Q.2. Compute LL (1) parsing table for the following grammar : **7M**

$S \rightarrow aAB / bA / \epsilon$

$A \rightarrow aAb / \epsilon$

$B \rightarrow bB / c$

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. **7M**

$E \rightarrow E + T / T$

$T \rightarrow TF / F$

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

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

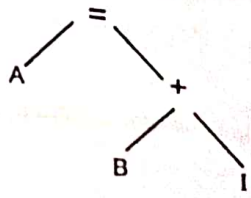
Q.1. What is syntax analysis or parsing? Give example.

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.

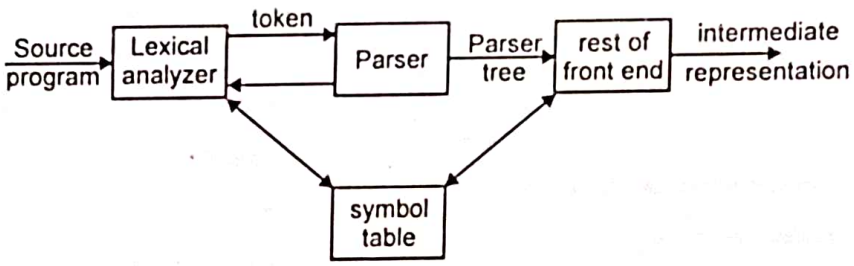


Fig. Position of parser in compiler model.

- 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.

(3) **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 :

$E \rightarrow E * E$

$E \rightarrow E * id$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

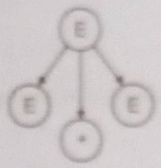
$E \rightarrow id + id * id$

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

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

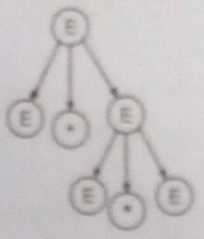
Parse tree for rightmost derivation of string id + id * id :

Step 1 : $E \rightarrow E+E$



Step 2 : $E \rightarrow E+E$

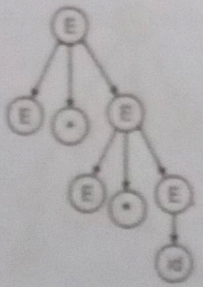
$E \rightarrow E+E*E$



Step 3 : $E \rightarrow E+E$

$E \rightarrow E+E*E$

$E \rightarrow E+E*id$

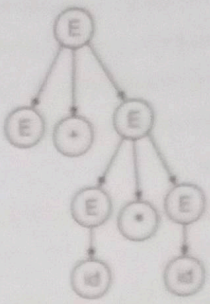


Step 4 : $E \rightarrow E+E$

$E \rightarrow E+E*E$

$E \rightarrow E+E*id$

$E \rightarrow E+id*id$



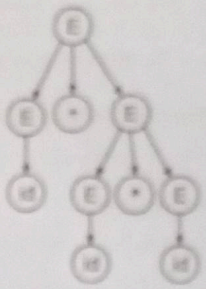
Step 5 : $E \rightarrow E+E$

$E \rightarrow E+E*E$

$E \rightarrow E+E*id$

$E \rightarrow E+id*id$

$E \rightarrow id+id*id$



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

(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

$E \rightarrow E * E$

$E \rightarrow E * id$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

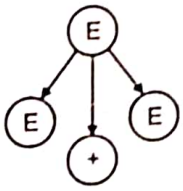
$E \rightarrow id + id * id$

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

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

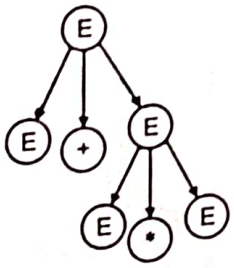
Parse tree for rightmost derivation of string id + id * id :

Step 1 : $E \rightarrow E + E$



Step 2 : $E \rightarrow E + E$

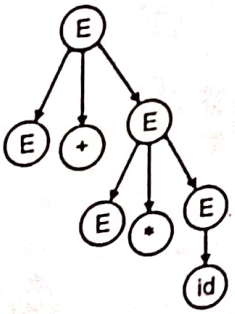
$E \rightarrow E + E * E$



Step 3 : $E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

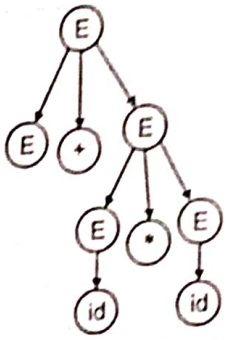


Step 4 : $E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$



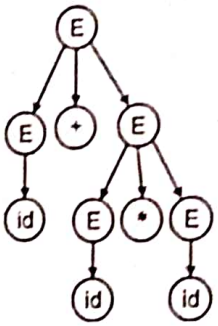
Step 5 : $E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$



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

$N_{old} = \phi$

$N_{new} = \phi$

for every production of the form $A \rightarrow \epsilon$

do

$N_{new} = N_{new} \cup \{A\}$

while $(N_{old} \neq N_{new})$ do

begin

$temp = V - N_{new}$

$N_{old} = N_{new}$

For every A in temp do

for every A-production of the form $A \rightarrow X_1 X_2 \ldots\ldots X_n$ 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 $L(G_1) = L(G) - \{\in\}$
- To eliminate ∈-productions from a grammar we use the following technique.
- If $A \rightarrow \in$, 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 → aA

  A → b / ∈

- To eliminate $A \rightarrow \in$ form the above grammar, we replace A on the right side of the production S → aA, to obtain a non ∈-productions S → 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 → aA / a

  A → b

Q.13. Eliminate ∈ - production from the grammar below :

S → ABAC

A → aA / ∈

B → bB / ∈

C → c

Ans.

- To eliminate A→∈ 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 → ABAC

  A → aA

- Replace each occurrence of A in each of these productions in order to obtain the non ∈-productions, to be added to the grammar.
- The list of these productions is :

  S → BAC / ABC / BC

  A → a
- Add these productions to the grammar and eliminate A → ∈ from the grammar. This gives us the following grammar :

  S → ABAC / BAC / ABC / BC

  A → aA / a

  B → bB / ∈

  C → c
- To eliminate B → ∈ from the grammar, the non ∈-productions, to be added are obtained as follows :

  The productions containing B on the right-hand side are :

  S → ABAC / BAC / ABC / BC

  B → bB
- Replace each occurrence of B in these productions to order the non ∈-productions, to be added to the grammar. The list of these productions is :

  S → AAC

  S → AC

  S → C

  B → b
- Add these productions to the grammar and eliminate A → ∈ from the grammar in order to obtain the following :

  S → ABAC / BAC / ABC / BC / AAC / AC / C

  A → aA / a

  B → bB / b

  C → c

Q.14. What is unit production? Give algorithm for eliminating unit production.

Ans. Eliminating unit production :

- A production of the form A → 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 → B in the grammar do

  {

  select a unit production A → B, such that there exist at least one non unit production

  B → a

  for every non unit production B → α do

  add production A → α to the grammar

  eliminate A → B from the grammar

  }

Q.15. Eliminate all the unit productions from the grammar :

S → AB

A → a

B → C / b

C → D

D → E

E → a

Ans. The given grammar contains productions

B → C

C → D

D → E

which are the unit productions.

- To eliminate these productions from the given grammar, we first select a unit production B → C.
- But since no nonunit C-production exist in the grammar, we then select C → D.
- But since no nonunit D-productions exist in the grammar, we next select D → E. There does exist a nonunit E-production : E → a. Hence we add D → a, to the grammar and eliminate D → E
- But since B → C and C → D are still there, we once again select a unit production B → C. Since no nonunit C-production exists in the grammar we select C → D.
- Now there exist a nonunit production D → a in the grammar. Hence we add C → a to the grammar and eliminate C → 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 $B \rightarrow \alpha B / \epsilon$.
- 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 $B \rightarrow Bb \mid c$. 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 :

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \epsilon$

$D \rightarrow EF$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$

- But since B → C is still there in the grammar, we once again select a unit production B → C.
- Now there exist a new unit production C → a in the grammar, so we add B → a to the grammar and eliminate B → C.
- Now no unit productions exist in the grammar. Therefore, the grammar that we get that does not contain unit productions is

S → AB

A → a

B → a / b

C → a

D → a

E → 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 → AB

A → a

B → 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 → aBDh

B → Bb / c

D → EF

E → g / ϵ

F → f / ϵ

- The grammar is left recursive because it contains a pair of productions B → Bb | c. To eliminate the left recursion from the grammar, replace this pair of productions with the following productions :

B → cC

C → bC / ϵ

- Therefore, the grammar that we get after the elimination of left recursion is :

S → aBDh

B → cC

C → bC / ϵ

D → EF

E → g / ϵ

F → f / ϵ

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)

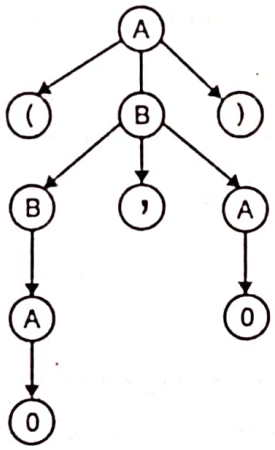
A → (B, A) *(∵ B → B, A)*

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))*

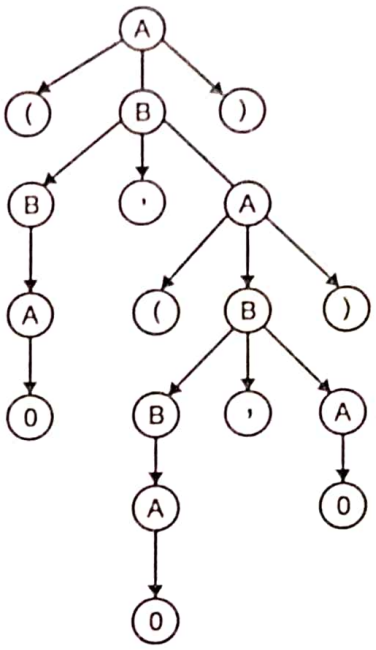
A → (0, (B, A)) *(∵ B → (B, A))*

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 α / β)

Left recursion can be eliminated by writing production of the form,

A → β A′

A′ → α A′ / ϵ

Using this rule, the given production can be replaced by,

B → AB′

B′ → , AB′ / ϵ

∴ The complete grammar after elimination of left recursion is :

A → (B)/0

B → AB′

B′ → , AB′ / ϵ

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 / ϵ **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

$A_j \rightarrow \delta_1\gamma / \delta_2\gamma \ldots\ldots / \delta_k\gamma$

where $A_j \rightarrow \delta_1 / \delta_2 \ldots\ldots / \delta_k$ 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

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 \text{ ----- } | \alpha\beta_n | \gamma$

where

γ represents all alternatives that do not begin with α by

$A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 | \text{-----} | \beta_n$

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 $S \rightarrow aAb$ to expand the tree as shown in fig. (a).

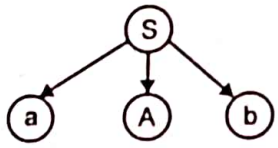


Fig.(a)

- 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).

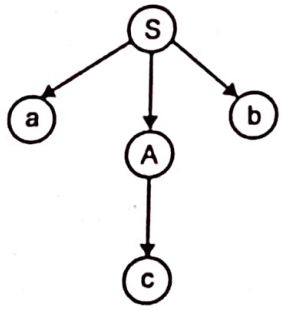


Fig.(b)

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 $S \rightarrow aAb$ to expand the tree as shown in Fig. (a).

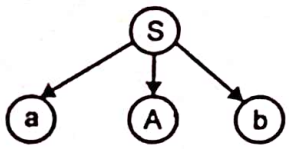


Fig.(a)

- The above grammar $S \rightarrow aAb$ 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 :

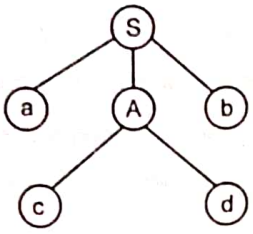


Fig.(b)

- 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.

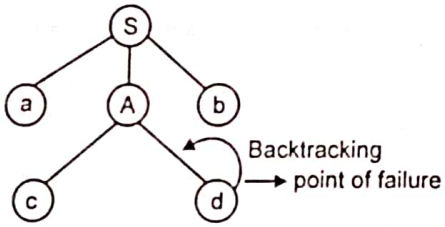


Fig.(c)

- 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.

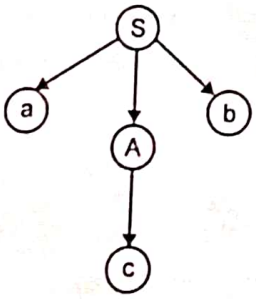


Fig. (d)

- 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(MD)

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,

$A \rightarrow A \propto / \beta$

Introduces a new non terminal A' and rewrite rule as

$A \rightarrow \beta A'$

$A' \rightarrow \in | \propto A'$

Example :

$E \rightarrow E + T / T$

So after elimination of left recursion it would be as

$E \rightarrow TE'$

$E' \rightarrow \in / + TE'$

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 fo the non-terminal E, Q and M. (E is the start symbol).

CT : S-12|10|M

Ans. Given grammar :

E → E + Q/Q

Q → Q • M/M

M → id

Recursive descent parser for above grammar include procedure 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

$E \to E + E / E * E / id$

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 <· b	a "yields precedence to" b
a = b	a "has the same precedence as" b
a ·> 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 + <· * and * ·> +.
- 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 <·, = and ·> holds. Here $ mark is the end of string and define $ <· b and b ·> $ for all terminals b. For example, take one right sententional form id + id * id and the precedence relations are as below.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Then the string with precedence relation inserted

$ <· id ·> + <· id ·> * <· id ·> $

- The handle can be found out by following processes :

(1) Scan the string from left end until the first ·> is encountered. In above example this occurs between first id and +.

(2) Then scan backward over any = S until a <· is encountered. In above example we scan backward to $.

(3) The handle contains everything to the first ·> and to the right of the <· 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.

$ <· + <· * ·> $

- 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 → 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 ∈ then we need to minus ∈ from the FIRST(X) and proceed to FIRST (Y).

FIRST(A) = FIRST(X) – ∈ 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 [∈,1]) then begin
For (every production Z → w such that X appears in w) do
```

VBD

For (every appearance of X in w) do

begin Let $w = a \times b_1, b_2, \ldots, b_i$;

{where $\alpha \in (N \cup \in)$; and $b_j \in N$ to $1 < i < t$}

Let $p_i = 1$; {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

$F[X] := F[X] + (FIRST\ [b_{p,n}] - [\epsilon])$

If nullable (b_p) then $p_i = p + 1$

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 ( )

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 $[b_{p\ n}] - [\epsilon]$)

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 → (L)/a

L → L,S/S **CT : S-11(7M)**

Ans. Given grammar : S → (L)/a

L → L,S/S

Step 1 : In given grammar,

L → L, S/S is left recursion grammar.

By removing left recursion, we get

S → (L)/a

L → SL'

L' → ,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 :

T. / N.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 : IT-09(SM)**

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 → aA' / 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' → NB' / ϵ

and

E → gE'

E' → E' / ϵ

The modified grammar becomes :

S → aBDh

B → CB'

B' → bB' / ϵ

D → Eh

E → g / ϵ

Step 2 : Compute FIRST and FOLLOW :

FIRST (E) = FIRST (gE) ∪ { ϵ } = {g, ϵ}

FIRST (D) = FIRST (E) = ϵ ∪ FIRST (h)

= {g, ϵ} – {ϵ} ∪ {h}

= {g, h}

FIRST (B') = FIRST (bB') ∪ ϵ

= {b, ϵ}

FIRST (B) = FIRST (cB') = {c}

FIRST (S) = FIRST (aBDh) = {a}

FOLLOW (S) = {$}

FOLLOW (E) = FIRST (h) ∪ FOLLOW (E)

= {h}

FOLLOW (B) = FIRST (Dh)

= {g,h}

FOLLOW (B') = FOLLOW (B) ∪ FOLLOW (B')

= {g, h}

FOLLOW (D) = FIRST (h)

= {h}

Step 3 : LL (1) parsing table from above set result :

N.T. \ 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 → iLuET / a

L → LS / ϵ

E → b

T → dLe / ϵ

CS : W-13(8M),S-09(6M),S-14(7M)

Ans. Given grammar :

S → iLuET / a

L → LS / ϵ

E → b

T → dLe / ϵ

Step 1 : First eliminate left recursion from L

L → LS / ϵ

Self recursion gives

L → ϵ L'

L' → S L' / ϵ

as L → ← L'

So

L' → S L' / ϵ

L' → S L / ϵ

So grammar becomes

S → iLuET / a

L → SL / ϵ

E → b

T → dLe / ϵ

Step 2 : Compute FIRST and FOLLOW :

FIRST(T) = FIRST (d Le) ∪ FIRST {ϵ}

= {d, ϵ}

FIRST(E) = FIRST {b}

= {b}

FIRST (L) = FIRST(SL) ∪ FIRST(ϵ)

= {i, a, } ∪ {ϵ}= {i, a, ϵ}

FIRST S) = FIRST (iL u ET) ∪ FIRST (a)

= { i, a}

FOLLOW(L) = FIRST(u ET) ∪ FOLLOW (L) ∪ FIRST (e)

= { u,e}

VBD

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.

Q47. 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(8M)

OR 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. 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 :

$E \rightarrow T E'$

$E' \rightarrow + T E' / \epsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' / \epsilon$

$F \rightarrow (E) / id$

Step 2 : Compute FIRST and FOLLOW set :

FIRST (F) = FIRST { (E) } ∪ FIRST {id}

= { (, id}

FIRST (T') = FIRST (* FT') ∪ FIRST (ϵ)

= {*, ϵ}

FIRST (T) = FIRST (FT')

= {(, id}

FIRST (E') = FIRST (+TE') ∪ FIRST (ϵ)

= {+, ϵ}

FIRST (E) = FIRST (TE') = { (, id}

FOLLOW (E) = {$,) }

FOLLOW (E') = FOLLOW (ϵ) = {$,) }

FOLLOW (T) = FIRST (E') – (ϵ) ∪ FOLLOW (E)

= {+, ϵ} – ϵ ∪ {$}

= {+, $}

FOLLOW (T') = FOLLOW (T)

= {$, +,)

FOLLOW (F) = FIRST (T') − {ϵ} ∪ FOLLOW (T)

= {*, ϵ} − {ϵ} ∪ {$, + }

= {*, $, + }

Step 3 : Predictive parsing table for given grammar :

N.T \ T.	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → + TE'			E' → ϵ	E' → ϵ
T	E → FT'			T → FT'		
T'		T' → ϵ	T' → * FT'			T' → ϵ
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 / ϵ

A → bB / cc

B → bd / ϵ

CS : S-10(SM)

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 / ϵ

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: VII SEM (Old)

Ans. Given grammar :

S → ABD

A → aBSB

B → bD

D → d/∈

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 :

N.T. \ 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) ....(

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 : II-I (2M)

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 : II-I (2M)

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'.

$A \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bc / \epsilon$

$D \rightarrow EF$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$ **CS : S-13(7M)**

Ans. Given grammar :

$A \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bc / \epsilon$

$D \rightarrow EF$

$E \rightarrow g / \epsilon$

$F \rightarrow f / \epsilon$

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 :

N.T. \ 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,

$S \rightarrow aAC / bB$

$A \rightarrow e\,A'$

$A' \rightarrow bc\,A' / bd\,A' / \epsilon$

$B \rightarrow f / g$

$C \rightarrow h / i$

Step 2 : Compute FIRST and FOLLOW :

FIRST (S) = {a, b}

FIRST (A) = {e}

FIRST (A') = {b, ∈}

FIRST (B) = {f, g}

FIRST (C) = {h, i}

FOLLOW (S) = {$}

FOLLOW (B) = FOLLOW (S)

= {$}

FOLLOW (C) = {FOLLOW (S) ∪ FIRST (A)}

= {$} ∪ {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	∈
S	A → aAc		S → bB								
A	A → e										
B							B → f	B → g			
C									h → ∈	C → i	
A' E			A' → bcA'							A → ∈	

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.

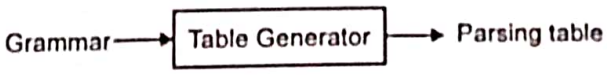


Fig.(a) Generating the parser

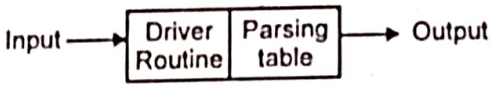


Fig.(b) Operation at the parser

- 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.

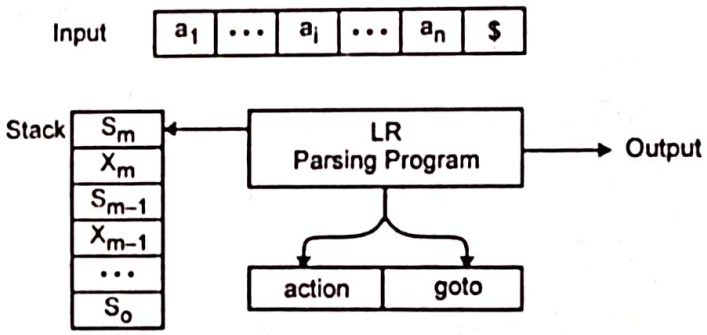


Fig.(c) Model of an LR parser.

- 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)

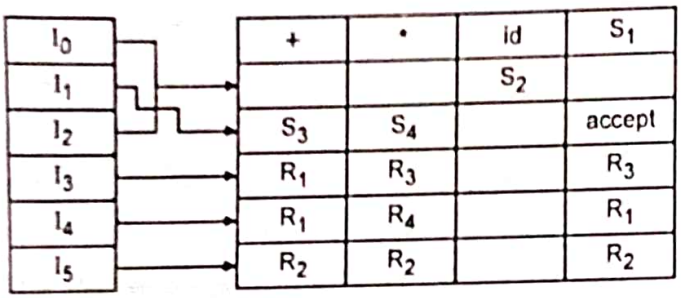
Ans.

- LR parsing table entries are made of two functions :

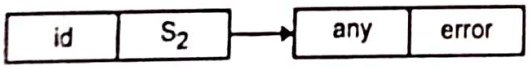
Parsing ACTION function and GOTO function.

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.

C.T. : W-09(4 M), S-09(3 M)

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)

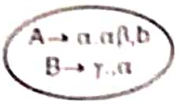


Fig.(b) LR(1) underlying set representations that can cause CLR/LALR parser conflicts.

(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.

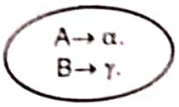


Fig.(c) LR(0) underlying set representations that can cause an SLR parser reduce-reduce conflicts.

- 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)

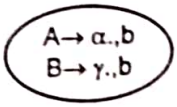


Fig.(d) LR(1) underlying set representations that can cause an CLR/LALR parser.

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. **CS : S-10(10M)**

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

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 :

$$I_0 = \text{Closure}(E' \rightarrow .E) = \left\{ \begin{matrix} E' \rightarrow .E \\ E \rightarrow .E + E \\ E \rightarrow .E * E \\ E \rightarrow .(E) \\ E \rightarrow .id \end{matrix} \right\} - (I_0)$$

$$\text{goto } (I_0, E) = \left\{ \begin{matrix} E' \rightarrow E. \\ E \rightarrow E. + E \\ E \rightarrow E. * E \end{matrix} \right\} - (I_1)$$

$$\text{goto } (I_0, () = \left\{ \begin{matrix} E \rightarrow (.E) \\ E \rightarrow .E + E \\ E \rightarrow .E * E \\ E \rightarrow .(E) \\ E \rightarrow .id \end{matrix} \right\} - (I_2)$$

$$\text{goto } (I_0, id) = \{E \rightarrow id.\} - (I_3)$$

$$\text{goto } (I_1, +) = \begin{Bmatrix} E \to E + .E \\ E \to .E + E \\ E \to .E * E \\ E \to .(E) \\ E \to .id \end{Bmatrix} = (I_4)$$

$$\text{goto } (I_1, *) = \begin{Bmatrix} E \to E * .E \\ E \to .E + E \\ E \to .E * E \\ E \to .(E) \\ E \to .id \end{Bmatrix} = (I_5)$$

$$\text{goto } (I_2, E) = \begin{Bmatrix} E \to (E.) \\ E \to E. + E \\ E \to E. * E \end{Bmatrix} = (I_6)$$

$$\text{goto } (I_2, () = \begin{Bmatrix} E \to (.E) \\ E \to .E + E \\ E \to .E * E \\ E \to .(E) \\ E \to .id \end{Bmatrix} = \text{same as } (I_2)$$

$$\text{goto } (I_2, id) = \{E \to id.\} = \text{same as } (I_3)$$

$$\text{goto } (I_4, E) = \begin{Bmatrix} E \to E + E. \\ E \to E. + E \\ E \to E. * E \end{Bmatrix} = (I_7)$$

$$\text{goto } (I_4, () = \begin{Bmatrix} E \to (.E) \\ E \to .E + E \\ E \to .E * E \\ E \to .(E) \\ E \to .id \end{Bmatrix} = \text{same as } (I_2)$$

$$\text{goto } (I_4, id) = \{E \to id.\} = \text{same as } (I_3)$$

$$\text{goto } (I_5, E) = \begin{Bmatrix} E \to E * E. \\ E \to E. + E \\ E \to E. * E \end{Bmatrix} = (I_8)$$

$$\text{goto } (I_5, () = \begin{Bmatrix} E \to (.E) \\ E \to .E + E \\ E \to .E * E \\ E \to .(E) \\ E \to .id \end{Bmatrix} = \text{same as } (I_2).$$

$$\text{goto } (I_5, id) = \{E \to id.\} = \text{same as } (I_3)$$

$$\text{goto } (I_6, +) = \text{same as } (I_4)$$

$$\text{goto } (I_6, *) = \text{same as } (I_5)$$

$$\text{goto } (I_6,)) = E \to (E). = (I_9)$$

$$\text{goto } (I_7, +) = \text{same as } (I_4)$$

$$\text{goto } (I_7, *) = \text{same as } (I_5)$$

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 :

$$I_0 = \text{Closure } (S' \to .S) = \begin{Bmatrix} S' \to .S \\ S \to .iCSeS \\ S \to .iCS \\ S \to .a \end{Bmatrix}$$

$$\text{goto } (I_0, S) = S' \to S. = (I_1)$$

$$\text{goto } (I_0, i) = \begin{Bmatrix} S \to i.CSeS \\ S \to i.CS \end{Bmatrix} (I_2)$$

$$\text{goto } (I_0, a) = S \to a. = (I_3)$$

$$\text{goto } (I_2, C) = \begin{Bmatrix} S \to iC.SeS \\ S \to iC.S \\ S \to .iCSeS \\ S \to .iCS \\ S \to .a \end{Bmatrix} - (I_4)$$

$$\text{goto } (I_4, S) = \begin{Bmatrix} S \to iCS.eS \\ S \to iCS. \end{Bmatrix} - (I_5)$$

$$\text{goto } (I_4, i) = \begin{Bmatrix} S \to i.CSeS \\ S \to i.CS \end{Bmatrix} \text{ same as } (I_2)$$

$$\text{goto } (I_4, a) = \{S \to a.\} - \text{same as } (I_3)$$

$$\text{goto } (I_5, e) = \begin{Bmatrix} S \to iCSe.S \\ S \to .iCS \\ S \to .iCSeS \\ S \to .a \end{Bmatrix} - (I_6)$$

$$\text{goto } (I_6, S) = \{S \to iCSeS.\} - (I_7)$$

$$\text{goto } (I_6, i) = \begin{Bmatrix} S \to i.CS \\ S \to i.CSeS \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_6, a) = \{S \to a.\}$$

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_4	R_3	1	2	4
I_1			Accept			
I_2			R_1			
I_3	S_3	S_4				6
I_4	S_3	S_4	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

$S \rightarrow A$ ...(I)

$A \rightarrow BA$ ...(II)

$A \rightarrow \epsilon$ ...(III)

$B \rightarrow aB$ ...(IV)

$B \rightarrow b$ ...(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 $B \rightarrow b$
$ aaaB	$	Reduction using $B \rightarrow aB$
$ aaB	$	Reduction using $B \rightarrow aB$
$ aB	$	Reduction using $B \rightarrow aB$
$ B	$	Reduction using $A \rightarrow \epsilon$
$ A	$	Reduction using $S \rightarrow A$
$ 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

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

$A \rightarrow r$

$B \rightarrow r$

CS : W-12 (10M)

Ans.

Step 1 : Augmented grammar is :

$S' \rightarrow S$

$S \rightarrow .\, wAz$ ... (I)

$S \rightarrow .\, xBz$ ... (II)

$S \rightarrow .\, wBy$ ... (III)

$S \rightarrow .\, xAy$ ... (IV)

$A \rightarrow .\, r$ ... (V)

$B \rightarrow .\, r$ ... (VI)

Step 2 : LR (1) set of items are as follows :

$$I_0 = \text{Closure}\{(S' \rightarrow .S)\} = \left\{\begin{matrix} S' \rightarrow .S, \$ \\ S \rightarrow .\, w\, A\, z, \$ \\ S \rightarrow .\, xBz, \$ \\ S \rightarrow .\, wBy, \$ \\ S \rightarrow .\, zAy, \$ \end{matrix}\right\} - (I_0)$$

$$\text{goto}\ (I_0, S) = \{S' \rightarrow S\ .\ , \$\} - (I_1)$$

$$\text{goto}\ (I_0, w) = \left\{\begin{matrix} S \rightarrow w\ .\ Az, \$ \\ S \rightarrow w\ .\ By, \$ \\ A \rightarrow .\, r, y \\ B \rightarrow .\, r, z \end{matrix}\right\} - (I_2)$$

$$\text{goto}\ (I_0, x) = \left\{\begin{matrix} S \rightarrow x\ .\ Bz, \$ \\ B \rightarrow .\, r, z \end{matrix}\right\} - (I_3)$$

$$\text{goto}\ (I_0, z) = \left\{\begin{matrix} S \rightarrow z\ .\ Ay, \$ \\ A \rightarrow .\, r, y \end{matrix}\right\} - (I_4)$$

$$\text{goto}\ (I_2, A) = \{S \rightarrow wA\ .\ z, \$\} - (I_5)$$

$$\text{goto}\ (I_2, B) = \{S \rightarrow wB\ .\ y, \$\} - (I_6)$$

$$\text{goto}\ (I_2, r) = \left\{\begin{matrix} A \rightarrow r\ .\ , z \\ B \rightarrow r\ .\ , z \end{matrix}\right\} - (I_7)$$

$$\text{goto}\ (I_3, B) = \{S \rightarrow xB\ .\ z, \$\} - (I_8)$$

$$\text{goto}\ (I_3, r) = \{B \rightarrow r\ .\ , z\} - (I_9)$$

$$\text{goto}\ (I_4, A) = \{S \rightarrow zA\ .\ y, \$\} - (I_{10})$$

$$\text{goto}\ (I_4, r) = \{A \rightarrow r\ .\ , y\} - (I_{11})$$

VBD

goto $(I_1, z) = \{S \to wAz\ .\ , \$\} = (I_{12})$

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 :

S.T. \ I.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.49. 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 :

$$I_0 = \text{Closure}\{E' \to .E\} = \left\{\begin{array}{c} E' \to .E, \$ \\ E \to .E + T, \$/+ \\ E \to .T\$/+ \\ T \to .TF, \$/+/a/b \\ T \to .F, \$/+/a/b \\ F \to .F^*, \$/+/a/b/^* \\ F \to .a, \$/+/a/b/^* \\ F \to .b, \$/+/a/b/^* \end{array}\right\} = (I_0)$$

$$\text{goto}(I_0, E) = \left\{\begin{array}{c} E' \to E., \$ \\ E \to E. + T, \$/+ \end{array}\right\} = (I_1)$$

$$\text{goto}(I_0, T) = \left\{\begin{array}{c} E \to T., \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F^*, \$/a/+/b/^* \\ F \to .a, \$/a/+/b/^* \\ F \to .b, \$/a/+/b/^* \end{array}\right\} = (I_2)$$

$$\text{goto}(I_0, F) = \left\{\begin{array}{c} T \to F., \$/+/a/b \\ F \to F.^*, \$/+/a/b/^* \end{array}\right\} = (I_3)$$

$$\text{goto}(I_0, a) = \{F \to a., \$/+/a/b/^*\} = (I_4)$$

$$\text{goto}(I_0, b) = \{F \to b., \$/+/a/b/^*\} = (I_5)$$

$$\text{goto}(I_1, +) = \left\{\begin{array}{c} E \to E + .T, \$/+ \\ T \to .F, \$/+/a/b \\ T \to .F, \$/+/a/b \\ F \to .F^*\$/+/a/b/^* \\ F \to .a\$/+/a/b/^* \\ F \to .b\$/+/a/b/^* \end{array}\right\} = (I_6)$$

$$\text{goto}(I_2, F) = \left\{\begin{array}{c} T \to TF., \$/+/a/b \\ F \to F.^*\$/+/a/b/^* \end{array}\right\} = (I_7)$$

$$\text{goto}(I_2, a) = \{F \to a., \$/+/a/b/^*\} = (I_4)$$

$$\text{goto}(I_2, b) = \{F \to b., \$/+/a/b/^*\} = (I_5)$$

$$\text{goto}(I_3, ^*) = \{F \to F^*., \$/+/a/b/^*\} = (I_8)$$

$$\text{goto } (I_6, T) = \begin{Bmatrix} E \to E + T\ .\ \$/+ \\ T \to T\ F\ .\ \$/+/a/b \\ F \to .F^*\ .\ \$/+/a/b/^* \\ F \to .a\ .\ \$/+/a/b/^* \\ F \to .b\ .\ \$/+/a/b/^* \end{Bmatrix} - (I_9)$$

$$\text{goto } (I_6, F) = \begin{Bmatrix} T \to F\ .\ \$/+/a/b \\ F \to F\ ^*\ .\ \$/+/a/b/^* \end{Bmatrix} \quad \text{.....same as } (I_3)$$

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)

$$\text{goto } (I_9, F) = \begin{Bmatrix} T \to TF\ .\ \$/+/a/b \\ F \to F\ .^*\ \$/a/+/b \end{Bmatrix} \quad \text{.....same as } (I_7)$$

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

$$I_0 = \{S' \to .S) = \begin{Bmatrix} S' \to .S, \$ \\ S \to .cA, \$ \\ S \to .ccB, \$ \end{Bmatrix}$$

goto $(I_0, S) = \{S' \to S.,\$\} - (I_1)$

$$\text{goto } (I_0, c) = \begin{Bmatrix} S \to c.A, \$ \\ S \to c.cB, \$ \\ A \to .cA, \$ \\ A \to .a, \$ \end{Bmatrix} - (I_2)$$

goto $(I_2, A) = \{S \to cA.,\$\} - (I_3)$

goto $(I_2, a) = \{A \to a.,\$\} - (I_4)$

$$\text{goto } (I_2, c) = \begin{Bmatrix} S \to cc.B, \$ \\ A \to c.A, \$ \\ A \to .cA, \$ \\ A \to .a, \$ \\ B \to .ccB, \$ \\ B \to .b, \$ \end{Bmatrix} - (I_5)$$

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 $(I_5, b) = \{B \to b.,\$\} - (I_8)$

$$\text{goto } (I_5, c) = \begin{Bmatrix} A \to c.A, \$ \\ B \to c.cB, \$ \\ A \to .cA, \$ \\ A \to .a, \$ \end{Bmatrix} - (I_9)$$

goto $(I_9, c) = \{A \to cA.,\$\} - (I_7)$

$$\text{goto } (I_9, c) = \begin{Bmatrix} A \to c.A, \$ \\ B \to cc.B, \$ \\ A \to .cA, \$ \\ A \to .a, \$ \\ B \to .ccB, \$ \\ B \to .b, \$ \end{Bmatrix} - (I_{10})$$

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 :

$$I_0 = \{E' \to .E\} = \left\{ \begin{array}{c} E' \to .E,\$ \\ E \to .E + T,\$/+ \\ E \to .T,\$/+ \\ T \to .TF,\$/+/a/b \\ T \to F,\$/+/a/b \\ F \to .F^*,\$/+/a/b/^* \\ F \to .a\$/+/a/b/^* \\ F \to .b\$/+/a/b/^* \\ F \to (E).\$/+/a/b/^* \end{array} \right\} - (I_0)$$

$$\text{goto } (I_0, E) = \begin{Bmatrix} E' \to E., \$ \\ E \to E. + T, \$/+ \end{Bmatrix} - (I_1)$$

$$\text{goto } (I_0, T) = \begin{Bmatrix} E \to T., \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F^*, \$/+/a/b/^* \\ F \to .a, \$/+/a/b/^* \\ F \to .b, \$/+/a/b/^* \\ F \to (E), \$/+/a/b/^* \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_0, F) = \begin{Bmatrix} T \to F., \$/+/a/b \\ F \to F.^* \$/+/a/b/^* \end{Bmatrix} - (I_3)$$

$$\text{goto } (I_0, a) = \{F \to a., \$/+/a/b/^*\} - (I_4)$$

$$\text{goto } (I_0, b) = \{F \to b., \$/+/a/b/^*\} - (I_5)$$

$$\{F \to (.E), \$|+|a|b|^*$$

$$\text{goto } (I_0, () = \begin{Bmatrix} E \to .E + T, \$/T \\ E \to .T, \$/+ \\ T \to .TF, \$/+/a/b \\ T \to .F, \$/+/a/b \\ F \to .F^* \$/+/a/b/^* \\ F \to .a \$/+/a/b/^* \\ F \to .b \$/+/a/b/^* \\ F \to .(E), \$/+/a/b/^* \end{Bmatrix} - (I_6)$$

$$\text{goto } (I_1, +) = \begin{Bmatrix} E \to E + .T, \$/+ \\ T \to .TF, \$/+/a/b \\ T \to .F, \$/a/+/b \\ F \to .F^* \$/+/a/b/^* \\ F \to .a \$/+/a/b/^* \\ F \to .b \$/+/a/b/^* \\ F \to (E), \$/+/a/b/^* \end{Bmatrix} - (I_7)$$

$$\text{goto } (I_2, F) = \begin{Bmatrix} T \to TF., \$/+/a/b \\ F \to F.^*, \$/+/a/b/^* \end{Bmatrix} - (I_8)$$

$$\text{goto } (I_2, b) = (I_5)$$

$$\text{goto } (I_2, a) = (I_4)$$

$$\text{goto } (I_3, ^*) = \{F \to F^*., \$/+/a/b/^*\} - (I_9)$$

$$\text{goto } (I_7, ^*) = (I_9)$$

$$\text{goto } (I_6, E) = \begin{Bmatrix} F \to (E.), \$/+/a/b/^* \\ E \to E. + T, \$, /T \end{Bmatrix} - (I_{10})$$

$$\text{goto } (I_6, T) = \begin{Bmatrix} E \to T.,, \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F^*, \$/+/a/b/ \\ F \to .a, \$/+/a/b/^* \\ F \to .b, \$/+/a/b/^* \\ F \to .(E), \$/+/a/b/^* \end{Bmatrix} - (I_{11})$$

$$\text{goto } (I_6, F) = \begin{Bmatrix} T \to F., \$/+/a/b \\ F \to F.^*, \$/+/a/b/^* \end{Bmatrix} \quad \text{..... same as } (I_3)$$

$$\text{goto } (I_6, a) = \text{same as } (I_4)$$

$$\text{goto } (I_6, b) = \text{same as } (I_5)$$

$$\text{goto } (I_6, () = \text{same as } (I_6)$$

$$\text{goto } (I_7, T) = \begin{Bmatrix} E \to E + T., \$/+ \\ T \to T.F, \$/+/a/b \\ F \to .F^*, \$/+/a/b/^* \\ F \to .a, \$/+/a/b/^* \\ F \to .b, \$/+/a/b/^* \\ F \to .(E). \$/a/b/^* \end{Bmatrix} - (I_{12})$$

$$\text{goto } (I_7, F) = \text{same as } (I_8)$$

$$\text{goto } (I_7, a) = \text{same as } (I_4)$$

$$\text{goto } (I_7, b) = \text{same as } (I_5)$$

$$\text{goto } (I_7, () = \text{ame as } (I_6)$$

$$\text{goto } (I_8, ^*) = \{F \to F^*., \$/+/a/b/^*\} \quad \text{..... same as } (I_9)$$

$$\text{goto } (I_{10},)) = \{F \to (E)., \$/+/a/b/^*\} - (I_3).$$

$$\text{goto } (I_{10}, +) = \begin{Bmatrix} E \to E + .T, \$/T \\ T \to .TF, \$/+/a/b \\ T \to .T, \$/+ \\ T \to .F, \$/+/a/b \\ F \to .F^* \$/+/a/b/^* \\ F \to .a \$/+/a/b/^* \\ F \to .b \$/+/a/b/^* \\ F \to .(E), \$/+/a/b/^* \end{Bmatrix} \text{.....Same as } (I_7)$$

$$\text{goto } (I_{11}, f) = \text{same as } (I_3)$$

$$\text{goto } (I_{11}, a) = (I_4)$$

$$\text{goto } (I_{11}, b) = (I_5)$$

$$\text{goto } (I_{11}, c) = \text{Same as } (I_6)$$

$$\text{goto } (I_{12}, F) = \begin{Bmatrix} T \to TF., \$/+/a/b \\ T \to F.^*, \$/+/a/b/^* \end{Bmatrix} \text{.....Same as } (I_8)$$

$$\text{goto } (I_{12}, a) = \text{same as } (I_4)$$

$$\text{goto } (I_{12}, b) = \text{same as } (I_5)$$

$$\text{goto } (I_{12}, () = \text{same as } (I_6)$$

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 → L / a

L → w G d S / d S we

G → b

Ans.

Step 1 : Augmented grammar :

S' → S

S → L / a

L → .w G d S / d S we

G → .b

Step 2 : LR (1) items are computed as follows :

$$I_0 = \text{Closure}\{(S' \to S)\} = \left\{\begin{matrix} S' \to .S, \$ \\ S \to .L, \$ \\ S \to .a, \$ \\ L \to .wGdS, \$ \\ L \to .dSwe, \$ \end{matrix}\right\} - (I_0)$$

goto $(I_0, S) = \{S' \to S., \$\} - (I_1)$

goto $(I_0, L) = \{S \to L., \$\} - (I_2)$

goto $(I_0, a) = \{S \to a., \$\} - (I_3)$

$$\text{goto } (I_0, w) = \left\{\begin{matrix} L \to w.GdS, \$ \\ G \to .b, d \end{matrix}\right\} - (I_4)$$

$$\text{goto } (I_0, d) = \left\{\begin{matrix} L \to d.Swe, \$ \\ S \to .L, w \\ S \to .a, w \\ L \to .wGdS, w \\ L \to .dSwe, w \end{matrix}\right\} - (I_5)$$

goto $(I_4, G) = \{L \to wG.dS, \$\} - (I_6)$

goto $(I_4, b) = \{G \to b., d\} - (I_7)$

goto $(I_5, S) = \{L \to dS.we, \$\} - (I_8)$

goto $(I_5, L) = \{S \to L., w\} - (I_9)$

goto $(I_5, a) = \{S \to a., w\} - (I_{10})$

$$\text{goto } (I_5, w) = \left\{\begin{matrix} L \to w.GdS, w \\ G \to b., d \end{matrix}\right\} - (I_{11})$$

$$\text{goto } (I_5, d) = \left\{\begin{matrix} L \to d.Swe, w \\ S \to .L, w \\ S \to .a, w \\ L \to .wGdS, w \\ L \to .dSwe, w \end{matrix}\right\} - (I_{12})$$

$$\text{goto } (I_6, d) = \left\{\begin{matrix} L \to wGd.S, \$ \\ S \to .L, \$ \\ S \to .a, \$ \\ L \to .wGdS, \$ \\ L \to .dSwe, \$ \end{matrix}\right\} - (I_{13})$$

goto $(I_8, w) = \{L \to dSw.e, \$\} - (I_{14})$

goto $(I_{11}, G) = \{L \to wG.dS, w\} - (I_{15})$

goto $(I_{11}, b) = \{G \to b., d\}$ – same as (I_7)

goto $(I_{12}, S) = \{L \to dS, we, w\} \to (I_{16})$

goto $(I_{12}, L) = \{S \to L., w\}$ – same as (I_9)

goto $(I_{12}, a) = \{S \to a., w\}$ – same as (I_{10})

$$\text{goto } (I_{12}, w) = \left\{\begin{matrix} L \to w.GdS, w \\ G \to .b, d \end{matrix}\right\} - \text{same as } (I_{11})$$

$$\text{goto } (I_{12}, d) = \left\{\begin{matrix} L \to d.Swe, w \\ S \to .L, w \\ S \to .a, w \\ L \to .wGdS, w \\ L \to .dSwe, w \end{matrix}\right\} - \text{same as } (I_{12})$$

goto $(I_{13}, S) = \{L \to wGdS., \$\} - (I_{17})$

goto $(I_{11}, L) = \{S \to L.,\$\}$ – 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 $(I_{16}, w) = \{L \to dSw.e,w\}$ – (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 → Aa / bAc / Bc /bBa

A → d

B → d

Ans.

Step 1 : Augmented grammar :

S' → .S ...(I)

S → .Aa ...(II)

S → .bAc ...(III)

S → .Bc ...(IV)

S → .bBa ...(V)

A → .d ...(VI)

B → .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 :

$$I_0 = \text{Closure}\{(S' \to .S, \$)\} = \left\{\begin{matrix} S' \to .S, \$ \\ S \to .B, \$ \\ B \to .\text{begin } DA \text{ end}, \$ \end{matrix}\right\}$$

$$\text{goto } (I_0, S) = \text{Closure}(\{S' \to S., \$\}) = \{S' \to S., \$\} \quad -(I_1)$$

$$\text{goto } (I_0, B) = \text{Closure}(\{S \to B., \$\}) = \{S \to B., \$\} \quad -(I_2)$$

$$\text{goto } (I_0, \text{begin}) = \left\{\begin{matrix} B \to \text{begin}.DA \text{ end}, \$ \\ D \to .Dd, \text{begin} / d \\ D \to ., \text{begin} / d \end{matrix}\right\} \quad -(I_3)$$

$$\text{goto } (I_3, D) = \left\{\begin{matrix} B \to \text{begin } D.A \text{ end}, \$ \\ D \to D.d, \text{begin} / d \\ D \to ., \text{begin} / d \\ A \to .A ; E, \text{end} / ; \\ A \to .E, \text{end} / ; \\ E \to .B, \text{end} / ; \\ B \to .\text{begin } DA \text{ end}, \text{end} / ; \\ S \to .B, \text{end} / ; \end{matrix}\right\} \quad -(I_4)$$

$$\text{goto } (I_4, A) = \left\{\begin{matrix} B \to \text{begin DA . end}, \$ \\ A \to A .; E \end{matrix}\right\} - (I_5)$$

$$\text{goto } (I_4, d) = \{(D \to D, \text{begin} / d)\} - (I_6)$$

$$\text{goto } (I_4, E) = \{A \to E ., \text{end} / ;\} - (I_7)$$

$$\text{goto } (I_8, B) = \left\{\begin{matrix} E \to B ., \text{end} / ; \\ S \to B ., \text{end} / ; \end{matrix}\right\} - (I_8)$$

$$\text{goto } (I_4, \text{begin}) = \left\{\begin{matrix} . B \to \text{begin . DA end}, \text{end} / ; \\ D \to . Dd, \text{begin} / d \\ D \to ., \text{begin} / d \end{matrix}\right\} - (I_9)$$

$$\text{goto } (I_4, S) = \{E \to S ., \text{end} / ;\} - (I_{10})$$

$$\text{goto } (I_5, \text{end}) = \{B \to \text{begin DA end} ., \$\} - (I_{11})$$

$$\text{goto } (I_5, ;) = \left\{\begin{matrix} A \to A ; . E, \text{end} / ; \\ E \to . B, \text{end} / ; \\ E \to . S, \text{end} / ; \\ B \to . \text{begin DA end} ; \text{end} / ; \\ S \to . B, \text{end} / ; \end{matrix}\right\} - (I_{12})$$

$$\text{goto } (I_9, D) = \left\{\begin{matrix} B \to \text{begin D . A end}, \text{end} / ; \\ D \to D . d, \text{begin} / d \\ A \to . A ; E, \text{end} / ; \\ A \to . E, \text{end} / ; \\ E \to . B, \text{end} / ; \\ E \to . S, \text{end} / ; \\ B \to . \text{begin DA end}, \text{end} / ; \\ S \to . B, \text{end} / ; \end{matrix}\right\} - (I_{13})$$

$$\text{goto } (I_{12}, E) = \{A \to A ; E ., \text{end} / ;\} - (I_{14})$$

$$\text{goto } (I_{12}, B) = \left\{\begin{matrix} E \to B ., \text{end} / ; \\ S \to B ., \text{end} / ; \end{matrix}\right\} - \text{same as } (I_8)$$

$$\text{goto } (I_{12}, S) = \{E \to S ., \text{end} / ;\} - \text{same as } (I_{10})$$

goto

$$(I_{12}, \text{begin}) = \left\{\begin{matrix} B \to \text{begin . DA end}, \text{end} / ; \\ D \to . Dd ; \text{begin} / d \\ D \to ., \text{begin} / d \end{matrix}\right\} - \text{same as } (I_9)$$

$$\text{goto } (I_{13}, A) = \left\{\begin{matrix} B \to \text{begin DA . end}, \text{end} / ; \\ A \to . A ; E, \text{end} / ; \end{matrix}\right\} - (I_{15})$$

$$\text{goto } (I_{13}, d) = \{D \to Dd ., \text{begin} / \} - \text{same as } (I_6)$$

$$\text{goto } (I_{13}, E) = \{A \to E ., \text{end} / ;\} - \text{same as } (I_7)$$

$$\text{goto } (I_{13}, B) = \{S \to B ., \text{end} / ;\} - \text{same as } (I_8)$$

$$\text{goto } (I_{13}, S) = \{E \to S ., \text{end} / ;\} - \text{same as } (I_{10})$$

goto

$$(I_{13}, \text{begin}) = \left\{\begin{matrix} B \to \text{begin . DA end}, \text{end} / ; \\ D \to . Dd ; \text{begin} / d \\ D \to ., \text{begin} / d \end{matrix}\right\} - \text{same as } (I_9)$$

$$\text{goto } (I_{15}, \text{end}) = \{B \to \text{begin DA end} ., \text{end} / ;\} - (I_{16})$$

$$\text{goto } (I_{15}, ;) = \left\{\begin{matrix} A \to A ; . E, \text{end} / ; \\ E \to . B, \text{end} / ; \\ E \to . S, \text{end} / ; \\ B \to . \text{begin DA end} ; \text{end} / ; \\ S \to . B, \text{end} / ; \end{matrix}\right\} - \text{same as } (I_{12})$$

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 :

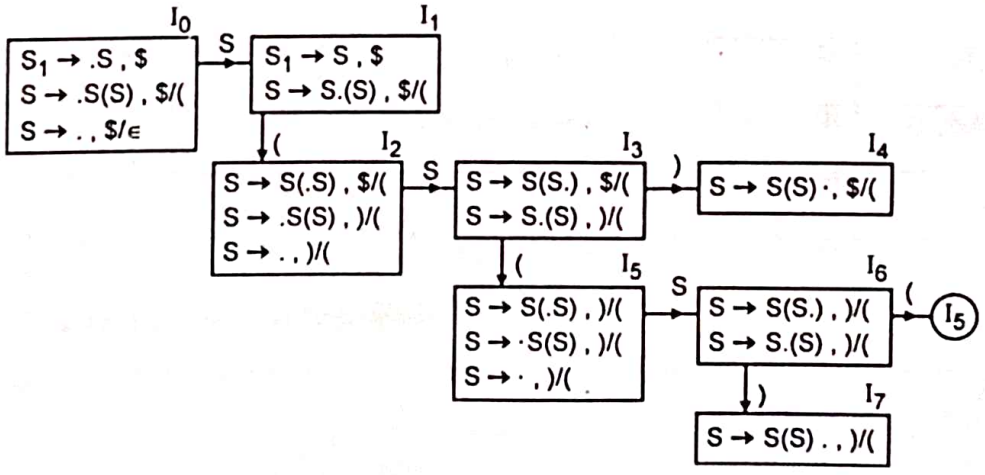
Augmented grammar :

$S_1 \rightarrow .S$

$S \rightarrow .S\,(s)$

$S \rightarrow .\in$

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 $S \rightarrow \epsilon$
$\$I_0SI_1(I_2SI_3(I_5SI_6$	)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.

4 BD

Q.76. Construct LR (0) parsing table for following grammar :

S → aIJh

I → IbSe / c

J → KLKr / ∈

K → d / ∈

L → p / ∈

Ans.

Step 1 : The augmented grammar :

S' → .S

S → .aIJh

I → .IbSe / c

J → .KLKr / ∈

K → .d / ∈

L → .p / ∈

Step 2 : Canonical collection of set of LR(1) item are as follows :

$$I_0 = \text{Closure}\{(S' \to .S)\} = \begin{Bmatrix} S' \to .S \\ S \to .aIJh \end{Bmatrix} - I_0$$

$$\text{goto}\ (I_0, S) = \{S' \to S.\} \quad -(I_1)$$

$$\text{goto}\ (I_0, a) = \begin{Bmatrix} S \to a.IJh \\ I \to .IbSe \\ I \to c \end{Bmatrix} \quad -(I_2)$$

$$\text{goto}\ (I_2, I) = \begin{Bmatrix} S \to aI.Jh \\ J \to .kLkr \\ J \to . \\ K \to .d \\ K \to . \\ I \to I.bSe \end{Bmatrix} \quad -(I_3)$$

$$\text{goto}\ (I_2, c) = \{I \to c.\} \quad -(I_4)$$

$$\text{goto}\ (I_3, J) = \{S \to aIJ.h\} \quad -(I_5)$$

$$\text{goto}\ (I_3, K) = \begin{Bmatrix} J \to K.LKr \\ L \to .P \\ L \to . \end{Bmatrix} \quad -(I_6)$$

$$\text{goto}\ (I_3, d) = \{K \to d.\} \quad -(I_7)$$

$$\text{goto}\ (I_3, b) = \begin{Bmatrix} I \to Ib.Se \\ S \to .aIJh \end{Bmatrix} \quad -(I_8)$$

$$\text{goto}\ (I_5, h) = \{S \to aIJh.\} \quad -(I_9)$$

$$\text{goto}\ (I_6, L) = \begin{Bmatrix} J \to KL.Kr \\ K \to .d \\ K \to . \end{Bmatrix} \quad -(I_{10})$$

goto $(I_6, P) = \{L \rightarrow P.\} - (I_{11})$

goto $(I_8, S) = \{I \rightarrow IbS.e\} - (I_{12})$

goto $(I_8, a) = \left\{\begin{matrix} S \rightarrow a.IJh \\ I \rightarrow .IbSe \\ I \rightarrow .e \end{matrix}\right\}$ – 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 $C = \{I_0, I_1 \ldots I_n\}$.

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 :

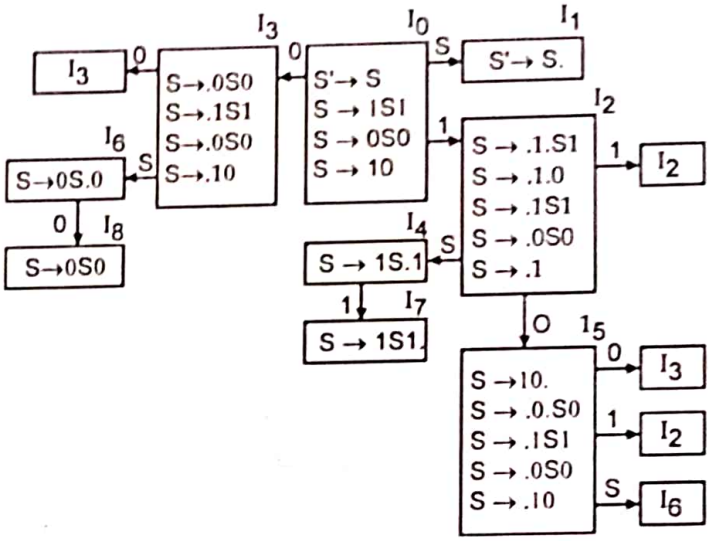
S' → .S

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 → d and stack is

oc36c89

State 89 with input $ reduces with C → 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' → S

S → CC

C → 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 → cC / c / d / $

C → cC . c / d / $

C → d . c / d / $

I_{47} = C → d . cd / $

I_{89} = C → 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(SM)

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 → L = R / R

L → * R / id

R → L

Ans.

Step 1 : Augmented grammar :

S' → . S

S → L = R

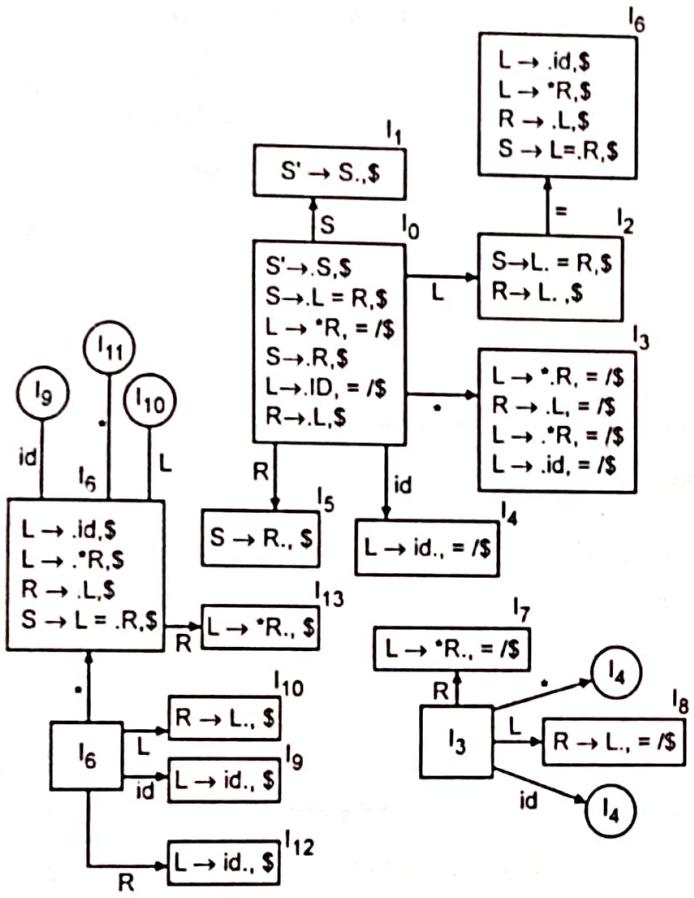
S → R

L → * R

L → id

R → 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 $(I_{10}, a) = (I_9)$

goto $(I_{10}, \wedge) = (I_6)$

goto $(I_{13},)) = S \to (R) . , \$ / , - (I_{15})$

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 :

$I_{28} = \{S \to a . , \$ /) / , \}$

$I_{39} = \{S \to \wedge . , \$ /) / , \}$

$I_{410} = \{S \to (. R) ,) / , / \$ \}$

$I_{513} = \{S \to (R .) , \$ /) / , \}$

$I_{1115} = \{S \to (R) . , \$ /) / , \}$

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 $S \to a$. 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

$S \to aSbS/bSaS/\in .$ **CT : S-10(8.3)**

Ans.

Step 1 : Augmented grammar :

$S' \to .S$ ... (I)

$S \to .aSbS$ ... (II)

$S \to .bSaS$ ... (III)

$S \to .\epsilon$ ... (IV)

Step 2 : Canonical collection of LR (1) items :

$$I_0 = \text{Closure}(S' \to .S) = \begin{Bmatrix} S' \to .S, \$ \\ S \to .aSbS, \$ \\ S \to .bSaS, \$ \\ S \to ., \$ \end{Bmatrix} - (I_0)$$

$$\text{goto}(I_0, S) = \{S' \to S ., \$\} - (I_1)$$

$$\text{goto}(I_0, a) = \begin{Bmatrix} S \to a . SbS, \$ \\ S \to .aSbS, b \\ S \to .bSaS, b \\ S \to ., b \end{Bmatrix} - (I_2)$$

$$\text{goto } (I_0, b) = \begin{Bmatrix} S \to b\ .\ SaS, \$ \\ S \to\ .\ aSbS, a \\ S \to\ .\ bSaS, a \\ S \to\ .\ , a \end{Bmatrix} - (I_3)$$

$$\text{goto } (I_1, S) = \{S \to bS\ .\ aS, \$\} - (I_4)$$

$$\text{goto } (I_3, a) = \begin{Bmatrix} S \to a\ .\ SbS, a \\ S \to\ .\ aSbS, b \\ S \to\ .\ bSaS, b \\ S \to\ .\ , b \end{Bmatrix} - (I_5)$$

$$\text{goto } (I_3, b) = \begin{Bmatrix} S \to a\ .\ SbS, a \\ S \to\ .\ aSbS, a \\ S \to\ .\ bSaS, a \\ S \to\ .\ , a \end{Bmatrix} - (I_6)$$

$$\text{goto } (I_6, S) = S \to bS\ .\ aS, a \quad - (I_7)$$

$$\text{goto } (I_7, a) = \begin{Bmatrix} S \to bSa\ .\ S, a \\ S \to\ .\ aSbS, a \\ S \to\ .\ bSaS, a \\ S \to\ .\ , a \end{Bmatrix} - (I_8)$$

$$\text{goto } (I_8, S) = S \to bSaS\ .\ , a \quad - (I_9)$$

$$\text{goto } (I_2, S) = S \to aS\ .\ bS, \$ \quad - (I_{10})$$

$$\text{goto } (I_2, a) = \begin{Bmatrix} S \to a\ .\ SbS, b \\ S \to\ .\ aSbS, b \\ S \to\ .\ bSaS, b \\ S \to\ .\ , b \end{Bmatrix} - (I_{11})$$

$$\text{goto } (I_2, b) = \begin{Bmatrix} S \to b\ .\ SaS, b \\ S \to\ .\ aSbS, a \\ S \to\ .\ bSaS, a \\ S \to\ .\ , a \end{Bmatrix} - (I_{12})$$

$$\text{goto } (I_{11}, S) = \{S \to aS\ .\ bS, b\} - (I_{13})$$

$$\text{goto } (I_{13}, b) = \begin{Bmatrix} S \to aSb\ .\ S, b \\ S \to\ .\ aSbS, b \\ S \to\ .\ bSaS, b \\ S \to\ .\ , b \end{Bmatrix} - (I_{14})$$

$$\text{goto } (I_{14}, S) = \{S \to aSbS\ .\ , b\} - (I_{15})$$

$$\text{goto } (I_{14}, b) = (I_{12})$$

$$\text{goto } (I_{14}, a) = (I_{11})$$

$$\text{goto } (I_{12}, b) = (I_6)$$

$$\text{goto } (I_{12}, a) = (I_5)$$

$$\text{goto } (I_4, a) = \begin{Bmatrix} S \to bSa\ .\ S, b \\ S \to\ .\ aSbS, a \\ S \to\ .\ bSaS, a \\ S \to\ .\ , a \end{Bmatrix} - (I_{16})$$

$$\text{goto } (I_{16}, S) = S \to bSaS\ .\ , \$ \quad - (I_{17})$$

$$\text{goto } (I_{16}, b) = (I_3)$$

$$\text{goto } (I_{16}, a) = (I_2)$$

$$\text{goto } (I_{10}, b) = \begin{Bmatrix} S \to aSb\ .\ S, \$ \\ S \to\ .\ aSbS, \$ \\ S \to\ .\ bSaS, \$ \\ S \to\ .\ , \$ \end{Bmatrix} - (I_{18})$$

$$\text{goto } (I_{18}, S) = S \to aSbS\ .\ , \$ \quad - (I_{19})$$

$$\text{goto } (I_{18}, a) = (I_2)$$

$$\text{goto } (I_{18}, b) = (I_3)$$

$$\text{goto } (I_5, S) = S \to aS\ .\ bS, a \quad - (I_{20})$$

$$\text{goto } (I_5, b) = (I_{12})$$

$$\text{goto } (I_5, a) = (I_{11})$$

$$\text{goto } (I_{20}, b) = \begin{Bmatrix} S \to aSb\ .\ S, a \\ S \to\ .\ aSbS, a \\ S \to\ .\ bSaS, a \\ S \to\ .\ , a \end{Bmatrix} - (I_{21})$$

$$\text{goto } (I_{21}, b) = (I_6)$$

$$\text{goto } (I_{21}, a) = S \to aSbS\ .\ , a - (I_{22})$$

$$\text{goto } (I_{12}, S) = \{S \to bS\ .\ aS, b\} \quad - (I_{23})$$

$$\text{goto } (I_{23}, a) = \begin{Bmatrix} S \to bSa\ .\ S, b \\ S \to\ .\ aSbS, b \\ S \to\ .\ bSaS, b \\ S \to\ .\ , b \end{Bmatrix} - (I_{24})$$

$$\text{goto } (I_{24}, S) = \{S \to bSaS\ .\ , b\} \quad - (I_{25})$$

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	I
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 :

$S \to abSa \mid aaAc \mid b$

$A \to dAb \mid b$ **CT : S-09(8.M)**

Ans.

Step 1 : Augmented grammar :

$S' \to S$ ... (I)

$S \to abSa$ ... (II)

$S \to aaAc$ .... (III)

$S \to b$ ... (IV)

$A \to dAb$ ... (V)

$A \to b$ ... (VI)

Step 2 : The canonical collection of LR(1) items are as follows :

$$I_0 = \text{Closure}(S' \to .S) = \begin{Bmatrix} S' \to .S, \$ \\ S \to .abSa, \$ \\ S \to .aaAc, \$ \\ S \to .b, \$ \end{Bmatrix} - (I_0)$$

$$\text{goto}(I_0, S) = \{S' \to S., \$\} - (I_1)$$

$$\text{goto}(I_0, a) = \begin{Bmatrix} S \to .abSa, \$ \\ S \to a.aAc, \$ \end{Bmatrix} - (I_2)$$

$$\text{goto}(I_0, b) = \{S \to b., \$\} - (I_3)$$

$$\text{goto}(I_2, a) = \begin{Bmatrix} S \to aa.Ac, \$ \\ A \to .dAb, c \\ A \to .b, c \end{Bmatrix} - (I_5)$$

$$\text{goto}(I_2, b) = \begin{Bmatrix} S \to ab.Sa, \$ \\ S \to .abSa, a \\ S \to .aaAc, a \\ S \to .b, a \end{Bmatrix} - (I_4)$$

$$\text{goto}(I_4, S) = \{S \to abS.a, \$\} - (I_6)$$

$$\text{goto}(I_4, a) = \begin{Bmatrix} S \to a.bSa, a \\ S \to a.aAc, a \end{Bmatrix} - (I_7)$$

$$\text{goto}(I_4, b) = \{b., a\} - (I_8)$$

$$\text{goto}(I_5, A) = \{S \to aaA.c, \$\} - (I_9)$$

$$\text{goto}(I_5, d) = \begin{Bmatrix} A \to d.Ab, c \\ A \to .dAb, c \\ A \to .b, c \end{Bmatrix} - (I_{10})$$

$$\text{goto}(I_5, b) = \{A \to b., c\} - (I_{11})$$

$$\text{goto}(I_6, a) = \{S \to abSa., \$\} - (I_{12})$$

$$\text{goto}(I_7, b) = \begin{Bmatrix} S \to ab.Sa, a \\ S \to .abSa, a \\ S \to .aaAc, a \\ S \to .b, a \end{Bmatrix} - (I_{13})$$

$$\text{goto}(I_7, a) = \begin{Bmatrix} S \to aa.Ac, a \\ A \to .dAb, c \\ A \to .b, c \end{Bmatrix} - (I_{14})$$

$$\text{goto}(I_9, c) = \{S \to aaAc., \$\} - (I_{15})$$

$$\text{goto}(I_{10}, A) = \{A \to dA.b, c\} - (I_{16})$$

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					
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_4 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 :

$S' \to S$

$S \to Aa | aAc | Bc | bBa$

$A \to d$

$B \to d$

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

$B \to bDAc$

$D \to Dd ; / \epsilon$

$A \to A ; E / \epsilon$

$E \to B / a$

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 :

$S \to aIJh$

$I \to IbSe/c$

$J \to KLKr/\epsilon$

$K \to d/\epsilon$

$L \to p/\epsilon$

CS : S-12(14M), S-13(13M)

Ans.

Step 1 : Augmented grammar :

$S' \to .S$ ... (I)

$S \to .aIJh$ ... (II)

$I \to .IbSe$ ... (III)

$I \to .c$ ... (IV)

$J \to .KLKr$ ... (V)

$J \to .\epsilon$ ... (VI)

$K \to .d$ ... (VII)

$K \to .\epsilon$ ... (VIII)

$L \to .P$ ... (IX)

$L \to .\epsilon$ ... (X)

Step 2 : Canonical states for grammar :

$I_0 = \text{Closure}(\{S' \to .S, \$\})$

$$= \begin{Bmatrix} S' \to .S, \$. \\ S \to .aIJh, \$. \end{Bmatrix} - (I_0)$$

$$\text{goto}(I_0, S) = \{S' \to S; \$\} - (I_1)$$

$$\text{goto}(I_0, a) = \begin{Bmatrix} S \to a.IJh, \$ \\ I \to .IbSe, d/p/r/h \\ I \to .c, d/p/r/h \end{Bmatrix} - (I_2)$$

$$\text{goto}(I_2, I) = \begin{Bmatrix} S \to aI.Jh, \$ \\ J \to .KLKr, h \\ J \to ., h \\ K \to .d, d/p/r \\ K \to ., d/p/r \\ I \to I.bSe, d/p/r/h \end{Bmatrix} - (I_3)$$

$$\text{goto}(I_2, c) = \{I \to c., d/p/r/h\} - (I_4)$$

$$\text{goto}(I_3, J) = \{S \to aIJ.h, \$\} - (I_5)$$

$$\text{goto}(I_3, K) = \begin{Bmatrix} J \to K.LKr, h \\ L \to .p, d/r \\ L \to ., d/r \end{Bmatrix} - (I_6)$$

$$\text{goto}(I_3, d) = \{K \to d., d/p/r\} - (I_7)$$

$$\text{goto}(I_3, b) = \begin{Bmatrix} I \to Ib.Se, d/p/r/h \\ S \to .aIJh, e \end{Bmatrix} - (I_8)$$

$$\text{goto}(I_5, h) = \{S \to aIJh.\$\} - (I_9)$$

$$\text{goto}(I_6, L) = \begin{Bmatrix} J \to KL.Kr, h \\ K \to .d, r \\ K \to ., r \end{Bmatrix} - (I_{10})$$

$$\text{goto}(I_6, p) = \{L \to p., d/r\} - (I_{11})$$

$$\text{goto}(I_8, a) = \begin{Bmatrix} S \to a.IJh, e \\ I \to .IbSe, d/p/r/h \\ I \to .c, d/p/r/h \end{Bmatrix} - (I_{13})$$

$$\text{goto}(I_8, S) = \{I \to IbS.e, d/p/r/h\} - (I_{12})$$

$$\text{goto}(I_{10}, K) = J \to KLK.r, h - (I_{14})$$

$$\text{goto}(I_{10}, d) = K \to d., r - (I_{15})$$

$$\text{goto}(I_{12}, e) = \{I \mapsto IbSe., d/p/r/h\} - (I_{16})$$

$$\text{goto}(I_{13}, I) = \begin{Bmatrix} S \to aI.Jh, e \\ I \to I.bSe, d/p/r/h \\ J \to .KLKr, h \\ J \to ., h \\ K \to .d, d/p/r \\ K \to ., d/p/r \end{Bmatrix} - (I_{17})$$

$$\text{goto}(I_{13}, c) = (I_4)$$

$$\text{goto}(I_{14}, r) = \{J \to KLKr., h\} - (I_{18})$$

$$\text{goto}(I_{17}, b) = (I_8)$$

$$\text{goto}(I_{17}, d) = (I_7)$$

$$\text{goto}(I_{17}, J) = (S \to aIJ.h, e) - (I_{19})$$

goto $(I_{17}, k) = (I_6)$

goto $(I_{19}, h) = S \rightarrow aIJh \cdot , e = (I_{20})$

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.

VBD

Q.52 Construct LALR(1) parsing table for grammar :

S → L/a

L → wGdS/dSwG

G → b

CS : S-14/15,M)

Ans.

Step 1 : Augmented grammar :

S' → .S ... (I)

S → .L ... (II)

S → .a ... (III)

L → .wGdS ... (IV)

L → .dSwG ... (V)

G → .b ... (VI)

Step 2 : Canonical set for given grammar :

$I_0 = \{S' \to .S, \$\}$

$$= \left\{ \begin{array}{c} S' \to .S, \$ \\ S \to .L, \$ \\ S \to .a, \$ \\ L \to .wGdS, b \\ L \to .dSwG, a/w/d \end{array} \right\} - (I_0)$$

goto $(I_0, S) = \{S' \to S; \$.\} - (I_1)$

goto $(I_0, L) = \{S \to L., \$\} - (I_2)$

goto $(I_0, d) = \left\{ \begin{array}{c} S \to d., \$ \\ L \to d.SwG, a/w/d \end{array} \right\} - (I_3)$

goto $(I_0, w) = \left\{ \begin{array}{c} L \to w.GdS, b \\ G \to .b, d \end{array} \right\} - (I_4)$

goto $(I_3, S) = \{L \to dS.wG, a/w/d\} - (I_5)$

goto $(I_3, L) = \{S \to L., w\} - (I_6)$

goto $(I_3, d) = \{S \to a., w\} - (I_7)$

goto $(I_4, G) = \{L \to wG.ds, b\} - (I_8)$

goto $(I_4, b) = \{G \to b., d\} - (I_9)$

goto $(I_5, w) = \left\{ \begin{array}{c} L \to dSw.G, a/w/d \\ G \to .b, a/w/d \end{array} \right\} - (I_{10})$

goto $(I_8, d) = \left\{ \begin{array}{c} L \to wGd.S, b \\ S \to .L, \$ \\ S \to .a, \$ \end{array} \right\} - (I_{11})$

goto $(I_6, P) = \{L \to P., d/r\} - (I_{11})$

goto $(I_{10}, G) = \{L \to dSwG., a/w/d\} - (I_{12})$

goto $(I_{10}, b) = \{G \to b., a/w/d\} - (I_{13})$

goto $(I_{11}, S) = \{L \to wGdS., b\} - (I_{14})$

goto $(I_{11}, L) = \{S \to L., \$\}$ same as $- (I_2)$

goto $(I_{11}, d) = \{S \to a., \$\} - (I_{15})$

As we see the lookaheads and states output,

(I_2) and $(I_6) \to I_{26}$

(I_7) and $(I_{15}) \to I_{715}$

(I_9) and $(I_{13}) \to I_{913}$

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			

VBD

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 :**

Add every item in I to closure (I)

(1)

(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 :

$$I_0 = \text{Closure}\{(S' \rightarrow .S)\} = \begin{Bmatrix} S' \rightarrow .S \\ S \rightarrow .L = R \\ S \rightarrow .R \\ L \rightarrow .*R \\ L \rightarrow .id \\ R \rightarrow .L \end{Bmatrix}$$

goto $(I_0, S) = \{S' \rightarrow S.\} - (I_1)$

goto $(I_0, L) = \begin{Bmatrix} S \rightarrow L. = R \\ R \rightarrow L. \end{Bmatrix} - (I_2)$

goto $(I_0, R) = \{S \rightarrow R.\} - (I_3)$

goto $(I_0, *) = \begin{Bmatrix} L \rightarrow *. R \\ R \rightarrow .L \\ L \rightarrow .id \\ L \rightarrow .*R \end{Bmatrix} - (I_4)$

goto $(I_0, id) = \{L \rightarrow id.\} - (I_5)$

goto $(I_2, =) = \begin{Bmatrix} S \rightarrow L = .R \\ R \rightarrow L. \end{Bmatrix} - (I_6)$

goto $(I_4, R) = \{L \rightarrow *R.\} - (I_7)$

goto $(I_4, L) = \{R \rightarrow L.\} - (I_8)$

goto $(I_4, *) = \{L \rightarrow *.R\} - (I_4)$

goto $(I_4, id) = \{L \rightarrow id.\} - (I_5)$

goto $(I_6, R) = \{S \rightarrow L = R.\} - (I_9)$

goto $(I_6, L) = \{R \rightarrow L.\} - (I_8)$

goto $(I_6, *) = \{L \rightarrow *R.\}$ – same as (I_4)

goto $(I_6, id) = \{L \rightarrow id.\}$ – 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_7			R_3			
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_1			R_1			
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)

VBD

Q.56. 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 $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha AW \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 W$. 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.57. What is ambiguous grammar? Check whether the given CFG is ambiguous or not?

CT : W-12(4M), S-12(3M)

$S \rightarrow aSbS|bSaS|\epsilon$

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 :

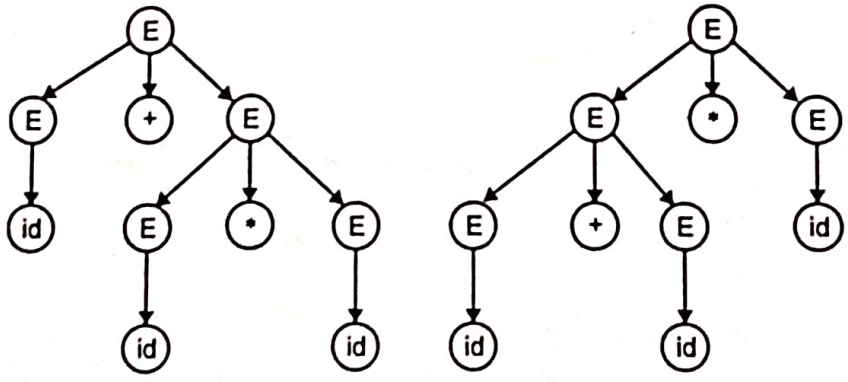


Fig. (a) Multiple parse tree

- As for the string id + id * id, more than one parse tree is created, so the given grammar is ambiguous.

Given grammar :

S → aSbS / bSaS / ∈

Let w = abab

Using right derivation,

S → aSbS	
S → aSb∈	/∵ S → ∈/
S → aSb	
S → abSaSb	/∵ S → bSaS/
S → ab∈a∈b	/∵ S → ∈/
S → abab	

S → aSbS	
S → aSbaSbS	/∵ S → aSbS/
S → aSbaSb∈	/∵ S → ∈/
S → aSba∈b	/∵ S → ∈/
S → a∈bab	/∵ S → ∈/
S → abab	

∴ There arises more than one parse tree for string w = abab, hence the given grammar is ambiguous.

The parse tree is as follows :

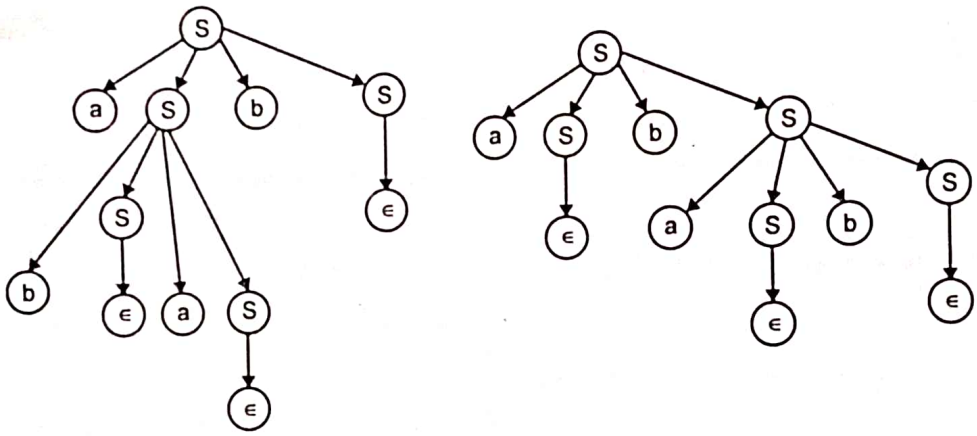


Fig.(b) Multiple parse tree for string ababa

Q.98. Given grammar

S → a / abSb / aAb

A → bS / aAAb

Check whether the grammar is ambiguous or not.

CT : S-14(5M)

Ans. Consider the string

w → abab.

The rules are :

S → a ..... (I)

S → abSb ..... (II)

S → aAb ..... (III)

A → bS ..... (IV)

A → aAAb ..... (V)

Using left most derivation :

S → ab<u>S</u>b	S → a<u>A</u>b
S → abab /∵ S → a/	S → abSbb /∵ A → bS/
	S → abab /∵ S → a/

- Here string w = abab can be generated in more than one way using leftmost derivation, so grammar is ambiguous.

VBD

Thus derivation tree for these two approaches are :

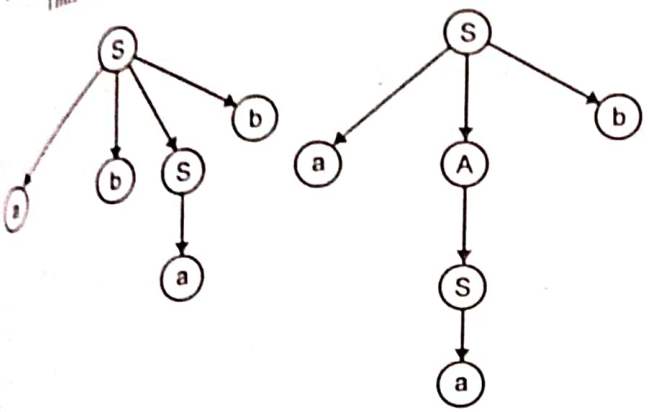


Fig. Multiple parse tree for string abab

PARSER GENERATOR

Q.99. Explain YACC in detail.

OR Write a note on YACC. **CS:W-14(5M)**

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.

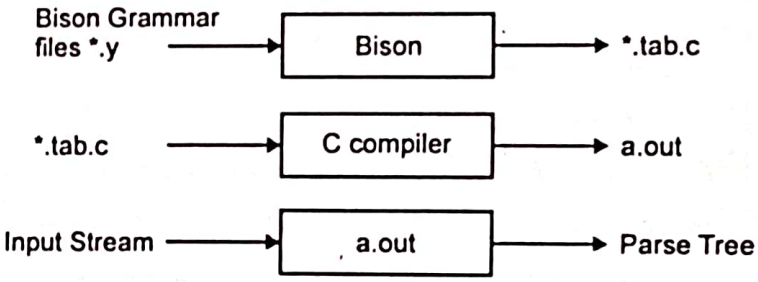


Fig. Bison parser

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.

APPENDIX - II

UNIT - I

RELATING REGULAR EXPRESSION AND FINITE AUTOMATA

Q.1. Explain the relationship between Finite Automata and Regular Expression.

Ans. Regular Expression (RE):

Regular expression is the language which is used to describe the language and is accepted by finite automata. Regular expressions are the most effective way to represent any language. Let Σ be an alphabet which denotes the input set.

The regular expression over Σ can be defined as follows:

(i) Φ is a regular expression which denotes the empty set.

(ii) ε is a regular expression and denotes the set {ε} and it is called a null string.

(iii) For each 'a' in Σ 'a' is a regular expression and denotes the set {a}.

(iv) If r and s are regular expressions denoting the language. L_1 and L_2 respectively then,

- r + s is equivalent to $L_1 \cup L_2$ union
- rs is equivalent to $L_1 L_2$ concatenation
- r* is equivalent to L_1 * closure

The r* is known as Kleen closure or closure which indicates occurrence of r for an infinite number of times.

Finite Automata (FA):

Finite automata is an abstract computing device. It is a mathematical model of a system with discrete inputs, outputs, states and a set of transitions from state to state that occurs on input symbols from the alphabet Σ.

Finite automata is defined as a 5-tuples

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

Where,

Q: Finite set called states.

Σ: Finite set called alphabets.

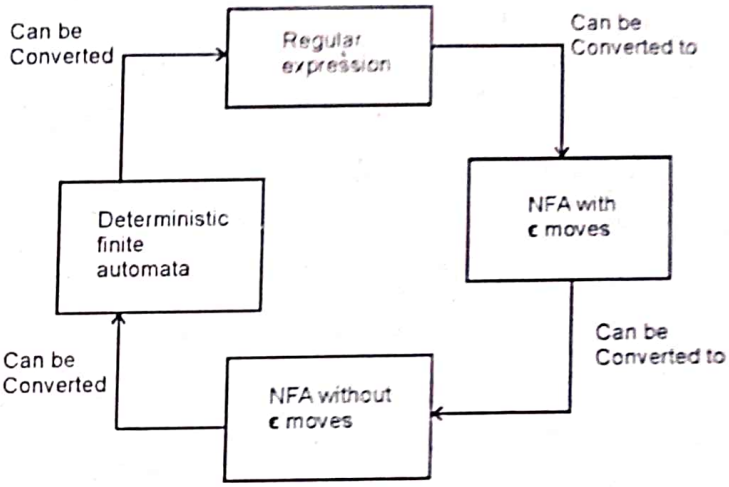
$\delta: Q \times \Sigma \rightarrow Q$ is the transition function.

$q_0 \in Q$ is the start or initial state.

F: Final or accept state.

Relationship:

The relationship between FA and RE is as follows:

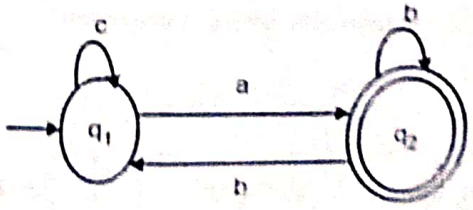


The above figure explains that it is easy to convert

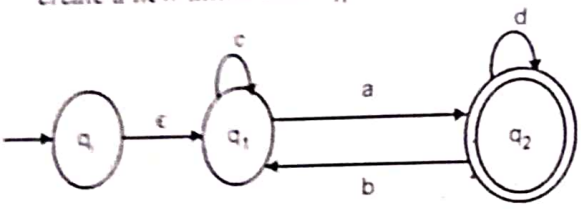
- RE to Non-deterministic finite automata (NFA) with epsilon moves.
- NFA with epsilon moves to without epsilon moves.
- NFA without epsilon moves to Deterministic Finite Automata (DFA).
- DFA can be converted easily to RE.

Q.2. Generate a Regular Expression for a Finite Automata with state elimination method.

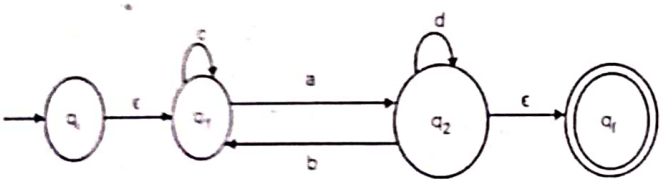
Ans. Conversion of Finite Automata into equivalent Regular Expression using the State Elimination Method is explained below:



Step 1: Initial state q_1 has incoming edge. So, create a new initial state q_i.



Step 2: Final state q_2 has outgoing edge. So, create a new final state.



Step 3: Start eliminating intermediate states one after another.

If there is path going to q_i to q_2 via q_1

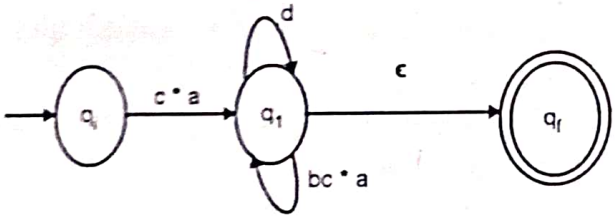
So after eliminating q_1 we can connect a direct path from q_i to q_2 having cost

$\varepsilon . c^* . a = c^* a$

There is a loop on q_2 using state q_i

So after eliminating state q_1 we can draw a direct loop on state q_2 having cost

$b.c^*.a = b.c^*.a$



Step 4: Now eliminate q_2.

After eliminating q_2 direct path from state q_i to q_f having cost.

$c^*a(d + bc^*a)^* \varepsilon = c^*a(d + bc^*a)^*$

Q.3. Can we convert non-deterministic finite automata into deterministic finite Automata?

Ans.

- Yes, we can convert a NFA into DFA. For every NFA there exists an equivalent DFA. The equivalence is defined in terms of languages acceptance. Since NFA is nothing but a finite automata in which zero, one or more transitions on an input symbols are permitted, it can always construct finite automata which will simulate all moves of DFA on a particular input symbol in parallel, then get a finite automata in which there will be exactly one transition on every input symbol. Here, corresponding to a NFA there exist a DFA.
- To construct DFA equivalent to NFA, it should be remembered that states of DFA are a collection of states of NFA.

Algorithm NFA - to - DFA:

Input: NFA with set of states

$N = \{n_0, n_1 ..... n_n\}$, with start state n_0.

Output: DFA, with set of states $D' = \{d_0, d_1, d_2 ..... d_n\}$, with start state d_0.

$d_0 = \varepsilon - \text{closure}(n_0)$

$D' = \{d_0\}$

set d_0 unmarked

while there is an unmarked state d in D'. {

  set d marked {

  For each input symbol 'a' {

    Let T be a set of states in NFA to which there is a transition on 'a' from some state n_i in d d' = ε-closure (T).

    If d' is not already present in D' {

      D'=D' U {d'}

      Add transition d→d',labeled 'a'

      set d' unmarked

      }

    }

  }

}

Q.4. Design Lexical Analyzer for the following LEX Program:

AUXILIARY DEFINITIONS

letter = A|B|C|.....|Z

digit = 0 |1|2|.....|9

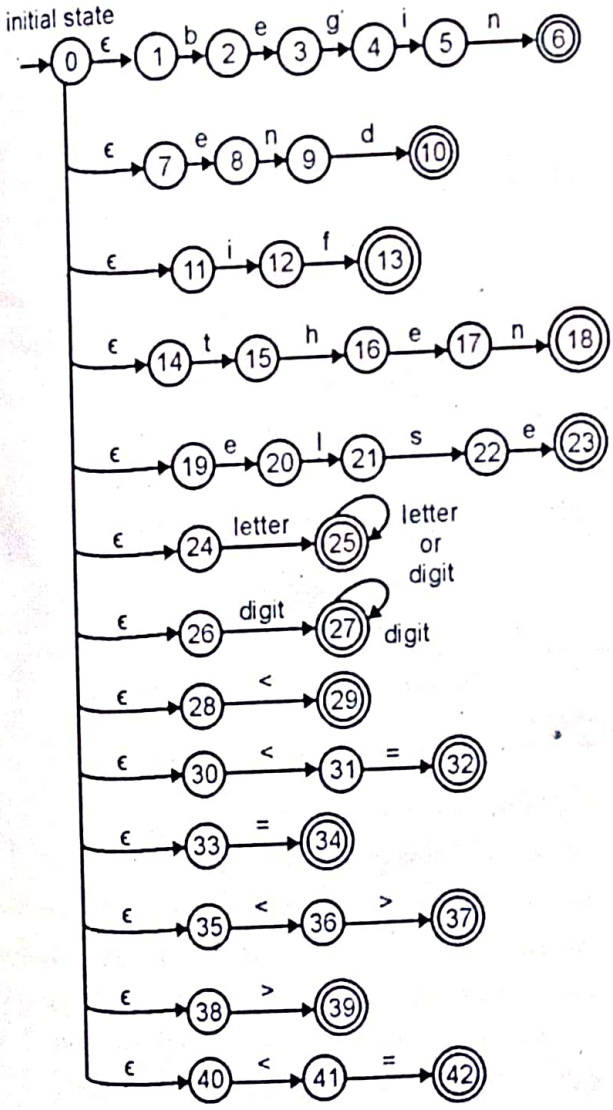
VBD

TRANSLATION RULES

begin {return 1}
end {return 2}
If {return 3}
then {return 4}
else {return 5}
letter (letter+digit)* {value=Install ( );return 6}
digit + {value=Install ( );return 7}
< {value=1;return 8}
<= {value=2;return 8}
= {value=3;return 8}
<> {value=4;return 8}
> {value=5;return 8}
>= {value=6;return 8}

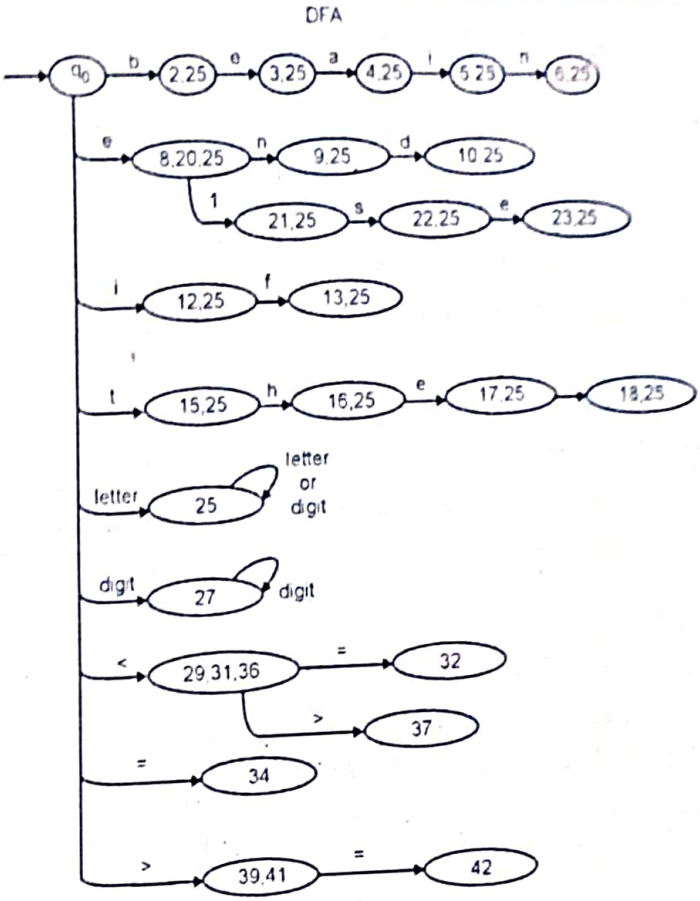
Ans. The combined NFA for various patterns will be

Combined NFA



Convert NFA to DFA – The corresponding DFA will be

DFA



States {0, 1, 7, 11, 14, 19, 24, 26, 28, 30, 33, 35, 38, 40} are combined and named as q_0 to make a starting state of DFA.

In combined NFA, transitions from state 1 to 2 and from state 24 to 25 are the same because Input 'b' is a letter.

States 2, 25 are combined. Similarly, other states are combined.

States 29, 31 and 36 are combined because they all reach after getting the input '<'.

Similarly, other states are combined.

UNIT - II

PUSHDOWN AUTOMATA

Q.5. Explain Pushdown Automata.

Ans. Pushdown Automata:

- A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

A pushdown automaton is:
"Finite state machine" + "a stack"

- A pushdown automaton has three components:

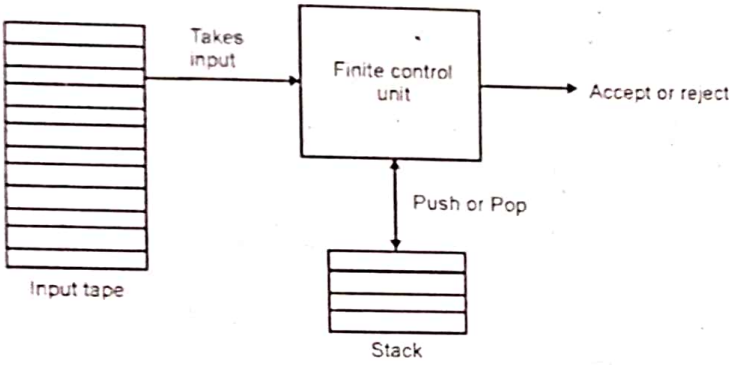
(i) An input tape,
(ii) A control unit, and
(iii) A stack with infinite size.

The stack head scans the top symbol of the stack.

- A stack does two operations:

(i) **Push:** A new symbol is added at the top.
(ii) **Pop:** The top symbol is read and removed.

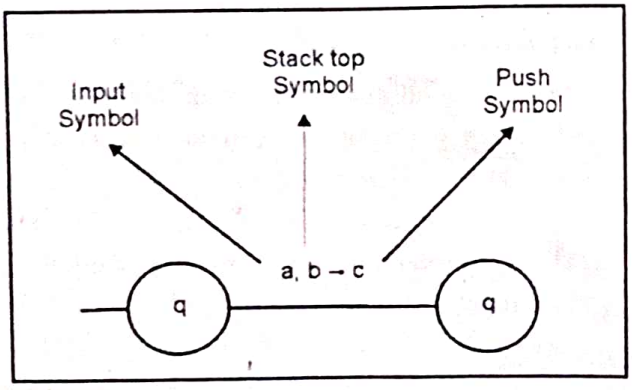
- A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



- A PDA can be formally described as a 7-tuple $(Q, \sum, S, \delta, q_0, I, F)$:

where
Q - The finite number of states
$\sum$ - Input alphabet
S - Stack symbols
δ - The transition function:
$Q \times (\sum \cup \{\varepsilon\}) \times S \times Q \times S^*$
q_0 - The initial state $(q_0 \in Q)$
I - The initial stack top symbol $(I \in S)$
F - A set of accepting states $(F \in Q)$)

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as a,b → c:



This means at state q_1, if we encounter an input string 'a' and top symbol of the stack is 'b', then we pop 'b', push 'c' on top of the stack and move to state q_2.

Terminologies Related to PDA:

(i) **Instantaneous Description:**

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where
q - is the state
w - is unconsumed input
s - is the stack contents

(ii) **Turnstile Notation:**

- The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol " ⊢ ".
- Consider a PDA $(Q, \sum, S, \delta, q_0, I, F)$. A transition can be mathematically represented by the following turnstile notation:
$(p, aw, T\beta) \vdash (q, w, \alpha b)$
- This implies that while taking a transition from state p to state q, the input symbol 'a' is consumed and the top of the stack 'T' is replaced by a new string 'α'.

Note: If we want zero or more moves of a PDA, we have to use the symbol ($\vdash^*$) for it.

UNIT - III

SEMANTIC ANALYSIS: ATTRIBUTE GRAMMER

Q.6. Explain Semantic Analysis.

Ans. Semantic Analysis:

- Semantic analysis is the third phase of Compiler. Semantic analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.

VBD

- Type checking is an important part of semantic analysis where compiler makes sure that each operator has matching operands.

Semantic Analyzer:

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Semantic Errors:

Errors recognized by semantic analyzer are as follows:

(a) Type mismatch
(b) Undeclared variables
(c) Reserved identifier misuse

Functions of Semantic Analysis:

(1) **Type Checking:**
Ensures that data types are used in a way consistent with their definition.

(2) **Label Checking:**
A program should contain labels references.

(3) **Flow Control Check:**
Keeps a check that control structures are used in a proper manner. (example: no break statement outside a loop).

Example:

float x = 10.1;
float y = x*30;

In the above example integer 30 will be type casted to float 30.0 before multiplication, by semantic analyzer.

Static and Dynamic Semantics:

(1) **Static Semantics:**
It is named so because of the fact that these are checked at compile time. The static semantics and meaning of program during execution, are indirectly related.

(2) **Dynamic Semantic Analysis:**
It defines the meaning of different units of program like expressions and statements. These are checked at runtime unlike static semantics.

RUN-TIME ENVIRONMENT, PROCEDURE ACTIVATION

Q.7. Explain Run-Time Environment.

Ans. Run-Time Environment:

- A Program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.
- By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.
- Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Q.8. Explain Activation Trees.

OR Explain Procedure Activation.

Ans. Activation Trees:

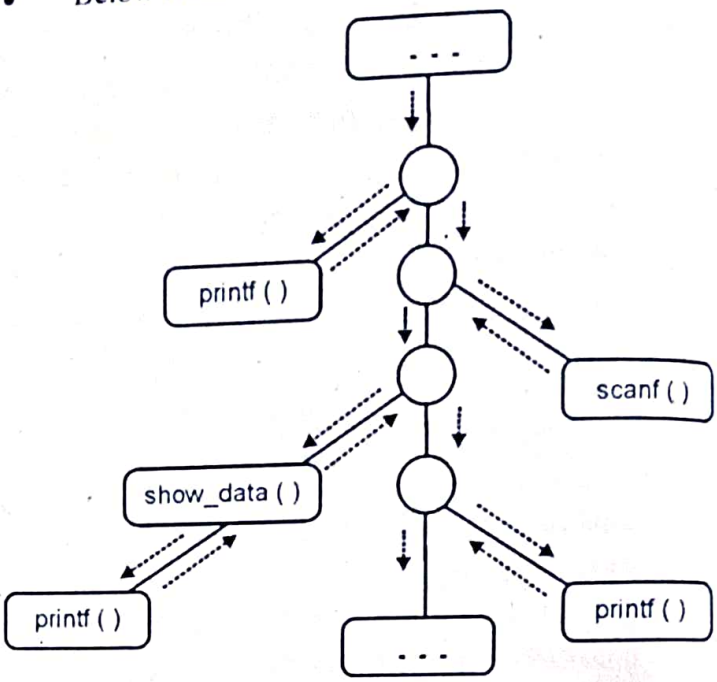
- A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.
- The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e. parameters which are used to send input to the called procedure.
Return Value	Stores return values.

- Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.
- We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.
- To understand this concept, we take a piece of code as an example:

```
. . .
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue...");
. . .
int show_data(char *user)
{
    printf("Your name is %s", username);
    return 0;
}
. . .
```

- Below is the activation tree of the code given.



- The procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

PARAMETER PASSING, VALUE RETURN

Q.9. Explain Parameter Passing.

Ans. Parameter Passing:

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

(i) r-value:

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

(ii) **l-value:**

- The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

- From this example, we understand that constant values like 1, 7, 12 and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

(1) **Formal Parameters:**

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

(2) **Actual Parameters:**

- Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
    int actual_parameter = 10;
    call fun_two(int actual_parameter);
}
fun_two(int formal_parameter)
{
    print formal_parameter;
}
```

- Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

(a) **Pass by Value:**

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

(b) **Pass by Reference:**

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

(c) **Pass by Copy-restore:**

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
   y = 10;
   copy_restore(y); //l-value of y is passed
   printf y; //prints 99
}
copy_restore(int x)
{
   x = 99; // y still has value 10 (unaffected)
   y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

(d) Pass by Name:

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

MEMORY ALLOCATION STORAGE ALLOCATION

Q.10. Explain Storage Allocation.

OR Explain Storage Allocation. What are the different Storage Allocations?

Ans. Storage Allocation:

Runtime environment manages runtime memory requirements for the following entities:

(i) Code:

It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

(ii) Procedures:

Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.

(iii) Variables:

Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Types of Allocation:

(1) Static Allocation

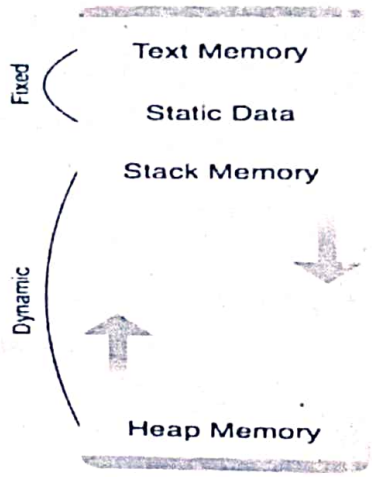
In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

(2) Stack Allocation:

Procedure calls and their activations are managed by means of stack memory allocation. It works in Last-In-First-Out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

(3) Heap Allocation

- Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



- As shown in the above Fig. the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Q.11. Explain Memory allocation.

OR What are the techniques of storage allocation in compiler design?

Ans. There are various storage allocation techniques which are as follows:

VBD

(1) **Static Allocation:**

- It is the simplest allocation scheme in which allocation of data objects is done at compile time because the size of every data item can be determined by the compiler.
- Recursive Subprogram and Arrays of adjustable length are not permitted in a language. In static allocation, the compiler can decide the amount of storage needed by each data object. Thus, it becomes easy for a compiler to identify the address of these data in the activation record.
- FORTRAN uses this kind of storage allocation strategies.

Advantages:

(i) It is easy to implement.

(ii) It allows type checking during compilation.

(iii) It eliminates the feasibility of running out of memory.

Disadvantages:

(i) It is incompatible with recursive subprograms.

(ii) It is not possible to use variables whose size has to be determined at run time.

(iii) The static allocation can be completed if the size of the data object is called compile time.

(2) **Dynamic Allocation (Stack Allocation):**

- The stack allocation is a runtime storage management technique. The activation records are pushed and popped as activations begin and end respectively.
- Storage for the locals in each call of the procedure is contained in the activation record for that call. Thus, locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when the call is made.
- It can be determined the size of the variables at a run time & hence local variables can have different storage locations & different values during various activations.
- It allows recursive subprograms.
- ALGOL language uses this strategy.
- On each execution of a procedure, an Activation Record is generated, which contains information like local data, actual parameter, return value, return address of a procedure. The Activation Record for that procedure is saved onto the stack.

Advantages:

(i) It supports recursion.

(ii) It creates a data structure for the data item dynamically.

Disadvantage:

Memory Addressing can be done using pointers and index Registers.

(3) **Heap Storage Allocation**

- It enables the allocation of memory in a non-nested design. Storage can be allocated and freed arbitrarily from an area known as Heap.
- Heap Allocation is helpful for executing data whose size varies as the program is running.
- Heap is maintained as a list of free space called free space list.

Advantage:

A large block of storage can be partitioned into smaller blocks at run time.

Disadvantage:

It creates the problem of fragmentation.

Q.12. Differentiate between Stack allocation and Heap allocation.

Ans.

Sr. No.	Static allocation	Heap allocation
(1)	Static allocation allocates memory on the basis of size of data objects.	Heap allocation makes use of heap for managing the allocation of memory at run time.
(2)	In static allocation, there is no possibility of creation of dynamic data structures and objects.	In heap allocation, dynamic data structures and objects are created.
(3)	In static allocation, the names of the data objects are fixed with storage for addressing.	Heap allocation allocates contiguous block of memory to data objects.

(4)	Static allocation is simple, but not efficient memory management technique.	Heap allocation does memory management in efficient way.
(5)	Static allocation strategy is faster in accessing data as compared to heap allocation.	Heap allocation is slow in accessing as there is chance of creation of holes in reusing the free space.
(6)	It is easy to implement.	It is comparatively expensive to implement.

UNIT - IV

GLOBAL OPTIMIZATION

Q.22. Explain Global Optimization.

Ans. Global Optimization:

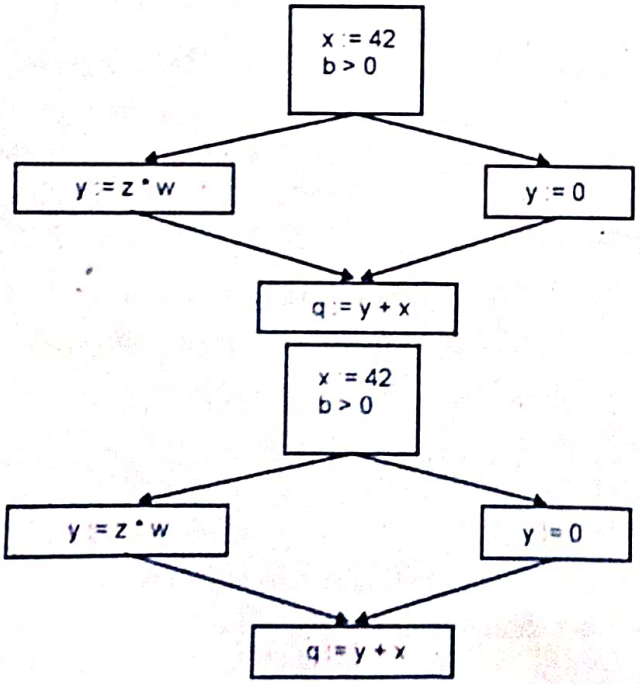
The scope of Global optimization is generally a program unit, usually spanning over number of basic blocks.

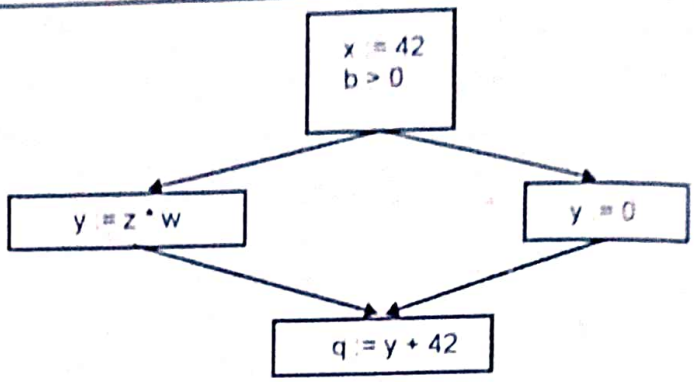
- It can achieved better transformation leading to more optimized.
- Program flow graph normally undergoes two types analysis:

(a) Control Flow Analysis.

(b) Data Flow Analysis.

- These optimizations can be extended to an entire control-flow graph.





UNIT - V

ARCHITECTURE DEPENDENT CODE IMPROVEMENT IN COMPILER DESIGN: INSTRUCTION SCHEDULING

Q.23. Explain Architecture dependent code improvement in compiler design: Instruction Scheduling.

OR Explain Code Optimizing Process.

OR What is Code Optimization?

Ans. Code Optimization:

- Code optimization is a program transformation approach that aims to enhance code by reducing resource consumption (i.e. CPU and memory) while maintaining high performance.
- In code optimization, high-level generic programming structures are substituted with low-level programming codes. The three guidelines for code optimization are as follows:

(i) In no way should the output code alter the program's meaning.

(ii) The program's speed should be increased, and it should use fewer resources if at all feasible.

(iii) The optimization step should be quick and not hinder the compilation process.

- At several stages of the compilation process, efforts to optimize the code might be made. Users can alter/rearrange the code at first or create the code using better algorithms. The compiler can improve loops and address computations after completing intermediate code. The compiler can leverage Memory Hierarchy and CPU registers while generating the target machine code.

Q.24. Why Code Optimization is necessary?

Ans. Code Optimization:

- Optimizing code in compiler design is important because it directly affects the performance of the compiled code.
- A well-optimized code runs faster and consumes fewer resources, leading to improved overall system performance and reduced energy consumption.
- Additionally, optimization can reduce the size of the generated code, which is important for embedded systems with limited memory.
- The optimization process can also help identify and eliminate bottlenecks in the code, leading to more efficient algorithms and improved software design. Overall, optimization is a critical step in the compiler design process that can greatly improve the end-user experience.

Q.25. Explain types of Code Optimization.

Ans. The code optimization process can be broadly classified into two types:

(i) Machine Independent Optimization

(ii) Machine Dependent Optimization

(i) Machine Independent Optimization:

This step of code optimization aims to optimize the intermediate code to produce a better target code. No CPU registers or absolute memory addresses are involved in the section of the intermediate code that is translated here.

(ii) Machine Dependent Optimization:

- After the target code has been created and converted to fit the target machine architecture, machine-dependent optimization is performed. It may use absolute memory references rather than relative memory accesses and requires CPU registers. Machine-dependent optimizers make a concerted attempt to maximize the memory hierarchy's benefits.
- A code has several statements, loops, branches, etc. So code optimization must be performed on all of them. The code optimization is done differently, considering the following:

(1) Loop Optimization:

The majority of programs in the system operate in a loop. It is vital to optimize the loops to save CPU cycles and memory. The following strategies can be used to improve loops.

(i) Loop-invariant Code:

It is a piece of code that sits in the loop and computes the same value each time an iteration is performed. This code may be moved out of the loop by storing it to be calculated just once rather than with each iteration.

(ii) Induction Analysis:

If a loop-invariant value changes the value of a variable within the loop, it is termed an induction variable.

(iii) Strength Reduction:

Some expressions use more CPU cycles, time, and memory than others. These expressions should be replaced with less expensive expressions without sacrificing the expression's output. For example, multiplication (x * 2) uses more CPU cycles than (x << 1) but produces the same output.

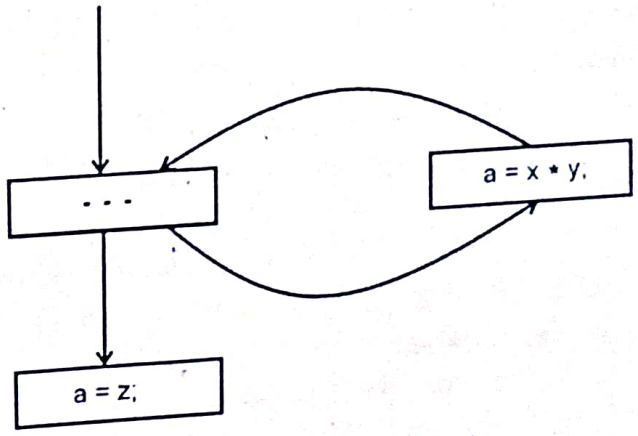


Fig.1. Loop Code flow chart

(2) Partially Dead Code:

- Some code statements include calculated values utilized only in particular conditions, i.e. the values are used sometimes and not others. Partially dead-code refers to such codes.
- The control flow diagram below shows a program section in which the variable 'a is utilized to assign the output of the equation 'x * y'. Let's

pretend that the 'a variable's value is never utilized within the loop. 'a' is given the variable 'z' value, which will be utilized later in the program, immediately after the control leaves the loop. We may infer that because the assignment code 'a' is never utilized anywhere, it is suitable for deletion.

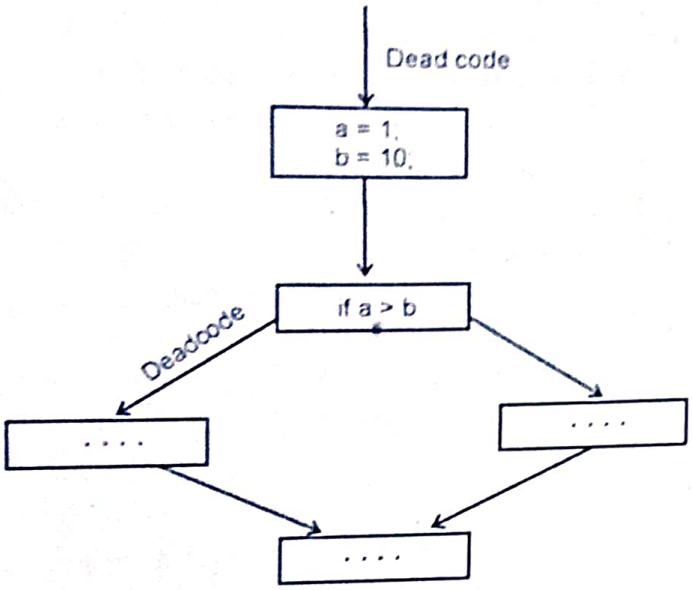


Fig.2. Dead Code flow chart

- Similarly, the conditional statement in the image above is always false, meaning that the code written in the "true" case will never be run and so may be eliminated.

Unreachable Code Elimination:

A control flow graph should be created first. An inaccessible code block does not have an incoming edge. The inaccessible branches can be deleted after continual propagation and folding.

(4) **Function Inlining:**

The body of the function takes the place of a function call. This saves a lot of time by eliminating the need to copy all parameters, store the return address, and so on.

(5) **Function Cloning:**

For different calling arguments, specialized codes for a function are constructed. Overloading a function is an example of this.

(6) **Partial Redundancy:**

In a parallel route, redundant expressions are calculated many times without changing the operands. Partial-redundant expressions, on the other hand, are calculated several times along a path without changing the operands. By employing a code-motion approach, loop-invariant code may be rendered largely redundant.

An example of a partially redundant code can be:

```
If (condition) {
    a = y OP z;
} else {
    ...
}
c = y OP z;
```

We assume that the operands' values (y and z) do not change when variable an is assigned to variable c. If the condition statement is true, y OP z is calculated twice; otherwise, it is computed once. As stated below, code motion may be utilized to remove redundancy:

```
If (condition) {
    ...
    tmp = y OP z;
    a = tmp;
    ...
} else {
    ...
    tmp = y OP z;
}
c = tmp;
```

y OP z should only be computed once, regardless of whether the condition is true or not.

TARGET CODE GENERATION

Q.16.Explain Target Code Generation.

Ans. Target Code Generation:

Target code generation is the final Phase of Compiler:

(i) **Input:** Optimized Intermediate Representation.

(ii) **Output:** Target Code.

(iii) **Task Performed:** Register allocation methods and optimization, assembly level code.