

12/11/2019

CH I

INTRODUCTION TO COMPILER

* Preprocessor (checks #)

define square(x) x*x

main()

{ int x=5, y;

y = 125 / square(x);

pf(y);

Pre processor

↓ HLL

Compiler

↓ LLL or assembly or m/c.

Lang. Lang.

Assembler

↓ multiple relocatable assembly lang.

working
of
compiler

Linker

↓ single relocatable assem lang.

loader

#

Compiler

Lexical analysis.

Syntax analysis.

Semantic analysis

Intermediate code gen.

Code optimisation

Target code generation

↓

LLL

HLL during compilation,

preprocessor reads # statement

2 #if finds square(x) in

program & replaces it by

x*x.

125/x*x = 125/5*5 = 125

Once preprocessor replaces text

it then removes # statement.

* m/c lang means instruction

set of that particular processor

* High level lang consists of many

shortcuts which are replaced

by preprocessor & hence converted
to pure HLL.

* Compiler o/p is assembly lang

because it is i/p to assembler

* The prog. after compilation

is still present in hard disk.

* To ^{run} program, we need to

bring prog. from hard disk to

main mem. First prog. is

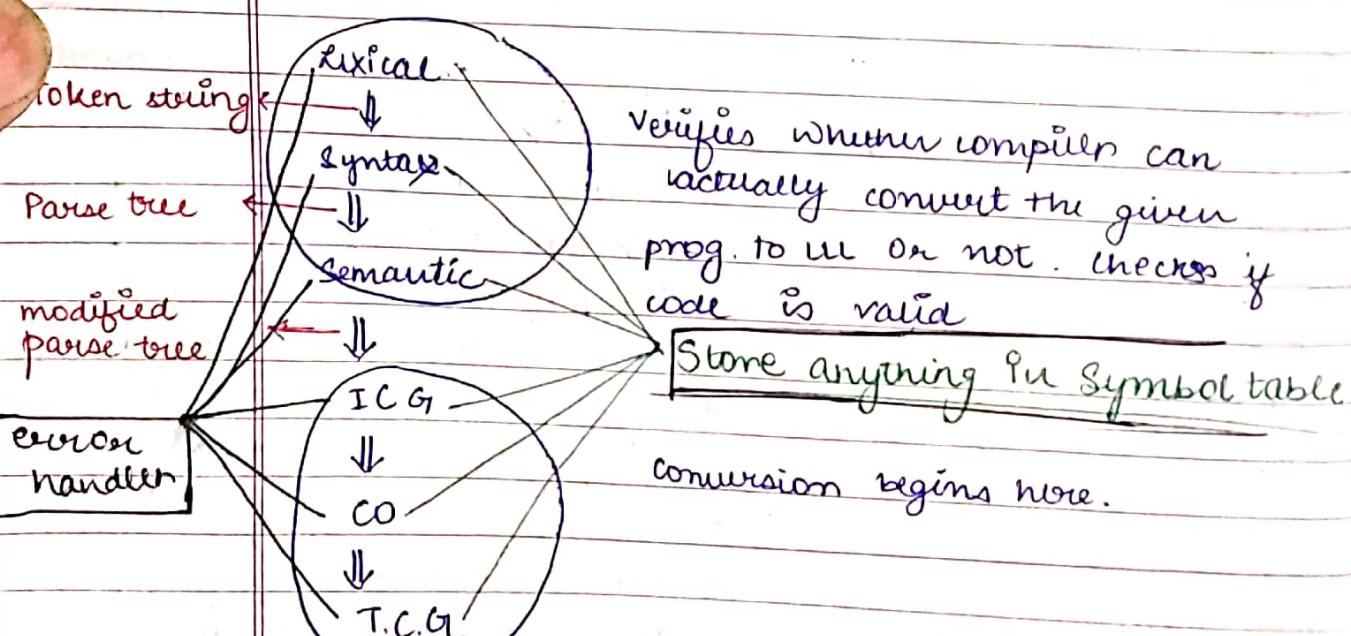
converted to relocatable prog. by

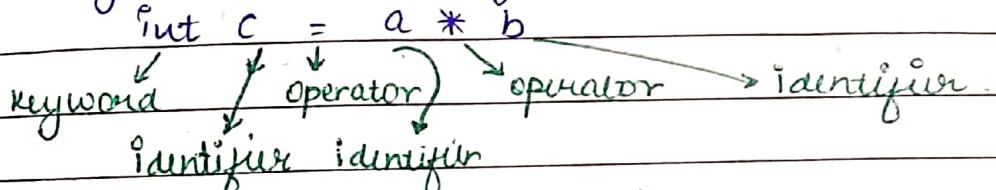
* Linker O/P :- multiple relocatable assembly language. They are linked since they are related to each other. If files are not linked & only 1 is sent to main memory then it will cause trouble in execution.

* Loader loads the single relocatable A.L file to main mem. Here prog. waits in ready queue to get its chance to run. It is in ready state. This file is called .exe file.

* Compiler is a software/ program which may contain a large amt of code. Here code is divided into various modules.

* Code Optimisation is where unnecessary code is removed. This phase is not present in some compilers because they optimise code in ICG phase itself.



- Lexical analyser basically introduces / identifies everyone in the program. It O/P a token string.

 It divides program to tokens.

- Once tokens are created, syntax analyser creates a parse tree for each token. eg. a token is created for "for". There is a grammar associated to each token which gives rules to that token. It creates a tree to check if it matches the grammar defined. If no, then syntax error.

eg Once turbo C is installed, we get rules / grammar for entire C lang. where grammar is defined for each possible token.

- Semantic analysis makes modified parse tree & it performs meaningful checking of tokens.
 eg. undeclared variable, type mismatch, multiple declaration

 If any problem occurs in any phase, it is given to error handler. If error handler has a solution to it, it will return the solⁿ to that phase.
 eg. type mismatch error sent to error handler. It returns a solⁿ :- "type conversion" to semantic analyser.

- User-defined variables are not present in compiler. They

Whenever declaration occurs, semantic analyser adds these entries to symbol table.

float x, a, b;

Date: / /
Page No. 24

eg.

$x = a + b * 60$



L.A.



x - identifier - (id, 1)

= assignment op.

a - identifier - (id, 2)

+ addition op.

b - identifier - (id, 3)

* - multiplication op.

60 integer const.

string

$(\text{id}, 1) = (\text{id}, 2) + (\text{id}, 3) * 60$

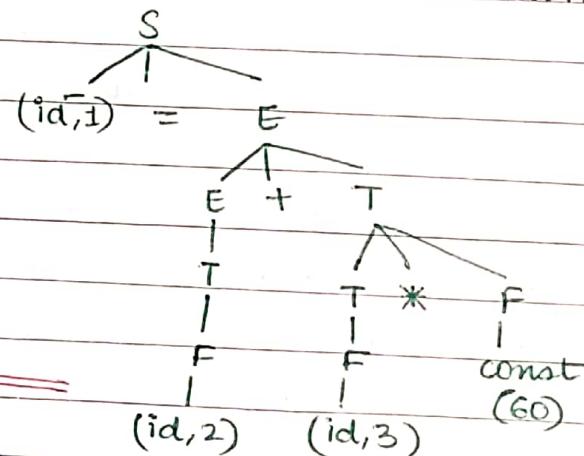


Syntax analyser

complex phase because of tree construction

grammar

$$\left\{ \begin{array}{l} S \rightarrow id = E \\ E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow id \mid \text{const} \end{array} \right.$$



semantic analyser



float
 $(\text{id}, 1) = E$ float

float E + T float

float T (float) 60

T * F

③ Once tree is constructed by syntax analyser, semantic analyser picks up a token & checks it in symbol table to identify its type. It makes use of token no. to directly find token.

If in lexical analysis phase, correct token no.s. are assigned to all identifiers then it helps all the other phases. It increases overall speed of compiler.

Date: / /

Page No. 25



Intermediate code generation



$$t_1 = b * 60.0$$

$$t_2 = a + t_1$$

$$x = t_2$$



Code optimisation



$$t_1 = b * 60.0$$

$$x = a + t_1$$



Target code generation



MUL t₁, b, 60

ADD x, a, t₁

Not in
syllabus

Ques. It is possible that when semantic analyser checks symbol table for "type" of "x", type may not be mentioned even if x entry is present. This is called declaration missing.

Ans. Semantic analyser does type checking as well as finds undefined variables.

$$x = a + b * 60$$

↓ i/p.

LA

$$x - (Id, 1)$$

$$a - (Id, 2)$$

Here a, b are not declared. So

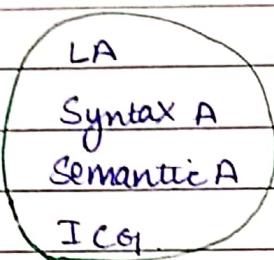
symbol table is empty. When this is sent to LA, LA has to give id to x. It checks in symbol table & finds that it is not present. So instead of

Now when semantic analyser wants to know type of identifier, it checks symbol table but type missing. So it sends declaration missing error.

- * Who can write in symbol table → All mainly who can write → semantic if not semantic then → Lexical
- * If we write float x,a,b then semantic analyser comes before LA, stores these variables in symbol table.
- * If semantic analyser has already written the entries in symbol table and in the program float x,a,b come again, SA comes into picture again to add them to symbol table. But since they are present already, ~~sem. A.~~ Sem. A. gives Multiple Declaration Error.
- * After compilation, we get Machine Dependent Code.
- * Some compilers stop at ICG phase. The opf of ICG is still machine independent code. Java implements this and hence we get Byte code. ∵ Java is platform independent / highly portable.
- * Compiler is a s/w which converts a language to another language.

Comparing C, Java :- compilation : Java faster (has only 4 phases)
 Running : C faster (we already have LLL)

There are some languages that don't require any compilation (like python). These are interpreted lang & they are very slow. Here interpreter takes in line by line, converts the line into ill & then executes it.



FRONT END OF COMPILER

CD MIDDLE LEVEL.

Target code generation, BACKEND OF COMPILER.

Compiler has a large amt. of code when a prog. has to be compiled, the compiler's code has to be executed.

To execute compiler's code, it should be present in main memory. So we run compiler's code phase by phase.

Bo

If total compilation is done at a time then it is known as single pass compiler otherwise multipass compiler.

If total compilation is avoided adv.

Single pass compilation :- less time

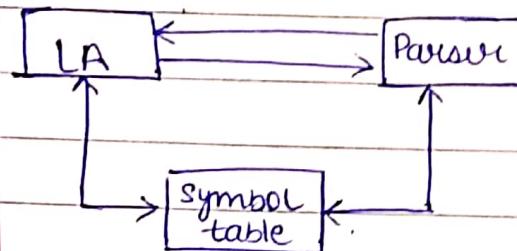
multi pass compilation :- ~~less space~~

disadv.

more space

more time

CH-2 LEXICAL ANALYSER:-



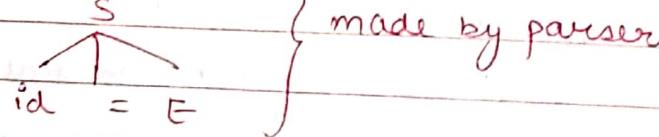
	1	2	3	4	5	6	7	8	9	row	col
	rn	rn									
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16											
17											
18											
19											
20											
21											
22											
23											
24											
25											
26											
27											
28											
29											
30											
31											
32											
33											
34											
35											
36											
37											
38											
39											
40											
41											
42											
43											
44											
45											
46											
47											
48											
49											
50											
51											
52											
53											
54											
55											
56											
57											
58											
59											
60											
61											
62											
63											
64											
65											
66											
67											
68											
69											
70											
71											
72											
73											
74											
75											
76											
77											
78											
79											
80											
81											
82											
83											
84											
85											
86											
87											
88											
89											
90											
91											
92											
93											
94											
95											
96											
97											
98											
99											
100											

Program

- ① Parser begins the process by making parse tree
 $S \rightarrow fE \mid wE \mid AE \mid BE \mid id = E$

But it has confusion, which one to take so it asks LA to give token().

- ② LA ~~is~~ is now present at 1st row 1st col. and starts reading. It gets to know that first 3 characters of 1st row give 1st token. It then returns that token (id) to parser.



- ③ Now again parser ends giveToken() to LA. Now LA reads next token as "id" but parser was expecting ". So syntax analyser informs error handler that syntax error occurred.

- ④ Error handler asks LA where it is right now & it makes an entry of syntax error along with LA location.

- ⑤ Just ~~when~~ when id comes into parse tree, semantic analysis begins its working by checking its ...

∴ All 3 phases LA, Syntax A. and Semantic A. are working simultaneously.

→ Lexical analysis is first phase of compiler which is also known as SCANNER. It will divide the given program into meaningful strings which are known as tokens.

→ Different types of tokens are :-

- ① Identifier
- ② Operator
- ③ Keyword
- ④ Constant
- ⑤ Special symbols.

→ Responsibility of lexical analyser :-

- ① Dividing the given program into tokens.
- ② It will eliminate white space characters.
- ③ It will ignore/eliminate comment lines.
- ④ It will help in giving error messages by providing row no. & column no.

Ex

Find no. of tokens present in the following C program :-

int main()

{

/* Finding max of a & b */

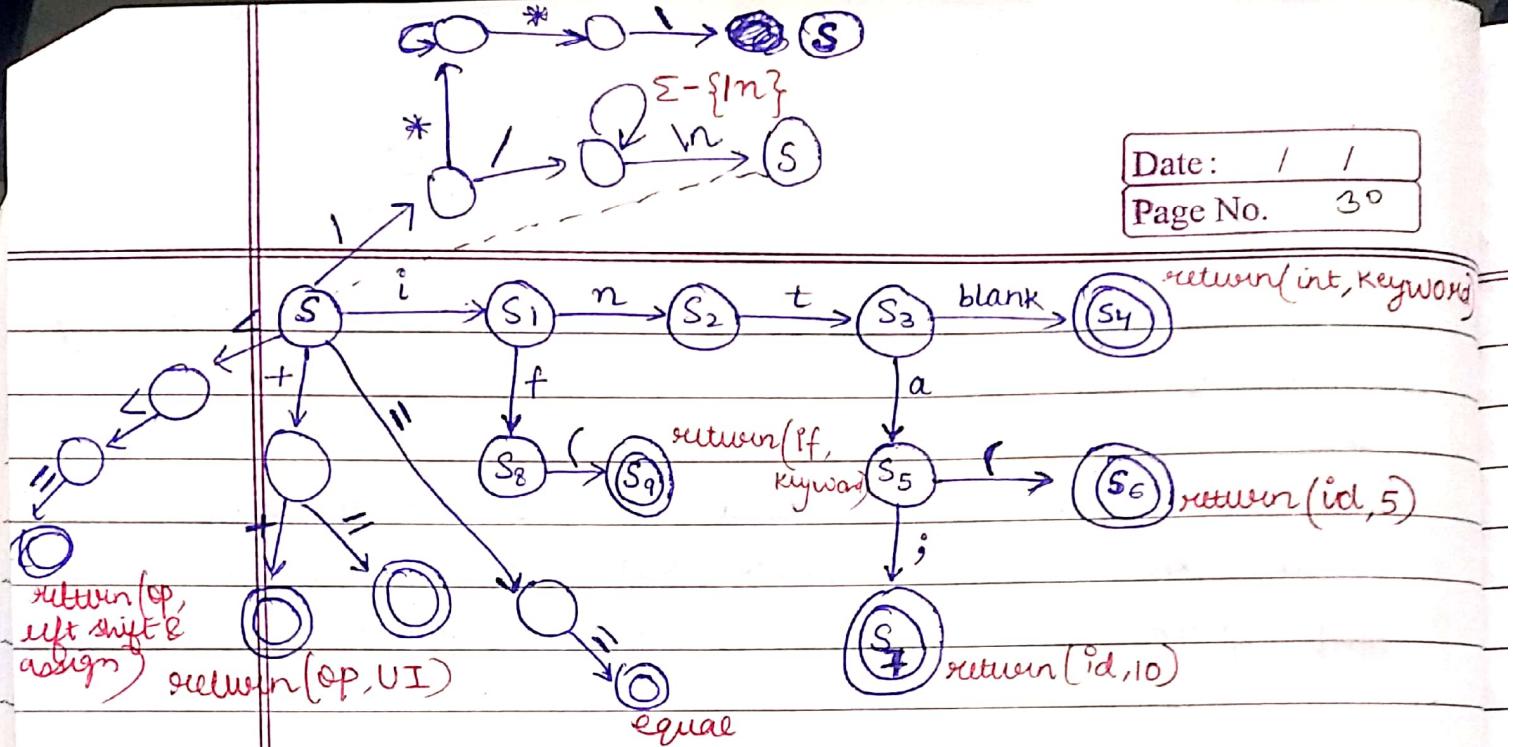
int a = 10, b = 20, max;

if (a < b)

max = b;

else

ans = 37



a is token
 b is token
 c is token } abc is token. Hence take the longest matching token.

If at start symbol we get + then go further

if blank → + is token

if + → ++ is token.

$+ = \{$ & $\div = \Rightarrow$ not present

$- = \{$ all present in

$* = \{$ C op. list

$/ = \{$

$\% = \{$ present in C

$\& = \{$ operator list

e.g. int main()

{ }

13 x = a + b. * c;

int x, a, b;

printf ("you and your family got hell %.d %.d, %d, %d)

Ans. We will not go inside " ". because 123.4 is 1 token & not 5. "a" is 1 token. Similarly anything inside " " is one string.

ex 3 main ()

{

$$x = a + b - c;$$

$$y = ++++--== */*++++---=-=-=$$

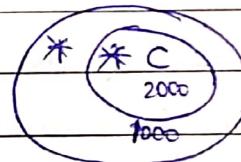
$$***+++=*=***/+-+/-;$$

printf ("%d %d %d", a, *a, **a, ***a, &a, &&a);

}

ans

a	b	c
10	1000	2000
1000	2000	3000



* is applied one at a time so they are 2 tokens

whereas ++a here ++ is applied on a to get 11.

∴ ans 49

ex

main ()

{

$$x = a + b;$$

$$y = xab - 123abcdef***ghi^*/* **$$

$$x = ab + cd - gh abcda / *** * / abc d$$

}

ans24

variable name cannot start with integer

if the comment never ends. & finally

entire program is over then it gives

token error. Also if "entire program

ends within /* */ then also token error.

$$y = xab - --- /* | ab "y = xab - ---$$

$$x = ---$$

$$x = ---$$

Q

main()

{

int x = 10;

int x = 10;

/* comment */ int x = 10;

}

ans

22

Q

Which one of the following string is said to be a token without looking at next I/P character.

- a) return
- b) if
- c) main
- d) +

a) cannot be because return followed by (or blank can be a keyword.

b) if cannot be because it can be keyword.

(c) main can be keyword or main can be ~~variable~~ variable operator

(e) ; can be a string or [, () ++]

ans will be where what follows the op does not impacts its meaning.

Q

T₁: a? (b/c)* a

T₂: b? (a/c)* b

T₃: c? (b/a)* c

(a) $T_1 T_2 T_3$ (b) $T_1 T_1 T_3$ (c) $T_2 T_1 T_3$ (d) $T_3 T_3$ a) $T_1 :- bba$ $T_2 :- acab$ $T_3 :- c$ $T_1 :-$ $T_1 :-$ $T_3 :-$ $T_3 :- bbaac$ $T_3 :- abc$ longest matching

If total string came from just 1 token then
ans would be that TOKEN.

~~PARTIAL~~

CH-3 PARSER

- * The process of deriving the string from the given grammar is known as derivation (Parsing).
- * Depending upon how derivation is performed, we have 2 types of parser
 - (1) Top down parser (2) Bottom up parser

WORKING OF TOP DOWN PARSER :-

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

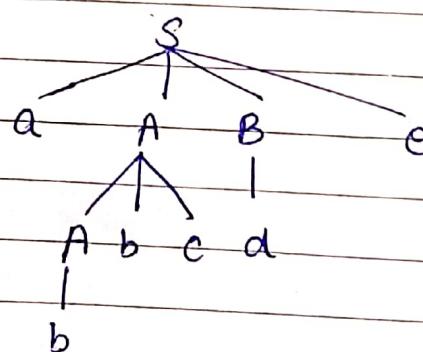
$$i/p :- abcd e \$$$

Lookahead symbol

Some T-D parser also apply back-track to decide which possibility to take TD with back-track : recursion
disambiguation

TD without back-track : LL(1)

Top-down begins from start symbol & proceeds to string.



Every parser follows leftmost derivation only.
because string is read from left to right

At variable A we have dilemma which one to choose out of Abc or b . This is the difficulty with top down parser as a variable has more than one possibility.

WORKING OF BOTTOM UP PARSER :-

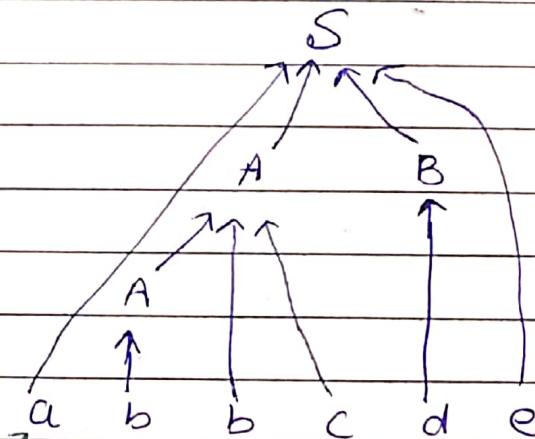
- It starts from string & goes to start symbol.
- Bottom up parser also scans the string from left to right.
- These are generally called LR parser.

$$S \rightarrow aABe$$

$$A \rightarrow Abc/b$$

$$B \rightarrow d$$

i/p :- abbcde



④ $S \rightarrow aABe$

③ $B \rightarrow d$

② $A \rightarrow Abc$

① $A \rightarrow b$

No induction when it was read first time. So skip

* Bottom up parser difficulty is which string to compress which one shouldn't be compressed.

* Handling substring is difficult here.

* Bottom up parser follows no derivation because no variables.

more
apt. *

Bottom up looks like right most parser down parser in reverse.

NO Parser follows Rightmost Derivation.

→ It follows reverse of rightmost derivation.

→ Identifying correct handle (substring) is always difficult in bottom up parser.

TOP DOWN PARSER

with backtracking

(Recursive descent)
Parser

without backtracking

(LL(\pm) parser)
etc.Non recursive descent
or

RECURSIVE DESCENT PARSER

start symbol.
 $S()$

{

select any production of S $(S \rightarrow x_1, x_2 x_3 \dots x_k)$ for ($i = 1 ; i \leq k ; i + 1$){ if (x_i is variable) $x_i()$;

else

if ($x_i == \text{lookAheadSymbol}$)
incr. pointer

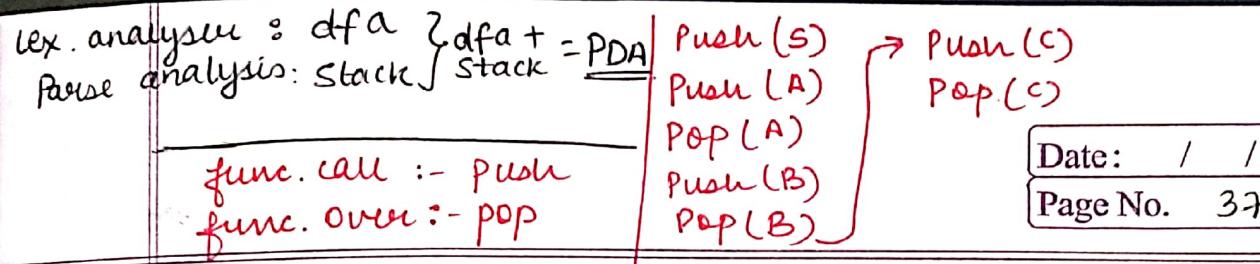
else

error \Rightarrow try another possibility
 \therefore Backtracking

{

eg.

 $S \rightarrow ABC \mid DEF \mid GHI$ $A \rightarrow ab \mid gh \mid mn$ $B \rightarrow cd \mid ij \mid op$ $C \rightarrow ef \mid kl \mid qr$ $D \rightarrow d \quad G \rightarrow g$ $E \rightarrow e \quad H \rightarrow h$ $F \rightarrow f \quad I \rightarrow i$



Ans Note that every parser requires an i/p string & grammar as its input.

- ① Recursive begins from start symbol with backtracking
- ② we have to derive a (i^{th} i/p symbol \Rightarrow lookahead symbol)
- ③ Select any production of S $S \rightarrow ABC$
- ④ Now since $K=3$, loop goes 3 times

$i=1$	$i=2$	$i=3$
A()	B()	C()
$A \rightarrow ab$	$B \rightarrow cd$	$C \rightarrow ef$
$i=1$	$i=2$	$i=1$
a b	c d	e f
check if $a == LAS$ Yes! ∴ incl. i/p ptr.	check if $b == LAS$ Yes! ∴ incl. i/p ptr.	check if $c == LAS$ Yes! ∴ incl. i/p ptr.
		check if $d == LAS$ Yes! ∴ incl. i/p ptr.
		check if $e == LAS$ No! ERROR. \Rightarrow Backtrack

If C failed in identifying i/p string, then entire work done by A, B, C has to be cancelled and another production of S has to be taken.

$C \rightarrow K\ell$	$i=1$	$i=2$
$i=1$	K	ℓ
check if $K == LAS$		
No!		
Error \Rightarrow Backtrack		
$C \rightarrow q\varphi$	$i=1$	$i=2$
q	φ	
check if $q == LAS$		
Yes! ∴ incl. i/p pointer		
check if $\varphi == LAS$		
Yes! ∴ incl. i/p pointer		

disadv: Time wasted in backtracking

eg2

$$S \rightarrow aS|b$$

i/p : aaab \$

S()

$$S \rightarrow aS$$

i = 1

a

a == LAS

Yes.

Incr. ptr.

i = 2

S()

$$S \rightarrow aS$$

i = 1

a

a == LAS

Yes.

Incr. i/p

ptr

i = 2

S()

$$S \rightarrow aS$$

i = 1

a

a == LAS

Yes.

Incr. i/p

ptr

i = 2

S

$$S \rightarrow aS$$

i = 1

a

a == LAS

No

Error \Rightarrow Backtrack

$$S \rightarrow b$$

i = 1

b == LAS

Yes.

Incr. ptr.

eg3

$$S \rightarrow Sa|b$$

i/p : baaa \$

S()

$$S \rightarrow Sa$$

S()

$$S \rightarrow Sa$$

S

i

- stack overflow meg.
- infinite loop.

Next recursion!!

eg $S \rightarrow ABC \mid DEF \mid GHI$.

$A \rightarrow ab \mid abc \mid abcd$

$B \rightarrow ef \mid gh \mid ij$

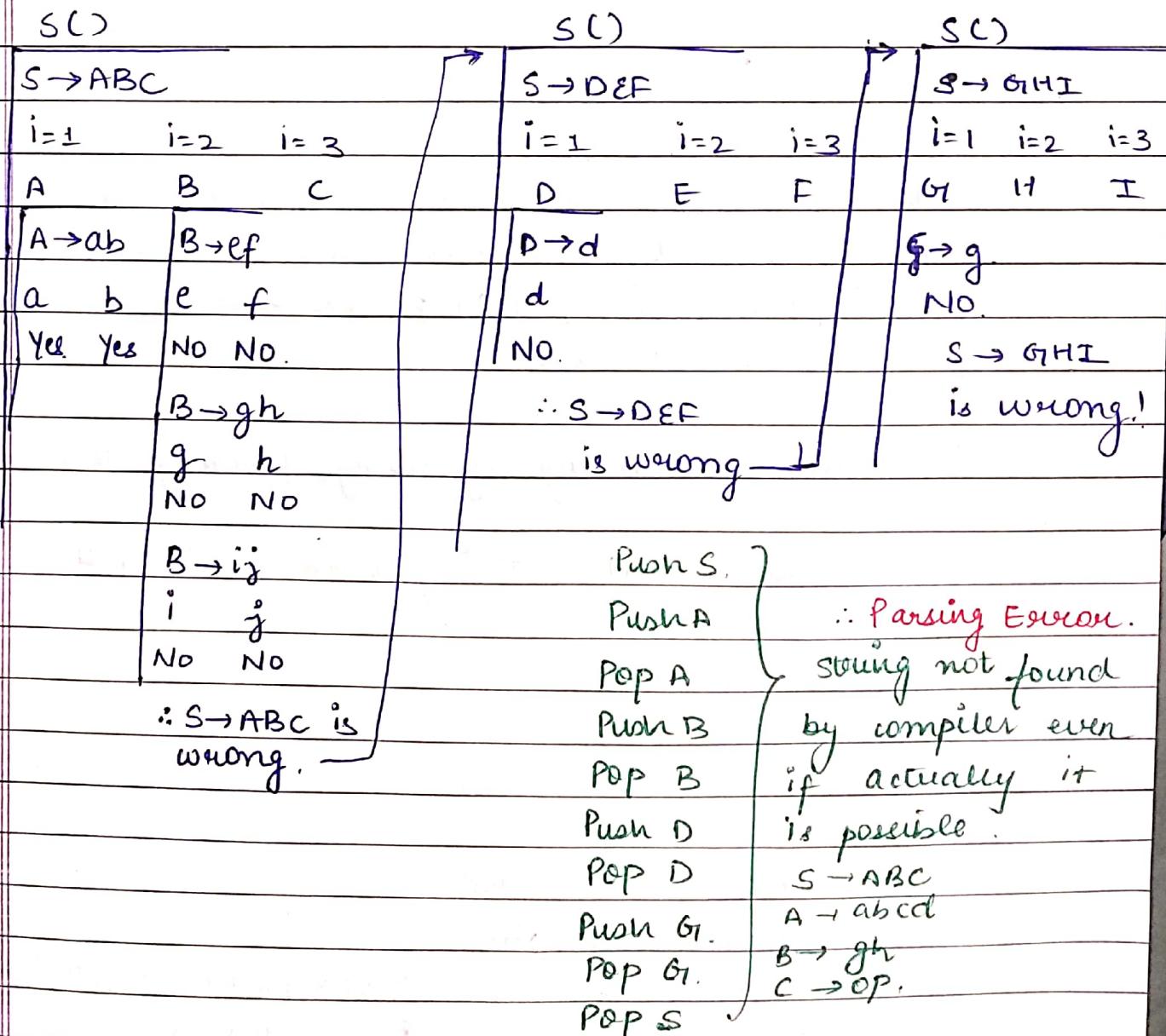
$C \rightarrow kl \mid mn \mid op$

$D \rightarrow d \quad G_1 \rightarrow g$

$E \rightarrow e \quad H \rightarrow h$

$F \rightarrow f \quad I \rightarrow i$

i/P :- abcdghop \$



- In recursive descent parser, every variable is having recursive func. (same code).
- If grammar contain left recursion, recursive descent parser may go to infinite loop.
- If grammar contain left factoring, recursive descent parser may give parser error.
- In recursive descent parser, lot of time is wasted in form of backtracking (exponential time)

Imp:-

• LL(1) PARSER :-

→ It has same algo as that of recursive descent. Only diff. is that instead of selecting any production, LL(1) parser selects the correct production.

→ Without backtracking by using parsing table.

→ To construct LL(1) parsing table, we need 2 functions :-

1. First()

2. Follow()

FIRST() :-

→ found for variables.

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

$$\text{First}(S) = \text{First}(abc) + \text{First}(def) + \text{First}(ghi)$$

$$= \text{First}(a) + \text{First}(d) + \text{First}(g)$$

$$= \{a, d, g\}$$

First (Terminal) \Rightarrow Terminal

First (ϵ) $\Rightarrow \epsilon$

First (A) :- contains set of all terminals present in 1st position of every string derived by A.

eg $S \rightarrow ABC$

$A \rightarrow a | d | e | f | \epsilon$

$B \rightarrow b | g | h | i | \epsilon$

$C \rightarrow c | j | k | \epsilon$

$$\text{First}(S) = \text{First}(ABC)$$

$$= \text{First}(A)$$

$$= F_i(a), F_i(d), F_i(e), F_i(f), F_i(\epsilon)$$

$$= a, d, e, f, \cancel{F_i(B \& C)}$$

↓

$$F_i(B)$$

$$b, g, h, i, \cancel{F_i(C)}$$

↓

S is generating 12. kinds of strings where each string begins from a, d, e, f, b, g, h, i, c, j, k, ϵ .

ex

$S \rightarrow aBDh$

$$\text{First}(S) = \{a\}$$

$B \rightarrow bC | \epsilon$

$$\text{First}(B) = \{b, \epsilon\}$$

$C \rightarrow c | \epsilon$

$$\text{First}(C) = \{c, \epsilon\}$$

$D \rightarrow FE$

$$\text{First}(D) = \{g, f, \epsilon\}$$

$F \rightarrow g | \epsilon$

$$\text{First}(F) = \{g, \epsilon\}$$

$E \rightarrow f | \epsilon$

$$\text{First}(E) = \{f, \epsilon\}$$

First(S) - go LHS of production & find S

Follow(S) : go RHS of production & find S

Date: / /
Page No. 42

eg.

$$S \rightarrow aSbS \mid bSas \mid \epsilon$$

$$\text{First}(S) = \{a, b, \epsilon\}$$

eg

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

$$\text{First}(S) = \{a, b\}$$

$$\text{First}(A) = \{\epsilon\}$$

$$\text{First}(B) = \{\epsilon\}$$

$$\text{First}(S) = \text{First}(AaAb), \text{First}(BbBa)$$

$$= \text{First}(A), \text{First}(B)$$

$$= \cancel{\text{First}(aAb)}, \cancel{\text{First}(bBa)}$$

a

b

$$= \{a, b\}$$

FOLLOW() :-

Follow(S) contains set of terminals present
in 1st position of every string immediate
right of S.

{a} = First(~~(S)~~) means it generates a string which
begins with a.

{b} = Follow(S) means it generates a string where after
S whatever string comes has its first letter
as b.

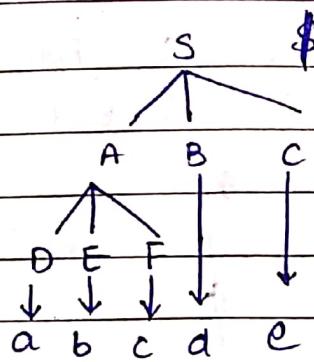
eg.

$$S \rightarrow ABC$$

$$A \rightarrow DEF$$

$$D \rightarrow a \quad B \rightarrow d$$

$$E \rightarrow b \quad C \rightarrow e$$



$$\text{FOLLOW}(D) = \text{First}(EF) = \text{First}(E) = b$$

$$\text{FOLLOW}(F) = \text{Follow}(A) = \text{First}(BC) = \text{First}(B) = d$$

$$\text{Follow}(C) = \text{Follow}(S) = \text{First}(\$) = \$$$

$\rightarrow A \rightarrow DEF$; follow(F) means after F who is there.

But there is nothing means $A \rightarrow DEF$ production is over. So we need to find follow(A)

eg2 $S \rightarrow ABC$

$$A \rightarrow DEF$$

$$D \rightarrow a$$

$$E \rightarrow \epsilon$$

$$F \rightarrow \epsilon$$

$$B \rightarrow G$$

$$C \rightarrow e$$

$$i/p: ace\$$$

$$\text{FOLLOW}(D) = \text{First}(EF) = \{\$ \} \text{First}(E) = \{ \epsilon \} \text{First}(F) = \{ c \}$$

$$\text{FOLLOW}(F) = \text{Follow}(A) = \text{First}(BC) = \{ e \}$$

$$\text{FOLLOW}(C) = \text{Follow}(S) = \text{First}(\$) = \{ \$ \}$$

eg3 $S \rightarrow ABC$

$$A \rightarrow DEF$$

$$D \rightarrow G$$

$$E \rightarrow \epsilon$$

$$B \rightarrow G$$

$$C \rightarrow G$$

$$\text{Follow}(B) = \text{First}(C) = \{\epsilon\} = \text{Follow}(S) = \text{First}(\$) = \{\$,\epsilon\}$$

$$\text{Follow}(D) = \text{First}(EF) = \text{First}(E) = \{\epsilon\} \quad \text{First}(F) = \{\epsilon\}$$

$$= \text{Follow}(A) \quad \text{Follow}(A) \text{ is } \text{Follow}(B)$$

$$= \text{First}(BC) = \text{First}(B) = \{\epsilon\} = \text{First} - \{C\} = \{\epsilon\}$$

$$= \text{Follow}(S) = \text{First}(\$) = \{\$\}$$

$\text{Follow}()$ Never gives $\{\epsilon\}$. It will give $\{\$\}$ if no letter comes.

Rules to find Follow:-

assume A is variable

- ① If A is start symbol then $\text{Follow}(A) = \{\$\}$
- ② $B \rightarrow CADG \quad \left. \begin{array}{l} \\ D \rightarrow b \end{array} \right\} \text{Follow}(A) = \text{First}(DG) = \text{First}(D) = \{b\}$
- ③ $B \rightarrow CA \quad \text{or} \quad B \rightarrow CAD$
 $D \rightarrow E$

After A
no one

$$\text{Follow}(A) = \text{Follow}(B)$$

Q

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow E + TE' \\ T &\rightarrow FT^* \\ T^* &\rightarrow \epsilon \mid *FT^* \\ F &\rightarrow id \mid (E) \end{aligned}$$

First()

$$\begin{aligned} E &: \{id, (\} \quad \checkmark \\ E' &: \{\epsilon, +\} \quad \checkmark \\ T &: \{id, (\} \quad \checkmark \\ T^* &: \{\epsilon, *\} \quad \checkmark \\ F &: \{id, (\} \quad \checkmark \end{aligned}$$

Follow()

$$\begin{aligned} &\{\), \$\} \quad \checkmark \\ &\{\), \$\} \quad \checkmark \\ &\{\epsilon + ,), \$\} \quad \checkmark \\ &\{\epsilon + , \$\} \quad \checkmark \\ &\{\epsilon + , \$\} \quad \checkmark \end{aligned}$$

Q $S \rightarrow aSbS \mid bSas \mid \epsilon$

$$\text{First}(S) = \{a, b, \epsilon\}$$

$$\text{Follow}(S) = \{b, a, \$\}$$

Q $S \rightarrow AbAb \mid BaBa$

$$A \rightarrow \epsilon$$

$$B \rightarrow C$$

First()		Follow()	
S	{b, a}		{\\$}
A	ϵ		{b}
B	ϵ		{a}

Q $S \rightarrow (L) \mid a$

$$L \rightarrow SL'$$

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

First()		Follow()	
S	(a		(\$,)
L	(a)
L'	$\epsilon, ,$)

Q $S \rightarrow aBDh$ $D \rightarrow FE$

$B \rightarrow bB \mid E$ $E \rightarrow f \mid \epsilon$

$C \rightarrow cC \mid \epsilon$ $F \rightarrow g \mid \epsilon$

First		Follow	
S	a		\$
B	b, ϵ		g, f, h
C	c, ϵ		
D	g, f, ϵ		h

LL(1) PARSING TABLE CONSTRUCTION ALGO :-

For each production $A \rightarrow \alpha$ repeat the following

(1) Add $A \rightarrow \alpha$ under $m[A, B]$ & $b \in \text{First}(\alpha)$

(2) If $\text{first}(\alpha)$ contains ϵ then add $A \rightarrow \alpha \Rightarrow$
under $m[A, C]$ & $c \in \text{follow}(A)$

Construct LL(1) parsing table for following grammar.

$$S \rightarrow (L) / a$$

$$L \rightarrow SL'$$

$$L' \rightarrow \epsilon, SL'$$

No. of rows in parsing table = No. of variables

No. of cols. in parsing table = No. of terminals + \$

	,	a	()	\$	
S		$S \rightarrow a$	$S \rightarrow (L)$			
L)			
L'	$L \rightarrow SL'$	$L \rightarrow SL'$				
	$L \rightarrow , SL'$			$L \rightarrow)$		

* If a grammar has 20 variables & 25 terminals
then size of parsing table = 20×26 .

* Only 1 entry should be present in each cell.

$L \rightarrow , SL' \Rightarrow$ Parser should be at L' row

when LAS = , then take this prod. to $M[L', ,]$

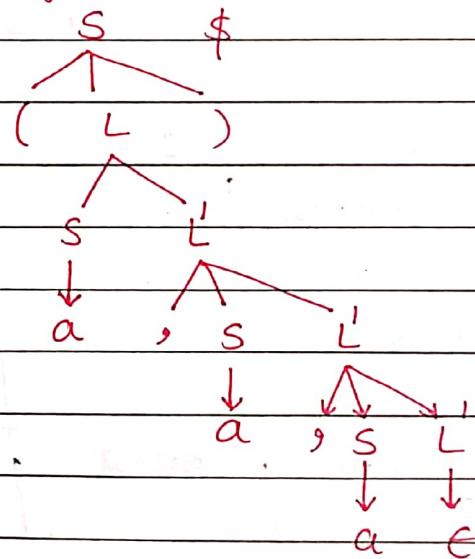
$$L \rightarrow SL' \Rightarrow m[L, \text{First}(SL')] \\ \Downarrow \text{First}(S) \\ \Downarrow (, a)$$

$$L' \rightarrow e \Rightarrow M(L', \text{first}(e))$$

e but e can not present
 $\text{Follow}(L') \Rightarrow)$

$$M[L',)]$$

For the given grammar, i/p : (a, a, a) \$



* If while deriving a string, we try to access a block which is empty (no production present) then it is an invalid string & parse error will be generated.

* NO. of empty blocks in LL(1) = error entries.

* If while writing productions in table, one block contains more than 1 entry. Then given grammar is not LL(1).

multiple productions. So given grammar is LL(1).

eg

Check whether given grammar is LL(1) or not.

$$S \rightarrow AaAb \mid BbBA$$

$$A \rightarrow C$$

$$B \rightarrow C$$

	First	Follow
S	a, b	\$
A	C	a, b
B	C	a, b, \$

	a	b	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBA$	
A	$A \rightarrow C$	$A \rightarrow C$	
B	$B \rightarrow C$	$B \rightarrow C$	Followed

Q

$$S \rightarrow aSbS \mid bSas \mid C \quad \text{LL(1) ??}$$

$$\text{First}(S) = \{a, b, C\}$$

$$\text{Follow}(S) = \{b, a, \$\}$$

	a	b	\$
S	$S \rightarrow aSbS$	$S \rightarrow bSas$	
	$S \rightarrow C$	$S \rightarrow C$	$S \rightarrow C$

Not in LL(1)

Q

$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow \epsilon \quad | \quad * \quad | \quad TE' \\
 T &\rightarrow FT' \\
 T' &\rightarrow \epsilon \quad | \quad + \quad | \quad FT' \\
 F &\rightarrow id \quad | \quad (E)
 \end{aligned}$$

First

Follow

E	id, (\$,)			
E'	$\epsilon, *$	\$,)			
T	id, (* \$,)			
T'	$\epsilon, +$	* \$,)			
F	id, (+ *, \$,)			

id () + * \$

E	$E \rightarrow TE'$	$E \rightarrow TE'$				
	(1)	(1)				
E'			$E' \rightarrow \epsilon$		$E' \rightarrow * TE'$	$E' \rightarrow \epsilon$
			(2)		(3)	(2)
T	$T \rightarrow FT'$	$T \rightarrow FT'$				
	(4)	(4)				
T'			$T' \rightarrow \epsilon$	$T' \rightarrow + FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
			(5)	(6)	(5)	(6)
F	$F \rightarrow id$	$F \rightarrow (E)$				
	(7)	(8)				

Q

$$\begin{aligned}
 S &\rightarrow aBDh \\
 B &\rightarrow bC \quad | \quad \epsilon \\
 C &\rightarrow cB \quad | \quad \epsilon \\
 D &\rightarrow FE \\
 E &\rightarrow CB \quad | \quad \epsilon \\
 F &\rightarrow bC \quad | \quad \epsilon
 \end{aligned}$$

	a	b	c	h	\$
S	$S \rightarrow aBDh$ (1)				
B		$B \rightarrow E$ (3)	$B \rightarrow E$ (3)	$B \rightarrow E$ (3)	
C		$B \rightarrow bC$ (2)	$C \rightarrow c$ (5)	$C \rightarrow cB$ (4)	$C \rightarrow c$ (5)
D					$D \rightarrow PE$ (6)
E				$E \rightarrow CB$ (7)	$E \rightarrow E$ (8)
F		$F \rightarrow bC$ (9)	$F \rightarrow E$ (10)	$F \rightarrow E$ (10)	

	First	Follow
S	a	\$
B	b, E	b, c, h, ∅, E, E
C	c, E	b, c, h, ∅, E, E
D	b, c, E	b, c, d, E, h
E	c, E	h
F	b, E	c, h

∴ Not LL(1)

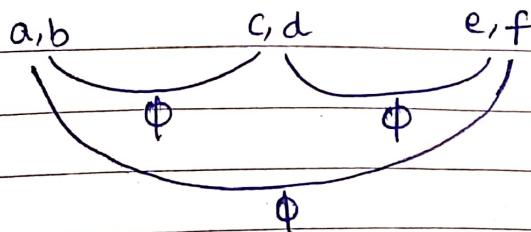
SHORTCUT TO CHECK GRAMMAR LL(1) OR NOT.

NOTE:- If G₁ has no null productions

$$S \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$$

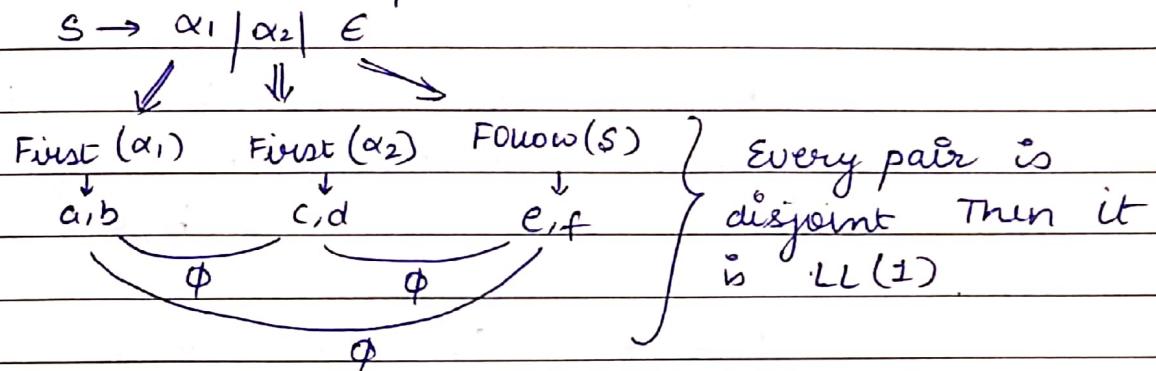
↓ ↓ ↗

First(α_1) First(α_2) First(α_3)



} each pair is disjoint. Then it is LL(1).

② If G contains null productions.



Q Check if the grammar is LL(1) or not?

$$S \rightarrow E/b$$

$$E \rightarrow b$$

$$S \rightarrow E \Rightarrow \text{First}(E) = b \quad \} \text{ Not LL(1).}$$

$$S \rightarrow b \Rightarrow \text{First}(b) = b$$

Q $S \rightarrow aS/a$

$$S \rightarrow aS \Rightarrow \text{First}(aS) = a \quad \} \text{ Not LL(1).}$$

$$S \rightarrow a \Rightarrow \text{First}(a) = a$$

* If grammar contains left factoring then it cannot be LL(1).

Q $S \rightarrow Sa/b$

$$S \rightarrow Sa \quad \text{First}(Sa) = \text{First}(S) = \text{First}(SA), b \quad \}$$

$$S \rightarrow b \quad \text{First}(b) = b$$

Not disjoint

\Rightarrow Non LL(1)

* If grammar contains left recursion then it cannot be LL(1).

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

$$\begin{array}{ll} S \rightarrow aSbS & \Rightarrow \text{First}(aSbS) = a \\ S \rightarrow bSaS & \Rightarrow \text{First}(bSaS) = b \\ S \rightarrow \epsilon & \Rightarrow \text{Follow}(S) = \$, b, a \end{array} \quad \left. \begin{array}{l} \text{Not} \\ \text{disjoint} \end{array} \right\}$$

NOT LL(1) as it is ambiguous also.

* Ambiguous grammar can never be LL(1)

Note:- Every regular grammar need not be LL(1).

eg. $S \rightarrow aS \mid a$ (NOT LL(1))

This is because that reg. grammar may contain left recursion or right recursion or ambiguity

* If given grammar is LL(1) then it cannot contain
 ↗ left recursion
 ↗ left factoring
 ↗ ambiguity

$$S \rightarrow ab \mid ac \mid ad \quad \Rightarrow \text{left factoring exists}$$

LL(1) fail

LL(2) means take 2 symbols as lookahead
 ab, ac, ad

∴ LL(2) ✓

in terms of power!

$$\text{LL}(1) \subseteq \text{LL}(2) \subseteq \text{LL}(3) \subseteq \dots \subseteq \text{LL}(K)$$

\leftarrow leftmost derivation
 scan i/p from left
 \downarrow
 \leftarrow reverse of rightmost derivation

Date: / /
Page No. 53

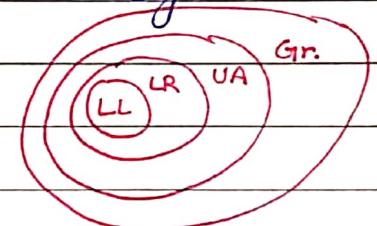
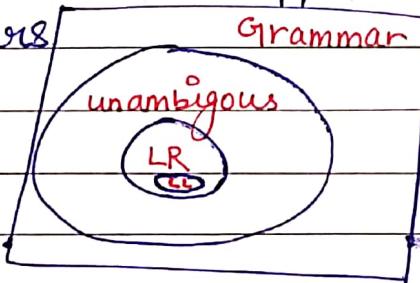
BOTTOM UP PARSER:

LR - Parser.

Operator Precedence Parser.

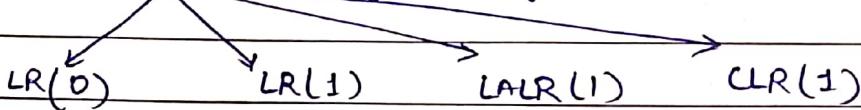
LR PARSER :-

- ① LR parsers will be applicable only for unambiguous grammars



- Every LR is unambiguous but every unambiguous need not be LR.
- Every LL is LR. but every LR need not be LL.

LR parsers are of 4 types



For all LR parsers, parsing algorithm is same, but ~~parsing~~ ^{derivation} table is different.

Imp LR PARSING ALGORITHM :-

Let X be the state number on top of stack & "a" is lookahead symbol.

1) if action $[x, a] = s_i$ then shift a & inc. i/p pointer.

2) if action $[x, a] = s_j$ and j^{th} production is $\alpha \rightarrow \beta$ then pop $2 * |\beta|$ symbols & replace by α .

If s_{m-1} state number below & then push goto $[s_{m-1}, \alpha]$.

- 3) If action $[x, a] = \text{blank}$ then parsing Error
- 4) If action $[x, a] = \text{accept}$ then successful completion of Parsing

ex $S \rightarrow AA$

$A \rightarrow aA/b$

i/p :- abab \$

AA { 3 handles
aA
b }

action			Goto	
	a	b	\$	
0	s_3	s_4		1 2
1			acc	
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	

LR(0) parsing table.

No. of rows = no. of states in DFA

No. of col = Action + goto

terminal + \$ variables.

→ LR parse table is almost double to that of LL parse table.

→ LR parser are more powerful than LL as they can accept more languages.



Date: / /

Page No. 55

BANCHITA

i/P	state of stack $0 \Rightarrow 0a3$ TOS	Reduction
<u>abab\$</u> LAS	$0a3 \Rightarrow 0a3b4$ TOS	go to A-3 = 6 to remember A in stack.
when red. goes on, i/p doesn't inv.	$0a3b4 \Rightarrow 0a3A6 \Rightarrow 0A2$ 0A2 $\Rightarrow 0A2a3$	A $\rightarrow b$ \rightarrow b replaced by A Remove $2x1 = 2$ symbols from stack.
b\$	$0A2 \Rightarrow 0A2a3$	A $\rightarrow aA$
\$	$0A2a3 \Rightarrow 0A2a3b4$	$2x2 = 4$. Pop 4 sym. a3A6 & replace by A. Then goto J $0A = 2$.. Push 2.
	NOW \$-4 $\Rightarrow r_3 \Rightarrow A \rightarrow b$ Push b4, replace by A then 0A2a3A6	Push $a \rightarrow aA$ 2A = ⑥
	NOW \$-6 $\Rightarrow r_2 \Rightarrow A \rightarrow aA$ Push $a \rightarrow aA$ 2A = ⑤	Push $a \rightarrow aA$ 2A = ⑤
	NOW \$-5 $\Rightarrow r_1 \Rightarrow S \rightarrow AA$ Push $S \rightarrow AA$ 2A = ④	Push $S \rightarrow AA$ 2A = ④
OS1		Push OS = 1
	NOW \$-1 $\Rightarrow acc$ <u>Stop</u>	Successful completion

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow \textcircled{1} A \mid b \end{aligned} \quad \left\{ \begin{array}{l} i/P = aabb\$ \\ \text{i/P.} \end{array} \right.$$

i/P.	stack	Reduction.
aabb\$	$0 \Rightarrow 0a3$	
abb\$	$0a3 \Rightarrow 0a3a3$	
bb\$	$0a3a3 \Rightarrow 0a3a3b4$	
b\$	$0a3a3b4$	$A \rightarrow b$ Pop b4, replace by A & push 6.
	$0a3a3A6$	$A \rightarrow aA$ Pop a3A6. replace by A & push 6.
	$0a3A6$	
b\$	$0A2 \Rightarrow 0A2a3b4$	
\$	$0A2 \Rightarrow 0A2b4$	$A \rightarrow aA$

VIABLE PREFIXES :- while doing bottom up parsing, the prefixes present in stack are called viable prefixes.

eg.

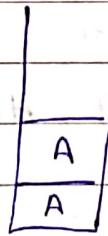


a, aa, aaa, aaab are viable prefixes.

For above grammar



Handle handle → NO other symbol can be pushed to BUP stack unless this handle is popped.



AA → viable prefix
AAA → Not viable prefix
b → viable prefix

ex3. i/p :- aaab \$

i/p	stack	Reduction
aaab \$	0a \Rightarrow 0a03	
aab \$	003 \Rightarrow 00303	
ab \$	00303 \Rightarrow 0030303	
b \$	0030303 \Rightarrow 0030303b4	$A \rightarrow b$
\$	0030303b4 \Rightarrow 0030303A6	$A \rightarrow aA$
	00303A6	$A \rightarrow aA$
	003A6	$A \rightarrow aA$
	0A2	$A \rightarrow aA$

17/11/2019

LR(0) PARSING TABLE CONSTRUCTION :-

To construct LR(0) parsing table, we need 2 functions

- i) closure()
- ii) goto()

$$G_1: S \rightarrow AA$$

Both G_1 & G' language are same

$$A \rightarrow aA/b$$

Only a unit production $S' \rightarrow S$

$$G': S' \rightarrow S$$

} Augmented grammar

$$S \rightarrow AA$$

} is added.

$$A \rightarrow aA/b$$

(as they have •)

↑ LR(0) items

$$S \rightarrow abc, S \rightarrow a.bc, S \rightarrow ab.c, S \rightarrow abc.$$

• Indicates how much production is over.

$S \rightarrow .abc$ - production not started

$S \rightarrow abc.$ - entire production over.

↳ final LR(0) item or

complete LR(0) item or

reduced LR(0) item

CLOSURE() :-

Closure() will be computed only for LR(0) items

e.g. closure($S \rightarrow .AA$) = ① $S \rightarrow .AA$ → (write eqⁿ as it is)

② $A \rightarrow .AA$ → $\{S \rightarrow .AA \text{ means parser is at } A\}$
 $\rightarrow .b$ → So we can take A's productions
Add . to A's production.

e.g. closure($S' \rightarrow S$) = ① $S' \rightarrow .S$ → (as it is. Now include S prodⁿ)
 $S \rightarrow .AA$ → (we can write only variable's prod)

$A \rightarrow .AA$? (as . is before terminals, hence)

eg 3 closure ($A \rightarrow \cdot aA$) :- ① $A \rightarrow \cdot aA$

Definition :- closure (I) :- ① add I

② if $A \rightarrow B.CDE$ is I and C is a variable which generates $C \rightarrow EFG$ is in grammar then add $C \rightarrow EFG$ to closure of I .

③ Repeat ② for every newly added LR(0) item until

* there is a terminal just after \cdot

GOTO :- requires a i/p

(i) LR(0) item

(ii) some input.

after, whatever is there, give it as i/p

ex ① Goto ($S \rightarrow \cdot AA$, $\overset{\uparrow}{A}$) = ① $S \rightarrow A.A$ (move the dot after the given i/p)
 ② $A \rightarrow \cdot AA$
 $\rightarrow \cdot b$ [whatever ans we get in ①, just do closure of it]

ex 2 Goto ($A \rightarrow \cdot AA$, a) = ① $A \rightarrow a.A$
 ② $A \rightarrow \cdot AA$

$\rightarrow \cdot b$

Definition Goto(I, x) :- ① add I by moving dot after x .
 ② Apply closure to first step.

LR(0) PARSING TABLE CONSTRUCTION ALGORITHM :-

① I_0 = closure (augmented LR(0) item)

② using I_0 construct DFA.

③ convert DFA to LR(0) Parsing table

Q1 construct LR(0) parsing table for following grammar

$$G_1: S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

(1)

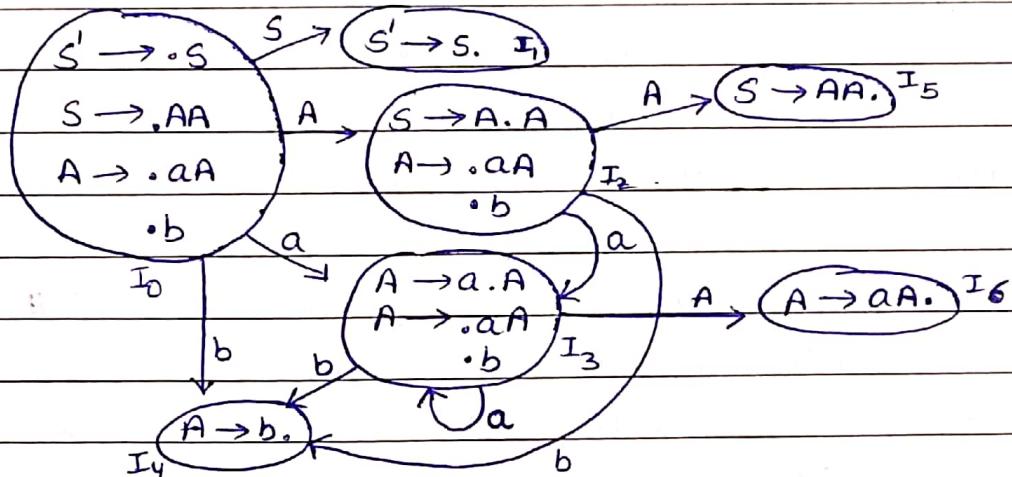
$$G'_1: S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

(2) closure ($S' \rightarrow \cdot S$)

(3)



	a (Action)	b	\$	S (goto)	A
I ₀	- S ₃	S ₄		1	2
I ₁			ACC		
I ₂	S ₃	S ₄			5
I ₃	S ₃	S ₄			6
I ₄	g ₃	g ₃	g ₃		since LR(0) is blind, we write reduced prod^n in opt.
I ₅	g ₁	g ₂	g ₁		

Diff b/w LR(0) & SLR(1) parsing table is that in LR(0) we write reduced production

blindly everywhere in action part. But in SLR(1) before writing it, we do some calculation to decide where reduced LR(0) item is written.

B/W LR(0) & SLR(1) :-

- ① Shift entries remain same True
- ② Reduced entries may differ True
- ③ Error entries may differ True
- ④ No. of states may differ False

Imp.

- Shift entries come because of non completed LR(0) item
- Reduced entries come because of completed LR(0) item
- Not every incomplete LR(0) item generates shift entry (Here see $I_0 \rightarrow A$, $I_0 \rightarrow S$)

- * * * • At a particular state, when a terminal is present just after dot then a shift entry is generated eg. at I_2 , 3 incomplete productions but only 2 shift entries as $A \rightarrow .aA$ & $A \rightarrow .b$
Only these 2 have terminal after ..

Augmented production is used to check whether S over or not because S occurs no where on RHS of given grammar.

If ~~dot~~ in a particular state, a production

in LR(0) Parsing table will

state. This is called RR conflict. Parser does not understand which one to pop from stack. $\boxed{s_1/s_3} \quad \boxed{s_1/s_3}$

If in a particular state, there are shift as well as reduced productions then LR(0) parser table will have S/R entry for that row. This is called SR conflict. Parser does not know whether it needs to push or pop.

RR
SR

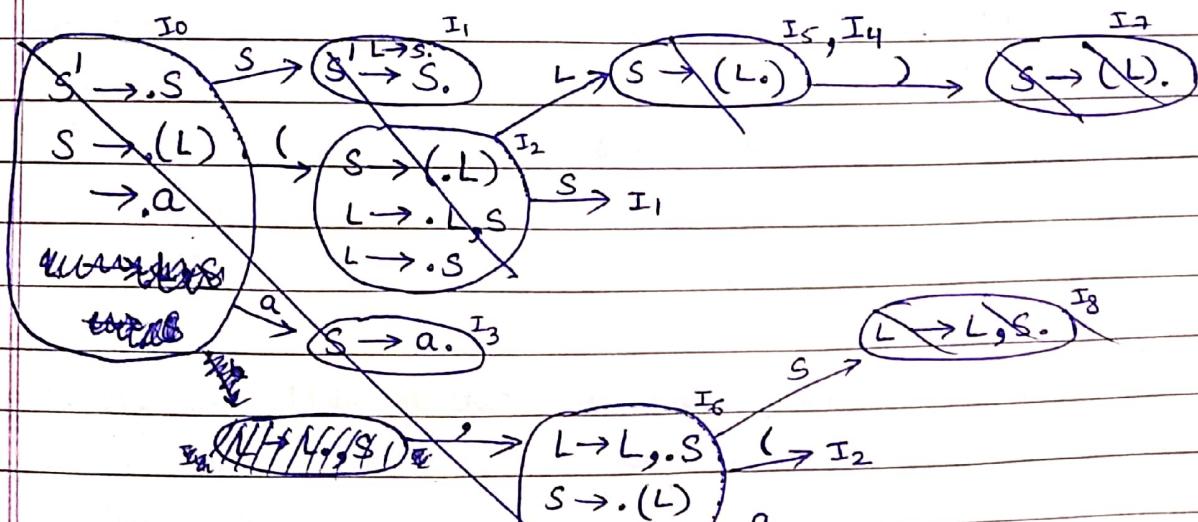
To get a conflict, minimum 2 productions are required & one of them should be reduced.

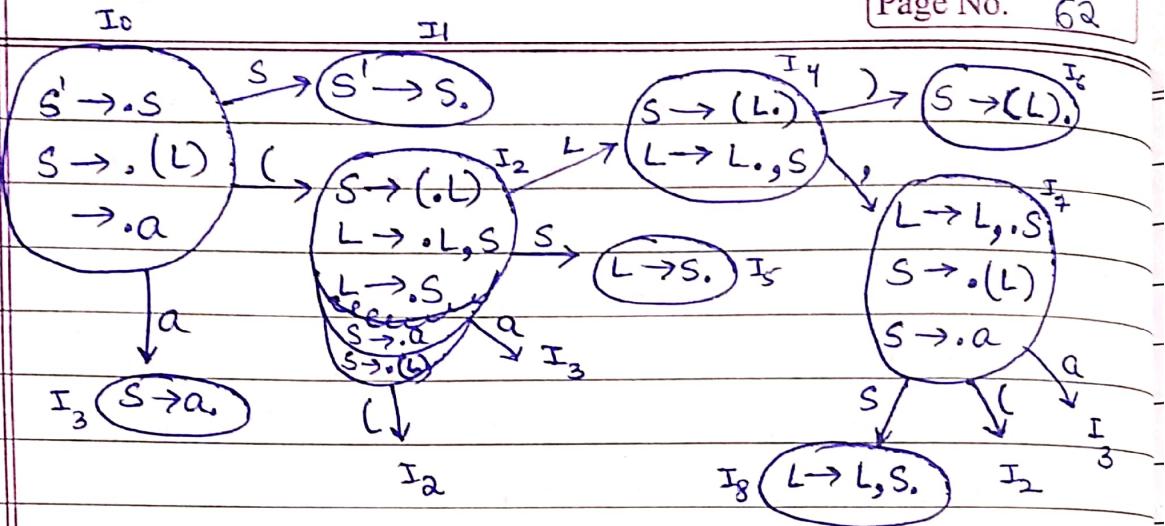
To check if grammar is LR(0) or not, draw DFA or LR(0) parsing table. If there is no conflict then given grammar is LR(0) grammar else no.

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

$$G: \quad S \rightarrow (L) \mid a$$

$$L \rightarrow L \, S \mid S$$





	Action					goto	
	a	,	()	\$	S ₃	L
I ₀	S ₃		S ₂			1	
I ₁					acc.		
I ₂	S ₃		S ₂			5	4
I ₃	rl ₂						
I ₄	S ₇			S ₆			
I ₅	rl ₄						
I ₆	rl ₁						
I ₇	S ₃		S ₂			8	
I ₈	rl ₃						

NO conflict

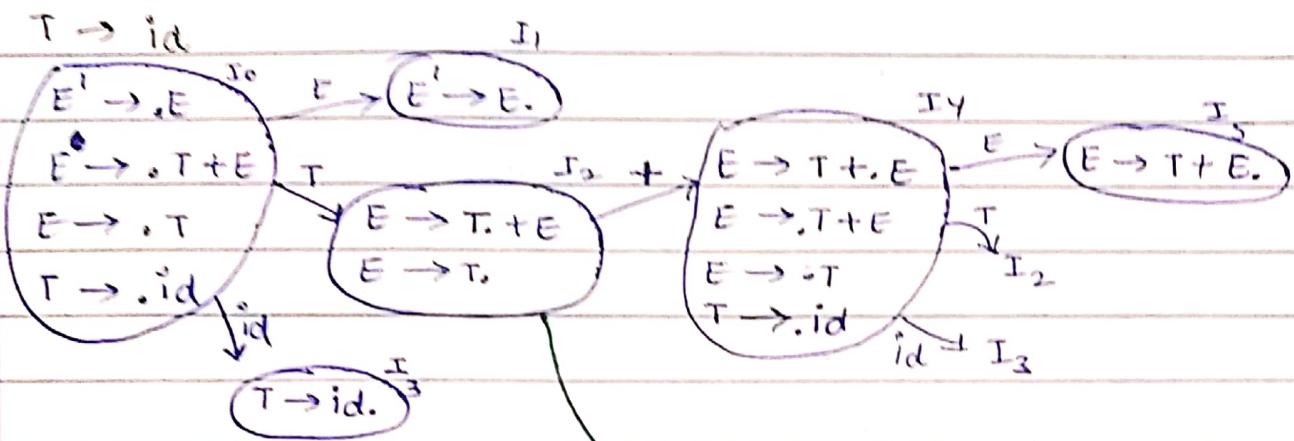
⇒ LR(0)
SLR(1)
CLR(1)
LALR(1)

} all exist

This grammar does not exist in LL(1) as left recursion is present.

Q Check following grammar is SLR(1) or not?

$$E \rightarrow T + E \mid T$$



Not in SLR(0)

May be in SLR(1)

This state has problem because 2
pseudⁿ present & one of them is
getting reduced. So IT MAYBE A
PROBLEM. But it does not guarantee
that problem is a conflict.

SLR

SR occurring on + so draw table
for state 2

+	id	\$
g ₂ / ₄	g ₂	g ₂

Problem would not have occurred even because of
shift & reduce be when $E \rightarrow T + E$. Here after dot
there is a variable whose entry would go in
goto part. and there is no conflict in goto part

NOTE:- A state having a problem is called inadequate state
here there is 1 inadequate state.

NOW CHECK FOR SLR(1) :-

Here the reduced entry goes to the columns which

+ id \$

S ₃		g ₂
----------------	--	----------------

Given grammar is SLR(1)

LALR(1)

CLR(1)

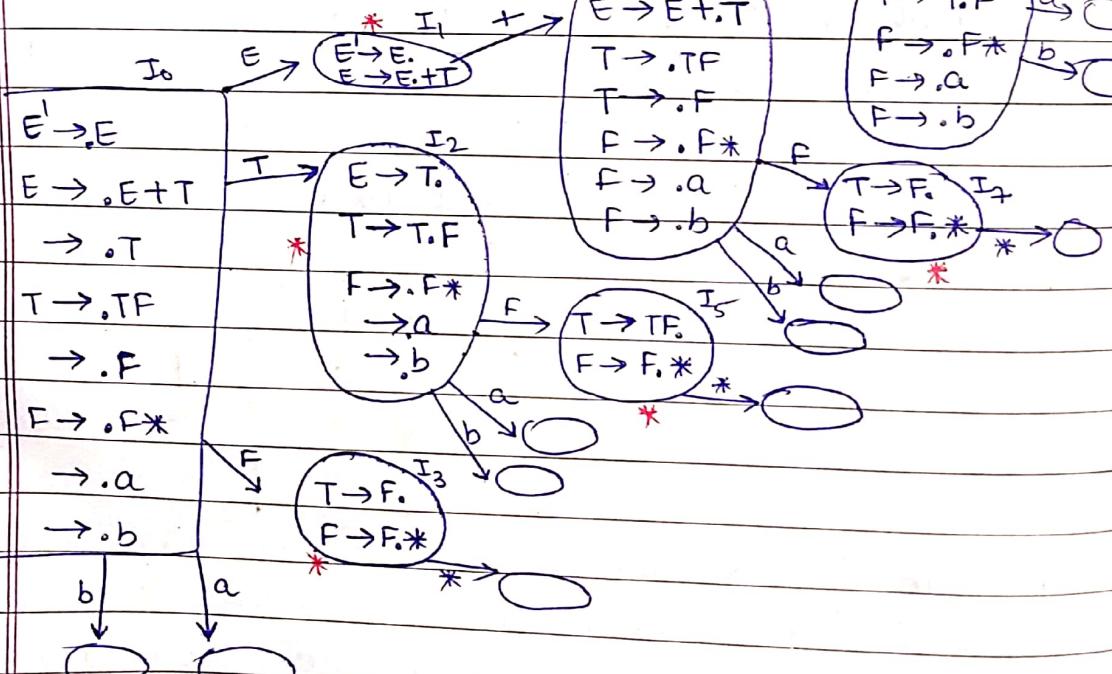
Q

Check the given grammar is SLR(1) or not?

$$E \rightarrow E^1 + T^2$$

$$T \rightarrow T^3 F^4$$

$$F \rightarrow F^5 *^6 | a/b^7$$



+ * a b * \$

IF LR(0)

X I ₁	I ₂	g ₂			S	S	g ₂
E → T ₀	I ₂	g ₂			S	S	g ₂
T → F ₁	I ₃	g ₄	S	S	g ₄	g ₄	g ₄
T → T ₂ F ₃	I ₅	g ₃	S	S	g ₃	g ₃	g ₃
E → E + T ₆	I ₆	g ₁			S	S	g ₁

I₁: not a problem
because E' → E₀ is
not a part of our
grammar. So no
reduction possibl

I₂: RR not possible

IF (LR(0))

$$\text{follow}(E) = \{\$, +\}$$

$$\text{follow}(T) = \{\$, +, a, b\}$$

$$\text{follow}(S) \neq \emptyset$$

For I_2 on this state, by reading terminal you are going outside
 \therefore 2 shift entries present.

But reduced entry everywhere

\therefore grammar is in SLR(1).

~~$\text{follow}(T) = \{\$, +, a, b\}^*$~~

\therefore 2-SR conflicts.

LL(1) X

FOR I_3 :- only 1 is getting reduced.

LR(0) X

No R-R.

SLR(1) ✓

But SR possible. Reduce



occur everywhere. \Rightarrow (SR - 1)

LALR(1), CLR(1) ✓

FOR I_5 :- 1-SR.

FOR I_6 :- 2-SR.

Total = 6 - SR conflicts.

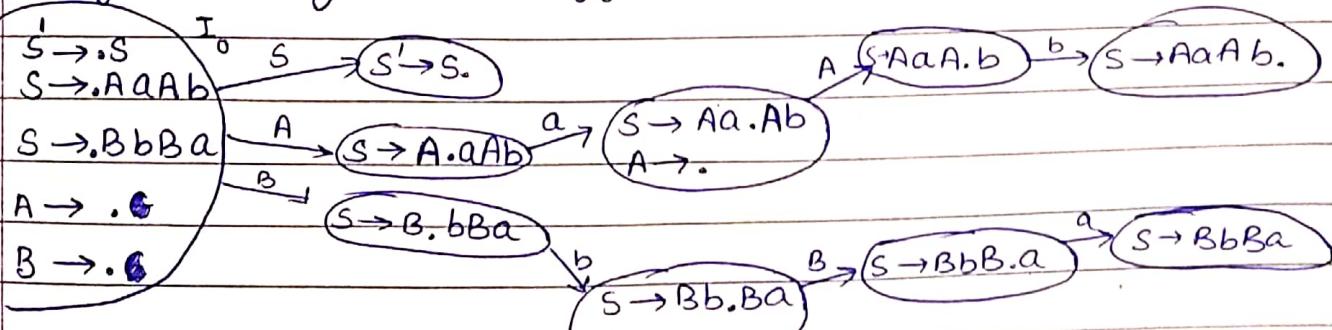
Total 4 inadequate states $(2, 3, 5, 6)$

Q. $S \rightarrow AaAb / BbBa$ check if SLR(1) or not?

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

This is in LL(1) \Rightarrow top down pass \Rightarrow since bottom up parser is more powerful hence CLR(0) will also satisfy because LL(1) is most powerful in topdown. and CLR(1) is most powerful in bottom up. LR(0), SLR(1), LALR(1) may or may not satisfy it.



There are 2 reductions. May be RR conflict?

3 RR conflict \Leftarrow For LR(0) :- a_4, a_5 present in entire row
 $a_f I_0 \Rightarrow$ Not LR(0).

2 RR conflict \Leftarrow For SLR(1) :- a_3, a_4 present in a, b of I_0 .
 \Rightarrow Not SLR(1).

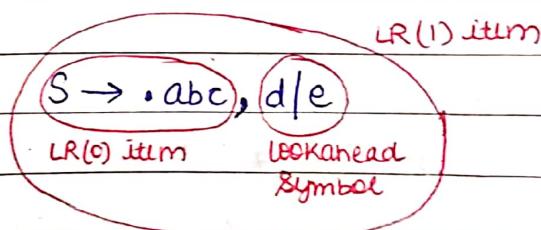
1 Inadequate state in LR(0) & SLR(1).

HOW TO CHECK FOR CLR(1) & LALR(1)?

Using LR(0) items \rightarrow LR(0) parser
 \downarrow SLR(1) parser

Using LR(1) items \rightarrow CLR(1) parser
 \downarrow LALR(1) parser.

LR(1) item = LR(0) item + lookahead symbol.



CLOSURE() & GOTO() FOR LR(1) ITEMS :-

$$G: S \rightarrow AA$$

$$A \rightarrow aA | b$$

$$G': S' \rightarrow S$$

$$S \rightarrow AA$$

Closure (I) :augmented
LR(0)
itemex1closure ($S' \rightarrow .S$, \$)

copied as it is from above

(1) $S' \rightarrow .S$, \$ $\rightarrow .S \rightsquigarrow$ whatever after S, take its First().
Here nothing after S, so \$(2) $S \rightarrow .AA$, \$(3) $A \rightarrow .aA$, a/b
.b, a/b} For lookahead symbol, go to prev. prod
and take First(A\$) = a/bex2 closure ($S \rightarrow .AA$, c/d)(1) $S \rightarrow .AA$, c/d(2) $A \rightarrow .aA$, a/b $\rightarrow .b$, a/b } For lookahead symbol, goto (1) &
find first(Acd) = a/b.Definition :- closure (I) =

(1) add I

(2) $A \rightarrow B.C(DE, a/b)$ if I and C $\rightarrow EFG$ is in grammar
then add $[C \rightarrow .EFG, \text{First}(DE, a/b)]$ to closure (I).

(3) repeat step (2)

GOTO ()Goto ($S \rightarrow .AA$, c/d), a)

LR(1) item

i/P (after ., a is present)

 $S \rightarrow a(A, cd)$ $A \rightarrow .AA$, c/d
.b, c/d

} taking First(c/d)

No. of states in LR(0), SLR(1), CLR(1)
 $\Rightarrow (n_1 = n_2) \leq n_3$

Date: / /
 Page No. 68

Def :-

Goto (I_i, x) =

- (1) add I by moving x after x
- (2) closure (1^{st} step)

Q

construct CLR(1) & IALR(1) parsing table for following grammar:-

$$G: S \rightarrow AA \quad (1)$$

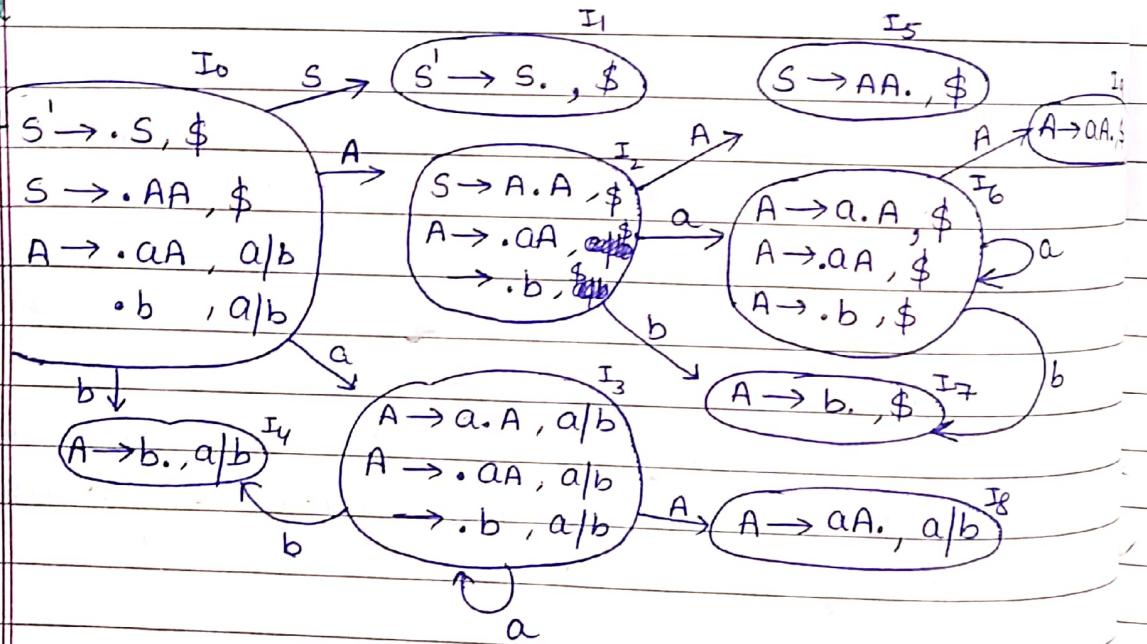
$$A \rightarrow aA/b \quad (2)$$

$$G' : - \quad S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA/b$$

For augmented production,
 LAS = \$



In LR(0) DFA, from I_2 on $i/p = a$ we go to I_3 .
 But here we go to diff. state.

A state having 2 meanings then separate final if any problem present with any path.

	ACTIONS			GOTO	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			acc.		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_{l_3}	r_{l_3}			
5	.		r_{l_1}		
6	s_6	s_7			9
7			r_{l_3}		
8	M_2	r_{l_2}			
9			r_{l_2}		

CLR(1) parsing table

To fill reduce entries, write them only in LAs.

For I_4 :- r_{l_3} goes in a,b.

For I_5 :- r_{l_1} goes in \$

For I_7 :- r_{l_3} goes in \$

For I_8 :- r_{l_2} goes in a,b

For I_9 :- r_{l_2} goes in \$

Only drawback of CLR(1) is that its cost is high.
To reduce the cost, we minimize the states by using LALR(1).

- In the above CLR(1) table, no conflict. Hence given grammar is CLR(1).
- CLR(1) parser is more powerful comparing to other parsers because it takes care everytime. The only disadvantage is it is not also

To decrease cost of CLR(1), we apply minimisation algo. [if 2 states differ only by lookahead then make it as single state.]

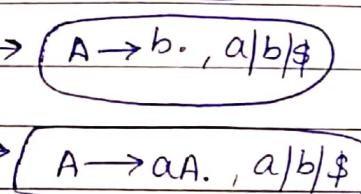
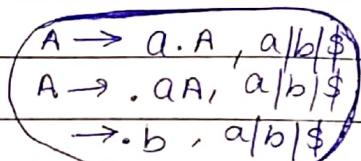
- Minimised $\text{curl}(1)$ is also known as $\text{LALR}(1)$.

$$\begin{array}{c} I_3, I_6 \equiv I_{36} \\ I_4, I_7 \equiv I_{47} \\ I_8, I_9 \equiv I_{89} \end{array}$$

a
a
a|

a

a
a
a|



Action				goal	
	a	b	\$	s	A
0	S_{36}	S_{47}		1	2
1			acc.		
2	S_{36}	S_{47}			5
3	S_3	S_4			89
4	rl_3	rl_3	rl_3		
5			rl_1		
6	S_6	S_7			9
7			rl_3		
8	rl_2	rl_2	rl_2		
9			rl_2^{\uparrow}		

LALR(1) parsing table

No. of states in LALR(1) = n_4

$$CLR(1) = n_3$$

$$SLR(1) = n_2$$

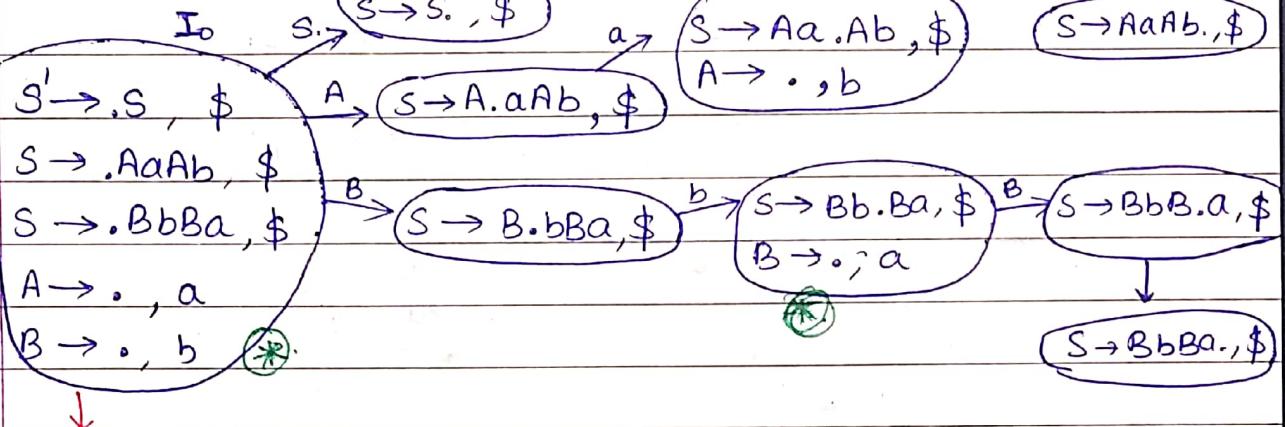
$$n_1 = n_2 = n_4 \leq n_3$$

Q Check if the following grammar is LALR(1) or not?

$$S \rightarrow AaAb \mid BbBa$$

$$A \rightarrow E$$

$$B \rightarrow E$$



In I_0 , RR conflict may be possible

But lookheads are diff. So no problem.

∴ In CLR(1) ✓

	a	b	\$
LR(0) \Rightarrow	x_3/x_4	x_3/x_4	x_3/x_4
SLR(1) \Rightarrow	x_3/x_4	x_3/x_4	
CLR(1) \Rightarrow	x_3	x_4	

No minimised state so CLR \equiv LALR.

∴ LALR(1) ✓

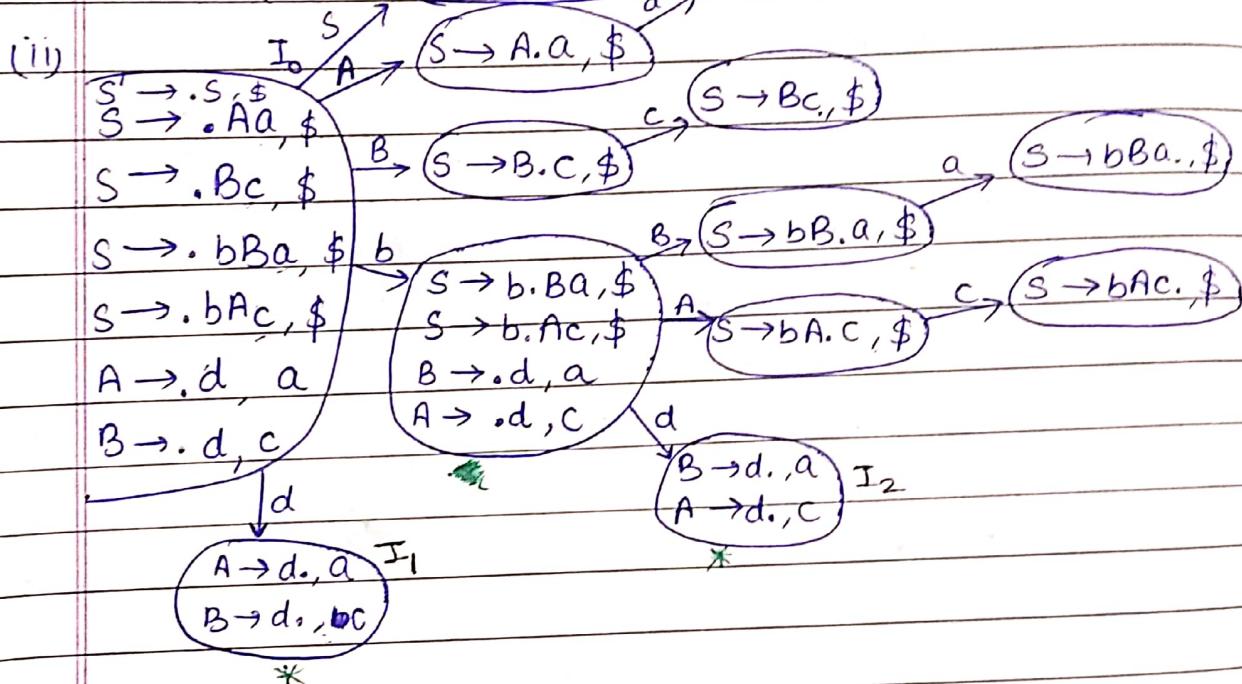
Q Check if following grammar is LALR(1) or not?

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

$$A \rightarrow d$$

$$B \rightarrow d$$

(i) Not LL(1).



	a	b	c	d	\$	
LR(0) $\Rightarrow I_1$	s_5/s_6	s_5/s_6	s_5/s_6	s_5/s_6	s_5/s_6	5RR } NOT LR(1)
	"	"	"	"	"	5RR } (10 RR)
I_2						
SLR(1) $\Rightarrow I_1$	s_5/s_6		s_5/s_6			2RR } Not SLR(1)
I_2	s_5/s_6		s_5/s_6			2RR } (4RR)
CLRL(1)	I_1	s_5	s_6			{ NO RR }
	I_2	s_6				

I_1 ~~I_2~~ states are minimised. Hence $\underline{CLR \neq LALR}$
 \therefore exists in LALR(1)

	a	b	c	d	\$
LALR(1) $\Rightarrow I_2$					
	s_5/s_6		s_5/s_6		

2 RR conflict
Not in LALR(1)

- ~~Ans~~
- * If CLR(1) don't have RR conflict then LALR(1) may or may not contain RR conflict.
 - * If CLR(1) don't have SR conflict then LALR(1) also does not have SR conflict.
 - If G is CLR(1) but not LALR(1) because RR conflict occurred.
 - If G is not LALR(1) because of SR conflict then definitely G is not CLR(1) because of SR conflict.

Q Given grammar which does not have RR conflict in any parser. This grammar is CLR(1). What about LALR(1)?

Ans It is also LALR(1) because RR not possible & since it is in CLR(1) \Rightarrow NO SR conflict

Q. Consider the following grammar:-

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

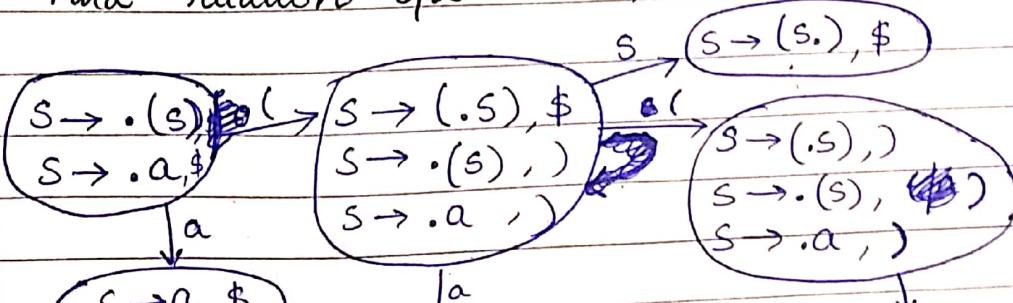
LR(0) \Rightarrow no. of states = n_1

SLR(1) \Rightarrow no. of states = n_2

LALR(1) \Rightarrow no. of states = n_3

CFLR(1) \Rightarrow no. of states = n_4 .

Find relation b/w n_1, n_2, n_3, n_4 ?



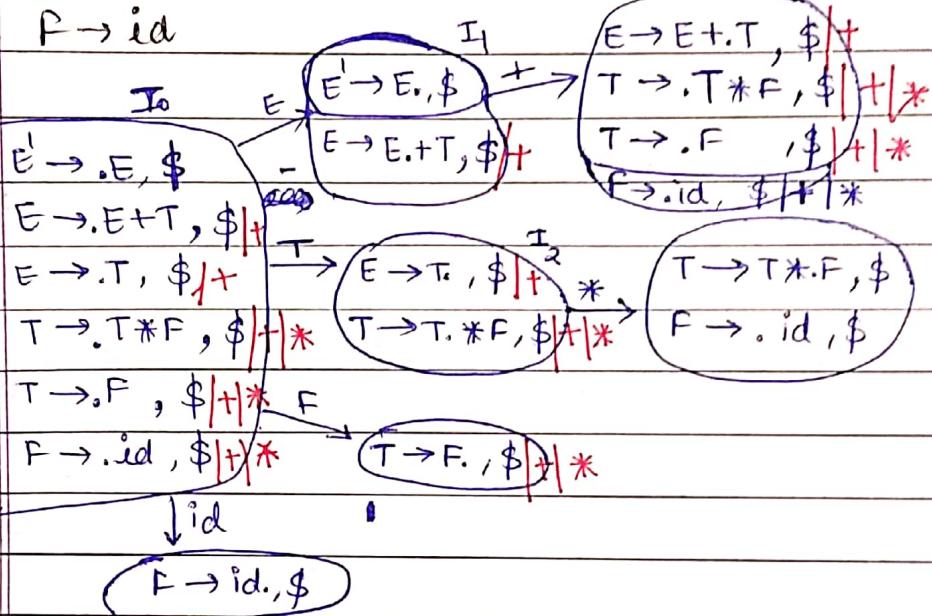
$$n_1 = n_2 = n_3 < n_4$$

Q

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow id$$

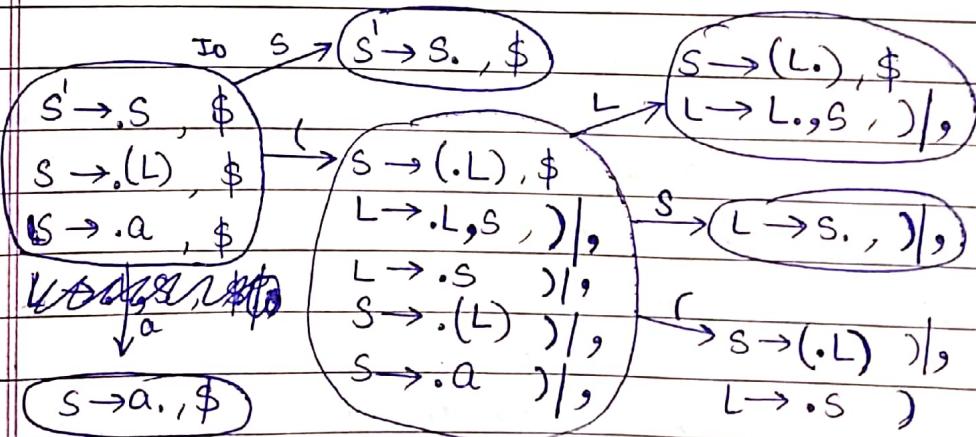


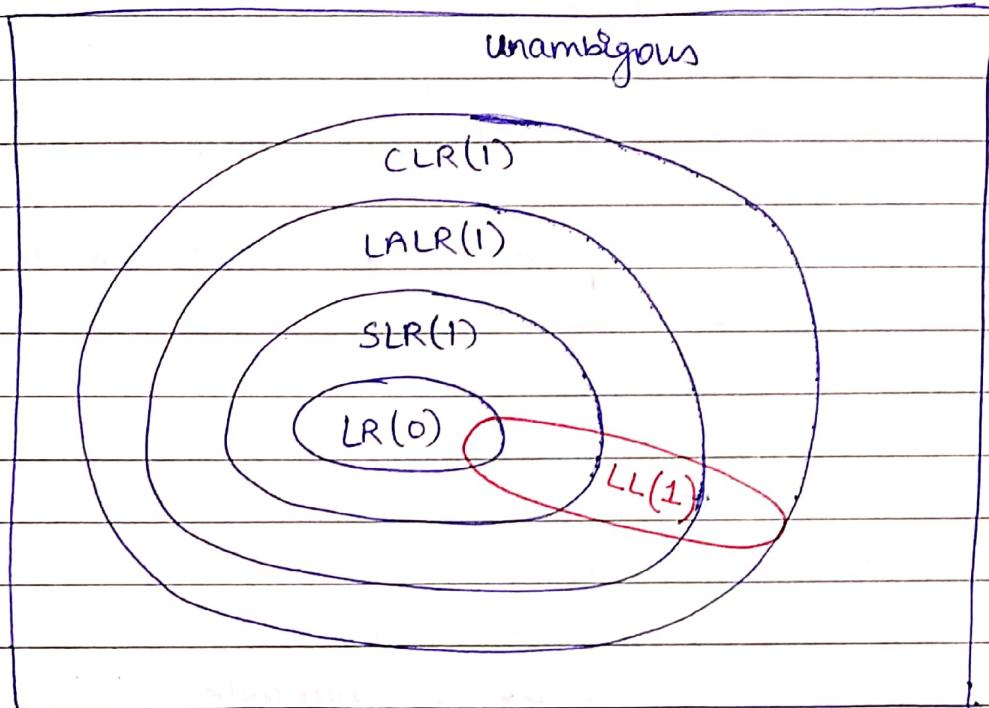
Q

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L, S \mid S$$

Find CLR(1) DFA only
4 - states.





Every $LL(1)$ is $LR(1)$ but every $LR(1)$ need not be $LL(1)$.

$$LL(1) \subseteq LR(1)$$

$$LL(2) \subseteq LR(2)$$

⋮

$$LL(k) \subseteq LR(k)$$

OPERATOR PRECEDENCE PARSER

- ① It is applicable only for operator grammar.
- ② A grammar G is said to be operator grammar if & only if:
 - (i) G does not have null productions
 - (ii) G does not have 2 adjacent variables on RHS of production

eg

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

\Rightarrow This is operator grammar.
ambiguous grammar.

This is the only parser which works on ambiguous grammar as well.

eg

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow id$$

\Rightarrow operator grammar
 \Rightarrow unambiguous grammar.

eg

$$S \rightarrow AB$$

$$A \rightarrow a$$

$$B \rightarrow b$$

unambiguous grammar

but not operator grammar.

eg

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

ambiguous grammar.

not operator grammar.

28/11/2019

SYNTAX DIRECTED TRANSLATION :-

$SDT = \text{Grammar} + \text{Semantic action}$

e.g. $S \rightarrow ABC \quad \{ \text{printf} ("Hi") \}$

e.g. $E \rightarrow E_1 + E_2 \quad \{ \text{if } (E_1.\text{type} == E_2.\text{type}) \}$

$E.\text{type} = E_1.\text{type}$

Parent E getting value by child E_1 .
else $\text{printf} ("Type mismatch")$

⇒ Synthesised attribute

SDT can be written for any phase.

Attributes (Every variable has its attributes)

Synthesised attribute

$S \rightarrow ABC \quad \{ S.\text{val} = f(A.\text{val}, B.\text{val}) \}$

$S.\text{val} = A.\text{val} + B.\text{val} - C.\text{val}$

A B C
A.val B.val C.val

at any point, value of S is calculated using only its children's attributes.

Inherited attribute

$S \rightarrow ABC \quad \{ B.\text{val} = f(S.\text{val} | A.\text{val}) \}$

$B.\text{val} = S.\text{val} + A.\text{val} + C.\text{val}$

A B C
A.val B.val C.val

at any point, val of B depends on attributes of parent or/and attributes of sibling.

TYPES OF SDT :-

① S-attributed definition :-

→ If attributes are required then only synthesised attributes are used.

→ It uses bottom up approach because children are reqd.

→ RHS of production at rightmost position

called as post order evaluation

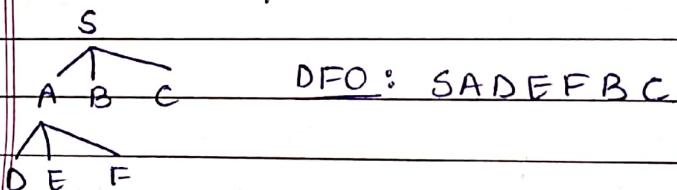
→ Semantic action ~~needs~~ has to be written at the end of production $S \rightarrow ABC \{ \}$.

→ It is more restricted and is subset of L-attributed. Every S is L also.

L-attributed

→ It uses both synthesised and inherited attribute but with a condition that only parent & left child's attributes can be used but not right child.

→ It follows depth first order Left to right evaluation



→ Semantic action can be placed anywhere on the RHS of production.

→ It is less restricted and is superset of S-attributed. Every L-attribute is not S-attribute.

APPLICATIONS OF SDT :-

Eg: 1 Construct SDT to execute given arithmetic expression

I/P: $2 + 3 * 4$

O/P: 14

For I/P : write grammar

For O/P : write semantic action

Date: / /

Page No. 79

Ans

① Write grammar for I/P.

$E \rightarrow E + T \quad \{ E.val = E.val + T.val, \text{print}(E.val) \}$

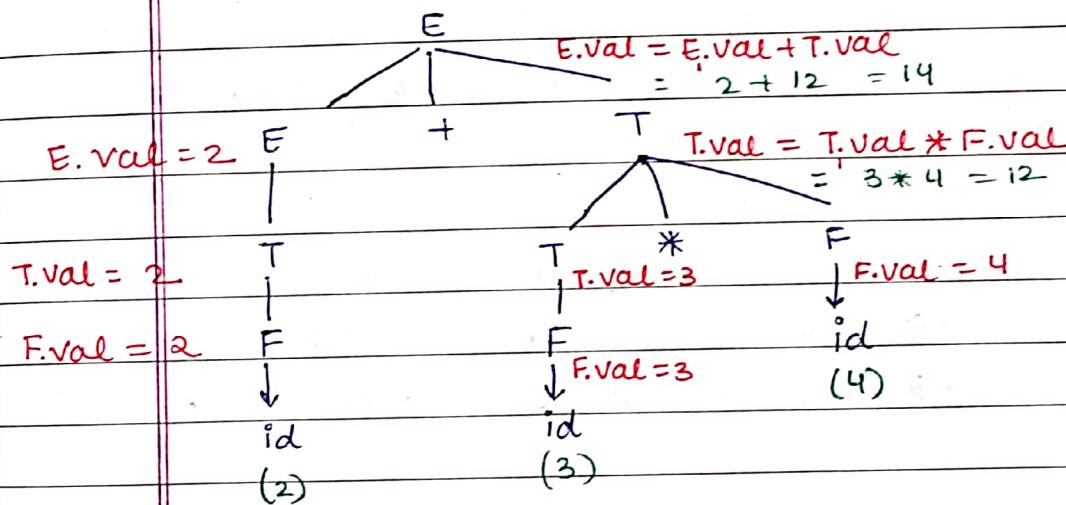
$E \rightarrow T \quad \{ E.val = T.val \}$

$T \rightarrow T * F \quad \{ T.val = T.val * F.val \}$

$T \rightarrow F \quad \{ F.val = F.val \}$

$F \rightarrow id \quad \{ F.val = id \}$

② Write its tree



③ Write semantic action for each production

screen

O/P: | 14

This tree is called Annotated Parse Tree or decorated parse tree. Here at every level, each variable has its attributes mentioned.

Here val is a synthesised attribute.

$\Rightarrow S\text{-attributed distribution}$

$\Rightarrow L\text{-attributed distribution}$

To decide priority of operators :- consider the grammar.

To decide what action has to be performed :- consider the semantic actions

eg

$$\begin{array}{ll} E \rightarrow E_1 * T & (E.\text{val} = E_1.\text{val} + T.\text{val}) \\ E \rightarrow T & (E.\text{val} = T.\text{val}) \\ T \rightarrow T_1 + F & (T.\text{val} = T_1.\text{val} * F.\text{val}) \\ T_1 \rightarrow F & (T_1.\text{val} = F.\text{val}) \\ F \rightarrow \text{id} & (F.\text{val} = \text{id}) \end{array}$$

Here * has less priority than +. ~~also~~

+ ~~operator~~ operator executed first. But carefully look at actions. * actually means perform + and + actually means perform *.

For gate, ques \rightarrow Given SDT & i/P, find O/P
 \rightarrow Given SDT & O/P, find i/P
 \rightarrow Given SDT, i/P, O/P, find missing statements

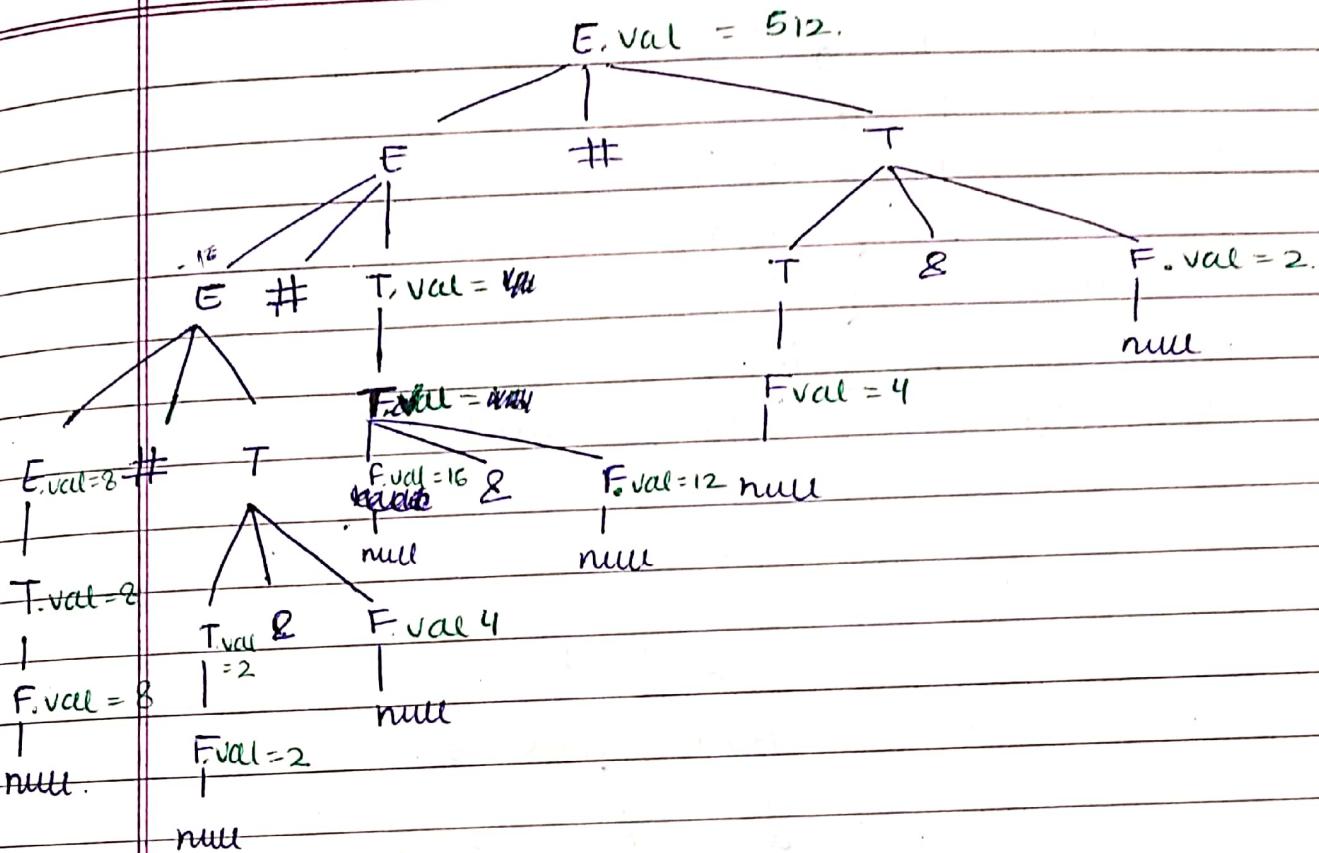
Q Consider the input

$$\begin{array}{ll} E \rightarrow E \# T & \{ E.\text{val} = E_1.\text{val} * T.\text{val} \} \\ | T & \{ E.\text{val} = T.\text{val} \} \\ T \rightarrow T \& F & \{ \} \\ | F & \{ T.\text{val} = F.\text{val} \} \\ F \rightarrow \text{null} & \{ F.\text{val} = \text{num} \} \end{array}$$

i/P :- 8 # 2 & 4 # 16 & 12 # 4 & 2

O/P : 512

- $T.\text{val} = T.\text{val} - F.\text{val}$
- $T.\text{val} = T.\text{val} + F.\text{val}$.
- $T.\text{val} = T.\text{val} \times F.\text{val}$



$\text{if } 8 \equiv - \quad \text{ans. } E.\text{val} = -128 \quad \text{No.}$
 $\text{if } 8 \equiv + \quad \text{ans. } E.\text{val} = (48 \times 28) * (6) > 512 \quad \text{No.}$
 $\text{if } 8 \equiv * \quad \text{ans. } > 512.$
(d) [But in gate O/P = -128 \therefore ans(a)]

(ii) i/p 10 # 8 & 6 # 9 & 4 # 5 & 2

Here 8 = -

= *

$$10 * 8 - 6 * 9 - 4 * 5 - 2$$

~~0000000000~~

$$10 * 2 * 5 * 3 = \underline{\underline{300}}$$

Q consider the following SOT :-

since the production has action at the end
 means when $S \rightarrow AS$ is completed only
 then perform action.

Date: / /
 Page No. 82

$$S \rightarrow AS \quad \{ \text{punitf } (1) \}$$

$$S \rightarrow AB \quad \{ \text{punitf } (3) \}$$

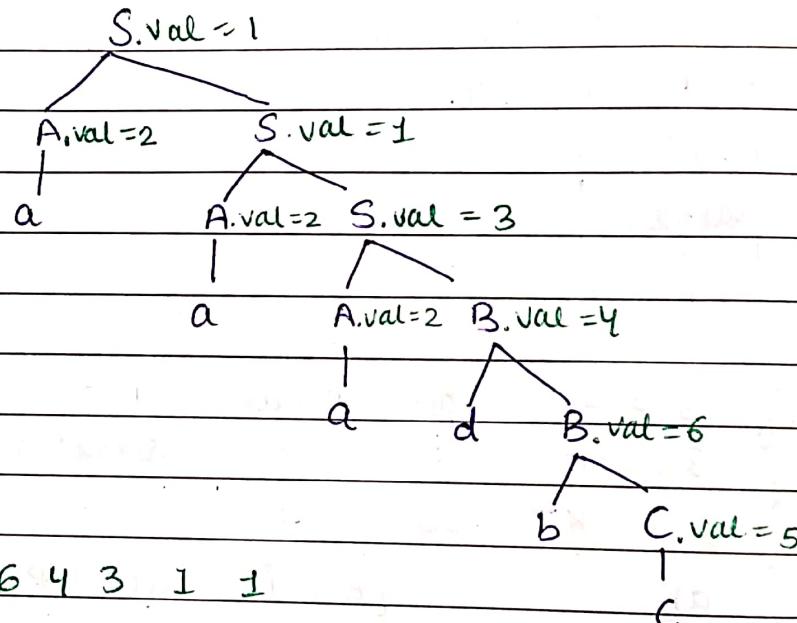
$$A \rightarrow a \quad \{ \text{punitf } (2) \}$$

$$B \rightarrow bC \quad \{ \text{punitf } (6) \}$$

$$B \rightarrow dB \quad \{ \text{punitf } (4) \}$$

$$C \rightarrow c \quad \{ \text{punitf } (5) \}$$

Find o/p for i/p: aaadbc.



Every parser scans from left to right.

Q Construct SDT to convert the given infix expression to postfix.

i/p :- $x = a + b * c$

O/P: $xabc+* =$

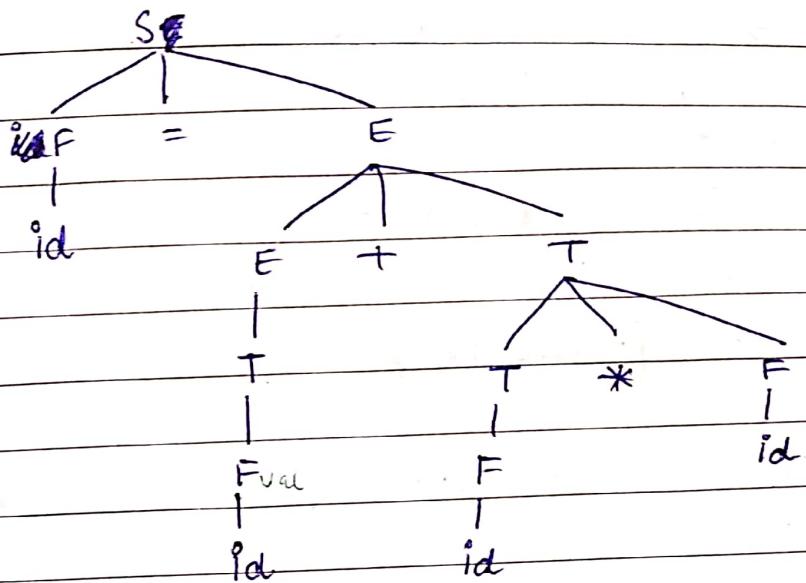
$$S \rightarrow F = E$$

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$



~~defn.~~
 $S \rightarrow F = E \quad \{ \text{print} (=) \}$

$E \rightarrow E + T \quad \{ \text{print} (+) \}$

$E \rightarrow T \quad \{ \text{print} ("") \} \mid \text{nothing}$

$T \rightarrow T * F \quad \{ \text{print} (*) \}$

$T \rightarrow F \quad \{ \text{print} ("") \} \mid \text{nothing}$

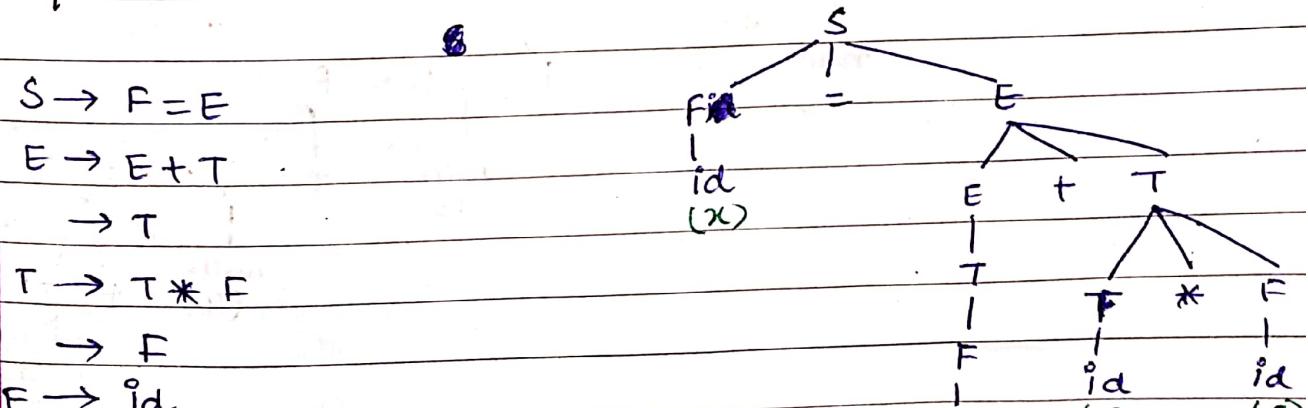
$F \rightarrow id \quad \{ \text{print}(id) \}$

No attribute used \Rightarrow S-attributed \Rightarrow L-attributed

Q) Construct SDT to convert infix to prefix :-

I/P:- $x = a + b * c$.

O/P:- $= x - t * b c$



$S \rightarrow \{ \text{primary} (=) \} F = E$

$E \rightarrow \{ \text{primary} (+) \} F + T$

$E \rightarrow T$

$T \rightarrow \{ \text{primary} (*) \} T * F$

$T \rightarrow F$

$F \rightarrow \text{id } \{ \text{primary} (\text{id}) \}$

This is L-attributed definition but not S-attributed

Had it been infix to infix then semantic action
in between. $S \rightarrow F \{ \} = E$ or $S \rightarrow F = \{ \} E$

Q

Consider the following SDT :-

$S \rightarrow T \{ \text{pf}(+) \} R$

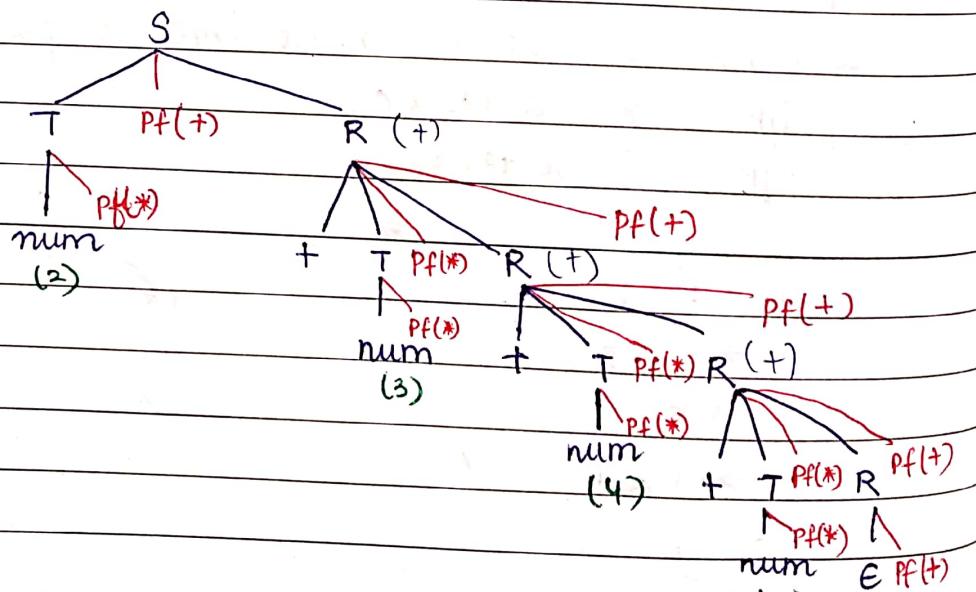
$R \rightarrow + T \{ \text{pf}(*) \} R \{ \text{pf}(+) \}$

$R \rightarrow E \{ \text{pf}(+) \}$

$T \rightarrow \text{num} \{ \text{pf}(*) \}$

i/p : $2 + 3 + 4 + 5$

o/p :



* * * * *

* + * * * * * + + +

Q Construct SDT to convert binary to decimal

$$\text{i/p : } 101.111 \quad | \quad 101$$

$$\text{o/p : } 85.875 \quad | \quad 5$$

$$\begin{array}{r} 0.5 \\ 0.25 \\ 0.125 \\ \hline 0.875 \end{array}$$

$$S \rightarrow L.L \quad \left\{ \begin{array}{l} S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \\ \end{array} \right\}$$

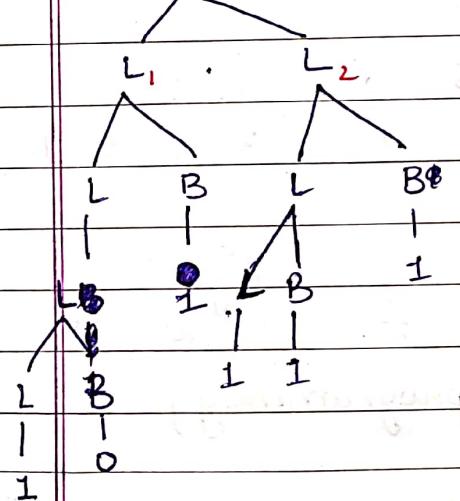
$$L \rightarrow L.B \quad \left\{ \begin{array}{l} S.dv = L.dv \end{array} \right.$$

$$L \rightarrow LB \quad \left\{ \begin{array}{l} L.dv = 2L_1.dv + B.dv \\ L.nb = L_1.nb + B.nb \end{array} \right\}$$

$$B \rightarrow 1 \quad \left\{ \begin{array}{l} L.dv = B.dv \\ L.nb = B.nb \end{array} \right.$$

$$B \rightarrow 0 \quad \left\{ \begin{array}{l} B.dv = 0 \\ B.nb = 1 \end{array} \right.$$

$$S \rightarrow L_1.dv + \frac{L_2.dv}{2^{L_2.nb}}$$



$$\begin{array}{r}
 1110(14) \\
 2 \times 1 + 0 \\
 \hline
 111(7) \\
 2 \times 1 + 1 \\
 \hline
 11(3) \\
 2 \times 1 + 1 \\
 \hline
 1(1) \\
 \end{array}
 \rightarrow \text{decimal value}$$

Multiply LHS by 2 & add RHS

To get LHS of decimal

$$2 \times \text{LHS} + \text{RHS}$$

$$\begin{array}{r}
 010 \\
 2 \times 1 = 2 + 0 = 2 \\
 \hline
 010
 \end{array}$$

To get RHS of decimal

check no. of bits of RHS
then

$$\begin{array}{r}
 01 \\
 2 \times 0 = 0 + 1 = 1 \\
 \hline
 01
 \end{array}$$

* * * * *

* + * * * * + + +

Q Construct SDT to convert binary to decimal

$$\text{I/P :- } 101.111 \quad | \quad 101$$

$$\text{O/P : } 85.875 \quad | \quad 5$$

$$\begin{array}{r} 0.5 \\ 0.25 \\ 0.125 \\ \hline 1.50 \end{array}$$

$$S \rightarrow L, L \quad \left\{ \begin{array}{l} S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \\ \end{array} \right\}$$

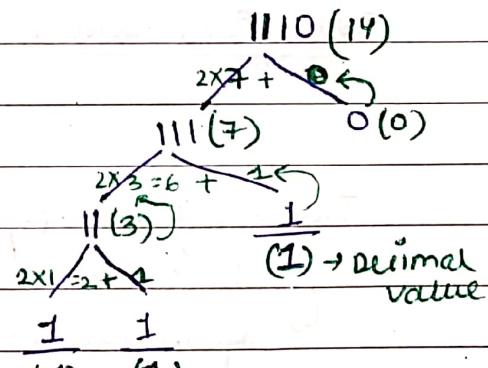
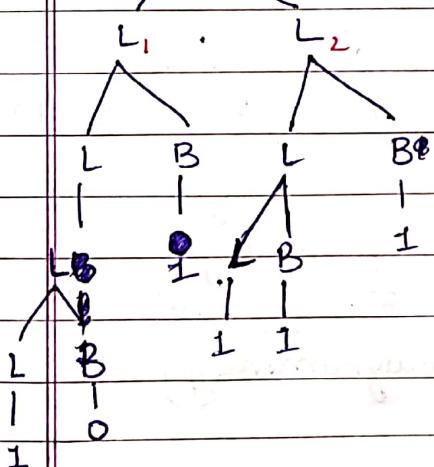
$$L \rightarrow L, B \quad \left\{ \begin{array}{l} S.dv = L.dv \end{array} \right\}$$

$$L \rightarrow L, B \quad \left\{ \begin{array}{l} L.dv = 2L_1.dv + B.dv \\ L.nb = L_1.nb + B.nb \end{array} \right\}$$

$$B \rightarrow 1 \quad \left\{ \begin{array}{l} L.dv = B.dv \\ L.nb = B.nb \end{array} \right\}$$

$$B \rightarrow 0 \quad \left\{ \begin{array}{l} B.dv = 0 \\ B.nb = 1 \end{array} \right\}$$

$$S \rightarrow L_1.dv + \frac{L_2.dv}{2^{L_2.nb}}$$



Multiply LHS by 2 & add RHS

To get LHS of decimal

$$2 \times \text{LHS} + \text{RHS}$$

$$\begin{array}{r} 010 \\ 2 \times 1 = 2 + 0 = 2 \\ \hline 010 \end{array}$$

To get RHS of decimal

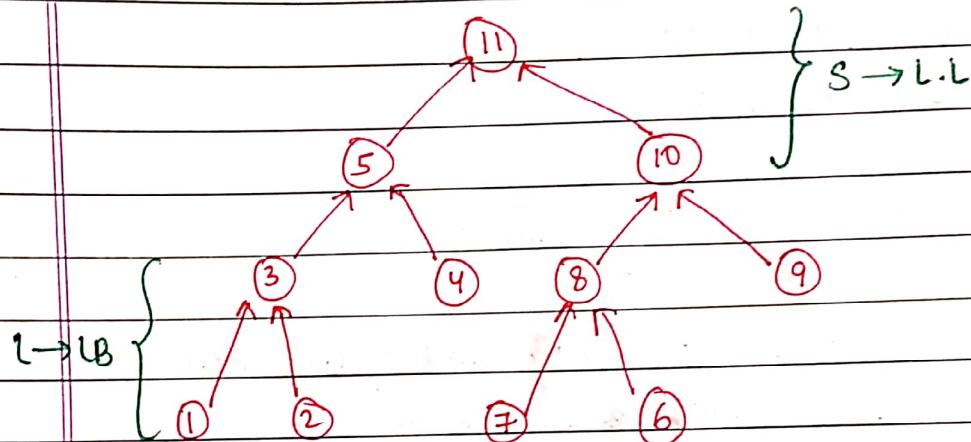
check no. of bits of RHS

then 1 digit

$$\begin{array}{r} 01 \\ 2 \times 0 = 0 + 1 = 1 \\ \hline 01 \end{array}$$

$a = b$ means a depends on b . $a \leftarrow b$

$$L.dv = 2Li.dv + Bi.dv$$



This graph is called dependency graph (Directed Acyclic graph)

Semantic action gives dependency graph.

Topological sort :- 9 6 7 8 10 4 1 2 3 5 11
6 9 7 8 4 1 2 3 5 10 11

Try to follow Depth First order left to Right.

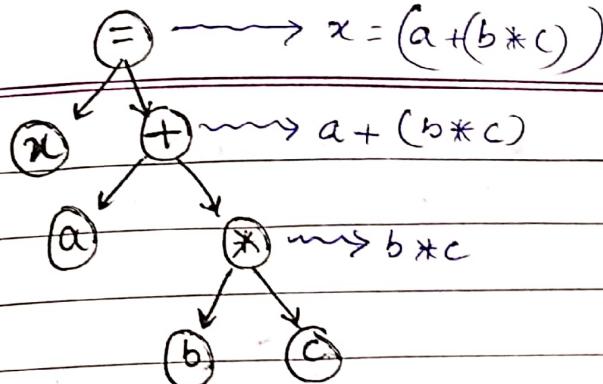
1 2 3 4 5 6 7 8 9 10 11

→ Topological sort (close to programming)

NOTE: If ques asked which semantic action done first?

ans Do topological sort close to programming.
on dependency graph.

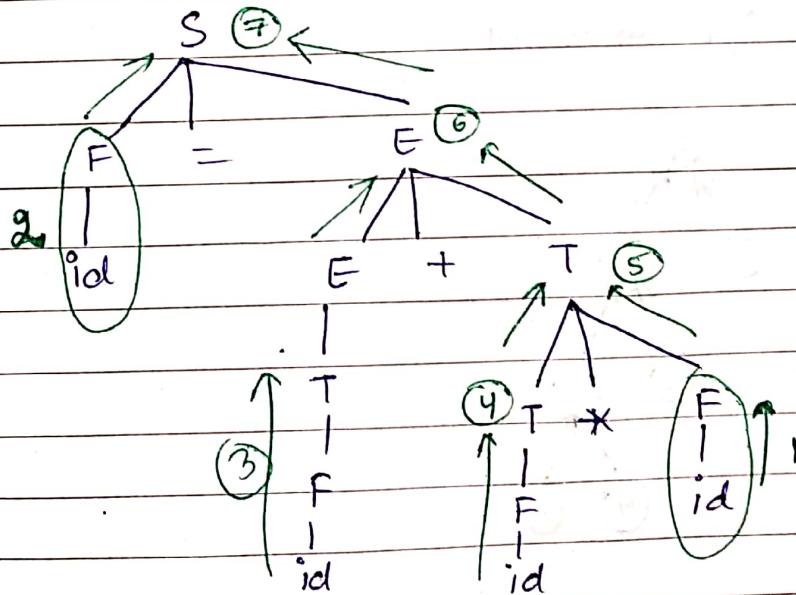
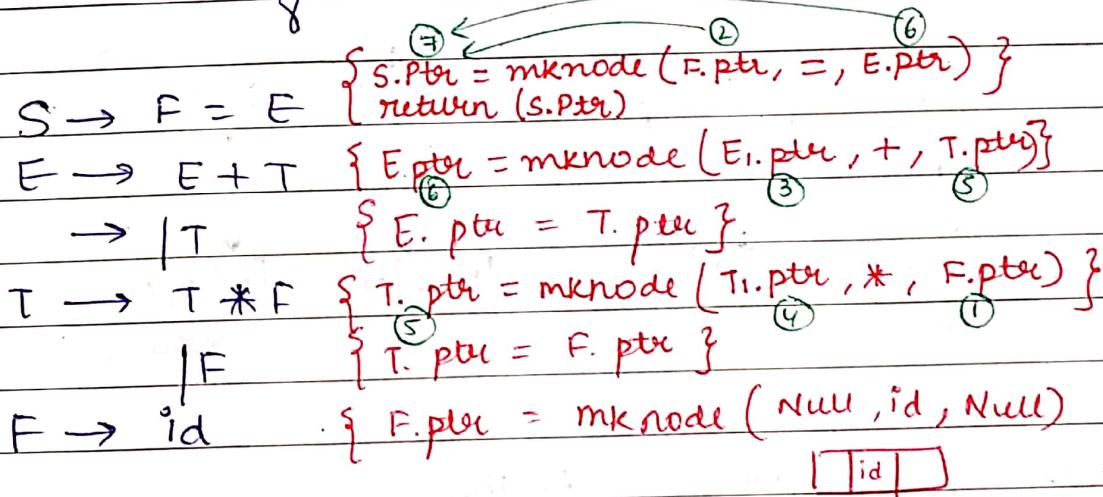
Q Construct SDT to create syntax tree for given arithmetic expression :-



This is a syntax tree

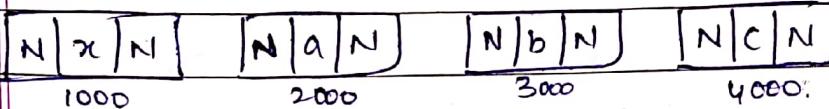
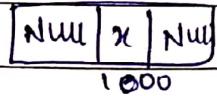
Different b/w syntax tree & parse tree is that in syntax tree, all intermediate nodes are operators & leaf nodes are operands.

Parse tree :- All intermediate nodes are variables & all leaf nodes are terminals.

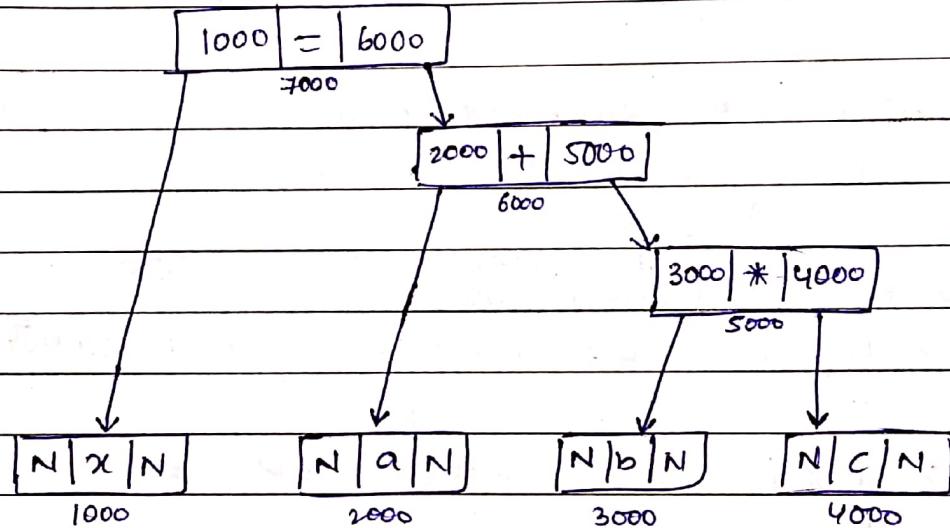


To convert parse tree to syntax tree:

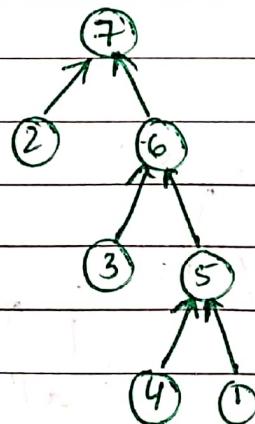
- ① Create a node for operand only \Rightarrow leaf node



②



ptr is a synthesised attribute
 \Rightarrow S, L attributed



Q

Construct SDF to generate intermediate code for given arithmetic expression

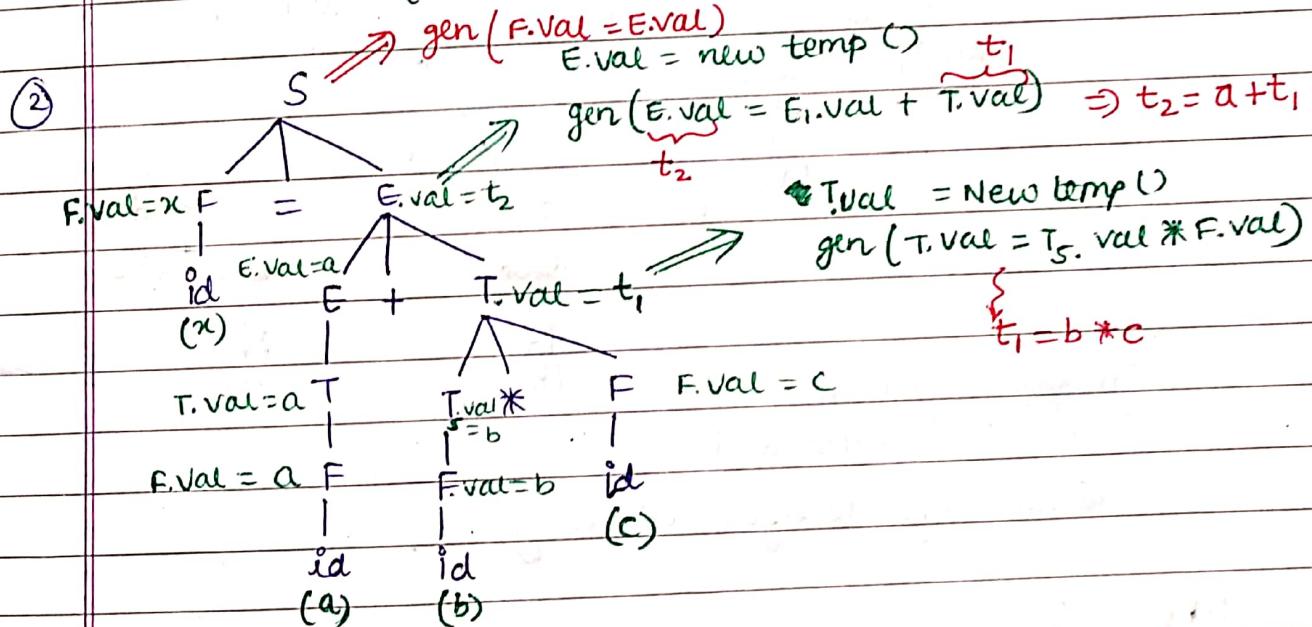
$$\text{I/P} : -x = a + b * c$$

$$\text{O/P} : \quad t_1 = b * c$$

$$t_2 = a + t_1$$

$$x = -t_2$$

$$\begin{aligned}
 ① \quad S &\rightarrow F = E \quad \{ \text{gen}(F.\text{val} = E.\text{val}) \} \\
 E &\rightarrow E + T \quad \{ E.\text{val} = \text{new temp}(); \text{gen}(E.\text{val} = E_1.\text{val} + T.\text{val}) \} \\
 | T &\quad \{ E.\text{val} = T.\text{val} \} \\
 T &\rightarrow T * F \quad \{ T.\text{val} = \text{new temp}(); \text{gen}(T.\text{val} = T_1.\text{val} * F.\text{val}) \} \\
 | F &\quad \{ T.\text{val} = F.\text{val} \} \\
 F &\rightarrow \text{id} \quad \{ F.\text{val} = \text{id} \}
 \end{aligned}$$



③

$\text{gen}()$ function generates O/P code i.e. it does not do any calculation. It prints as it is.
 $\text{new temp}()$ is a function which creates a new variable whenever it is called.

Q Consider the following SDT

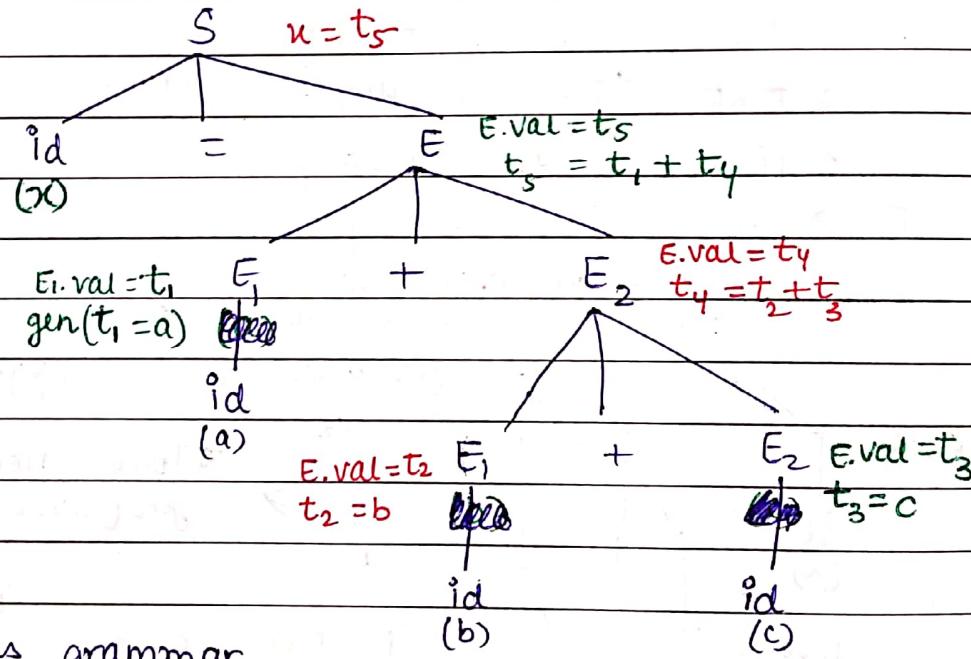
$$S \rightarrow id = E \quad \{ \text{gen}(id = E.\text{val}) \}$$

$$E \rightarrow E_1 + E_2 \quad \{ \begin{array}{l} E.\text{val} = \text{newTemp}() \\ \text{gen}(E.\text{val} = E_1.\text{val} + E_2.\text{val}) \end{array} \}$$

$$\begin{array}{l} | \\ id \quad | \\ \text{gen}(E.\text{val} = \text{New temp}(), id) \end{array}$$

$$\text{Q/P: } x = a + b + c$$

$$\text{O/P: ?}$$



Ambiguous grammar.

Cannot be expanded.

∴ Assume right to left associativity

$$\text{O/P: } t_1 = a$$

$$t_2 = b$$

$$t_3 = c$$

$$t_4 = t_2 + t_3$$

$$t_5 = t_1 + t_4$$

$$x = t_5$$

Q Construct SDT to store type information into symbol table.

I/P :- int x, y, z

O/P:-

S.NO	V.Name	V.Type
1	x	int
2	y	int
3	z	int

Symbol table

When we have to add an entry to symbol table, call :-

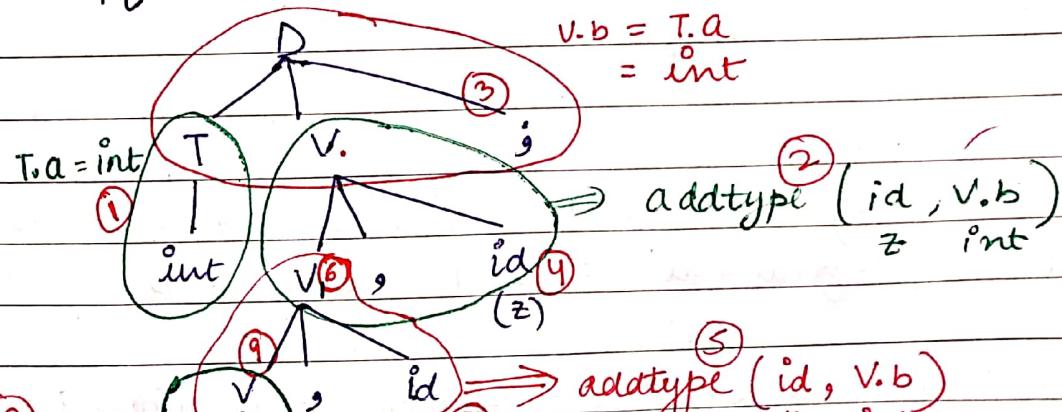
addtype (V.N , V.T)

Now write context-free grammar to generate all C declarations.

D \rightarrow TV; { V.b = T.a }

V \rightarrow V, id { addtype (id, V.b), V.b = V.b }
 | id { addtype (id, V.b) }

T \rightarrow int { T.a = int }
 | char { T.a = char }
 | float { T.a = float }



To store it in symbol table, "type" required.

It asks its parents & then to its siblings

- we used Inherited attribute.

'a' is synthesised attribute

'b' is inherited attribute

Parent gives variable type

Child gives variable name.

③ based on ①

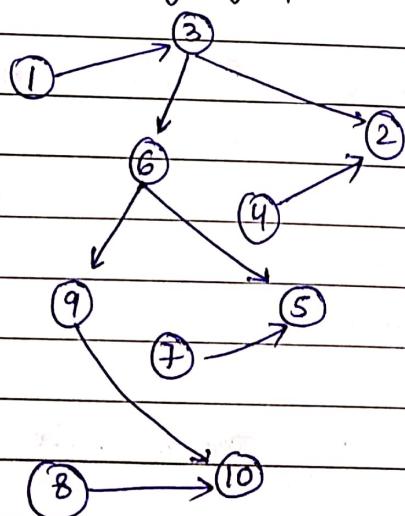
② " " ③ & ④

⑧ " " ⑤ & ⑦

⑨ " " ⑥

10 " " ⑧ & ⑨

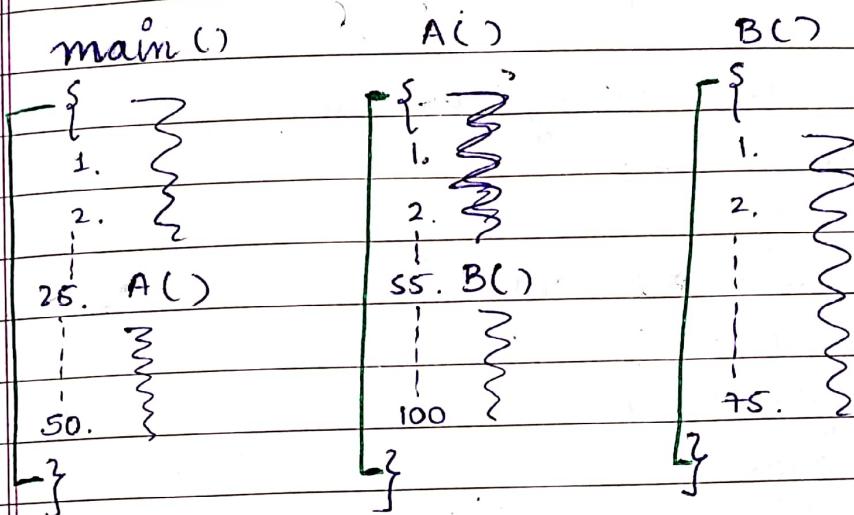
Dependency graph.



① Topological sort :- 1 3 6 4 2 9 7 5 8 10

② Topological sort :- 8 7 4 1 3 2 6 5 9 10

RUNTIME ENVIRONMENT :-



- * 24 lines of code executed in `main()`. Before going to execute `A()`, we need to store this work done. This is stored in a stack called ACTIVATION RECORD.
- * It contains local variables, ~~global~~ ^{temp} variables, return addr.
- * At line 25, we have to execute `A()`. When `A` is completed at its return statement, we return back to line 25.

main func. activation record holds:-

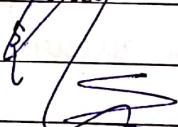
local variables, ~~global~~ ^{temporary} variables, return addr., control link (addr. of `A()`), actual parameters `A(m,n)`, system status (RAM, harddisk size, processor etc), access link (from which func. you ~~wants~~ ^{access} the code eg for `A()`, access link is `main()`)

Creating `main()` activation record as soon as `main()` starts executing is better.

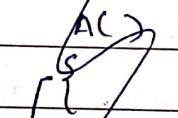
Main activation record initially contains Null.

- (1) whenever ~~func~~ A() is called, its activation record will be created & pushed inside stack
- (2) whenever A() is running, activation record gets updated
- (3) A() activation record contain :-
 - (i) local variables
 - (ii) temporary variables
 - (iii) return addr.
 - (iv) control link
 - (v) access link
 - (vi) actual parameters
 - (vii) machine status

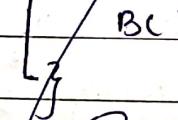
main()



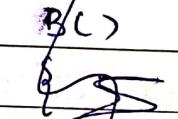
A()



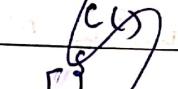
B();



C()



D()



main()

{ } }
A();[{ }]
B();

B();

[{ }]
C();[{ }]
J();;

G();;

D();

[{ }]
F();;[{ }]
E();[{ }]
F();[{ }]
A();

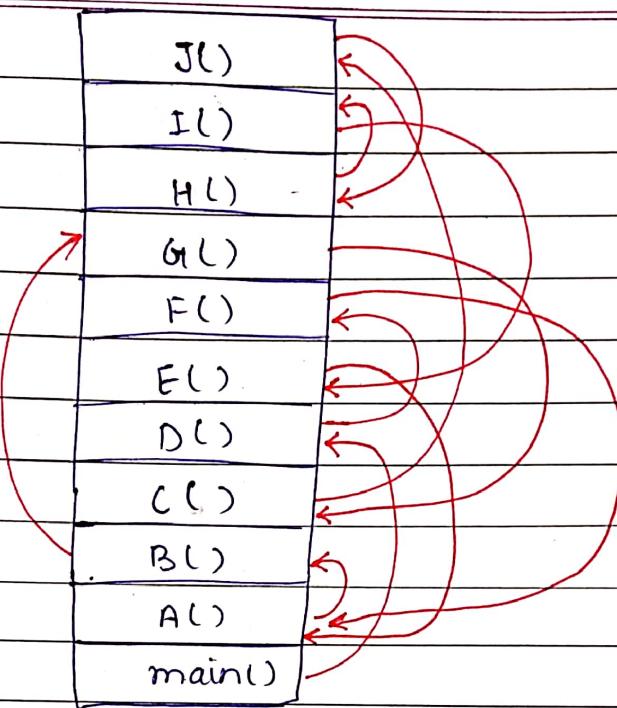
G();

[{ }]
C();
};
}A();;
}; } || E() ended.
H(); .

[{ }]

I();

[{ }]
E();;
J();
[{ }]
H();;
};
I();;O();;
}; } || main over.



main control link = DL

A() " " = B()

B() " " = G()

C() " " = J()

D() " " = F()

E() " " = A()

F() " " = A()

G() " " = C()

H() " " = ~~I()~~

I() " " = E()

J() " " = H()

A() access link = main()

B() access link = main()

C() access link = B()

D() " " = B()

E() " " = main()

F() " " = E()

G() " " = E()

H() " " = main()

A() functions
code is present
in which
function?

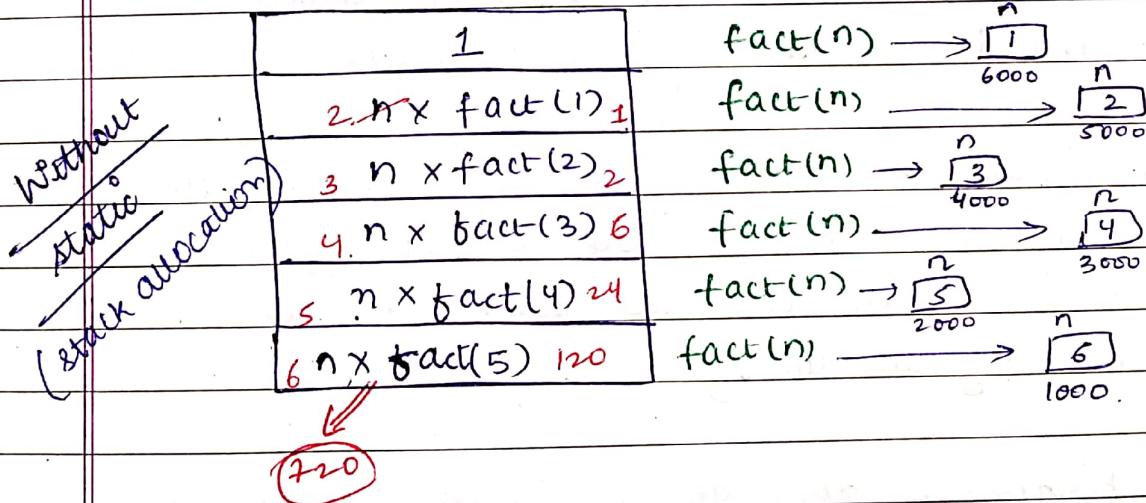
STORAGE ALLOCATION TECHNIQUES :-

1. static storage allocation
2. stack storage allocation
3. Heap storage allocation

```

fact(n)
{
    if (n ≤ 1)
        return n;
    else
        return [n * fact(n-1)]
}

```

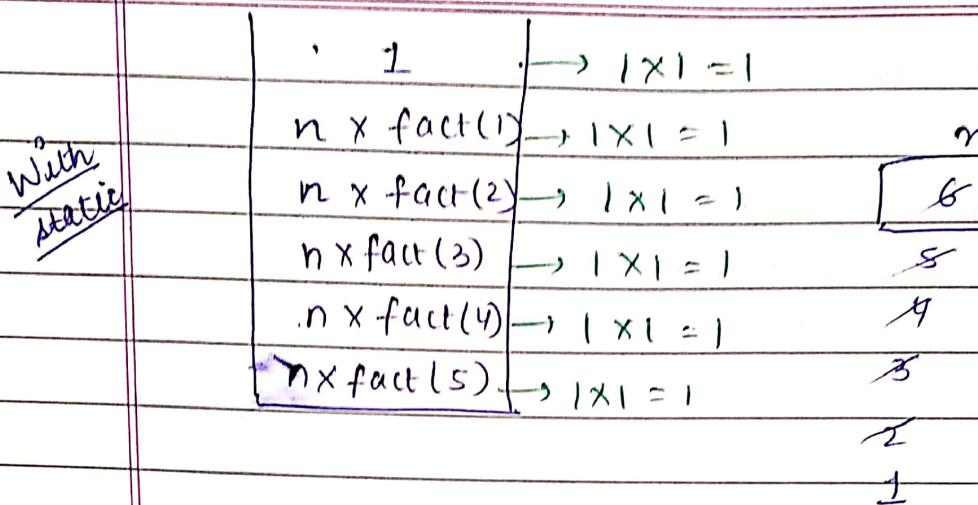


Here memory created 6 times because 6 func. calls
 $(\text{int } n)$ declaration.

STATIC STORAGE ALLOCATION :-

But if $\text{fact}(\text{static int } n)$ is used then
mem for n is created only once.

But here we end up saving space but
answer may be wrong as of recursion
is no longer occurring



- (1) for static variable, memory will allocate in static area. and memory is allocated only once.
For static variable, it occurs only once at compile time.
- (2) The drawback with static storage allocation is that it does not support recursion.
- (3) One more drawback of static storage allocation is dynamic data structure is not supported (i.e. user has no control over memory allocation).
- (4) If binding is done at compile time, it cannot be changed at run-time.

STACK STORAGE ALLOCATION :-

- (1) When a function is called, its activation record is created and is pushed inside the stack.
- (2) Whenever function completes its execution, it is popped out of the stack.

- ③ It supports recursion
- ④ Drawback with stack storage allocation is that dynamic data structure is not supported.
i.e. (whenever you want, you cannot insert/delete).
Func. call → Push ↴ it does not ask user. So not
Func. over → Pop ↴ dynamic.
- ⑤ If a function is over, it is popped. If this func. is required again, it has to be pushed ~~again~~ & computed again. This problem is solved by dynamic programming where we store the result somewhere.
- ⑥ In stack storage allocation, always local variables belong to new activation record only. This means if a stack contains 6 elements then only TOS is pointed.

HEAP STORAGE ALLOCATION :-

- ① Memory allocation & deallocation can be done anytime based on user requirement.
C ~~malloc~~ - allocation , free - deallocation
C++ new - allocation , delete - deallocation
Java new - allocation ; deallocation is automatic
- ② It supports dynamic data structure
- ③ It can implement recursion.

main ()

{

int a = 10, b = 20;

f (a * b, a - b, b);

return (a, b);

}

f (int b, int a, int c)

{

c += a - b;

a -= b + c;

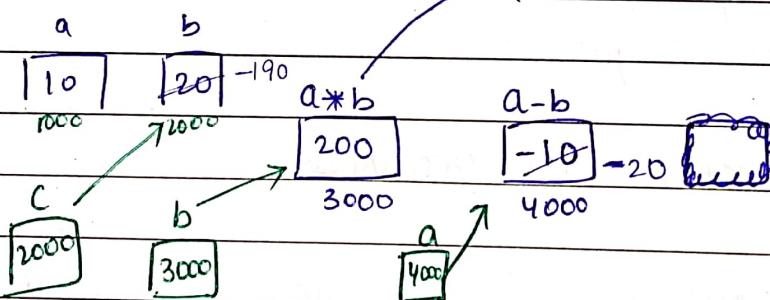
}

<u>ans</u>	a	b	b	a	c
By value.	10	20	200	-10	20

-20 -190

O/P: 10, 20.

By ref:-



O/P: 10, -190.

Lecture 6

24/11/2019

Date: / /
Page No. 101

WORKBOOK

A-3, C-4, B-2, D-1 (b).

Pg 57 Q2
Q3

Q4

int a, b, c
float d, e

S.N	V.N	V.T
1	a	int
2	b	int
3	c	int
4	d	float
5	e	float

~~effic.~~

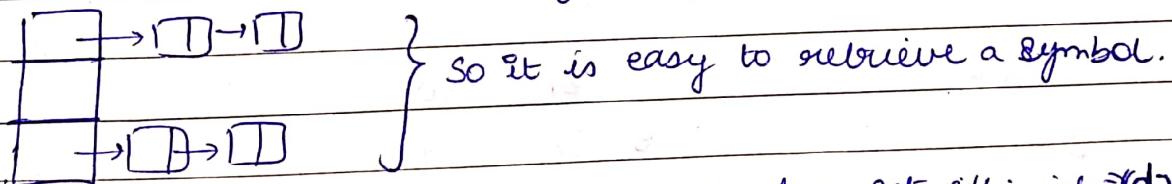
Both array & linked list can be used to implement symbol table -

Or.

[a] int | → [b] int | → [c] int | → [d] int | → [e] int [i]

But creation of symbol table using LL is better than array because in array we need to fix the size at the start when we don't know how many variables will be used in program.

But also LL can be modified by using chaining



∴ ans: (c). Had we asked most efficient (d)

Q5 (d)

(Q9) (a)

(Q13) (d)

(Q17) (d)

Q6 (a)

(Q10) (d)

(Q14) (c)

(Q18) (c)

Q7 (C)

(Q11) (C)

(Q15) (b)

(Q19) (C)

Q8 (C)

(Q12) (C)

(Q16) (C)

(Q20) (d)

21 $L(L+D)^*$ \Rightarrow identifier

regular exp or pattern matching
 $\underline{abc} \Rightarrow$ lexeme.

22 $S_1 \checkmark$ (a)
 $S_2 \checkmark$

23 (b).

24 (b). while $a();$
~~while~~ while $(a, b);$

Note $x = a, b \in$ ~~if~~ will take a value

$x = (a, b) \in$ It will take 5 value.

25 (c). Here we need to check i/p - layer.

syntax tree is o/p of syntax analysis but i/p to semantic analyser.

26 line 4: ; missing $x = 1$

integer & variable cannot be side by side.

(b).

27 (d).

31 40

28 58

32 34

29 35

33 0

30 22.

34. (a).

CH-2

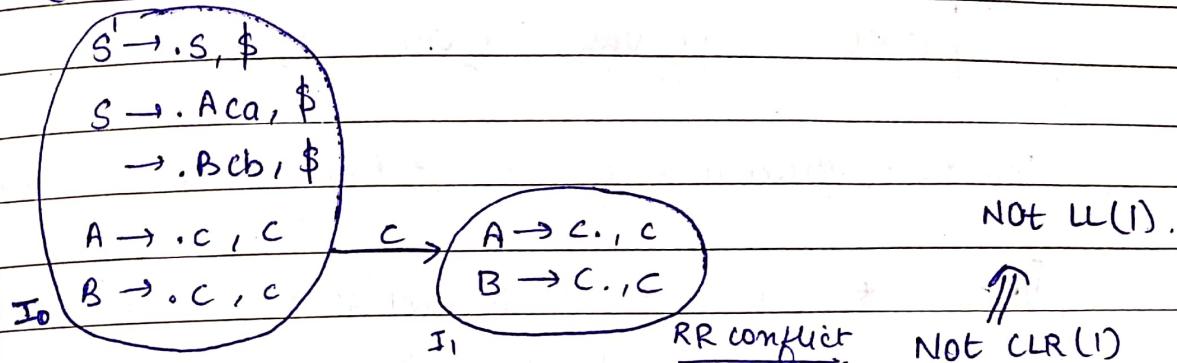
① (d). There maybe ambiguity

② (d).

③ (a)

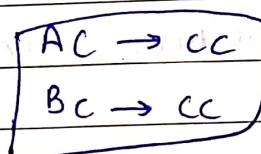
5 (d) construct DFA

6 (a).



when grammar given is big then expand that i/p first which gives immediate reduction as reduction is req. for any kind of conflict (SR, RR).

LL(1) X LL(2) X



(d)

8 over over over over $a \checkmark$
 $b \checkmark$
 $c \times$
 $d \checkmark$ } (c).

9 (b).

10 (d).

11 (b)

12 $a \rightarrow \checkmark$
 $b \rightarrow \checkmark$
 $c \rightarrow \times$
 $d \rightarrow \checkmark$ } ans (c) \checkmark

13 (b) Parser detects only syntax errors.

14 (c).

15 Shift reduce \rightarrow LR parser. (c).

16 (c) (19) (a)

17 (b) (20) (a)

22

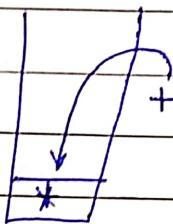
$$E \rightarrow \text{num.} \quad \{ E.\text{val} = \text{number.val} \}$$

$$| E+E \quad \{ E.\text{val} = E_1.\text{val} + E_2.\text{val} \}$$

$$| E * E \quad \{ E.\text{val} = E_1.\text{val} * E_2.\text{val} \}$$

Ambiguous grammar as + & * have same priority

Ex: $3 * 2 + 1$ result is ambiguous as + & * have same priority.



We don't know whether to push or to pop.

This is called Shift Reduce Conflict.

construct LALR(1) parsing table.

To ~~solve~~ this we pass this to YACC [Yet Another Compiler Compiler] tool.

Since this is ambiguous grammar, definitely some conflict will come in LR parser table. But when ~~use~~ YACC ~~parser~~ is used it convert SR conflict to a shift operation. It gives priority to shift operation.

In YACC if R₅R₆ conflict occurs then it gives priority to the reduction whose production comes first in grammar.

ans (C).

(b) also correct but it does not tell how YACC resolves.

\Rightarrow stronger ans. $\stackrel{\circ}{\text{is}} \text{(C)}$.

23.

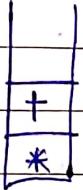
$$3 + 2 * 1.$$



$$3 + (2 * 1)$$

(5)

$$3 * 2 + 1$$



$$3 * (2 + 1)$$

(9)

* Pushed by YACC becoz
when SR conflict occurs,
priority is given to S.

→ whichever comes late is evaluated first as
priority is ~~eg~~ equal. (right to left assoc.)
ans (b)

24.

(b)

25.

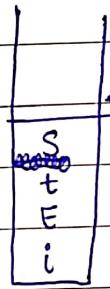
(d) Handle definition

26.

(b). CLR(I) should also have SR copy.

27.

$S \rightarrow iEES \mid iETSES$



i/p: e

Here already handle is present
at TOS. We are in dilemma whether
to pop & get iEES or push &
wait for iETSES.

We may push or pop \Rightarrow SR conflict

Since type of parser is not mentioned, we go with
general ans: \Rightarrow (c).

28. (C).

29.

Sticky handles :- XX, OX, I.

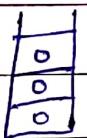
LR(1) item : $I X \rightarrow O.X, O\gamma$



variable
prefix
✓



Not a
variable
prefix
✗

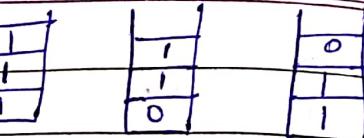


viable
prefix
✓



No.

No. 2
is not
possible



~~No. 2~~ is
can't be
together

∴ ans (a)

when i/p is 0. & production $X \rightarrow 1., 0$

This means if 0 comes then don't push. simply reduce

30 (A)

31 ~~(C)~~

32

$$\begin{aligned} S' &\rightarrow \cdot S, \$ \\ S &\rightarrow \cdot X, \$ \\ X &\rightarrow \cdot A+B, 1 \\ &\quad \cdot 2, 1 \\ A &\rightarrow \cdot 2, 1 \end{aligned}$$

2

$$\begin{aligned} X &\rightarrow 2., 4 \\ A &\rightarrow 2., 1 \end{aligned}$$

NOT CLR(1)

33 (C)

34 (a)

35 (b)

36 (C)

37 (a)

38 (C)

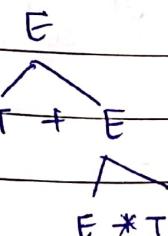
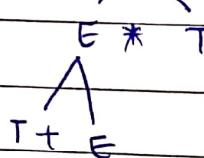
39 (C)

(40) (C)

(41) (a)

(42) (C)

(43). E



Non empty \Rightarrow non E

ambiguous.

44 $F \rightarrow idF' \quad \left. \begin{array}{l} \text{option(a) } X \\ \text{option(b) } X \end{array} \right\}$ (c) X
 $F' \rightarrow E \mid +2F' \quad \left. \begin{array}{l} \\ \text{(b) } X \end{array} \right\}$

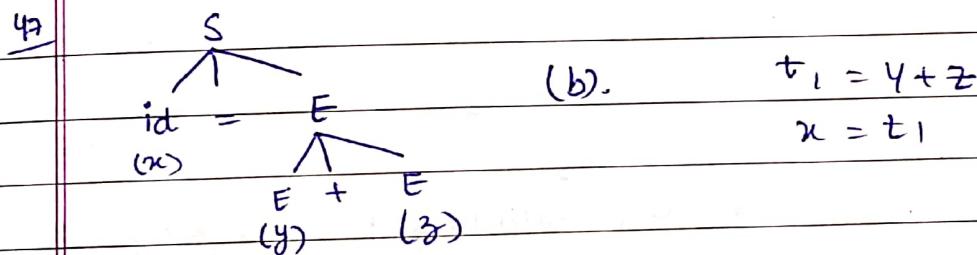
↓ after removing null production

$F \rightarrow idF' \mid id. \quad \left. \begin{array}{l} \text{matches option(a) now} \end{array} \right\}$

$F' \rightarrow +2F' \mid +2 \quad \left. \begin{array}{l} \\ \text{(a).} \end{array} \right\}$

45 For () eras ; inside and not,
 \therefore (b).

46 b has higher priority than a ans (b).



48 (c).

49 (a).

50 SR parser \equiv LR parser \equiv Bottom up parser.

(a).

51 Every reg. set has atleast 1 grammar.
 Every reg. grammar can be satisfied by LR(0).
 Reg. set \equiv Reg. lang.

every Reg. lang is LR(1).

Every reg. lang is unambiguous.

But reg. grammar can be ambiguous.

P - False. Some reg. grammar can be ambiguous.

Q - True

(53) (a)

(63) (b)

(54) (a)

(64) (d)

(55) (d)

(65) (b)

(56) (d)

(66) (c)

(57) (c)

(67) dynamic type checking is done at run time. Type checking at compile time is lesser than that of runtime.

(58) (a)

~~a = 20~~

(59) (a)

b = 30.

(60) (d)

if ($a < b$)

(61) (c)

{

(62) (b)

}

68 (c)

?

69 (a)

else {

70 (c)

?

71 (c)

during runtime type checking it won't go in else part. But there may be error in else part.

72

∴ dynamic TC won't find all errors.

73

∴ ans (c).

74

75

76

77

78

85. PQS Q S R.

79

80

81

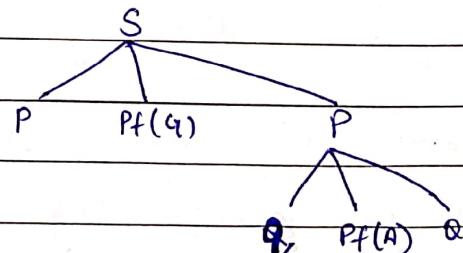
82 (d)

83 (b)

84

85 No option matches.

86 all are false (d)



Q AAAAT

83

$$\begin{array}{l} A \rightarrow SB|S \\ B \rightarrow ,SB|;S \\ S \rightarrow a \end{array} \xrightarrow{\begin{array}{l} \text{replaces} \\ \text{by } a \end{array}} \begin{array}{l} A \rightarrow AB|a \\ B \rightarrow ,AB|;a \end{array}$$

(c) is not an operator grammar.

b/w (a) & (b) check which is equivalent to given grammar.

ans (b).

T7

Lift sentential form \equiv Liftmost Derivation Tree.

$$\begin{array}{c} ABCDEF \\ \downarrow \\ \text{First}(D)=d. \end{array} \left\{ \begin{array}{l} \text{Follow}(C) \text{ in LMD} = d. \\ \text{Follow}(C) \text{ in RMD} = d \end{array} \right.$$

Follow depends on what comes next. It does not depend upon LMD or RMD.

(C).

T9

Handles: as, b

on stack we can have, as, b, a

aas ✓	} Possible in stack.
a ✓	
aa ✓	

bb X

Both (a) & (d) lead to reduction but bb cannot occur in stack. \therefore ans. (A)

T8

(a)

$$S \rightarrow AB$$

$$A \rightarrow a$$

$$B \rightarrow b$$

LMD	RMD
S	S
AB	AB

Not same

Derivation Trees are same but derivations are not same.

CH-3

① (b)

② statically typed \equiv only 1 type lang.
untyped lang \equiv NO type
(c)

③ ~~ans~~ ~~III~~ II X III X True when few L-attributes
ans (b).

④ 1 X 3 X (a) ✓

⑤ 1 ✓
2 ✓ } (d)
3 ✓

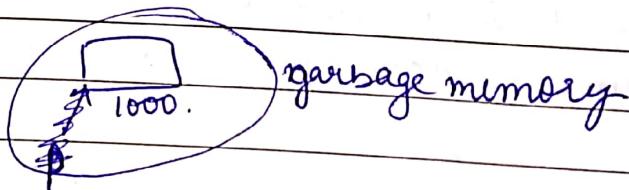
⑥ (c)

⑦ (d) Prog. counter \cong return addr.
reg. values \cong local variables

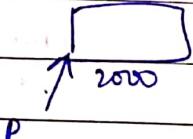
⑧ (a)

⑨ (c)

⑩ $P = \text{malloc}$



$P = \text{malloc}$



P

CH-4 INTERMEDIATE CODE GENERATOR

Ex

$$x = a + b * c$$



$$t_1 = b * c;$$

$$t_2 = a + t_1;$$

$$x = t_2;$$

Ex

$$x = (a+b) * (a-b) + c$$

advantage with intermediate code generation is
will get machine independent code.

REPRESENTATION OF IC:-

LINEAR CODE

eg: source
3 address
code
(max 3 addr.)

$$t_1 = a + b$$

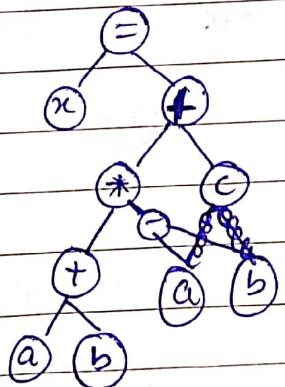
3 address
code
(max 3 addr.)

Postfix
notation

$$(a+b)(a-b)*c +$$

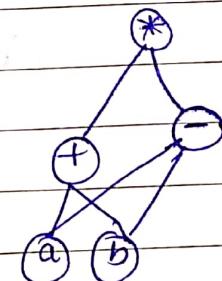
TREE FORM

Syntax
Tree



DAG

common subexpression
elimination



Ex3 $x = ((a+b) * (a+b)) - ((a+b) * (a-b))$

$$t_1 = a + b$$

$$t_2 = t_1 * t_1$$

$$t_3 = a - b$$

$$t_4 = t_1 * t_3$$

$$t_2 = a + b$$

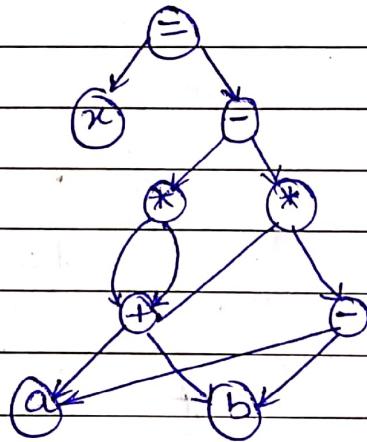
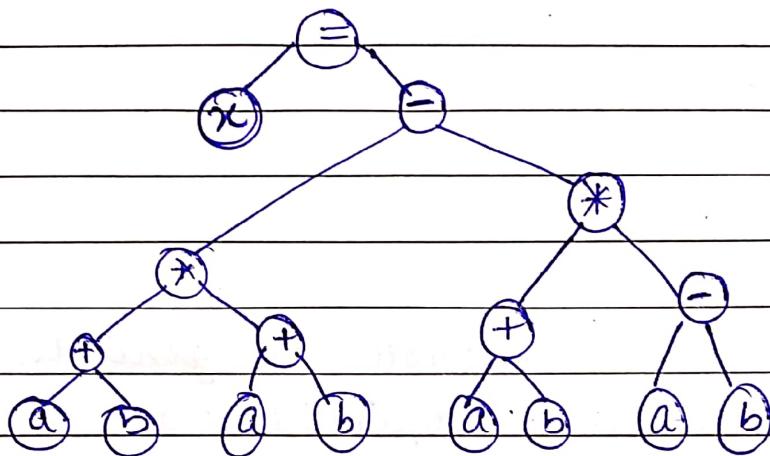
$$t_3 = t_1 * t_2$$

$$t_4 = a + b$$

$$t_5 = a - b$$

$xab + ab * ab + ab - * - =$

postfix Notation



Q $x = a + a + a + a + a$

$t_1 = a + a$

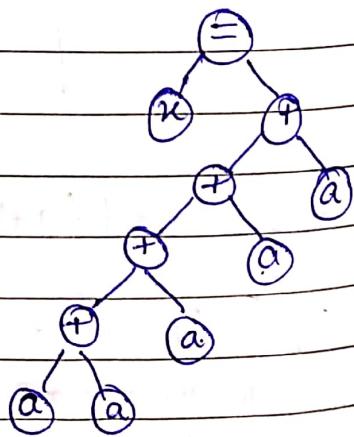
$t_2 = t_1 + a$

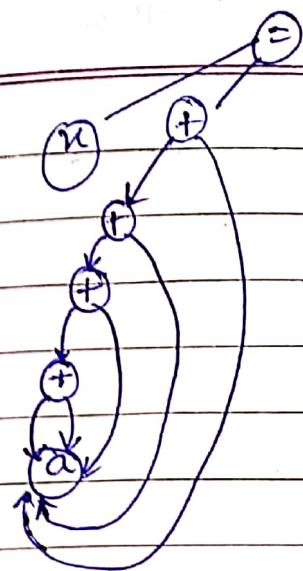
$t_3 = t_2 + a$

$t_4 = t_3 + a$

$x = t_4$

$a a + a + a + a +$





THREE ADDRESS CODE :-

- ① $x = y \text{ op. } z$
- ② $x = \text{op. } y$.
- ③ $x = y$
- ④ $x = a[i]$
- ⑤ $a[i] = x$
- ⑥ $x = a[i][j]$ No. there are 4 addr.
- ⑦ ~~⑧~~ $\text{if } (x < y) \text{ goto } z$
- ⑧ $\text{goto } x$.

Q. Write Three Address Code. for following C program.

$i = 9;$

$\text{for}(i = 25; i \leq 93; i++)$

i

$x = a + b * c;$

}

10002 218

(op)

10m

1000	$i = 9$	BLOCK 1.
1001	$i = 25$	
1002	if $i \leq 93$ goto 1004	B2
1003	goto 1009	B3.
1004	$t_1 = b * c$	
1005	$t_2 = a + t_1$	
1006	$x = t_2$	B4
1007	$i = i + 1$	
1008	goto 1002.	
1009		B5

- This shows for every recursion, non recursion is possible.
- Inside comp., everything is ICG. i.e. further changed to target code.
- Filling addresses which are not available initially after completion of intermediate code. This process is called Backpatching.

1st statement is a leader.

Target of goto is a leader.

Next statement after goto is leader

? How to
find leaders

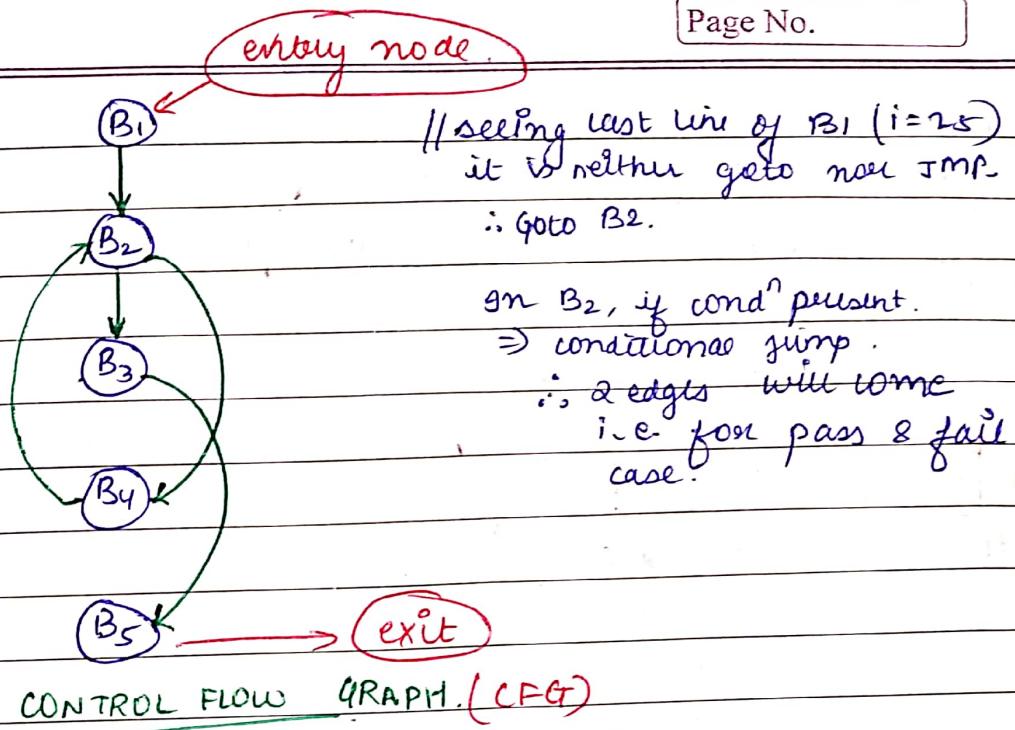
Leaders:- 1000, 1002, 1003, 1004, 1009.

One leader to next leader is one basic block.

No. of Basic Blocks = No. of leaders

Def. of basic block :-

It is a set of 3 addr. statements where control enters at the beginning & leaves at the end. In between there is no halt or jump.



No. of nodes in CFG = 7.

No. of edges in CFG = 7

DFT :- $B_1 \ B_2 \ B_4 \ B_2$

\uparrow
again same. \Rightarrow cycle present

\therefore Apply DFT on CFG to detect cycle or loop.

Q Write three address code for following C-program.

$i = 00093;$

switch (i)

{

case 1 : $x_1 = a_1 + b_1 * c_1;$

break;

case 90 : $x_2 = a_2 + b_2 *$

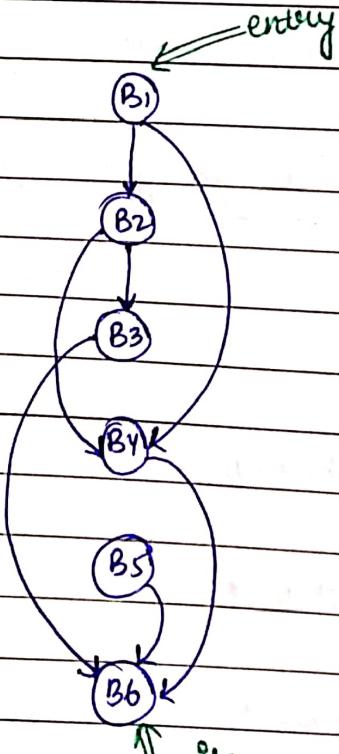
break;

default : $x_3 = a_3 + b_3 * c_3$

}

L1	1000	$i = 93$	B1
1001	if $i = 1$ goto 1007		
b2	1002	goto if $i = 90$ goto 1011	B2
b3	1003	$t_5 = b_3 * c_3$	
1004	$t_6 = a_3 + t_5$ B3		
1005	$x_3 = t_6$		
b4	1006	goto 1000	1015
b4	1007	$t_1 = b_1 * c$	
1008	$t_2 = a_1 + t_1$ B4		
1009	$x_1 = t_2$		
1010	goto 10 ¹⁵ 1016		
b5	1011	$t_3 = b_2 * c_2$	
1012	$t_4 = a_2 + t_3$ B5		
1013	$x_2 = t_4$		
1014	goto 10 ¹⁵ 1016		
b6	1015	HALT	B6

There are 6 leaders



Q

Write TAC for following statement in C language:-

$$x = a[i][j]$$

$$a[1 \dots 90, 1 \dots 75]$$

•

c (size of element)

row major order.

$$\text{loc}(a[i][j]) = a + [(i-1)*75 + (j-1)] \times c$$

$$1000 \quad t_1 = i - 1$$

$$1001 \quad t_2 = t_1 * 75$$

$$1002 \quad t_3 = j - 1$$

$$1003 \quad t_4 = t_2 + t_3$$

$$1004 \quad t_5 = t_4 \times c$$

$$1005 \quad t_6 = a + t_5$$

$$1006 \quad x = *t_6$$

unary operator. [$x = \text{op } y$]

REPRESENTATION OF TAC :-

1. Quadruple
2. triple
3. indirect triple.

$$\text{ex} \quad x = a + b * c - d / e + f$$

$$t_1 = b * c$$

$$t_2 = d / e$$

$$t_3 = a + t_1$$

$$t_4 = t_3 - t_2$$

$$t_5 = t_4 + f$$

$$x = t_5$$

Date: / /

Page No.

Quadruple Number	Operator	operand 1	operand 2	Result	
1	*	b	c	t ₁	
2	/	d	e	t ₂	
3	+	a	t ₁	t ₃	
4	-	t ₃	t ₂	t ₄	
5	+	t ₄	f	t ₅	
6	=	t ₅		x	

Triple

S.No.	Operator.	Op 1	Op.2	wherever Result is stored
1	*	b	c	→ on index itself
2	/	d	e	
3	+	a	①	
4	-	③	②	
5	+	④	f	
6	=	⑤ x	⑥	

Saved space but we cannot move result wherever we want.

Indirecttriple

S.NO.	
1.	
2.	
3.	→ 5000
4.	
5.	
6.	

Here we overcome disadv. of triple by shifting the result from index to a particular address.

STATIC SINGLE ASSIGNMENT.

It is 3 addr. code where each address has single assignment.

$$\begin{aligned} t_1 &= z * a \\ t_2 &= y + t_1 \\ x &= t_2 \end{aligned}$$

} Every variable has single meaning. \Rightarrow SSA ✓

$$\begin{aligned} a &= z * a \\ z &= y + a \end{aligned}$$

} Here 1 variable (a) is having 2 meanings. \Rightarrow NOT SSA.

$$\begin{aligned} z &= z * a \\ z &= y + z \\ \cancel{t_1} &= z * a \\ \cancel{x} &= y + t_1 \end{aligned}$$

} Here z has 2 meaning \Rightarrow SSA X
} Every variable has just 1 meaning. \Rightarrow SSA ✓

Q $x = a + b * a + b$

c = b * a	x = b * a	we cannot store $b * a$ into a or b because they have to be used afterwards. a & b are live variables so they cannot be replaced.
a = a + c	a = a + x	
$x = a + b$	$x = a + b$	3 var.

Total 4 variables

Total 1 temp. Variable.

$$c = b * a$$

$$d = a + c$$

$$x = d + b$$

Total variables = 5

Total temp. variables = 2

$$x = (a * b) + (a * b) + (a * b)$$

TAC.

$$t_1 = a * b$$

$$t_2 = t_1 + t_1$$

$$x = t_2 + t_1$$

$$\text{Total var} = 3$$

$$\text{Total temp var} = 0$$

SSA

$$t_1 = a * b$$

$$t_2 = t_1 + t_1$$

$$x = t_2 + t_1$$

$$\text{Total var} = 5$$

$$\text{Total temp var} = 2$$

$$a \quad x = a * b$$

$$y = x + c$$

$$z = a * b$$

$$w = y + z$$

$$w = y + z$$

$$= x + c + z \quad [\text{expanded } y]$$

$$= x + c + a * b \quad [\text{expanded } z]$$

$$= a * b + c + a * b$$

$$= a * b + c + a * b$$

TAC

$$a = a * b$$

$$c = a + c$$

$$w = c + a$$

$$\text{Total var} = 4$$

$$\text{Temp var} = 0$$

SSA

$$t_1 = a * b$$

$$t_2 = t_1 + c$$

$$w = t_2 + t_1$$

$$\text{Total var} = 6$$

$$\text{Temp var} = 2$$