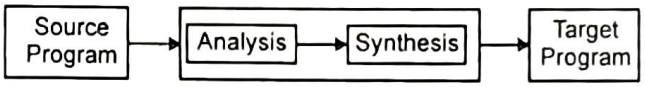


Fig.(a) Analysis and synthesis model

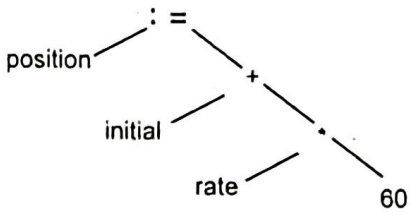


Fig.(b) Syntax tree for position : = initial + rate * 60.

- In compiling, analysis consist of three phases :

(1) **Linear analysis :** In this analysis, the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

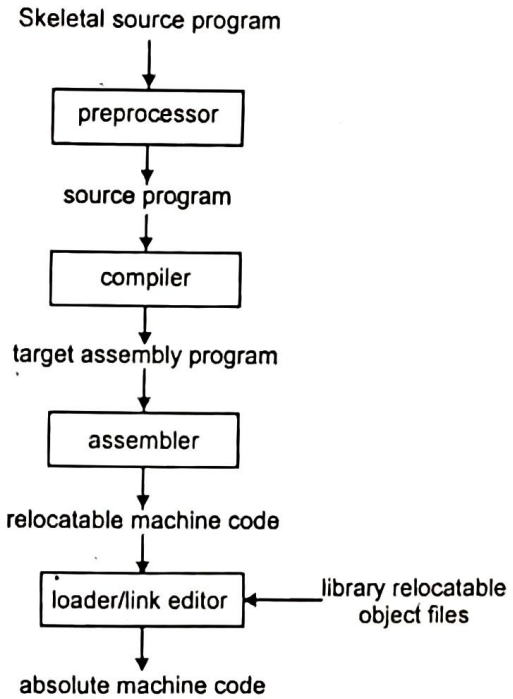


Fig.(c) A language processing system

(2) **Hierarchical analysis :** In this analysis, characters or tokens are grouped hierarchically into nested collections with collective meaning.

(3) **Semantic analysis :** In this analysis, certain checks are performed to ensure that the components of a program fit together meaningfully.

Advantages :

(i) Each phase has a well defined work.

(ii) Each phase handles a logical activity in the process of compilation.

(iii) Source and machine independent code optimization is possible.

(iv) Due to optimization phase front and back end phases can be deployed.

(v) Using this phases, compiler is retargetable i.e. compiler can be easily modified for another target machine or source language.

Q.12. Explain various phases of compiler in detail.

CS : S-11,12(8M), W-12(4M)

OR What are the different phases of compiler? Explain any one in detail. ***CS : W-11(6M)***

OR What are the various stages in the process of generation of target code from the source code by the compiler? How these stages are broadly classified? Explain each. ***CT : S-14(10M)***

OR Explain each phase of compiler. Also discuss input required and output produced by each phase of compiler with suitable example. ***CS : W-09(6M)***

OR Explain complete compilation process for high level language program. ***CT : W-10(4M)***

OR Draw and explain general structure of compiler design. ***CS : S-14(6M)***

Ans. Phases of compiler :

- The process of compilation is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step.
- For this reason, it is customary to partition the compilation process into a series of sub-processes called phases as shown in Fig.(a).

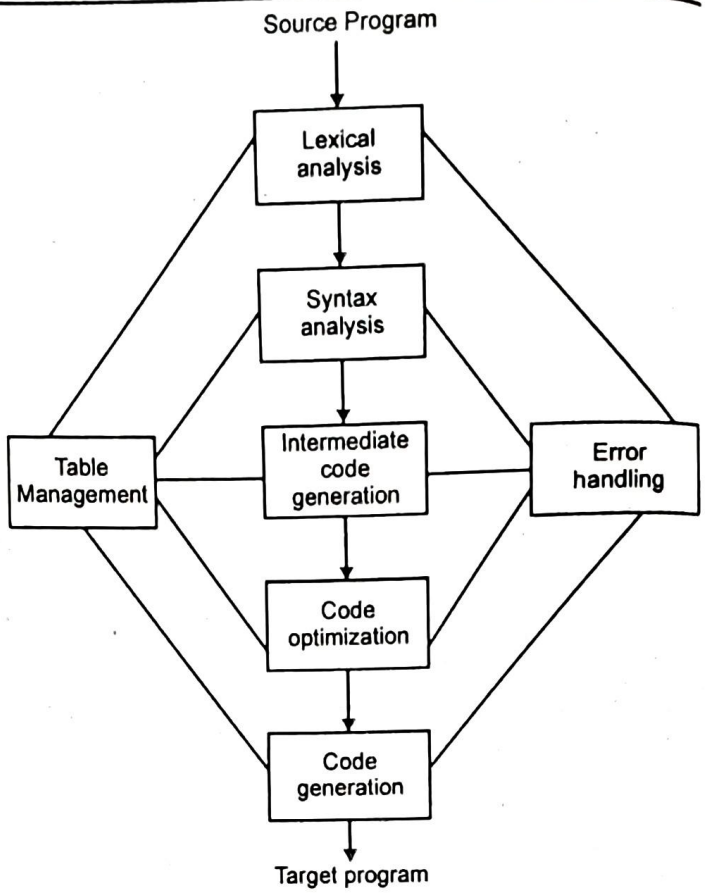


Fig.(a) Phases of compiler

(1) Lexical analysis :

- The lexical analysis is the interface between the source program and the compiler.
- The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of characters that can be treated as a single logical entity.
- Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. For example, in the FORTRAN statement.

  IF (5 . EQ. MAX); GOTO 100 .....('a')

  We find the following eight tokens :

  IF ; (; 5 ; EQ. ; MAX ;) GOTO ; 100
- What is called a token depends on the language at hand and to some extent on the discretion of the compiler designer, but in general each token is a substring of the source program that is to be treated as a single unit.
- There are two kinds of token; specific strings such as if or a semicolon and classes of strings such as identifiers, constants or labels.

- To handle both types of tokens, we shall treat a token as a pair consisting of two parts: a token type and a token value.
- A token such as the identifier MAX., above, has a type "identifier" and a value consisting of the string MAX.

Finding tokens :

- To find the next token, the lexical analyser examines successive characters in the source program, starting from the first character not yet grouped into a token.
- The lexical analyser may be required to search many characters beyond the next token in order to determine what the next token actually is.
- For example, Suppose the lexical analyser has last isolated the left parenthesis as a token in statement ('a'). We may represent the situation as follows:

IF (5 . EQ . MAX) GOTO 100

- Note that blanks have been removed, since they are ignored in FORTRAN.
- When the parser asks for the next token, the lexical analyzer reads all the characters between 5 and Q, inclusive, to determine the next token is just the constant 5.
- The reason it has to scan as far as it does is that until it sees the Q; it is not sure it has seen the complete constant ; it could be working on a floating-point constant such as 5-E-10.
- After determining that the next token is the constant 5, the lexical analyzer repositions its input pointer at the first dot, the character following the token.
- The lexical analyzer may return token type "constant" to the parser, and the value associated with this "constant" could be the numerical value 5 or a pointer to the string 5.
- When statement ('a') is completely processed by the lexical analyzer, the token stream might look like :

If ([const, 341] eq [id, 729] goto [label, 554])

- The relevant entries of the symbol table are suggested in Fig.

341	Constant, integer, value = 5
554	Label, value = 100
729	Variable, integer, value = MAX

Fig. (b) Symbol table

(2) Syntax analysis :

- The parser has two functions.
- It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language.
- It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.
- For example, if a PL/I program contains the expression :

A + / B

Then after lexical analysis, this expression might appear to the syntax analyzer as the token sequence.

id + / id

- On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formation rules of a PL/I expression.
- The second aspect of syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped together.
- For example, the expression

A / B * C

has two possible interpretations :

(a) Divide A by B and then multiply by C ; or

(b) Multiply B by C and then use the result to divide A.

- Each of these two interpretations can be represented in terms of a parse tree, a diagram which exhibits the syntactic structure of the expression.
- Parse tree that reflects orders (a) and (b) are shown in Fig. (c) and (d), respectively.

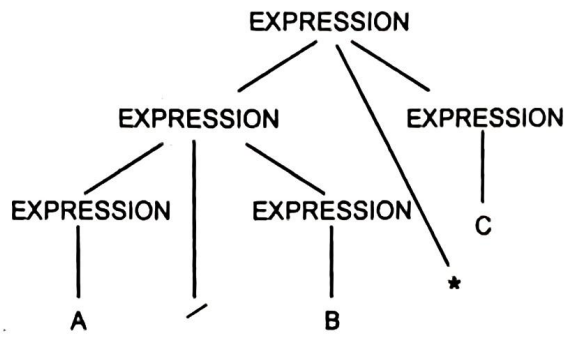


Fig.(c)

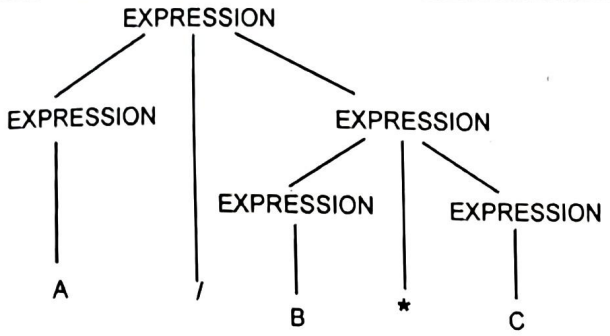


Fig.(d)

(3) Intermediate code generation :

- The output of the syntax analyzer is some representation of a parse tree.
- The intermediate code generation of a phase transforms this parse tree into an intermediate - language representation of the source program.

Three-Address Code :

- A typical three- address code statement is A = B op C

  where A, B and C are operands and op is binary operator.
- The parse tree in Fig. (c) might be converted into the three address code sequence.

  $T_1 := A / B$

  $T_2 := T_1 * C$

  where, T_1 and T_2 are names of temporary variables.
- In addition to statements that use arithmetic operators, an intermediate-language needs unconditional and simple conditional branching statements, in which at most one relation is tested to determine, whether or not a branch is to be made.
- Higher-level flow of control statements such as while-do statements, or if then-else statements, are translated into these lower-level conditional three- address statements.
- For example, Consider the following while- statement

  While A > B & A <= 2 * B – 5 do

  A : = A + B

  A straight forward algorithm for translation would produce intermediate code like that shown below :

  L_1 : if A > B goto L_2

  goto L_3

  L_2 : $T_1 := 2 * B$

  $T_2 := T_1 - 5 \rightarrow$

  if $A \leq T_2$ goto L_4

  goto L_3

  L_4 : A:= A + B

  goto L_1

  L_3 : Exit

(4) Optimization :

- Object programs that are frequently executed should be fast and small.
- Certain compilers have within them a optimization phase that tries to apply transformations to the output of the intermediate code generator, in an attempt to produce an intermediate-language version of the source program from which a faster or smaller object-language program can ultimately be produced.
- Optimizing compilers merely attempt to produce a better target program than would be produced with no "optimization".
- A good optimizing compiler can improve the target program by perhaps a factor of two in overall speed, in comparison with a compiler that generates code carefully but without using the specialized techniques generally referred to as code optimization.
- There are two ways of optimization :

  (1) Local optimization

  (2) Loop optimization.

(5) Code Generation :

- The code generation phase converts the intermediate code into a sequence of machine instructions.
- A simple minded code generator might map the statement A := B + C into the machine code sequence.

  LOAD B

  ADD C

  STORE A
- However, such a straight-forward macro-like expansion of intermediate code into machine code usually produces a target program that contains many redundant loads and stores and that utilizes the resources of the target machine inefficiently.
- To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of the registers.
- Knowing what quantities reside in registers, the code generator can generate loads and stores only when necessary.

- Many computers have only a few high-speed registers in which computations can be performed particularly quickly.
- A good code generator would therefore attempt to utilize these registers as efficiently as possible. This aspect of code generation is called register allocation.

(6) **Table management :**

- The table management, or book-keeping, portion of the compiler keep track of the names used by the program and records essential information about each, such as its type.
- The data structure used to record this information is called a symbol table.

(7) **Error handling :**

- One of the most important functions of compiler is the detection and reporting of errors in the source program.
- The error messages should allow the programmes to determine exactly whereby virtually all the phases of a compiler.
- A phase of the compiler discover an error; it must report the error to the error handler which issues an appropriate diagnostic message.

Q.13. What is front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into two distinct stages?

CT : W-11(6M),S-12(4M)

OR Explain the structure of typical two pass compiler. Explain each component in detail. **CT : S-10(7M)**

Ans.

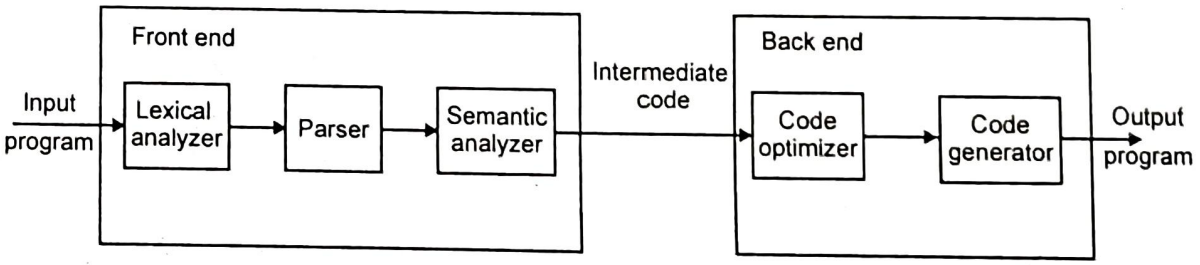


Fig. Front end back end model

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

(1) By keeping the same front end and attaching different back ends one can produce a compiler for same source language on different machines.

(2) By keeping different front ends and same back end one can compile several different languages on the same machine.

Q.14. Discuss the factors which affect number of passes in compiler design. **CS : W-10(2M)**

Ans.

- Several phases of compiler are grouped into one pass. Thus pass of compiler is simply collection of various phases.
- For example, Lexical analysis, syntax analysis and intermediate code generation is grouped together and forms a single pass.
- Various factors affecting number of passes in compiler are :

  (i) Forward reference

  (ii) Storage limitation

  (iii) Optimization
- The compiler can require more than one passes to complete subsequent information.

Q.15. How can you differentiate between a phase and pass?

CS : W-10(3M)

OR What is difference between a phases and a pass of a compiler?

CT : S-12(4M)

Ans.

- In compilers, the process of compilation can be carried out with the help of various phases such as lexical analysis, syntax analysis, intermediate code generation, code generation and code optimization.
- Whereas in case of passes, various phases are combined together to form a single pass. An input program can be compiled using a single pass or using two passes.
- Different groups of phase form corresponding front end and back ends of the compilers. The advantage of front-end and back-end model of compiler is that : same source language can be compiled on different machines or several different languages can be compiled on the same machine.
- Compiling a source program into single pass is difficult because of forward reference that may occur in the program.
- Similarly if we group all the phases into a single pass then we may be forced to keep the entire program in the memory.
- And then memory requirement may be large. On the other hand it is desirable to have relatively few passes, because it becomes time consuming to read and write intermediate files.

Q.16. Discuss how the number of passes of compiler can be reduced.

Ans.

- It is desirable to have relatively few passes from the point of view of reducing the compilation time. And for reducing the number of passes, it is required to group several phases in one pass.
- For some of the passes, grouping into one pass is not a major problem.
- For example, The lexical analyzer and syntax analyzer, because the interphase between them is single token i.e. the processing required to be done with token is independent of other token.
- Therefore, these phases can be easily grouped together with lexical analyzer working as subroutine for syntax analyzer who is the in-charge of entire analysis activity.
- Whereas grouping of some of the phases in one pass is not easy, e.g., grouping of intermediate code generation and target code generation is not easy because it is often very hard to perform code generation until the intermediate code has been generated because here the interface between the two is not in terms of only one intermediate instruction.
- This is because certain language permits use of variables before their declaration. Similarly, many languages permit forward jumps also.
- Therefore, it is not possible to generate target code for a construct until complete instruction is available to overcome this problem so as to enable the merging of intermediate code generation and target.

Q.17. What are the various compiler writing tools? Explain each in brief. **CT : S-09(5M), CS : W-11(4M), S-14(7M)**

OR Explain in detail various compiler writing tools.

CT : S-10,13(6M)

OR Which tools are available to write a compiler? **CS : S-13(3M)**

OR Explain the useful compiler-construction tools.

Ans. Compiler-writing tools :

- The compiler writer, like any programmer, can profitably use software tools such as debuggers, version managers, profilers and so on.
- In addition to these software-development tools, other more specialized tools have been developed for implementation of various phases of a compiler.
- Some general tools have been created for the automatic design of specific compiler components. These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of a compiler.
- The following is a list of some useful compiler-construction tools :

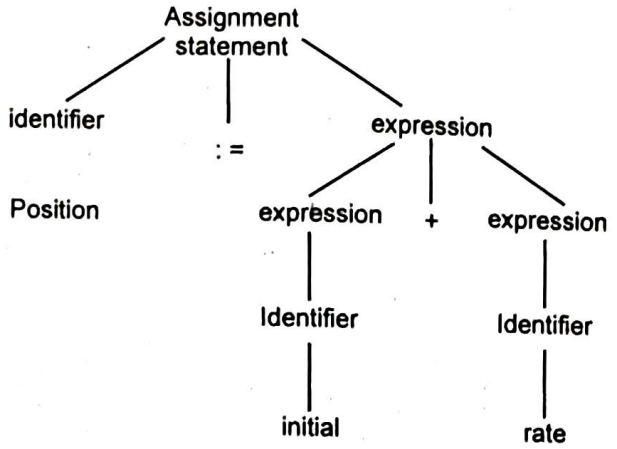
(1) Parser generators :

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

(2) **Scanner generators :** These automatically generate lexical analyzers, normally from a specification based on regular expression.

(3) **Syntax-directed translation engines :**

- These produce collection of routines that walk the parse tree, such as figure generating intermediate code.
- The basic idea is that one or more "translations" are associated with each node of the parse tree and each translation is defined in terms of translations at its neighbour nodes in the tree.



4) **Automatic code generators :**

Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.

The rules must include sufficient detail that we can handle the different possible access method for data : e.g., variables may be in registers, in a fixed (static) location in memory or may be allocated a position on a stack.

The basic technique is "template matching". The intermediate code statements are replaced by "templates" that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template.

(5) **Data-flow engines :**

- Much of the information needed to perform good code optimization involves "data-flow analysis", the gathering of information about how values are transmitted from one part of program to each other part.
- Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

Q.18. Why design of a compiler is complex than that of an assembler?

CS : S-09(3M), CT : S-09(2M)

OR Compiler is very complex to design than assembler. Justify..

CS : W-09(4M), CT : W-09(3M)

Ans.

- Compiler takes as its source code a C program i.e. a file of ASCII characters.
- It produces either an assembly language program, as an intermediate step or else machine code directly.
- The assembler takes as its source code an assembly language programme i.e. also final of ASCII character s, it used this to produce machine code.
- A compiler is definitely complex to write than assemblers. An assembler, in essence, is just a table that maps short strings (instruction mnemonics) to numbers (opcodes).
- The harder part of writing an assembler is just creating the table efficiently.
- A compiler on other hand, needs a much more complex passer (depending on the language) and needs to manage stack positions and register usage in arbitary circumstances,

Q.19. Explain the output of lexical analysis phase. Give the importance of this output in next phase of compiler. *CS : W-14(3M)*

Ans.

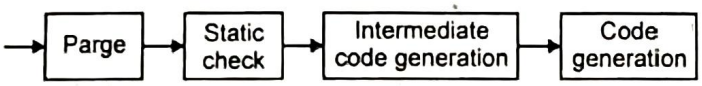
- The lexical analysis is the first phase which reads the character from the source program and group them into the stream of token.
- Token is logically cohesive sequence of characters such as identifiers, operators, keywords etc.
- A sequence forming a token is called lexeme.
- The output of lexical analyzer is a stream of token which is passed to the next phase i.e. syntax analysis.
- The syntax analyzer groups token together into syntactic structure.
- In next phase after lexical analysis, the compiler verifies, whether the tokens generated by lexical analyzer are grouped according the syntactic rules of the language or not.
- If they are grouped according to rules, then group of token generated by the lexical analyzer is a accepted or the valid construct of the language.

- The syntactic structure can be regarded as a tree whose leaves are the token.
- The interior nodes of the tree shows the string of token that logically belong together.

Q.20. What would happen if code generation phase is implemented after intermediate code generation?

Ans.

- Code generation phase generate the target code taking input as intermediate code generation phase. The output of intermediate code generation may be given directly to code generation or may pass through code optimization before generating code.
- Intermediate code is generated using the parse rule producing a language from the input language. Intermediate code has following property :
- Simple enough to be translated to assembly code.
- Complex enough to capture the complication of high level language.



- The input to code generation consist of intermediate representation of source program produced by front end, together with information in symbol table.
- It is used to determine run time address of data objects denoted by names in intermediate representation.
- So if code generation is implemented after intermediate code generation the intermediate code representation is not possible.
- So error removal process is not so efficient, so require more time for compilation.

Q.21. What is role of symbol table for each phase of compiler?

CT : W-12(6M)

Ans.

- An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.
- These attributes may provide information about the storage allocated for an identifier, its type, its scope and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument and the type returned if any.
- A symbol table is a data structure containing the record of each identifier, with field for the attributes of identifier the data structure allows us to find the record for each identifier quickly and store or retrieve data from that record quickly.
- When ab in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.
- However, the attributes of an identifier cannot normally be determined during lexical analysis.
- For example, in a pascal declaration like VAR position, initial, rate : real ; the type real is not known when position, initial and rate are seen by the lexical analyzer.
- The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.
- For example, when doing semantic analysis and intermediate code generation, we need to know what the types of identifiers are, and so that we can check that the source program uses them in valid ways and so that we can generate the proper operations on them.
- The code generator typically enters and uses detained information about the storage assigned to identifiers.

Q.22. What are the errors that may encounter in each phase of a compiler? ***CS : S-10,12(5M)***

OR What kinds of errors may be generated by different phases of compiler?

Ans.

- The program contain errors at many different levels. For example error can be :

(1) Lexical, such as mis-spelling an identifier, keyword or operator.

(2) Syntactic, such as an arithmetic expression with unbalanced parentheses.

(3) Semantic, such as an operator applied to an incompatible operand.

(4) Logical, such as an infinitely recursive call.

- Often much of the error detection and recovery in a compiler is centered around the syntax analysis phase.
- One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analysis disobeys the grammatical rules defining the programming language.
- Another is the precision of modern parsing methods they can direct the presence of syntactic errors in programs very efficiently.

- Accurately detecting the presence of syntactic errors in program very efficiently. Accurately detecting the presence of semantic and logical errors at compile time is a much more difficult task.

DESIGN OF LEXICAL ANAYZER

Q.23. Explain the role of lexical analyzer in compiler designing.

OR Explain the block schematic of lexical analysis.

OR Discuss the issues in design of a lexical analyzer.

CS : W-10,13(4M)

Ans.

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

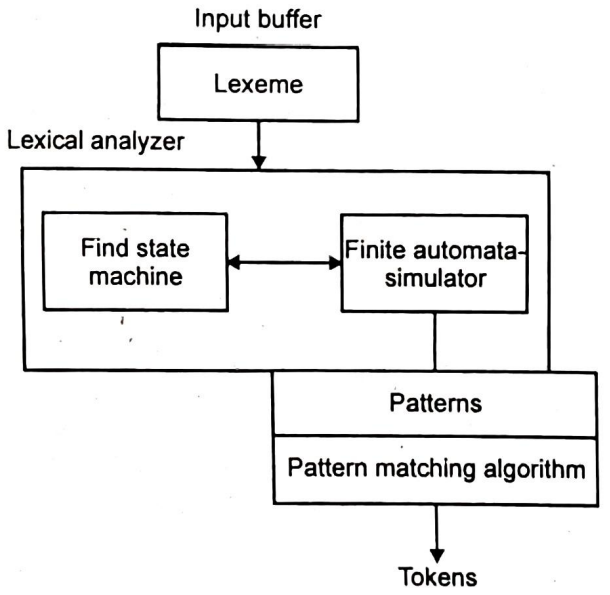


Fig.(a) Block schematic of lexical analyzer

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

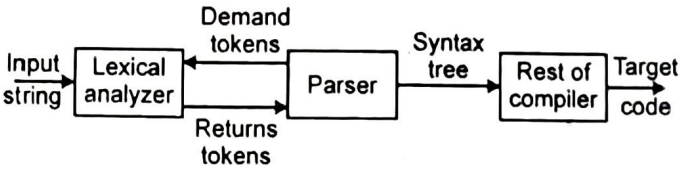


Fig.(b) Role of lexical analyzer

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

Functions of lexical analyzer :

(1) It produces stream of tokens.

(2) It eliminates blank and comments.

(3) It generates symbol table which stores the information about identifiers, constants encountered in the input.

(4) It keeps track of line numbers.

(5) It reports the error encountered while generating the tokens.

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

Issues in lexical analysis :

There are several reason for separating the analysis phase of compiling into lexical analysis and parsing :

(i) Simpler design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one or other of these phases.

(ii) Compiler efficiency is improved.

(iii) Compiler portability is enhanced.

(iv) Identifying the tokens of the language for which lexical analyzer is to be built and to specify these tokens by using suitable notations.

Q.24. What is the role of input buffering scheme in lexical analyzer?

Ans.

- The lexical analyzer scans be characters of source program one by one to find the 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 times 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.
- An important technique that uses two input buffers that are reloaded alternately is shown in Fig.

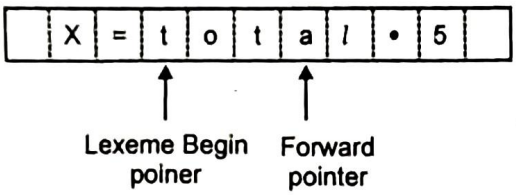


Fig. Input buffer

- Each buffer is of the same size N, where N is the size of a disk block, for example 1024 bytes. Thus, instead of one character, N characters can be read at a time.
- The pointers used in the input buffer for recognizing the lexeme are as follows :

(i) Pointer lexemeBegin points the beginning of the current lexeme being discovered.

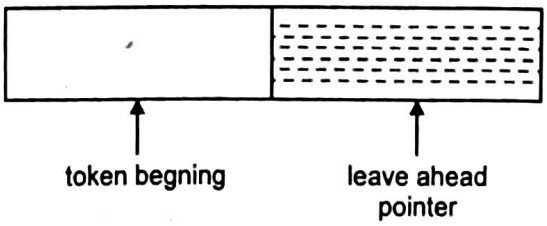
(ii) Pointer forward scans ahead until a pattern match is found for lexeme.

- Initially, both pointers point to the first character of the next lexeme to be found.
- The forward pointer is scanned ahead until a match for a pattern is found.
- After the lexeme is processed, both the pointers are set to the character following that processed lexeme.
- For example, in Fig. the lexemeBegin 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.

Q.25. Why lexical analyzer reads few characters beyond the token in advance before declaring validity of token? Explain with example. *CS : W-13(5M)*

Ans.

- The logical analyzer may be required to search many characters and the next token because sometimes logical analyzer is not sure that it has been complete token until it scans many characters beyond.



- If (5. EQ. MAX) GOTO 100.
- The string to left of arrow represents token by logical analyzer.
- When parser ask for next token logical analyzer read all character between 5 and a inclusive to determine that next token is just constant 5.

SCANNER GENERATOR : LEX, FLEX

Q.26. Write short note on scanner generator.

Ans. Scanner generator :

- These automatically generated lexical analyzer, normally form a specification based on regular expression.
- Several tools has been built for constructing lexical analyzer from special purpose notations based on regular expression. eg. lex tool.
- Lex compiler is input specification as the lex language.
- Lex like specification can be used even if a lex compiler is not available the specification can be manually transcribed into working program using the transition diagram techniques.
- A specification of a lexical analyzer is prepared by creating a program lex l in the lex language.
- Then lex. l is run through the lex compiler to produce a C program lex yy. C.
- The program lex.yy.C consist of a tabular representation of transition diagram constructed from regular expression of lex. l.

Q.27. Write a short note on LEX. *CS : W-14(2M)*

OR Explain Lex in details with the help of example.

OR Explain Lex.

OR What is LEX? *CS : S-10,11(4M), CT : W-09(2M)*

Ans. LEX :

- Lex is a program generator designed for lexical processing of character input streams.

- It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.
- Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages".
- Lex can write code in different host languages. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users.
- Lex turns the user's expressions and actions into the host general-purpose language; the generated program is named yylex.
- The yylex program will recognize expressions in a stream and perform the specified actions for each expression as it is detected.

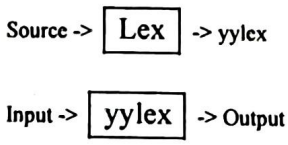


Fig. An overview of Lex

- Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex.
- Lex generates a deterministic finite automata from the regular expressions in the source.
- Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.
- Lex source is a table of regular expressions and corresponding program fragments.
- The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions.
- As each such string is recognized the corresponding program fragment is executed.
- The recognition of the expressions is performed by a deterministic finite automata generated by Lex.
- The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

Structure of a Lex :

The structure of a Lex file is intentionally similar to that of a YACC file; files are divided into three sections, separated by lines, as follows :

(1) **Definition Section :** The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

(2) **Rules Section :** The rules section associates regular expression patterns with C statements. When the lexer identifies text in the input matching a given pattern, it will execute the associated C code.

(3) **C code section :** The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Q.28. Describe a tool for study of Lex.

OR Explain any one tool for writing lexical analyzer. *CS : S-13(3M)*

Ans.

- Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions.
- A tool, called Lex that has been widely used to specify lexical analyzers for a variety of languages, refer to the tool as Lex compiler, and its input specification as the Lex language.
- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l, together with a standard routine that uses the table to recognize lexemes.
- The actions associated with regular expression in lex.l are pieces of C code and are carried over directly to lex.yy.c. Finally lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

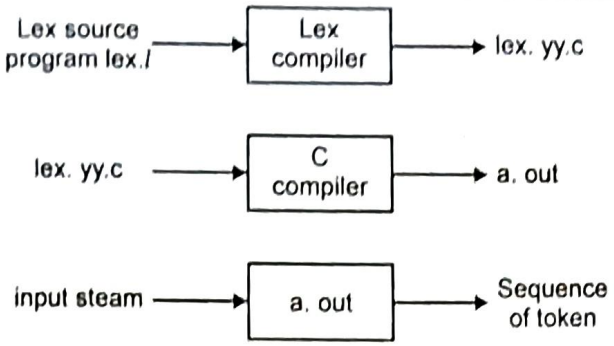


Fig. Creating a lexical analyzer with lex

Q.29. Give general specification of LEX program of its compiling process. **CT : W-09(2M)**

OR Give specification of LEX program. **CS : S-10(2M)**

OR Explain in brief, the format of LEX input file. **CT : S-10(3M)**

OR Explain structure of LEX. **CS : S-11(2M)**

OR Write a LEX program to produce a lexical analyzer that recognizes identifiers. **CT : S-11(6M)**

Ans.

- LPDT (Language Processor Development Tools) generates the analysis phase of a language processor whose source language is L.
- The LPDT requires the following two inputs :

(1) Specification of a grammar of language L.

(2) Specification of semantic actions to be performed in the analysis phase.

- It generates programs that perform lexical, syntax and semantic analysis of the source program and construct the IR.
- These programs collectively form the analysis phase of the language processor.
- These are the lexical analyzer generator LEX and the parse generator YACC.
- The input to these tools is a specification of the lexical and syntactic constructs of L, and the semantic actions to be performed on recognizing the constructs.
- The specification consists of a set of translation rules of the form

  <string specification> {<semantic action>}

  where <semantic action> consists of C code.
- This code is executed when a string matching <string specification> is encountered in the input.
- LEX and YACC generate C programs which contain the code for scanning and parsing, respectively, and the semantic actions contained in the specification.
- Figure shows a schematic for developing the analysis phase of a compiler or language L using LEX and YACC.

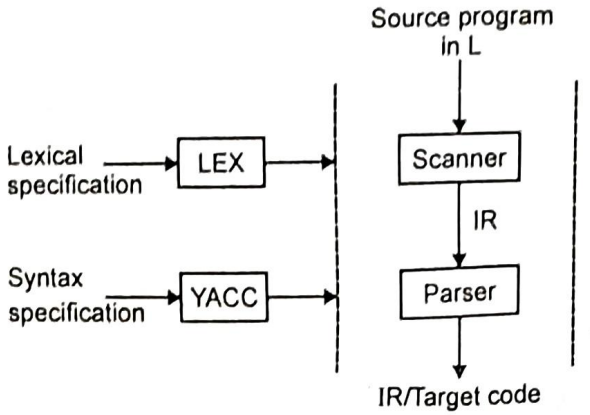


Fig.(a) Using LEX and YACC

LEX :

- LEX accepts an input specification which consists of two components.
- The first component is a specification of strings representing the lexical units in L, e.g. id's and constants.
- This specification is in the form of regular expression.
- The second component is a specification of semantic actions aimed at building an IR.

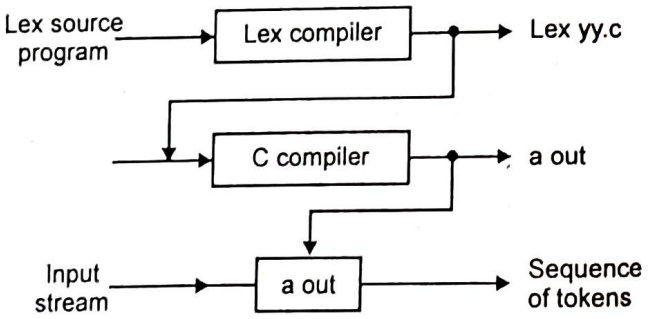


Fig.(b) Creating a lexical analyser with lex

- LEX has been widely used to specify lexical analyzer for variety of languages. Tool has Lex compiler so its input specification has Lex language.
- Lex is a software that takes as input specification of a set of regular expression together with actions to be taken.
- The output of Lex is a program that recognizes the regular expression and acts appropriately on each.

Structure of Lex Programs :

The general form of a lex program like lex.l is

declarations

%%

translation rules

%%

auxiliary functions

lx program to recognize identifier name, integer constant, floating point constant and arithmetic operator is as follows :

```
%{
/*Literal block*/
#include "scan.h" /*Scanner header file*/
int line = 1; /*Current line number*/
int pos = 0; /*Start position of token*/
int epos = 1; /*End position of token*/
OpType op; /*Last operator scanned*/
void lex_err(char *, char *); /*Reports lexical errors*/
%}
/*Regular definitions*/
letter          [A-Za-z]/*letter can be underscore*/
digit           [0-9]
blank_str       [ \t]+
identifier      ({letter} ({letter} | {digit})*
int_literal     {digit}+str_literal \"([^\"\n])*\"
/*Regular definitions to match invalid tokens*/
open_string     \"({^\"\n])*$
%% /*Second section*/
\n {line++; epos = 1;}
{blank_str}  {epos +=yyleng;}
[Ii][Ff]  {pos = epos; epos += 2; return IF_TK;}
[Tt][Hh][Ee][Nn] {pos = epos; epos +=4; return THEN_TK;}
[Ee][Ll][Ss][Ee] {pos = epos; epos +=4; return ELSE_TK;}
[Ee][Nn][Dd] {pos = epos; epos +=3; return
END_TK;}
[Ww][Hh][Ii][Ll][Ee] {pos = epos; epos +=5;
return WHILE_TK;}
[Dd][Oo]        {pos = epos; epos +=2; return
DO_TK;}
[Rr][Ee][Aa][Dd]  {pos = epos; epos +=4; return READ_TK;}
[Ww][Rr][Ii][Tt][Ee] {pos = epos; epos +=5; return WRITE_TK;}
{identifier}  {pos = epos; epos += yyleng; return ID;}
{str_literal}  {pos = epos; epos += yylen; return STRLIT;}
{int_literal}  {pos = epos; epos += yyleng; return INTLIT;}
/*Regular definition to identify arithmetic operator*/
"+" {op = ADD; pos = epos; epos += 1; return ADDOP;}
"-" {op = SUB; pos = epos; epos += 1; return ADDOP;}
"*" {op = MUL; pos = epos; epos += 1; return MULOP;}
"/" {op = DIV; pos = epos; epos += 1; return MULOP;}
"=" {op = EQ; pos = epos; epos += 1; return RELOP;}
"<>" {op = NE; pos = epos; epos += 2; return RELOP;}
"<" {op = LT; pos = epos; epos += 1; return RELOP;}
"<=" {op = LE; pos = epos; epos += 2; return RELOP;}
">" {op = GT; pos = epos; epos += 1; return RELOP;}
">=" {op = GE; pos = epos; epos += 2; return RELOP;}
":=" {op = ASGN; pos = epos; epos += 2; return ASGNOP;}
"," {pos = epos; epos += 1; return ' , ';}
";" {pos = epos; epos += 1; return ' ; ';}
"(" {pos = epos; epos += 1; return ' ( ';}
")" {pos = epos; epos += 1; return ' ) ';}
/*Regular definition to indentify Unsigned integer or floating point
numbers*/
        digit → [0-9]
        digits → digit
        number → digits (. digits)?(E+-]?digits)?
}
```

Q.30. Explain the regular expressions in LEX.

Ans. Regular expressions in LEX :

- In Lex, the regular expression is a pattern description using a meta language, a language that we use to describe particular patterns of interest.
- The characters used in this meta language are part of the standard ASCII character list.
- The characters that form regular expression are :

Sr. No.	Character	Description
(1)	.	matches any single ASCII character except the new line character ("\n").
(2)	*	matches zero or more copies of the preceding expression or character.
(3)	[ ]	This is to represent the character class that matches any character within the brackets.
(4)	^	If any character set or expression comes just after the operator, it will accept all the characters except the one within the character class, that is, this character can be used to negate any character class within the square brackets .
(5)	-	A dash inside the square brackets indicates a character range, for example [0 – 9] means the same thing as [0123456789] and [A – Z] means all the English alphabet letters A to Z.
(6)	{ }	It indicates how many times the previous pattern is allowed to match which contains one or two number. For example, A {1,3} matches one to three occurrences of the letter A.
(7)	\	It is used to escape meta-characters and as part of the usual C escape sequence, for example, "\n" is a new line character, while "*" is literal asterisk.
(8)	+	Matches one or more characters of the preceding regular expressions or a character set. For example, [0 – 9] + matches one or more combinations of the character 0 to 9. This is, this regular expression accepts the strings 0112, 1231456, or 9012, but not an empty string.
(9)	?	It matches zero or one occurrence of the preceding regular expression, for example, – [0 – 9] + matches a signed number including an optional leading minus.
(10)	|	Matches any one of the preceding or following regular expressions, for example, pen/pencil/eraser. This expression will match any one of the three words.
(11)	"..."	It interprets everything within the equation marks as meta-characters other than the C escape sequences.
(12)	()	Groups a series of regular expressions together into a new expression. For example, (10101) represent a character sequence 10101.
(13)	<>	A name or a list of names in angle brackets at the beginning of a pattern makes that pattern apply only in the given start states.
(14)	<<EOF>>	In Flex, the special pattern matches the end of the file.

Q.31. Write regular expression for signed integer constant.

CS : S-11(2M)

Ans. Regular expression for signed integer constant :

The exponential form of real number is a decimal or signed integer followed by E and signed integer i.e.,

Exponential → (decimal / signed integer) E

signed integer

Sign → + / – / E

Digit → 0 / 1 / ..... / 8 / 9

Integer → digit (digit) *

Floating → integer / E

signed - integer → Sign integer

Decimal → Sign - integer floating / signal integer.

Q.32. What is FLEX?

OR Explain FLEX in details.

Ans. FLEX :

- Flex is a fast lexical analyzer generator tool for programming that recognizes lexical patterns in the input with the help of Flex specifications.
- Flex specification contains two parts :

(i) patterns and (ii) corresponding action.

- When we write a flex specification, we create a set of patterns which the lexer matches against the input.
- Each time one of the patterns matches, the corresponding action part is invoked (which is a C code). In this way, a Lex program divides the input into tokens.
- Flex itself does not produce an executable program; instead, it translates the Lex specifications into a file containing a C subroutine called yylex().
- More precisely, all the rules in the rules section will automatically be converted into the C statements by the Flex tool and will be put under the function name of yylex(). That is, whenever we call the function yylex, C statements corresponding to the rules will be executed.
- That is, we call yylex() to run the lexer. The generated lexical analyzer, by default it is lex.yy.c, can be compiled using regular C compiler along with any other files and Flex libraries.
- The program that we write in a Lex program contains the Lex specification and other C statements and subroutines. This file is named with an extension <filename>. *l* (for example, file.*l*).
- When this Lex program is passed to the Flex, it translates the <filename>. *l* into a file named lex.yy.c, which is a C program. Fig. shows the phases of a Lexical analyzer.

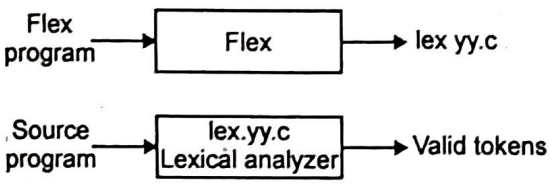


Fig. Phases of a lexical analyzer.

- The file lex.yy.c is also called the lexer or lexical analyzer.
- This C program is compiled with a normal C compiler and it produces an executable program. This can be executed like a normal C executable file.

Q.33. Give the structure of FLEX program.

Ans. Structure of FLEX program :

- Any Flex program consists of three sections separated by a line with just %% in it.

Definition section

%%

Rules section

%%

User code (Auxiliary) section

The basic structure of a Lex program. It consists of three sections.

(i) A definition section

(ii) A rules section

(iii) A user defined section

- The first section, i.e. the definition section contains different user defined Lex options used by the lexer. It also creates an environment for the execution of the Lex program.
- The definition section helps to create an atmosphere in two areas. First, it creates an environment for the lexer, which is a code.
- This area of the Lex specification is separated by " % {" and " %}", and it contains C statements, such as global declarations, commands, including library files and other declarations, which will be copied to the lexical analyzer (i.e. lex.yy.c) when it is passed through the Flex tool.
- In other words, Flex copies all the statements in this C declaration section bracketed by "% {" and " %}" to the lexical analyzer file, which is lex.yy.c.
- Secondly, the definition section provides an environment for the Flex tool to convert the Lex specifications correctly and efficiently to a lexical analyzer.
- This section mainly contains declarations of simple name definitions to simplify the scanner specifications and declarations of start condition. The statements in this section will help the Lex rules to run efficiently.
- The second section of any Lex program is the rules section that contains the patterns and actions that specify the Lex specifications.
- A pattern is in the form of a regular expression to match the largest possible string. Once the pattern is matched, the corresponding action part is invoked.
- The action part contains normal C language statements(s). They are enclosed within braces (i.e. "{" and "}"), if there is more than one statement, to make these component statements into a single block of statement.
- The Lex tools take everything after the pattern to be the action, while others only read the first statement on the line and silently ignore other statements. Always use braces to make the code clear, if the action has more than one statement or more than one line large.

- The lexer always tries to match the largest possible string, but when there are two possible rules that match the same length, the lexer uses the first rule in the Lex specification to invoke its corresponding action.
- The third and the final section of the Lex program is the user defined or user subroutine section. It is also known as auxiliary section.
- This section contains any valid C code. Flex copies the contents of this section into the generated lexical analyzer as it is. The simplest Flex program is :

  % %

  which generates a scanner that simply copies its input (one character at a time) to its output.

PROGRAMMING LANGUAGE GRAMMAR

Q.34. What are the elements to be considered while designing a new language processor?

OR Discuss salient features of a good programming language.

OR Write down reasons to explain why programmers prefer one language over another.

OR Give the features of High-level programming language.

OR What are the features of programming language?

Ans.

- A programming language is a notation with which people can communicate algorithms to computers and to one another. Hundreds of programming languages exist. They differ in their degree of closeness to natural or mathematical language one hand and to machine language on the other. They differ also in the type of problem for which they are best suited.
- Some of the aspects of high-level languages which make them preferable to machine or assembly languages are the following :

(1) **Ease of understanding :**

- A high-level language program is generally easier to read, write and prove correct than is an assembly- language program because a high-level language usually provides a more natural notation for describing algorithms than does assembly language.
- Even high level languages, some are easy to use than others.
- A good programming language should provide features for modular design of easy to understand programs. A good language should also enable control flow to be specified in a clean understandable manner.

(2) **Naturalness :** Much of the understandability of high level programming language comes from the ease with which one can express an algorithm in that language.

(3) **Portability :** Users must often be able to run their programs on a variety of machines. Languages such as FORTRAN or COBOL have relatively well-defined "standard version", and programs conforming to the standard should run on any machine.

(4) **Efficiency of use :**

- This area cover a number of aspects of both program and language design. One would like to be able to translate source program into efficient object code. One would also like the compilation process itself to be efficient.
- It is often more important that the programmer be able to implemented programs in a way that makes efficient use of this time.
- A high level programming language should have facilities for defining data structures, macros, subroutines, etc.
- Particular high-level language features facilitate reliable programming. Some such features are :

(a) Data structures.

(b) Scope rules that allow modifications to be made to a piece of a program without unwittingly affecting other portions of the same program.

(c) Flow-of-control constructs that clearly identify the looping structures in a program.

(d) Subroutines, allowing a program to be designed in small pieces.

Q.35. Explain the following terms :

(i) Syntax and semantics.

(ii) Source code and object code. **CS : W-14(6M)**

OR Compare and contrast :

(i) Syntax and Semantics

(ii) Source code and Object code.

Ans.

(i) **Syntax and Semantics :**

- A program in any programming language can be viewed as string of symbols or characters choosen from some alphabet.
- But any arbitrary string of characters from the choosen alphabet does not represent a valid program.

- The rules telling whether a string is a valid program or not are called the syntax of the programming language.
- Certain notations, namely regular expressions and context-free grammars, are useful only for specifying much of the syntax of programming language, but also for helping in the construction of their compilers.
- If a program is valid program, then it is essential to know what a program means for it's faithful compilation into machine language program that will do what the programmer expects.
- The rules that give meaning to program are called the Semantics of a programming language is much harder to specify than it's syntax.

(ii) Source code and Object code :

- A compiler is a program which performs a job of translating a program written in high-level language into an equivalent program in a language close to machine.
- The program form written by the programmer using high-level language which is input to the compiler is called as Source code.
- The output program constructed by the compiler is as object code.
- For example, A program written in Pascal is source code and produced by Pascal compiler when this program is given as input is Object code.

Q.36. For a string S, define the following terms with an example :

(1) Prefix (2) Suffix

(3) Substring (4) Proper suffix

(5) Proper prefix (6) Length of string

(7) Language.

Ans.

(1) Prefix :

- If S is some string, then any string formed by discarding zero or more trailing symbols of s is called prefix of S.
- For example, abc is a prefix of abcde

(2) Suffix :

- If S is some string, then any string formed by deleting zero or more of the leading symbols of S is called suffix of S.
- For example; cde is suffix of abcde.

(3) Substring :

- A substring of S is any string obtained by deleting a prefix and a suffix from S. Moreover, any prefix or suffix of S is a substring of S, but a substring need not be a prefix or suffix.
- For example, cd is substring of abcde but not a prefix or suffix.

(4) Proper suffix : Let W = xy be a string. Then 'y' is a proper tail or suffix, if x is non empty.

(5) Proper prefix : Let W= xy be a string. Then 'x' is a proper prefix, if y is non empty.

(6) Length of string :

- If W is the string, then length of string is denoted as | W| and it is count of no. of symbols of which W is made up of.
- For example, If W = xyz, then | W| = 3. If | W | = 0, then the string is called as empty string and we use $\in$ to denote empty string.

(7) Language :

- It is a set of string formed from some specific alphabet.
- Simple sets such as the empty set, having no member or $\{\in\}$, the set containing only the empty string, are languages under this definition.
- If L and M are language then L.M or LM is the languages consisting of all strings xy which can be found by selecting a string x from L, a string y from M, and concatenating them in that order.

  LM = { xy / x is in L and y is in M}
- For example, L = {a, b} M = { c, d}

  then Reduce LM = { ac, ad, bc, bd }.

(8) Alphabet : It is finite set of symbols denoted by letter $\sum$.

(9) Production :

- A production of a grammar is a rule relating the variable.
- A typical production states that the language associated with a given variables contains strings from the language of certain other variables possibly along with some terminals.

Q.37. Discuss various parameter passing techniques with suitable example.

OR Explain with example parameter passing techniques used in programming languages. **CT : S-12(6M)**

OR Write short note on parameter transmission. **CT : W-12(4M)**

OR Write short note on parameter passing.

Ans. Parameter passing :

- When one procedure calls another, the usual method of communication between them is through nonlocal names and through parameters of the called procedure.

- Both nonlocals and parameters used by the procedure is to exchange the values of a[i] and a[j]. Here, array a is nonlocal to the procedure exchange and i and j are parameters.

For example, consider a pascal procedure swap with nonlocal and parameters.

(1) procedure exchange (i, j : integer) ;

(2) var x : integer

(3) begin

(4) x := a[i] ; a[i] : = a [j] ; a[j] : = x;

(5) end

- There are several common methods for associating actual and formal parameters. They are : call by value, call by reference, copy restore, call- by name and macro expansion.

(I) Call-by-value :

- This is the simplest method of passing parameters.
- The actual parameters are evaluated and their r-values are passed to the called procedure.
- Call by value is used in C and pascal parameters are usually passed this way.

Example :

Consider a pascal program with procedure swap.

(1) program reference (input, output);

(2) var a, b : integer;

(3) procedure swap (Var x, y : integer);

(4) begin

(5) temp := x;

(6) x = y ;

(7) y := temp;

(8) end;

(9) begin

(10) a : = 1; b : = 2;

(11) swap (a, b);

(12) writeln (a'=',a) ; writeln (b'=', b);

(13) end.

(II) Call-by reference :

- When parameters are passed by reference, the caller passes to the called procedure a pointer to the storage address of each actual parameter.

(1) If an actual parameter is a name or an expression having an l-value, then that l-value itself is passed.

(2) However, the actual parameter is an expression, like a + b or 2, that has no.1 value then the expression is evaluated in a new location, and the address of that location is passed.

- A reference to a formal parameter in the called procedure becomes, in the target code, an indirect reference through the pointer passed to the called procedure.

Example :

Consider a program with swap function using call by reference :

(1) swap (x, y)

(2) int *x, *y:

(3) { int temp;

(4) temp=*x ; *x =*y ; *y = temp;

(5) }

(6) main ( )

(7) { int a = 1, b = 2;

(8) swap (&a, &b);

(9) printf ("a is now %d, b is now %a\n", a, b);

(10) }

(III) Copy restore :

A hybrid between call-by-value and call-by-reference is copy-restore linkage.

Example :

Consider a simple program denostrate the use of copy restore.

(1) program copyout (input, output) ;

(2) var a : integer;

(3) procedure unsafe (vax x : integer);

(4) begin x : = 2; a : = 0 ;

(5) begin

(6) a :=1, unsafe (a); writeln (a);

(7) end.

(IV) Call-by-name :

Call-by-name is traditionally defined by the copy-rule of Algol which is :

(1) The procedure is treated as if it were a macro; that is, its body is substituted for the call in the caller, with actual parameters literally substituted for the formals. Such a literal substitution is called macro expansion or in-line expansion.

(2) The local names of the called procedure are kept distinct from the names of the calling procedure.

(3) The actual parameters are surrounded by parentheses if necessary to preserve their integrity.

Q.38. How parameter passing affects working of subroutine?

Ans.

- In subroutine (subprogram) formal parameter i.e. local data with subprogram is called actual parameter may be called in subprogram i.e. data object which is shared with the other subprogram.
- When a subprogram transfers control to another subprogram, there must be an association of the actual parameter of the calling subprogram with the formal parameter of the called program.
- Two approaches are often used; The actual parameter may be evaluated and that value passed to the formal parameter, or the actual object may be passed to the formal parameter.
- Whenever a formal parameter corresponding to a by name actual parameter is referenced in a subroutine, the thunk compiled for that parameter is executed.
- During call by reference, in execution of subprogram references to formal parameter names are treated as ordinary local variable references.
- During call by value, the alias to the actual parameter is created and that to be passed to subprogram execution program.

Q.39. Explain with suitable examples, the concept of *l*-value and r-value of an expression. **CT : W-13(4M)**

Ans. L-value and r-value :

- There is a distinction between the meaning of identifiers on the left and right sides of an assignment. In each of the assignments.

  i : = S ;

  i : = i + 1;

- The right side specifies an integer value, while the left side specifies where the value is to be stored. Similarly, if p and q are pointers to characters.

  $p^+ := q^+$;

- The right side q^+ specifies a character, while p^+ specifies where the character is to be stored. The term *l*-value and r-value after values that are appropriate on the left and right sides of an assignment, respectively. That is r-values are what we usually think of as "values," while *l*-values are locations.

Q.40. Explain significance of L and R value in writing assignment statements for a particular progra-mming languages.

CT : S-10(4M)

Ans. Significance of *l*-and r-values:

- Here *l*-value means left value and r-value means right value. The expression has the following significance of *l*-and r-values :

(1) Expression with single name has same *l*- and r-value as that of name.

(2) Expression has a *l*-value if it denotes location.

Examples :

(a) Every name has an *l*-value, n-namely the location or locations reserved for its value.

(b) If A is an array name, the *l*-value of A [I] is the location/locations reserved for the I^{th} element of the array. The r-value of A [I] is the value stored there.

(c) Constant has an r-value but no *l*-value.

(d) If P is a points, its r-value is the location to which P points and its *l*-value is the location in which the value of P itself is stored.

REGULAR LANGUAGES

Q.41. Give formal definition of Regular language.

Ans. Regular language :

- The collection of regular languages over an alphabet Σ is defined recursively as follows :

(i) The empty language ϕ is a regular language.

(ii) For each $a \in \Sigma$, the singleton language {a} is a regular language.

(iii) If A and B are regular languages, then $A \cup B$ (union), A . B (concatenation) and A^* are regular languages.

(iv) No other languages over Σ are regular.

Example : All finite languages are regular, in particular the empty string language $\{\epsilon\} = \phi^*$ is regular.

- A simple example of a language that is not regular is the set of strings $\{a^n b^n \mid n \geq 0\}$.
- It cannot be recognized with a finite automata, since a finite automata has finite memory and it cannot remember the exact number of a's.

Q.42. Mention the properties of regular language.

Ans. In theoretical computer science, a regular language is a formal language that satisfies the following equivalent properties :

(i) It can be accepted by a deterministic finite state machine.

(ii) It can be accepted by a non-deterministic finite state machine.

(iii) It can be accepted by an alternating finite automation.

(iv) It can be described by a formal regular expression.

(v) It can be generated by regular grammar and prefix grammar.

(vi) It can be accepted by a read-only Turing machine.

FINITE AUTOMATA

Q.43. Define and explain finite Automata.

Ans. Finite Automata :

- Finite Automata is a basic model of computation, which will translate the I/P into O/P without any human interaction.
- It is a mathematical model of finite state machine. It consists of finite sets of states, I/P symbols and set of transitions.
- In finite state machine the internal state of machine alters when the required machine receives an I/P and generates the O/P.
- It depends upon present state and I/P and generates the O/P.
- It depends upon present state and I/P applied, the O/P generated is the result generated after the process satisfying certain characteristics.

Formal Definition :

Finite Automata is denoted by five tuple :

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

where,

$Q \rightarrow$ Set of states, $(q_0, \ldots\ldots q_n)$.

$\Sigma \rightarrow$ Set of I/P symbols or alphabet.

$\delta \rightarrow$ Set of transitions, $Q \times \Sigma \rightarrow Q$.

$q_0 \rightarrow$ Initial state, $q_0 \subseteq Q$.

$F \rightarrow$ Set of final state, $F \subseteq Q$.

Capability of Finite Automata :

- It is used to decide whether the input is accepted or not accepted.
- If the machine terminates with final states the I/P is accepted.
- There are two types of Finite Automata (FA) :

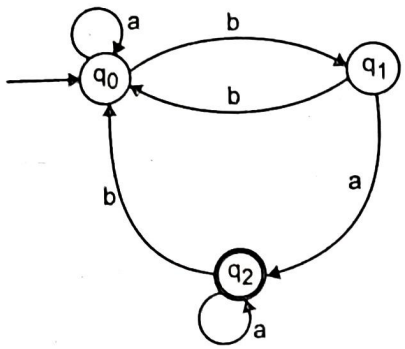
(1) Deterministic Finite Automata (DFA).

(2) Non Deterministic Finite Automata (NFA).

(1) Deterministic Finite Automata (DFA) :

If there is a single (unique) transition from one state to other with a single I/P symbol, then that automata is called as Deterministic Finite Automata.

Example :



A deterministic finite automata is a unituple.

$M = \{Q, \Sigma, \delta, q_0, F\}$

$M = \{(q_0, q_1, q_2), (a, b), \delta, (q_0), (q_2)\}$

Here, every state (q_0, q_1, q_2) for Input symbol (a,b), performs unique transition, like transitions are:

(1) $\delta(q_0, a) \rightarrow q_0$

(2) $\delta(q_0, b) \rightarrow q_1$

(3) $\delta(q_1, a) \rightarrow q_2$

(4) $\delta(q_1, b) \rightarrow q_0$

(5) $\delta(q_2, a) \rightarrow q_2$

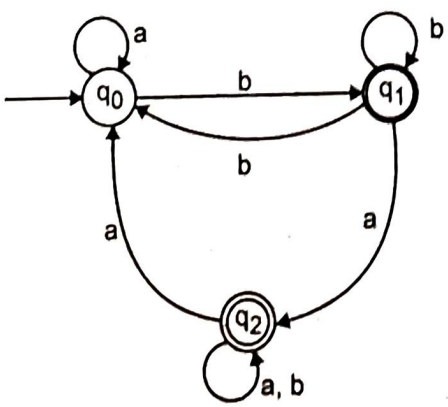
(6) $\delta(q_2, b) \rightarrow q_0$

} Unique transitions

(2) Non Deterministic Finite Automata (NFA) :

If there are more than one transitions from one state to another state with a single I/P symbol, then that automata is called as Non-Deterministic Finite Automata.

Example :



$M = \{(q_0, q_1, q_2), (a, b), \delta, (q_0), (q_1)\}$

Here, every state (q_0, q_1, q_2) for Input symbol (a, b), performs more than one transition shown as below :

(i) $\delta(q_0, a) \rightarrow \{q_0, q_2\}$

$\delta(q_0, b) \rightarrow \{q_1\}$

(ii) $\delta(q_1, a) \rightarrow \{q_2\}$

$\delta(q_1, b) \rightarrow \{q_0, q_1\}$

(iii) $\delta(q_2, a) \rightarrow \{q_2\}$

$\delta(q_2, b) \rightarrow \{q_2\}$.

Q.44. Give various application of finite automata.

OR Explain briefly application of finite automata.

OR Explain with suitable example the application of finite automata.

Ans. Application of Finite Automata are as follows :

(1) Lexical Analyzers.

(2) Text Editors.

(3) Switching circuit design.

(4) Text processing program.

(1) Lexical Analyzers :

- The tokens of programming languages can be expressed as regular sets.

  Example : ALGOL identifiers which are upper or lower - case letters followed by any sequence of letters and digits with no limit on length, may be expressed as :

  (letter) (letter/digit)*

- A number of lexical-analyzer generators take as input, a sequence of regular expressions describing the tokens and produce a single finite automaton recognizing any token.
- Usually, they convert the regular expression to an NFA with transitions and then construct subsets of states to produce a DFA directly, rather than first eliminating $\in$-transitions.
- Each final state indicates the particular token found, so the automaton is really a Moore machine.

(2) Text Editors :

Certain text editors and similar programs permit the substitution of a string for any string matching a given regular expression.

Example :

- The UNIX text editor allows a command such as : s/ bbb */b/ that substitutes a single blank for the first string of two or more blanks found in a given line.
- Let "any" denote the expression $a_1 + a_2 + ... + a_n$ where a_i's are all of a computers characters except the "new line" character.
- We could convert a regular expression r to a DFA that accepts any*r.
- The UNIX text editor converts the regular expression any* r is converted to an NFA with transitions and the NFA is then simulated directly.

Q.45. Discuss the role of finite automata in compiler.

Ans. Role of finite automata :

- The finite automata are used as the mathematical model that can be used to recognize the regular expressions.
- The regular expressions are built to match the pattern of the lexeme to identify the tokens. In order to recognize tokens the regular expressions can be created.
- These regular expressions then can be converted into the equivalent NFA. The NFA then can be converted to DFA.
- Input string is read character by character and lexical analysis is done using the finite automata so that valid tokens can be generated.

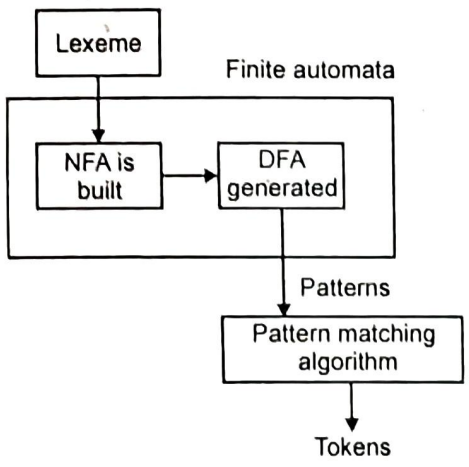


Fig. Role of finite automata

Q.46. Differentiate clearly between NFA and DFA.

OR Explain the point of differences between the determinism and non-determinism.

Ans.

Sr. No.	Non-deterministic finite automata (NFA)	Deterministic finite automata (DFA)
(1)	If the basic finite automata model is modified to allow zero, one, or more transitions from one state to another state on the same input symbol. The new finite automata is called NFA	DFA consist of a finite set of states and a set of transitions from state to state occur on single input symbols choosen from an alphabet Σ.
(2)	NFA is denoted by 5 tuple. $(Q, \Sigma, \delta, q_0, F)$ where, Q = finite set of states. Σ = set of input symbol. q_0 = start states. F = final states. δ = transition function from $Q \times \Sigma$ to Q.	DFA is denoted by 5 tuple $(Q, \Sigma, \delta, q_0, F)$ where, Q = set of states. Σ = set of input symbol. q_0 = start states. F = final states. δ = transition function from $Q \times \Sigma$ to Q.
(3)	In NFA there exists one or more transition from state to state on same input symbol.	In DFA, there exists exactly one transition from state to state on same input symbol.
(4)	In NFA state has an ϵ - transition.	In DFA no state has an ϵ - transition i.e. a transition on input ϵ.
(5)	NFA can recognize the regular set slower than DFA.	DFA can recognize the regular set faster than NFA.

Q.47. Give finite automata for a typical identifier of a programming language and also write the complete input required for LEX to produce an analyser that recognizes identifiers.

CS : S-09(5M), CT : S-09(6M)

Ans.

Fig. Finite automata for typical identifier

where, letter = a | b | ..... | Z | A | B | ..... | Z

digit = 0 | 1 | ..... | 19

Auxiliary definition :

letter = a | b | .....z | A | B | ..... | Z

digit = 0 | 1 | 2 |..... 19

identifier = letter (letter | digit) *

Translation rules :

identifier { LEXUAL = install id ( ) ; return (identifier) ; }

Lex program :

```
% { * Program for recognizing identifiers *}
% }
        letter [ A – Y a – z]
        digit [ 0 – 9]
        id letter (letter / digit / – ) *
        % %
        Auto / break / case / char / default / double /
        do / define / ..... { }
        {id} {printf ("\n identifier found % S", yy. Lex. 8) ;}
        % %
        main ( )
        {
              yy.lex ( ) ;
        }
```

Q.48. How is finite automata useful for lexical analysis? How many tokens are generated in the following statement?

printf("%d%f%",rollno,per_mark); ***CT : S-13(7M)***

Ans.

- A finite automata is an abstract machine that serves as an recognizer for the string that comprise a regular language.
- The idea is that we can feed an input string into a finite automata and will answer "yes" or "no" depending on whether or not the input string belong to the language that the automata recognizes.
- Lexical analysis is the power of reading the source text of a program and converting it into a sequence of tokens.
- The lexical structure of more or less every programming language can be specified by a regular language, a common way to implement a lexical analyzer is to :

(1) Specify regular explanation for all of the kinds of token in the language. The disjunction of all the regular explanation thus describes any possible token in the language.

(2) Convert the overall regular expression specifying all possible token into a deterministic finite automata.

(3) Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer.

- This approach is useful for programs called lexical analyzer generators exist to automate the entire processor.
- So in this way finite automata is useful for lexical analysis.

REGULAR EXPRESSIONS

Q.49. What is regular expression?

Ans. Regular expression :

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

(1) ε is a regular expression that denotes the set containing empty strings.

(2) If R1 and R2 are regular expression then R = R1 + R2 (same can also be represented as R = R1 | R2) is also regular expression which represents union operation.

(3) If R1 and R2 are regular expression then R = R1.R2 is also a regular expression which represents concatenation operation.

(4) If R1 is a regular expression than R = R1* is also a regular expression which represents kleen closure.

- A language denoted by regular expression is said to be a regular set or a regular language.

Example : Write a regular expression (R.E.) for a language containing the strings of length two over $\Sigma = (0,1)$

R.E. = (0 + 1) (0 + 1)

Q.50. Give the notations used for representing regular expression.

Ans. Notations used for representing regular expressions :

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

(1) **One or more instances :**

- To represent one or more instances + sign is used. If r is a regular expression then r* denotes one or more occurrences of r.
- For example, set of strings in which there are one or more occurrences of 'a' over the input set (a) then the regular expression can be written as a*. It basically denotes the set of {a, aa, aaa, aaaa, ...)

(2) **Zero or more instances :**

- To represent zero or more instances * sign is used. If r is a regular expression then r* denotes zero or more occurrences of r.
- For example, set of strings in which there are zero or more occurrences of 'a' over the input set (a) then the regular expression can be written as a*. It basically denotes the set of { ε , a, aa, aaa, ...).

(3) **Character class :**

- A class of symbols can be denoted by [ ].
- For example, [012] means 0 or 1 or 2. Similarly a complete class of a small letters from a to z can be represented by a regular expression [a-z]. The hyphen indicates the range. We can also write a regular expression for representing any word of small letters as [a-z]*.

Q.51. List the rules for constructing regular expressions. Write some properties to compose additional regular expressions. What is a regular definition? Give a suitable example.

Ans.

- The rules for constructing regular expressions over some alphabet Σ are divided into two major classifications which are as follows :

(i) Basic rules (ii) Induction rules

(i) **Basic rules :** There are two rules that form the basis :

(1) ϵ is a regular expression, and L $\{\epsilon\}$ is $\{\epsilon\}$, that is its language contains only an empty string.

(2) If a is a symbol in Σ, then a is a regular expression, and L {a} = {a}, which implies the language with one string, of length one, with a in its one position.

(ii) **Induction rules :**

- There are four induction rules that built larger regular expression recursively from smaller regular expressions.
- Suppose R and S are regular expression with languages L {R} and L {S}, respectively.

(1) (R) (S) is a regular expression representing the language L (R) . L (S).

(2) (R)|(S) is a regular expression representing the language L (R) ∪ L (S).

(3) (R)* is a regular expression representing the language (L (R))*.

(4) (R) is a regular expression representing L (R). This rule states that additional pairs of parentheses can be added around expressions without modifying the language.

Properties of regular expression :

- To compose additional regular expressions, the following properties are to be considered, a finite number of times :

(1) If a_1 is a regular expression, then (a_1) is also a regular expression.

(2) If a_1 is a regular expression, then a_1^* is also a regular expression.

(3) If a_1 and a_2 are two regular expressions, then $a_1 a_2$ is also a regular expression.

(4) If a_1 and a_2 are two regular expressions, then $a_1 + a_2$ is also a regular expression.

Regular definition : If Σ = alphabet set, then a regular definition is a sequence of definitions of the form :

$D_1 \rightarrow R_1$

$D_2 \rightarrow R_2$

.....

$D_n \rightarrow R_n$

where,

D_2 is a new symbol, not in Σ and not the same as any of the other D's.

R_1 is a regular expression over the alphabet $\Sigma \cup (D_1, D_2, ...D_{1-1})$

For example, let us consider the C identifiers that are strings of letters, digits, and underscores. Here, we give a regular definition for the language of C identifiers.

letter _ → A | B | .... | Z | a | b | ..... | z | _

digit → 0 | 1 | ..... | 9.

id → letter _ (letter _ | digit)*

POINTS TO REMEMBER :

(1) A compiler is a program that translates a high-level language program into functionally equivalent low-level language program.

(2) Tokens are symbolic names for the entities that make up the text of program.

(3) A lexeme is a sequence of characters from the input that match a pattern.

(4) A pattern is a rule that specifics when a sequence of characters from the input constitutes a token.

(5) An interpreter translates a programming language into a simplified language called intermediate code.

(6) A cross compiler is a compiler capable of creating executable code for a platform other than the one on which compiler is running.

(7) Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicate program.

(8) A lexical analysis is the interface between the source program and the compiler.

(9) The code generation phase converts the intermediate code into a sequence of machine instructions.

(10) Lex is a program generator designed for lexical processing of a character input stream.

(11) Flex is a fast lexical analyzer generator a tool for programming that recognizes lexical patterns in the input with help of flex specification.

(12) A finite automata is a restriction model of compiler having finite and fixed storage capability.

(13) If there are more than one transition from one state to another state with a single input symbol, then that automata is called as non-deterministic finite automata (NFA).

(14) The finite automata in which there exist only one transition from one state with a single input symbol to the another state is called as deterministic finite automata (DFA).

(15) The language accepted by finite automata can be represented by some expression called as regular expression.

UNIT - II

CONTENTS

UNIVERSITY PAPER SOLUTIONS
SINCE SUMMER - 2009

Note to the students :

- The questions given below are from previous Nagpur University examination papers.
- Though these questions are from old university papers, they have been included in the new syllabus.
- Questions which are out of new syllabus are not included here.

SUMMER - 09 (CS)

Q.1. Why an ambiguous grammar can not be LL(1)? **3M**

Ans. **P.2-29, Q.41.**

Q.2. What are the disadvantages of a non-predictive top down parser? **4M**

Ans. **P.2-21, Q.24.**

Q.3. Is the grammar with following production LL(1) ?

$S \rightarrow i\ L\ u\ E\ T / a$

$L \rightarrow LS / \epsilon$

$E \rightarrow b$

$T \rightarrow d\ L\ e / \epsilon$ **6M**

Ans. **P.2-32, Q.46.**

Q.4. Obtain LR parsing table for the following grammar which do not contain any multiple entry. The table should have minimum possible number of states.

$E \rightarrow E + T / T$

$T \rightarrow TF / F$

$T \rightarrow F^* / (E) / a / b / \epsilon$ **10M**

Ans. **P.2-53, Q.71.**

Q.5. What is handle? Calculate handles for the string "(a, (a, a))" using following grammar :

$S \rightarrow a / \wedge / (T)$

$T \rightarrow T, S / S$ **3M**

Ans. **P.2-24, Q.31.**

SUMMER - 09 (CT)

Q.1. Construct LR(1) parsing table for the following grammar :

$S \rightarrow B$

$B \rightarrow$ begin DA end

$D \rightarrow Dd / \epsilon$

$A \rightarrow A; E / E$

$E \rightarrow B / S$ **10M**

Ans. **P.2-57, Q.74.**

Q.2. Explain the working of bottom-up parsers. **4M**

Ans. **P.2-23, Q.30.**

Q.3. Construct LALR parsing table for the grammar :

$S \rightarrow abSa / aaAc / b$

$A \rightarrow dAb / b$. **10M**

Ans. **P.2-70, Q.88.**

Q.4. Explain various conflicts which arises in LR parsing. **3M**

Ans. **P.2-46, Q.63.**

WINTER - 09 (CS)

Q.1. Get LL(1) parsing table for following grammar.

$A \rightarrow aCDq / aBg / \epsilon$

$C \rightarrow p / \epsilon / C t / BD / rAb$

$D \rightarrow d / \epsilon$

$B \rightarrow e / \epsilon$ **9M**

Ans. **P.2-30, Q.44.**

Q.2. Why we calculate FIRST and FOLLOW? Explain with respect to top-down parsing. **4M**

Ans. **P.2-26, Q.35.**

Q.3. Obtain conflict free parsing table for the following grammar using LR parsing method.

$S \rightarrow iCSeS / iCS / a$ **13M**

Ans. **P.2-48, Q.66.**

WINTER - 09 (CT)

Q.1. What are the disadvantages of non-predictive parser? **3M**

Ans. **P.2-21, Q.24.**

Q.2. Explain the working of bottom-up parsers. **4M**

Ans. **P.2-23, Q.30.**

Q.3. Construct LL(1) parsing table for given grammar :

$S \rightarrow aBDh$

$B \rightarrow Bb$

$B \rightarrow c$

$D \rightarrow Eh$

$E \rightarrow g / E.$ **7M**

Ans. **P.2-31, Q.45.**

Q.4. Draw LR (1) parsing table for the following grammar :

$S \rightarrow L / a$

$L \rightarrow wGdS / dSwe$

$G \rightarrow b.$ **9M**

Ans. **P.2-55, Q.72.**

Q.5. Explain various conflicts which arise during LR parsing. **4M**

Ans. **P.2-46, Q.63.**

SUMMER - 10 (CS)

Q.1. Give significance of FIRST and FOLLOW set in relation with top-down parsing. **4M**

Ans. **P.2-26, Q.35.**

Q.2. Is the following grammar LL(1)?

$S \rightarrow aSA / \epsilon$

$A \rightarrow bB / cc$

$B \rightarrow bd / \epsilon$ **5M**

Ans. **P.2-35, Q.49.**

Q.3. Why preprocessing of grammar is required in top-down parsing? Explain. **3M**

Ans. **P.2-20, Q.23.**

Q.4. Construct LR(0) parsing table for the grammar :

$E \rightarrow E + E / E * E / (E) / id$. **10M**

Ans. **P.2-47, Q.65.**

Q.5. If erroneous input is given to LALR and LR(1) parser, which parser detects error earlier? What will be action of other parser at that situation? **3M**

Ans. **P.2-64, Q.81.**

SUMMER - 10 (CT)

Q.1. Write an algorithm to find canonical collections of LR (1) items. Also give an algorithm to construct ACTION and GOTO table for the same. **7M**

Ans. **P.2-77, Q.93.**

Q.2. Test whether the following grammar is LL (1) or not :

$S \rightarrow A \# 0,$

$A \rightarrow Ad/aB/aC$

$C \rightarrow c$

$B \rightarrow bBC/r$ **7M**

Ans. **P.2-36, Q.50**

Q.3. Explain the problems in top down parsing. **4M**

Ans. **P.2-19, Q.22.**

Q.4. Test whether the following grammar is LALR or not.

$S \rightarrow aSbS/bSaS/\epsilon$ **8M**

Ans. **P.2-68, Q.87.**

Q.5. Explain the significance of computing FIRST and FOLLOW. **2M**

Ans. **P.2-26, Q.35.**

WINTER - 10 (CS)

Q.1. Modify the following CFG so as to make it suitable for top-down parsing. Construct LL (1) parser for modified CFG. Show moves made by this LL (1) parser on input id + id * id.

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$ **8M**

Ans. **P.2-34, Q.48.**

Q.2. Describe data structures used for storing LR parsing tables. **5M**

Ans. **P.2-43, Q.59.**

Q.3. Find canonical states of following grammar :

$B \rightarrow bDAe$

$D \rightarrow Dd; / \epsilon$

$A \rightarrow A; E / \epsilon$

$E \rightarrow B / a$ **13M**

Ans. **P.2-73, Q.90.**

WINTER - 10 (CT)

Q.1. Show that no left-recursive grammar can be LL(1). **3M**

Ans. **P.2-28, Q.37.**

Q.2. Construct LL (1) parsing table for the following grammar.

$S \rightarrow aABb$

$A \rightarrow c/\epsilon$

$B \rightarrow d/\epsilon$

Also show the sequence of action of LL (1) parser on input acdb. **7M**

Ans. **P.2-33, Q.47.**

Q.3. Show that an ϵ -free LL (I) grammar can parse a sentence without FOLLOW ( ) set. **3M**

Ans. **P.2-28, Q.40.**

Q.4. Design LR (1) parsing table for the following grammar.

$S \rightarrow Aa / bAc / Bc / bBa$

$A \rightarrow d$

$B \rightarrow d$ **13M**

Ans. **P.2-56, Q.73.**

SUMMER - 11 (CS)

Q.1. Construct LR(0) parser for the grammar :

$S \rightarrow alJh$

$l \rightarrow lbSe / c$

$J \rightarrow KLKr / \epsilon$

$K \rightarrow d / \epsilon$

$L \rightarrow p / \epsilon$ **13M**

Ans. **P.2-61, Q.76.**

Q.2. Comment on the statement "Shift reduce conflict may get generated in LALR parser because of merging of states of LR (1), even though it is not already there in LR (1)". **5M**

Ans. **P.2-65, Q.82.**

Q.3. Comment on following statements :

(i) Left recursive grammar is not suitable for top-down parsing.

(ii) Every unambiguous grammar is LR grammar. **8M**

Ans. **P.2-44, Q.60.**

SUMMER - 11 (CT)

Q.1. Show that no left-recursive grammar can be LL(1). **3M**

Ans. **P.2-28, Q.37.**

Q.2. Construct predictive parsing table for the following grammar and tell whether the grammar is in LL(1) or not.

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$ **7M**

Ans. **P.2-29, Q.43.**

Q.3. Show that an $\in$-free LL(1) grammar can parse a sentence without FOLLOW ( ) Set. **3M**

Ans. **P.2-28, Q.40.**

Q.4. Construct a SLR (1) parsing table for the following grammar.

$S \rightarrow 0S0/1S1/10$ **10M**

Ans. **P.2-63, Q.78.**

Q.5. How can LR parsing table be implemented? **3M**

Ans. **P.2-43, Q.59.**

Q.6. Construct LR(1) parsing table for the following grammar

$S \rightarrow A$

$A \rightarrow BA / \in$

$B \rightarrow aB / b$

Also show the sequence of actions of LR(1) parser on input aaab. **13M**

Ans. **P.2-49, Q.67.**

WINTER - 11 (CS)

Q.1. Construct LL (1) parsing table for the grammar given below, and show stack and buffer entries for the string 'aabbdcc' using parsing table

$S \rightarrow aSA / \in$

$A \rightarrow bB / cc$

$B \rightarrow bd / \in$ **10M**

Ans. **P.2-35, Q.49.**

Q.2. Show that left recursive grammar is not LL(1). **3M**

Ans. **P.2-28, Q.37.**

Q.3. Construct canonical LR parsing table for grammar

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$ **8M**

Ans. **P.2-77, Q.94.**

Q.4. Comment on following statements :

(1) "Shift-Reduce conflict gets generated in LALR parser as a result of merging of states of LR(1), which is not already there in LR(1) parser."

(2) "Every LR (1) grammar is SLR but reverse is not true." **5M**

Ans. **P.2-65, Q.82.**

WINTER - 11 (CT)

Q.1. Consider the following grammar with start symbol A :

$A \rightarrow (B)/0$

$B \rightarrow B, A/A$

(a) Give parse tree for the string "(0, 0)" and "(0, (0, 0))". **3M**

(b) Show the derivation of the string (0, (0, 0)) using the leftmost derivation. **2M**

(c) Eliminate the left recursion from the grammar. **2M**

Ans. **P.2-17, Q.17.**

Q.2. Consider the following grammar over the alphabet {a, b, d}

$S \rightarrow ABD$

$A \rightarrow a/BSB$

$B \rightarrow b/D$

$D \rightarrow d/\in$

Construct LL (1) parsing table for the above grammar and check whether the grammar is LL (1) or not. **6M**

Ans. **P.2-37, Q.51.**

Q.3. Consider the following grammar :

$S \rightarrow S (s)/\in$

(a) Construct LR (1) set of items.

(b) Construct LR (1) Parsing table.

(c) Show the parsing stack and the action of an LR (1) parser of the input string "(( ) ( ))". **13M**

Ans. **P.2-59, Q.75.**

SUMMER - 12 (CS)

Q.1. Construct LALR (1) parser for the grammar

$S \rightarrow aIJh$

$I \rightarrow IbSe / c$

$J \rightarrow KLKr / \in$

$K \rightarrow d / \epsilon$

$L \rightarrow P / \epsilon$ **14M**

Ans. **P.2-74, Q.91.**

Q.2. Modify the following CFG so as to make it suitable for top-down parsing. Construct LL(1) parser for modified CFG. Show moves made by this LL(1) parser on input "id + id * id"

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$ **8M**

Ans. **P.2-34, Q.48.**

Q.3. List the disadvantages and advantages of top-down backtracking parser. **5M**

Ans. **P.2-21, Q.25**

SUMMER - 12 (CT)

Q.1. With reference to the grammar below :

$E \rightarrow E + Q/Q$

$Q \rightarrow Q * M/M$

$M \rightarrow id$

Write down the procedures of the recursive descent parser for the non-terminal E, Q and M. (E is the start symbol). **10M**

Ans. **P.2-22, Q.28.**

Q.2. What is ambiguous grammar? Check whether the given grammar is ambiguous or not.

$S \rightarrow aSbS / bSaS / \epsilon$ **3M**

Ans. **P.2-79, Q.97.**

Q.3. Construct LR (1) parsing table for the following grammar :

$S \rightarrow cA / ccB$

$A \rightarrow cA / a$

$B \rightarrow ccB / b$

Also show the moves of stack implementation for input string "ccccb." **10M**

Ans. **P.2-52, Q.70.**

Q.4. Find the reduced grammar that is equivalent to the CFG, given below :

$S \rightarrow aB / bF$

$A \rightarrow BAd / bSF / q$

$B \rightarrow aSB / bBF$

$F \rightarrow SBD / aBF / ad.$ **3M**

Ans. **P.2-18, Q.19.**

WINTER - 12 (CS)

Q.1. Construct LR (1) Parsing table for the following grammar.

$S \rightarrow wAz / xBz / wBy / xAY$

$A \rightarrow r$

$B \rightarrow r$ **8M**

Ans. **P.2-50, Q.68.**

Q.2. Find FIRST and FOLLOW for the following grammar. Construct the parsing table and find out whether the grammar is LL (1) or not

$E \rightarrow 10 * T / 5 + T$

$T \rightarrow PS$

$S \rightarrow QP / E$

$Q \rightarrow + / *$

$P \rightarrow a / b / c$ **10M**

Ans. **P.2-38, Q.52.**

Q.3. Explain the term handle with example. **3M**

Ans. **P.2-24, Q.31.**

Q.4. What is backtracking in top-down parsing? How can it be avoided? **5M**

Ans. **P.2-19, Q.22.**

WINTER - 12 (CT)

Q.1. What is ambiguous grammar? Check whether the given CFG is ambiguous or not?

$S \rightarrow aSbS / bSaS / \epsilon$ **4M**

Ans. **P.2-79, Q.97.**

Q.2. Construct the LL(1) parsing - table for the given CFG :

$S \rightarrow \{P\}$

$P \rightarrow P; P / \epsilon$

$P \rightarrow a/b$ **7M**

Ans. **P.2-39, Q.53.**

Q.3. Is shift-shift conflict possible in bottom up parsing? Justify your answer with example. **3M**

Ans. P.2-25, Q.32.

Q.4. What is handle? How does handle detection help in bottom up parsing? **3M**

Ans. P.2-24, Q.31.

Q.5. Construct LR(1) Parsing table for given grammar :

S → L / a

L → wGdS/dSwe

G → b. **10M**

Ans. P.2-55, Q.72.

SUMMER - 13 (CS)

Q.1. Get LL(1) parsing table for the following grammar and check validity of string "acbh."

A → aBDh

B → cC

C → bc / ∈

D → EF

E → g | ∈

F → f | ∈ **7M**

Ans. P.2-40, Q.56.

Q.2. Is LL(0) powerful than LL(1)? Justify. **6M**

Ans. P.2-29, Q.42.

Q.3. Obtain LALR (1) parsing table for following grammar

S → aUh

I → IbSe / c

J → KLKr / e

K → d / e

L → p / e **13M**

Ans. P.2-74, Q.91.

Q.4. Comment on following statement "for specification of token of any programming language regular expressions are used while for specification of grammar CFG are used". **4M**

Ans. P.2-10, Q.6.

SUMMER - 13 (CT)

Q.1. What is viable prefix? Calculate all viable prefixes for the string "(a, (a, a))" using following grammar :

S → a / ^ / (T)

S → T, s/s **3M**

Ans. P.2-78, Q.95.

Q.2. Write an algorithm to eliminate left recursion from the given grammar. Remove left recursion from the following grammar using this algorithm :

S → Aa/b

A → Ac/S d/ ∈ **2M**

Ans. P.2-17, Q.18.

Q.3. Explain the significance of FIRST and FOLLOW with respect to top-down parsing. **4M**

Ans. P.2-26, Q.35.

Q.4. Compute FIRST and FOLLOW and make parsing table for it.

E → E + T / T

T → T * F / F

F → (E) / id

Show the moves made by this LL(1) parser on input id + id * id. **7M**

Ans. P.2-34, Q.48.

WINTER - 13 (CS)

Q.1. Comment on truth/falsehood of the statement "LL(0) is less powerful than LR(0)". Justify your answer. **5M**

Ans. P.2-45, Q.61.

Q.2. Get LL(1) parsing table for the following grammar and check validity of string "iaubde."

S → iLuET / a

L → LS / ∈

E → b

T → dLe / ∈ **8M**

Ans. P.2-32, Q.46.

Q.3. Following grammar represents if-statement of programming languages. Obtain conflict free LR parsing table for the grammar.

S → iCS / iCSeS / a

$C \rightarrow c$ **8M**

Ans. **P.2-48, Q.66.**

Q.4. Define following terms in reference to LR parsing table construction :

(i) Augmentation

(ii) LR (0) item

(iii) Closure of LR (0) item **6M**

Ans. **P.2-45, Q.62.**

WINTER - 13 (CT)

Q.1. Show that the language is LL (1) or not :

$L\{a^n cb^n \mid n \geq 1\}$ **7M**

Ans. **P.2-40, Q.55.**

Q.2. What are the steps used to construct the predictive parse table for a grammar G ? Describe the top down parsing action for the model of predictive parser. **7M**

Ans. **P.2-21, Q.26.**

Q.3. Construct LALR parser for the following grammar :

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

$T \rightarrow S, T / S$

$R \rightarrow T$

Show the actions for the parser inputs :

(i) (a, a, $\wedge$)

(ii) a $\wedge$ (a) **13M**

Ans. **P.2-67, Q.86.**

SUMMER - 14 (CS)

Q.1. What are the disadvantages of recursive parsers? Explain with the help of example. **6M**

Ans. **P.2-21, Q.27**

Q.2. Obtain LL(1) parsing table for the following grammar :

$S \rightarrow iLuET / a$

$L \rightarrow LS / \epsilon$

$E \rightarrow b$

$T \rightarrow dLe / \epsilon$ **7M**

Ans. **P.2-32, Q.46.**

Q.3. Obtain LALR(1) parsing table for grammar :

$S \rightarrow L / a$

$L \rightarrow wGdS / dSwG$

$G \rightarrow b$ **13M**

Ans. **P.2-76, Q.92.**

SUMMER - 14 (CT)

Q.1. Given gammar

$S \rightarrow a / abSb / aAb$

$A \rightarrow bS / aAAb$

Check whether the grammar is ambiguous or not. **5M**

Ans. **P.2-80, Q.98.**

Q.2. What do you mean by left recursion and left factoring? Explain each with suitable example. **4M**

Ans. **P.2-16, 18, Q.16, Q.20.**

Q.3. Construct LL (1) parsing table for the grammar.

$E \rightarrow E + T / T$

$T \rightarrow T^* F / F$

$F \rightarrow (E) / id$

Show the moves made by the parser for the string

$w = id * id * id$ **13M**

Ans. **P.2-34, Q.48.**

Q.4. Write an algorithm for construction of parsing table for SLR parser. **6M**

Ans. **P.2-63, Q.77.**

Q.5. Given grammar

$S \rightarrow Aa / aAc / Bc / bBa$

$A \rightarrow d$

$B \rightarrow d$

Check whether the given grammar is LALR (1) or not. **7M**

Ans. **P.2-72, Q.89.**

WINTER - 14 (CS)

Q.1. Why we need to compute FIRST and FOLLOW in LL (1) parsing? Explain with example. **6M**

Ans. **P.2-26, Q.35.**

Q.2. Compute LL (1) parsing table for the following grammar :

$S \rightarrow aAB / bA / \epsilon$

$A \rightarrow aAb / \epsilon$

$B \rightarrow bB / c$ **7M**

Ans. **P.2-39, Q.54.**

Q.3. Explain : **6M**

(1) Top down parsing

(2) Bottom up parsing

(3) Handle

Ans. **P.2-19, 23, 24, Q.21, Q.29, Q.31.**

Q.4. Construct LR(1) parser for following grammar.

$E \rightarrow E + T / T$

$T \rightarrow TF / F$

$F \rightarrow F^* / a / b$ **7M**

Ans. **P.2-51, Q.69.**

Q.5 Write note on YACC. **3M**

Ans. **P.2-81, Q.99.**

Note : Also refer Paper Solutions at the end of the book.

SOLVED QUESTION BANK

[Sequence given as per syllabus]

INTRODUCTION

Syntax analyzer is the second phase of compiler which perform parsing i.e. syntax analysis. In this phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language.

This chapter shows how to check whether an input string is a sentence of a given grammar and how to construct, if desired, a parse tree for the string. As every compiler performs some type of syntax analysis, usually after lexical analysis, the input to a parser is typically a sequence of tokens. The outputs of the parser can be of many different forms. This chapter assumes for simplicity that the output in some representation of the parse tree.

SYNTAX ANALYSIS

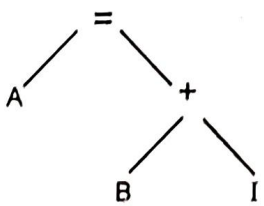
Q.1. What is syntax analysis or parsing? Give example.

Ans. Syntax Analysis :

- Syntax analysis processes the string of descriptors synthesized by the lexical analyzer to determine the syntactic structure of the input statement. This process is known as parsing.
- The result of parsing is a representation of the syntactic structure of a statement.
- The easiest representation to visualize is in the form of a syntax tree.

Example :

The statement A = B + I could be represented by the syntax tree shown below :



Q.2. What are issues in syntax-analysis? What is the need of CFG?

Ans. Two issues are involved when designing the syntax-analysis phase of a compilation process :

(1) All valid constructs of a programming language must be specified. That is, we form a specification of what tokens the lexical analyzer will return, and we specify in what manner these tokens are to be grouped so that the result of the grouping will be a valid construct of the language.

(2) A suitable recognizer is required to be designed to recognize whether a string of tokens generated by the lexical analyzer is a valid construct or not.

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

Q.3. Explain the role of parser.

Ans. Role of parser :

- The parser obtains a strings of tokens from the lexical analyzer as shown in below fig.

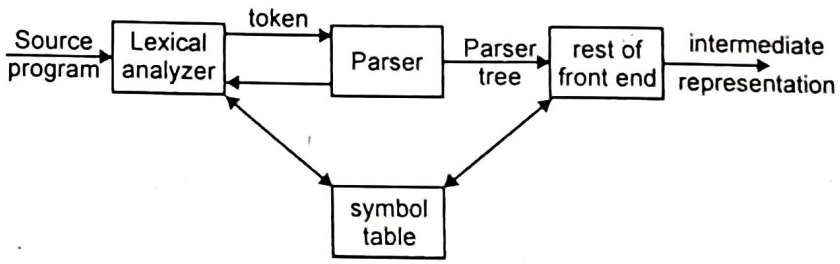


Fig. Position of parser in compiler model.

- The string can be generated by the grammar for the source language. The parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.
- There are three general types of parsers for grammars. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These methods are inefficient to use in production compilers.
- The methods commonly used in compilers are classified as being either top-down or bottom-up.
- Top-down parsers build parser trees from the top to bottom (root to leaves), while bottom-up parsers start from the leaves and work up to the root. In both cases, the input to the parser is scanned from left to right, one symbol at a time.
- The most efficient top-down and bottom-up methods work only on subclasses of grammars, but several of these subclasses, such as the LL and LR grammars, are expressive enough to describe most syntactic constructs in programming languages.
- The output of the parser is some representation of the parser tree for the stream of tokens produced by the lexical analyzer.
- There are number of tasks that might be conducted during parsing such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis and generating intermediate code.

SYNTAX SPECIFICATION OF PROGRAMMING LANGUAGES

Q.4. What is CFG? Explain with its notations.

Ans. Context Free Grammar :

- A context free grammar is a finite set of variables, each of which represents a language.
- CFG notation specifies a context-free language that consists of terminals, non-terminals, a start symbol and productions.
- The terminals are nothing more than tokens of the language, used to form the language constructs.
- Non-terminals are the variable that denote a set of strings. For example, S and E are non-terminals that denote statement strings and expression strings respectively.

- Grammar productions specify the manner in which the terminals and string sets, defined by the non terminals, can be combined to form a set of strings defined by a particular non terminal.
- For example, consider the production S → aSb. This production specifies that the set of strings defined by the non terminal S are obtained by concatenating terminal a with any string belonging to the set of strings defined by nonterminal S and then with terminal b.
- Each production consists of a nonterminal on the left-hand side, and a string of terminals and nonterminals on the right-hand side.
- The left-hand side of a production is separated from the right-hand side using the "→" symbol, which is used to identify a relation on a set (V ∪ T).
- Therefore context-free grammar is a four tuple denoted as :

  G = (V, T, P, S)

  Where :

(1) V is a finite set of symbols called as nonterminals or variables.

(2) T is a set of symbols that are called as terminals,

(3) P is a set of production and

(4) S is a member of V, called as start symbol.

Example :

G = ({S} , {a, b}, P, S) where P contains :

P = {S → asa,

S → bsb,

S → ∈

}

Q.5. What are the standard notation used in CFG?

Ans. Standard notation :

Following are the standard notations used in context free grammar :

(1) The capital letters toward the start of the alphabet are used to denote nonterminals (e.g. A, B, C, etc.)

(2) Lowercase letters toward the start of the alphabet are used to denote terminals (e.g. a, b, c, etc.)

(3) S is used to denote the start symbol.

(4) Lowercase letters toward the end of the alphabet (e.g. u, v, w, etc.) are used to denote strings of terminals.

(5) The symbols α, β, γ and so forth are used to denote strings of terminals as well as strings of nonterminals.

(6) The capital letters toward the end of alphabet (e.g. X, Y, and Z) are used to denote grammar symbols, and they may be terminals or nonterminals.

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

Q.6. Comment on following statement "for specification of token of any programming language regular expressions are used while for specification of grammar CFG are used". **CS : S-13(4M)**

Ans.

- The first task of lexical analyzer is to identify the tokens from string. After identification user use the suitable notation to specify this tokens.
- This notation should be compact, precise and easy to understand.
- The advantage of using regular expression notation to specify token is that when regular expressions are used, the recognizer for token ends up begin a DFA.
- Therefore applying regular expressions for specification of token for lexical analyzer is simple process.
- It involves transforming the regular expression into finite automata and generating program for simulating the finite automata.
- Syntax analyzer works to test whether the token generated by lexical analyzer are grouped according to syntactic rules or not.
- The notation for the construct specification should be compact, precise and easy to understand.
- The syntax structure specification for programming language uses CFG because for this class of grammar, we can automatically construct an efficient parser or recognizer that determine if source program is syntactically correct.

Example :

The regular expression is (a/b) (a/b/0/1)* and context free grammar is

S → aA/ bA

A → aA/ bA/0A/1A/ ∈

- Any syntactic construct that can be describe by a regular expression can also be describe by context free grammar.

Q.7. What is derivation tree? Give example.

Ans. Derivation tree or parse tree :

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

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

(2) The root node of the tree is labeled by the start symbol of the grammar.

(3) The interior nodes are labeled by the nonterminals.

(4) If an interior node has a label A, and it has n descendents with labels $X_1, X_2, \ldots\ldots, X_n$ from left to right, then the production rule $A \rightarrow X_1 X_2 X_3 \ldots\ldots X_n$ must exist in the grammar.

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

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

The tree shown in fig. is a derivation tree for a string id + id * id

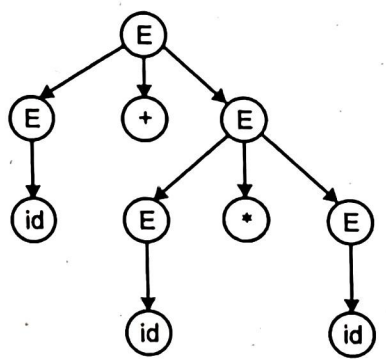


Fig. Derivation tree for the string id + id*id

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

Q.8. What are the orders of derivation? Explain in brief.

Ans. Order of derivation :

- There are two types of order of derivation :

(1) Left-most order of derivation.

(2) Right-most order of derivation.

(1) Left-most order of derivation :

- The left-most order of derivation is that order of derivation in which a left most non terminal is considered first for derivation at every stage in the derivation process.

Example :

One of the leftmost order of derivation of a string id + id * id is

$E \rightarrow E + E$

$E \rightarrow id + E$

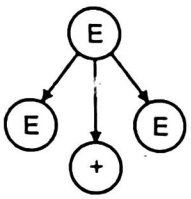
$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

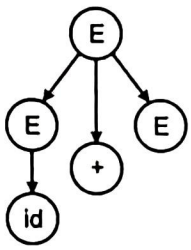
$E \rightarrow id + id * id$

Parse tree for leftmost derivation of string id+ id * id.

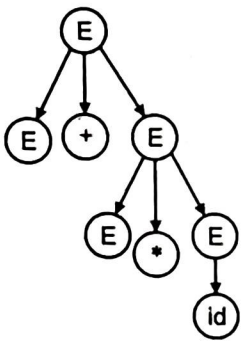
Step 1 : $E \rightarrow E + E$



Step 2 : $E \rightarrow E + E \rightarrow id + E$



Step 3 : $E \rightarrow E + E \rightarrow id + E \rightarrow id + E * E$



(2) **Right-most order of derivation :**

- In right-most order of derivation, the right-most non terminals is considered first
- Consider the same production for left-most derivation, the right-most derivation for string id + id * id is :

$E \rightarrow E * E$

$E \rightarrow E * id$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

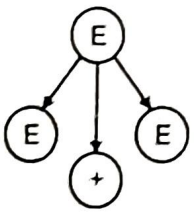
$E \rightarrow id + id * id$

Example : One of the rightmost order of derivation of id + id* id is

$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * id \rightarrow E + id * id \rightarrow id + id * id.$

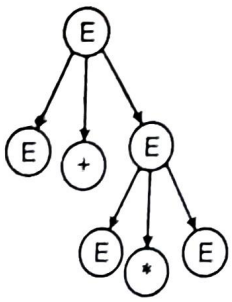
Parse tree for rightmost derivation of string id + id * id :

Step 1 : $E \rightarrow E + E$



Step 2 : $E \rightarrow E + E$

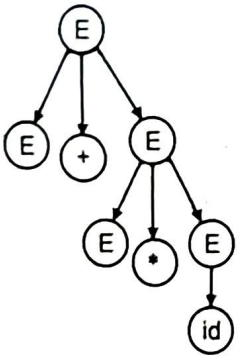
$E \rightarrow E + E * E$



Step 3 : $E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

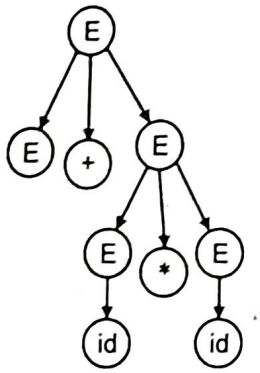


Step 4 : $E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$



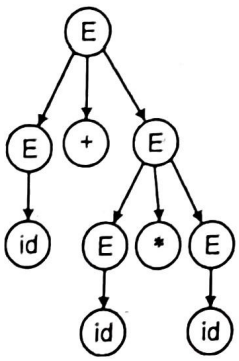
Step 5 : $E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$



The parse tree that gets generated using the leftmost order of derivation of id + id * id, given above, as well the parse tree that gets generated, using the rightmost order of derivation of id + id * id, given above is same, hence these orders are equivalent.

Q.9. Explain useless grammar symbols in brief with example.

Ans. Useless Grammar Symbols :

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

$X \xrightarrow{*} w$, where w is in T *

$S \xrightarrow{*} \alpha X \beta \xrightarrow{*} w$, w is in L(G).

- That is, a grammar symbol X is useless if it does not derive to terminal strings.

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

  $S \rightarrow aB / bX$

  $A \rightarrow BAd / bSX / q$

  $B \rightarrow aSB / bBX$

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

  $S \rightarrow bX$

  $A \rightarrow bSX / q$

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

  $S \rightarrow bX$

  $X \rightarrow ad$

Q.10. What is ϵ-production and nullable non terminals? Give algorithm for identifying nullable nonterminals.

Ans. **ϵ-production and nullable nonterminals :**

- A production of the form $A \rightarrow \epsilon$ is called a "ϵ - production".
- If A is a nonterminal and if $A \Rightarrow \epsilon$ (i.e., if A derive to an empty string in zero, one, or more derivations), then A is called a "nullable nonterminal.

Algorithm for identifying nullable nonterminals :

Input : G = (V, T, P, S)

Output : Set N (i.e., the set of nullable nonterminals)

{We maintain N_{old} and N_{new} to continue iterations}

```
begin
    N_old = ϕ
    N_new = ϕ
    for every production of the form A → ϵ
    do
    N_new = N_new ∪ {A}
    while (N_old ≠ N_new) do
    begin
        temp = V – N_new
        N_old = N_new
```

For every A in temp do

for every A-production of the form

$A \rightarrow X_1 X_2 \ldots X_n$ in P do

if each X_i is in N_{old} then

$N_{new} = N_{new} \cup \{A\}$

end

$N = N_{new}$

end

Q.11. Identify the nullable non-terminals in following production :

S → ACB / CbB/Ba

A → da / BC

B → gC / ∈

C → ha / ∈

Ans. By applying the algorithm for identifying nullable nonterminals the results after each iteration are shown below :

Initially :

$N_{old} = \phi$

$N_{new} = \phi$

After the execution of the first for loop :

$N_{old} = \phi$

$N_{new} = \{B, C\}$

After the first iteration of the while loop :

$N_{old} = \{B, C\}$

$N_{new} = \{B, C, A\}$

After the second iteration of the while loop :

$N_{old} = \{B, C, A\}$

$N_{new} = \{B, C, A, S\}$

After the third iteration of the while loop :

$N_{old} = \{B, C, A, S\}$

$N_{new} = \{B, C, A, S\}$

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

Q.12. How to eliminate ∈ -productions? Explain with example.

Ans. Eliminate ∈ -productions :

- Given a grammar G, containing ∈-productions. if L(G) does not contains ∈, then it is possible to eliminate all ∈-productions in the given grammar G.
- Whereas, if L(G) contains ∈, then elimination of all ∈-productions from G gives a grammar G_1 for which $L(G_1) = L(G) - \{\epsilon\}$
- To eliminate ∈-productions from a grammar we use the following technique.
- If $A \rightarrow \epsilon$, is an ∈-production to be eliminated, then we look for all those production in the grammar, whose right side contain A, and replace each occurrence of A in each of these productions by ∈.
- Thus we obtain the non ∈-productions, to be added to the grammar that the language generation remains the same.
- For example, consider the following grammar :

  $S \rightarrow aA$

  $A \rightarrow b / \epsilon$

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

  $S \rightarrow aA / a$

  $A \rightarrow b$

Q.13. Eliminate ∈ - production from the grammar below :

S → ABAC

A → aA / ∈

B → bB / ∈

C → c

Ans.

- To eliminate $A \rightarrow \epsilon$ from the grammar, the non ∈-productions to be added are obtained as follows :

  The list of the productions containing A on right hand side is :

  $S \rightarrow ABAC$

  $A \rightarrow aA$

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

  $S \rightarrow BAC / ABC / BC$

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

  $S \rightarrow ABAC / BAC / ABC / BC$

  $A \rightarrow aA / a$

  $B \rightarrow bB / \epsilon$

  $C \rightarrow c$
- To eliminate $B \rightarrow \epsilon$ from the grammar, the non ϵ-productions, to be added are obtained as follows :

  The productions containing B on the right-hand side are :

  $S \rightarrow ABAC / BAC / ABC / BC$

  $B \rightarrow bB$
- Replace each occurrence of B in these productions to order the non ϵ-productions, to be added to the grammar. The list of these productions is :

  $S \rightarrow AAC$

  $S \rightarrow AC$

  $S \rightarrow C$

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

  $S \rightarrow ABAC / BAC / ABC / BC / AAC / AC / C$

  $A \rightarrow aA / a$

  $B \rightarrow bB / b$

  $C \rightarrow c$

Q.14. What is unit production? Give algorithm for eliminating unit production.

Ans. Eliminating unit production :

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

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

  {

  select a unit production $A \rightarrow B$, such that there exist at least one non unit production

  $B \rightarrow a$

  for every non unit production $B \rightarrow \alpha$ do

  add production $A \rightarrow \alpha$ to the grammar

  eliminate $A \rightarrow B$ from the grammar

  }

Q.15. Eliminate all the unit productions from the grammar :

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow C / b$

$C \rightarrow D$

$D \rightarrow E$

$E \rightarrow a$

Ans. The given grammar contains productions

$B \rightarrow C$

$C \rightarrow D$

$D \rightarrow E$

which are the unit productions.

- To eliminate these productions from the given grammar, we first select a unit production $B \rightarrow C$.
- But since no nonunit C-production exist in the grammar, we then select $C \rightarrow D$.
- But since no nonunit D-productions exist in the grammar, we next select $D \rightarrow E$. There does exist a nonunit E-production : $E \rightarrow a$. Hence we add $D \rightarrow a$, to the grammar and eliminate $D \rightarrow E$
- But since $B \rightarrow C$ and $C \rightarrow D$ are still there, we once again select a unit production $B \rightarrow C$. Since no nonunit C-production exists in the grammar we select $C \rightarrow D$.
- Now there exist a nonunit production $D \rightarrow a$ in the grammar. Hence we add $C \rightarrow a$ to the grammar and eliminate $C \rightarrow D$.

- But since $B \rightarrow C$ is still there in the grammar, we once again select a unit production $B \rightarrow C$.
- Now there exist a non unit production $C \rightarrow a$ in the grammar, so we add $B \rightarrow a$ to the grammar and eliminate $B \rightarrow C$.
- Now no unit productions exist in the grammar. Therefore, the grammar that we get that does not contain unit productions is :

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow a / b$

$C \rightarrow a$

$D \rightarrow a$

$E \rightarrow a$

- But we see that the grammar symbols C, D and E, becomes useless as result of the elimination of unit productions, because they will not be used in the derivation of any w in L(G). Hence, we can eliminate them from the grammar to obtain :

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow a / b$

- Therefore, we conclude that to obtain the grammar in the most simplified form, we have to eliminate unit productions first. We then eliminate the useless grammar symbols.

Q.16. What is left recursive grammar? Explain the method to eliminate left-recursion.

OR What do you mean by left recursion? Explain with suitable example. **CT : S-14(2M)**

Ans. Eliminate left recursion :

- If a grammar contains a pair of productions of the form $A \rightarrow A\alpha / \beta$, then the grammar is a "left recursive grammar".
- If left-recursive grammar is used for specification of the language, then the top-down parser designed for the grammar's language may enter into an infinite loop during the parsing process on some erroneous input.
- This is because a top-down parser attempts to obtain the left-most derivation of the input string w; hence, the parser may see the same nonterminal A every time as the left-most nonterminal. And every time, it may do the derivation using $A \rightarrow A\alpha$.
- Therefore, for top-down parsing, nonleft-recursive grammar should be used.
- Left recursion can be eliminated from the grammar by replacing $A \rightarrow A\alpha / \beta$, with the productions $A \rightarrow \beta B$ and $B \rightarrow \alpha\beta / \epsilon$.
- In general, if a grammar contain productions :

$A \rightarrow A\alpha_1 / A\alpha_2 .... / A\alpha_m / \beta_1 / \beta_2 .... / \beta_n$ /, then the left recursion can be eliminated by adding the following productions in place of the above

$A \rightarrow \beta_1 B / \beta_2 B / .../ \beta_n B$

$B \rightarrow \alpha_1 B / \alpha_2 B / .../ \alpha_m B / \epsilon$

- Consider the following grammar :

$S \rightarrow aBDh$

$B \rightarrow Bb / c$

$D \rightarrow EF$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$

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

$B \rightarrow cC$

$C \rightarrow bC / \epsilon$

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

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \epsilon$

$D \rightarrow EF$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$

Q.17. Consider the following grammar with start symbol A

A → (B)/0

B → B, A/A

(a) Give the parse tree for the string "(0, 0)" and "(0, (0, 0))".

(b) Show the derivation of the string (0, (0, 0)) using the leftmost derivation.

(c) Eliminate the left recursion from the grammar.

CT : W-11(7M)

Ans.

(a) For w = (0, 0), the leftmost derivation is as follows :

A → (B)

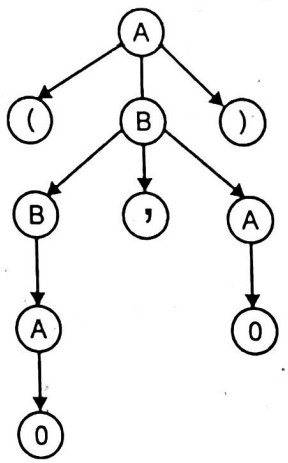
A → (B, A) *(∵ B → B, A)*

A → (A, A) *(∵ B → A)*

A → (0, A) *(∵ A → 0)*

A → (0, 0)

∴ The parse tree for above derivation is :



The leftmost derivation for the string (0, (0, 0)) is :

A → (B)

A → (B, A) *(∵ B → B, A)*

A → (A, A) *(∵ B → A)*

A → (0, A) *(∵ A → 0)*

A → (0, (B)) *(∵ A → (B))*

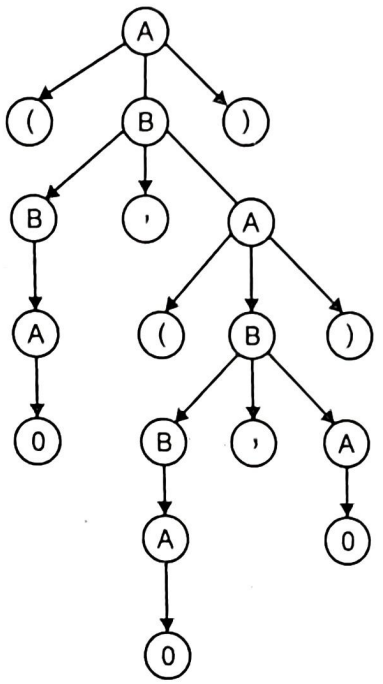
A → (0, (B, A)) *(∵ B → (B, A))*

A → (0, (A, A)) *(∵ B → A)*

A → (0, (0, A)) *(∵ A → 0)*

A → (0, (0, 0)) *(∵ A → 0)*

∴ The parse tree for above derivation is :



The given production has left recursion in the following statement,

B → B, A/A (Similar to A → A ∝ / β)

Left recursion can be eliminated by writing production of the form,

A → β A′

A′ → ∝ A′ / ∈

Using this rule, the given production can be replaced by,

B → AB′

B′ → ,AB′ / ∈

∴ The complete grammar after elimination of left recursion is :

A → (B) / 0

B → AB′

B′ → ,AB′ / ∈

Q.18. Write an algorithm to eliminate left recursion from given grammar. Remove left recursion from following grammar using this algorithm :

S → Aa/b

A → Ac/Sd / ∈ **CT : S-13(2M)**

Ans. Algorithm for left recursion :

(1) Arrange the non-terminal of G in some order $A_1, A_2, A_3, \ldots A_n$.

(2) For i = 1 to n do

begin

For j = 1 to i – 1 do

replace each production of the form $A_i \rightarrow A_j\gamma$ by the production

$A_j \rightarrow \delta_1\gamma / \delta_2\gamma$ ..... $/ \delta_k\gamma$

where $A_j \rightarrow \delta_1 / \delta_2$ ..... $/ \delta_k$ are all the current A_j production.

Eliminate the immediate left-recursion among the A_i productions.

end.

The given grammar is :

$S \rightarrow Aa$

$S \rightarrow b$

$A \rightarrow Ac$

$A \rightarrow Sd$

$A \rightarrow \epsilon$

The grammar contain left recursion in production $A \rightarrow Ac$.

Substituting S in A production we get,

$A \rightarrow Ac / (Aa / b)d / \epsilon$

$A \rightarrow Ac / Aad / bd / \epsilon$

Applying algorithm to eliminate left recursion, we get,

$A \rightarrow \epsilon A' / bdA'$

$A' \rightarrow \epsilon A' / adA' / \epsilon$

Production after eliminating left recursion :

$S \rightarrow Aa / b$

$A \rightarrow \epsilon A' / bdA'$

$A' \rightarrow cA' / adA' / \epsilon$

Q.19. Find reduced grammar that is equivalent to CFG given below :

$S \rightarrow aB / bF$

$A \rightarrow BAd / bSF / q$

$B \rightarrow aSB / bBF$

$F \rightarrow SBD / aBF / ad.$ ***CT : S-12(3M)***

Ans. Reduced grammar can be obtained by eliminating useless symbols as follows :

(1) B is useless since it does not derive terminal string, so eliminate B.

(2) A is useless because it does not appear in sentential form of start symbol.

(3) D is also useless symbol, so eliminate the productions containing D.

(4) F is useful symbol as it derive terminal string and also appears in sentential form

∴ Reduced grammar is :

$S \rightarrow bF$

$F \rightarrow ad$

Q.20. What do you mean by left factoring? Explain with suitable example. ***CT : S-14(2M)***

OR Give an algorithm to eliminate left factoring from a grammar.

Ans. Left factoring :

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- If $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ are two A-productions, and the input begins with a non-empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$. The decision by expanding A to $\alpha A'$. The input derived from α, we expand A' to β_1 or to β_2.
- The original productions become

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

Algorithm to eliminate left factoring :

(1) For each nonterminal A, find the longest prefix α common to two or more of its alternatives.

(2) If $a \neq \epsilon$ i.e. there is non trivial common prefix, replace all the A productions

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$ ----- $| \alpha\beta_n | \gamma$

where

γ represents all alternatives that do not begin with α by

$A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 |$ ----- $| \beta_n$

Here A' is a new non-terminal.

DESIGN OF TOP DOWN PARSER

Q.21. Explain top-down parsing. **CS : W-14(2M)**

Ans. Top-down parsing :

- When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.
- Top down parsing attempts to find the left-most derivations for an input string w, which is equivalent to constructing a parse tree for the input string w that starts from the root and creates the nodes of the parse tree in a predefined order.
- Consider the top-down parser for the following grammar :

  $S \rightarrow aAb$

  $A \rightarrow cd / c$

  Let input string be w = acb.
- The parser initially creates a tree consisting of a single node, labeled S, and the input pointer points to a, the first symbol of input string w. The parser then uses the S-production S → aAb to expand the tree as shown in fig. (a).

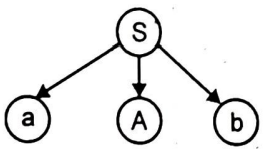


Fig.(a)

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

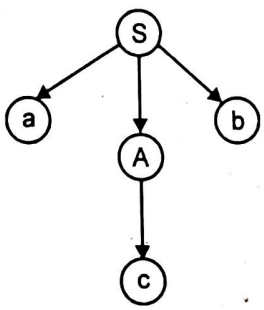


Fig.(b)

The parser now has the match for the second input symbol. So, it advances the pointer to b, the third symbol of w, and compares it to the label of the next leaf.

Q.22. Explain the problems in top-down parsing. **CT : S-10(4M)**

OR What is backtracking in top-down parsing? How can it be avoided? **CS : W-12(5M)**

Ans. Backtracking :

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

  $S \rightarrow aAb$

  $A \rightarrow cd / c$

  Let the input string be w = acb. The parser initially creates a tree consisting of a single node, labeled S and the input pointer points to a, the first symbol of input string w. The parser then uses the S-production S → aAb to expand the tree as shown in Fig. (a).

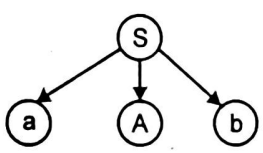


Fig.(a)

- The above grammar S → aAb expands the root node.

- The left-most leaf labeled a matches the first input symbol of w. Hence the parser will advance the input pointer to 'c' which is second symbol and consider the next leaf labeled A.
- It will expand A using first alternative for A in order to obtain tree.
- Now A have two alternative grammars which one to select is difficult task.
- First the grammar A → cd is applied then the tree will be as follows :

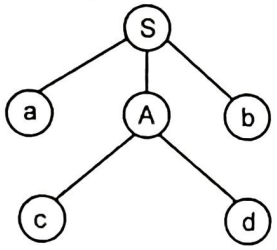


Fig.(b)

- Now the input pointer point c and the left leaf node is also c when compaired with w string. Now the input pointer advances to "b" but our leaf node is d, this is the pointer of failure as the leaf node "d" and the input string from w after "a" does not matches.

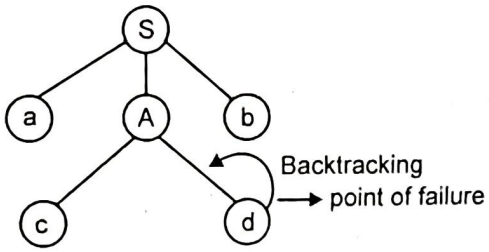


Fig.(c)

- Here the backtracking is needed and the parser will reset the input pointer to the second symbol the position where the pointer encountered A and will try second alternative of A.
- In backtracking we try the second grammar A → c and the tree becomes.

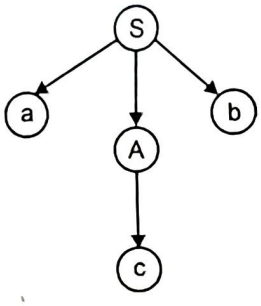


Fig. (d)

- The leaf node c matches with the input pointer again it advances to next input pointer i.e. b and again it is the match with leaf node so, we get the perfect match and parser will halt and announces the successful completion of parsing.
- The backtracking problem in top-down parser can be solved that is a top-down parser can function as a deterministic recognizer if it is capable of predicting or detecting which alternatives are right choices for the expansion of nonterminals during the parsing of input string w.

Q.23. Why preprocessing of grammar is required in top-down parsing? Explain. **CS : S-10(4M)**

Ans.

- Preprocessing of grammar is required to eliminate left recursion :
- It means that the grammar for assignment statement is left recursive. In the process of expanding the first non terminal, that nonterminal is generated again.

For example :

expression → expression + Term

↑ ↑

- Parsing, here initiated by driver program proceeds by expanding production until a terminal (token) is finally generated.
- Then that terminal is read from the input and procedure continues.

Elimination of left recursion :

There is formula to do it for each rule which contain a left recursive option,

$A \rightarrow A \alpha / \beta$

Introduces a new non terminal A′ and rewrite rule as

$A \rightarrow \beta A'$

$A' \rightarrow \epsilon | \alpha A'$

Example :

$E \rightarrow E + T / T$

So after elimination of left recursion it would be as

$E \rightarrow TE'$

$E' \rightarrow \epsilon / + TE'$

Q.24. What are the disadvantages of a non-predictive top down parser? **CS : S-09(4M), CT : W-09(3M)**

Ans. Disadvantages of a non-predictive top-down parser :

(1) A left recursive grammar can cause a non predictive parser (i.e. recursive descent parser), even one with backtracking to go into an infinite loop.

(2) In situation like natural language parsing, non predictive parser are not very efficient.

(3) In non-predictive parser (backtracking parser), the order in which alternatives are tried affects the language accepted by the parser.

Q.25. List the disadvantages and advantages of top-down backtracking parser. **CS : S-12(5M)**

Ans. Advantage of top down backtracking parsing :

(1) It can handle any context-free language.

Disadvantages of top down backtracking parsing :

(1) Semantic actions cannot be performed while making a prediction. The actions must be delayed until the prediction is known to be a part of a successful parse.

(2) Precise error reporting is not possible. A mismatch merely triggers backtracking. A source string is known to be erroneous only after all predictions have failed.

(3) Top down backtracking might take exponential time.

Q.26. What are the steps used to construct the predictive parsing table for a grammar G? Describe the top-down parsing action for a model of predictive parser. **CT : W-13(7M)**

Ans. These are the steps to construct predictive parsing table :

(1) Eliminate left recursion if any from the grammar G.

(2) Perform left factoring on grammar G.

(3) Find FIRST and FOLLOW on the symbol of grammar G.

(4) Construct predictive parse table.

(5) Check if the given input string can be accepted by the parser.

Implementation of table driven predictive parser :

(1) In table driven predictive parser, three data structures are used which are as follows :

(a) An input buffer

(b) A stack

(c) A parsing table

(2) The input buffer is used to hold the string which is to be parsed and the string is followed by a $ symbol i.e. right end marker to indicate the end of the input string.

(3) Stack is used to hold the sequence of grammar symbols and the bottom of the stack consists of $ with the start symbol of the grammar above the $.

(4) The parsing table is obtained by calculating the FIRST and FOLLOW.

(5) The parser is controlled by the program that is as follows :

(a) x is the symbol at the top of the stack and next input symbol is a.

(b) If x = a = $, parser announces the successful completion of parsing and halts.

(c) If x = a ≠$, the parser pops the x out of stack and advances the input pointer to the next input symbol.

(d) If x is nonterminal then the program consults the parsing table entry TABLE [x, a]. If TABLE [x, a] = x → UVW then parser replaces x on the top of the stack by UVW in such a manner that U will come on top. If TABLE [x, a] = error then the parser calls the error routine.

(6) If parser contain multiple entries then the parser is still non-deterministic.

(7) The parser will be deterministic if and only if there are no multiple entries in parsing table.

(8) After applying the algorithm on grammar if the parsing table does not contain the multiple entries then all such grammar constitute a subset of CFG's called as "LL(1)".

RECURSIVE DESCENT PARSING

Q.27. Explain recursive descent parsing. Give its procedure.

OR What are the disadvantages of recursive parser? Explain with example. **CS : S-14(6M)**

Ans. Recursive descent parsing :

- It is the most general form of top-down parsing. It may involve backtracking, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal.
- Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree

- In many runtime cases, top-down parser require no backtrack. To avoid this if given current i/p symbol is expanded which is having more than one production like $A \rightarrow \alpha_1 | \alpha_2 | \cdots \alpha_n$ then parser must be able to decide which is the unique alternative that derive a string begining with a
- For example,

  If condition then statement

  else statement/ while

  condition do statement

  begin statement list end
- Here the keyword if, while, begin each of which can explain alternative that could possibly to find statement.
- In order to avoid the necessary which can be implemented in tabular form.

Procedure for recursive descent parser :

S → T$

T → [T]/[ ]T / [T]T * in

```
S ( ) = T ( )
{
      If (input = = 'E')
   {
      advance( );
      If (T ( ) ! = error).
      If (input = = 'J').
   {
      advance ( );
      If (input = = end marker)
      return (success);
      else
      return (errors);
      else
   }  // inner if end.
   }  // outer if end
      return (errors);
   }
      else
      return (error);
   }
   T ( )
   {
      If (input = = '[')
   {  advance ( );
      If (input = = ']')
      advance ( );
   }
      else
      return (error);
   }
   main ( )
{
append the end marker to the string w to be parsed;
Set the input pointer to the leftmost token of w;
   if (s ( ) ! = error)
   printf("successful completion of the parsing");
   else
   printf("failure");
}
```

Q.28. With reference to the grammar below:

E → E + Q/Q

Q → Q * M/M

M → id

Write down the procedures of the recursive descent parser for the non-terminal E, Q and M. (E is the start symbol).

CT : S-12(10M)

Ans. Given grammar :

E → E + Q/Q

Q → Q * M/M

M → id

Recursive descent parser for above grammar include procedure for E, Q and M.

```
Procedure E ( )
{
```

```
        Q();
        EPRIME();
}
Procedure Q()
{
        M();
        QPRIME();
}
Procedure EPRIME()
{
        If input = " + "
        {
            advance();
            Q();
            EPRIME();
            return true;
        }
        return false;
}
Procedure QPRIME()
{
        If input = " * "
        {
            advance();
            M();
            QPRIME();
            return true;
        }
        return false;
}
Procedure M()
{
        If input = " id "
        {
            advance();
            return true;
        }
        return false;
    }
```

DESIGN OF BOTTOM-UP PARSER

Q.29. Explain bottom-up parsing. *CS : W-14(2M)*

Ans. Bottom-up parsing :

- Bottom up parsing starts with the input symbols and tries to construct the parse tree upto the start symbol.
- Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.
- Consider the input string as id + id * id for following production :

  $S \rightarrow E$

  $E \rightarrow E + T$

  $E \rightarrow E * T$

  $E \rightarrow T$

  $T \rightarrow id$

- The bottom-up parsing for above input string can be done as follows :

  id + id * id

  T + id * id *($\because T \rightarrow id$)*

  E + id * id *($\because E \rightarrow T$)*

  E + T * id *($\because T \rightarrow id$)*

  E * id *($\because E \rightarrow E + T$)*

  E * T *($\because T \rightarrow id$)*

  E *($\because E \rightarrow E * T$)*

  S *($\because S \rightarrow E$)*

Q.30. Explain the working of bottom-up parser.

CT : S-09, W-09(4M)

Ans.

- Bottom-up parsing can be defined as on attempt to reduce the input string w to the start symbol of a grammar by tracing out the rightmost derivations of w in reverse.
- This is equivalent to constructing a parse tree for the input string w by starting with leaves and proceeding towards the root.
- It is an attempt to construct the parse tree from bottom to up.
- This involves searching for the substring that matches the right side of any of the productions of any grammar.

- This substring is replaced by the left hand side non-terminal of the production if this replacement leads to the generation of the sentential form that comes one step before in the rightmost derivation.
- This is process of replacing the right side of the production by the left side non-terminal is called, "reduction".
- For example, if the rightmost derivation sequence of some w is :

  $S \to \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4 \ldots\ldots \to \alpha_{n-1} \to w$ the bottom up parser starts with w and searches for the occurrence of a substring of w that matches the right side of some production $A \to \beta$ such that the replacement of β by A will lead to the generation of α_{n-1}.
- The parser replaces β by A_1 then it searches for the occurrence of a substring of α_{n-1} that matches the right side of some production $\beta \to \gamma$ such that the replacement of γ by β will lead to the generation of α_{n-2}.
- This process continues until the entire w substring is reduced to S, or until the parser encounter an error.

Q.31. What is handle? Calculate handles for the string "(a, (a, a))" using following grammar :

$S \to a / \wedge / (T)$

$T \to T, S / S$ **CS : S-09(3M)**

OR Explain the term handle with example. **CS : W-12(3M)**

OR What is handle? How does handle detection help in bottom up parsing? **CT : W-12(3M)**

OR Explain handle. **CS : W-14(2M)**

Ans. Handle of right sentential form :

- A handle of a right sentential form γ is a production $A \to \beta$ and a position of β in γ.
- The string β will be found and replaced by A to produce the previous right sentential form in the rightmost derivation of γ i.e. if $S \to \alpha A\beta \to \alpha\gamma\beta$ then $A \to \gamma$ is a handle of $\alpha\gamma\beta$ in the position following α.

Example :

Consider the following grammar

$\to E + E / E * E / id.$

and the rightmost derivation :

$E \to E + E \to E + E * E$ *(consider $E \to E * E$)*

$E \to E + E * id$ *(as $E \to id$)*

$E \to E + id * id$

(Consider from rightmost and replace E by id)

$E \to id + id * id$

- The handles of the sentential forms occurring in the above derivation is as follows :

Sentential form	Handle
id + id * id	E → id at the position preceding +
E + id * id	E → id at the position following +
E + E * id	E → id at the position following *
E + E * E	E → E * E at the position following +
E + E	E → E+E at the position preceding at end marker.

- Therefore, the bottom-up parsing is an attempt to detect the handle of a right sentential form.
- Whenever handle is detected, the reduction is performed. This is equivalent to perform rightmost derivations in reverse and is called "handle pruning".

Example :

$S \to a / \wedge / (T)$

$T \to T, S / S.$

and string is "(a, (a, a))"

The rightmost derivation of string (a, (a, a)) is

$S \to (T)$

$S \to (T, S)$ *($\because T \to (T, S)$)*

$S \to (T, (T))$ *($\because S \to (T)$)*

$S \to (T, (T, S))$ *($\because T \to T, S$)*

$S \to (T, (T, a)$ *($\because S \to a$)*

$S \to (T, (S, a))$ *($\because T \to S$)*

$S \to (T, (a, a))$ *($\because S \to a$)*

$S \to (S, (a, a)) \to (a, (a, a))$

Table for sentential forms occurring in the following derivation :

Sentential form	Handle
(a, (a, a))	S → a at the position preceding the first comma.
(S, (a,a))	T → S at the position preceding first comma.
(T, (a, a)	S → a at the position preceding second comma.
(T, (S, a)	T → S at the position preceding second comma.
(T, (T, a)	S → a at the position following second comma.
(T, (T, S)	T → (T,S) at the position following first comma.
(T, (T))	S → T at the position following first comma.
(T, S)	T → (T, S) at the position following the first bracket.
(T)	S → (T) at the position before end marker.

Q.32. Is shift-shift conflict possible in bottom up parsing? Justify your answer with example. **CT : W-12(3M)**

Ans.

- If it is impossible to perform a reduction, there are token remaining in the undigested input, then we transfer a token from input on the stack. This is called a shift.

  For example, using the grammar above, suppose the stack contained C and the input contained id + id. It is impossible to perform a reduction on production, as it does not match the entire right side of any of our production.
- So we shift the first character of the input onto the stack, giving us id on stack and + id remaining in the input.
- When only shift-shift technique is applied then the production not get reduce or shift reduce strategy divides the string that we are trying parse into two parts : an undigested part and a semi-digested part is put on stack.
- If height of the two above case apply, then this generate an errors.
- The sequence on the stack does not match the right hand side of any production.
- If shifting the next input token would create a sequence on the stack that cannot eventually be reduced to the start symbol, a shift action would be futile.

Q.33. Explain operator precedence parser.

Ans. Operator precedence parser :

- For a certain small class of grammar, we can easily construct efficient shift-reduce parser by hand.
- These grammars have the property that no production right side is 'E' or has two adjacent non-terminals. A grammar with latter property is called an operator grammar.
- Consider the following grammar :

  $E \rightarrow EAE / (E) / -E / id$

  $A \rightarrow + / - / * / / / \uparrow$

  This grammar is not operator grammar, because the right side EAE has two consecutive non-terminals. But if we substitute for A then we get operator grammar :

  $E \rightarrow E + E / E - E / E*E / E / E / E \uparrow E / (E) / -E / id.$
- But operator precedence parser has some disadvantages.

(1) It is hard to handle tokens like minus sign, which has two different precedences.

(2) One cannot always be sure the parser accept exactly the desired language.

(3) Only small class of grammars can be parsed using operator precedence technique.

- In operator precedence parsing, we define three disjoint precedence relations α, =, and > between certain pairs of terminals. These precedences have following meaning :

Relation	Meaning
$a <\cdot b$	a "yields precedence to" b
$a = b$	a "has the same precedence as" b
$a \cdot > b$	a "takes precedence over" b

- There are two ways of determining what precedence relation should hold between a pair of terminals.
- The first method is based on associativity and precedence of operators. For example, if * is to have higher precedence than +, we make + $<\cdot$ * and * $\cdot>$ +.
- The second method of selecting operator precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects corrects associativity and precedence in its parse tree.
- Suppose that between ai and ai + 1 exactly one of the relations $<\cdot$ = and $\cdot>$ holds. Here $ mark is the end of string and define $ $<\cdot$ b and b $\cdot>$ $ for all terminals b. For example, take one right sententional form id + id * id and the precedence relations are as below.

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

Then the string with precedence relation inserted

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

- The handle can be found out by following processes :

(1) Scan the string from left end until the first $\cdot>$ is encountered. In above example this occurs between first id and +.

(2) Then scan backward over any = S until a $<\cdot$ is encountered. In above example we scan backward to $.

(3) The handle contains everything to the first $\cdot>$ and to the right of the $<\cdot$ encountered in step (2), including any intervening or surrounding non-terminals. This is necessary so that two adjacent non-terminals do not appear in a right sententional form. In the above example is the first id.

- The id is reduced to E and then we get the right sententional form E + id * id. After reducing two remaining ids, we get E + E * E. Now, the string can be obtained by deleting the non-terminals. Inserting the precedence relation.

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

- This relation indicate the right sententional form E + E * E and the handle is E * E. This can be reduced to E and then further reduced by E as E + E.

DESIGN OF LL(1) PARSER

Q.34. What is LL (1) grammar? Explain.

Ans. LL(1) grammar :

- In LL(1) the first L stands for left to right scan of the input, next L stands for the left most derivation and the (1) stands that next input symbol is used to decide the next parsing process.
- In LL(1) parsing system the parsing is done by scanning the input from left to right and input string is derived in leftmost order.
- The next input symbol is used to decide what is to be done in next parsing process.
- The predictive parser is therefore LL(1) because it scans the input from left to right and attempts to obtain the leftmost derivation of it.
- The given grammar is LL(1) only when its parsing table does not contain the multiple entries. If table contains multiple entries then it is not LL(1).

Q.35. Why we calculate FIRST and FOLLOW? Explain with respect to top-down parsing? ***CS : W-09(4M)***

OR Give significance of FIRST and FOLLOW set in relation with top down parsing. ***CS : S-10 (4M), CT : S-13(4M)***

OR Explain the significance of computing FIRST and FOLLOW. ***CT : S-10(2M)***

OR Why we need to compute FIRST and FOLLOW in LL(1) parsing? Explain with example. ***CS : W-14 (6M)***

Ans. Rules to calculate FIRST :

If $A \rightarrow XYZ$

(1) FIRST (A) = FIRST (XYZ) :

Conditions :

(a) IF X is terminal then simply write the FIRST of X and stop parsing here don't advance to Y

FIRST(A) = FIRST(X).

(b) If x is non terminal and FIRST(x) involves $\in$ then we need to minus $\in$ from the FIRST(X) and proceed to FIRST (Y).

FIRST(A) = FIRST(X) – $\in$ U FIRST(Y).

(c) FIRST(Y) and FIRST(Z) are also computed as above.

(d) FIRST is to be computed from bottom to up.

Significance of FIRST :

- In LL(1) parsing the information of the leftmost character of the non terminal should be obtained.

- This information is collected in the form of the set which is called as FIRST of every non terminal.
- For example, consider grammar

  $S \rightarrow a\,Ab$

  $A \rightarrow cd / ef$

  Here the FIRST(S) = {a}

  FIRST(A) = FIRST (cd) $\cup$ FIRST(ef) = {c, e}
- Thus while deriving S, the parser looks at the next input symbol in the string, if it happen to be a terminal 'a' then derive S using $S \rightarrow a\,Ab$ otherwise report errors.
- Thus after calculating FIRST of every non terminal, the parser can decide the right production for the derivation of required string.

Rules to calculate FOLLOW :

(1) If S is start symbol, then FOLLOW(S) always contains $.

(2) If there is a production

$A \rightarrow PQR$ and FIRST(R) does not contain $\in$ then the FOLLOW(Q) will be FIRST(R).

(3) If there is a production

$A \rightarrow PQR$ and FIRST(R) contains $\in$ then FOLLOW(Q) will be FIRST(R) $- \in$ is FOLLOW(A).

(4) FOLLOW of any non terminal can never be $\in$.

(5) If there is a production,

$A \rightarrow PQR$ then FOLLOW(R) will be FOLLOW(A).

Significance of FOLLOW( ) :

- For the grammar containing $\in$ production such as $A \rightarrow \in$, it is very difficult to decide when A is to be derived to $\in$, using the FIRST of right side of production.
- Hence to decide where the production $A \rightarrow \in$ is to be added to the table FOLLOW of every non terminal must be calculated.
- The derivation $A \rightarrow \in$ in needed when the parser is on the verge of expanding non terminal A and the symbol which is appearing in the input happen to be the terminal which occur immediately after A.
- Hence it is concluded that the production $A \rightarrow \in$ is to be added in the table at [A, b] for every b which is immediately follow A.
- Thus to compute set of all such terminals it is very necessary to calculate FOLLOW ( ).

Q.36. Write the algorithm for calculating FIRST and FOLLOW.

Ans. **(i) FIRST algorithm :**

```
Set R[X] = ϕ for all non-terminal X in G;
Repeat
For every non-terminal X in G do begin
For every production X → w do design
Let X1, X2, X3.....Xr = w;
rX : = 1;
more : = true;
while more do begin
if rX > r then begin
R[X] : = R[X] + [∈];
more : = false;
end
else if it is terminal (Xrx), then begin
R[X] : = R[X] + [XrX];
more : = false;
end
else begin
R[X] : = R[X] + {R[XrX]−[∈]};
if not nullable (XrX) then more : = false
end;
rX : = rX + 1;
end {while}
end {for}
end {for}
until no member of R[X] has been augmented.
```

(ii) FOLLOW algorithm :

```
For all tokens X in G do F[X]; = [ ];
Let S be the start token of G;
F[S]: = [1];
repeat
For every token X in G do
If not (X in [ϵ,1]) then begin
For (every production Z → w such that X appears in w) do
```

For (every appearance of X in w) do

begin Let w = a × $b_1, b_2 \ldots\ldots, b_r$;

(where $\alpha \in (N \cup \epsilon)$; and $b_i \in N$ to $1 < i < r$)

Let p: = 1; (p = position in β)

Let more : = true;

while more do begin

if p > r then begin

F[X]: = F[X] + F[Z];

more : = false;

end

else begin

F[X]: = F[X] + (FIRST $[b_{p,n}]$ – [ϵ])

If nullable (b_p) then p: = p + 1

else more : = false;

end {if}

end {while}

end {for}

end; if, {for}

until (no F (S) has been augmented)

Q.37. Show that no left recursive grammar can be LL(1).

CT : W-10, S-11 (3M)

OR Show that left recursive grammar is not LL(1). **CS : W-11 (3M)**

Ans.

- A grammar to be LL(1) for every pair of production $A \rightarrow \alpha | \beta$ in the grammar, FIRST (α) and FIRST (β) should be disjoint.
- If a grammar is left recursive then the set of production of G form $A \rightarrow A\alpha | \beta$ and, so FIRST ($A\alpha$) and FIRST (β) will not be disjoint sets.
- Therefore grammar containing a pair of production $A \rightarrow A\alpha | \beta$ i.e. a left recursive grammar can not be LL(1).

Q.38. Why we require to calculate FIRST (Non-terminal) in LL(1) parsing?

Ans.

(1) In LL(1) Parsing, first L stand for left to right scanning of string i.e. string will be checked characters by character by reading it from left to right.

(2) Because of these condition, a parse tree is to be constructed in the leftmost fashion i.e. what the second L stand for.

(3) This is to be achieved by predicting one character in each step, i.e. what one signifies in LL(1).

- In other words, in order to satisfy (1), (2) and (3), the most important condition, information about the leftmost character of each nonterminal should be obtained. That information if collected in the form of the set will be referred to as FIRST (every Non-terminal) and therefore in order to parse string in LL (1) fashion FIRST (every Non-terminal) must be calculated.

Q.39. State whether the following statements are TRUE / FALSE. Justify your answers briefly.

(i) If a grammar G contains a production $A \rightarrow \alpha/\beta$, then the grammar will not be LL(1) if following condition holds :

(a) $\beta \rightarrow \Sigma$ and

(b) FIRST (α) contains a production terminal symbol which is in FOLLOW (A).

(ii) If a grammar G contains a production pair $A \rightarrow \alpha/\beta$ and if α and β derive string beginning with same terminal then G is not LL(1) grammar.

Ans.

(i) The statement is TRUE, because if these conditions are satisfied, then the parser has more than one derivations possible if the next input symbol is the terminal which is common to both FIRST (α) and FOLLOW (A), and the symbol on the top of stack is A i.e. the parsing table will have multiple entries, hence the grammar is not LL(1).

(ii) The statement is TRUE, because if the above condition is satisfied, then the parser has more than one derivations possible if the next input symbol is the terminal which is common to both FIRST (α) and FIRST (β), and symbol on the top of stack is A, i.e. the parsing table will have multiple entries hence the grammar is not LL (1).

Q.40. Show that an ϵ-free LL (1) grammar can parse a sentence without FOLLOW ( ) set. **CT : S-11,W-10(3M)**

Ans.

- When grammar is ϵ-free, it can be decided that when the parser should do derivation using the productions by computing FIRST ( )

set of each production. But when the grammar is not ϵ-free, it is not possible when A should be derived using $A \rightarrow \epsilon$ in case of production $A \rightarrow \epsilon$.

- For this purpose, we must have to find out FOLLOW set. An ϵ-free LL (1) grammar can parse a sentence without FOLLOW ( ) set.

Q.41. Why an ambiguous grammar cannot be LL(1)? **CS : S-09(3M)**

Ans.

- The LL(1) grammar parsing table obtained by the application of LL(1) algorithm.
- Some of the entries in the parsing take may end up being multiple defined entries.
- If parsing table contain multiple entries then the parser is non deterministic.
- The parser will be a deterministic recognizer if and only if there are no multiple entries in the parsing table.
- All such grammar constitute a subset of CFGs called "LL(1)" grammar.
- As the parsing table entries of the LL(1) grammar are not multiple, so grammar is not ambiguous.
- Along with above condition the following condition must be satisfied :

  FIRST (α) $\cap$ FOLLOW (A) = ϕ

  for production $A \rightarrow \alpha | \beta$

  for every pair of production

  {

  (1) FIRST (α) $\cap$ FIRST (β) = ϕ and

  if FIRST (β) contains ϵ and FIRST (α) does not contain ϵ then

  (1) FIRST (α) $\cap$ FOLLOW (A) = ϕ

  }
- LL(1) satisfies both of the condition so LL(1) is not ambiguous grammar.

Q.42. IS LL(0) powerful than LL(1)? Justify. **CS : S-13(6M)**

Ans.

- LL(0) parser do look at the tokens, but they don't decide which production to apply upon them.
- They determine only if the sequence belongs to the language or not.
- This means that every non terminal symbol must have a single right hand side and that may be no recursion.
- The sequence of production to apply to parse an input with that grammar requires zero lookaheads so called as LL(0).
- In LL(0) languages, there are no choices, so the input sequence is either accepted and parsed or rejected.
- But in LL(1) grammar there is practically great interest as parser for these grammars are easy to construct and computer languages are designed to be LL(1).
- LL(1) parser generates 1 token of lookahead and can make more parsing decisions by recognizing whether following tokens belong to a regular language or not.
- In short, LL(0) is more weak or less powerful than LL(1) as the LL(0) parser has to purely base its decisions on current non-terminal means that it can't take one of many different action based on which productions might be used.

Q.43. Construct predictive parsing table for the following grammar and tell whether the grammar is in LL (1) or not :

S $\rightarrow$ (L)/a

L $\rightarrow$ L,S/S **CT : S-11(7M)**

Ans. Given grammar : S $\rightarrow$ (L)/a

L $\rightarrow$ L,S/S

Step 1 : In given grammar,

L $\rightarrow$ L, S/S is left recursion grammar.

By removing left recursion, we get

S $\rightarrow$ (L)/a

L $\rightarrow$ SL'

L' $\rightarrow$,SL'/ ϵ

Step 2 : Compute FIRST and FOLLOW :

FIRST (S) = {(, a)}

FIRST (L) = {FIRST (S)}

= {(, a)}

FIRST (L') = {, , ϵ}

FOLLOW (S) = {$} $\cup$ FIRST (L')

= {$} ∪ FIRST (L') - ϵ FOLLOW (L)

= {$} ∪ {, , ϵ} - ϵ ∪ {)}

= {$, , ,)}

FOLLOW (L) = {)}

FOLLOW (L') = {FOLLOW (L)}

= {)}

Step 3 : The predictive parsing table is :

N.T. \ T.	(	)	a	,	$
S	S→(L)		S → a		
L	L → SL'		L → SL'		
L'		L' → ϵ		L → , SL'	

It contains no multiple entries. Hence it is LL (1) grammar.

Q.44. Get LL(1) parsing table for following grammar

A → aCDq / aBg / ϵ

C → p / ϵ / C t / BD / rAb

D → d / ϵ

B → e / ϵ **CS : W-09(9M)**

Ans. Given grammar :

A → aCDq / aBg / ϵ

C → p / ϵ / C t / BD / rAb

D → d / ϵ

B → e / ϵ

Step 1 : In given grammar,

A → aCDq / aBg / ϵ

contains left recursion

Eliminate left factoring from A, we get

A → a A' / ϵ

A' → CDq / Bg

Also, production C i.e.

C → C t / p / ϵ / BD / rAb

contains left recursion.

So, by eliminating left recursion, we get

C → pC' / ϵ C' / BDC' / rAbC'

C → pC' / C' / BDC' / rAbC'

C' → tC' / ϵ

Therefore the grammar becomes

A → aA' / ϵ

A' → CDq / Bg

B → e / ϵ

C → C' / pC' / BDC' / rAbC'

C' → tC' / ϵ

D → d / ϵ

Step 2 : Compute FIRST and FOLLOW :

FIRST(A) = FIRST (aA') ∪ FIRST(ϵ)

= {a, ϵ}

FIRST(D) = FIRST(d) ∪ FIRST (ϵ)

FIRST(D) = {d, ϵ}

FIRST(B) = FIRST (e) ∪ FIRST (ϵ)

= {e, ϵ}

FIRST (C') = FIRST (tC') ∪ ϵ = {t, ϵ}

FIRST(C) = FIRST (pC') ∪ FIRST (tC') ∪ FIRST (BDC') ∪ FIRST (rAbC') ∪ FIRST (ϵ)

= {p} ∪ {t} ∪ {FIRST (B) – ϵ ∪ FIRST (DC')} ∪ {r} ∪ FIRST (ϵ)

= {p} ∪ {t} ∪ {r} ∪ {t}

= {p, t, e} ∪ {d} ∪ {t, ϵ} ∪ {r}

= {p, t, e, d, r, ϵ}

FIRST(A') = FIRST (CDq) ∪ FIRST (Bg)

= FIRST (C) – ϵ ∪ FIRST (Dq) ∪ FIRST (Bg)

= {p, t, e, d, r} ∪ {FIRST (D) – ϵ ∪ FIRST (q)} ∪ FIRST (Bg)}

= {p, t, e, d, r} ∪ {d, q} ∪ {FIRST (B) – {ϵ} ∪ FIRST (g)}

= {p, t, e, d, r,q} ∪ {e} ∪ {g}

= {p, t, e, d, r, q, g}

FOLLOW(A) = {$} ∪ FIRST (bC')

= {$, b}

FOLLOW (A') = FOLLOW (A)

= {$, b}

FOLLOW(C) = FIRST (Dq)

= FIRST(D) – ϵ ∪ FIRST (q)

= {d, q}

FOLLOW(B) = FIRST(g) ∪ FIRST (DC')

= {g} ∪ {FIRST (D) – ϵ ∪ FIRST (C') }

= {g} ∪ {{d} ∪ {FIRST (C') – ϵ ∪ FOLLOW (C)}}

= {g,d,t} ∪ {d,q}

= {g,d,t,q}

FOLLOW (C') = FOLLOW (C) = {d, q}

FOLLOW(D) = FIRST(q) ∪ FIRST (C')

= {q} ∪ {FIRST (C') – ϵ ∪ FOLLOW (C)}

= {q} ∪ {t} ∪ {d, q}

= {t, d, q}

Step 3 : Parsing table for given grammar :

T. / N.T.	a	b	d	e	g	r	t	q	p	$
A	A → aA'	A → ϵ								A → ϵ
A'			A' → CDq	A' → CDq	A' → Bg	A' → CDq	A' → CDq	A' → CDq	A' → CDq	
B			B → ϵ	B → e	B → ϵ		B → ϵ	B → ϵ		
C			C → BC'D C → ϵ	C → BDC'		C → rAbC'	C → BDC'	C → ϵ	C → pC'	
C'			C' → ϵ				C' → tC'	C' → ϵ		
D			D → d D → ϵ				D → ϵ	D → ϵ		

It contains multiple entries. Therefore it is not a LL(1) grammar.

Q.45. Construct LL (1) parsing table for given grammar:

S → aBDh

B → Bb/c

D → Eh

E → g / E. *CT : W-09(7M)*

Ans. Given grammar :

S → aBDh

B → Bb/c

D → Eh

E → g / E

Step 1 : In given grammar,

B → Bb/c and E → g/E

contains left recursion,

By eliminating left recursion, we get

B → cB'

$B' \rightarrow bB' / \epsilon$

and

$E \rightarrow gE'$

$E' \rightarrow E' / \epsilon$

The modified grammar becomes :

$S \rightarrow aBDh$

$B \rightarrow CB'$

$B' \rightarrow bB' / \epsilon$

$D \rightarrow Eh$

$E \rightarrow g / \epsilon$

Step 2 : Compute FIRST and FOLLOW :

FIRST (E) = FIRST (gE) $\cup$ { $\in$ } = {g, ϵ}

FIRST (D) = FIRST (E) = $\epsilon \cup$ FIRST (h)

= {g, ϵ} – {ϵ} $\cup$ {h}

= {g, h}

FIRST (B') = FIRST (bB') $\cup \epsilon$

= {b, ϵ}

FIRST (B) = FIRST (cB') = {c}

FIRST (S) = FIRST (aBDh) = {a}

FOLLOW (S) = {$}

FOLLOW (E) = FIRST (h) $\cup$ FOLLOW (E)

= {h}

FOLLOW (B) = FIRST (Dh)

= {g,h}

FOLLOW (B') = FOLLOW (B) $\cup$ FOLLOW (B')

= {g, h}

FOLLOW (D) = FIRST (h)

= {h}

Step 3 : LL (1) parsing table from above set result :

T. / N.T.	a	b	c	g	h	$
S	S → aBDh					
B			B → cB'			
B'		B' → bB'		B' → ϵ	B' → ϵ	
D				D → Eh	D → Eh	
E				E → gh	E → ϵ	

As grammar contains no multiple entries so grammar is LL (1).

Q.46. Get LL(1) parsing table for the following grammar and check validity of string "iaubde."

$S \rightarrow iLuET / a$

$L \rightarrow LS / \epsilon$

$E \rightarrow b$

$T \rightarrow dLe / \epsilon$

CS : W-13(8M),S-09(6M),S-14(7M)

Ans. Given grammar :

$S \rightarrow iLuET / a$

$L \rightarrow LS / \epsilon$

$E \rightarrow b$

$T \rightarrow dLe / \epsilon$

Step 1 : First eliminate left recursion from L

$L \rightarrow LS / \epsilon$

Self recursion gives

$L \rightarrow \epsilon L'$

$L' \rightarrow S L' / \epsilon$

as $L \rightarrow \leftarrow L'$

So

$L' \rightarrow S L' / \epsilon$

$L' \rightarrow S L / \epsilon$

So grammar becomes

$S \rightarrow iLuET / a$

$L \rightarrow SL / \epsilon$

$E \rightarrow b$

$T \rightarrow dLe / \epsilon$

Step 2 : Compute FIRST and FOLLOW :

FIRST(T) = FIRST (d Le) $\cup$ FIRST {ϵ}

= {d, ϵ}

FIRST(E) = FIRST {b}

= {b}

FIRST (L) = FIRST(SL) $\cup$ FIRST(ϵ)

= {i, a, } $\cup$ {ϵ}= {i, a, ϵ}

FIRST S) = FIRST (iL u ET) $\cup$ FIRST (a)

= { i, a}

FOLLOW(L) = FIRST(u ET) $\cup$ FOLLOW (L) $\cup$ FIRST (e)

= { u,e}

FOLLOW (S) = FIRST (L)

= {$} ∪ FIRST (L)

= {$, i, a, ∈}

FOLLOW (E) = FIRST (T)

= {d, ∈}

FOLLOW (T) = FIRST (S)

= {$, a, i, ∈ }

Step 3 : The predictive parsing table is :

N.T. \ T.	i	u	a	b	∈	e	d
S	S → iLuET		S → a				
L	L → SL		L → SL		L →∈		
E				E → b			
T					T →∈		T → dLe

As there is no multiple entries present in the grammar, so it is LL(1).

Step 4 : Steps involved in parsing the string "iaubde":

Stack contents	Unspent Input	Moves
$S	iaubde $	Derivation using S → iLuET
$ TEuLi	iaubde $	Pop i off the stack
$ TEuL	aubde $	Derivation using L → SL
$ TEuLS	aubde $	Derivation using S → a
$ TEuLa	aubde $	Pop a off the stack
$ TEuL	ubde $	Derivation using L → ∈
$ TEu	ubde $	Pop u off the stack
$ TE	bde $	Derivation using E → b
$ Tb	bde $	Pop b off the stack
$ T	de $	Derivation using T → dLe
$ eLd	de $	Pop d off the stack
$ eL	e $	Derivation using L → ∈
$ e	e $	Pop e off the stack
$	$	Announce successful completion of parsing.

Q.47. Construct LL(1) parser table for the following grammar.

S → aABb

A → c / ∈

B → d / ∈

Also show the sequence of action of LL(1) parse on input acdb. **CT : W-10(7M)**

Ans. **Given grammar :**

S → aABb

A → c / ∈

B → d / ∈

Step 1 : Compute FIRST and FOLLOW :

FIRST (S) = FIRST (aABb) = {a}

FIRST (A) = FIRST (c) ∪ FIRST (∈) = {c, ∈}

FIRST(B) = FIRST {d} ∪ FIRST {∈}= {d, ∈}

FOLLOW (S) = { $}

FOLLOW (A) = FIRST (Bb)

= FIRST (B) – { ∈ } ∪ FIRST (b)

= {d, ∈} – {∈} ∪ {b} = {d, b}

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

Step 2 : The predictive parsing table is :

N.T. \ T.	a	b	c	d	$
S	S → aABb				
A		A →∈	A → c	A →∈	
B		B →∈		B → d	

As table does not contain multiple entries, so given grammar is LL(1).

Step 3 : Steps involved in parsing the string acbd :

Stack contents	Unspent Input	Moves
$S	acdb$	Derivation using S → aABb
$bBAa	acdb$	Popping a off the stack and advancing one position in the input.
$bBA	cdb$	Derivation using A → c
$bBc	cdb$	Popping c off the stack and advancing one position in the input.
$bB	db$	Derivation using B → d
$bd	db$	Popping d off the stack and advancing one position in the input.
$b	b$	Popping b off the stack and advancing one position in the input.
$	$	Announce successful completion of parsing.

Q.48. Compute FIRST and FOLLOW and make parsing table for it

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Show the moves made by this LL(1) parser on input id + id * id. *CT : S-13(7M),S-14(13M)*

OR Modify the following CFG so as to make it sutiable for top-down parsing. Construct LL(1) parser for modified CFG. Show moves made by this LL(1) parser on input id+id*id. *CS : W-10,S-12(8M)*

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Ans. Given grammar :

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Step 1 : Eliminate left recursion from E and T prodcution, we get grammar as :

$E \rightarrow T E'$

$E' \rightarrow + T E' / \epsilon$

$T \rightarrow F T'$

$T' \rightarrow * FT' / \epsilon$

$F \rightarrow (E) / id$

Step 2 : Compute FIRST and FOLLOW set :

FIRST (F) = FIRST { (E) } $\cup$ FIRST {id}

= { (, id}

FIRST (T') = FIRST (* FT') $\cup$ FIRST (ϵ)

= {*, ϵ}

FIRST (T) = FIRST (FT')

= {(, id}

FIRST (E') = FIRST (+TE') $\cup$ FIRST (ϵ)

= {+, ϵ}

FIRST (E) = FIRST (TE') = { (, id}

FOLLOW (E) = {$,) }

FOLLOW (E') = FOLLOW (ϵ) = {$,) }

FOLLOW (T) = FIRST (E') – (ϵ) $\cup$ FOLLOW (E)

= {+, ϵ} – ϵ $\cup$ {$}

= {+, $}

FOLLOW (T') = FOLLOW (T)

= {$, +,}

FOLLOW (F) = FIRST {T'} – {∈} ∪ FOLLOW (T)

= {*, ∈} – {∈} ∪ {$, + }

= {*, $, + }

Step 3 : Predictive parsing table for given grammar :

T. / N.T	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → + TE'			E' → ∈	E' → ∈
T	E → FT'			T → FT'		
T'		T' → ∈	T' → * FT'			T' → ∈
F	F → id			F → (E)		

As there are no multiple entires in table, so grammar is LL (1).

Step 4 : Steps involved in parisng the string id+id *d :

Stack	Input	Moves
$ E	id+id*id $	Derivation using E → E + T`
$ T + E	id+id*id $	Derivation using E → T
$ T+T	id+id*id $	Derivation using T → F
$ T+F	id+id*id $	Derivation using F → id
$ T+id	id+id*id $	Pop off 'id' from stack
$ T+	+id*id $	Pop off '+' from stack
$ T	id*id $	Derivation using T → T * F
$ F * T	id*id $	Derivation using T → F
$ F * F	id*id $	Derivation using F → id
$ F * id	id*id $	Pop off 'id' from stack
$ F *	*id $	Pop off '*' from stack
$ F	id $	Derivation using F → id
$ id	id $	Pop off 'id' from stack
$	$	Announce completion

Q.49. Is the following grammar LL(1)?

S → aSA / ∈

A → bB / cc

B → bd / ∈

CS : S-10(5M)

OR Construct LL(1) parsing table for the grammar given below and show stack and buffer entries for the string 'aabbdcc' using parsing table.

S → aSA / ∈

A → bB / cc

B → bd / ϵ **CS : W-11(10M)**

Ans. Given grammar :

S → aSA / ϵ

A → bB / cc

B → bd / ϵ

Step 1 : Compute FIRST and FOLLOW :

FIRST (S) = FIRST (aSA) ∪ FIRST (ϵ)

= {a} ∪ {ϵ}

= {a, ϵ}

FIRST (A) = FIRST (bB) ∪ FIRST (cc)

= {b} ∪ {c}

= {b, c}

FIRST (B) = FIRST (bd) ∪ FIRST (ϵ)

= {b} ∪ {ϵ}

= {b, ϵ}

FOLLOW(S) = {$} ∪ FIRST (A)

= {$, b, c}

FOLLOW (A) = FOLLOW(S)

= {$, b, c}

FOLLOW(B) = FOLLOW(A)

= {$, b, c}

Step 2 : Parsing table for given production :

T. / N.T.	a	b	c	$
S	S → aSA	S → ϵ	S → ϵ	S → ϵ
A		A → bB	A → cc	
B		B → bd B → ϵ	B → ϵ	B → ϵ

∴ As table contains multiple entries, so given grammar is not LL(1)

Step 3 : Moves for parsing string 'aabbdcc' is as follows :

Stack	Input	Moves
$ aSA	aabbdcc $	Derivation using S → aSA
$ ASa	aabbdcc $	Pop off 'a' from stack
$ AS	abbdcc $	Derivation using S → aSA
$ AASa	abbdcc $	Pop off 'a' from stack
$ AAS	bbdcc $	Derivation using S → ϵ
$ AA	bbdcc $	Derivation using A → bB
$ ABb	bbdcc $	Pop off 'b' from stack
$ AB	bdcc $	Derivation using B → bd
$ Adb	bdcc $	Pop off 'b' from stack
$ Ad	dcc $	Pop off 'd' from stack
$ A	cc $	Derivation using A → cc
$ cc	cc $	Pop off 'c' from stack
$ c	c $	Pop off 'c' from stack
$	$	Announce successful completion of the parsing.

Q.50. Test whether the following grammar is LL(1) or not :

S → A #

A → Ad / aB / aC

C → c

B → bBC / r . **CT : S-10(7M)**

Ans. Given grammar :

S → A #

A → Ad / aB / aC

C → c

B → bBC / r .

Step 1 : The grammar contains left factoring in production A → Ad / aB / ac

∴ By eliminating left factoring, we get,

A → Ad / aA′

A′ → B / c

Also, the grammar contains left recursion in production

A → Ad / aA′

∴ By eliminating left recursion, we get

A → aA′ A″

A″ → dA″ / ϵ

∴ The grammar becomes

S → A #

A → aA′ A″

A″ → dA″ / ϵ

A′ → B / C

C → c

B → bBC / r

Step 2 : Compute FIRST and FOLLOW :

FIRST (B) = FIRST (bBC) ∪ FIRST (r)

= {b, r}

FIRST (C) = FIRST{c} = {c}

FIRST (A') = FIRST (B) ∪ FIRST (C)

= {b, r} ∪ {c} = {b, r, c}

FIRST (A'') = FIRST (dA'') ∪ FIRST (ϵ)

= {d, ϵ}

FIRST (A) = FIRST (aA'A'') = {a}

FIRST (S) = FIRST (A #)

= {a}

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (#) = {#}

FOLLOW (A') = FIRST (A'') – ϵ ∪ FOLLOW (A)

= {d, ϵ} – ϵ ∪ {#}

= {d, #}

FOLLOW (A'') = FOLLOW (A) = {#}

FOLLOW (B) = FOLLOW (A') = {d, #}

FOLLOW(C) = FOLLOW (A') ∪ FOLLOW (B)

= {d, #}

Step 3 : Parsing table for given grammar :

T. / N.T.	#	a	d	c	r	b	$
S		S → A #					
A		A → aA' A''					
A'				A' → C	A' → B/C	A' → B	
A''	A'' → ϵ		A'' → dA''			B → bBc	
B					B → r		
C				C → C			

∴ As table does not contain multiple entries, so the given grammar is LL(1).

Q.51. Consider the following grammar over the alphabet {a, b, d}

S → ABD

A → a/BSB

B → b/D

D → d/ϵ

Construct LL (1) parsing table for the above grammar and check whether the grammar is LL (1) or not. ***CT : W-11(6M)***

Ans. Given grammar :

S → ABD

$A \rightarrow a/BSB$

$B \rightarrow b/D$

$D \rightarrow d/\in$

Step 1 : Compute FIRST and FOLLOW :

FIRST (D) = {d, ∈}

FIRST (B) = {b, d, ∈ }

FIRST (A) = FIRST (a) ∪ FIRST (BSB)

= {a} ∪ FIRST (B) – { ∈} ∪ FIRST (SB)

= {a} ∪ {b, d, ∈} – {∈} ∪ FIRST (S) – {∈} ∪ FIRST (B)

= {a, b, d, ∈)

FIRST (S) = FIRST (ABD)

= FIRST(A) – {∈} ∪ FIRST (B) – {∈} ∪ FIRST (D)

= {a, b, d, ∈)

FOLLOW (S) = {$} ∪ FIRST (B) – {∈} ∪ FOLLOW (A)

= {$} ∪ {b, d, ∈} – {∈} ∪ FOLLOW (A)

= {b, d, $} ∪ FOLLOW (A) ....(i)

FOLLOW (A) = FIRST (B) – {∈} ∪ FIRST (D)

– {∈} ∪ FOLLOW(s)

= {b, d} ∪ {d, ∉} – {∉} ∪ FOLLOW(s)

= {b, d} ∪ FOLLOW (s) ...(ii)

Solving eq. (i) & eq.(ii)

FOLLOW(S) = {b, d, $} ∪ {b, d} ∪ FOLLOW (S)

= {b, d, $} ∪ {b, d}

= {b, d, $) ...(iii)

Putting value of FOLLOW(s) of eq. (iii) in eq. (ii)

FOLLOW (A) = {b, d} ∪ {b, d, $}

= {b, d, $}

FOLLOW (B) = FIRST (D) – { ∈} ∪ FOLLOW(S) ∪ FIRST (S) –

{∈} ∪ FIRST (B) – {∈} ∪ FOLLOW (A)

= {d} ∪ {b, d, $} ∪ {a, b, d} ∪ {b, d} ∪ {b, d, $}

= {a, b, d, $}

FOLLOW(D) = FOLLOW(S) ∪ FOLLOW (B)

= {b, d, $} ∪ {a, b, d, $}

= {a, b, d, $}

Step 2 : Parsing table for given grammar is :

T. / N.T.	a	b	d	$
S	S → ABD	S → ABD	S → ABD	S → ABD
A	A → a	A → BSB	A → BSB	A → BSB
B	B → D	B → D B → D	B → D	B → D
D	D → ∈	D → ∈	D → d D → ∈	D → ∈

Since the parsing table contains multiple entries therefore the grammar is not LL(1).

Q.52. Find FIRST and FOLLOW for the following grammar. Construct the parsing table and find out whether the grammar is LL (1) or not

E → 10 * T / 5 + T

T → PS

S → QP / E

Q → + / *

P → a / b / c *CS : W-12(10M)*

Ans. Given grammar :

E → 10 * T / 5 + T

T → PS

S → QP /E

Q → + / *

P → a / b / c

Step 1 : Compute FIRST and FOLLOW :

FIRST (P) = FIRST (a) ∪ FIRST (b) ∪ FIRST (c)

= {a, b, c}

FIRST (Q) = {+, *}

FIRST (S) = FIRST (QP) ∪ FIRST (E) ....(1)

FIRST (T) = FIRST (PS) = {a, b, c}

FIRST (E) = FIRST (10*T) ∪ FIRST (5+T)

= {10,5}

From (1) :

FIRST (S) = {+, *, 10,5}

As, there is no ∈ production in given grammar, so no need to calculate FOLLOW.

Step 2 : Parsing table for given grammar :

T. / N.T.	10	*	5	+	a	b	c
E	E → 10*T		E → 5+T				
T					T → PS	T → PS	T → PS
S	S → QP/E	S → QP/E	S → QP/E	S → QP/E			
Q		Q → *		Q → +			
P					P → a	P → b	P → c

As table doesn't contain multiple entries, so given grammar is LL(1).

Q.53. Construct the LL(1) parsing table for the given CFG :

S → {P}

P → P; P/ ∈

P → a/b *CT : W-12(7M)*

Ans. Given grammar :

S → {P}

P → P; P/ ∈

P → a/b

Step 1 : Given grammar contain left recursion in production P → P ; P / ∈

∴ By eliminating left recursion, we get

P → P'/ a P'/ b P'

P' →; PP' / ∈

∴ The grammar becomes,

S → {P}

P → P'/ aP'/bP'

P' → ; PP' / ∈

Step 2 : Compute FIRST and FOLLOW :

FIRST (S) = FIRST ({P}) = {{}

FIRST (P) = FIRST (P') ∪ FIRST (aP') ∪ FIRST (bP') ....(1)

FIRST (P') = FIRST (; PP') ∪ ∈

= {; , ∈}

From (1) :

FIRST (P) = {a, b, ; , ∈ }

FOLLOW (S) = {$}

FOLLOW (P) = FIRST (}) ∪ FIRST (P')

− ∈ ∪ FIRST (P) − ∈ ∪ FIRST (})

= {}} ∪ { ;, ∈} − ∈ ∪ {a, b, ; , ∈} − ∈ ∪ (})

= { }, ; , a, b}

FOLLOW (P') = FOLLOW (P) = {; ,} , a, b}

Step 3 : Parsing table for given grammar is as follows :

T. / N.T.	{	}	a	b	;	$
S	S → {P}					
P		P → ∈	P → aP' P → ∈	P → bP' P → ∈	P → P' P → ∈	
P'		P' → ∈	P' → ∈	P' → ∈	P' → ;PP' P' → ∈	

As there are multiple entries in table, so given grammar is not LL (1).

Q.54. Compute LL (1) parsing table for the following grammar :

S → aAB / bA / ∈

A → aAb / ∈

B → bB / c *CS : W-14(7M)*

Ans. **Given grammar :**

$S \rightarrow aAB / bA / \epsilon$

$A \rightarrow aAb / \epsilon$

$B \rightarrow bB / c$

Step 1 : Compute FIRST and FOLLOW :

FIRST (S) = FIRST (aAB) ∪ FIRST (bA) ∪ ϵ

= {a, b, ϵ}

FIRST (A) = FIRST (aAb) ∪ ϵ = {a, ϵ}

FIRST (B) = FIRST (bB) ∪ FIRST (c) = {b, c}

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (B) ∪ FOLLOW (S) ∪ FIRST (b)

= {b, c} ∪ {$} ∪ {b} = {b, c, $}

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

Step 2 : Parsing table for above grammar :

T. / N.T.	a	b	c	$
S	S → aAB	S → bA		S → ϵ
A	A → aAB	A → ϵ	A → ϵ	A → ϵ
B		B → bB	B → c	

As above table does not contain multiple entries, so given grammar is LL(1).

Q.55. Show that the language is LL (1) or not :

L {$a^n cb^n / n \geq 1$} . ***CT : W-13(7M)***

Ans. **Given language :**

L {$a^n cb^n / n \geq 1$}

Step 1 : It can be represented by the context free grammar as follows .

$S \rightarrow aAb$

$A \rightarrow aAb / c$

Step 2 : Compute FIRST and FOLLOW :

FIRST (S) = FIRST (aAb) = {a}

FIRST (A) = FIRST (aAb) ∪ FIRST (c) = {a, c}

FOLLOW (S) = {$}

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

Step 3 :

As A have multiple production so check intersection for it.

FIRST (aAb) ∩ FIRST (c) = ϕ

. Grammar is LL(1)

Step 4 : Parsing table for given grammar :

T. / N.T.	a	b	c	$
S	S → aAb			
A	A → aAb		A → c	

As above table does not contain multiple entries, so given grammar is LL(1).

Q.56. Get LL(1) parsing table for following grammar and check validity of string 'acbh'.

A → aBDh

B → cC

C → bc / ϵ

D → EF

E → g / ϵ

F → f/ ϵ ***CS : S-13(7M)***

Ans. **Given grammar :**

A → aBDh

B → cC

C → bc / ϵ

D → EF

E → g / ϵ

F → f/ ϵ

Step 1 : Compute FIRST and FOLLOW :

FIRST (A) = FIRST (aBDh) = {a}

FIRST (B) = FIRST (cC) = {c}

FIRST (C) = FIRST (bc) ∪ ϵ = {b, ϵ}

FIRST (D) = FIRST (EF) ....(1)

FIRST (E) = FIRST (g) ∪ ϵ = {g, ϵ}

FIRST (F) = FIRST (f) ∪ ϵ = {f,ϵ}

From (1) ⇒

FIRST (D) = FIRST (E) – {ϵ} ∪ FIRST (F)

= {g, ϵ} – {ϵ} ∪ {f, ϵ}

= {g, f, ϵ}

FOLLOW (A) = {$}

FOLLOW (B) = FIRST (Dh)

= FIRST (D) – {ϵ} ∪ FIRST (h)

= {g, f, ϵ} – {ϵ} ∪ {h}

= {g, f, h}

FOLLOW (D) = FIRST (h) = {h}

FOLLOW (C) = FOLLOW (B) = {g, f, h}

FOLLOW (E) = FIRST (F)

= FIRST (F) – {∈} ∪ FOLLOW (D)

= {f, ∈} – {∈} ∪ {h}

= {f, h}

FOLLOW (F) = FOLLOW (D) = {h}

Step 2 : Parsing table for given grammar :

T. / N.T.	a	b	c	g	f	h	$
A	A → aBDh						
B			B → cC				
C		C → bC		C → ∈	C → ∈	C → ∈	
D				D → EF	D → EF		
E				E → g	E → ∈	E → ∈	
F					F → f	F → ∈	

∴ As the table does not contain multiple entries, so the given grammar is LL(1)

Step 3 : Moves for checking validity of string 'acbh' :

Stack	Input	Moves
$A	acbh$	Derivation using A → aBDh
$hDBa	acbh$	Pop off 'a' from stack.
$hDB	cbh$	Derivation using B → cC
$hDCc	cbh$	Pop off 'c' from stack.
$hDC	bh$	Derivation using C → bC
$hDCb	bh$	Pop off 'b' from stack.
$hDC	h$	Derivation using C → ∈
$hD	h$	Derivation using D → EF
$hFE	h$	Derivation using E → ∈
$hF	h$	Derivation using F → ∈
$h	h$	Pop off 'h' from stack
$	$	Announce successful completion.

Q.57. Transform the following grammar so that it will be LL (1), without changing the language. Hence construct LL(1) parsing table for the modified grammar

S → a AC / bB

A → Abc / Abd / e

B → f / g

C → h / i

Ans. Given grammar : $S \rightarrow a\,AC / bB$

$A \rightarrow Abc / Abd / e$

$B \rightarrow f / g$

$C \rightarrow h / i$

Step 1 : Apply left factoring we get,

$S \rightarrow aAC / bB$

$A \rightarrow e\,A'$

$A' \rightarrow bcA' / bdA' / \epsilon$

$B \rightarrow f / g$

$C \rightarrow h / i$

Step 2 : Compute FIRST and FOLLOW :

FIRST (S) = {a, b}

FIRST (A) = {e}

FIRST (A') = {b, e}

FIRST (B) = {f, g}

FIRST (C) = {h, i}

FOLLOW (S) = {$}

FOLLOW (B) = FOLLOW (S)

= {$}

FOLLOW (C) = {FOLLOW (S) $\cup$ FIRST (A)}

= {$} $\cup$ {e}

= {$, e}

FOLLOW (A) = {c}

FOLLOW (A') = FOLLOW (A)

={C}

Step 3 : Paring table for given grammar :

T. / N.T.	a	c	b	d	$	e	f	g	h	i	ϵ
S	$A \rightarrow aAc$		$S \rightarrow bB$								
A	$A \rightarrow e$										
B							$B \rightarrow f$	$B \rightarrow g$			
C									$h \rightarrow \epsilon$	$C \rightarrow i$	
A' E			$A' \rightarrow bcA'$							$A \rightarrow \epsilon$	

As table does not contain multiple entries, hence given grammar is LL(1)

LR PARSING

Q.58. Explain LR Parser.

Ans. LR parser :

- LR parsers are the bottom-up-parser.
- These parsers are called LR parsers, because they scan the input from left to right and construct a rightmost derivation in reverse.
- This technique is generally called as LR(k) parsing ; the 'L' stands for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse and the k for number of input symbols of lookahead that are used in making parsing decisions.
- When (k) is omitted, it is assumed to be 1.
- LR parsing is attractive for following reasons :

(1) LR parsers can be constructed to recognize virtually all programming language constructs for which context -free grammars can be written.

(2) LR parsing method is the most general than any other parser.

(3) The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

(4) An LR parser can detect a syntactic error as soon as it is possible to do so on a left to right scan of the input.

- There are three widely used algorithms available for constructing an LR parser :

(1) **SLR (1) :**

- It is a simple LR parser.
- It works on smallest class of grammar.
- This is simple and fast construction.

(2) **LR (1) :**

- It works on complete set of LR(1) grammar.
- It generates large table and large number of states.
- This has slow construction.

(3) **LALR (1) :**

- It is a lookahead LR parser.
- It works on intermediate size of grammar.
- Its number of states are same as in SLR(1).

Operation at the parser :

- Logically, LR parser consists of two parts, a driver routine and a parsing table.
- The driver routine is the same for all LR parsers, only the parsing table changes from one parser to another.

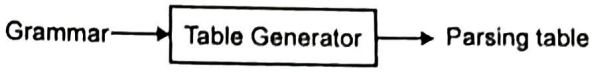


Fig.(a) Generating the parser

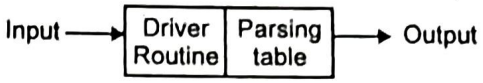


Fig.(b) Operation at the parser

- The schematic form of an LR parser is shown in fig.(c).
- It consists of an input, an output, a stack, a driver program and a parsing table that has two parts i.e. action and goto.
- The driver program is the 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.
- The program uses a stack to store a string of the form $S_0X_1 S_1X_2 S_2.....X_m S_m$, where S_m is on top. Each X_i is a grammar symbol and each S_i is a symbol called a state.

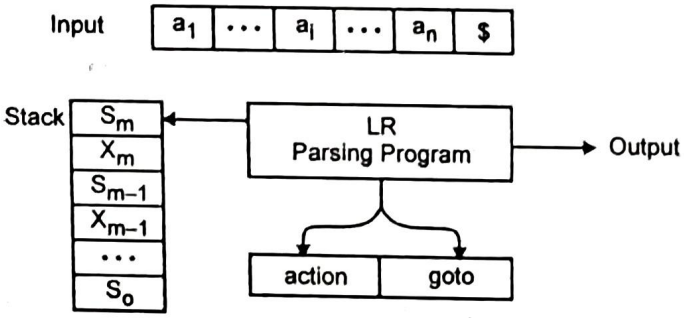


Fig.(c) Model of an LR parser.

- The parsing table consists of two parts, a parsing action function 'action' and a goto function 'goto'.

Q.59. Describe data structures used for storing LR parsing tables.

CS : W-10(5M)

OR How can LR parsing table be implemented?

CT : S-11(3M)

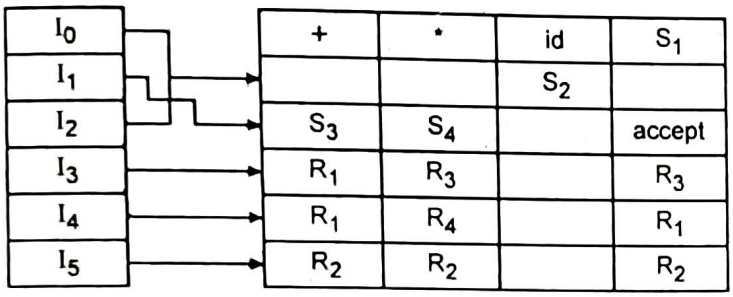
Ans.

- LR parsing table entries are made of two functions :

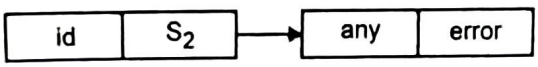
Parsing ACTION function and GOTO function.

Representation of ACTION table :

- The simplest way of representing action table is to use 2D array, but a parser with several hundred states may have thousands of entries.
- Thus efficient storage utilization, more space, for encoding is required.
- Since many rows of action table are identical, an array of pointers are calculated for each state to save considerable space at negligible cost.
- The pointer for the states with same action point to the same location as shown below :



- To access information, a number is assigned to each terminal from zero to one less than the number of terminals and use this integer as an offset from the pointer value of each state.
- Further reduction in space is possible by creating a list of action for each state, containing pair of terminal symbols and action for that terminal symbol.
- The most frequent action like error action can be appended at the end of the k list as shown in below :
- For state I_0, in above parsing table, the list will be,



Representation of GOTO table :

- The effective way of representing the GoTo table is to make a list of pairs for each non terminal A.
- Each pair is of the form (current state, next state) which indicates, GOTO C (current state, A) = next state.
- The error entries in GOTO table are never consulted. We can replace each error entry by the most common non error entry in its column is represented by any in place of current state.

Q.60. Comment on following statements :

(1) Left recursive grammar is not suitable for TOP-DOWN parsing.

(2) Every unambiguous grammar is LR grammar.

CS : S-11(8M)

Ans.

(1) Suppose we have left recursive production rule

$A \rightarrow Aa / b$

and now we try to match the rule.

- So we are checking whether we match an A here, but in order to do that, we must first check whether we can match an A here. That is impossible and it mostly is.
- Using recursive descent parser that obviously represent an infinite recursion.
- So, using elimination of the left recursion, the infinite recursion execution can be eliminated.
- As top down parsing algorithm builds the parse tree from the top (start symbol) to down.
- If there is left recursion in this condition the top down parser doesn't work.
- Top down parsers can not handle left recursion as :

A grammar is left recursive if $\exists A \in NT$ such that $\exists$ a derivation $A + A\alpha$ for some string $\alpha \in (NT \cup T) +$.

- If top down grammar is left recursive then.

(i) This can lead to non termination in a top down parser.

(ii) For a top down parsers any recursion must be right recursion.

(iii) So we have to convert left recursion to right recursion.

(iv) So non termination or infinite recursion is bad property of compiler.

- So top down parser doesn't contain left recursion.

(2) Every unambiguous grammar is LR because ambiguous grammars are often simpler.

- For instance, compare

$E \rightarrow E + E / E * E / (E) / a$

to the equivalent unambiguous grammar

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / a$

- The second grammar has important virtues though :

(i) It is unambiguous.

(ii) It reflects the fact that operators + and * are left associative.

(iii) It reflects the fact that * has higher precedence than +,

$E \rightarrow E + E / E * E / (E) / a$

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / a$

- Unambiguous grammar generates every sentence exactly one parse tree that respects the stipulations of higher precedence of operators.
- So, LR grammars is unambiguous in nature.

Q.61. Comment on truth / falsehood : "LL(0) is less powerful than LR(0)". Justify your answer. **CS : W-13(5M)**

Ans.

- At right level, the difference between LL parsing and LR parsing is that LL parsers begin at the start symbol and try to apply production to arrive at the target string whereas LR parser begins at the target string and try to arrive back at the start symbol.
- An LL parser is a left to right, left most derivation i.e. we consider the input symbol from the left to the right and attempt to construct a left most derivation.
- This is done by beginning at the start symbol and repeatedly expanding out the leftmost non terminal until we arrive at the target string.
- An LR parser is a left to right, rightmost derivation, meaning that we can scan from left to right and attempt to construct a rightmost derivation.
- During LL parser, the parser continuously choose between two actions :

(1) **Predict :** Based on leftmost non terminal and some number of lookahead tokens, choose which production ought to be applied to get closer to input string.

(2) **Match :** Match the leftmost guessed terminal symbol with the leftmost unconsumed symbol of input.

- In LR parser, there are two actions.

(1) **Shift :** Add next token of input to a buffer for consideration.

(2) **Reduce :** Reduce the collection of terminals and non terminals in this buffer back to some non terminal by reversing a production.

- LL parser tend to be easier to write by hand, but they are less powerful than LR parser and accept a much smaller set of grammars than LR parser do.
- LR parser are far more powerful as they tend to have much more complex and are almost always generated by tools like Yacc and bison.

Q.62. Define following terms in reference to LR parsing table construction :

(1) Augmentation

(2) LR(0) item

(3) Closure of LR(0) item. **CS : W-13(6M)**

Ans.

(1) Augmentation :

- To construct DFA that recognizes the viable prefixes we make use of augmented grammar which is defined as follows,

  if G = (V, T, P, S) is given grammar than augmented grammar will be $G_1 = (V \cup \{S_1\}, \quad T, P \cup \{S_1 \rightarrow S\}, S_1\}$; i.e. we add unit production $S_1 \rightarrow S$ to the grammar G and make S_1 the new start symbol. The resulting grammar will be an augmented grammar.
- The purpose of augmented grammar is to make it explicitly clear to parse when to accept the string.
- The parsing will stop when the parser is on verge of carrying out the reduction using $S_1 \rightarrow S$.
- A NFA that recognizes the viable prefixes will be a finite automata whose states correspond to the production items of augmented grammar.
- Every item represents one state in the automata with initial state corresponding to an item $S_1 \rightarrow S$.
- The transitions in the automata are defined as follows :

  $\delta(A \rightarrow \alpha . X \beta, X) = A \rightarrow \alpha X . \beta$

  $\delta(A \rightarrow \alpha . B \beta, \epsilon) = \beta \Rightarrow . \gamma$
- This NFA can then be transformed into a DFA using subset construction method.

Example :

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id$

The augmented grammar is as follows :

$E' \rightarrow E$

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id$

(2) LR(0) item :

- The set of items that corresponds to the states of DFA that recognize viable prefixes is called a "canonical collection".
- An algorithm exists that directly obtains canonical collection of LR(0) sets of items, thereby allowing us to obtain the DFA.
- Using this algorithm, we can directly obtain a DFA that recognizes the viable prefixes, rather than going through NFA to DFA transformation.
- The algorithm uses finding LR(0) sets and goto function.

An algorithm to find canonical collection of sets of LR(0) items :

- Let C be canonical collection of set of LR(0) items we maintain C_{new} and C_{old} to continue iteration.

Input : Augmented grammar.

Output : Canonical collection of sets of LR(0) items i.e. set C.

(1) $C_{old} = \phi$

(2) Add closure $(\{S_1 \rightarrow S\})$ to C.

(3) While $C_{old} \neq C_{new}$ do

{

temp = $C_{new} - C_{old}$

$C_{old} = C_{new}$

for every I in temp do

for every X in VUT (i.e. for every grammar symbol X)

do

if goto (I, X) is not empty and not in C_{new} then

add goto (I, X) to C_{new}

}

(4) $C = C_{new}$

(3) Closure of LR(0) item :

- The set closure (I) where I is a set of items is computed as follows :

(1) Add every item in I to closure (I).

(2) Repeat

for every item of the form

$A \rightarrow \alpha . B \beta$ in closure (I) do.

for every production $B \rightarrow \gamma$ do

add $B \rightarrow .\gamma$ to closure (I).

Until no new item can be added to closure (I).

Example :

If set I = { $E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .id$

}

then

goto (I, T) = Closure ({

$E \rightarrow T .$

$T \rightarrow T . * F$

})

= { $E \rightarrow T .$

$T \rightarrow T . * F$

}

Q.63. Explain various conflicts which arise during LR parsing.

CT : W-09(4M), S-09(3M)

Ans. Parser conflicts :

LR parser may encounter two types of conflicts :

(1) Shift-reduce conflicts.

(2) Reduce-reduce conflicts.

(1) Shift-reduce conflict :

- A shift-reduce (S-R) conflict occurs in an SLR parser state if the underlying set of LR(0) item representations contains items of the form depicted in Figure and FOLLOW (B) contains terminal a.

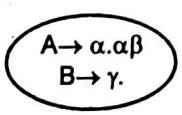


Fig.(a) LR(0) underlying set representations that can cause SLR parser conflicts.

- Similarly, an S-R conflict occurs in a state of the CLR or LALR parser if the underlying set of LR(1) items representation contains items of the form shown in Fig.(b)

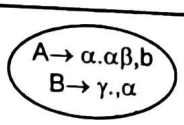


Fig.(b) LR(1) underlying set representations that can cause CLR/LALR parser conflicts.

(2) Reduce-reduce conflict :

- A reduce-reduce (R-R) conflict occurs if the underlying set of LR(0) items representation in a state of an SLR parser contains items of the form shown in Fig.(c) and FOLLOW(A) and FOLLOW(B) are not disjoint sets.

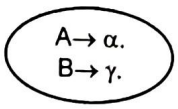


Fig.(c) LR(0) underlying set representations that can cause an SLR parser reduce-reduce conflicts.

- Similarly, an R-R conflict occurs if the underlying set of LR(1) items representation in a state of a CLR or LALR parser contains items of the form shown in fig.(d)

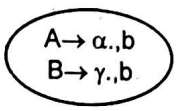


Fig.(d) LR(1) underlying set representations that can cause an CLR/LALR parser.

Q.64. Differentiate between LL and LR parsing.

Ans.

Sr. No.	LL	LR
(1)	It does a leftmost derivation.	It does a rightmost derivation in reverse.
(2)	It starts with the root non terminal on the stack.	It ends with the root non-terminal on the stack.
(3)	It ends when the stack is empty.	It starts with an empty stack.
(4)	Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
(5)	Builds the parse tree top-down.	Builds the parse tree bottom-up.
(6)	Continuously pops a non-terminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding non-terminal.
(7)	Expands the non-terminals.	Reduces the non-terminals.
(8)	Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
(9)	Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Q.65. Construct LR (0) parsing table for the following grammar.

$E \rightarrow E + E / E * E / (E) / id$. **CS : S-10(10M)**

Ans. Given grammar :

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

Step 1 : The augmented grammar :

$E' \rightarrow .E$

$E \rightarrow .E + E$ .....(I)

$E \rightarrow .E * E$ ....(II)

$E \rightarrow .(E)$ ....(III)

$E \rightarrow .id$ ....(IV)

Step 2 : Construct LR (0) parsing states :

$$I_0 = \text{Closure}(E' \rightarrow .E) = \left\{ \begin{matrix} E' \rightarrow .E \\ E \rightarrow .E + E \\ E \rightarrow .E * E \\ E \rightarrow .(E) \\ E \rightarrow .id \end{matrix} \right\} - (I_0)$$

$$\text{goto } (I_0, E) = \left\{ \begin{matrix} E' \rightarrow E. \\ E \rightarrow E. + E \\ E \rightarrow E. * E \end{matrix} \right\} - (I_1)$$

$$\text{goto } (I_0, () = \left\{ \begin{matrix} E \rightarrow (.E) \\ E \rightarrow .E + E \\ E \rightarrow .E * E \\ E \rightarrow .(E) \\ E \rightarrow .id \end{matrix} \right\} - (I_2)$$

$$\text{goto } (I_0, id) = \{E \rightarrow id.\} - (I_3)$$

$$\text{goto } (I_1, +) = \left\{\begin{matrix} E \to E + .\, E \\ E \to .\, E + E \\ E \to .\, E * E \\ E \to .\, (E) \\ E \to .\, id \end{matrix}\right\} - (I_4)$$

$$\text{goto } (I_1, *) = \left\{\begin{matrix} E \to E * .\, E \\ E \to .\, E + E \\ E \to .\, E * E \\ E \to .\, (E) \\ E \to .\, id \end{matrix}\right\} - (I_5)$$

$$\text{goto } (I_2, E) = \left\{\begin{matrix} E \to (\ E .) \\ E \to E .\, + E \\ E \to E .\, * E \end{matrix}\right\} - (I_6)$$

$$\text{goto } (I_2, () = \left\{\begin{matrix} E \to (\, .\, E) \\ E \to .\, E + E \\ E \to .\, E * E \\ E \to .\, (E) \\ E \to .\, id \end{matrix}\right\} - \text{same as } (I_2)$$

$$\text{goto } (I_2, id) = \{E \to id\ .\} - \text{same as } (I_3)$$

$$\text{goto } (I_4, E) = \left\{\begin{matrix} E \to E + E . \\ E \to E .\, + E \\ E \to E .\, * E \end{matrix}\right\} - (I_7)$$

$$\text{goto } (I_4, () = \left\{\begin{matrix} E \to (\, .\, E) \\ E \to .\, E + E \\ E \to .\, E * E \\ E \to .\, (E) \\ E \to .\, id \end{matrix}\right\} - \text{same as } (I_2)$$

$$\text{goto } (I_4, id) = \{E \to id\ .\} - \text{same as } (I_3)$$

$$\text{goto } (I_5, E) = \left\{\begin{matrix} E \to E * E . \\ E \to E .\, + E \\ E \to E .\, * E \end{matrix}\right\} - (I_8)$$

$$\text{goto } (I_5, () = \left\{\begin{matrix} E \to (\, .\, E) \\ E \to .\, E + E \\ E \to .\, E * E \\ E \to .\, (E) \\ E \to .\, id \end{matrix}\right\} - \text{same as } (I_2)\,.$$

$$\text{goto } (I_5, id) = \{E \to id\ .\} - \text{same as } (I_3)$$

$$\text{goto } (I_6, +) = \text{same as } (I_4)$$

$$\text{goto } (I_6, *) = \text{same as } (I_5)$$

$$\text{goto } (I_6,)) = E \to (E)\,. - (I_9)$$

$$\text{goto } (I_7, +) = \text{same as } (I_4)$$

$$\text{goto } (I_7, *) = \text{same as } (I_5)$$

Step 3 : Parsing table would be computed as follows :

T. / N.T.	ACTION						GOTO
	id	+	*	(	)	$	E
I_0	S_3				S_2		1
I_1		S_4	S_5			accept	
I_2	S_3				S_2		6
I_3		R_4	R_4			R_4	
I_4	S_3				S_2		7
I_5	S_3				S_2		8
I_6		S_4	S_5		S_9		
I_7		R_1	S_5		R_1		
I_8		R_2	R_2		R_2		
I_9		R_3	S_3		R_3		

As it doesn't contain multiple entries, so grammar is LR (0).

Q.66. Obtain conflict free parsing table for following grammar using LR parsing method

$S \to iCSeS / iCS / a$.

CS : W-09(13M), W-13(8M)

Ans.

Step 1 : Augmented grammar :

$S' \to .\, S$

$S \to .\, iCSeS$

$S \to .\, iCS$

$S \to .\, a$

Step 2 : Parsing action for given grammar is as follows :

$$I_0 = \text{Closure } (S' \to .\, S) = \left\{\begin{matrix} S' \to .\, S \\ S \to .\, iCSeS \\ S \to .\, iCS \\ S \to .\, a \end{matrix}\right\}$$

$$\text{goto } (I_0, S) = S' \to S\,. - (I_1)$$

$$\text{goto } (I_0, i) = \left\{\begin{matrix} S \to i .\, CSeS \\ S \to i .\, CS \end{matrix}\right\} (I_2)$$

$$\text{goto } (I_0, a) = S \to a\,. - (I_3)$$

$$\text{goto } (I_2, C) = \begin{Bmatrix} S \to iC.SeS \\ S \to iC.S \\ S \to .iCSeS \\ S \to .iCS \\ S \to .a \end{Bmatrix} - (I_4)$$

$$\text{goto } (I_4, S) = \begin{Bmatrix} S \to iCS.eS \\ S \to iCS. \end{Bmatrix} - (I_5)$$

$$\text{goto } (I_4, i) = \begin{Bmatrix} S \to i.CSeS \\ S \to i.CS \end{Bmatrix} \text{ same as } (I_2)$$

$$\text{goto } (I_4, a) = \{S \to a.\} - \text{same as } (I_3)$$

$$\text{goto } (I_5, e) = \begin{Bmatrix} S \to iCSe.S \\ S \to .iCS \\ S \to .iCSeS \\ S \to .a \end{Bmatrix} - (I_6)$$

$$\text{goto } (I_6, S) = \{S \to iCSeS.\} - (I_7)$$

$$\text{goto } (I_6, i) = \begin{Bmatrix} S \to i.CS \\ S \to i.CSeS \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_6, a) = \{S \to a.\}$$

Step 3 : Parsing table for grammar is as follows :

T. / N.T.	ACTION				GOTO
	i	e	a	$	S
I_0	S_2		S_3	S_2	1
I_1				S	Accept
I_2	S_2			S_3	4
I_3					
I_4		R_3		R_3	
I_5	S_2			r_2	
I_6					6
I_7		R_1		R_1	

As there are no multiple entries so grammar is LR.

Q.67. Construct LR (1) parsing table for following grammar

S → A

A → BA / ∈

B → aB / b .

Also show the sequence of action of LR(1) parser on input aaab.

CT : S-11(13M)

Ans.

Step 1 : Augmented grammar :

S' → .S

S → .A ----- (I)

A → .BA ----- (II)

A → .∈ ---- (III)

B → .aB ---- (IV)

B → .b ----- (V)

Step 2 : Canonical collection of LR (1) item :

$$I_0 = \text{Closure } (S' \to .S) = \begin{Bmatrix} S' \to .S, \$ \\ S \to .A, \$ \\ A \to .BA, \$ \\ A \to ., \$ \\ B \to .aB, a|b|\$ \\ B \to .b, a|b|\$ \end{Bmatrix}$$

$$\text{goto } (I_0, S) = \{S' \to S., \$\} - (I_1)$$

$$\text{goto } (I_0, A) = \{S \to A., \$\} - (I_2)$$

$$\text{goto } (I_0, a) = \begin{Bmatrix} B \to a.B, a|b|\$ \\ B \to .aB, a|b|\$ \\ B \to .b, a|b|\$ \end{Bmatrix} - (I_3)$$

$$\text{goto } (I_0, B) = \begin{Bmatrix} A \to B.A, \$ \\ A \to .BA, \$ \\ A \to ., \$ \\ B \to .aB, a|b|\$ \\ B \to .b|a|b|\$ \end{Bmatrix} - (I_4)$$

$$\text{goto } (I_0, b) = \{B \to b; a|b|\$\} - (I_5)$$

$$\text{goto } (I_3, B) = \{B \to aB., \$|a|b\} - (I_6)$$

$$\text{goto } (I_3, a) = (I_3)$$

$$\text{goto } (I_3, b) = (I_5)$$

$$\text{goto } (I_4, A) = \{A \to BA., \$\} - (I_7)$$

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

$$\text{goto } (I_4, a) = (I_3)$$

$$\text{goto } (I_4, b) = (I_5)$$

Step 3 : LR (1) parsing table is as follows :

T. / N.T.	ACTION			GOTO		
	a	b	$	S	A	B
I_0	S_3	S_5	R_3	1	2	4
I_1			Accept			
I_2			R_1			
I_3	S_3	S_5				6
I_4	S_3	S_5	R_3		7	4
I_5	R_5	R_5	R_5			
I_6	R_4	R_4	R_4			
I_7			R_2			

Production are numbered as

$S \to A$ ...(I)

$A \to BA$ ...(II)

$A \to \epsilon$ ...(III)

$B \to aB$ ...(IV)

$B \to b$ ...(V)

Step 4 : Sequence of action of LR (1) parser for input w = a a a b :

Stack contents	Input buffer	Action taken by parser
$ I_0	aaab$	Shift a on TOS
$ a	a a b $	Shift a
$ aa	a b $	Shift a
$aaa	b $	Shift b
$ aaab	$	Reduction using $B \to b$
$ aaaB	$	Reduction using $B \to aB$
$ aaB	$	Reduction using $B \to aB$
$ aB	$	Reduction using $B \to aB$
$ B	$	Reduction using $A \to \epsilon$
$ A	$	Reduction using $S \to A$
$ S	$	Parser in configuration ($ S, $) so it halts and announces successful completion of parsing

Here parser enter into subsequence states after carrying out appropriate action like shift, reduce etc. So states are not shown in stack contents.

Q.68. Construct LR (1) parsing table for following grammar

$S \to wAz / xBz / wBy / xAy$

$A \to r$

$B \to r$

CS : W-12 (8M)

Ans.

Step 1 : Augmented grammar is :

$S' \to S$

$S \to .wAz$ ... (I)

$S \to .xBz$ ... (II)

$S \to .wBy$ ... (III)

$S \to .xAy$ ... (IV)

$A \to .r$ ... (V)

$B \to .r$ ... (VI)

Step 2 : LR (1) set of items are as follows :

$$I_0 = \text{Closure}\{(S' \to .S\} = \begin{Bmatrix} S' \to .S, \$ \\ S \to .wAz, \$ \\ S \to .xBz, \$ \\ S \to .wBy, \$ \\ S \to .zAy, \$ \end{Bmatrix} - (I_0)$$

$$\text{goto } (I_0, S) = \{S' \to S., \$\} - (I_1)$$

$$\text{goto } (I_0, w) = \begin{Bmatrix} S \to w.Az, \$ \\ S \to w.By, \$ \\ A \to .r, y \\ B \to .r, z \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_0, x) = \begin{Bmatrix} S \to x.Bz, \$ \\ B \to .r, z \end{Bmatrix} - (I_3)$$

$$\text{goto } (I_0, z) = \begin{Bmatrix} S \to z.Ay, \$ \\ A \to .r, y \end{Bmatrix} - (I_4)$$

$$\text{goto } (I_2, A) = \{S \to wA.z, \$\} - (I_5)$$

$$\text{goto } (I_2, B) = \{S \to wB.y, \$\} - (I_6)$$

$$\text{goto } (I_2, r) = \begin{Bmatrix} A \to r., z \\ B \to r., z \end{Bmatrix} - (I_7)$$

$$\text{goto } (I_3, B) = \{S \to xB.z, \$\} - (I_8)$$

$$\text{goto } (I_3, r) = \{B \to r., z\} - (I_9)$$

$$\text{goto } (I_4, A) = \{S \to zA.y, \$\} - (I_{10})$$

$$\text{goto } (I_4, r) = \{A \to r., y\} - (I_{11})$$

goto $(I_5, z) = \{S \to wAz., \$\} - (I_{12})$

goto $(I_6, y) = \{S \to wBy., \$\} - (I_{13})$

goto $(I_8, z) = \{S \to xBz, \$\} - (I_{14})$

goto $(I_{10}, y) = \{S \to zAy., \$\} - (I_{15})$.

Step 3 : Parsing table is computed as follows :

T. / N.T.	ACTION						GOTO		
	w	z	x	y	r	$	S	A	B
I_0	S_2	S_4	S_3				1		
I_1						accept			
I_2					S_7			5	6
I_3					S_9				8
I_4					S_{11}			10	
I_5		S_{12}							
I_6				S_{13}					
I_7		R_5		R_6					
I_8		S_{14}							
I_9		R_6							
I_{10}				S_{10}					
I_{11}				R_5					
I_{12}						R_1			
I_{13}						R_3			
I_{14}						R_2			
I_{15}						R_4			

As table does not contain multiple entries so grammar is LL (1).

Q.69. Construct LR(1) parser for following grammar :

$E \to E + T/T$

$A \to TF / F$

$F \to F^* / a / b$ **CS : W-14 (7M)**

Ans.

Step 1 : Augmented grammar :

$E' \to E$

$E \to .E + T$ ... (I)

$E \to .T$ ... (II)

$E \to .TF$ ... (III)

$E \to .F$ ... (IV)

$F \to .F^*$ ... (V)

$F \to .a$ ... (VI)

$F \to .b$ ... (VII)

Step 2 : LR (1) items are as follows :

$$I_0 = \text{Closure}\{E' \to .E\} = \left\{\begin{array}{c} E' \to .E, \$ \\ E \to .E + T, \$/+ \\ E \to .T, \$/+ \\ T \to .TF, \$/+/a/b \\ T \to .F, \$/+/a/b \\ F \to .F^*, \$/+/a/b/^* \\ F \to .a, \$/+/a/b/^* \\ F \to .b, \$/+/a/b/^* \end{array}\right\} - (I_0)$$

$$\text{goto } (I_0, E) = \left\{\begin{array}{c} E' \to E., \$ \\ E \to E. + T, \$/+ \end{array}\right\} - (I_1)$$

$$\text{goto } (I_0, T) = \left\{\begin{array}{c} E \to T., \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F^*, \$/a/+/b/^* \\ F \to .a, \$/a/+/b/^* \\ F \to .b, \$/a/+/b/^* \end{array}\right\} - (I_2)$$

$$\text{goto } (I_0, F) = \left\{\begin{array}{c} T \to F., \$/+/a/b \\ F \to F.^*, \$/+/a/b/^* \end{array}\right\} - (I_3)$$

$$\text{goto } (I_0, a) = \{F \to a., \$/+/a/b/^*\} - (I_4)$$

$$\text{goto } (I_0, b) = \{F \to b., \$/+/a/b/^*\} - (I_5)$$

$$\text{goto } (I_1, +) = \left\{\begin{array}{c} E \to E + .T, \$/+ \\ T \to .F, \$/+/a/b \\ T \to .F, \$/+/a/b \\ F \to .F^* \$/+/a/b/^* \\ F \to .a \$/+/a/b/^* \\ F \to .b \$/+/a/b/^* \end{array}\right\} - (I_6)$$

$$\text{goto } (I_2, F) = \left\{\begin{array}{c} T \to TF., \$/+/a/b \\ F \to F.^* \$/+/a/b/^* \end{array}\right\} - (I_7)$$

$$\text{goto } (I_2, a) = \{F \to a., \$/+/a/b/^*\} - (I_4)$$

$$\text{goto } (I_2, b) = \{F \to b., \$/+/a/b/^*\} - (I_5)$$

$$\text{goto } (I_3, ^*) = \{F \to F^*., \$/+/a/b/^*\} - (I_8)$$

$$\text{goto } (I_6, T) = \left\{\begin{matrix} E \to E + T\,.\,\$/+ \\ T \to T\,.\,F, \$/+/a/b \\ F \to .\,F^*, \$/+/a/b/^* \\ F \to .\,a, \$/+/a/b/^* \\ F \to .\,b, \$/+/a/b/^* \end{matrix}\right\} - (I_9)$$

$$\text{goto } (I_6, F) = \left\{\begin{matrix} T \to F\,.\,, \$/+/a/b \\ F \to F\,.^*, \$/+/a/b/^* \end{matrix}\right\} \quad \text{.....same as } (I_3)$$

goto $(I_6, a) = \{F \to a\,.\,, \$/+/a/b/^*\}$ .....same as (I_4)

goto $(I_6, b) = \{F \to b\,.\,, \$/+/a/b/^*\}$ .....same as (I_5)

$$\text{goto } (I_9, F) = \left\{\begin{matrix} T \to TF\,.\,, \$/+/a/b \\ F \to F\,.^*, \$/a/+/b \end{matrix}\right\} \quad \text{.....same as } (I_7)$$

goto $(I_9, a) = \{F \to a\,.\,, \$/+/a/b/^*\}$ .....same as (I_4)

goto $(I_9, b) = \{F \to b\,.\,, \$/+/a/b/^*\} - (I_5)$.

Step 3 : Parsing table is as follows :

T. / N.T.	ACTION					GOTO		
	+	*	a	b	$	E	T	F
I_0						1	2	3
I_1	S_6				accept			
I_2	R_2		S_4	S_5	R_2			7
I_3	R_4	S_8	R_4	R_4	R_4			
I_4	R_6	R_6	R_6	R_6	R_6			
I_5	R_7	R_7	R_7	R_7	R_7			
I_6			S_4	S_5			9	3
I_7	R_3	S_8	R_3	R_3	R_3			
I_8	R_5	R_5	R_5	R_5	R_5			
I_9	R_1		S_4	S_5	R_1			7

Q.70. Construct LR (1) parsing table for following grammar :

CT : S-12 (10M)

$S \to cA/ccB$

$A \to cA / a$

$B \to ccB / b$

Also show the moves of stack implementation for input string "ccccb".

Ans.

Step 1 : Augmented grammar :

$S' \to S$

$S \to .cA$ ... (I)

$S \to .ccB$ ... (II)

$A \to .cA$ ... (III)

$A \to .a$ ... (IV)

$B \to .ccB$ ... (V)

$B \to .b$ ... (VI)

Step 2 : The canonical collection of set of LR (1) item are as follows :

$$I_0 = \{S' \to .S\} = \left\{\begin{matrix} S' \to .\,S, \$ \\ S \to .\,cA, \$ \\ S \to .\,ccB, \$ \end{matrix}\right\}$$

goto $(I_0, S) = \{S' \to S\,.\,, \$\} - (I_1)$

$$\text{goto } (I_0, c) = \left\{\begin{matrix} S \to c\,.\,A, \$ \\ S \to c.\,cB, \$ \\ A \to .\,cA, \$ \\ A \to .\,a, \$ \end{matrix}\right\} - (I_2)$$

goto $(I_2, A) = \{S \to cA\,.\,, \$\} - (I_3)$

goto $(I_2, a) = \{A \to a\,.\,, \$\} - (I_4)$

$$\text{goto } (I_2, c) = \left\{\begin{matrix} S \to cc\,.\,B, \$ \\ A \to c\,.\,A, \$ \\ A \to .\,cA, \$ \\ A \to .\,a, \$ \\ B \to .\,ccB, \$ \\ B \to .\,b, \$ \end{matrix}\right\} - (I_5)$$

goto $(I_5, B) = \{S \to ccB\,.\,, \$\} - (I_6)$

goto $(I_5, A) = \{A \to cA\,.\,, \$\} - (I_7)$

goto $(I_5, a) = \{A \to a\,.\,, \$\} - (I_4)$

goto $(I_5, b) = \{B \to b\,.\,, \$\} - (I_8)$

$$\text{goto } (I_5, c) = \left\{\begin{matrix} A \to c\,.\,A, \$ \\ B \to c.\,cB, \$ \\ A \to .\,cA, \$ \\ A \to .\,a, \$ \end{matrix}\right\} - (I_9)$$

goto $(I_9, c) = \{A \to cA\,.\,, \$\} - (I_7)$

$$\text{goto } (I_9, c) = \left\{\begin{matrix} A \to c\,.\,A, \$ \\ B \to cc\,.\,B, \$ \\ A \to .\,cA, \$ \\ A \to .\,a, \$ \\ B \to .\,ccB, \$ \\ B \to .\,b, \$ \end{matrix}\right\} - (I_{10})$$

goto $(I_9, a) = \{A \to a\,.\,, \$\} - (I_4)$

goto $(I_{10}, A) = \{A \to cA.,\$\} - (I_7)$

goto $(I_{10}, B) = \{B \to ccB.,\$\} - (I_{11})$

goto $(I_{10}, c) = \begin{Bmatrix} A \to c.A,\$ \\ B \to c.cB,\$ \\ A \to .cA,\$ \\ A \to .a,\$ \end{Bmatrix} - (I_9)$

goto $(I_9, a) = \{A \to a.,\$\} - (I_4)$

goto $(I_9, b) = \{B \to b.,\$\} - (I_8)$

Step 3 : The parsing table is as follows :

T. / N.T.	ACTION				GOTO		
	a	b	c	$	S	A	B
I_0			S_2		1		
I_1				accept			
I_2	S_4		S_5			3	
I_3				R_1			
I_4				R_4			
I_5	S_4	S_8	S_9			7	6
I_6				R_2			
I_7				R_3			
I_8				R_6			
I_9	S_4		S_{10}			7	
I_{10}	S_4	S_8	S_9			7	11
I_{11}				R_5			

Since there are no multiple entries in parsing table so grammar is LR (1).

Step 4 : Moves made by parser :

Stack contents	Buffer contents	Action taken by parser
$\$ I_0$	ccccb$	Shift c, enter into I_2
$\$ I_0 c I_2$	ccc b $	Shift c, enter into I_5
$I_0 c I_2 c I_5$	ccb $	Shift c, enter into I_9
$\$ I_0 c I_2 c I_5 c I_9$	c b $	Shift c, enter into I_{10}
$\$ I_0 c I_2 c I_5 c I_9 I_{10} b I_8$	$	Reduction using $B \to b_1$ enter into I_{11}
$\$ I_0 c I_2 c I_5 c I_9 I_{10} B I_{11}$	$	Reduction by using $B \to ccB$ and enter into I_6
$\$ I_0 c I_2 c I_5 B I_6$	$	Reduce by using $S \to ccB$
$\$ I_0 S I_4$	$	Accept and announce successful completion of parsing

Q.71. Obtain LR parsing table for the following grammar which do not contain any multiple entry. The table should have minimum possible number of states.

$E \to E + T/T$

$T \to TF / F$

$F \to F^* /(E)/a/b$ **CS : S-09 (10M)**

Ans.

Step 1 : Augmented grammar :

$E' \to .E$ ... (I)

$E \to .E + T$ ... (II)

$E \to .T$ ... (III)

$T \to .TF$ ... (IV)

$T \to .F$ ... (V)

$F \to .F^*$ ... (VI)

$F \to .a$ ... (VII)

$F \to .b$ ... (VIII)

$F \to .(E)$ ... (IX)

Step 2 : LR (1) items are computed as follows :

$$I_0 = \{E' \to .E\} = \left\{ \begin{matrix} E' \to .E,\$ \\ E \to .E + T,\$/+ \\ E \to .T,\$/+ \\ T \to .TF,\$/+/a/b \\ T \to F,\$/+/a/b \\ F \to .F^*,\$/+/a/b/^* \\ F \to .a\$/+/a/b/^* \\ F \to .b\$/+/a/b|/^* \\ F \to (E).\$/+/a/b/^* \end{matrix} \right\} - (I_0)$$

goto $(I_0, E) = \begin{Bmatrix} E' \to E., \$ \\ E \to E.+T, \$/+ \end{Bmatrix} -(I_1)$

goto $(I_0, T) = \begin{Bmatrix} E \to T., \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F*, \$/+/a/b/* \\ F \to .a, \$/+/a/b/* \\ F \to .b, \$/+/a/b/* \\ F \to (E), \$/+/a/b/* \end{Bmatrix} -(I_2)$

goto $(I_0, F) = \begin{Bmatrix} T \to F., \$/+/a/b \\ F \to F.*\$/+/a/b/* \end{Bmatrix} -(I_3)$

goto $(I_0, a) = \{F \to a., \$/+/a/b/*\} -(I_4)$

goto $(I_0, b) = \{F \to b., \$/+/a/b/*\} -(I_5)$

$\{F \to (.E), \$|+|a|b|*$

goto $(I_0, () = \begin{Bmatrix} E \to .E+T, \$/T \\ E \to .T, \$/+ \\ T \to .TF, \$/+/a/b \\ T \to .F, \$/+/a/b \\ F \to .F*\$/+/a/b/* \\ F \to .a\$/+/a/b/* \\ F \to .b\$/+/a/b/* \\ F \to .(E), \$/+/a/b/* \end{Bmatrix} -(I_6)$

goto $(I_1, +) = \begin{Bmatrix} E \to E+.T, \$/+ \\ T \to .TF, \$/+/a/b \\ T \to .F, \$/a/+/b \\ F \to .F*\$/+/a/b/* \\ F \to .a\$/+/a/b/* \\ F \to .b\$/+/a/b/* \\ F \to (E), \$/+/a/b/* \end{Bmatrix} -(I_7)$

goto $(I_2, F) = \begin{Bmatrix} T \to TF., \$/+/a/b \\ F \to F.*, \$/+/a/b/* \end{Bmatrix} -(I_8)$

goto $(I_2, b) = (I_5)$

goto $(I_2, a) = (I_4)$

goto $(I_3, *) = \{F \to F*., \$/+/a/b/*\} -(I_9)$

goto $(I_7, *) = (I_9)$

goto $(I_6, E) = \begin{Bmatrix} F \to (E.), \$/+/a/b/* \\ E \to E.+T, \$, /T \end{Bmatrix} -(I_{10})$

goto $(I_6, T) = \begin{Bmatrix} E \to T.,, \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F*, \$/+/a/b/ \\ F \to .a, \$/+/a/b/* \\ F \to .b, \$/+/a/b/* \\ F \to .(E), \$/+/a/b/* \end{Bmatrix} -(I_{11})$

goto $(I_6, F) = \begin{Bmatrix} T \to F., \$/+/a/b \\ F \to F.*, \$/+/a/b/* \end{Bmatrix}$ ..... same as (I_3)

goto $(I_6, a) =$ same as (I_4)

goto $(I_6, b) =$ same as (I_5)

goto $(I_6, () =$ same as (I_6)

goto $(I_7, T) = \begin{Bmatrix} E \to E+T., \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F*, \$/+/a/b/* \\ F \to .a, \$/+/a/b/* \\ F \to .b, \$/+/a/b/* \\ F \to .(E).\$/a/b/* \end{Bmatrix} -(I_{12})$

goto $(I_7, F) =$ same as (I_8)

goto $(I_7, a) =$ same as (I_4)

goto $(I_7, b) =$ same as (I_5)

goto $(I_7, () =$ ame as (I_6)

goto $(I_8, *) = \{F \to F*., \$/+/a/b/*\}$ ..... same as (I_9)

goto $(I_{10},)) = \{F \to (E)., \$/+/a/b/*\} -(I_3)$.

goto $(I_{10}, +) = \begin{Bmatrix} E \to E+.T, \$/T \\ T \to .TF, \$/+/a/b \\ T \to .T, \$/+ \\ T \to .F, \$/+/a/b \\ F \to .F*\$/+/a/b/* \\ F \to .a\$/+/a/b/* \\ F \to .b\$/+/a/b/* \\ F \to .(E), \$/+/a/b/* \end{Bmatrix}$ .....Same as (I_7)

goto $(I_{11}, f) =$ same as (I_3)

goto $(I_{11}, a) = (I_4)$

goto $(I_{11}, b) = (I_5)$

goto $(I_{11}, c) =$ Same as (I_6)

goto $(I_{12}, F) = \begin{Bmatrix} T \to TF., \$/+/a/b \\ T \to F.*, \$/+/a/b/* \end{Bmatrix}$ .....Same as (I_8)

goto $(I_{12}, a) =$ same as (I_4)

goto $(I_{12}, b) =$ same as (I_5)

goto $(I_{12}, () =$ same as (I_6)

Step 3 : Parsing table is as follows :

T. / N.T.	ACTION					GOTO		
	+	*	a	b	$	E	T	F
I_0			S_4	S_3		1	2	3
I_1	S_6				accept			
I_2	R_2		S_4	S_3	R_2			7
I_3	R_4	S_8	R_4	R_4	R_4			
I_4	R_6	R_6	R_6	R_6	R_7			
I_5			S_4	S_5			3	9
I_6	R_3				R_3			
I_7	R_5				R_5			
I_8	R_1		S_4	S_5	R_1			7
I_9								
I_{10}								
I_{11}								
I_{12}								

Q.72. Design LR (1) parsing table for the following grammar :

CT : W-09 (9M), W-12(10M)

$S \rightarrow L / a$

$L \rightarrow w\ G\ d\ S / d\ S\ we$

$G \rightarrow b$

Ans.

Step 1 : Augmented grammar :

$S' \rightarrow .S$

$S \rightarrow .L / a$

$L \rightarrow .w\ G\ d\ S / d\ S\ we$

$G \rightarrow .b$

Step 2 : LR (1) items are computed as follows :

$$I_0 = \text{Closure}\{(S' \rightarrow S)\} = \left\{\begin{matrix} S' \rightarrow .S, \$ \\ S \rightarrow .L, \$ \\ S \rightarrow .a, \$ \\ L \rightarrow .wGdS, \$ \\ L \rightarrow .dSwe, \$ \end{matrix}\right\} - (I_0)$$

$\text{goto}\ (I_0, S) = \{S' \rightarrow S., \$\} - (I_1)$

$\text{goto}\ (I_0, L) = \{S \rightarrow L., \$\} - (I_2)$

$\text{goto}\ (I_0, a) = \{S \rightarrow a., \$\} - (I_3)$

$$\text{goto}\ (I_0, w) = \left\{\begin{matrix} L \rightarrow w.GdS, \$ \\ G \rightarrow .b, d \end{matrix}\right\} - (I_4)$$

$$\text{goto}\ (I_0, d) = \left\{\begin{matrix} L \rightarrow d.Swe, \$ \\ S \rightarrow .L, w \\ S \rightarrow .a, w \\ L \rightarrow .wGdS, w \\ L \rightarrow .dSwe, w \end{matrix}\right\} - (I_5)$$

$\text{goto}\ (I_4, G) = \{L \rightarrow wG.dS, \$\} - (I_6)$

$\text{goto}\ (I_4, b) = \{G \rightarrow b., d\} - (I_7)$

$\text{goto}\ (I_5, S) = \{L \rightarrow dS.we, \$\} - (I_8)$

$\text{goto}\ (I_5, L) = \{S \rightarrow L., w\} - (I_9)$

$\text{goto}\ (I_5, a) = \{S \rightarrow a., w\} - (I_{10})$

$$\text{goto}\ (I_5, w) = \left\{\begin{matrix} L \rightarrow w.GdS, w \\ G \rightarrow b., d \end{matrix}\right\} - (I_{11})$$

$$\text{goto}\ (I_5, d) = \left\{\begin{matrix} L \rightarrow d.Swe, w \\ S \rightarrow .L, w \\ S \rightarrow .a, w \\ L \rightarrow .wGdS, w \\ L \rightarrow .dSwe, w \end{matrix}\right\} - (I_{12})$$

$$\text{goto}\ (I_6, d) = \left\{\begin{matrix} L \rightarrow wGd.S, \$ \\ S \rightarrow .L, \$ \\ S \rightarrow .a, \$ \\ L \rightarrow .wGdS, \$ \\ L \rightarrow .dSwe, \$ \end{matrix}\right\} - (I_{13})$$

$\text{goto}\ (I_8, w) = \{L \rightarrow dSw.e, \$\} - (I_{14})$

$\text{goto}\ (I_{11}, G) = \{L \rightarrow wG.dS, w\} - (I_{15})$

$\text{goto}\ (I_{11}, b) = \{G \rightarrow b., d\}$ – same as (I_7)

$\text{goto}\ (I_{12}, S) = \{L \rightarrow dS., we, w\} \rightarrow (I_{16})$

$\text{goto}\ (I_{12}, L) = \{S \rightarrow L., w\}$ – same as (I_9)

$\text{goto}\ (I_{12}, a) = \{S \rightarrow a., w\}$ – same as (I_{10})

$$\text{goto}\ (I_{12}, w) = \left\{\begin{matrix} L \rightarrow w.GdS, w \\ G \rightarrow .b, d \end{matrix}\right\} - \text{same as}\ (I_{11})$$

$$\text{goto}\ (I_{12}, d) = \left\{\begin{matrix} L \rightarrow d.Swe, w \\ S \rightarrow .L, w \\ S \rightarrow .a, w \\ L \rightarrow .wGdS, w \\ L \rightarrow .dSwe, w \end{matrix}\right\} - \text{same as}\ (I_{12})$$

$\text{goto}\ (I_{13}, S) = \{L \rightarrow wGdS., \$\} - (I_{17})$

goto $(I_{13}, L) = \{S \to L., \$\}$ – same as (I_2)

goto $(I_{13}, a) = \{S \to a., \$\}$ – same as (I_3)

goto $(I_{13}, w) = \begin{Bmatrix} L \to w.GdS, \$ \\ G \to .b, d \end{Bmatrix}$ – same as (I_4)

goto $(I_{13}, d) = \begin{Bmatrix} L \to d.Swe, \$ \\ S \to .L, w \\ S \to .a, w \\ L \to .wdS, w \\ L \to .dSwe, w \end{Bmatrix}$ – same as (I_5)

goto $(I_{14}, e) = \{L \to dSwe., \$\}$ – (I_{18})

goto $(I_{15}, d) = \begin{Bmatrix} L \to wGd.S, w \\ S \to .L, w \\ S \to .a, w \\ L \to .wdS, w \\ L \to .dSwe, w \end{Bmatrix}$ – same as (I_{19})

goto $(I_{16}, w) = \{L \to dSw.c, w\}$ – (I_{20})

goto $(I_{19}, S) = \{L \to wGdS., w\}$ – (I_{21})

goto $(I_{19}, L) = \{S \to L., w\}$ – same as (I_9)

goto $(I_{19}, a) = \{S \to a., w\}$ – same as (I_{10})

goto $(I_{19}, w) = \begin{Bmatrix} L \to w.GdS, w \\ G \to .b, d \end{Bmatrix}$ – same as (I_{11})

goto $(I_{19}, d) = \begin{Bmatrix} L \to d.Swe, w \\ S \to .L, w \\ S \to .a, w \\ L \to .wGdS, w \\ L \to .dSwe, w \end{Bmatrix}$ – same as (I_{12})

goto $(I_{20}, C) = \{L \to dSwe., w\}$ – (I_{22})

Step 3 : Parsing table is computed as follows :

T. / N.T.	ACTION						GOTO		
	a	b	e	d	w	$	R	L	G
I_0				S_3	S_4		1	2	
I_1						accept			
I_2						R_1			
I_3						R_2			
I_4		S_7							6
I_5	S_{10}			S_{12}	S_{11}		8	9	
I_6				S_{13}					
I_7					R_5				
I_8						S_{14}			
I_9						R_1			
I_{10}						R_2			
I_{11}		S_7							15
I_{12}	S_{10}				S_{12}	S_{11}		16	9
I_{13}	S_3				S_5	S_4		17	2
I_{14}				S_{18}					
I_{15}					S_{19}				
I_{16}						S_{20}			
I_{17}							R_3		
I_{18}							R_4		
I_{19}	S_{10}				S_{12}	S_{11}		21	9
I_{20}				S_{22}					
I_{21}						R_3			
I_{22}						R_4			

As table doesn't contain multiple entries, so grammar is LR (1).

Q.73. Design LR (1) parsing table for the following grammar :

CT : W-10 (13M)

$S \to Aa / bAc / Bc / bBa$

$A \to d$

$B \to d$

Ans.

Step 1 : Augmented grammar :

$S' \to .S$ ...(I)

$S \to .Aa$ ...(II)

$S \to .bAc$ ...(III)

$S \to .Bc$ ...(IV)

$S \to .bBa$ ...(V)

$A \to .d$ ...(VI)

$B \to .d$ ...(VII)

Step 2 : Collect LR (1) items as follows :

$$I_0 = \text{Closure}\{(S' \to .S)\} = \left\{\begin{matrix} S' \to .S, \$ \\ S \to .Aa, \$ \\ S \to .bAc, \$ \\ S \to .Bc, \$ \\ S \to .bBa, \$ \\ A \to .d, a \\ B \to .d, c \end{matrix}\right\} - (I_0)$$

$$\text{goto } (I_0, S) = \{S' \to S., \$\} \quad -(I_1)$$

$$\text{goto } (I_0, A) = \{S \to A.a, \$\} - (I_2)$$

$$\text{goto } (I_0, b) = \left\{\begin{matrix} S \to b.Ac, \$ \\ A \to .d, c \\ S \to b.Ba, \$ \\ B \to .d, a \end{matrix}\right\} \quad -(I_3)$$

$$\text{goto } (I_0, B) = \{S \to B.c, \$\} \quad -(I_4)$$

$$\text{goto } (I_0, d) = \left\{\begin{matrix} A \to d., a \\ B \to d., c \end{matrix}\right\} \quad -(I_5)$$

$$\text{goto } (I_2, a) = \{S \to Aa., \$\} \quad -(I_6)$$

$$\text{goto } (I_3, A) = \{S \to bA.c, \$\} \quad -(I_7)$$

$$\text{goto } (I_3, d) = \left\{\begin{matrix} A \to d., c \\ B \to d., a \end{matrix}\right\} \quad -(I_8)$$

$$\text{goto } (I_3, B) = \{S \to bB.a, \$\} \quad -(I_9)$$

$$\text{goto } (I_4, c) = \{S \to Bc., \$\} \quad -(I_{10})$$

$$\text{goto } (I_7, c) = \{S \to bAc., \$\} \quad -(I_{11})$$

$$\text{goto } (I_9, a) = \{S \to bBa., \$\} \quad -(I_{12})$$

Step 3 : Construct LR (1) parsing table as follows :

T. / N.T.	ACTION					GOTO		
	a	b	c	d	$	S	A	B
I_0		S_3		S_5		1	2	4
I_1					accept			
I_2	S_6							
I_3				S_8			7	9
I_4			S_{10}					
I_5								
I_6	R_2							
I_7			S_{11}					
I_8								
I_9	S_{12}							
I_{10}								
I_{11}								
I_{12}	r_1							

As table doesn't contain multiple entries, so given grammar is LR (1).

Q.74. Construct LR (1) parsing table for the following grammar :

CT : S-09(10M)

$S \to B$

$B \to \text{begin } DA \text{ end}$

$D \to Dd / \epsilon$

$A \to A ; E / E$

$E \to B / S$

Ans.

Step 1 : Augmented grammar is as follows :

$S' \to .S$

$S \to .B$

$B \to .\text{begin } DA \text{ end}$

$D \to .Dd \mid \epsilon$

$A \to .A ; E \mid E$

$E \to .B \mid S$

Step 2 : The canonical collection of set of LR (1) items are as follows :

$$I_0 = \text{Closure}\{(S' \to .S, \$)\} = \left\{\begin{matrix} S' \to .S, \$ \\ S \to .B, \$ \\ B \to .\text{begin DA end}, \$ \end{matrix}\right\}$$

$$\text{goto } (I_0, S) = \text{Closure}(\{S' \to S., \$\}) = \{S' \to S., \$\} \quad -(I_1)$$

$$\text{goto } (I_0, B) = \text{Closure}(\{S \to B., \$\}) = \{S \to B., \$\} \quad -(I_2)$$

$$\text{goto } (I_0, \text{begin}) = \left\{\begin{matrix} B \to \text{begin}.DA \text{ end}, \$ \\ D \to .Dd, \text{begin} / d \\ D \to .\text{begin} / d \end{matrix}\right\} \quad -(I_3)$$

$$\text{goto } (I_3, D) = \left\{\begin{matrix} B \to \text{begin } D.A \text{ end}, \$ \\ D \to D.d, \text{begin} / d \\ D \to ., \text{begin} / d \\ A \to .A ; E, \text{end} / ; \\ A \to .E, \text{end} / ; \\ E \to .B, \text{end} / ; \\ B \to .\text{begin DA end}, \text{end} / ; \\ S \to .B, \text{end} / ; \end{matrix}\right\} \quad -(I_4)$$

$$\text{goto } (I_4, A) = \left\{\begin{pmatrix} B \to \text{begin DA . end}, \$ \\ A \to A . ; E \end{pmatrix}\right\} - (I_5)$$

$$\text{goto } (I_4, d) = \{(D \to D. \text{ begin / d})\} - (I_6)$$

$$\text{goto } (I_4, E) = \{A \to E., \text{end / ;}\} - (I_7)$$

$$\text{goto } (I_8, B) = \left\{\begin{matrix} E \to B., \text{end / ;} \\ S \to B., \text{end / ;} \end{matrix}\right\} - (I_8)$$

$$\text{goto } (I_4, \text{begin}) = \left\{\begin{matrix} . B \to \text{begin . DA end, end / ;} \\ D \to . Dd, \text{begin / d} \\ D \to ., \text{begin / d} \end{matrix}\right\} - (I_9)$$

$$\text{goto } (I_4, S) = \{E \to S., \text{end / ;}\} - (I_{10})$$

$$\text{goto } (I_5, \text{end}) = \{B \to \text{begin DA end .}, \$\} - (I_{11})$$

$$\text{goto } (I_5, ;) = \left\{\begin{matrix} A \to A ; . E, \text{end / ;} \\ E \to . B, \text{end / ;} \\ E \to . S, \text{end / ;} \\ B \to . \text{begin DA end ; end / ;} \\ S \to . B, \text{end / ;} \end{matrix}\right\} - (I_{12})$$

$$\text{goto } (I_9, D) = \left\{\begin{matrix} B \to \text{begin D . A end, end / ;} \\ D \to D . d, \text{begin / d} \\ A \to . A ; E, \text{end / ;} \\ A \to . E, \text{end / ;} \\ E \to . B, \text{end / ;} \\ E \to . S, \text{end / ;} \\ B \to . \text{begin DA end, end / ;} \\ S \to . B, \text{end / ;} \end{matrix}\right\} - (I_{13})$$

$$\text{goto } (I_{12}, E) = \{A \to A ; E., \text{end / ;}\} - (I_{14})$$

$$\text{goto } (I_{12}, B) = \left\{\begin{matrix} E \to B., \text{end / ;} \\ S \to B., \text{end / ;} \end{matrix}\right\} - \text{same as } (I_8)$$

$$\text{goto } (I_{12}, S) = \{E \to S., \text{end / ;}\} - \text{same as } (I_{10})$$

goto

$$(I_{12}, \text{begin}) = \left\{\begin{matrix} B \to \text{begin . DA end, end / ;} \\ D \to . Dd ; \text{begin / d} \\ D \to ., \text{begin / d} \end{matrix}\right\} - \text{same as } (I_9)$$

$$\text{goto } (I_{13}, A) = \left\{\begin{matrix} B \to \text{begin DA . end, end / ;} \\ A \to . A ; E, \text{end / ;} \end{matrix}\right\} - (I_{15})$$

$$\text{goto } (I_{13}, d) = \{D \to Dd., \text{begin / }\} - \text{same as } (I_6)$$

$$\text{goto } (I_{13}, E) = \{A \to E., \text{end / ;}\} - \text{same as } (I_7)$$

$$\text{goto } (I_{13}, B) = \{S \to B., \text{end / ;}\} - \text{same as } (I_8)$$

$$\text{goto } (I_{13}, S) = \{E \to S., \text{end / ;}\} - \text{same as } (I_{10})$$

goto

$$(I_{13}, \text{begin}) = \left\{\begin{matrix} B \to \text{begin . DA end, end / ;} \\ D \to . Dd ; \text{begin / d} \\ D \to ., \text{begin / d} \end{matrix}\right\} - \text{same as } (I_9)$$

$$\text{goto } (I_{15}, \text{end}) = \{B \to \text{begin DA end ., end / ;}\} - (I_{16})$$

$$\text{goto } (I_{15}, ;) = \left\{\begin{matrix} A \to A ; . E, \text{end / ;} \\ E \to . B, \text{end / ;} \\ E \to . S, \text{end / ;} \\ B \to . \text{begin DA end ; end / ;} \\ S \to . B, \text{end / ;} \end{matrix}\right\} - \text{same as } (I_{12})$$

Step 3 : Parsing table is as follows :

T. / N.T.	ACTION					GOTO				
	begin	end	d	;	$	S	B	D	A	E
I_0	S_3					1	2			
I_1					accept					
I_2					R_1					
I_3	R_4		R_4 S_4							
I_4	R_4 S_{10}		R_4 S_6			9	8		5	7
I_5		S_{11}		S_{12}						
I_6	R_3		R_3							
I_7		R_6		R_6						

T. / N.T.	ACTION					GOTO				
	begin	end	d	;	$	S	B	D	A	E
I_8		R_7 R_1		R_7 R_1						
I_9		R_8		R_8						
I_{10}	R_4		R_4					13		
I_{11}				R_2						
I_{12}	S_{10}					9	8			14
I_{13}	S_{10}		S_6			9	8		15	7
I_{14}		R_5		R_5						
I_{15}		S_{16}		S_{12}						
I_{16}		R_5		R_5						

As grammar contains multiple entries, so grammar is not LR (1).

Q.75. Consider the following grammar :

$S \rightarrow S\,(s)/\in$

(a) Construct LR (1) set of items.

(b) Construct LR (1) Parsing table.

(c) Show the parsing stack and the action of an LR (1) parser of the input string "(( ) ( ))". *CT : W-11(13M)*

Ans.

(a) Construct LR (1) set of items :

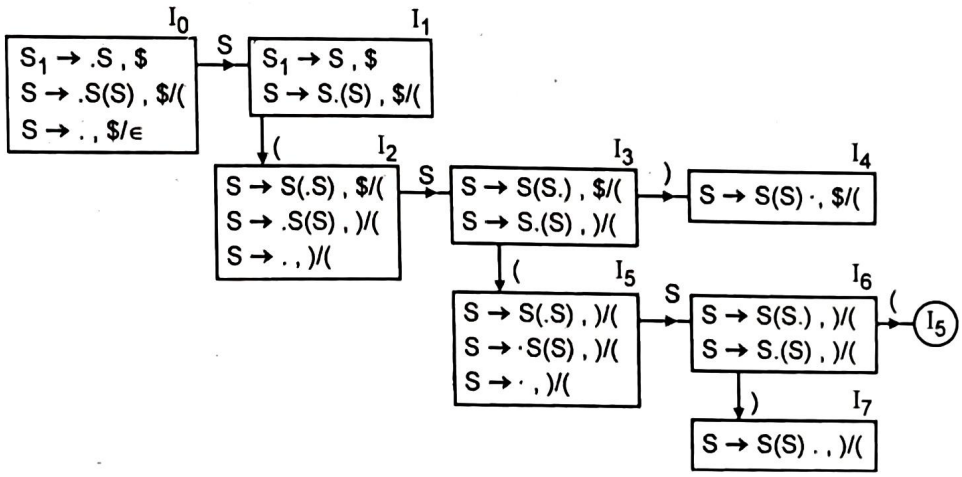
Augmented grammar :

$S_1 \rightarrow .S$

$S \rightarrow .S\,(s)$

$S \rightarrow .\in$

The LR(1) set of items are :



(b) Construct LR(1) parsing table :

T. / N.T.	ACTION			GOTO
	(	)	$	S
I_0	R_2		R_2	1
I_1	S_2		Accept	
I_2	R_2	R_2		3
I_3	S_5	S_4		
I_4	R_1		R_1	
I_5	R_2	R_2		6
I_6	S_5	S_7		
I_7	R_1	R_1		

Parsing table do not contain multiple entries. Hence grammar is LR(1).

(c) **Parsing stack contents and action of parser for $\omega = (( ) ( ))$:**

Stack contents	I/P buffer contents	Action of parser
$\$I_0$	(0 0)$	Reduce by using production $S \rightarrow \epsilon$
$\$I_0SI_1$	(0 0)$	Shift (, enter into I_2
$\$I_0SI_1(I_2$	0 0)$	Reduce by using rule $S \rightarrow \epsilon$
$\$I_0SI_1(I_2SI_3$	0 0)$	Shift (, enter into I_5
$\$I_0SI_1(I_2SI_3(I_5$	)0)$	reduce by using rule $S \rightarrow \epsilon$
$\$I_0\ SI_1\ (I_2SI_3(I_5SI_6$	)0)$	Shift), enter into I_7
$\$I_0SI_1\ (I_2SI_3(I_5SI_6)\ I_7$	0)$	Reduce by using rule $S \rightarrow S(S)$
$\$I_0SI_1(I_2SI_3$	))$	Shift (, enter into I_5
$\$I_0SI_1\ (I_2SI_3(I_5$	))$	reduce by using $S \rightarrow \epsilon$
$\$I_0SI_1(I_2SI_3(I_5SI_6$	))$	Shift), enter into I_7
$\$I_0SI_1\ (I_2SI_3\ (I_5SI_6)\ I_7$	)$	Reduce by using rule $S \rightarrow S(S)$
$\$I_0SI_1\ (I_2SI_3)$	)$	Shift), enter into I_4
$\$I_0SI_1\ (I_2SI_3)I_4$	$	Reduce by using $S \rightarrow S(S)$
$\$I_0SI_1$	$	Accept and halt, announce successful completion of parsing.

Q.76. Construct LR (0) parsing table for following grammar : **CS : S-11 (13M)**

$S \rightarrow aIJh$

$I \rightarrow IbSe / c$

$J \rightarrow KLKr / \epsilon$

$K \rightarrow d / \epsilon$

$L \rightarrow p / \epsilon$

Ans.

Step 1 : The augmented grammar :

$S' \rightarrow S$

$S \rightarrow .aIJh$

$I \rightarrow .IbSe / c$

$J \rightarrow .KLKr / \epsilon$

$K \rightarrow .d / \epsilon$

$L \rightarrow .p / \epsilon$

Step 2 : Canonical collection of set of LR(1) item are as follows :

$$I_0 = \text{Closure}\{(S' \rightarrow .S)\} = \begin{Bmatrix} S' \rightarrow .S \\ S \rightarrow .aIJh \end{Bmatrix} - I_0$$

$$\text{goto } (I_0, S) = \{S' \rightarrow S.\} \quad -(I_1)$$

$$\text{goto } (I_0, a) = \begin{Bmatrix} S \rightarrow a.IJh \\ I \rightarrow .IbSe \\ I \rightarrow c \end{Bmatrix} \quad -(I_2)$$

$$\text{goto } (I_2, I) = \begin{Bmatrix} S \rightarrow aI.Jh \\ J \rightarrow .kLkr \\ J \rightarrow . \\ K \rightarrow .d \\ K \rightarrow . \\ I \rightarrow I.bSe \end{Bmatrix} \quad -(I_3)$$

$$\text{goto } (I_2, c) = \{I \rightarrow c.\} \quad -(I_4)$$

$$\text{goto } (I_3, J) = \{S \rightarrow aIJ.h\} \quad -(I_5)$$

$$\text{goto } (I_3, K) = \begin{Bmatrix} J \rightarrow K.LKr \\ L \rightarrow .P \\ L \rightarrow . \end{Bmatrix} \quad -(I_6)$$

$$\text{goto } (I_3, d) = \{K \rightarrow d.\} \quad -(I_7)$$

$$\text{goto } (I_3, b) = \begin{Bmatrix} I \rightarrow Ib.Se \\ S \rightarrow .aIJh \end{Bmatrix} \quad -(I_8)$$

$$\text{goto } (I_5, h) = \{S \rightarrow aIJh.\} \quad -(I_9)$$

$$\text{goto } (I_6, L) = \begin{Bmatrix} J \rightarrow KL.Kr \\ K \rightarrow .d \\ K \rightarrow . \end{Bmatrix} \quad -(I_{10})$$

goto $(I_6, P) = \{L \rightarrow P.\}$ $-(I_{11})$

goto $(I_8, S) = \{I \rightarrow IbS.e\}$ $-(I_{12})$

goto $(I_8, a) = \begin{Bmatrix} S \rightarrow a.IJh \\ I \rightarrow .IbSe \\ I \rightarrow .c \end{Bmatrix}$ – same as (I_2)

goto $(I_{10}, K) = \{J \rightarrow KLk.r\}$ $-(I_{13})$

goto $(I_{10}, d) = \{K \rightarrow d.\}$ – same as (I_7)

goto $(I_{12}, e) = \{I \rightarrow IbSe.\}$ – same as (I_{14})

goto $(I_{13}, r) = \{J \rightarrow KLK.r\}$ – same as (I_{15})

Step 3 : The parsing table is as follows :

T. / N.T.	ACTION									GOTO				
	a	b	c	d	e	p	r	h	$	S	I	J	K	L
I_0	S_2									1				
I_1									Accept					
I_2			S_4								3			
I_3		S_8		S_7, R_7		R_7	R_7	R_5				5	6	
I_4			R_3			R_3	R_3	R_3						
I_5								S_9						
I_6				R_9		S_{11}	R_9							10
I_7				R_6		R_6	R_6							
I_8	S_2									12				
I_9					R_1				R_1					
I_{10}				S_7, R_7		R_7	R_7						13	
I_{11}				R_8			R_8							
I_{12}					S_{14}									
I_{13}							S_{15}							
I_{14}				R_2		R_2	R_2	R_2						
I_{15}								R_4						

As table contains multiple entries, it is not a LR (0) grammar.

DESIGN OF SLR

Q.77. Write an algorithm for construction of parsing table for SLR parser. **CT : S-14(6M)**

Ans. **Algorithm for construction of parsing table for SLR parser :**

Input :

The canonical collection set of items for augumented grammar G-refered as "C".

Output :

An LR parsing table which consist of parsing function ACTION and a GOTO function.

Procedure Let C = $\{I_0, I_1 .... I_n\}$.

The states of the parser are 0, 1, ..., n state i being constructed from I_i .

Construction of action table :

(1) For every state I_i in C do

For every terminal symbol a do

If goto (I_i , a) = I_j then

Make action [I_i , a] = S_j [for "shift" and enter into state j]

(2) For every state I_p in C whose underlying set of LR (0) items contains an item of the form $A \rightarrow \alpha$. do

For every b in FOLLOW (A) do.

Make action [I_i , b] = R_k [For 'reduce" by $A \rightarrow \alpha$]

(3) Make [I_i , $] = accept, if I_i contains an item $S_1 \rightarrow S$. [for "accept"]

Construction of Goto table :

For every I_i in C do

For every nonterminal A do

If goto (I_i , A) = I_j then

Make GoTo [I_i , A] = j

All entries not defined above are made "error".

Q.78. Consider the grammar

S → 1S1/0S0/10

Get parsing SLR (1) parsing table for the above grammar. **CT : S-11(10M)**

Ans.

Step 1 : Augmented grammar :

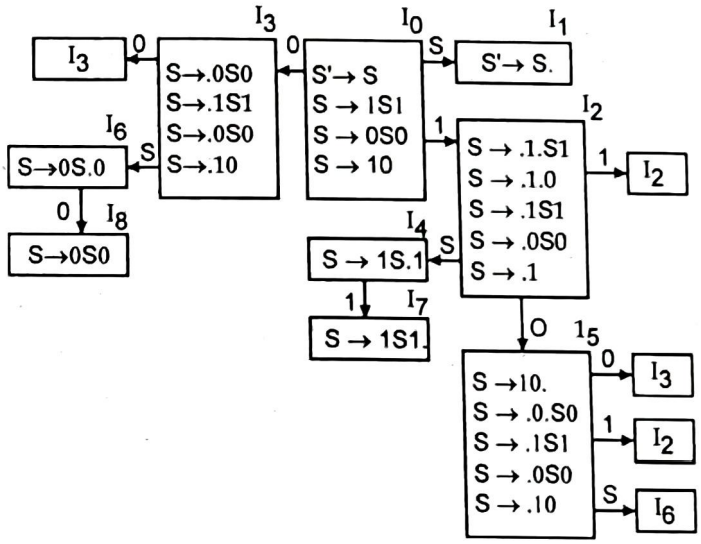
S' → .S

S → .1S1

S → .0S0

S → .10

Step 2 : Make the initial state by taking the dot product of the above grammar :



FOLLOW (S) = {$, 1, 0}

Step 3 : The parsing table will be :

T. / N.T.	ACTION			GOTO
	0	1	$	S
I_0	S_3	S_2		1
I_1			Accept	
I_2	S_5	S_2		4
I_3	S_3	S_2		6
I_4		S_7		
I_5	S_3	S_2		6
I_6	S_8			
I_7	R_2	R_2	R_2	
I_8	R_3	R_3	R_3	

It contains no multiple entries

∴ It is SLR (1)

DESIGN OF LALR

Q.79. Explain the construction of the LALR parsing table.

Ans. **Construction of the LALR parsing table :**

The steps in constructing an LALR parsing table are as follows :

(1) Obtain the canonical collection of sets of LR(1) items.

(2) If more than one set of LR(1) items exists in the canonical collection obtained that have identical cores or LR(0)s, but which have different lookaheads, then combine these sets of LR(1) items to obtain a reduced collection, C_1, of sets of LR(1) items.

(3) Construct the parsing table by using this reduced collection, as follows.

Construction of the action table :

(1) For every state I_i in C_1 do

for every terminal symbol a do

if goto $(I_i, a) = I_j$ then

make action $[I_i, a] = S_j$

*/ *for shift and enter into the state j*/*

(2) for every state I_i in C_1 whose underlying set of LR(1) items contains an item of the form $A \rightarrow \alpha$, a, do

make action $[I_i, a] = R_k$

*/*where k is the number of the production $A \rightarrow \alpha$*

*standing for reduce by $A \rightarrow \alpha$ */*

(3) make $[I_i, \$]$ = accept, if I_i contains an item $S_1 \rightarrow S .,\ \$$

Construction of the Goto Table :

Table goto table simply maps transitions on nonterminals in the DFA. Therefore, the table is constructed as follows :

for every I_i in C_1 do

for every terminal A do

if goto $(I_i, A) = I_j$ then

make goto $[I_i, A] = j$

Q.80. Write an algorithm for construction of LALR parsing table.

Ans. **Algorithm for construction of LALR parsing table :**

Input : An augmented grammar G'.

Output : The LALR parsing table functions; action and goto for G'.

Method :

(1) Construct $C = \{I_0, I_1, \ldots\ldots, I_n\}$, the collection of sets of LR (1) items.

(2) For each core present among the set of LR (1) items, find all sets having that core, and replace these sets by their union.

(3) Let $C' = \{J_0, J_1, \ldots\ldots, J_m\}$ be the resulting sets of LR (1) items. The parsing actions for state i are constructed from J_i in the same manner as LR (1) parsing table algorithm. If there is a parsing action conflict, the algorithm fails to produce a parser and the grammar is said not to be LALR (1).

(4) The goto table is constructed as follows. If J is the union of one or more sets of LR (1) items, that is, $J = I_1 \cup I_2 \ldots\ldots \cup I_k$, then the cores of goto (I, X), goto (I_2, x), ……., goto (I_k, X) are the same, since $I_1, I_2, \ldots\ldots I_k$ all have the same core. Let K' be the union of all sets of items having the same core as goto (I_1, X). Then goto (J, X) = K

The table produced by algorithm is called the LALR parsing table for G. If there are no parsing action conflicts, then the given grammar is said to be an LALR (1) grammar. The collection of sets of items constructed in step (3) is called the LALR (1) collection.

Q.81. If erroneous input is given to LALR and LR(1) parser, which parser detects error earlier? What will be action of other parser at that situation? **CS : S-10(3M)**

Ans.

- If erroneous input is given to LALR and LR(1) parser, LR(1) parser will detect error earlier.
- When presented with erroneous input, the LALR parser may proceed to do some reductions after the LR parser has declared error.
- However LALR parser will never shift another symbol after the LR parser declares an error.
- For example, an input ccd followed by $ the LR parser will put oc3c3d4 on stack and in state (4) an error is declare and current input is $. While LALR parser will push onto the stack : oc36c36d47

- State 47 with input $ reduces with $c \rightarrow d$ and stack is : oc36c36c89.
- State 89 with input $ reduces with $C \rightarrow cC$ and stack is oc36c89.
- Similar reduction applied and stack is oc2.
- Ending finally with an error in state (2) and current input $.

For above example grammar is

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC / d$

It has following LALR item collection

$C = \{I_0, I_1, I_2, I_5, I_{36}, I_{47}, I_{89}\}$

$I_{36} : C \rightarrow c.C / c / d / \$$

$C \rightarrow .cC, c/d, \$$

$C \rightarrow d, c/d/\$$

$I_{47} = C \rightarrow d., cd/\$$

$I_{89} = C \rightarrow cC., c/d/\$.$

Q.82. Comment on following statement :

(1) "Shift reduce conflict gets generated in LALR parser as a result of merging of states of LR(1), which is not there in LR(1) parser."

(2) Every LR(1) grammar is SLR but reverse is not true.

CS : S-11, W-11(5M)

Ans.

- LR parser is a bottom up parser. This parser scan the input from left to right and construct a rightmost derivation in reverse hence called as LR parser.
- In shift action parser will shift next input symbol onto stack.
- In reduce action parser locates the left end of the handle within stack and decide with what non terminal to replace the handle.
- The accept parser announces successful completion of the parsing.
- The Goto table maps every state and nonterminal pair into a state i.e. transition on nonterminal.
- The LALR is constructed which has the power in between SLR(1) and CLR(1) and storage required is same as that of SLR(1).
- Every state corresponds to the set of LR(1) items, the information about the lookaheades is also available in state itself.
- LALR parser is obtained by combining those states of LR(1) item.
- CLR which has identical core parts of item which differs in the lookaheads in their set of item representation.
- Hence it may happen that even if there is no reduce-reduce conflict in state of CLR or LR(1) parser but it may appear in LALR parser.

Q.83. Every LR(1) is SLR but reverse is not true.

Ans.

- Every SLR(1) is a canonical LR(1) grammar, but canonical LR(1) parser may have more states than the SLR(1) parser.
- An LR(1) grammar is not necessarily SLR(1), because an LR(1) parser splits state based on differing lookaheads, may avoid conflicts that would otherwise result if using the full follow set.

Q.84. Compare SLR, LR (1) and LALR parsers.

Ans.

Sr. No.	SLR	LR (1)	LALR
(1)	SLR is comparatively less powerful.	LR(1) is more powerful as compared to SLR.	LALR is more powerful than SLR.
(2)	The number of states of SLR parser is less.	The number of states of LR(1) parser is more than the SLR parser.	The number of states of LALR is same as SLR.
(3)	In SLR parser, lookahead items are not available.	In LR(1) parser lookahead items are available.	In LALR parser lookahead items are available.

Q.85. Construct LALR parsing table for the following grammar.

$S \rightarrow L = R / R$

$L \rightarrow * R / id$

$R \rightarrow L$

Ans.

Step 1 : Augmented grammar :

$S' \rightarrow . S$

$S \rightarrow .L = R$

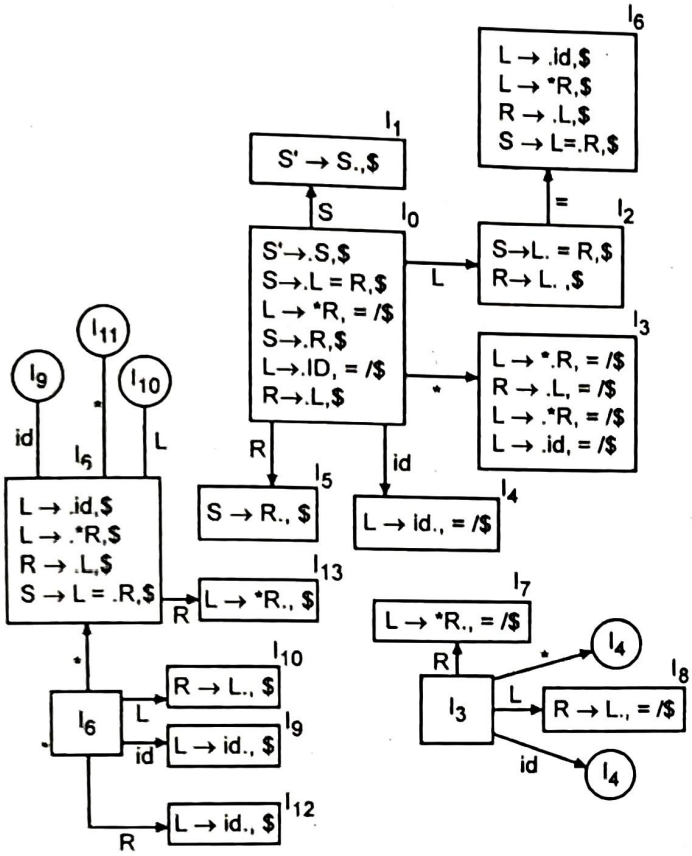
$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .id$

$R \rightarrow .L$

Step 2 : Parsing action is shown below :



Step 3 : The LR (1) parsing table will be :

T. / N.T.	ACTION			GOTO			
	=	*	id	$	S	L	R
I_0		S_3	S_4		1	2	
I_1				Accept			
I_2	S_6			R_5			
I_3		S_3	S_4			8	7
I_4	R_4			R_4			
I_5				R_3			
I_6		S_{11}	S_9			10	12
I_7	R_2			R_2			
I_8	R_5			R_5			
I_9				R_4			
I_{10}				R_5			
I_{11}		S_{11}	S_9			10	13
I_{12}				R_1			
I_{13}				R_2			

It contains no multiple entries:

Hence it is LR (1).

The states (I_8, I_{10}), (I_4, I_9), (I_3, I_{11}) and (I_7, I_{13}) are such that core part of items in these are same, hence they are combined as shown below :

$I_{810} \Rightarrow R \rightarrow L.$

$I_{49} \Rightarrow L \rightarrow id., = /\$$

$I_{311} \Rightarrow L \rightarrow *.R, = /\$$

$R \rightarrow .L, = /\$$

$L \rightarrow .*R, = / \$$

$L \rightarrow .id, = / \$$

$I_{713} \Rightarrow L \rightarrow *R., = / \$$

The productions of the grammar are numbered as :

$S \rightarrow .L = R \rightarrow (1)$

$L \rightarrow .*R \rightarrow (2)$

$S \rightarrow .R \rightarrow (3)$

$L \rightarrow .id \rightarrow (4)$

$R \rightarrow .L \rightarrow (5)$

The LALR parsing table will be :

T. / N.T.	ACTION				GOTO		
	=	*	id	$	S	L	R
I_0		S_{311}	S_{49}		1	2	
I_1				Accept			
I_2	S_6			R_5			

T. / N.T.	ACTION				GOTO		
	=	^	id	$	S	L	R
I_{311}		S_{311}	S_{49}			810	713
I_{49}				R_4			
I_5				R_3			
I_6		S_{311}	S_{49}			108	12
I_{713}	R_2			R_2			
I_{810}	R_5			R_5			
I_{12}				R_1			

It contain no multiple entries.

∴ It is LALR.

Q.86. Construct LALR parsing table for the following grammar.

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

$T \rightarrow S, T / S$

$R \rightarrow T$

Show the actions for the parser inputs :

(i) (a, a, ∧)

(ii) a ∧ (a) *CT : W-13(13M)*

Ans.

Step 1 : Augmented grammar :

$S' \rightarrow .S$

$S \rightarrow .a$

$S \rightarrow .\wedge$

$S \rightarrow .(R)$

$T \rightarrow .S, T$

$T \rightarrow .S$

$R \rightarrow .T$

Step 2 : Canonical collection of LR (1) items are :

$$I_0 = \text{Closure}(S' \rightarrow .S) = \left\{ \begin{matrix} S' \rightarrow .S, \$ \\ S \rightarrow .a, \$ \\ S \rightarrow .\wedge, \$ \\ S \rightarrow .(R), \$ \end{matrix} \right\} - I_0$$

goto $(I_0, S) = \{S' \rightarrow S., \$\} - (I_1)$

goto $(I_0, a) = \{S' \rightarrow a., \$\} - (I_2)$

goto $(I_0, \wedge) = \{S' \rightarrow \wedge., \$\} - (I_3)$

$$\text{goto } (I_0, () = \left\{ \begin{matrix} S \rightarrow (.R), \$ \\ R \rightarrow .T,) \\ T \rightarrow .S, T) \\ T \rightarrow .S,) \\ S \rightarrow .a,) / \\ S \rightarrow .\wedge) / , \\ S \rightarrow .(R) .) / , \end{matrix} \right\} - (I_4)$$

goto $(I_4, R) = \{S \rightarrow (R.), \$\} - (I_5)$

goto $(I_4, T) = \{R \rightarrow T.,)\} - (I_6)$

$$\text{goto } (I_4, S) = \left\{ \begin{matrix} T \rightarrow S., T) \\ T \rightarrow S.,) \end{matrix} \right\} - (I_7)$$

goto $(I_4, a) = \{S \rightarrow a.,) / , \} - (I_8)$

goto $(I_4, \wedge) = S \rightarrow (\wedge.,) / ,) - (I_9)$

$$\text{goto } (I_4, () = \left\{ \begin{matrix} S \rightarrow (.R),) / , \\ R \rightarrow .T,) \\ T \rightarrow .S, T,) \\ T \rightarrow .S,) \\ S \rightarrow .a,) / , \\ S \rightarrow .a,) / , \\ S \rightarrow .\wedge) / , \\ S \rightarrow .(R) .) / , \end{matrix} \right\} - (I_{10})$$

goto $(I_5,)) = S \rightarrow (R.), \$ - (I_{11})$

$$\text{goto } (I_7, ,) = \left\{ \begin{matrix} T \rightarrow S, .T,) \\ T \rightarrow .S, T,) \\ T \rightarrow .S,) \\ S \rightarrow .a,) / , \\ S \rightarrow .\wedge,) / , \\ S \rightarrow .(R),) / , \end{matrix} \right\} - (I_{12})$$

goto $(I_{12}, S) = (I_7)$

goto $(I_{12}, \wedge) = (I_9)$

goto $(I_{12}, C) = (I_{10})$

goto $(I_{12}, T) = \{T \rightarrow S, T.,)\} - (I_{14})$

goto $(I_{12}, a) = (I_8)$

goto $(I_{10}, R) = \{S \rightarrow (R.),) / , \} - (I_{13})$

goto $(I_{10}, T) = (I_6)$

goto $(I_{10}, S) = (I_7)$

goto $(I_{10}, a) = (I_8)$

goto $(I_{10}, \wedge) = (I_9)$

goto $(I_{13},)) = S \rightarrow (R)\,.\,) / , - (I_{15})$

Step 3 : Select same LR (1) items and different lookaheads :

I_2 and I_8 are same

I_3 and I_9 are same

I_4 and I_{10} are same

I_5 and I_{13} are same

I_{11} and I_{15} are same

Combine states are as follows :

$I_{28} = \{S \rightarrow a\,.\,, \$ /) / , \}$

$I_{39} = \{S \rightarrow \wedge\,.\,, \$ /) / , \}$

$I_{410} = \{S \rightarrow (\,.\,R) ,) / , / \$\}$

$I_{513} = \{S \rightarrow (R\,.\,) , \$ /) / , \}$

$I_{1115} = \{S \rightarrow (R)\,.\,, \$ /) / , \}$

Step 4 : LALR table is as follows :

T. / N.T.	ACTION						GOTO		
	a	^	(	)	,	$	S	T	R
I_0	S_{28}	S_{39}	S_{410}				1		
I_1						accept			
I_{28}				R_1	R_1	R_1			
I_{39}				R_2	R_2	R_2			
I_{410}	S_{28}	S_{39}	S_{410}				7	6	513
I_{513}				S_{1115}					
I_6				R_6					
I_7				R_5	S_{12}				
I_{12}	S_{28}	S_{39}	S_{410}				7	14	
I_{14}				R_4					
I_{1115}				R_3	R_3	R_3			

As grammar doesn't contain multiple entries, so grammar is LALR

(i) $w = a \wedge (a)$:

Stack contents	I/P buffer	Action of parser
$ I_0	a ∧ (a)	Shift to I_{28}, advance input pointer by one position towards right and enter into I_{28}.
$ I_0	∧ (a)	Error occurs, hence rejects.

(ii) $w = (a, a, \wedge)$:

Stack contents	I/P buffer	Action of parser
$ I_0	(a, a, ∧)	Shift and enter into S_{410}.
$ I_0 (I_{410}	a, a, ∧)	Shift and enter into I_{28}.
$ I_0 (I_{410} a I_{28}	, a, ∧)	Reduction using $S \rightarrow a\,.$ and goto I_7.
$ I_0 (I_{410} S I_7	a, ∧)	Reject since No valid moves.

Q.87. Test whether the following grammar is LALR or not

$S \rightarrow aSbS/bSaS/\in .$ **CT : S-10(8M)**

Ans.

Step 1 : Augmented grammar :

$S' \rightarrow .S$ ... (I)

$S \rightarrow .aSbS$ ... (II)

$S \rightarrow .bSaS$ ... (III)

$S \rightarrow .\in$ ... (IV)

Step 2 : Canonical collection of LR (1) items :

$$I_0 = \text{Closure}(S' \rightarrow .S) = \begin{Bmatrix} S' \rightarrow .S, \$ \\ S \rightarrow .aSbS, \$ \\ S \rightarrow .bSaS, \$ \\ S \rightarrow ., \$ \end{Bmatrix} - (I_0)$$

$$\text{goto } (I_0, S) = \{S' \rightarrow S\,.\,, \$\} - (I_1)$$

$$\text{goto } (I_0, a) = \begin{Bmatrix} S \rightarrow a\,.\,SbS, \$ \\ S \rightarrow .aSbS, b \\ S \rightarrow .bSaS, b \\ S \rightarrow ., b \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_0, b) = \begin{Bmatrix} S \to b\,.\,SaS, \$ \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix} - (I_3)$$

$$\text{goto } (I_3, S) = \{S \to bS\,.\,aS, \$\} - (I_4)$$

$$\text{goto } (I_3, a) = \begin{Bmatrix} S \to a\,.\,SbS, a \\ S \to .\,aSbS, b \\ S \to .\,bSaS, b \\ S \to ., b \end{Bmatrix} - (I_5)$$

$$\text{goto } (I_3, b) = \begin{Bmatrix} S \to a\,.\,SbS, a \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix} - (I_6)$$

$$\text{goto } (I_6, S) = S \to bS\,.\,aS, a \quad - (I_7)$$

$$\text{goto } (I_7, a) = \begin{Bmatrix} S \to bSa\,.\,S, a \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix} - (I_8)$$

$$\text{goto } (I_8, S) = S \to bSaS\,., a \quad - (I_9)$$

$$\text{goto } (I_2, S) = S \to aS\,.\,bS, \$ \quad - (I_{10})$$

$$\text{goto } (I_2, a) = \begin{Bmatrix} S \to a\,.\,SbS, b \\ S \to .\,aSbS, b \\ S \to .\,bSaS, b \\ S \to ., b \end{Bmatrix} - (I_{11})$$

$$\text{goto } (I_2, b) = \begin{Bmatrix} S \to b\,.\,SaS, b \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix} - (I_{12})$$

$$\text{goto } (I_{11}, S) = \{S \to aS\,.\,bS, b\} - (I_{13})$$

$$\text{goto } (I_{13}, b) = \begin{Bmatrix} S \to aSb\,.\,S, b \\ S \to .\,aSbS, b \\ S \to .\,bSaS, b \\ S \to ., b \end{Bmatrix} - (I_{14})$$

$$\text{goto } (I_{14}, S) = \{S \to aSbS\,., b\} - (I_{15})$$

$$\text{goto } (I_{14}, b) = (I_{12})$$

$$\text{goto } (I_{14}, a) = (I_{11})$$

$$\text{goto } (I_{12}, b) = (I_6)$$

$$\text{goto } (I_{12}, a) = (I_5)$$

$$\text{goto } (I_4, a) = \begin{Bmatrix} S \to bSa\,.\,S, b \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix} - (I_{16})$$

$$\text{goto } (I_{16}, S) = S \to bSaS\,., \$ \quad - (I_{17})$$

$$\text{goto } (I_{16}, b) = (I_3)$$

$$\text{goto } (I_{16}, a) = (I_2)$$

$$\text{goto } (I_{10}, b) = \begin{Bmatrix} S \to aSb\,.\,S, \$ \\ S \to .\,aSbS, \$ \\ S \to .\,bSaS, \$ \\ S \to ., \$ \end{Bmatrix} - (I_{18})$$

$$\text{goto } (I_{18}, S) = S \to aSbS\,., \$ \quad - (I_{19})$$

$$\text{goto } (I_{18}, a) = (I_2)$$

$$\text{goto } (I_{18}, b) = (I_3)$$

$$\text{goto } (I_5, S) = S \to aS\,.\,bS, a \quad - (I_{20})$$

$$\text{goto } (I_5, b) = (I_{12})$$

$$\text{goto } (I_5, a) = (I_{11})$$

$$\text{goto } (I_{20}, b) = \begin{Bmatrix} S \to aSb\,.\,S, a \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix} - (I_{21})$$

$$\text{goto } (I_{21}, b) = (I_6)$$

$$\text{goto } (I_{21}, a) = S \to aSbS\,., a - (I_{22})$$

$$\text{goto } (I_{12}, S) = \{S \to bS\,.\,aS, b\} \quad - (I_{23})$$

$$\text{goto } (I_{23}, a) = \begin{Bmatrix} S \to bSa\,.\,S, b \\ S \to .\,aSbS, b \\ S \to .\,bSaS, b \\ S \to ., b \end{Bmatrix} - (I_{24})$$

$$\text{goto } (I_{24}, S) = \{S \to bSaS\,., b\} \quad - (I_{25})$$

Here states I_2, I_5, I_{11} ate having same core items and different lookahead so combine such states

$$I_2, I_5, I_{11} \Rightarrow A = \begin{Bmatrix} S \to a\,.\,SbS, \$ / a / b \\ S \to .\,aSbS, b \\ S \to .\,bSaS, a \\ S \to ., b \end{Bmatrix}$$

$$I_3, I_6, I_{12} = B = \begin{Bmatrix} S \to b\,.\,SaS, \$ / a / b \\ S \to .\,aSbS, a \\ S \to .\,bSaS, a \\ S \to ., a \end{Bmatrix}$$

$$I_4, I_7, I_{23} = C = S \to bS\,.\,aS, \$ / a / b$$

$$I_{10}, I_{13}, I_{20} = D = S \to aS\,.\,bS, \$ / a / b$$

$$I_{14}, I_{18}, I_{21} = E = \begin{Bmatrix} S \to aSb\,.\,S, \$ / a / b \\ S \to .\,bSaS, \$ / a / b \\ S \to .\,aSbS, \$ / a / b \\ S \to .\,, \$ / a / b \end{Bmatrix}$$

$$I_8, I_{16}, I_{24} = F = \begin{Bmatrix} S \to bSa\,.\,S, \$ / a / b \\ S \to .\,aSbS, \$ / a / b \\ S \to .\,bSaS, \$ / a / b \\ S \to .\,, \$ / a / b \end{Bmatrix}$$

$I_{15}, I_{19}, I_{22} = G = S \to aSbS.\,, \$ / a / b$

$I_9, I_{17}, I_{25} = H = \{S \to bSaS.\,, \$ / a / b$

The final states are as follows :

I_0, I_1, A, B, C, D, E, F, G, H

Step 3 : The parsing table is as follows :

T. / N.T.	ACTION			GOTO
	a	b	$	S
I_0	S_A	S_B	R_3	1
I_1			accept	
A	S_A	S_B / R_3		D
B	S_A / R_3	S_B		C
C	S_F			
D		S_E		
E	S_A / R_3	S_B / R_3	R_3	G
F	S_A / R_3	S_B / R_3	R_3	H
G	R_1	R_1	R_1	
H	R_2	R_2	R_2	

Q.88. Construct LALR parsing table for the grammar :

S → abSa | aaAc | b

A → dAb | b *CT : S-09(8M)*

Ans.

Step 1 : Augmented grammar :

S' → S ... (I)

S → abSa ... (II)

S → aaAc .... (III)

S → b ... (IV)

A → dAb ... (V)

A → b ... (VI)

Step 2 : The canonical collection of LR(1) items are as follows :

$$I_0 = \text{Closure}(S' \to .S) = \begin{Bmatrix} S' \to .S, \$ \\ S \to .abSa, \$ \\ S \to .aaAc, \$ \\ S \to .b, \$ \end{Bmatrix} - (I_0)$$

goto $(I_0, S) = \{S' \to S., \$\} - (I_1)$

$$\text{goto } (I_0, a) = \begin{Bmatrix} S \to .abSa, \$ \\ S \to a.aAc, \$ \end{Bmatrix} - (I_2)$$

goto $(I_0, b) = \{S \to b., \$\} - (I_3)$

$$\text{goto } (I_2, a) = \begin{Bmatrix} S \to aa.Ac, \$ \\ A \to .dAb, c \\ A \to .b, c \end{Bmatrix} - (I_5)$$

$$\text{goto } (I_2, b) = \begin{Bmatrix} S \to ab.Sa, \$ \\ S \to .a\,bSa, a \\ S \to .aaAc, a \\ S \to .b, a \end{Bmatrix} - (I_4)$$

goto $(I_4, S) = \{S \to abS.a, \$\} - (I_6)$

$$\text{goto } (I_4, a) = \begin{Bmatrix} S \to a.bSa, a \\ S \to a.aAc, a \end{Bmatrix} - (I_7)$$

goto $(I_4, b) = \{b., a\} - (I_8)$

goto $(I_5, A) = \{S \to aaA.c, \$\} - (I_9)$

$$\text{goto } (I_5, d) = \begin{Bmatrix} A \to d.Ab, c \\ A \to .dAb, c \\ A \to .b, c \end{Bmatrix} - (I_{10})$$

goto $(I_5, b) = \{A \to b., c\} - (I_{11})$

goto $(I_6, a) = \{S \to abSa., \$\} - (I_{12})$

$$\text{goto } (I_7, b) = \begin{Bmatrix} S \to ab.Sa, a \\ S \to .abSa, a \\ S \to .aaAc, a \\ S \to .b, a \end{Bmatrix} - (I_{13})$$

$$\text{goto } (I_7, a) = \begin{Bmatrix} S \to aa.Ac, a \\ A \to .dAb, c \\ A \to .b, c \end{Bmatrix} - (I_{14})$$

goto $(I_9, c) = \{S \to aaAc., \$\} - (I_{15})$

goto $(I_{10}, A) = \{A \to dA.b, c\} - (I_{16})$

goto $(I_{10}, d) = \begin{Bmatrix} A \to d.Ab, c \\ A \to .dAb, b \\ A \to .b, b \end{Bmatrix} - (I_{17})$

goto $(I_{10}, b) = \{A \to b., b\} - (I_{18})$

goto $(I_{13}, S) = \{S \to abS.a, a\} - (I_{19})$

goto $(I_{13}, a) = \begin{Bmatrix} S \to a.bSa, a \\ S \to a.aAc, a \end{Bmatrix}$ – same as (I_7)

goto $(I_{13}, b) = \{S \to b., a\}$ – same as (I_8)

goto $(I_{14}, A) = \{S \to aaA.c, a\} - (I_{20})$

goto $(I_{14}, b) = \{A \to b., c\}$ – same as (I_{10})

goto $(I_{14}, d) = \begin{Bmatrix} A \to d.Ab, c \\ A \to .dAb, b \\ A \to .b, b \end{Bmatrix}$ – same as (I_{11})

goto $(I_{16}, b) = \{A \to dAb., c\} - (I_{21})$

goto $(I_{17}, A) = \{A \to dA.b, b\} - (I_{22})$

goto $(I_{17}, d) = \{A \to d.Ab, b\}$ – same as (I_{17})

goto $(I_{17}, b) = \{A \to b., b\}$ – same as (I_{18})

goto $(I_{19}, a) = \{S \to abSa., a\} - (I_{23})$

goto $(I_{20}, c) = \{S \to aaAc., a\} - (I_{24})$

goto $(I_{22}, b) = \{A \to dAb., b\} - (I_{25})$

Step 3 : LR (1) parsing table is as follows :

T. / N.T.	ACTION					GOTO	
		a	b	c	d	S	S
I_0	S_2	S_3					2
I_1					accept		
I_2	S_5	S_4					
I_3					R_3		
I_4	S_7	S_8					
I_5		S_{11}		S_{10}			9
I_6	S_{12}						
I_7	S_{14}	S_{13}					
I_8	R_3						
I_9			S_{15}				
I_{10}		S_{18}		S_{17}			16
I_{11}			R_5				
I_{12}					R_1		
I_{13}	S_7	S_8				19	
I_{14}		S_{11}		S_{10}			20
I_{15}				R_2			
I_{16}		S_{21}					
I_{17}		S_{18}		S_{17}			
I_{18}		R_5					
I_{19}	S_{23}						
I_{20}			S_{24}				
I_{21}			R_4				
I_{22}		S_{25}					
I_{23}	R_1						
I_{24}	R_2						
I_{25}		R_4					

Since there are no more than one entries in each cell so the grammar is LR (1).

Step 4 : Now checking for LALR :

Combine the states where common LR (0) items but different lookaheads states are :

(1) I_2 and I_7 combined as

$$I_{27} = \begin{Bmatrix} S \to a \cdot bSa, \$/a \\ S \to a \cdot aAc, \$/a \end{Bmatrix}$$

(2) States I_3 and I_8 combined as

$$I_{38} = \{S \to b., \$/a\}$$

(3) I_4 and I_{13} combined as

$$I_{413} = \begin{Bmatrix} S \to ab \cdot Sa, \$/a \\ S \to \cdot abSa, a \\ S \to \cdot aaAc, a \\ S \to \cdot b, a \end{Bmatrix}$$

(4) I_5 and I_{14} combined as

$$I_{514} = \begin{Bmatrix} S \to aa \cdot Ac, \$/a \\ A \to \cdot dAb, c \\ A \to \cdot b, c \end{Bmatrix}$$

(5) I_6 and I_{19} combined as

$I_{619} = \{S \to abS \cdot a, \$/a\}$

(6) I_9 and I_{20} combined as

$I_{920} = \{S \to aaA \cdot c, \$/a\}$

(7) I_{10} and I_{17} :

$$I_{1017} = \begin{Bmatrix} A \to d \cdot Ab, c/b \\ A \to \cdot dAb, b \\ A \to \cdot b, b \end{Bmatrix}$$

(8) I_{11} and I_{18} :

$I_{1118} = \{A \to b \cdot, c/b\}$

(9) I_{12} and I_{23} :

$I_{1223} = \{S \to abSa \cdot, \$/a\}$

(10) I_{15} and I_{24} :

$I_{1524} = \{S \to aaAc \cdot, \$/a\}$

(11) I_{16} and I_{22} :

$I_{1622} = \{A \to dA \cdot b, c/b\}$

(12) I_{21} and I_{25} :

$I_{2125} = \{A \to dAb \cdot, c/b\}$

So there are total 14 states so draw LALR parsing table for it.

T. / N.T.	ACTION					GOTO	
	a	b	c	d	$	S	A
I_0	S_{27}	S_{38}				1	
I_1					accept		
I_{27}	S_{514}	S_{413}					
I_{38}	R_3						
I_{413}	S_{27}	S_{38}				619	
I_{514}		S_{1118}		S_{1017}			920
I_{619}	S_{1223}						
I_{920}			S_{1524}				
I_{1017}		S_{1118}		S_{1017}			1622
I_{1118}		R_5	R_5				
I_{1223}					R_1		
I_{1524}					R_2		
I_{1622}		S_{2125}					
I_{2125}		R_4					

As there are no multiple entries in the parsing table so grammar is LALR.

Q.89. Given grammar :

$S \to Aa / aAC / Bc / bBa$

$A \to d$

$B \to d$

Check whether given grammar is LALR(1) or not.

CT : S-14(7M)

Ans.

Step 1 : Augmented grammar :

$S' \to S$

$S \to Aa \mid aAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

Step 2 : The canonical collection of sets of LR(1) items is :

$I_0 = \{$

$S' \to . S, \$$

$S \to . Aa, \$$

$S \to . aAc, \$$

$S \to . Bc, \$$

$S \to . bBa, \$$

$A \to . d, a$

$B \to . d, c$

$\}$

goto $(I_0, S) = \{S' \to S., \$\} = I_1$

goto $(I_0, A) = \{S \to A.a, \$\} = I_2$

goto $(I_0, B) = \{S \to B.c, \$\} = I_3$

goto $(I_0, a) = \{S \to a.Ac, \$$

$A \to .d, c\} = I_4$

goto $(I_0, b) = \{S \to b.Ba, \$$

$B \to .d, a\} = I_5$

goto $(I_0, d) = \{A \to d., a$

$B \to d., c\} = I_6$

goto $(I_2, a) = \{S \to Aa., \$\} = I_7$

goto $(I_3, c) = \{S \to Bc., \$\} = I_8$

goto $(I_4, A) = \{S \to aA.c, \$\} = I_9$

goto $(I_4, d) = \{A \to d., c\} = I_{10}$

goto $(I_5, B) = \{S \to bB.a, \$\} = I_{11}$

goto $(I_5, d) = \{B \to d., a\} = I_{12}$

goto $(I_9, c) = \{S \to aAc., \$\} = I_{13}$

goto $(I_{11}, a) = \{S \to bBa., \$\} = I_{14}$

Since no sets of LR(1) items in the canonical collection have identical LR(0)-part items and differ only in their lookaheads, the LALR(1) parsing table for the above grammar is as shown in Table.

Step 3 : LALR(1) parsing table for grammar :

T. / N.T.	ACTION					GOTO		
	a	b	c	d	$	S	A	B
I_0	S_4	S_5		S_6		1	2	3
I_1					accept			
I_2	S_7							
I_3			S_8					
I_4				S_{10}			9	
I_5				S_{12}				11
I_6	R_5		R_6					
I_7				R_1				
I_8				R_3				
I_9			S_{13}					
I_{10}			R_5					
I_{11}	S_{14}							
I_{12}	R_6							
I_{13}					R_2			
I_{14}					R_4			

As the table do not contain multiple entries, so grammar is LALR(1).

Q.90. Find canonical states of grammar

$B \to bDAe$

$D \to Dd ; / \epsilon$

$A \to A ; E / \epsilon$

$E \to B / a$ **CS : W-10(3M)**

Ans.

Step 1 : Augmented grammar :

$B' \to .B$

$B \to .bDAe$

$D \to .Dd ;$

$D \to .\epsilon$

$D \to .A ; E$

$A \to .\epsilon$

$E \to .B$

$E \to .a$

Step 2 : Canonical states for grammar :

$I_0 = \{B' \to .B, \$\}$

$$= \left\{ \begin{matrix} B' \to .B, \$. \\ B \to .bDAe, \$. \end{matrix} \right\} - (I_0)$$

goto $(I_0, B) = \{B' \to B., \$\} - (I_1)$

$$\text{goto } (I_0, b) = \left\{ \begin{matrix} B \to b.DAe, \$ \\ D \to .Dd ;, d \\ D \to ., d \end{matrix} \right\} - (I_2)$$

$$\text{goto}(I_2, D) = \begin{Bmatrix} B \to bD.Ae, \$ \\ A \to .A;E, \$ \\ A \to ., \$ \end{Bmatrix} - (I_3)$$

$$\text{goto}(I_3, A) = \begin{Bmatrix} B \to bDA.e, \$ \\ A \to A.;E, \$/; \end{Bmatrix} - (I_4)$$

$$\text{goto}(I_4, e) = B \to bDAe., \$ - (I_5)$$

$$\text{goto}(I_4, ;) = \begin{Bmatrix} B \to A;.E, \$/; \\ E \to .B., \$/; \\ E \to ., \$ \end{Bmatrix} - (I_6)$$

Q.91. Construct LALR(1) parser for the grammar :

S → aIJh

I → IbSe/c

J → KLKr/ ∈

K → d/ ∈

L → p/ ∈

CS : S-12(14M), S-13(13M)

Ans.

Step 1 : Augmented grammar :

S' → .S ... (I)

S → .aIJh ... (II)

I → .IbSe ... (III)

I → .c ... (IV)

J → .KLKr ... (V)

J → .∈ ... (VI)

K → .d ... (VII)

K → .∈ ... (VIII)

L → .P ... (IX)

L → .∈ ... (X)

Step 2 : Canonical states for grammar :

I_0 = Closure ({S' → .S, $})

$$= \begin{Bmatrix} S' \to .S, \$ \\ S \to .aIJh, \$ \end{Bmatrix} - (I_0)$$

$$\text{goto}(I_0, S) = \{S' \to S., \$\} - (I_1)$$

$$\text{goto}(I_0, a) = \begin{Bmatrix} S \to a.IJh, \$ \\ I \to .IbSe, d/p/r/h \\ I \to .c, d/p/r/h \end{Bmatrix} - (I_2)$$

$$\text{goto}(I_2, I) = \begin{Bmatrix} S \to aI.Jh, \$ \\ J \to .KLKr, h \\ J \to ., h \\ K \to .d, d/p/r \\ K \to ., d/p/r \\ I \to I.bSe, d/p/r/h \end{Bmatrix} - (I_3)$$

$$\text{goto}(I_2, c) = \{I \to c., d/p/r/h\} - (I_4)$$

$$\text{goto}(I_3, J) = \{S \to aIJ.h, \$\} - (I_5)$$

$$\text{goto}(I_3, K) = \begin{Bmatrix} J \to K.LKr, h \\ L \to .p, d/r \\ L \to ., d/r \end{Bmatrix} - (I_6)$$

$$\text{goto}(I_3, d) = \{K \to d., d/p/r\} - (I_7)$$

$$\text{goto}(I_3, b) = \begin{Bmatrix} I \to Ib.Sc, d/p/r/h \\ S \to .aIJh, e \end{Bmatrix} - (I_8)$$

$$\text{goto}(I_5, h) = \{S \to aIJh., \$\} - (I_9)$$

$$\text{goto}(I_6, L) = \begin{Bmatrix} J \to KL.Kr, h \\ K \to .d, r \\ K \to ., r \end{Bmatrix} - (I_{10})$$

$$\text{goto}(I_6, p) = \{L \to p., d/r\} - (I_{11})$$

$$\text{goto}(I_8, a) = \begin{Bmatrix} S \to a.IJh, e \\ I \to .IbSe, d/p/r/h \\ I \to .c, d/p/r/h \end{Bmatrix} - (I_{13})$$

$$\text{goto}(I_8, S) = \{I \to IbS.e, d/p/r/h\} - (I_{12})$$

$$\text{goto}(I_{10}, K) = J \to KLK.r, h - (I_{14})$$

$$\text{goto}(I_{10}, d) = K \to d., r - (I_{15})$$

$$\text{goto}(I_{12}, e) = \{I \to IbSe., d/p/r/h\} - (I_{16})$$

$$\text{goto}(I_{13}, I) = \begin{Bmatrix} S \to aI.Jh, e \\ I \to I.bSe, d/p/r/h \\ J \to .KLKr, h \\ J \to ., h \\ K \to .d, d/p/r \\ K \to ., d/p/r \end{Bmatrix} - (I_{17})$$

$$\text{goto}(I_{13}, c) = (I_4)$$

$$\text{goto}(I_{14}, r) = \{J \to KLKr., h\} - (I_{18})$$

$$\text{goto}(I_{17}, b) = (I_8)$$

$$\text{goto}(I_{17}, d) = (I_7)$$

$$\text{goto}(I_{17}, J) = (S \to aIJ.h, e) - (I_{19})$$

goto $(I_{17}, k) = (I_6)$

goto $(I_{19}, h) = S \rightarrow aIJh.,\ e \ - (I_{20})$

Step 3 : Parsing table :

T. / N.T.	ACTION									GOTO				
	a	b	c	d	e	p	r	h	$	S	I	J	K	L
I_0	S_2									1				
I_1									accept					
I_2			S_4								3			
I_3		S_8		$S_1\ R_7$		R_7	R_7	R_5					5	6
I_4				R_3		R_3	R_3	R_3						
I_5								S_9						
I_6						S_{11}								10
I_7				R_6		R_6	R_6				12			
I_8	S_{13}								R_1					
I_9														
I_{10}				S_{15}									14	
I_{11}				R_8			R_8							
I_{12}					S_1									
I_{13}			S_4											
I_{14}							S_{18}							
I_{15}							R_6							
I_{16}				R_2		R_2	R_2	R_2						
I_{17}		S_8		$S_7\ r_7$		R_7	R_7	R_5				19	6	
I_{18}								R_4						
I_{19}								S_{20}						
I_{20}						R_1								

As it contains shift reduce conflict

So it is not LALR (1) table.

Q.92. Construct LALR(1) parsing table for grammar :

$S \rightarrow L/a$

$L \rightarrow wGdS/dSwG$

$G \rightarrow b$ **CS : S-14(13M)**

Ans.

Step 1 : Augmented grammar :

$S' \rightarrow .S$ ... (I)

$S \rightarrow .L$ ... (II)

$S \rightarrow .a$ ... (III)

$L \rightarrow .wGdS$ ... (IV)

$L \rightarrow .dSwG$ ... (V)

$G \rightarrow .b$ ... (VI)

Step 2 : Canonical set for given grammar :

$I_0 = \{S' \rightarrow .S, \$\}$

$$= \left\{ \begin{array}{c} S' \rightarrow .S, \$ \\ S \rightarrow .L, \$ \\ S \rightarrow .a, \$ \\ L \rightarrow .wGdS, b \\ L \rightarrow .dSwG, a/w/d \end{array} \right\} - (I_0)$$

goto $(I_0, S) = \{S' \rightarrow S; \$.\} - (I_1)$

goto $(I_0, L) = \{S \rightarrow L., \$\} - (I_2)$

goto $(I_0, d) = \left\{ \begin{array}{c} S \rightarrow d., \$ \\ L \rightarrow d.SwG, a/w/d \end{array} \right\} - (I_3)$

goto $(I_0, w) = \left\{ \begin{array}{c} L \rightarrow w.GdS, b \\ G \rightarrow .b, d \end{array} \right\} - (I_4)$

goto $(I_3, S) = \{L \rightarrow dS.wG, a/w/d\} - (I_5)$

goto $(I_3, L) = \{S \rightarrow L., w\} - (I_6)$

goto $(I_3, d) = \{S \rightarrow a., w\} - (I_7)$

goto $(I_4, G) = \{L \rightarrow wG.ds, b\} - (I_8)$

goto $(I_4, b) = \{G \rightarrow b., d\} - (I_9)$

goto $(I_5, w) = \left\{ \begin{array}{c} L \rightarrow dSw.G, a/w/d \\ G \rightarrow .b, a/w/d \end{array} \right\} - (I_{10})$

goto $(I_8, d) = \left\{ \begin{array}{c} L \rightarrow wGd.S, b \\ S \rightarrow .L, \$ \\ S \rightarrow .a, \$ \end{array} \right\} - (I_{11})$

goto $(I_6, P) = \{L \rightarrow P., d/r\} - (I_{11})$

goto $(I_{10}, G) = \{L \rightarrow dSwG., a/w/d\} - (I_{12})$

goto $(I_{10}, b) = \{G \rightarrow b., a/w/d\} - (I_{13})$

goto $(I_{11}, S) = \{L \rightarrow wGdS., b\} - (I_{14})$

goto $(I_{11}, L) = \{S \rightarrow L., \$\}$ same as $- (I_2)$

goto $(I_{11}, d) = \{S \rightarrow a., \$\} - (I_{15})$

As we see the lookaheads and states output,

(I_2) and $(I_6) \rightarrow I_{26}$

(I_7) and $(I_{15}) \rightarrow I_{715}$

(I_9) and $(I_{13}) \rightarrow I_{913}$

So optimized states are I_{26}, I_{715}, I_{913}

Step 3 : Parsing table is as follows :

T. / N.T.	ACTION				GOTO			
	a	d	w	b	S	S	L	G
I_0		S_3	S_4			1	26	
I_1					accept			
I_{26}			R_1		R_1			
I_3		S_{715}				5	26	
I_4				I_{913}				I_8
I_5			S_{10}					
I_{715}			R_2					
I_8		S_{11}						
I_{913}	R_5	R_5	R_5					
I_{10}	R_4	R_4	R_4	S_{913}				I_2
I_{11}		S_{715}				14	2	
I_{12}								
I_{14}				R_4				
I_{15}					R_2			

DESIGN OF CLR

Q.93. Write an algorithm to find canonical collection of LR(1) items. Also give an algorithm to construct ACTION and GOTO table for the same. **CT : S-10(7M)**

Ans. Algorithm to find canonical collections of LR (1) items :

(1) Add every item in I to closure (I)

(2) Repeat

For every item of the form $A \rightarrow \alpha . B \beta$, a in closure (I) do

for every production $\beta \rightarrow \gamma$ do. Add $\beta - \gamma$, First (βa) to closure (I) until no new items can be added to closure (I).

Algorithm to construct Action Table and Goto Table :

Action Table :

(1) for every state I_i in C do

for every terminal symbol a do

if goto $(I_i, a) = I_j$ then

make action $[I_i, a] = S_j$

(2) for every state I_i in C whose underlying set of LR (1) item contains an item of the form

$A \rightarrow \alpha$, a do

Make action $[I_i, a] = R_k$

(3) Make $(I_i, \$) =$ accept if I_i contains an item $S_1 \rightarrow S., \$$

Goto table :

for every I_i in c do

for every non terminal A do

if goto $(I_i, A) = I_j$ then

make goto $(I_i, A) = j$

Q.94. Construct canonical LR (0) parsing table for grammar :

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow * R$

$L \rightarrow id$

$R \rightarrow L$ **CS : W-11(8M)**

Ans.

Step 1 : The augmented grammar :

$S' \rightarrow .S$

$S \rightarrow .L = R$ ... (I)

$S \rightarrow .R$ ... (II)

$S \rightarrow .*R$ ... (III)

$L \rightarrow .id$ ... (IV)

$R \rightarrow .L$ ... (V)

Step 2 : Parsing action of grammar is illustrated below :

$$I_0 = \text{Closure}\{(S' \rightarrow .S)\} = \begin{Bmatrix} S' \rightarrow .S \\ S \rightarrow .L = R \\ S \rightarrow .R \\ L \rightarrow .*R \\ L \rightarrow .id \\ R \rightarrow .L \end{Bmatrix}$$

$$\text{goto } (I_0, S) = \{S' \rightarrow S.\} - (I_1)$$

$$\text{goto } (I_0, L) = \begin{Bmatrix} S \rightarrow L. = R \\ R \rightarrow L. \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_0, R) = \{S \rightarrow R.\} - (I_3)$$

$$\text{goto } (I_0, *) = \begin{Bmatrix} L \rightarrow *.R \\ R \rightarrow .L \\ L \rightarrow .id \\ L \rightarrow .*R \end{Bmatrix} - (I_4)$$

$$\text{goto } (I_0, id) = \{L \rightarrow id.\} - (I_5)$$

$$\text{goto } (I_2, =) = \begin{Bmatrix} S \rightarrow L = .R \\ R \rightarrow L. \end{Bmatrix} - (I_6)$$

$$\text{goto } (I_4, R) = \{L \rightarrow *R.\} - (I_7)$$

$$\text{goto } (I_4, L) = \{R \rightarrow L.\} - (I_8)$$

$$\text{goto } (I_4, *) = \{L \rightarrow *.R\} - (I_4)$$

$$\text{goto } (I_4, id) = \{L \rightarrow id.\} - (I_5)$$

$$\text{goto } (I_6, R) = \{S \rightarrow L = R.\} - (I_9)$$

$$\text{goto } (I_6, L) = \{R \rightarrow L.\} - (I_8)$$

$$\text{goto } (I_6, *) = \{L \rightarrow *R.\} - \text{same as } (I_4)$$

$$\text{goto } (I_6, id) = \{L \rightarrow id.\} - \text{same as } (I_5)$$

Step 3 : The parsing table is as follows :

T. / N.T.	ACTION				GOTO		
	=	*	id	$	S	L	R
I_0		S_4	S_5		1	2	3
I_1				accept			
I_2	S_6 S_5			R_5			
I_3				R_2			
I_4		S_4	S_5			8	7
I_5	R_4			R_4			
I_6		S_4	S_5			8	9
I_7	R_3			R_3			
I_8	R_5			R_5			
I_9				R_1			

As grammar contain multiple entries so it is not CLR (1).

Q.95. What is viable prefix? Calculate all viable prefixes for the string "(a, (a, a))" using following grammar :

S → a / ∧ / (T)

S → T, s/s *CT : S-13(3M)*

Ans. Viable prefix :

- Viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of shift/reduce parser.
- It is always possible to add terminal symbols to the end of a viable prefix to obtain a right sentential form

Given grammar :

S → a / ∧ / (T)

S → T, s / s

string w ≡ (a,(a, a))

Action for parser :

Stack	Input	Action
$	(a,(a,a))$	shift (
$(	a,(a,a))$	shift a
$(a	,(a,a))$	reduce by S → a
$(S	,(a,a))$	reduce by T → S
$(T	,(a,a))$	shift ,
$(T,	a,a))$	shift (
$(T,(	,a))$	reduce by S → a
$(T,(S	,a))$	reduce by T → S
$(T,(T	,a))$	shift ,
$(T,(T,	a))$	shift a
$(T,(T,a	))$	reduce by S → a
$(T,(T,S	))$	reduce by T → T, S
$(T,(T	))$	shift +)
$(T,(T)	)$	reduce by S → (T)
$(T,S	)$	reduce by T → T, S
$(T	)$	shift)
$(T)	$	reduce by S → (T)
$S	$	Accept

Now viable prefixes for some handles are as follows :

Handle	Viable prefixes
a	(a
a	(T, (a
a	(T, (T, a
S	(S
S	(T, (S
S	(T, (T, S
S	(T, S
(S'	(T, (S
(T, (T	(T, (T
(T, a	(T, (T, a
T, a	(T, (T, a
T, S	(T, S
T)	(T)

Q.96. Discuss use of viable prefix in parsing.

Ans.

- Viable prefix of the grammar, that is, prefixes of right-sententional forms that do not contain any symbols to the right of the handle.
- A viable prefix is so called because it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sententional form. Therefore, there is apparently no error as long as the portion of the input seen to a given point can be reduced to a viable prefix.
- We say item $A \rightarrow \beta_1,\beta_2$ is valid prefix $\alpha\beta_1$ if there is a derived $S^1 \underset{rm}{\overset{*}{\Rightarrow}} \alpha AW \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 W$. In general, an item will be valid for many viable prefixes.
- To construct a "Simple" LR parser for a grammar, if such parser exists, the central idea is the construction of DFA from the grammar and then turn this DFA into an LR parsing table. The DFA recognizes viable prefixes of grammar.
- We can easily compute the set of valid items for each viable prefix that can appear on the stack of an LR parser.
- In fact, it is a major theorem of LR parsing theory that the set of valid item for a viable prefix γ is exactly the set of item reached from the initial state along a path labelled γ. The DFA constructed from the canonical collection of set of item with transition given by GOTO.
- In essence, the set of valid items embodies all the useful information that can begin from the stack.

DEALING WITH AMBIGUITY OF THE GRAMMAR

Q.97. What is ambiguous grammar? Check whether the given CFG is ambiguous or not?

$S \rightarrow aSbS|bSaS|\in$

CT : W-12(4M), S-12(3M)

Ans. Ambiguous grammar :

- If there exists more than one parse tree for some string "w" in L(G) then G is said to be an ambiguous grammar.
- An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence.

Example :

Consider a grammar having the productions listed below :

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

For string w = id + id * id, there exists more than one parse tree as shown below :

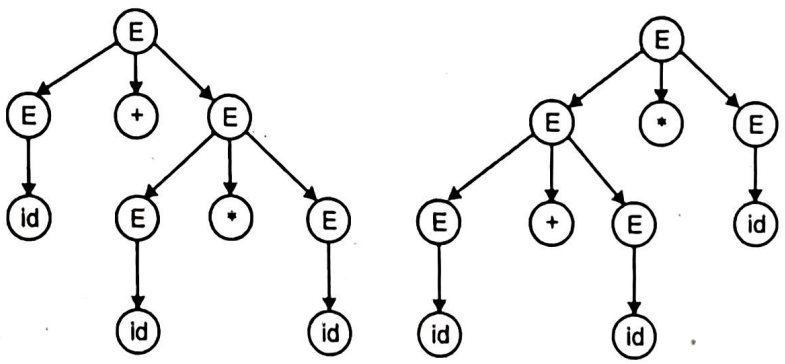


Fig. (a) Multiple parse tree

- As for the string id + id * id, more than one parse tree is created, so the given grammar is ambiguous.

Given grammar :

$S \rightarrow aSbS / bSaS / \epsilon$

Let w = abab

Using right derivation,

$S \rightarrow aSbS$		$S \rightarrow aSbS$	
$S \rightarrow aSb\epsilon$	$/\because S \rightarrow \epsilon/$	$S \rightarrow aSbaSbS$	$/\because S \rightarrow aSbS/$
$S \rightarrow aSb$		$S \rightarrow aSbaSb\epsilon$	$/\because S \rightarrow \epsilon/$
$S \rightarrow abSaSb$	$/\because S \rightarrow bSaS/$	$S \rightarrow aSba\epsilon b$	$/\because S \rightarrow \epsilon/$
$S \rightarrow ab\epsilon a\epsilon b$	$/\because S \rightarrow \epsilon/$	$S \rightarrow a\epsilon bab$	$/\because S \rightarrow \epsilon/$
		$S \rightarrow abab$	
$S \rightarrow abab$			

∴ There arises more than one parse tree for string w = abab, hence the given grammar is ambiguous.

The parse tree is as follows :

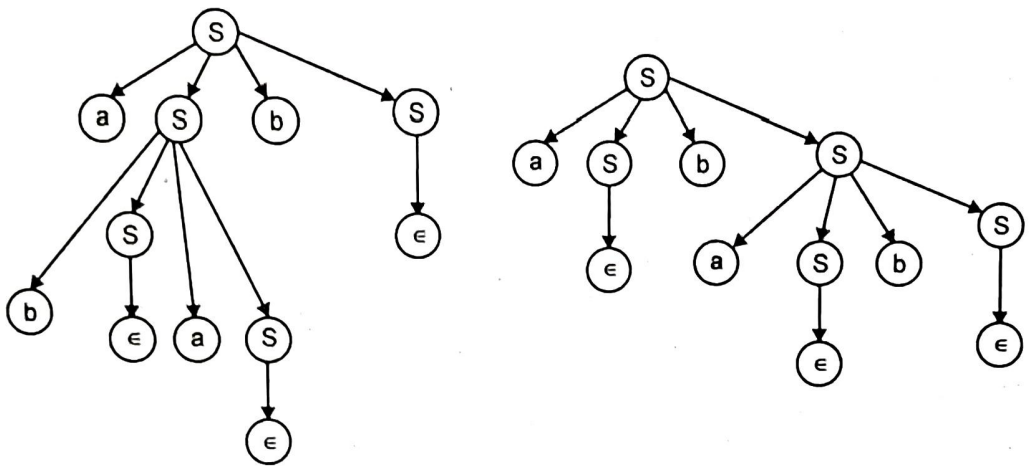


Fig.(b) Multiple parse tree for string ababa

Q.98. Given grammar

$S \rightarrow a / abSb / aAb$

$A \rightarrow bS / aAAb$

Check whether the grammar is ambiguous or not.

CT : S-14(5M)

Ans. Consider the string

w → abab.

The rules are :

$S \rightarrow a$ ..... (I)

$S \rightarrow abSb$ ..... (II)

$S \rightarrow aAb$ ..... (III)

$A \rightarrow bS$ ..... (IV)

$A \rightarrow aAAb$ ..... (V)

Using left most derivation :

$S \rightarrow ab\underline{S}b$ $\quad S \rightarrow a\underline{A}b$

$S \rightarrow abab$ $/\because S \rightarrow a/$ $\quad S \rightarrow abSbb$ $/\because A \rightarrow bS/$

$S \rightarrow abab$ $/\because S \rightarrow a/$

- Here string w = abab can be generated in more than one way using leftmost derivation, so grammar is ambiguous.

Thus derivation tree for these two approaches are :

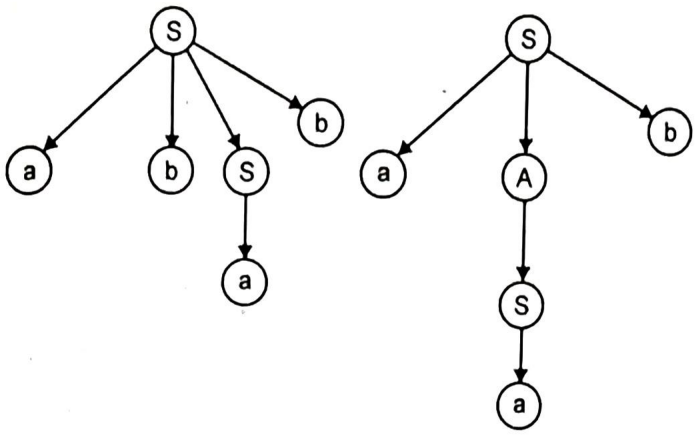


Fig. Multiple parse tree for string abab

PARSER GENERATOR

Q.99. Explain YACC in detail.

OR Write a note on YACC. **CS:W-14(3M)**

Ans. YACC (Yet Another Compiler-Compiler) :

- YACC provides a general tool for describing the input to a computer program.
- The YACC user specifies the structure of his input, together with code to be invoked as each such structure is recognized.
- YACC turns such a specification into subroutine that handles the input process ; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.
- Computer program input generally has some structure ; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts.
- An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use and often are lax about checking their inputs for validity.
- The input subroutine produced by YACC calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers.
- The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.
- YACC is written in portable C. The class of specifications accepted is a very general one, e.g., LALR(1) grammars with disambiguating rules.
- In addition to compilers for C, APL, Pascal, RATFOR, etc., YACC has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system and a Fortran debugging system.

Q.100. Explain bison as parser generator.

Ans. Bison :

- Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables.
- As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables.
- Bison is upward compatible with YACC.
- The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

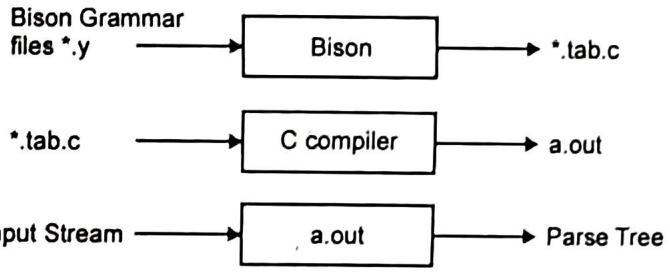


Fig. Bison parser

Steps to use Bison :

(1) Write a lexical analyzer to process input and pass tokens to the parser (calc. lex).

(2) Write the grammar specification for bison (calc.y), including grammar rules, yyparse( ) and yyerror( ).

(3) Run Bison on the grammar to produce the parser. (Makefile).

(4) Compile the code output by Bison, as well as any other source files.

(5) Link the object files to produce the finished product.

POINTS TO REMEMBER :

(1) Syntax analysis processes the string of descriptors synthesized by lexical analyzer to determine the syntactic structure of input statement. This process is known as parsing.

(2) A context free grammar is a finite set of variables each of which represents a language.

(3) Terminals are tokens of language used to form language constructs.

(4) Non-terminals are the variables that denote a set of strings.

(5) Derivation tree is the graphical representation of how the string is derived and generated from grammar.

(6) Leftmost derivation considers leftmost nonterminal first for derivation at every stage in derivation.

(7) Rightmost derivation considers rightmost non terminal first at every stage in derivation.

(8) If a grammar contains a pair of production of form $A \rightarrow A\alpha / \beta$, then grammar is a left-recursive grammar.

(9) Parser for a grammar is a process which take a string as a input and produce a parser tree as output.

(10) Parsing is of two types :

(i) Top down parsing.

(ii) Bottom up parsing.

(11) Top down parsing starts constructing parse tree from the start symbol and then tries to transform the start symbol to the input.

(12) Bottom up parsing starts constructing parse tree with leaves and proceeding towards the root.

(13) Backtracking is the process of repeatedly scanning the input which is the problem in top-down parsing.

(14) LL(1) grammar is a top-down parser in which parsing is done by scanning the input from left to right and input string is derived in leftmost order.

(15) LR parser is a bottom-up parser which scan the input from left to right and construct the rightmost derivation in reverse.

(16) LR parser encounter two types of conflicts :

(i) Shift-reduce parser conflict.

(ii) Reduce-reduce conflict.

(17) Viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of shift/reduce parser.

(18) An ambiguous grammar is one that produces more than one leftmost or rightmost derivation for some sentence.

(19) YACC is a Yet Another Compiler Constructor which has ability to automatically recover from the errors.

UNIT - III

CONTENTS

UNIVERSITY PAPER SOLUTIONS
SINCE SUMMER - 2009

Note to the students :

- The questions given below are from previous Nagpur University examination papers.
- Though these questions are from old university papers, they have been included in the new syllabus.
- Questions which are out of new syllabus are not included here.

SUMMER - 09 (CS)

No question asked in University exam.

SUMMER - 09 (CT)

Q.1. Translate the following expression :

– (a * b) * (c + d) / (a * b + c) into

(i) Quadruples

(ii) Triples

(iii) Indirect Triples. **6M**

Ans. **P.3-18, Q.21.**

WINTER - 09 (CS) & WINTER - 09 (CT)

No question asked in University exam.

SUMMER - 10 (CS)

Q.1. Explain the following concept :

(i) Semantic actions.

(ii) L-attribute definition **4M**

Ans. **P.3-4, 10, Q.4, 13.**

SUMMER - 10 (CT)

No question asked in University exam.

WINTER - 10 (CS)

Q.1. Define :

(i) Inherited attribute

(ii) Synthesized attribute

(iii) Semantic action **3M**

Ans. **P.3-4, Q.4, 6.**

WINTER - 10 (CT)

No question asked in University exam.

SUMMER - 11 (CS)

Q.1. Generate three address code for the following statement using SDTS.

A : = – B * (C + D). **3M**

Ans. **P.3-19, Q.23.**

SUMMER - 11 (CT) & WINTER - 11 (CS)

No question asked in University exam.

WINTER - 11 (CT)

Q.1. Describe various representations of three-address codes. Translate the expression :

– (a + b) * (c + d) + (a + b + c)

into Quadruples and Triples. **6M**

Ans. **P.3-19, Q.22.**

SUMMER - 12 (CS)

Q.1. Generate three address code for the following statement using SDTS.

A : = – B * (C + D) **3M**

Ans. **P.3-19, Q.23.**

SUMMER -12 (CT) To SUMMER -13 (CS)

No question asked in University exam.

SUMMER - 13 (CT)

Q.1. Write quadruple, triples and indirect triples for the following expression : **5M**

– (a + b) * (c + d) + (a + b + c)

Ans. **P.3-19, Q.22.**

WINTER - 13 (CS)

No question asked in University exam.

WINTER - 13 (CT)

Q.1. Translate the following into : **5M**

(i) TAC

(ii) Quadruple

(iii) Triple representations

– (a + b) * (c + d) + (a + b + c).

Ans. **P.3-19, Q.22.**

SUMMER - 14 (CS) & SUMMER - 14 (CT)

No question asked in University exam.

WINTER - 14 (CS)

Q.1. Explain the data structures used for representation of three address code and compare them. **6M**

Ans. **P.3-17, Q.18.**

Note : Also refer Paper Solutions at the end of the book.

SOLVED QUESTION BANK

[Sequence given as per syllabus]

INTRODUCTION

This unit develops the theme of the translation of languages guided by context free grammars we associate information with a programming language construct by attaching attributes to the grammar. Symbols representing the construct values for attributes are computed by semantic rules associated with the grammar productions.

There are notational framework for intermediate code generation that is an extension of context-free grammars. This framework is called as syntax-directed translation scheme. It allows subroutines or 'semantic actions' to be attached to the productions of context free grammar. These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.

We will see what are inherited, synthesized attribute, evaluation of attributes, L and S attributed definition, implementation of SDTS, evaluation of expression using semantic actions and finally the postfix, syntax tree and three address code used for intermediate code representation.

SPECIFICATION OF TRANSLATIONS

Q.1. Write short note on specification of translation.

Ans. Specification translation :

- It involves specifying what the construct is, as well as specifying the translating rules for construct.
- Whenever a compiler encounters that construct in a program, it will translate the construct according to the rules of translation.
- Translation does not necessarily mean generating either intermediate code or object code.
- Translation also involves adding information into the symbol table as well as performing construct specific computations.
- For example, if a construct is a declarative about the constructs type attribute into symbol table.
- If construct is an expression then its translation generates the code for evaluating the expression.
- Translation of a construct involves manipulating the values of various quantities.

When translating the declarative statement int a, b, c, the compiler need to extract the type int and add it to the symbol table records of a, b, c.

- This require compiler to keep the track of the type int as well as the pointers to the symbol records containing a, b and c.
- Syntax directed definitions use CFG to specify the syntactic structure of the construct.
- It associates a set of attributes with each grammar symbol and with each production, it associates a set of semantic rules for computing the values of the attributes of the grammar symbols appearing in that production.
- Therefore the grammar and the set of semantic rules constitute syntax directed definitions.

SYNTAX DIRECTED DEFINITION

Q.2. What is syntax-directed definitions?

Ans. Syntax directed definition :

- A syntax-directed definition 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.
- If we think of a node for the grammar symbol in a parse tree as a record with fields for holding information, then an attribute corresponds to the name of a field.

Q.3. Explain form of a Syntax-directed definition.

Ans.

- In a Syntax-directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b := f(C_1, C_2, \ldots\ldots C_k)$, where f is a function and either.

(1) b is a synthesized attribute of A and $C_1, C_2, \ldots\ldots C_k$ are attributes belonging to the grammar symbols of the production, or

(2) b is a inherited attribute of one of the grammar symbols on the right side of the production and $C_1, C_2, \ldots\ldots C_k$ are attributes belonging to the grammar symbol of the production.

- An attribute grammar is a syntax-directed definition in which the function in semantic rules cannot have side effects.

Q.4. Explain semantic action. **CS : S-10(2M), W-10(1M)**

Ans. Semantic action :

- A compiler has to do more than just recognize if a sequence of characters forms a valid sentence in the language.
- The semantic action of a parser perform useful operation.

  (1) Build an abstract parse tree.

  (2) Type checking.

  (3) Evaluation in the case of an interpreter.

  (4) Begin the translation process in the case of a compiler.
- In recursive descent parser the semantic action are mixed in with the control flow of parsing.
- In some compiler constructors such as Java CC and Yacc the semantic actions are attached to the production rules.
- In other compiler constructions sable CC for example the syntax tree automatically generated.
- Each symbol, terminal or non terminal, may have its own type of semantic value.

  exp → INT

  exp → exp PLUS exp.

  exp → exp MINUS exp.

  exp → exp MUL exp.

  exp → exp DIV exp.

  exp → MINUS exp.

SYNTAX DIRECTED TRANSLATION SCIIEME

Q.5. What is Syntax directed translation scheme? Justify the necessity for intermediate code and give some example.

Ans. Syntax directed translation scheme :

- A syntax directed translation scheme is a notational framework for intermediate code generation which is an extension of context-free grammars.
- It allows subroutines or "Semantic action" to be attached to the productions of a context-free grammar.
- These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.
- Enable the compiler designer to express the generation of code directly in terms of the syntactic of the source language.

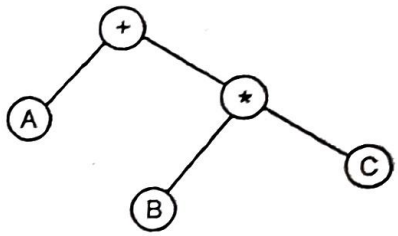
Necessity for the intermediate code. :

- Syntax directed translation scheme can be used for translation of source code to target code, which generally speeds up the compiler, the optimization of machine or assembly code is harder. Hence some form of intermediate code is generated on which optimization can be performed easily.
- The reason why efficient machine or assembly code is hard to generate is that one is immediately forced to choose a particular register to hold result difficult. Usually the intermediate code is choosen in such a way like assembly language each statement involves at most one operation, but unlike assembly code, the register in which each operation occurs is left unspecified.

Example of intermediate code :

(1) **Postfix notation :** abc* - is a postfix notation for a + b*c.

(2) **Syntax tree :** For a + b*c syntax tree is shown below.



(3) **Three address code :** The three address code for a + b*c will be

T_1 : b * c

T_2 : a + T_1 , where T1 and T2 are compiler generated temporaries.

INIIERITED AND SYNTIIESIZED ATTRIBUTES

Q.6. What are the implementations of the translations specified by syntax-directed definitions?

OR What are synthesized and inherited attributes?

OR Define synthesized and inherited attribute. **CS : W-10(2M)**

Ans.

- Attributes are associated with the grammar symbols that are the labels of the parse tree node.
- When a semantic rule is evaluated, the parser computes the value of an attribute at a parse tree node.
- To evaluate the semantic rules and carry out translation, the parse tree must be traversed and the values of the attributes at the nodes computed.
- The order in which we traverse the parse tree nodes depends on the dependencies of the attributes at the parse tree node.

- Carrying out the translation specified by the syntax-directed definition involves :

(1) Generating the parse tree for the input string w.

(2) Finding out the traversal order of the parse tree nodes by generating a dependency graph and doing a topological sort of that graph.

(3) Traversing the parse tree in the proper order and getting the semantic rules evaluated.

Synthesized and inherited attributes :

- The attributes associated with a grammar symbols are classified into two types :

(A) Synthesized attributes.

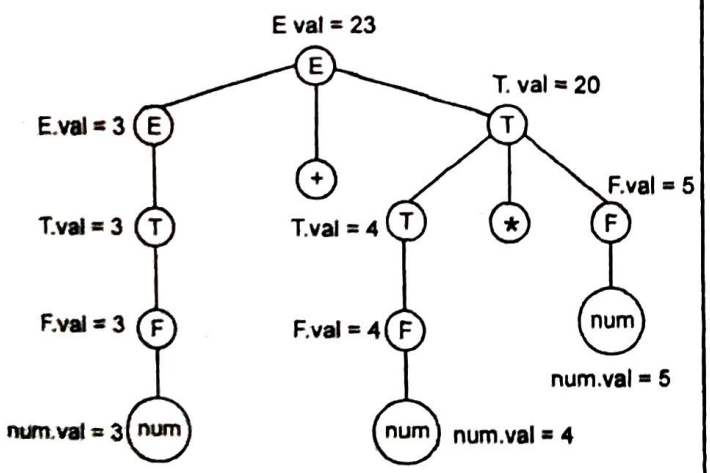
(B) Inherited attributes.

(A) Synthesized attributes :

- If value of attribute at parse tree node is determined from attribute values of the children node, then attribute is defined as synthesized attribute.
- They are evaluated during a single bottom up traversal of parse tree and are as follows :

$E \rightarrow E_1 + T$	E. val : = E_1 . val + T. val
$E \rightarrow T$	E. val : = T. val
$E \rightarrow T_1 * F$	T. val : = T_1 . val * F. val
$T \rightarrow F$	T. val : = F. val
$F \rightarrow id$	F. val : = num. lexval

- The above shown SDTS using synthesized attribute specify the translations, that are needed to be carried by expression evaluator.
- For example, parse tree for an expression 3 + 4 * 5 along with values of attribute at the nodes of parse tree is shown below.



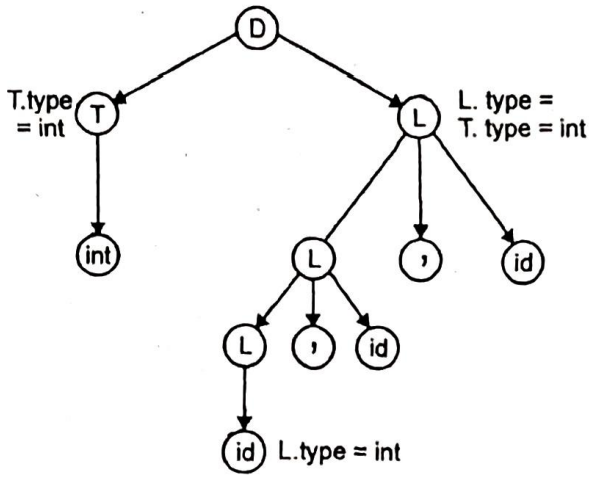
- Syntax directed definitions that use synthesized attributes only are known as "S-attributed" definitions.

(B) Inherited attributes :

- They are those initial values at a node in the parse tree which are defined in terms of attributes of the parent and / or siblings of that node.
- For example,

$D \rightarrow TL$	L. type = T. type
$T \rightarrow$ int	T. type = int
$T \rightarrow$ real	T. type = real
$L \rightarrow L_1$, id	L_1 . type = L. type
	enter (id . prt, L . type)
$L \rightarrow$ id	enter (id . prt, L . type)

- A parse tree for an input string int id_1 , id_2 , id_3 along with values of the attributes at the nodes of parse tree is as shown below.



- Inherited attributes are convenient for expressing the dependency of a programming language construct on the context in which it appears when inherited attributes are used, then the inter dependencies among the attributes at the nodes of the parse tree must be taken into account when evaluating their semantic rules.
- The interdependencies among attributes are depicted by a directed graph called a "dependency graph".
- If the semantic rule is in the form of a procedure call fun ($a_1, a_2, a_3, ..., a_k$), then it can be transformed into form b = fun ($a_1, a_2, a_3, ..., a_k$), where b is a dummy synthesized attribute.

DEPENDENCY GRAPHS

Q.7. Write short note on dependency graphs.

Ans. Dependency graph :

- If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.
- The interdependencies among the inherited and synthesized attributes at that nodes in a parse tree can be depicted by a directed graph, called a dependency graph.
- Before constructing a dependency graph for a parse tree, we put each semantic rule into the form b : = $f(c_1, c_2, ...c_k)$ by introducing a dummy synthesized attribute b for each semantic rule that consists of a procedure call.
- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.
- The algorithm for construction of dependency graph for a given parse tree is as follows :

  for each node n in the parse tree do

  for each attribute a of the grammar symbol at n do

  construct a node in the dependency graph for a :

  for each node n in the parse tree do

  for each semantic rule b : = $f(c_1, c_2, ...c_k)$ associated with the production used at n do

  for i = 1 to k do

  construct an edge from the node for ci to the node for b :
- For example, fig. (c) shows the dependency graph for the parse tree in fig. (b). nodes in the dependency graphs are marked by numbers.
- The parse tree in fig (b) is constructed from fig (a).

Production	Semantic rules
D → TL	L. in : = T. type
T → int	T. type : = integer
T → real	T. type : = real
L → L_1 , id	L_1 . in : = L. in addtype (id. Entry, L. in)
L → id	addtype (id. Entry, L. in)

Fig. (a) Syntax-directed definition with inherited attribute L. in.

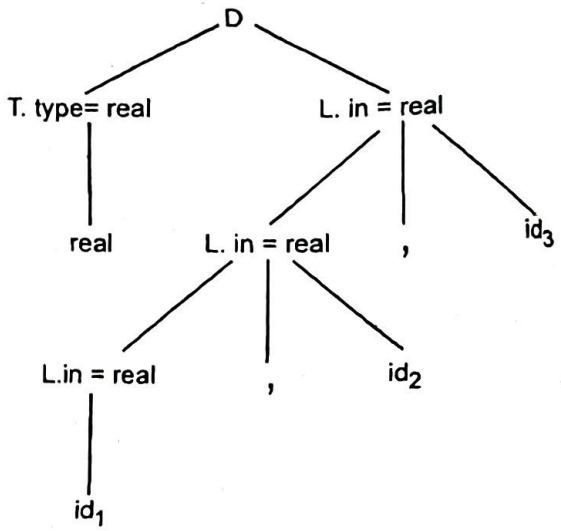


Fig. (b) Parse tree with inherited attribute in at each node labelled L.

- In fig. (c) , there is an edge to node 5 for L in from node 4 for T-type because the inherited attribute L in depends on the attribute T.type according to the semantic rule L. in : = T. type for the production D → TL.
- The two downward edges into nodes 7 and 9 arise, because L_1 in depends on L. in according to the semantic rule L_1 . in = L. in for the production L → L_1 , id. Each of the semantic rules addtype (id. entry, L. in) associated with the L-production leads to the creation of a dummy attribute.
- Nodes 6, 8 and 10 are constructed for these dummy attributes.

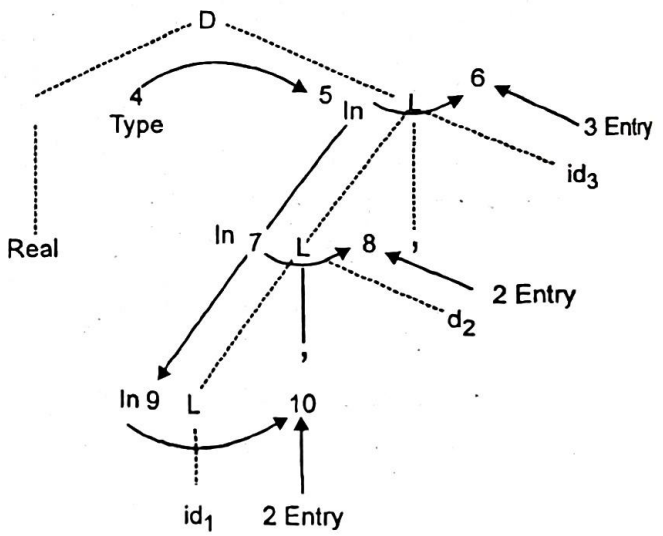


Fig. (c) Dependency graph for parse tree of fig. (b)

Q.8. Explain the term 'topological sort'.

Ans. Topological sort :

- A topological sort of a directed cyclic graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes: that is, if $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.
- That is, in the topological sort, the dependent attributes $c_1, c_2, \ldots c_k$ in a semantic rule $b := f(c_1, c_2, \ldots c_k)$ are available at a node before f is evaluated.
- For example, each of the edges in the dependency graph in figure goes from a lower-numbered node to a higher-numbered node.
- Hence a topological sort of the dependency graph is obtained by writing down the nodes in the order of their numbers.
- From this topological sort, we obtain the following program. We write a_n for the attribute associated with the node numbered n in the dependency graph.

$a_4 :=$ real;

$a_5 := a_4$;

addtype (id_3.entry, a_5);

$a_7 := a_5$

$a_9 := a_7$;

addtype (id_1 . Entry , a_9) ;

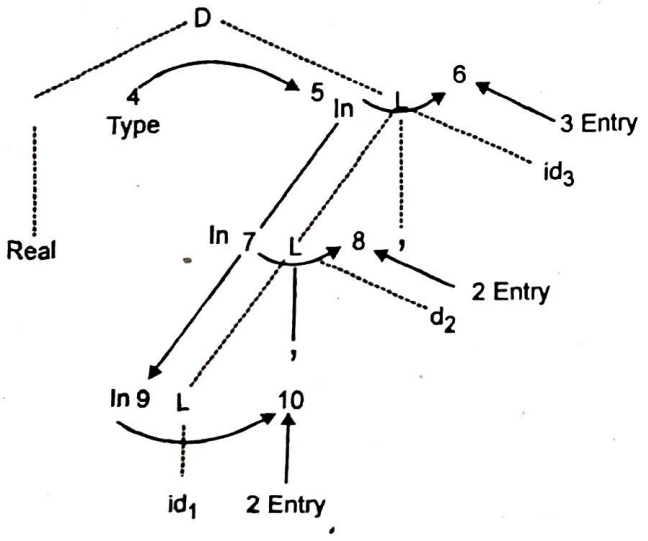


Fig. (d) Dependency graph

EVALUATION ORDER

Q.9. Write a short note on evaluation order.

Ans. Evaluation order :

- A topological sort of a directed acycle graph is any ordering m_1, $m_2, \ldots m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- If $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.
- That is, in the topological sort, the dependent attributes c_1, $c_2, \ldots c_k$ in a semantic rule $b := f(c_1, c_2, \ldots c_k)$ are available at a node before f is evaluated.
- From a topological sort of the dependency graph we obtain an evaluation order for the semantic rules.
- Evaluation of the semantic rules in this order yield the translation of the input string.
- The several methods for evaluating semantic rules are as follows :

(1) Parse-tree method :

- At compile time, these method obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input.
- These methods will fail to find an evaluation order only if the dependency graph for the particular parse tree under consideration has a cycle.

(2) Rule-based method :

- At compiler-construction time, the semantic rules associated with productions are analyzed, either by hand or by a specialized tool.
- For each production, the order in which the attributes associated with that production are evaluated is predetermined at compiler construction time.

(3) Oblivious method :

- An evaluation order is chosen without considering the semantic rule.
- For example, if translation takes place during parsing than the order of evaluation is forced by the parsing method, independent of the semantic rules.
- An oblivious evaluation order restricts the class of syntax-directed definitions that can be implemented.

- Rule based an oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.
- A syntax-directed definition is said to be circular if the dependency graph for some parse tree generated by its grammar as a cycle.

TOP DOWN AND BOTTOM UP EVALUATION

Q.10. What is top down and bottom up evaluation of attributes?

Ans.

- A syntax analyzer is a program that perform syntax analysis.
- A parser obtain a string of tokens from lexical analyzer and verifies whether or not string is a valid construct of the source language i.e. whether or not it can be generated by the grammar for the source language.
- The parser either attempts to derive the string of tokens w from the start symbol S or it attempts to reduce w to the start symbol of grammar by tracing the derivations of w in reverse.
- An attempt to derive w from the grammar's start symbol S is equivalent to an attempt to construct the top-down parse tree i.e. it starts from the root node and proceeds towards the leaves.
- Similarly an attempt to reduce w to the grammar's start symbol S is equivalent to an attempt to construct bottom up tree.

Alphabet : An alphabet is a finite set of symbols denoted by symbol Σ.

Language :

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

TOP DOWN PARSING

Q.11. What is top down parsing?

Ans. Top down parsing :

- Top down parsing attempts to find the left most derivation for an input string w, which is equivalent to constructing a parse tree for the input string w that starts from the root and creates the nodes of the parse tree in predefined order.
- The reason that top-down parsing seeks the left most derivations for an input string w and not the right most derivations is that input string w is scanned by the parser from left to right.
- Since top down parsing attempts to find the leftmost derivations for an input string w, a top down parser may require backtracking because in the attempt to obtain the left most derivation of input string w, a parser may encounter a situation in which a non-terminal A is required to be derived next and there are multiple A productions, such as $A \rightarrow \alpha_1 | \alpha_2 | .... | \alpha_n$.
- In such a situation deciding which A production to use for derivation of A is a problem.
- So, the parser will select one of the A-production to derive A and if this derivation finally leads to the derivation of w, then parser announces the successful completion of parsing.
- For example, consider top-down parser for following grammar

  $S \rightarrow aAb$

  $A \rightarrow cd / c$
- Let input string be w = acb the parser initially creates a tree consisting of a single node, labeled S and input pointer points to a, the first symbol of input string w.
- The parser then user the S-production $S \rightarrow aAb$ to expand tree shown in Fig. (a).

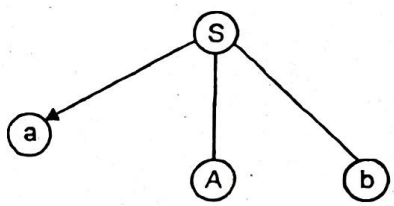


Fig. (a)

- The leftmost leaf labeled a, matches the first input symbol of w. Hence parser will now advance the input pointer to c, the second symbol of string w and consider the next leaf labeled A.
- It will then expand A using first alternative for A in order to obtain the tree as shown in Fig. (b)

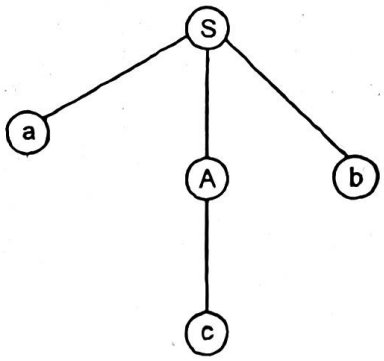


Fig.(b)

- The parser now has the match for the second input symbol, so it advances the pointer to b, the third symbol of w and compare it to the label of the next leaf.
- If the label does not match d, it reports failure and goes back (backtracks) to A as shown in Fig. (c)

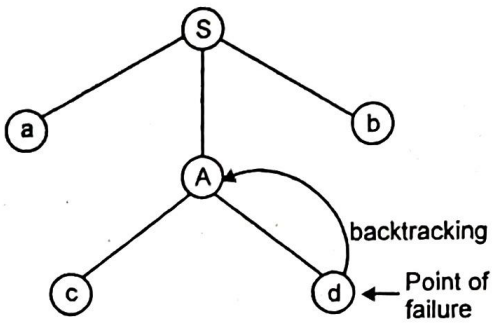


Fig.(c)

- The parser now reset the input pointer to the second input symbol - the position it had when the parser encountered A and it will try a second alternative for A in order to obtain the tree.
- If the leaf c matches the second symbol, if next leaf b matches the third symbol of w, then the parser will halt and announce the successful completion of parsing.

BOTTOM UP PARSING

Q.12. What is bottom up parsing?

Ans. Bottom up parsing :

- This parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the rightmost derivations of w in reverse.
- This is equivalent to constructing a parse tree for the input string w by starting with leaves and proceeding toward the root i.e. attempting to construct the parse tree from bottom up.
- This involves searching for the substring that matches the right side of any of the productions of the grammar.
- This substring is replaced by the lefthand side nonterminal of the production if this replacement leads to the generation of the sentential form that comes one step before in the right most derivation.
- This process of replacing the right side of the production by the left side nonterminal is called "reduction".
- Hence, reduction is nothing more than performing derivations in reverse.
- The reason why bottom up parsing tries to trace out the right most derivations of an input string w in reverse and not the left most derivation is because the parse scans the input string w from left to right, one symbol per token at a time.
- And to trace out right most derivations of an input string w in reverse, the token of w must be made available in a left to right order.
- For example, if the rightmost derivation sequence of some w is

$$S \to \alpha_1 \to \alpha_2 \to \alpha_3 \to \ldots\ldots \to \alpha_{n-1} \to w$$

then bottom up parse starts with w and searches for the occurrence of a substring of w that matches the right side of some production $A \to \beta$ such that the replacement of β by A will lead to the generation of α_{n-1}.

- The parser replaces β by A, then it searches for the occurrence of a substring at α_{n-1} that matches the right side of some productions $\beta \to r$ such that replacement of r by β will lead to the generation of α_{n-2}.
- This process continues until the entire w substring is reduced to S, or until the parser encounters an error.
- Bottom up parsing involves the selection of a substring that matches the right side of the production.
- These reduction to the nonterminal on the left side of the production represents one step along the reverse of a right most derivation.
- It leads to the generation of the previous right most derivation.
- This means that selecting a substring that matches the right side of production is not enough ; the positions of this substring in the sentential form is also important.

A handle of a right sentential form :

(1) A handle of a right sentential form r is a production $A \to \beta$ and a position of β in r.

(2) The string β will be found and replaced by A to produce the priors right sentential form in the right most derivation of r.

(3) If $S \rightarrow \alpha A\beta \rightarrow \alpha r \beta$, then $A \rightarrow r$ is a handle of $\alpha r \beta$, in the position following α.

(4) For example, consider the grammar :

$E \rightarrow E + E / E * E / id$

and the right most derivation :

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

The handles of the sentential forms occurring in the above derivations are shown in table below.

Sentential form	Handle
id + id * id	E → id at position preceding +.
E + id * id	E → id at position following +
E → E * id	E → id at position following *
E + E * id	E → E * E at position following +
E + E	E → E + E at position preceding the end marker.

- So, the bottom up parsing is an attempt to detect the handle of a right sentential form.
- This is equivalent to performing right most derivation in reverse and is called "handle pruning".
- So if the right most derivation sequence of some w is $S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \ldots \rightarrow \alpha_{n-1} \rightarrow w$, then handle pruning starts with w, the n^{th} right sentential form, the handle $A_n \rightarrow B_n$ of w is located and β_n is replaced by the left side of production $A_n \rightarrow B_n$ in order to obtain α_{n-1}.

L AND S ATTRIBUTE DEFINITIONS

Q.13. Explain the term S-attributed definition and L-attributed definition.

OR Explain L-attributed definitions S-attributed definition.

CS : S-10(2M)

Ans. S-attributed definition :

- A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition. A parse tree showing the values of attributes at each node is called an annotated parse tree.
- 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.
- Whenever, a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.
- A translator for an S-attributed definition can often be implemented with the help of an LR-parser generator.
- From an S-attributed definition, the parser generator can construct a translator that evaluates attributes as it parses the input.
- A bottom-up parser uses a stack to hold information about sub-trees that have been parsed. We can use extra fields in the parser stack to hold the values of synthesized attributes.

L - attributed definition :

- An inherited attribute is one whose value at a node in a parser tree is defined in terms of attributes at the parent and / or siblings of that node.
- Inherited attributes are convenient for expressing the dependence of a programming language, constructed on the context in which it appears.
- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \rightarrow X_1, X_2, \ldots X_n$ depends only on

(1) The attributes of the symbols $X_1, X_2, \ldots X_{j-1}$ to the left of X_j in the production, and

(2) The inherited attributes of A.

$D \rightarrow TL$	L . Type = T . Type
$T \rightarrow$ int	T . type = int
$T \rightarrow$ real	T . type = real
$L \rightarrow L_1$. id	L_1 . type = L . type ; enter (id . ptr, L . type) ;
$L \rightarrow$ id	enter (id . ptr, L . type)

- The syntax directed definition given above is an example of the L – attributed definition, because the inherited attribute L . type depends on T. type, and T is to the left of L in the production $D \rightarrow TL$.
- Similarly the inherited attribute L_1 . type depends on the inherited attribute L . type, and L is parent of L_1 in the production $L \rightarrow L_1$, id .
- Note that every S-attributed definition is L-attributed.

IMPLEMENTATION OF SDTS

Q.14. Write short note on implementation of SDTS.

Ans. Implementation of SDTS :

- A syntax-directed translation scheme is a convenient description of what we would like done.
- The output defined is independent of the kind of parser used to construct the parse tree or the kind of mechanism used to compute the translations.
- Thus, a syntax-directed translation scheme provides a method for describing an input-output mapping and that description is independent of any implementation.
- Another convenience of the approach is that it is easy to modify.
- New productions and semantic actions can often be added without disturbing the existing translations being computed.
- Having written a syntax-directed translation scheme, our next task is to convert it into a program that implements the input-output mapping described.

Mechanism to implement syntax directed translation :

- A useful requirement is that we have a bottom-up parser for the grammar. Some type of LR (1) parser would be quite adequate for most schemes.
- However, we must augment the parser with some mechanism for computing the translations.
- To compute the translation at a node A associated with a production $A \rightarrow XYZ$, we need only the values of the translations associated with nodes labeled X, Y and Z.
- These nodes will be roots of subtrees in the forest representing the partially constructed parse tree.
- The nodes X, Y and Z will become children of node A after reduction by $A \rightarrow XYZ$. Once the reduction has occurred we do not need the translations of X, Y and Z any longer.
- One way to implement a syntax-directed translator is to use extra fields in the parser stack entries corresponding to the grammar symbols.
- These extra fields hold the values of the corresponding translations.
- Let us suppose the stack is implemented by a pair of arrays STATE and VAL, as shown in Fig. (a). Each STATE entry is a pointer (or index) to the LR (1) parsing table.
- If the i-th STATE symbol is E, then VAL [i] will hold the value of the translation E. VAL associated with the parse tree node corresponding to this E.
- TOP is a pointer to the current top of the stack. We assume semantic routines are executed just before each reduction.
- Before XYZ is reduced to A, the value of the translation of Z is in VAL [TOP], that of Y in VAL [TOP + 1] and that of X in VAL [TOP + 2].
- After the reduction, TOP is incremented by 2 and the value of A. VAL appears in VAL [TOP].

Example :

- Consider an example of how a syntax-directed translation scheme can be used to specify a "desk calculator" program and how that translation scheme can be implemented by a bottom-up parser that invokes program fragment to compute the semantic actions.
- The desk calculator is to evaluate arithmetic expressions involving integer operands and the operators + and *. We assume that an input expression is terminated by $. The output is to be the numerical value of the input expression. For example, for the input expression 23 * 5 + 4$, the program is to produce the value 119.
- We use the nonterminals S (for complete sentence), E (for expression) and I (for integer). The productions are

  $S \rightarrow E\$$

  $S \rightarrow E + E$

  $S \rightarrow E * E$

  $S \rightarrow (E)$

  $S \rightarrow I$

S → I digit

I → digit

- The terminals are $, +, *, parentheses and digit, which we assume stands for any of the digits 0, 1, ...9.

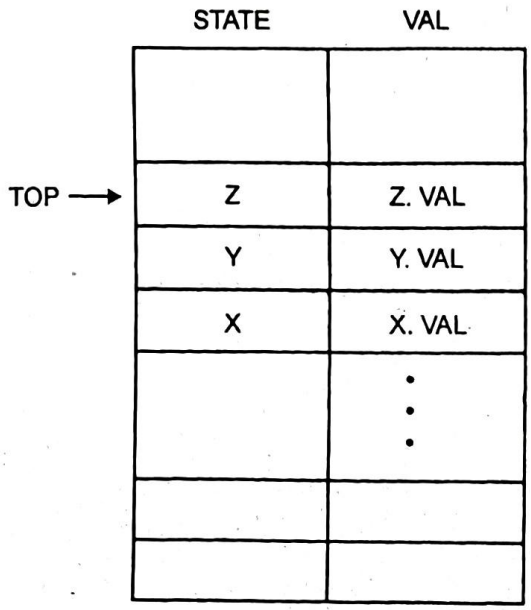


Fig. (a) Stack before reduction

- We must now add the semantic actions to the productions. With each of the nonterminal E and I we associate one integer-valued translation, called E. VAL and I. VAL, respectively, which denotes the numerical value of the expression or integer represented by a node of the parse tree labeled E or I.
- With the terminal digit we associate the translation LEXVAL, which we assume is the second component of the pair (digit, LEXVAL) returned by the lexical analyzer when a token of type digit is found.
- One possible set of semantic actions for the desk calculator grammar is shown in table (1). Using this syntax-directed translation scheme, the input 23 * 5 + 4$ would have the parse tree and translations shown in Fig. (b).
- To implement this syntax-directed translation scheme we need to construct a lexical analyzer and a bottom-up parser and we must make the parser invoke a program fragment to implement a semantic action just before making each reduction.
- A compiler-complier would tie the parser and the semantic action program fragments together, producing one module.

Table (1) Syntax-directed translation scheme for desk calculator

Sr. No.	Production	Semantic action
(1)	$S \rightarrow E\$$	{print E. VAL}
(2)	$E \rightarrow E^{(1)} + E^{(2)}$	{E. VAL : = E.$^{(1)}$ VAL + E.$^{(2)}$ VAL }.
(3)	$E \rightarrow E^{(1)} * E^{(2)}$	{E. VAL : = E.$^{(1)}$ VAL * E.$^{(2)}$ VAL }.
(4)	$E \rightarrow (E^{(1)})$	{E. VAL : = E.$^{(1)}$ VAL}.
(5)	$E \rightarrow I$	{E. VAL : = I.VAL}.
(6)	$I \rightarrow I^{(1)}$ digit	{E. VAL : = 10 * I.$^{(1)}$ VAL + LEXVAL}.
(7)	I → digit	{I. VAL : = LEXVAL}.

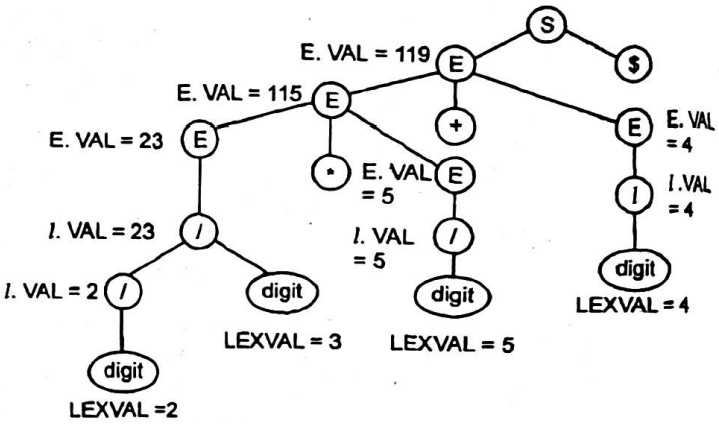


Fig. (b) Parse tree with translation

- The construction of the lexical analyzer is easy. We assume the lexical analyzer scans the input and partitions it into tokens, which here are just the terminals of the grammar.
- Every time the parser calls for a shift and needs the next token, the lexical analyzer skips over blanks to find the next nonblank input symbol, which will be one of the terminals.
- In the case that one of 0, 1, ... 9 is the next nonblank, the lexical analyzer returns the token digit and a value, denoted LEXVAL, which will be the numerical value of the digit found.
- To implement the semantic actions we cause the parser to execute the program fragments of table (2) just before making the appropriate reduction.

Table (2) Implementation of desk calculator

Sr. No.	Production	Semantic action
(1)	S → E$	print VAL [TOP]
(2)	E → E + E	VAL [TOP] : = VAL [TOP] + VAL [TOP – 2]
(3)	E → E * E	VAL [TOP] : = VAL [TOP] * VAL [TOP – 2]
(4)	E → (E)	VAL [TOP] : = VAL [TOP – 1]
(5)	E → I	none
(6)	I → I digit	VAL [TOP] : = 10 VAL [TOP] * VAL [TOP] + LEXVAL
(7)	I → digit	VAL [TOP] : = LEXVAL

- Fig. e shows the sequence of moves made by the parser on input 23 * 5 + 4$.
- The contents of the STATE and VAL fields of the parsing stack are shown after each move.
- We have again taken the liberty of replacing stack states by their corresponding grammar symbols.
- We take the further liberty of using, instead of digit on the stack, its associated LEXVAL.

Table (3) Sequence of moves

Sr. No.	Input	STATE	VAL	Production used
(1)	23 * 5 + 4$	-	-	
(2)	3 * 5 + 4$	2	-	
(3)	3 * 5 + 4$	I	2	I → digit
(4)	* 5 + 4$	I3	2_	
(5)	* 5 + 4$	I	(23)	I → I digit
(6)	* 5 + 4$	E	(23)	E → I
(7)	5 + 4$	E*	(23)_	
(8)	+ 4$	E * 5	(23)_ _	
(9)	+ 4$	E * I	(23)_ 5	I → digit
(10)	+ 4$	E * E	(23)_ 5	E → I
(11)	+ 4$	E	(115)	E → E * E
(12)	4$	E +	(115)_	
(13)	$	E + 4	(115)_ _	
(14)	$	E + I	(115)_ 4	I → digit
(15)	$	E + E	(115)_ 4	E → I
(16)	$	E	(119)	E → E + E
(17)	-	E$	(119)_	
(18)	-	S	-	S → E$

- Consider the sequence of events on seeing the input symbol 2. In the first move the parser shifts the state corresponding to the token digit (Whose LEXVAL is 2) onto the stack (The state is represented by LEXVAL which is 2).
- On the second move the parser reduces by the production I → digit and then invokes the semantic action I. VAL = LEXVAL. The program fragment implementing this semantic action causes the VAL of the stack entry for digit to acquire the value 2.
- Note that after each reduction and semantic action the top of the VAL stack contains the value of translation associated with the left side of the reducing production.

EVALUATION OF EXPRESSIONS USING SEMANTIC ACTIONS

Q.15. Write in brief about evaluation of expression using semantic actions.

Ans. Semantic actions :

- A syntax-directed translation scheme is merely a context-free grammar in which a program fragment called an output action (or sometimes a semantic action or semantic rule) is associated with production.
- For example, suppose output action α is associated with production $A \rightarrow XYZ$. The action α is executed whenever the syntax analyzer recognizes in its input a substring w which has a derivation of the form $A \Rightarrow XYZ \overset{*}{\Rightarrow} w$.
- In a bottom-up parser, the action is taken when XYZ is reduced to A.

- In a top-down parser the action is taken when A, X, Y or Z is expanded, whichever is appropriate.
- The output action may involve the computation of values for variables belonging to the compiler, the generation of intermediate code, the printing of an error diagnostic, or the placement of some value in a table.
- For example, the values computed by action α quite frequently are associated with the parse tree nodes corresponding to the instance of A to which α is reduced.
- A value associated with a grammar symbol is called a translation of that symbol. The translation may be a structure consisting of fields of various types. The rules for computing the value of a translation can be as involved as we wish.
- We shall usually denote the translation fields of a grammar symbol X with names such as X. VAL, X. TRUE, and so forth.
- If we have a production with several instances of the same symbol on the right, we shall distinguish the symbols with superscripts. For example, suppose we have the production and semantic action

$E \rightarrow E^{(1)} + E^{(2)}$

{E. VAL : = $E^{(1)}$. VAL + $E^{(2)}$. VAL}

- The semantic action is enclosed in braces and it appears after the production.
- Here the semantic action is a formula which states that the translation E.VAL associated with the E on the left side of the production is determined by adding together the translations associated with the E's on the right side of the production.
- Note that the terminal symbol + in the productions is "translated" into its usual meaning by the semantic rule.
- This translation is suitable not for a compiler, but for a "desk calculator" program that actually evaluates expressions rather than generating code for them.
- In most compilers we need an action that generates code to perform the addition.
- It defines the value of the translation of the nonterminal on the left side of the production as a function of the translations of the nonterminals in the right side.
- Such a translation is called a synthesized translation.

Consider the following production and action

$A \rightarrow XYZ$ {Y.VAL : = 2 * A.VAL}

- Here the translation of a nonterminal on the right side of the production is defined in terms of a translation of the nonterminal on the left. Such a translation is called an inherited translation.
- Consider the following syntax-directed translation scheme suitable for a "desk calculator" program, in which E.VAL is an integer-valued translation.

Production	Semantic action
$S \rightarrow E^{(1)} + E^{(2)}$	{E. VAL : = E. $^{(1)}$ VAL + E. $^{(2)}$ VAL }.
E → digit	{E. VAL : = digit }.

- Here digit stands for any digit between 0 and 9.
- The values of the translations are determined by constructing a parse tree for an input string and then computing the values the translations have at each node.
- For example, suppose we have the input string 1 + 2 + 3. A parse tree for this string is shown in Fig. (a).

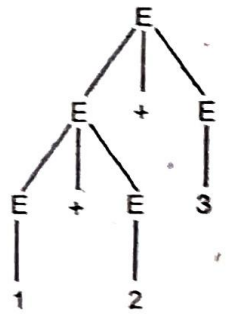


Fig.(a) Parse tree for expression 1 + 2 + 3.

- Consider the bottom leftmost E. This node corresponds to a use of the production E → 1.
- The corresponding semantic actions sets E.VAL = 1.
- Thus we can associate the value 1 with the translation E.VAL at the bottom leftmost E.
- Similarly, we can associate the value 2 with the translation E.VAL at the right sibling of this node.

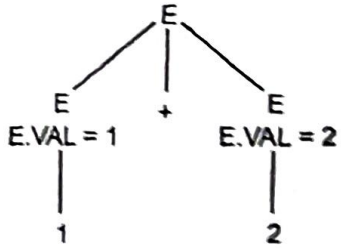


Fig.(b) Subtree with previously computed translations.

- Now consider the subtree shown in Fig. (b). The value of E.VAL at the root of this subtree is 3, which we calculate using the semantic rule

  $E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$

- In applying this rule we substitute the value of E.VAL of the bottom left most E for $E^{(1)}.VAL$ and the value of E.VAL at its right sibling E for $E^{(2)}.VAL$.

- Continuing in this manner we derive the values shown in Fig. (c) for the translations at each node of the complete parse tree.

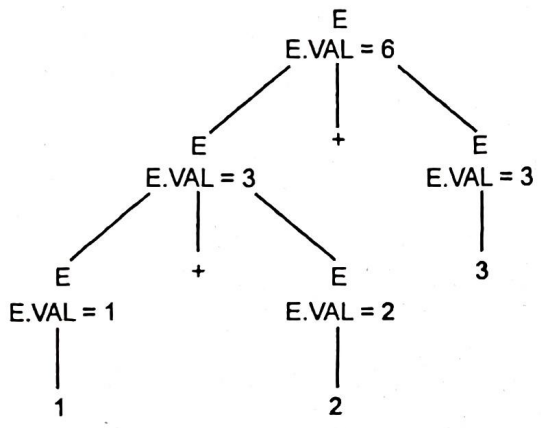


Fig. (c) Complete parse tree

- We see from this example that whenever we apply the rule

  $E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$

  the values of $E^{(1)}.VAL$ and $E^{(2)}.VAL$ have been computed by a previous application of this rule or the rule E.VAL : = digit. That is to say, if we have a translation A.VAL, then the formulas for all productions with an A on the left side must produce a value that can be used for every occurrence of A. VAL in a semantic rule associated with a production having A on the right.

INTERMEDIATE CODE REPRESENTATION (POSTFIX, SYNTAX TREE, TAC)

Q.16. What are different intermediate forms of source program used in compilers? Discuss the relative advantages and disadvantages.

OR Write short note on intermediate code.

Ans. Intermediate code :

- Many compilers translate the source code into a language which is intermediate in complexity between programming language (high level) and machine code which is called as intermediate code.
- The reason for this is, if source is directly translated into machine code or assembly code, generation of optimal or relatively good code is difficult task.
- Four kinds of intermediate code are often used in compilers, they are:

  (1) Postfix notation

  (2) Syntax trees

  (3) Quadruples

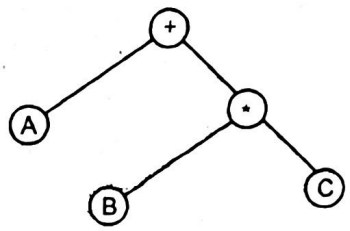
  (4) Triples

(1) Postfix notation :

- In this notation, the operator is placed at the right end in the expression example ab+. If e_1 and e_2 are any two postfix expressions and O is a binary operator, the result of applying O to the values denoted by e_1 and e_2 is indicated in the postfix notation by e_1 e_2 O.
- No parentheses are needed, because the position and priority of the operations permit only one way to decode a postfix expression.
- If O is k operators and $e_1, e_2, e_3, \ldots\ldots e_k$ are postfix expressions, then the results of applying O to these expressions, is denoted as $e_1e_2e_3\ldots e_k$ O.

(2) Syntax trees :

- This is a representation of source program, especially in optimizing compilers when the intermediate code needs to be extensively restructured.
- A syntax tree is variant of parse tree in which each leaf represents an operand and each interior node an operator, e.g., a syntax tree A+B*C is shown below.



(3) **Three-Address Code (Quadruples and Triples) :**

- This intermediate code is preferred in many compilers, especially those doing extensive code optimization, because it allows convenient rearrangement of code.
- It consists of sequence of statements, typically of the general form A: = B Op C, where A, B, C are either programmer defined names, constants or compiler generated temporary names; Op stands for operator.
- Since each statement usually contains three addresses two for operands and one for results, such is called as three address code.
- In actual compiler, these statements can be implemented in one of the following ways :

(i) **Quadruple :**

- It is a list of records with each record consisting of four fields; this representation is known as quadruples. The fields can be labelled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The contents of the rest of the fields are normally pointers to the symbol table entries of names.
- A typical quadruple representation is shown below :

	OP	ARG1	ARG2	RESULT
(0)	+	A	B	T1
(1)				
(2)				

(ii) **Triples :**

- This is a list of records, each consisting of only three fields, which is labelled as OP, ARG1, ARG2 respectively.
- The contents to the records of list itself, because RESULT field is absent.
- Typical triple representation is shown below :

	OP	ARG1	ARG2
(0)	+	A	B
(1)	*	(0)	C

(iii) **Indirect triple :**

In this, a list pointer to the three address statement represented as triple maintained, as shown below.

	LIST	OP	ARG1	ARG2
(0)	(14)	+	A	B
(1)	(15)	*	(14)	C

Comparison :

- The postfix notation is useful if the language is mostly expressive.
- The problem with this notation is in handling flow control.
- One of the solutions is to introduce labels and conditional and unconditional jumps into postfix code.
- The postfix code can then stored in single dimensional array (in which each being either opened).
- Operands are represented by pointer to symbol table and operators by integer codes. Therefore, the table will be just an index of the array.
- Using quadruple, the location for each temporary can be immediately accessed via symbol table. In triple we have no idea unless we scan code.
- In optimizing compiler quadruple permits convenient movement of statement around, whereas in Triples the movement of a statement that defines temporarily value requires us to change all pointer to that statement in ARG1 and ARG2 fields.
- Indirect triples present no such problems because a separate list of pointer into the triple structure is maintained, hence to move the statements recording of this list is required to be done, and no change in the triple structure.
- Hence the utility of the indirect triples is almost same that of quadruples.

Q.17. Explain the term 'Three-address code'.

Ans. Three-address code :

- Syntax trees and postfix notations are of two kinds of intermediate representations.
- Three-address code is a sequence of statements of the general form

  x : = y op z

  where x, y and z are names, constants or complier-generated temporaries; op stands for an operator, such as fixed or floating point arithmetic operator, or a logical operator on Boolean valued data.

- Note that there should be only one operator on the right side of a statement.
- Thus a same language expression like x + y + z might be treated into a sequence

  $t_1 := y * z$

  $t_2 := x + t_1$
- The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Q.18. Explain the data structures used for representation of three address code and compare them. **CS : W-14(6M)**

Ans. Three address codes can be represented by using one of the following data structures :

(1) Quadruple :

- This is a list of record, with each record consisting of four fields. The fields can be labeled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The representation is shown below.

	OP	ARG1	ARG2	RESULT
(0)	+	A	B	T1
(1)				

(2) Triples :

- This is a list of records, each consisting of only three fields which can be tabled as OP, ARG1, ARG2 respectively.
- The contents of ARG1 and ARG2 fields are either pointer to the symbol table entries or pointer to the records of list itself, because RESULT fields are absent.
- Typical triple representation is shown below.

	OP	ARG1	ARG2
(0)	+	A	B
(1)	*	(0)	C

(3) Indirect triple :

In this, a list of pointer three address statements represented as triples is maintained, as shown below.

	LIST		OP	ARG1	ARG2
(0)	(14)	(14)	+	A	B
(1)	(15)	(15)	*	(14)	C

Comparison :

- Using quadruple, the location for each temporary can be immediates accessed via symbol table.
- In triple we have no idea unless we scan code in optimizing compiler.
- Quadruple permits convenient movement of statements around, whereas in Triples. The movement of a statement in ARG1 Indirect Triple and Quadruples require about same amount of space Indirect Triple can save some space, compared to Quadruples if the same temporary value is used more than once.

Q.19. What are the different types of three-address statements?

Ans. The common three-address statements which we have used are as follows :

(1) Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.

(2) Assignment instructions of the form x : = op y, where op is a unary operation.

(3) Copy statements of the form x : = y, where the value of y is assigned to x.

(4) The unconditional jump goto L. The three-address statement with label L is the next to be executed.

(5) Conditional jumps such as if x relop y goto y.

(6) Param x and call p, n for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of the three-address statements :

param x_1

param x_2

.....

param x_n

call p, n

generated as part of a call of the procedure p $(x_1, x_2, ...x_n)$

(7) Indexed assignments of the form x ; y [i] and x [i] = y.

(8) Address and pointer assignments of the form x : = & y, x : = * y and * x : =y.

Q.20. Give the scheme to produce three-address code for assignments.

Ans.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id : E$	S. code : = E. code || gen (id. Place ': =' E. place)
$E \rightarrow E_1 + E_2$	E. place : = newtemp; E.code : = E_1 .code || E_2 . code || Gen(E.place ': =' E_1 . Place '+' E_2 . Place)
$E \rightarrow E_1 * E_2$	E.place : = newtemp; E.code : = E_1 .code || E_2 . code || Gen(E.place : =' E_1 place '*' E_2 place)
$E \rightarrow -E_1$	E. place : = newtemp; E.code:= E_1 code || Gen (E.place ': =' 'Uminus' E_1 place).
$E \rightarrow (E_1)$	E. place : = E_1 . place; E.code := E_1 . code
$E \rightarrow id$	E.place : = id. place; E. code : = ''

Now, the scheme to produce three-address code for assignment is

$S \rightarrow id : "E$ {p : = lookup (id. name);

if p ≠ nil, then

emit (p ': = ' E. place)

else error}

$E \rightarrow E_1 + E_2$ {E. place : = newtemp;

emit (E. place ': =' E_1 place '+' E_2 . place)}

$E \rightarrow E_1 * E_2$ {E.place : = newtemp;

emit (E.place ': = ' E_1 . Place ': = ' '*' E_2 . place)}

$E \rightarrow -E_1$ {E.place : = newtemp;

emit (E.place ': = ' 'Uminus ' E_1 . place)}

$E \rightarrow (E_1)$ {E. place : = E_1 . place}

$E \rightarrow id$ {p : = lookup (id.name);

if p ≠ nil, then

E. place : = p

else error}

Q.21. Translate following expression

– (a * b) * (c + d) / (a * b + c) into

(i) Quadruples

(ii) Triples

(iii) Indirect triple CT : S-09(6M)

Ans.

(i) Quadruple :

Sr. No.	Operator	operand 1	operand 2	result
(1)	*	a	b	t_1
(2)	uniminus	t_1		t_2
(3)	+	c	d	t_3
(4)	*	t_2	t_3	t_4
(5)	+	t_1	c	t_5
(6)	/	t_4	t_5	t_6

(ii) Triples :

Sr. No.	Operator	operand 1	operand 2
(1)	*	a	b
(2)	uniminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	(1)	c
(6)	+	(4)	(5)

(iii) Indirect triples :

Sr. No.	Statement	Operator	operand 1	operand 2
(1)	(14)	+	a	b
(2)	15	–	(14)	-
(3)	16	+	c	d
(4)	17	*	15	16
(5)	(18)	+	14	c
(6)	19	+	17	18

Q.22. Translate the following expression :

– (a + b) * (c + d) + (a + b + c) into

(i) Quadruple

(ii) Triples

(iii) Indirect triples.

OR Describe various TAC and translate the expression

– (a + b) * (c + d) + (a + b + c) into quadruple and triples.

CT : W-11(6M), S-13,W-13(5M)

Ans.

(i) Quadruple :

Sr. No.	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t2
(2)	–	t1		t2
(3)	+	c	d	t3
(4)	*	t2	t3	t4
(5)	+	t1	c	t5
(6)	+	t4	t5	t6

(ii) Triples :

Sr. No.	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	–	1)	
(3)	+	c	d
(4)	*	2)	3)
(5)	+	1)	c
(6)	+	4)	5)

(iii) Indirect triples :

Sr. No.	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	–	1)	
(3)	+	c	d
(4)	*	2)	3)
(5)	+	1)	c
(6)	+	4)	5)

Q.23. Generate TAC for following statement using SDTS.

A : = – B * (C + D). **CS : S-11,12(3M)**

Ans. The three address code will be

$t_1 = -B_1$

$t_2 = C + D$

$t_3 = t_1 * t_2$

$a = t_3$

POINTS TO REMEMBER :

(1) The attributes associated with a grammar symbol are classified into two categories :

(i) Synthesized attributes.

(ii) Inherited attributes.

(2) An attribute is said to be synthesized if its value at a parse tree node is determined by the attribute values at the child nodes.

(3) Inherited attributes are those whose initial value at a node in the parse tree is defined in terms of the attributes of the parsing or siblings of that node.

(4) While translating a source program into a functionally equivalent object code representation, a parser may first generate an intermediate representation that is called as intermediate code generation.

(5) The intermediate representations are as follows :

(i) Postfix notation.

(ii) Syntax tree.

(iii) Three-address code.

(6) Syntax tree is condensed form of parse tree the operator and keyword nodes of the parse tree are moved to their parent and a chain of single productions is replaced by single link.

(7) Three address code is a sequence of statements of the form $x = y$ op z since a statement involves no more than three references, it is called three address code.

(8) Three address code can be represented in following ways :

(i) Quadruple representation.

(ii) Triple representation.

(iii) Indirect triple representation.

(9) In postfix notation, the operator follows the operand.