UNIT-II

Top-down Parsing

Objective:
To understand Top down parsing.

Syllabus:
Syntax analysis, role of a parser, classification of parsing techniques, top-down parsing techniques-recursive descent parsing, first and follow, LL(1) grammars, non-recursive predictive parsing.

Learning Outcomes:
Students will able to

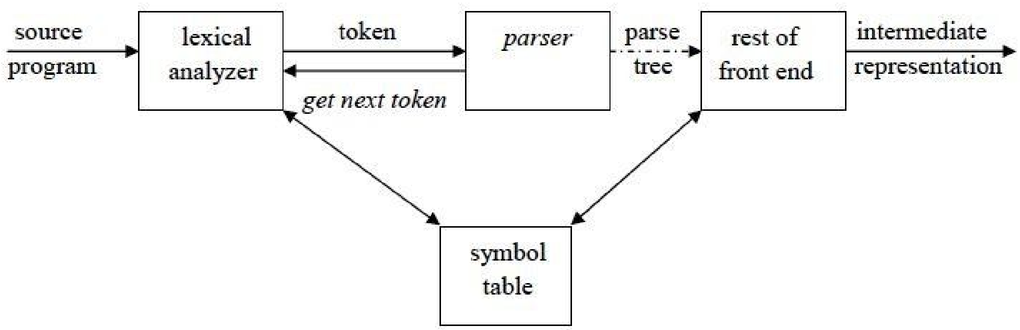
- explain the role of parser.
- calculate first and follow for the given grammar.
- construct predictive parsing table for a given grammar.

Learning Material

Syntax Analysis:

- Syntax analysis or parsing is the second phase of a compiler.
- Lexical analyzer can identify tokens with the help of regular expressions and pattern rules.
- The output of lexical analyzer is given as input to the syntax analysis phase.
- Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar which is recognized by push-down automata.
- CFG is a superset of regular grammar.

Role of the Parser:



- It verifies the structure generated by the tokens based on the grammar.

- It constructs the parse tree.
- It reports the errors.
- It performs error recovery.

Ambiguity in context free grammars:

A context-free grammar G such that some word has two parse trees is said to be ambiguous.

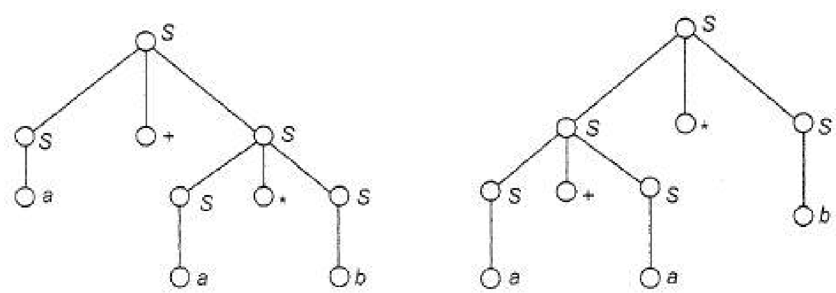
(or)

A word which has more than one leftmost derivation or more than one rightmost derivation is said to be ambiguous.

Example:

G = ({S}, {a, b, +, *}, P. S), where P consists of S →S+S | S*S | a | b

We have two derivation trees for *a* + *a* * *b*



Associativity:

- I
  f an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators.
- If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.
- Operations such as addition, multiplication, subtraction, and division are left associative. If the expression contains:
  id op id op id
  it will be evaluated as: (id op id) op id
  For example, (id + id) + id
- Operations like exponentiation are right associative, i.e., the order of evaluation in the same expression will be:
  id op (id op id)
  For example, id ^ (id ^ id)

Precedence:

- If two different operators share a common operand, the precedence of operators decides which will take the operand.
- As in the previous example, mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be

interpreted as:
2 + (3 * 4)

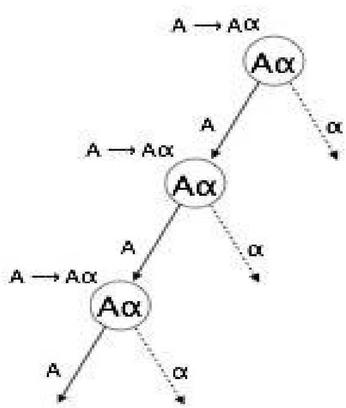
- These methods decrease the chances of ambiguity in a language or its grammar.

Left Factoring:

- Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.
- If a top-down parser encounters a production having common prefixes like
  $A \rightarrow \alpha\beta \mid \alpha y \mid \ldots$
- After Left factoring the above productions can be written as
  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta \mid y \mid \ldots$

Left Recursion:

- A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol.
- Left-recursive grammar is considered to be a problematic situation for top-down parsers.
- Productions of the form $A \rightarrow A\alpha \mid \beta$ is an example of left recursion.



Removal of Left Recursion:

The production $A \rightarrow A\alpha \mid \beta$ is converted into following productions
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

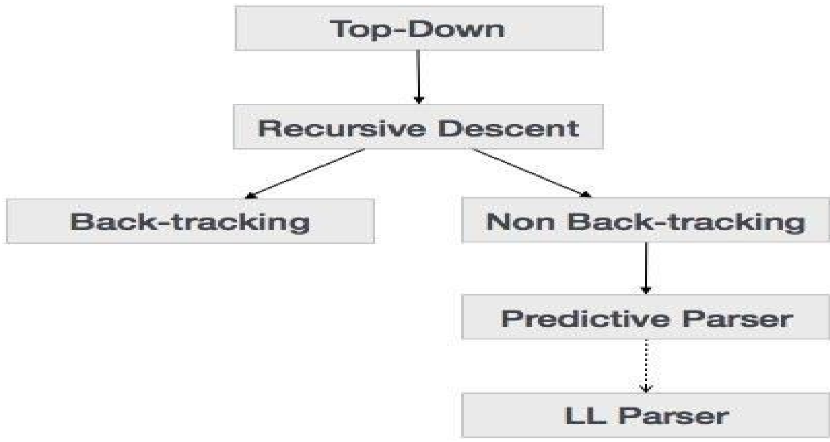
Classification of Parsers:

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types:

1. Top down parsing
2. Bottom up parsing

Top-down Parsing:

Top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes.



Types of top-down parsing:

1. Recursive descent parsing
2. Predictive parsing

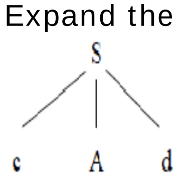
Backtracking:

- Backtracking is a technique in which if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production.
- This technique may process the input string more than once to determine the right production.

Example:
Consider the grammar G
$S \rightarrow cAd$
$A \rightarrow ab \mid a$
Input string w=cad.
The parse tree can be constructed using the following top-down approach:

Step1:

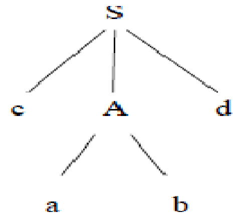
Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S.



Step2:

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

Expand A using the first alternative.



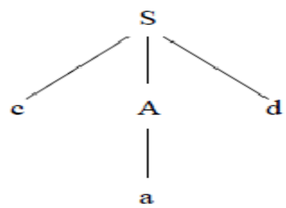
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called backtracking.

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Recursive descent parsing:

- It is a common form of top-down parsing.
- It is called recursive, as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- Elimination of left-recursion must be done before parsing.

Example:

Consider the grammar for arithmetic expressions

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE'

E' → +TE' |ε

T → FT'

T' → *FT' | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

Recursive procedures:

```
Procedure E( )
 begin
           T( );
           EPRIME( );

end
 Procedure EPRIME( )
 begin
         if input_symbol='+' then ADVANCE( );
         T( );
         EPRIME( );
         End
         end
         Procedure T( )
           begin
                 F( );
                 TPRIME( );
           end
         Procedure TPRIME( )
           begin
                 if input_symbol='*'
                 then ADVANCE( );
                     F( );
                 TPRIME( );
           end
         Procedure F( )
           begin
                 if input-symbol='id'
                 then ADVANCE( );
                 else if input-symbol='('
                 then ADVANCE( );
                 E( );
                 else if input-symbol=')'
                 then ADVANCE( );
           end
                 else ERROR();
```

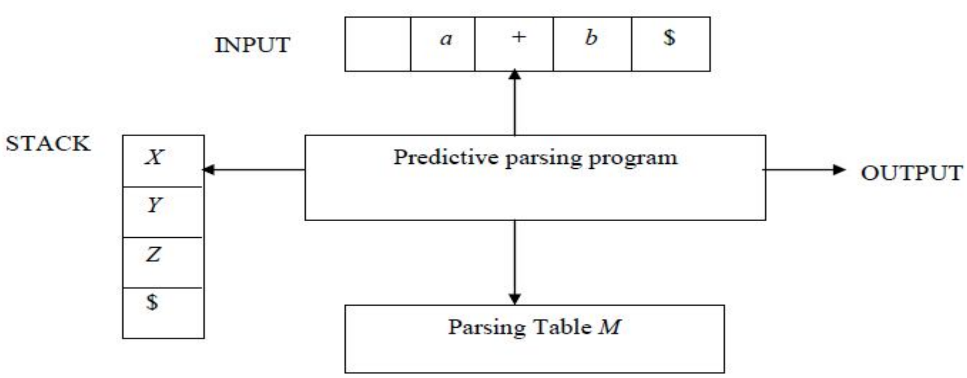
Stack implementation:

PROCEDURE	INPUT STRING
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id_id*id
TPRIME( )	id_id*id
EPRIME( )	id_id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id***id
ADVANCE( )	id+id*id
TPRIME( )	id+id*id
ADVANCE( )	id+id*id
F( )	id+id*id
ADVANCE( )	id+id***id**
TPRIME( )	id+id***id**

Predictive parsing:

- Predictive parsing is a special case of recursive descent parsing where nobacktracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser:



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array *M*[*A*, *a*], where **'*A*'** is a non-terminal and **'*a*'** is a terminal.

Algorithm for nonrecursive predictive parsing:

Input : A string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

set *ip* to point to the first symbol of *w*$;

repeat

let *X* be the top stack symbol and *a* the symbol pointed to by *ip*;

if *X* is a terminal or $

then **if** *X* = *a*

then

pop *X* from the stack and advance *ip*

else *error*()

else /* *X* is a non-terminal */

if *M*[*X*, *a*] = *X* →*Y1Y2* … *Yk* **then**

begin

pop *X* from the stack;

push *Yk*, *Yk-1*, … ,*Y1* onto the stack, with *Y1* on top;

output the production *X* → *Y1 Y2 . . . Yk*

end

else*error*()

until *X* = $

Predictive parsing program:

The parser is controlled by a program that considers *X*, the symbol on top of stack, and *a*, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If *X* = *a* = $, the parser halts and announces successful completion of parsing.
2. If *X* = *a* ≠ $, the parser pops *X* off the stack and advances the input pointer to the next input symbol.
3. If *X* is a non-terminal , the program consults entry *M*[*X*, *a*] of the parsing table *M*. This entry will either be an *X*-production of the grammar or an

error entry.

If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by UVW

If $M[X, a]$ =**error**, the parser calls an error recovery routine.

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G:

1. FIRST
2. FOLLOW

Rules for first ( ):

1. If X is terminal, then FIRST(X) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is non-terminal and $X \rightarrow a\alpha$ is a production then add a to FIRST(X).
4. If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST($Y1$),…,FIRST(Yi-1); that is, $Y1,\ldots.Yi\text{-}1 \Rightarrow \varepsilon$. If ε is in FIRST ($Y_j$) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow ( ):

1. If S is a start symbol, then FOLLOW(S) contains \$.
2. If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table: Input: Grammar G

Output: Parsing table M

Method:

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(α) and \$ is in FOLLOW(A) , add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

Example:

Consider the following grammar:

E → E+T | T

T→T*F | F

F → (E) | id

After eliminating left-recursion the grammar is

E→ TE’

E’→ +TE’ | ε

T → FT’

T’→ *FT’ | ε

F→(E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E’) ={+ ,ε}

FIRST(T) = { (, id}

FIRST(T’) = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E’) = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T’) = { +, $,) }

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

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’ → ε	T’ → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a

E → b

After eliminating left factoring, we have

S→iEtSS' | a

S'→eS |ε

E→b

Calculate FIRST () and FOLLOW ()

FIRST(S) = { i, a }

FIRST(S') = {e,ε}

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S') = {$,e}

FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

The grammar is not LL (1) grammar.

UNIT-II

Assignment-Cum-Tutorial Questions

A. Objective Questions

1. A CFG is ambiguous if ______________. [ ]
   a) the grammar contains useless non-terminals.
   b) It produces more than one parse tree for some word
   c) It produces more than one LMD or more than one RMD for some word
   d) Both b & c
2. The advantage of eliminating left recursion is _____________ [ ]
   a) Avoids back tracking c) avoids parser to go into an infinite loop
   b) Avoids ambiguity d) all the above.
3. Left factoring is compulsory in designing recursive descent parser? [True|False]
4. The no. of procedures to be defined in recursive descent parser depends on_______.
5. Can we design a recursive descent parser with ambiguous grammar? [True |False]
6. Recursive descent parser is a ______________parser.
7. LL(1) is top-down parser or bottom-up parser? [True |False]
8. Is every LL (1) grammar is unambiguous grammar? [True | False]
9. In LL(1),First L indicates [ ]
   a) Left to Right scanning of input
   b) Left Most Derivation
   c) Left recursion
   d) None of these
10. In LL (1), 1 indicates ____________.
11. Can every unambiguous grammar is parsed by LL (1)? [True | False]
12. The following grammar is [ ]

A→ AaB | a B→ aB | a

a) ambiguous b) unambiguous c) left recursive d) both b & c

13. Which of the following statements are correct? [ ]

G1: S→ S(S)S | ε is ambiguous

G2: S→+SS | *SS | a is ambiguous

a) Only statement 1 is true

b) Only statement 2 is true

c) Statement 1 is true and statement 2 is false

d) Both statements 1 and 2 are false.

14. A→A α | β is left recursive then its equivalent production are [ ]

a) A-> β R, R-> α R | ε b) A-> α R, R-> β R | ε

c) A-> α R | epsilon, R->β R | β d) None of these

15. Which of the following derivation does a top-down parser use while parsing an input string? The input is assumed to be in LR order

a) LMD b) LMD in reverse c) RMD d) RMD in reverse

16. Which of the following is true about the grammar S→aSa | bS | c?

a) Ambiguous and LL(1) b) Unambiguous and LL(1) [ ]

c) Left recursive and LL(1) d) Left factoring and LL(1)

17. The grammar A → AA | (A) | ε is not suitable for predictive-parsing because the grammar is

a) ambiguous b) left-recursive [ ]

c) right-recursive d) an operator-grammar

18. Consider the following grammar:

S→ aABC A→ BC B→ c | d C→ d | ε

What is FOLLOW (C)?

a) {$} b) {c,d} c) {c,d,$} d) {c,d, ε } [ ]

19. Consider the following grammar: [ ]

S→ aABC A→ BC B→ c | ε C→ d | ε

What is FIRST (C)?

a) {$} b) {c,d} c) {c,d,$} d) {c,d,ε}

SECTION-B

SUBJECTIVE QUESTIONS

1. Construct syntax tree for the expression a=b*-c+b*-c
2. List out the rules for First and Follow?
3. Construct predictive parsing table for the following grammar.

E→E+T/T, T →T*F/F, F →(E)/id

4. Define the term Left factoring.
5. Given the grammar

S→ (L) ⍰a

L→ L,S ⍰S

a) Make necessary changes to make it suitable for LL(1) parsing. Construct FIRST and FOLLOW sets.

b) Construct the predictive parsing table. Show the moves made by the predictive parser on the input (a,(a , a)).

6. Which one of the following grammar is ambiguous?

A)S →S0 |1 B) S →0S | 1 C) S →SS |0|1 D) S →SS+ |a

7. The following grammar is Ambiguous or not?

A→AaB | a B→ aB |a

8. Construct recursive descent parsing for the following grammar.

S→cAd A→ab | a

9. Eliminate left factoring from the following grammar.

S→bAd | bAe |ed A→ e | bA

10. Eliminate the left-recursion in the following grammar.

S→A | B A→Aa | □ B→ Bb | Sc | □

11. Define left-factoring. Do the left-factoring for the given grammar

S→iEtS | iEtSeS | a E→b

12. Define Right Most Derivation with example.

13. a) Compute FIRST and FOLLOW for the grammar and construct predictive parsing table.

S→iCtSS' | a   S'→eS | Є   C→b

b) Consider the predictive parsing table from above question and show the sequence of moves made by the parser for w=abba.

14. Explain algorithms to find FIRST and FOLLOW and find FIRST and FOLLOW of following grammar:

S→aBbSA | d   A→eS | ε   B→f

15. Consider the following grammar:

S →L = R

S →R

L →* R

L →id

R →L

Construct LL(1) parsing table for the above grammar. State whether the above mentioned grammar is LL(1) or not and give reasons for either cases.

SECTION-C

GATE QUESTIONS

1. Consider the following grammar: **[GATE 2017]**

P→ xQRS   Q→ yz | z   R→ w | ε   S→ y

What is FOLLOW (Q)?

b) {R}   b) {w}   c) {w,y}   d) {w, $}

2. Consider the grammar defined by the following production rules, with two operators ∗ and + **[GATE 2014]**

S → T * P , T → U | T * U , P → Q + P | Q

Q → Id , U → Id

Which one of the following is TRUE? [ ]

A. + is left associative, while ∗ is right associative

B. + is right associative, while ∗ is left associative

C. Both + and $*$ are right associative

D. Both + and $*$ are left associative

3. For the grammar below, a partial LL(1) parsing table is also presented along with the grammar.

Entries that need to be filled are indicated as E1, E2, and E3. ϵ is the empty string,

$ indicates end of input, and, | separates alternate right hand sides of productions **[GATE 2012]**

$S \rightarrow a\,A\,b\,B \mid b\,A\,a\,B \mid \varepsilon$
$A \rightarrow S$
$B \rightarrow S$

	a	b	$
S	**E1**	**E2**	$S \rightarrow \varepsilon$
A	$A \rightarrow S$	$A \rightarrow S$	error
B	$B \rightarrow S$	$B \rightarrow S$	**E3**

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

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

(C) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = ∅

(D) FIRST(A) = {a, b} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

4. Consider the date same as above question. The appropriate entries for E1, E2, and E3 are **[GATE 2012]**

(A) E1: $S \rightarrow aAbB,\ A \rightarrow S$
E2: $S \rightarrow bAaB,\ B \rightarrow S$
E3: $B \rightarrow S$

(B) E1: $S \rightarrow aAbB,\ S \rightarrow \varepsilon$
E2: $S \rightarrow bAaB,\ S \rightarrow \varepsilon$
E3: $S \rightarrow \varepsilon$

(C) E1: $S \rightarrow aAbB,\ S \rightarrow \varepsilon$
E2: $S \rightarrow bAaB,\ S \rightarrow \varepsilon$
E3: $B \rightarrow S$

(D) E1: $A \rightarrow S,\ S \rightarrow \varepsilon$
E2: $B \rightarrow S,\ S \rightarrow \varepsilon$
E3: $B \rightarrow S$

a) A b) B c) C d) D

5. A grammar G is LL(1) if and only if the following conditions hold for two distinct productions **[NET 2014]**

$A \rightarrow \alpha \mid \beta$

I. First $(\alpha) \cap$ First $(\beta) \neq \{a\}$ where a is some terminal symbol of the grammar.

II. First $(\alpha) \cap$ First $(\beta) \neq \boldsymbol{\varepsilon}$

III. First $(\alpha) \cap$ Follow $(A) = \emptyset$ if $\boldsymbol{\varepsilon} \in$ First (β)

a) I and II b) I and III c) II and III d) I, II and III