1] Context free grammar.

A context free grammar is a 4 tuple $G = (V, T, P, S)$ where

V is finite set of variables / non-terminals.

T is finite set of terminals (tokens)

P is finite set of prod^n of the form

$A \rightarrow \alpha$

where $A \in V$ & $\alpha \in (V \cup T)^*$

$S \in V$ is a start symbol.

Ex -

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid$

here E & A are variables / non-terminals.

id, +, -, /, * are terminals.

2] Syntax analysis:-

In syntax analysis phase the source program is analyzed to check whether it conforms to the source language's syntax & to determine its phase structure.

This phase is often separated into two-phases:-

① lexical analysis - which produces stream of tokens.

② parser - which determines the phrase structure of the program based on context free grammar for the language.

3] Parsing :-

Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree & if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they build parse trees:

① top-down : A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.

② bottom-up : A bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

4] Parse tree-

A parse tree is the graphical representation of the structure of a sentence according to its grammar.

The parse tree may be viewed as a representation for a derivation that filters out the choices regarding the order of replacement.

Let the production P be -

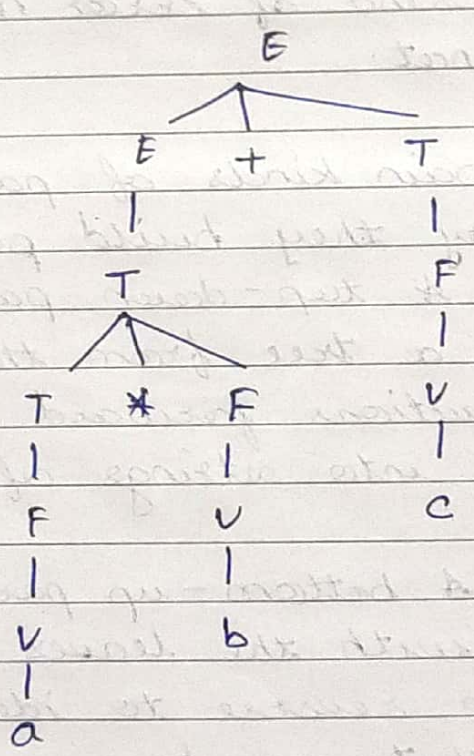
$E \rightarrow T \mid E+T$

$T \rightarrow F \mid T*F$

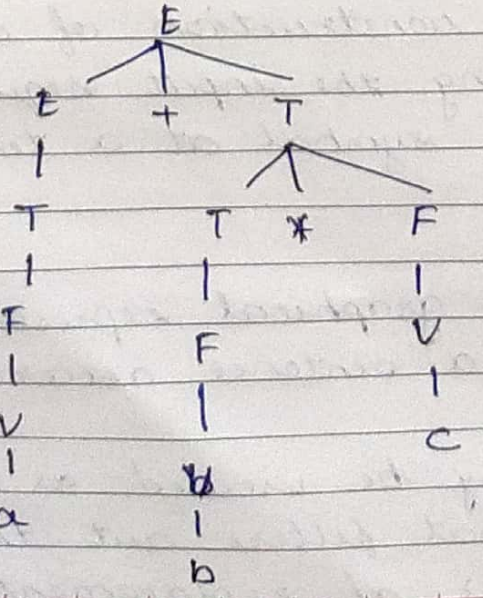
$F \rightarrow V \mid (E)$

$V \rightarrow a \mid b \mid c \mid d$.

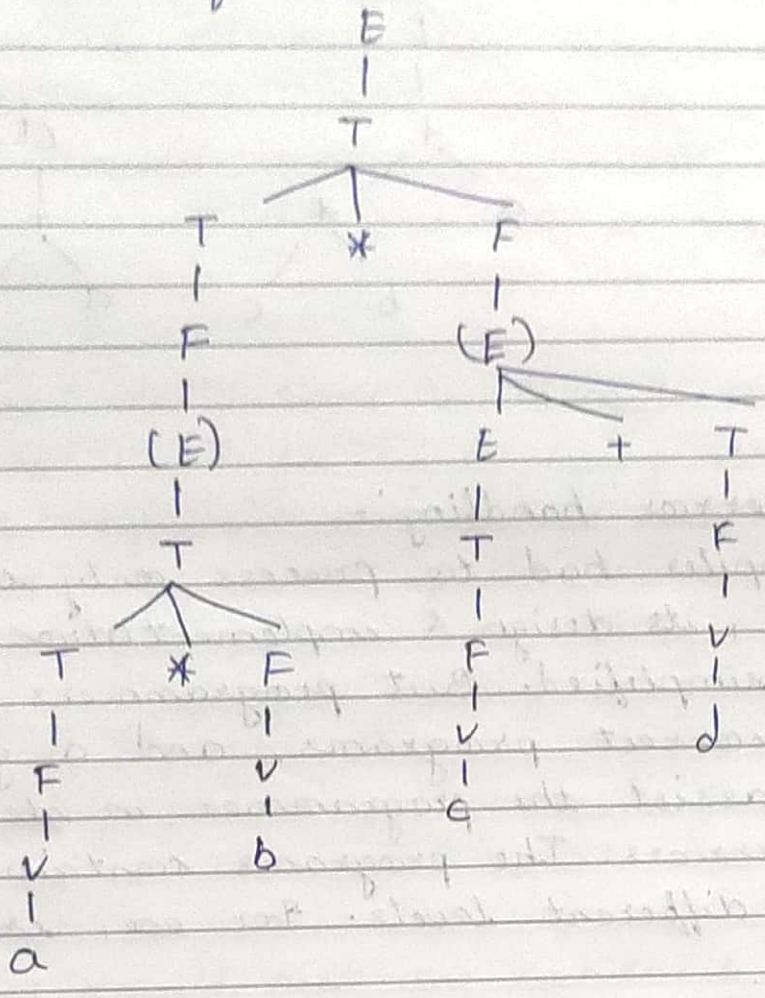
Parse tree for a * b + c



Parse tree for a + b * c



Parse tree for (a * b) * (c + d)



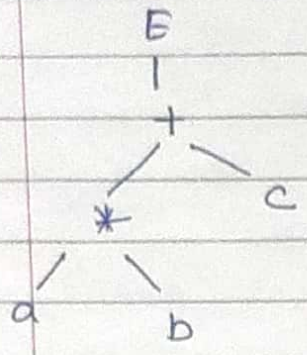
5] Syntax tree -

Parse tree can be presented in a simplified form with only the relevant structure info by :-

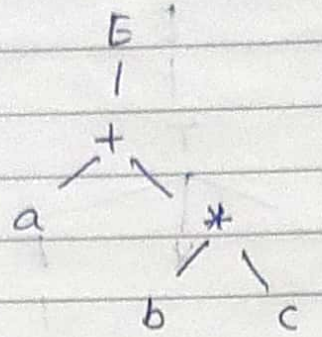
- leaving out chains of derivation (whose sole purpose is to give operators difference precedence)
- labeling the nodes with the operators in question rather than a non-terminal.

The simplified parse tree is sometimes called structural tree or syntax tree.

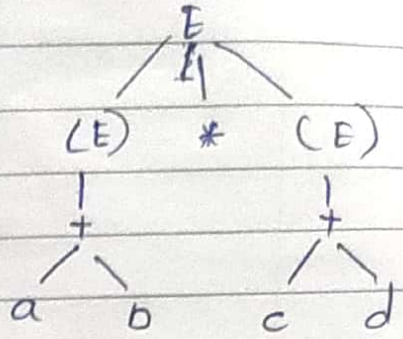
a * b + c



a + b * c



(a+b) * (c+d)



6] Syntax error handling :-

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying & locating errors. The programs contain errors at many different levels. For eg, errors can be:

1) lexical - such as misspelling an identifier, keyword or operator.
2) syntactic - such as an arithmetic expression with unbalanced parenthesis.
3) semantic - such as an operator applied to an incompatible operand.
4) logical - such as an infinitely recursive call.

a++b

-a++

Much of error detection & recovery in a compiler is centered around the syntax analysis phase. The goals of error handler in a parser are:

1) It should report the presence of errors clearly & accurately.

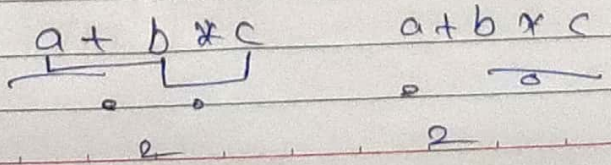
2) It should recover from each error quickly enough to be able to detect subsequent errors.
3) It should not significantly slow down the processing of correct programs.

7] Ambiguity -
Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parser for an ambiguous grammar has to choose somehow which tree to return. There are a number of sol^ to this; the parser could pick one arbitarily or we can provide some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

Any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.



8] Left recursion -
If there is any non terminal A, such that there is a derivation $A \rightarrow A\alpha$ for some string α then, grammar is left recursive.

Algorithm for eliminating left recursion -
1. Group all the A prod^n together -
$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | \beta_1 | \beta_2 | \ldots$

where,
A is the left recursive non-terminal
α is any string of terminals
β is any string of terminals & non-terminals that does not begin with A.

2. Replace the above A prod^n by -
$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots$
$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots$
where A' is a new non terminal.

Top-down parsers cannot handle left recursive grammars.

If our exp grammar is left recursive :-
- This can lead to non-termination in a top-down parser.
- for a top-down parser, any recursion must be right recursion.
- we would like to convert left recursion to right recursion.

9] Eliminate left recursion.

$S \to ABC$

$A \to Aa \mid Ad \mid b$

$B \to Bb \mid e$

$C \to Cc \mid g$

S Ab

ABC

→

$S \to ABC$

$A \to bA'$

$A' \to aA' \mid \epsilon \mid dA'$

$B \to eB'$

$B' \to bB' \mid \epsilon$

$C \to gC'$

$C' \to cC' \mid \epsilon$

Rough

$A \to Aa \mid b$

$A \to bA'$

$A' \to aA' \mid \varepsilon$

$A \to Ad \mid b$

$A \to bA'$

$A' \to dA' \mid \varepsilon$

$A \to Aa \mid b$

$A \to bA'$

$A' \Rightarrow aA' \mid \varepsilon$

10] Eliminate left recursion.

$E \to E+T \mid T$

$T \to T*F \mid F$

→

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

11] Eliminate left recursion.

$E \to E+T \mid E-T \mid T$

$T \to T*F \mid T/F \mid F$

→

$E \to TE'$

$E' \to +TE' \mid -TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid /FT' \mid \epsilon$

$E \to E+T \mid T$

$E \to TE'$

12] Eliminate left recursion.

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

→ transforming -

$S \rightarrow Aa \mid b$

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

$A \rightarrow Ac \mid \epsilon bd$

$A \rightarrow \epsilon bd A'$

$A' \rightarrow cA' \mid \epsilon$

$A \rightarrow Aad \mid \epsilon$

$A \rightarrow \epsilon A'$

$A' \rightarrow adA' \mid \epsilon$

$A \rightarrow bd$

⇓

$S \rightarrow Aa \mid b$

$A \rightarrow A' \mid bdA'$

$A' \rightarrow cA' \mid \epsilon \mid adA' \mid bdA'$

13] Eliminate left recursion.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

→ transforming →

$S \rightarrow Aa \mid b$

$A \rightarrow Aac \mid d \mid bc$

$S \rightarrow Aa \mid b$

$S \rightarrow Sca \mid b$

$S \rightarrow Aaca$

$A \rightarrow Aac \mid d$

$A \rightarrow dA'$

$A' \rightarrow acA' \mid \epsilon$

$A \rightarrow Aac \mid bc$

$A \rightarrow bcA'$

$A' \rightarrow acA' \mid \epsilon$

⇓

$S \rightarrow Aa \mid b$

$A \rightarrow dA' \mid bcA'$

$A' \rightarrow acA' \mid \epsilon$

14] Left factoring -

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decisions until we have seen some

enough of the input to make the right choice.

algorithm :-
For all $A \in$ non-terminal, find the longest prefix α that occurs in two or more right-hand sides of A.
If $\alpha \neq \epsilon$ then replace all of the ^prod^,
$A \to \alpha\beta_1 | \alpha\beta_2 | \dots | \gamma$ with
$A \to \alpha A' | \gamma$
$A' \to \beta_1 | \beta_2 | \dots | \epsilon$
where A' is a new element of non-terminal.

Ex - $V \to \alpha\beta | \alpha\gamma | \delta$
→ $V \to \alpha V' | \delta$
$V' \to \beta | \gamma$

15] Eliminate left factoring.
$S \to V := int$
$V \to alpha\ '['\ int\ ']' | alpha.$
→
$S \to V := int$
$V \to alpha\ V'$
$V' \to '['\ int\ ']' | \epsilon$

16] Eliminate left factoring.
$S \to iEtS | iEtSeS | a$
$E \to c$
→ $S \to iEtSS' | a$
$S' \to \epsilon | eS$
$E \to c$

17] Top down parsing -

Top down parsing is the construction of a parse tree by starting at start symbol & guessing each derivation until we reach a string that matches input. That is, construct tree from root to leaves.

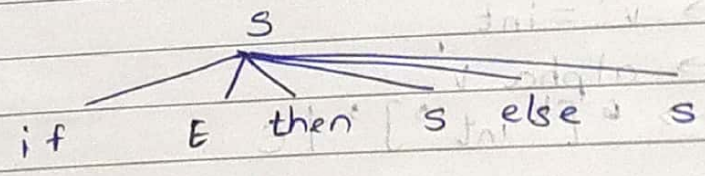
The advantage of top-down parsing is that a parser can directly be written as a program. Table-driven top-down parsers are of minor practical relevance. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.

for ex - S → if E then S else S | while E do S | print
E → true | false | id.

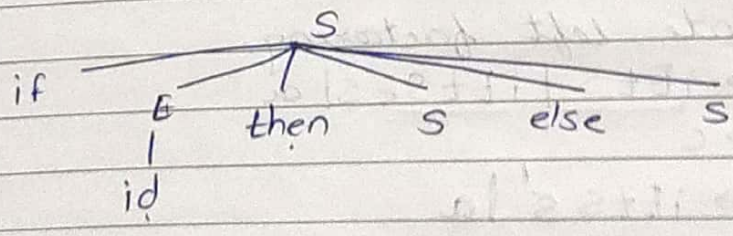
string - If id then while true do print else print.

1. S

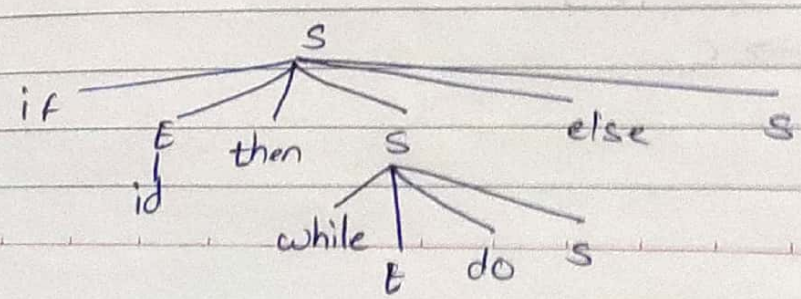
2.



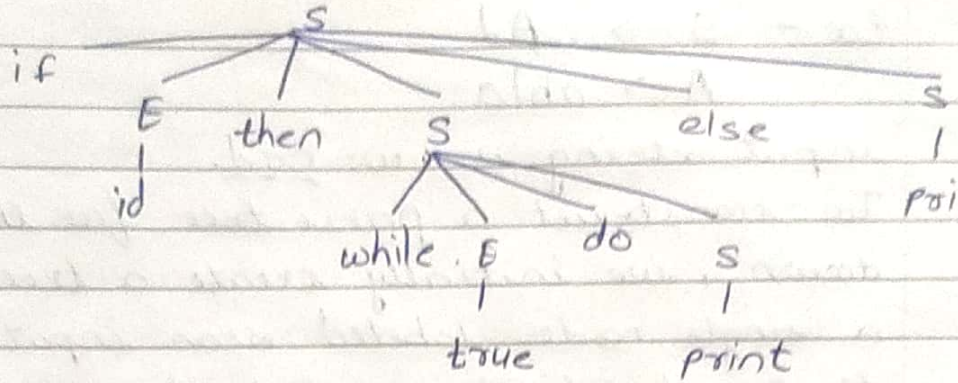
3.



4.



5.



18] Recursive descent - parsing -

Top - down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from root & creating the nodes of the parse tree in preorder.

The special case of recursive - descent parsing, called predictive parsing, where no backtracking is required. The general form of top - down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of input.

Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which prod^n to use.

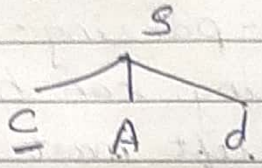
Recursive descent parser is a top down parser involving backtracking. It makes repeated scans of input. Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.

Ex - $S \rightarrow cAd$

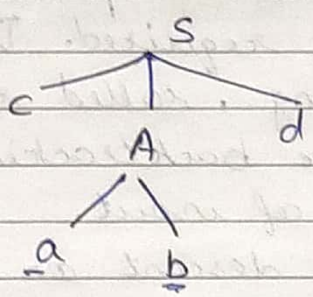
$A \rightarrow abla.$

input string is $w = cad$.

To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled S. An input pointer points to c, the first symbol of w, we then use the first prod^n for S to expand tree & obtain tree →

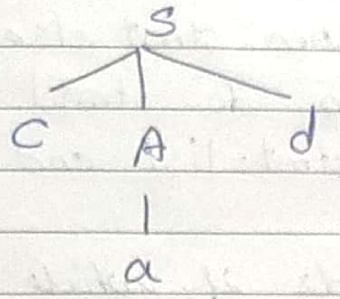


Now, we advance the input pointer to a, the second symbol of w and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree.



We now have a match for the second input symbol so we advance the input pointer to d, the third input symbol & compare d against next leaf, labeled b. Since b does not match d, we report failure & go back to A to see where there is any alternative for A that we have not tried but might produce a match.

In going back to A, we must reset the input pointer to position 2, we now try second alternative for A to obtain tree :-



The leaf matches second symbol of w & the leaf d matches the third symbol.

The left recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand A, we may eventually find ourselves again trying to expand A without having consumed any input.

19] Predictive parsing - LL(1)

Predictive parsing is top-down parsing without backtracking or look-a-head. For many languages, make perfect guesses (avoid backtracking) by using 1- symbol look-a-head, i.e., if

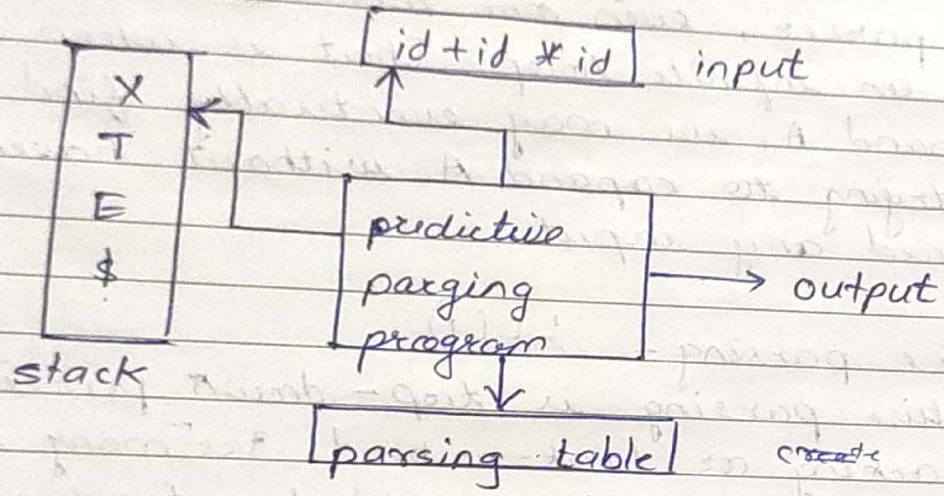
$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$.

choose correct α_i by looking at first symbol it derive. If ϵ is an alternative, choose it last.
This approach is also called as predictive parsing.
There must be atmost one prod^n in order to avoid backtracking. If there is no such prod^n then no parse tree exists & an error is returned.
The grammar must not be left-recursive.
Predictive parsing works well on those fragments of programming languages in which keywords occur frequently.
Eg -

stmt → if exp then stmt else stmt |
while exp do stmt |
begin stmt-list end.

then the keywords if, while & begin tell which alternative is the only one that could possibly succeed if we are to find a statement.

The model of predictive parser is as follows :-



A predictive parser has :-
stack,
input,
parsing table,
output.

The input buffer consists the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
The stack consists of a sequence of grammar symbols with $ on the bottom, indicating the bottom of stack. Initially the stack consists

of the start symbol of grammar on top of $.

Recursive descent & LL parsers are often called predictive parsers, because they operate by predicting the next step in a derivation.

20] Computing first →

If X is a grammar symbol, then first(X) =

- If X is terminal symbol, then
  first(X) = {X}.
- If X → ε, then first(X) = {ε}
- If X is non-terminal (variable) & X → aα, then first(X) = {a}.

If X → Y1, Y2, Y3 then first(X) will be.

① If Y is terminal then
first(X) = first(Y1, Y2, Y3) = {Y1}.

② If Y1 is non-terminal (variable) & if Y1 does not derive to an empty string, i.e., if first(Y1) does not contain ε then,
first(X) = first(Y1, Y2, Y3) = first(Y1)

③ If first(Y1) contains ε then
first(X) = first(Y1, Y2, Y3) = first(Y1) - {ε} ∪ first(Y2, Y3).

similarly first(Y2, Y3) = {Y2}, if Y2 is terminal otherwise if Y2 is non-terminal then
first(Y2, Y3) = first(Y2) if & first(Y2) does not contain ε.

If first (Y_2) contains ϵ then
first (Y_2, Y_3) = first (Y_2) − {ϵ} ∪ first (Y_3).

2] Find first of following -

① $S \rightarrow abc \mid def \mid ghi$

$S \rightarrow abc$
$S \rightarrow def$
$S \rightarrow ghi$

ans = first (S) = a, d, g.

② $S \rightarrow ABC \mid ghi \mid jkl$
$A \rightarrow a \mid b \mid c$
$B \rightarrow b$
$D \rightarrow d$

ans = first (D) = d
first (B) = b
first (A) = a, b, c
first (S) = first (ABC) ∪ first (ghi) ∪ first (jkl)
= first (A) ∪ first (ghi) ∪ first (jkl)
= a, b, c, g, j.

③ $E \rightarrow TE'$
$E' \rightarrow *TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow \epsilon \mid +FT'$
$F \rightarrow id \mid (E)$

Termial = a, b, $
non terminal = A
variable

ans = first (F) = id, (
first (T') = ϵ, +
first (T) = first (F) = id, (
first (E') = *, ϵ
first (E) = first (T) = id, (

④ $S \to ABC$
$A \to a|b|\epsilon$
$B \to c|d|\epsilon$
$C \to e|f|\epsilon$

a, b, ε
aBC
bBC
εBC BC

ans = first (C) = e, f, ε
first (B) = c, d, ε
first (A) = a, b, ε
first (S) = first (ABC)
= first (A) − {ε} ∪ first (B) − {ε}
= ~~a, b~~, ∪ first (C)
= a, b, c, d, e, f, ε

22] Computing follow →
If S is the start symbol, Follow(S) = {$}

If prod^n is of form $A \to \alpha B\beta$, $\beta \neq \epsilon$
- if first (β) does not contain ε then,
follow (B) = { first (β)}

- if first (β) contains ε then
follow (B) = first (β) − {ε} ∪ follow (A).

$A \to \alpha B \beta$

Q3) Construct first & follow for grammar

$A \to BC \mid EFGH \mid H$

$B \to b$

$C \to c \mid \varepsilon$

$DE \to e \mid \varepsilon$

$F \to CE$

$G \to g$

$H \to h \mid \varepsilon$

→

* First set.

first (H) = {h, ε}

first (G) = {g}

first (F) = first (C) ∪ first (E)
= {c, e, ε}

first (E) = {e, ε}

first (C) = {c, ε}

first (B) = {b}

first (A) = first (BC) ∪ first (EFGH) ∪ first (H)
= ~~first (B) ∪ first (E) ∪ first (H)~~
= ~~{b, e, ε, h}~~
= first (B) ∪ first (E) − {ε} ∪ first (FGH)
∪ first (H)
= {b} ∪ {e} ∪ first (F) − {ε} ∪ first (GH)
~~∪ {h} − {ε}~~ ∪ {h, ε}
= {b} ∪ {e} ∪ {c, e} ∪ {g} ∪ {h, ε}
= {b, e, c, g, h, ε}

Page No.
Date

* follow set.

follow(A) = {$}

follow(B) = first(C) ~~= {c, ε} - {ε} ∪ follow(A)~~
= {c, $}

follow(C) = first(E) - {ε} ∪ follow(F) ∪ follow(A)
= {e, $, g}

follow(E) = first(F) - {ε} ∪ first(G)
= {c, e, g}

follow(F) = first(G)
= {g}

follow(G) = first(H) - {ε} ∪ follow(A)
= {h, $}

follow(H) = follow(A)
= {$}

Q.24. Construct a predictive parsing table for the given grammar (or) check whether the given grammar is LL(1) or not.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

→

Steps:-
1. Eliminate left recursion
2. left factoring.
3. first & follow functions
4. predictive parsing table
5. parse the input string.

step 1:- Eliminating left recursion.

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

step 2:- Left factoring.

no left factoring.

step 3:- first & follow functions.

first (E) = first (T) = first (F) = { (, id }

first (E') = { +, ε }

first (T) = { (, id }

first (T') = { *, ε }

first (F) = { (, id }.

follow (E) = { $,) }

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

follow (E') = { $,) }

follow (T') = { +, $,), }

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

step 4:- predictive parsing table.

If $A \rightarrow \alpha$ consider row of A & column of first (α).

1] M[A, a] = $A \rightarrow \alpha$, a is in first of (α)

2] M[A, b] = $A \rightarrow \alpha$, if ε is in first (α), b is in follow (A).

where M is table.

E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \varepsilon$		$E' \to \varepsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		$T' \to \varepsilon$
F			$F \to (E)$		$F \to id$	

As there are no multiple entries in table, hence the given grammar is LL(1).

step 5:-
Moves made by predictive parser on the input id * id + id.

stack	input	output
$ E	id * id + id $	$E \to TE'$
$ E' T	id * id + id $	$T \to FT'$
$ E' T' F	id * id + id $	$F \to id$
$ E' T' id	id * id + id $	
$ E' T'	* id + id $	$T' \to *FT'$
$ E' T' F *	* id + id $	
$ E' T' F	id + id $	$F \to id$
$ E' T' id	id + id $	
$ E' T'	+ id $	$T' \to \varepsilon$
$ E'	+ id $	$E' \to +TE'$
$ E' T +	+ id $	
$ E' T	id $	$T \to FT'$

$E'T'F	id $	F → id
$E'T'id	id $	
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	

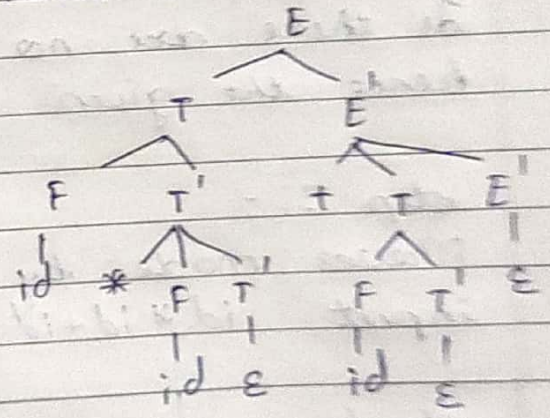
We can notice $ in input & stack, i.e., both are empty.

∴ Predictive parser accepts the given input string.

∴ output is → (optional)

E → TE'
T → FT'
F → id
T' → *FT'
F → id
T' → ε
E' → +TE'
T → FT'
F → id
T' → ε
E' → ε

Parse tree = (optional)



25] LL(1) grammar :-
The first L stands for "left-to-right scan of input". The second L stands for "left-most derivation". The '1' stands for 1 token of look-a-head.

No LL(1) grammar can be ambiguous or left recursive.

If there were no multiple entries in the recursive descent parser table, the given grammar is LL(1).

If the grammar G is ambiguous, left recursive then the recursive descent table will have atleast one multiple defined entry.

LL(1) = top-down, predictive parsing.

The weakness of LL(1) parsing is that, it must predict which prod^n to use.

26] Construct a predictive parsing table for the given grammar. (or)
check whether the grammar is LL(1) or not.
$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b.$

→ step 1:- Eliminating left recursion.
no left recursion.

step 2:- left factoring
no left factoring.

step 3:- computing first & follow.

first (E) = b
first (S) = { i, a }
first (S') = {e, ε}

follow (S) = { $, e}
follow (S') = { $, e}
follow (E) = {t}.

step 4:- predictive parsing table.

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

As the table contains multiple defined entry, the grammar is not LL(1).

27] Construct the first & follow & predictive parse table for the grammar.

$S \rightarrow A\,C\,\$$

$C \rightarrow c \mid \epsilon$

$A \rightarrow aBCd \mid BQ \mid \varepsilon$

$B \rightarrow bB \mid d$

$Q \rightarrow q.$

→ First :-

first (Q) = {q}

first (B) = {b, d}

first (A) = {a, b, d, c, ε}

first (C) = {c, ε}

first (S) = {a, b, d, c, ε}

follow (S) = {#}

follow (A) = {c, $}

follow (C) = {$, d}

follow (B) = {c, d, q, b}

follow (Q) = {c, $}

	a	b	c	d	q	ε	$	#
~~A~~S	S→AC$	S→AC$	S→AC$	S→AC$		S→AC$		
B		B→bB		B→d				
C			C→c			C→ε		
A	A→aBCd	A→BQ		A→BQ		A→ε		
Q					Q→q.			

Page No.
Date

Moves made by predictive parser on the input abdcdc$ is :-

stack	input	output
#S	abdcdc$#	S → Ac$
#$cA	abdcdc$#	A → aBCd
#$CdCBa	abdcdc$#	
#$CdCB	bdcdc$#	B → bB
#$CdCBb	bdcdc$#	
#$CdCB	dcdc$#	B → d
#$CdCd	dcdc$#	
#$Cdc	cdc$#	C → c
#$Cdc	cdc$#	
#$Cd	dc$#	
#$C	c$#	C → c
#$c	c$#	
#$	$#	
#	#	accepted.

28] Bottom up parsing -
Bottom up parser builds a derivation by working from the input sentence back towards the start symbol S.
Right most derivation in reverse order is done in bottom up parsing.

In terms of parse tree, this is working from leaves to root.

Ex- $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

input : $id_1 + id_2 * id_3$.

⇒ Right most derivation.
$E \to E+E$
$\to E+E*E$
$\to E+E*id_3$
$\to E+id_2*id_3$
$\to id_1+id_2*id_3$.

bottom-up approach: go from left to right.
$id_1 + id_2 * id_3$
$E + id_2 * id_3$ $\quad E \to id$
$E + E * id_3$ $\quad E \to id$
$E + E * E$ $\quad E \to id$
$E + E$ $\quad E \to E*E$
E $\quad E \to E+E$.
↑
start symbol.

Hence acceptable.

29] Top down vs bottom up parsing -

Top down parsing	Bottom up parsing.
construct tree from root to leaves.	construct tree from leaves to root.
Produces ~~if~~ left-most derivation.	Produces reverse right-most derivation.
Recursive descent, LL parsers.	shift reduce, LR, LALR.

30] handles -
Always making progress by replacing a substring with LHS of a matching prod^n will not lead to the goal/start symbol.

A handle of a string is a substring that matches the right side of production and whose red^n to non-~~rule~~ terminal on the left side of prod^n represents one step along the reverse of a right most derivation.

If the grammar is unambiguous, every right sentential form has exactly one handle.

31] Handle pruning -
Keep removing handles, replacing them with corresponding LHS of prod^n, until we reach start symbol of grammar.

Ex - $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

input string : abbcde

Handles during parsing of abbcde

Right-sentential form	handle	reducing prod^n
abbcde	b	$A \rightarrow b$
aAbcde	Abc	$A \rightarrow Abc$
aAde	d	$B \rightarrow d$
aABe	aABe	$S \rightarrow aABe$
S		

Here 'b' is the handle as it matches with right side of prod^n.

If handle matches with right side of prod^n, replace it with left side of prod^n

32] Shift - reduce parsing -
Shift reduce parsing uses a stack to hold grammar symbols & input buffer to hold string to be parsed, because handles always appear at the top of the stack, i.e., there's no need to look deeper into the state.

A shift - reduce parser has four actions -
① shift - next word is shifted onto stack until a handle is formed.

② reduce - right end of handle is at top of stack, locate left end of handle within the stack. Pop handle off

stack & push appropriate LHS.

③ accept - stop parsing on successful completion of parse & report success.

④ error - call an error reporting / recovery routine.

33] shift reduce parsing conflicts -

① shift - reduce -
Both a shift action & a reduce action are possible in the same state.

② reduce - reduce -
Two or more distinct reduce actions are possible in the same state.

34] ① shift - reduce parsing :-
- shift - reduce parsing is a process of reducing a string to the start symbol of grammar.
- It uses stack to hold grammar & an input to hold the string.

- A shift reduce parsing ~~pr~~ performs two actions:
① shift
② reduce.

- At shift action, the current symbol in the input string is pushed to a stack.

- At each reduction, the symbol will be replaced by the non-terminals.

- Example ①

grammar

$S \to S+S$

$S \to S-S$

$S \to (S)$

$S \to a$

input string

$a_1 - (a_2 + a_3)$

ans -

stack	input string	action
$	$a_1 - (a_2 + a_3)$ $	shift a_1
$a_1	- (a2 + a3) $	reduce by $S \to a$
$S	- (a2 + a3) $	shift -
$S-	(a2 + a3) $	shift (
$S-(	a2 + a3) $	shift a2
$S-(a2	+ a3) $	reduce by $S \to a$
$S-(S	+ a3) $	shift +
$S-(S+	a3) $	shift a3
$S-(S+a3	) $	reduce by $S \to a$
$S-(S+S	) $	shift)
$S-(S+S)	$	reduce by $S \to S+S$
$S-(S)	$	reduce by $S \to (S)$
$S-S	$	reduce by $S \to S-S$
$S	$	accept

LL(1)

LR(0)

LR(0)

- Example ②

grammar

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

input string.

$id_1 + id_2 * id_3$.

stack	input	action
$	$id_1 + id_2 * id_3 $	shift id_1
$ id_1	$+ id_2 * id_3 $	reduce by $E \rightarrow id$
$ E	$+ id_2 * id_3 $	shift +
$ E +	$id_2 * id_3 $	shift id_2
$ E + id_2	$* id_3 $	reduce by $E \rightarrow id$
$ E + E	$* id_3 $	reduce by $E \rightarrow E+E$ shift *
~~$ E + E *~~	~~$id_3 $~~	~~shift id_3~~
~~$ E + E * $id_3~~	~~$~~	~~reduce by $E \rightarrow id$~~
~~$ E + E * E~~	~~$~~	
$ E	$* id_3 $	shift *
$ E *	$id_3 $	shift id_3
$ E * id_3	$	reduce by $E \rightarrow id$
$ E * E	$	reduce by $E \rightarrow E*E$
$ E	$	accept.

- Example ③

grammar

$E \rightarrow 2E2$

$E \rightarrow 3E3$

$E \rightarrow 4$

i/p string

~~34~~ 32423.

stack	input	Action
$	32423$	Action
$3	2423$	shift 3
$32	423$	shift 2
$324	23$	shift 4
$32E	23$	reduce E→4
$32E2	3$	shift 2
$3E	3$	reduce E→2E2
$3E3	$	shift 3
$E	$	reduce E→3E3
		accept

35] Operator grammar :-

A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar.

A grammar is said to be operator precedence grammar if it has 2 properties :-

(i) no RHS of any prod has ε

(ii) no two non-terminals are adjacent on RH.S.

Ex - $E \to E+E \mid E*E \mid id$. ✓

$S \to SAS \mid a$ ✗

$A \to bSb \mid b$

There are 3 operator precedence relations :-

1) $a \doteq b$ → means a has higher precedence than b. (or) a takes precedence over b.

2) $a \lessdot b$ → means a yields precedence to b (or) a has lower precedence than b.

3) $a \doteq b$ → means a has same precedence as b.

3b] Operator precedance parser.

- This parser is only used for operator grammar.
- Ambiguous grammar are not allowed in any parser except operator precedance parser.
- operation precedence relation table.

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

	id	+	*	$
id	−	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	−

Two ids will never be compared because they will never be compared side by side.
id has highest priority.
$ has least precedence.
+ has left associativity.
* has left associativity.

input string	precedence rel^n inserted	action
id + id * id	$ < id > + < id > * > id	E → id
E + id * id	$ + < id > * > id	

input string	precedence rel^n inserted	action
id + id * id	$ < id > + < id > * < id > $	E → id
E + id * id	$ + < id > * < id > $	E → id
E + E * id	$ + * < id > $	E → id
E + E * E	$ + * $	
E + E * E	$ < + < * > $	E → E * E
E + E	$ < + > $	E → E + E
E	$ $	accepted.

Disadvantages of operator precedance parsing:-

1. hard to handle tokens like minus sign, which has two different precedancies.
2. only a small class of grammars can be parsed.
3. usage is limited.

37] LR - Parsing -
The 'L' is for left-to-right scanning of the input & 'R' is for constructing a right most derivation in reverse.

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.
2. The LR parsing method is the most general non-backtracking shift-reduce parsing method known.
3. The class of grammars that can be parsed using LR methods is a proper subset 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.

38] LR parsers -
LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 & k=1.
LR(1) is of practical relevance.

k stands for no. of input symbols of look-a-head that are used in making parsing decisions.
When (k) is omitted, 'k' is assumed to be 1.

LR(1) parsers are table-driven, shift reduce parsers that use a limited right context (1 token) to handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar. A grammar is LR(1), if given a right-most derivation.

Grammar rule combined with a dot indicates a position in its RHS.
Ex- $A \rightarrow XYZ$ generates 4 LR(0) items.

$A \rightarrow .XYZ$ indicates that the parser is looking for a string that can be derived from XYZ.

$A \rightarrow XY.Z$ indicates that the parser has seen a string derived from XY & is looking for one derivable from Z.

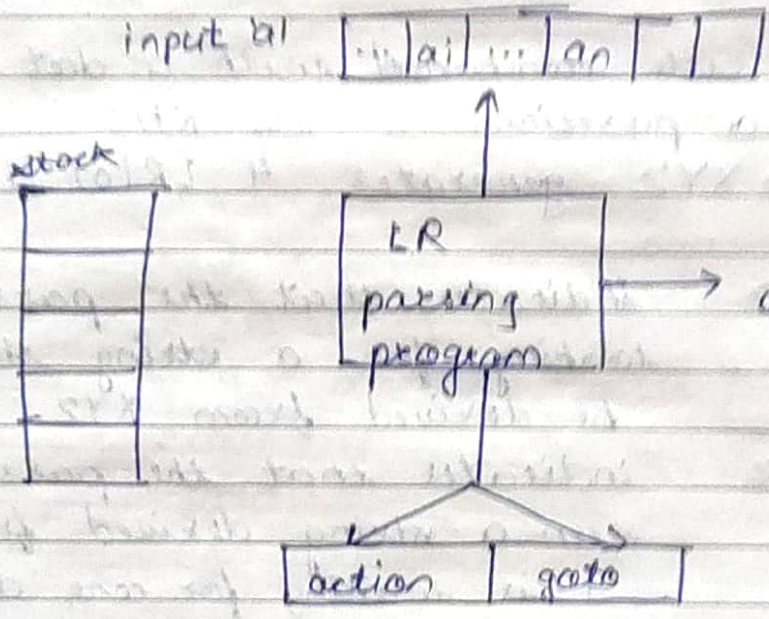
$A \rightarrow X.YZ$ indicates that the parser has seen a string derived from X & is looking for derivable from YZ.

$A \rightarrow XYZ.$ indicates that the parser has seen a string derived from XYZ.

LR(0) items play a key role in SLR(1) table constr algorithm.
LR(1) items play a key role in the LR(1) & LALR(1) table constr algorithm.
LR parsers have more info available than

LL parsers when choosing a prod':
LR knows everything derived from RHS plus k lookahead symbols.
LL just knows k lookahead symbols into whats derived from RHS.

39] LR parsing algorithm :-



It consists of an input, an output, a stack, a driver program and a parsing table that has two parts :- action & goto.

The LR parser program determines Sm, the current state on the top of the stack & ai, the current input symbol. It then consults action [Sm, ai], which can have one of four values :-
1. shift S, where S is state.
2. reduce by a grammar prod' A → β
3. accept
4. error.

The function 'goes to' takes a state & grammar symbol as arguments & produces a state. The 'goto' function of parsing table constructed from a grammar. G using the SLR, canonical LR or LALR method.

40] LR(0) parsing -

$S \rightarrow AA$

$A \rightarrow aA \mid b$

→ ~~step 1 :- Augmented grammar.~~

~~$S' \rightarrow S$~~

~~$S \rightarrow AA$~~

~~$A \rightarrow aA \mid b$~~

d.abc
da.bc

step 1 :-

$S \rightarrow AA$ -①

$A \rightarrow aA$ -②

$A \rightarrow b$ -③

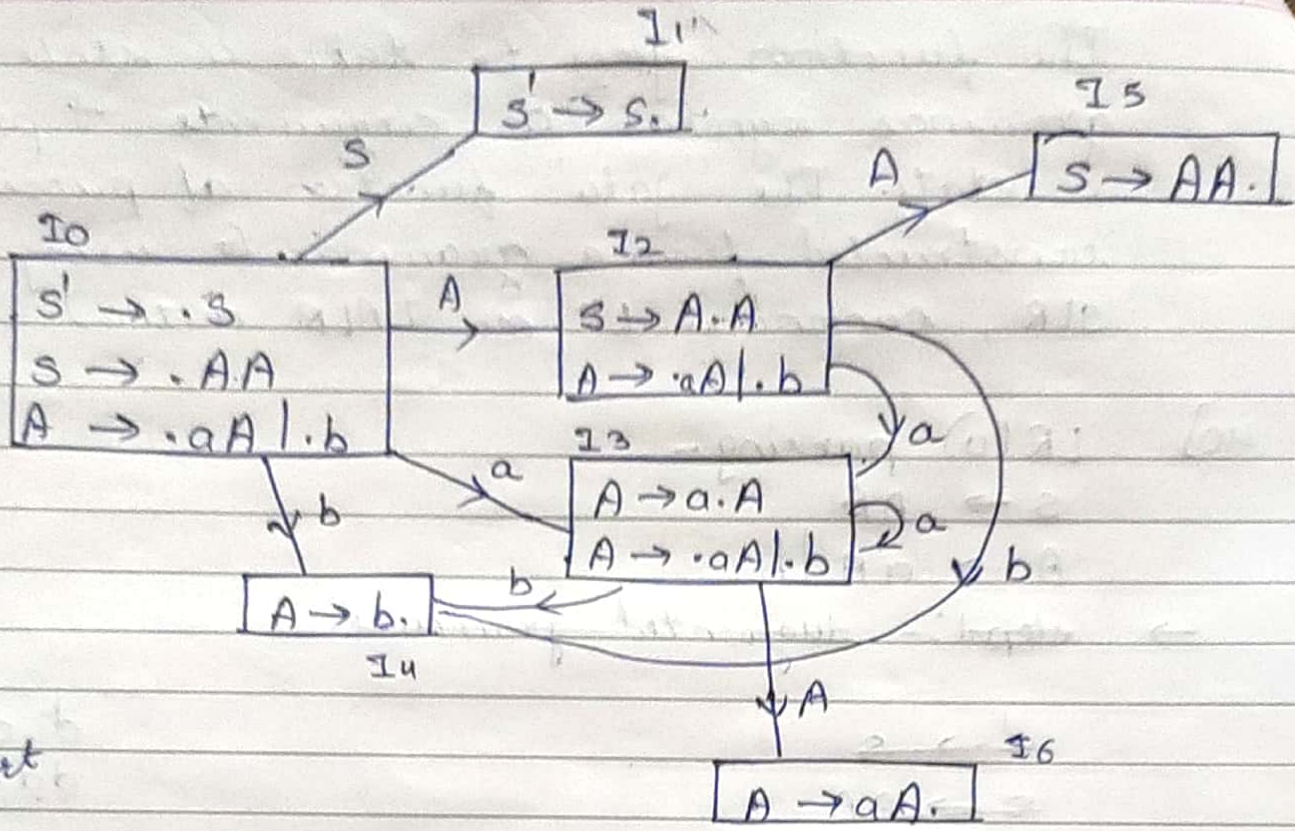
step 2 :- Augmented grammar

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA \mid .b$

step 3 :- applying closure.



action part contains terminals.

goto part contains variables.

step 4:- LR(0) parsing table.

	action			goto	
	a	b	$	A	S
0	s_3	s_4		2	1
1			accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

~~S → aA~~

Q.] SLR(1) parsing.

$S \rightarrow AA$

$A \rightarrow aA \mid b$.

step 1:-

$S \rightarrow AA$ -①

$A \rightarrow aA$ -②

$A \rightarrow b$ -③

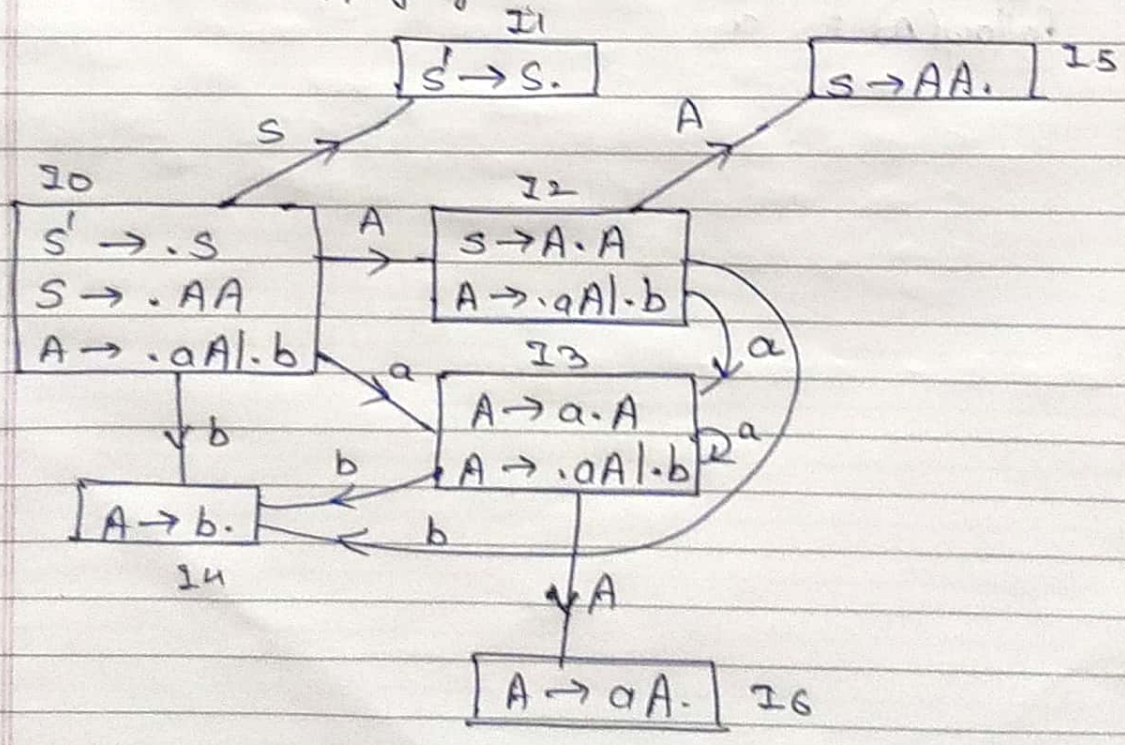
step 2:- augmented grammar.

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA \mid .b$.

step 3:- applying closure



step 4:- SLR(1) parsing table.
goto is same as LR(0)
shift actions are same as LR(0)
reduce actions are different than LR(0).
don't write reduce action for whole row. Write only in follow's column.

	action			goto	
	a	b	$	A	S
0	s_3	s_4		2	1
1			accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_4	r_4	r_4		
5			r_5		
6	r_6	r_6	r_6		

follow(S) = $
follow(A) = a

42] CLR parsing table.

CLR & LALR use LR1 canonical items

~~S → aAd | aBe | bAe~~

S → aAd | bBd | aBe | bAe

A → c

B → c

step 1:-

S → aAd - ①

S → bBd - ②

S → aBe - ③

S → bAe - ④

A → c - ⑤

B → c - ⑥

step 2:- Augmented grammar

S' → .S, $ $ is lookahed

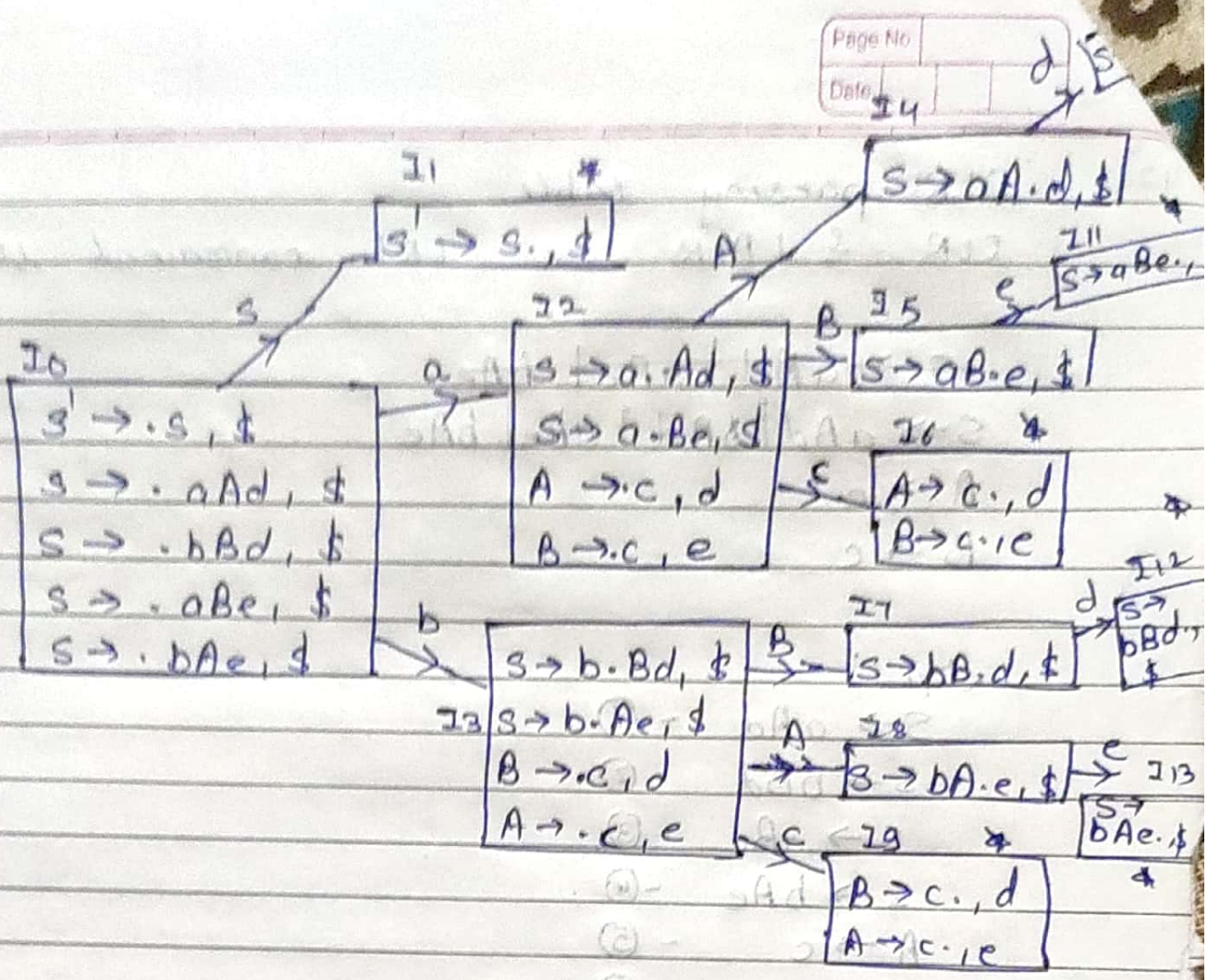
S → .aAd, $

S → .bBd, $

S → .aBe, $

S → .bAe, $

step 3:- closure apply.



step 4:- CLR parsing table.
goto is same as LR(0)
shift moves are same as LR(0).
reduce moves should not be written in whole row. Reduce moves should be written only in lookahead columns.

	action						goto		
	a	b	c	d	e	$	A	B	S
0	S_2	S_3				~~accept~~			1
1						accept			
2			S_6				4	5	
3			S_9				8	7	
4				S_{10}					
5					S_{11}				
6				r_5	r_6				
7				S_{12}					
8					S_{13}				
9				r_6	r_5				
10					~~r_5~~	r_1			
11						r_3			
12						r_2			
13						r_4			