INDEX

Compiler Design

Name .. Akansha Mittal

Roll No. .....

S.No.	DATE	TOPIC	Page No.	Remarks
1.		Introduction to compiler design	3-10	
2.		Lexical Analyzer	13-19	
3.		Parsers	21-75	
		→ Grammar	21-31	
		→ Parsers	31-32	
		→ Top down parser	33-	
		• Recursive descent parser	33-36	
		• LL(1) parser	36-49	
		→ Bottom up Parser	50-	
		• LR parsers	50-74	
		• Operator precedence parser	75	
4.		Syntax directed translations	81-95	
5.		Runtime Environment	99-103	
6.		Intermediate code generation	105-117	
7.		Code optimization	119-127	

Lect-1

Compiler Design

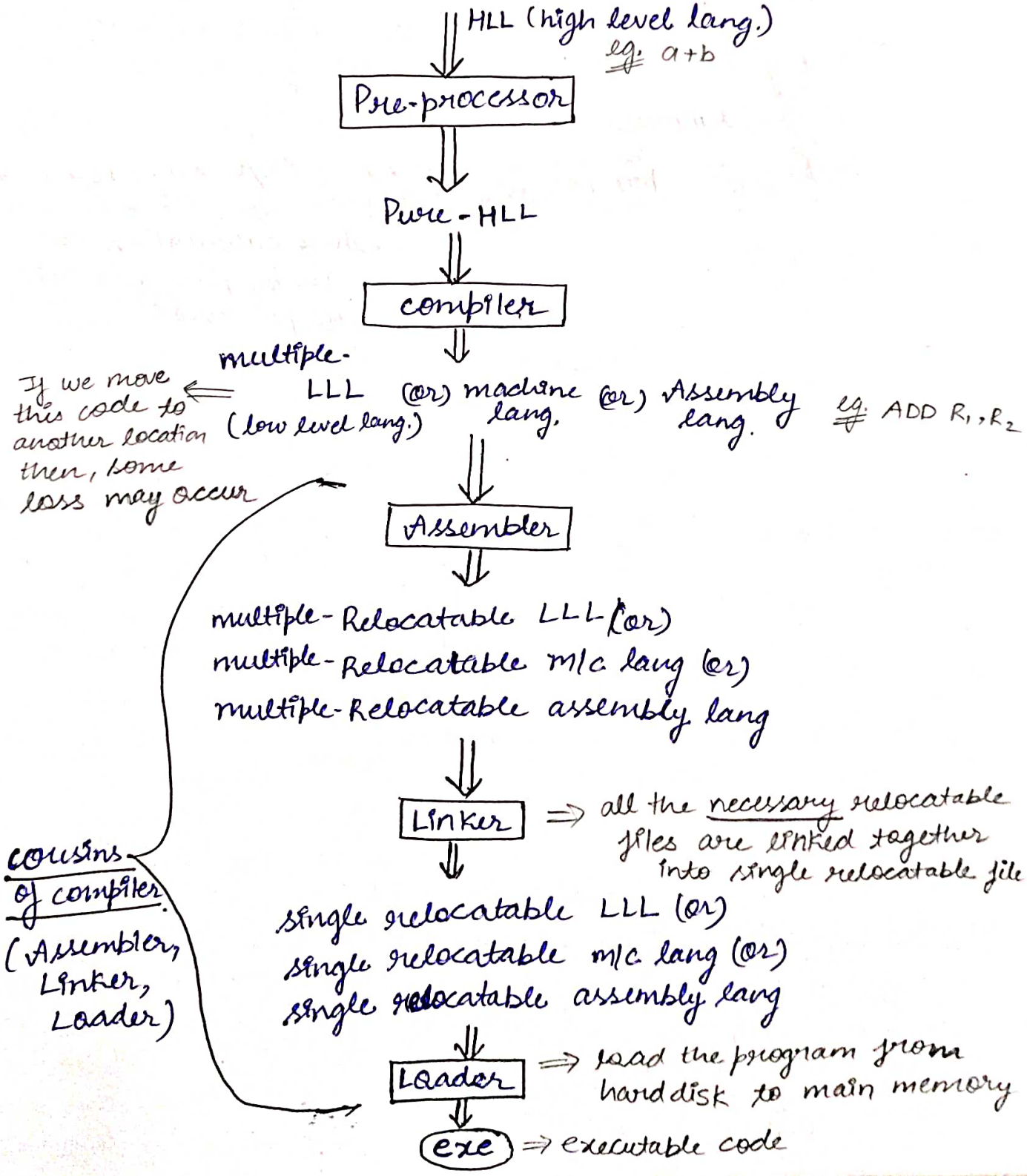
1. Introduction to compiler design
2. Lexical Analyzer
3. Parsers
4. Syntax directed translation
5. Intermediate code generation
6. Code optimization

Workbook + Notes ✓

+ PYQ

Introduction to Compiler design

Language processing :-



(4)

Note :-

(1) Pre-processor don't do any calculation but it will do only substitution. (replace user-defined shortcuts by actual)

eg. `#include<stdio.h>` → include file

`#define square(x) x*x` → replace square(x) with x*x everywhere in code.

eg.

```
#define square(x) x*x
main()
{
  int y, x = 4;
  y = 64/square(x);
  printf(y);
}
```

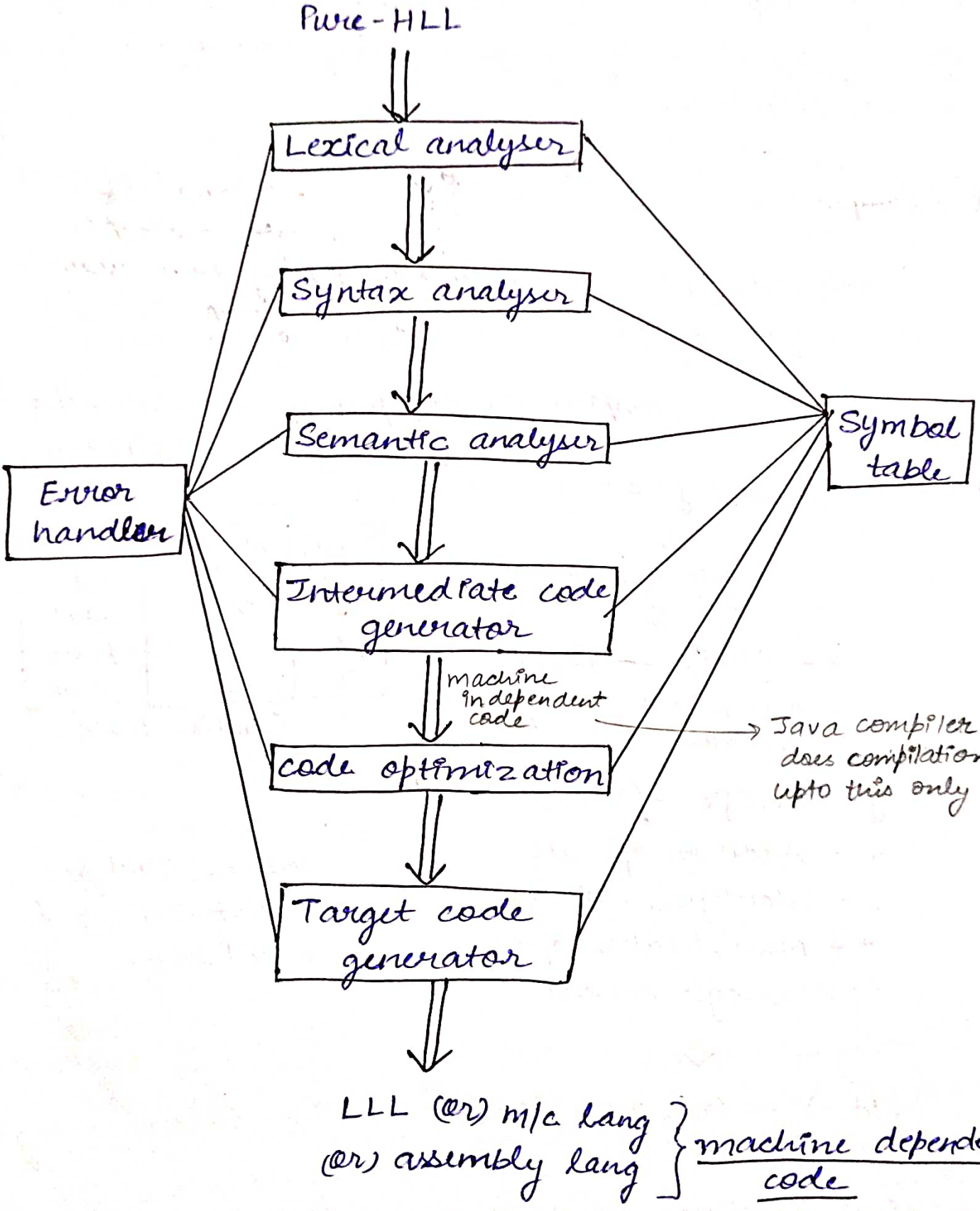
This line will be deleted by preprocessor after replacing square(x) by x*x everywhere in the program.

pre-processor → 64/x*x = 64/4 *4 = 16*4 = 64

These calculations are done by processor not pre-processor

→ Phases of compiler

("functions" acc. to the terminology of programming)



- Symbol table contains identifiers (variable & func" name) along with its information.
- During compilation, if any of the phase want, they can store something in Symbol table.

During compilation, if any problem occur during any of the phase of compiler, they can inform it to error handler.

only identifiers are stored in symbol table bcz only they can vary acc. to program while keywords, operators etc are fixed for a language (they are already known to compiler)

★ Example

```
float x, y, z;
x = y + z * 60
```

Let's see how compiler is compiling 2nd statement i.e, x = y + z * 60

HLL string ⇒ x = y + z * 60

⇓

Lexical Analyser

⇓

x - identifier - (id, 1)

↖ token no. (S.No. of x in symbol table)

= - Assignment operator

y - identifier - (id, 2)

\+ - Addition operator

z - identifier - (id, 3)

* - Multiplication operator

60 - integer constant

⇓

(id, 1) = (id, 2) + (id, 3) * 60 ⇒ token string

This symbol table was made during compilation of 1st statement (float x,y,z)

S.No.	variable name	variable type
1	x	float
2	y	float
3	z	float

Symbol table

It doesn't contain any func" bcz given code doesn't have any func"

~~Lexical analyser gives token no. to identifiers.~~

This type info is stored by semantic analyser during compilation of "float x,y,z".

Lexical Analyser :- dividing the HLL string to token. It identifies tokens based on its rules.

i/p ⇒ HLL string , o/p ⇒ token string

Syntax Analyser :- It checks whether it can generate syntax tree using its grammars or not.

i/p : token string

o/p : syntax tree (parse tree)

If possible ⇒ NO syntax error

Not possible ⇒ Syntax error

(id,1) = (id,2) + (id,3) * 60

⇓

Syntax Analyser ⟸

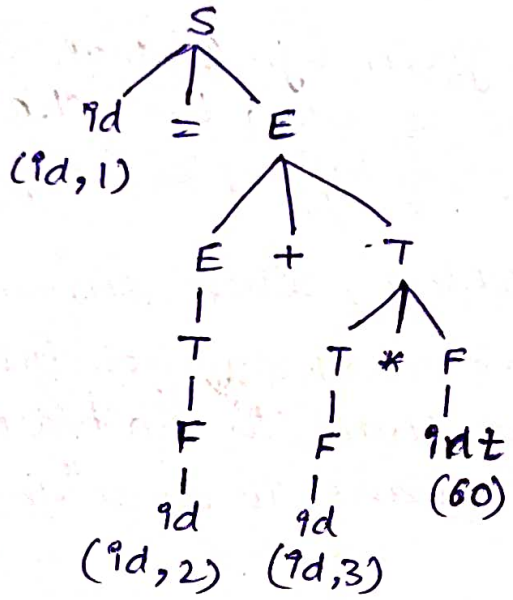
$S \rightarrow id = E$

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow id / int$

Syntax analyser generally contains many grammars using which it generates syntax tree.



- Syntax analyser is also called **Parser**.

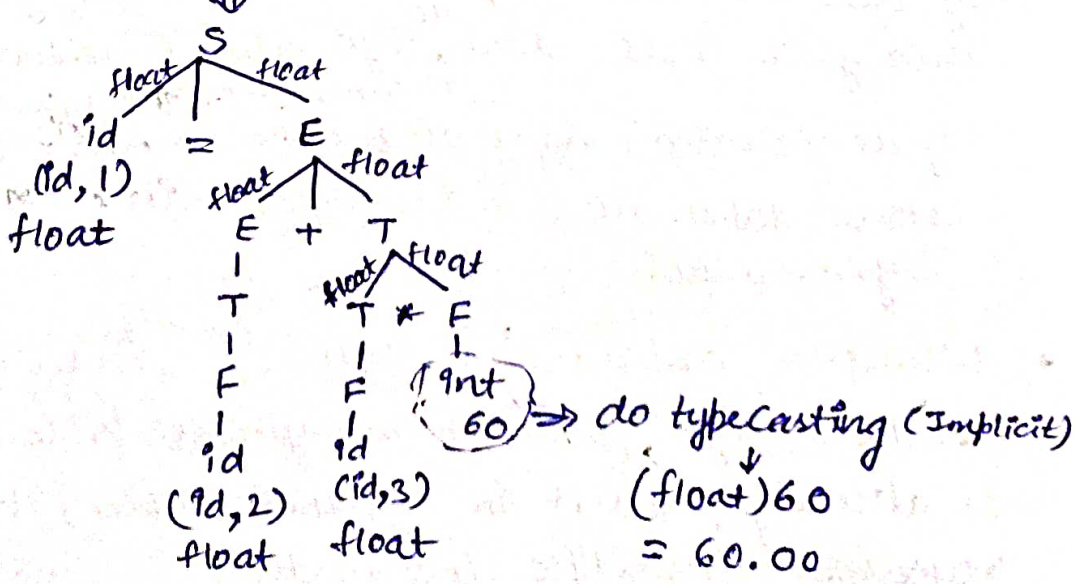
Semantic analyser

i/p : Syntax tree

o/p : Modified syntax tree

purpose of Semantic Analyser is type checking.

Semantic Analyser



- While doing type checking, to know the type, semantic analyser will know type from the symbol table of corresponding token no..

- Any of the phase can store in symbol table. But if we want to select one, then, semantic analyser is the one who stores in symbol table. ~~because it needs type information again & again.~~

- If token no. is not given by lexical analyser then, other phases have to scan the whole symbol table for the knowing the type of corresponding token.

- While doing type checking, when semantic analyser goes to corresponding row of symbol table using token no. & finds that there is no variable type available for that, it means it is a "undeclared variable"

~~So, symbol table do~~

- Functionalities of Semantic Analyser :-
  1) Type checking.
  2) Identifying undeclared. ~~variables~~
  3) Multiple declaration.

- If declaration is present ⇒ directly semantic analyser will store all info about identifier in symbol table. (token no, name, type)
  If declaration is not present ⇒ lexical analyser will store token no. & identifier name of identifier in symbol table.

- Whenever declaration comes, directly semantic analyser goes & store info about it in symbol table. If its entry is already present in symbol table, it means it is a ~~variable~~ "multiple declaration"

Q Which phase stores info in symbol table?

All can store

↓ If it is not in option

semantic analyser

↓ If it is also not in option

Lexical analyser

↓ If it is also not in option

select any phase

Intermediate code generator

(O/p of semantic analyser)

⇓

Intermediate code generator

⇓

$t_1 = z * 60.0$
$t_2 = y + t_1$
$x = t_2$

} machine independent code

code optimization

⇓

code optimization ⇒ This phase is optional

⇓

$t_1 = z * 60.0$
$x = y + t_1$

⇓

target code generation

⇓

MUL R_1, R_2
ADD R_3, R_4

} machine dependent code

NOTE :-

Java supports portability bcz its compiler does compilation only till intermediate code generation, so, machine independent code will be generated while C compiler runs all phases of compiler and generates machine dependent code.

- Phases of compiler :-

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate code generation

} Front end of compiler

5) code optimization → middle end of compiler
6) Target code generation → Back end of compiler

code optimization ⇒ optional phase of compiler

Compiler itself is a long code. If whole code of compiler can be stored in main memory then, go for Single pass compiler otherwise, we have to go for Multi pass compiler.

Single pass compiler ⇒ More space, less time
Multi pass compiler ⇒ Less space, More time

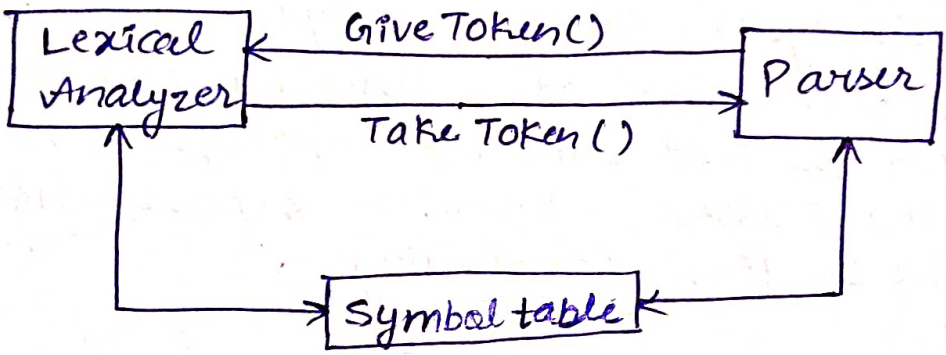
↓ Virtual m/m concept is used to run compiler code

↓ bcz context switching takes place many times.

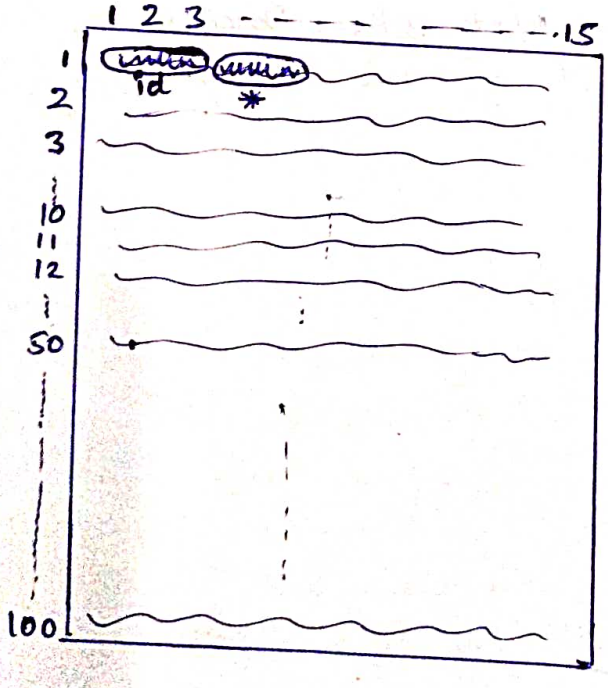
Lexical Analyzer

Purpose of Lexical Analyzer :- Tokenization

- Lexical Analyzer does not generate token on its own, it generates token only on the demand of Parser.



eg. There is a C program ~~which~~ of 100 rows and 15 columns.

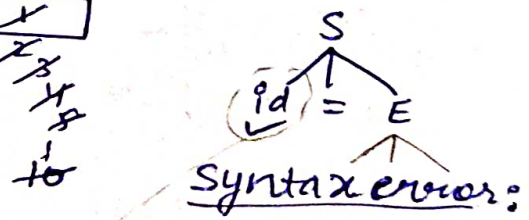


$S \rightarrow w|w|w|w|id=E|w|w|-$

$E \rightarrow w|w|w|$

position of lexical analyzer

Row #	Cal #
1	1



Syntax error:
expected '=' but '*' came at row 1 column 10

This info is given by lexical analyzer to error handler

Syntax analyser informed about error to error handler

still compilation continue

Here, 'id' came so, it will call semantic analyzer for type checking

NOTE :-

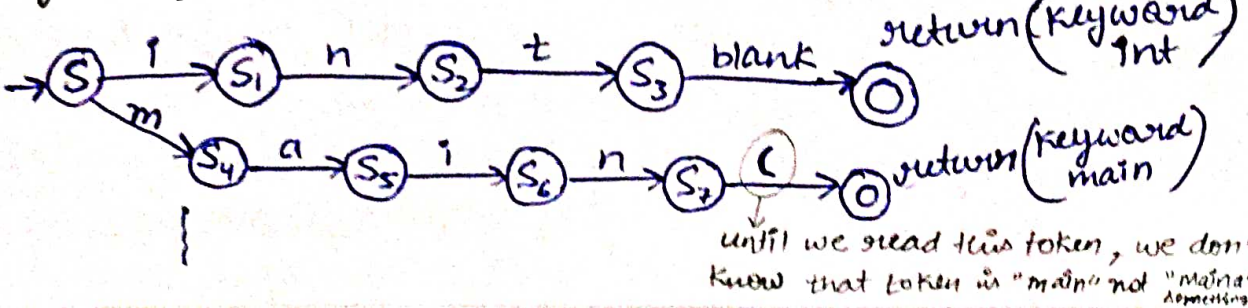
1) Lexical Analyzer is 1st-phase of compiler which will scan and divide the given program into meaningful words which are known as tokens.
2) Lexical Analyzer is known as scanner bcz it will scan the given program.
3) Lexical Analyzer will help in giving error msgs by providing row no. & column no..
4) Lexical Analyzer, Syntax Analyzer & Semantic Analyzer will work at a time by synchronizing each other so that within one pass, all 3-will complete their work, otherwise 3-passes needed which will be time consuming.

Q1 Consider the following c-program:

```
1   2    3 4
int main()
5 {                                   Not token (meant for users only)
/* finding maximum element out of a && b */
6 int 7 a 8, 9 b 10, 11 max 12;
13 a 14 = 15 10 16; 17 b 18 = 19 20 20;
21 if 22 23 24 25 (a < b) 26
27 max 28 = 29 b 30;
   else 31
32 max 33 = 34 a 35;
36 return 37 (38 max) 39;
} 40 41
```

Count total no. of tokens present in the above c-program? 41 Ans

Sol:-



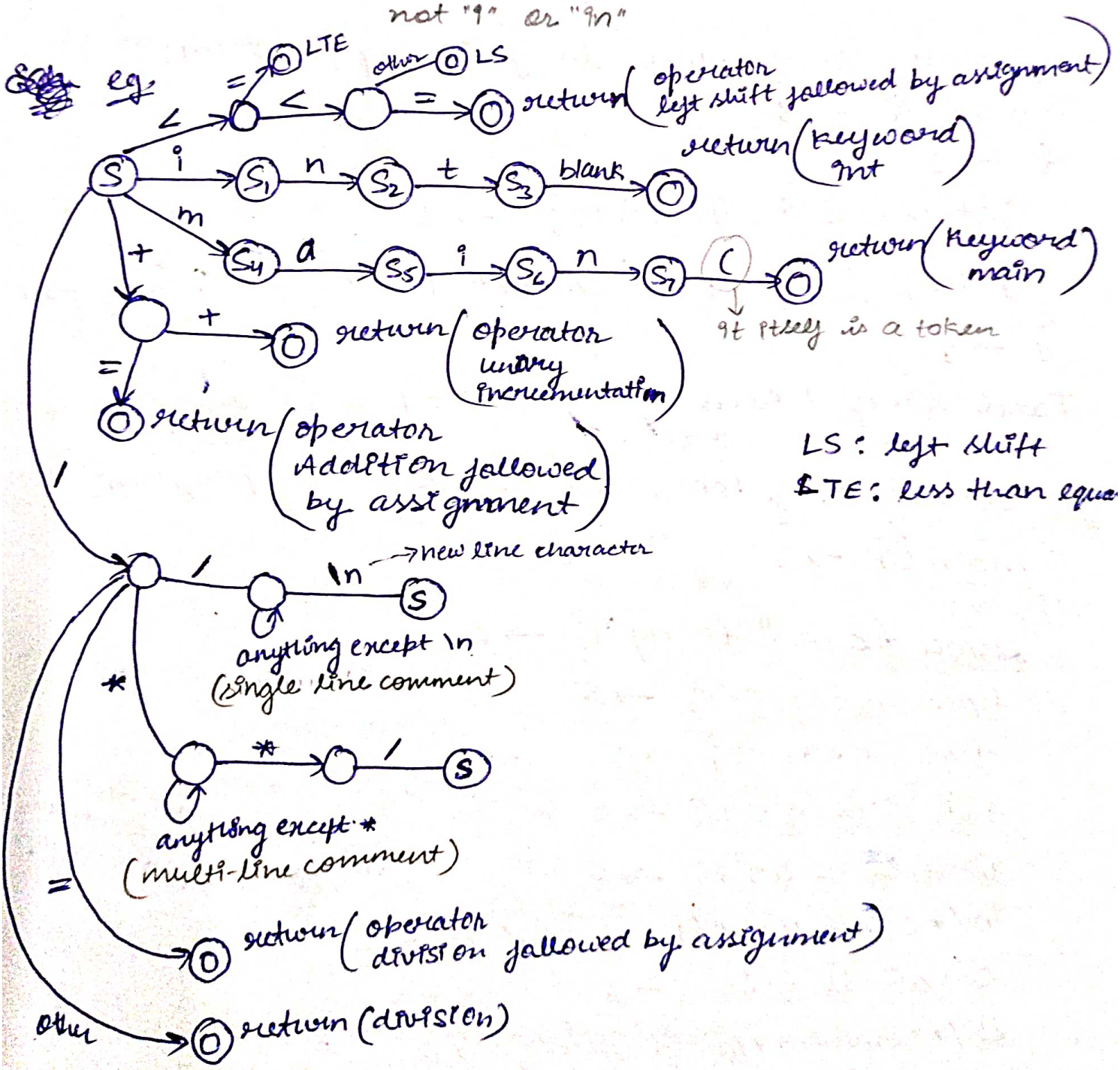
Imp
NOTE :-

1) Lexical Analyzer uses DFA to do tokenization.

DFA : deterministic finite automata

2) While doing tokenization, we should always give importance to longest matching.

eg: i => token
in => token
int => token
} If "int" is given then, consider "int" as token not "i" or "in"



LS : left shift
LTE : less than equa

Q2 Find no. of tokens in the following program:

```
main()   → 3
}        → 1
x = a+b * c;  → 8
int x, a, b, c;  → 9
y = x+a;  → 6
{   → 1
```

This code contains syntax and semantic error but not lexical error.

Total no. of tokens = 3+1+8+9+6+1 = 28 Ans

Q3 Count no. of tokens in following program:

```
main()()  → 3
{   → 1
int x == 10, y << = 10;   → 9
printf("%d %d %d correct answer is");(x);   → 7
}  → 1
```

Total no. of tokens = 1+3+9+7+1 = 21 Ans

Q4 Count no. of token in the following program:

* ⇒ operator
c ⇒ identifier
*c ⇒ combinely is not any type of token

```
main()()  → 3
{  → 1
char *c = "string";  → 6
float b = 20.7;  → 5
char d = 'e';  → 5
int f = 257;  → 5
in t f = 257;  → 6
in/* comment */t f = 257;  → 6
ch ar d = 'e';  → 6
ch/* comment */ar d = 'e';  → 6
}  → 1
```

Here, comment is not counted as token but it is treated as gap

Total no. of tokens = 1+3+6+5+5+5+6+6+6+6+1 = 50 Ans

Q5 Lexical Analyzer uses the following patterns to recognize tokens T_1, T_2 and T_3 over the alphabet $\{a, b, c\}$.

T_1 : a? (b/c)* a

T_2 : b? (a/c)* b

T_3 : c? (b/a)* c

Note that x? means 0 (or) 1 occurences of the symbol x. Note also that the analyzer outputs token that matches the longest possible prefix of the string.

If bbaacabc is processed by analyzer, which one of the following is the sequence of tokens of outputs?

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

Ans

bbaacabc

bbaac, abc

T_3 T_3

Taking longest matching possible

Q6 Find the number of tokens for each statement below if no lexical error in that statement

1) printf ("%d" , printf ("gate")) ; ⇒ 10

2) int x ; /* ⇒ 5

3) int x ; /* --- */ ⇒ 3

Not token

4) x = = y /*abcd****/ abcd */ ; ⇒ 9

Not token

5) int **p ;

int x = 10 ; y ;

"**" is not any operator in C

* ⇒ multiplication or dereference operator depending on context

(18)

x(13) =(14) *(15) b(16) ++(17) ++(18) +(19) y(20) ;(21) → 21

6) char(1) ch(2) =(3) 'A'(4) ;(5) → 5

7) ch(1) /* comment */ (Not token) ar(2) ch(3) =(4) "gate"(5) ;(6) → 6

8) char(1) ch(2) =(3) "gate ; ⇒ Token error
(double quotes started but not ende

9) ch /* comment ar = "gate" ; ⇒ Token error
(comment started but not ended)

10) char(1) ch(2) = 'a ; ⇒ Token error
(single quote started but not ended)

Q7 ~~what are the~~ Which strings below are said to be tokens in C-language without looking at next i/p character?

1) ; ✓

2) return ✗ bcz variable name may be returna

3) main ✗ bcz variable name may be maina

4) int ✗ bcz variable name may be intabcd

5) >> ✗ bcz it can be >>=

6) (✓

7) && ✓ bcz REMEMBER!!
In C, &= is a operator
but && = is not an operator

NOTE :-

While doing tokenization, lexical analyzer will remove comment lines, blank space, white space character (tabs, newline). So, don't count them as token.

- The set of string described by a rule is called pattern associated with the token.
- A lexeme is a sequence of character in the source program that is matched by pattern for a token.

Parsers

✯ Grammar

$$G(V, T, P, S)$$

where, V : Variable
T : Terminals
P : Productions
S : Start symbol

eg.1 $G(V, T, P, S)$

P : $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$

$V = \{S, A, B\}$
$T = \{a, b\}$
$S = S$

eg.2 $G(V, T, P, S)$

P : $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$

$V = \{S, a, b\}$
$T = \{A, B\}$
$S = A$

- In general, if not given specifically,
  capital letters ⇒ variables
  small letters ⇒ terminals

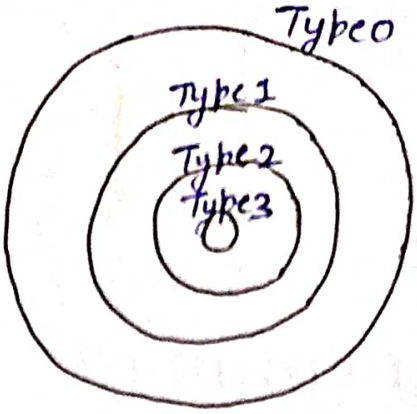
⇒ Types of grammars, according to chomsky :-

1) Type-0 (Unrestricted grammar)
2) Type-1 (context sensitive grammar)
3) Type-2 (Context free grammar)
4) Type-3 (Regular grammar)

Type-0

$\alpha \rightarrow \beta$

where $\alpha, \beta \in (V+T)^*$



Type-1

$\alpha \rightarrow \beta$

where $\alpha, \beta \in (V+T)^*$, $|\alpha| \leq |\beta|$

$T_3 \subseteq T_2 \subseteq T_1 \subseteq T_0$

eg:
$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$
} Type 1 ✓

$S \rightarrow AB$
$ABC \rightarrow DE$
$aB \rightarrow cd$
} Type 1 ✗, Type 0 ✓

Type-2

$A \rightarrow (V+T)^*$

where, A is single variable

eg:
$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$
} Type-2 ✓

$S \rightarrow AB$
$AC \rightarrow CD$
$C \rightarrow EF$
} Type2 ✗, Type 1 ✓

Type-3

$A \rightarrow T^*B \mid T^*$ (Right linear grammar)

where A, B are single variable

(or)

$A \rightarrow BT^* \mid T^*$ (Left linear grammar)

where A, B are single variable

NOTE:-

1) By default, all grammars are Type-0.

⇒ Context free grammar

Q1 Give CFG for $L = \{ a^m b^n \mid m, n \geq 1 \}$

Soln

$S \rightarrow AB$

$A \rightarrow a \mid aA$

$B \rightarrow b \mid bB$

Q2 Give CFG for $L = \{ a^m b^n c^n \mid m, n \geq 1 \}$

Soln

$S \rightarrow AB$

$A \rightarrow a \mid aA$

$B \rightarrow bc \mid bBc$

Q3 Give CFG for $L = \{ a^i b^{1+j} c^{i+j} \mid i, j \geq 1 \}$

Soln

$S \rightarrow AB$

$A \rightarrow ab \mid aAb$

$B \rightarrow bc \mid bBc$

Q4 Give CFG for $L =$ {set of all pallindromes over the alphabet a, b}

Soln

eg. ϵ, a, b, aa, bb, aba, bab, abba, baab ....

$S \rightarrow \epsilon \mid a \mid b \mid aSa \mid bSb$

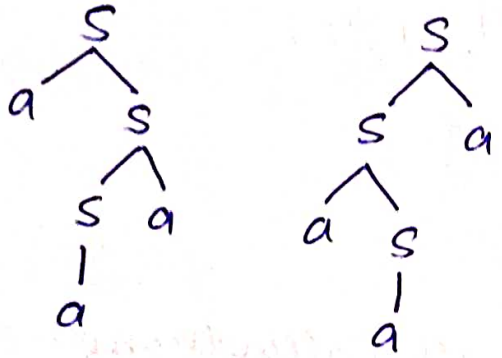
Q1 Consider the following grammar:

$S \rightarrow aS \mid Sa \mid a$

Input string: aaa

How many parse trees are possible to generate above i/p string?

Sol:



So, There are 4 possible parse trees

So, It is an ambiguous grammar

NOTE:-

checking given grammar is ambiguous grammar or not is a undecidable problem.

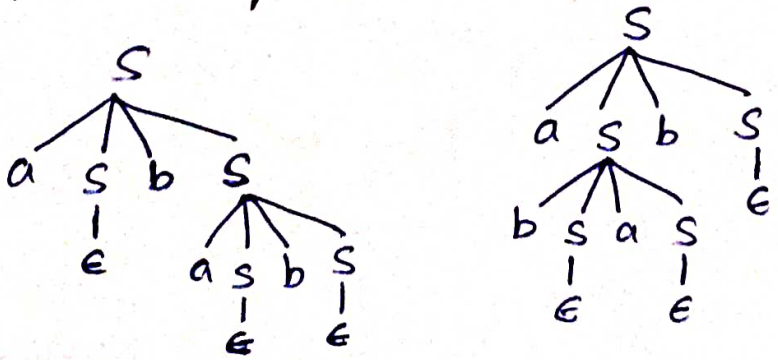
Q2 Consider the following grammar?

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

i/p string: abab

Draw all possible parse trees?

Sol:



2 parse trees are possible

So, It is ambiguous grammar

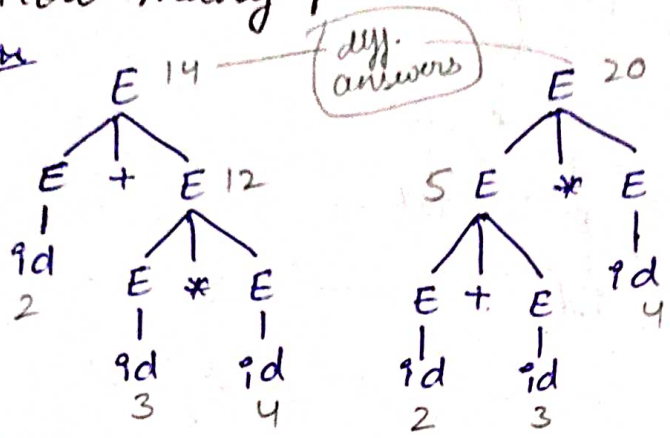
Q3 Consider the following grammar:

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

I/p string : id + id * id 2 + 3 * 4

How many parse trees are possible?

Soln



2 parse trees are possible

so, it is ambiguous grammar

NOTE:-

More than 1 - parse tree means more than 1 answer.

This is the drawback of ambiguous grammar.

⇒ Conversion from ambiguous grammar to equivalent unambiguous grammar

eg.1 $E \rightarrow E+E \mid E*E \mid id$

⇓ Equivalent unambiguous grammar (According to C language)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

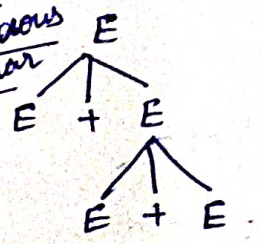
⇓

* → high priority (left to right associativity)

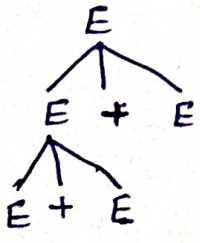
+ → low priority (left to right associativity)

string : id + id + id

using ambiguous grammar

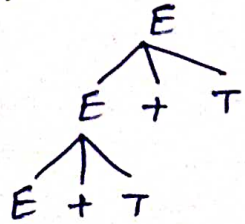


Right to left associativity



left to right associtivity

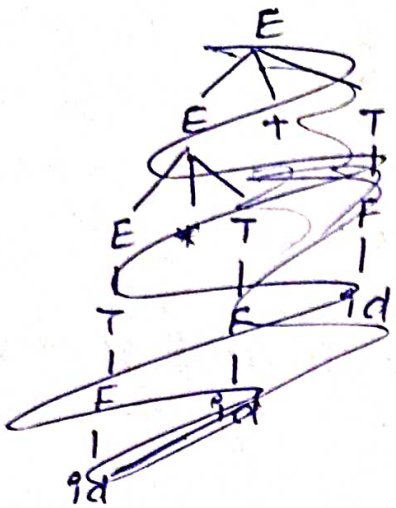
using unambiguous grammar

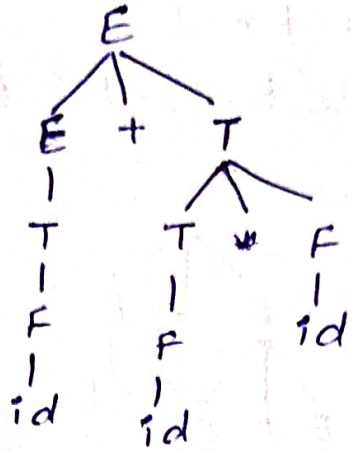


left to right associativity only

String : id + id * id

using unambiguous grammar





eg.2 $E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid (E) \mid id$

Equivalent unambiguous grammar (Acc. to C language)

Here, priority: ( ) > *, / > +, -

comparing operators, operand has more priority

Here, all above operators has L-R ~~priority~~ associativity.

$A \rightarrow A+B \mid A-B \mid B$

$B \rightarrow B*C \mid B/C \mid C$

$C \rightarrow (A) \mid id$ → bcz anything can come inside bracket

(OR)

$A \rightarrow A+B \mid A-B \mid B$

$B \rightarrow B*C \mid B/C \mid C$

$C \rightarrow (A) \mid D$

$D \rightarrow id$

Eg:3

Q3. Write equivalent unambiguous grammar for the following rules :

		Priority	Associativity
↑ , #	—	(3)	— (L-R)
⊕ , *	—	(2)	— (R-L)
- , =	—	(5) Highest	— (L-R)
/ , &	—	(1) Lowest	— (R-L)
+ , $	—	(4)	— (L-R)

Sol:

$A \rightarrow B/A \mid B \& A \mid B$

$B \rightarrow C \oplus B \mid C * B \mid C$

$C \rightarrow C \uparrow D \mid C \# D \mid D$

$D \rightarrow D + E \mid D \$ E \mid E$

$E \rightarrow E - F \mid E = F \mid F$

$F \rightarrow (A) \mid id$

eg.4 $S \rightarrow aS \mid Sa \mid a$

⇓ Equivalent unambiguous grammar

$S \rightarrow aS \mid a$

NOTE :-

There is no algo to convert given ambiguous grammar into equivalent unambiguous grammar. It is undecidable problem.

⇒ Recursion

left recursion

$S \rightarrow S\alpha \mid \beta$

where,

$\alpha \in (V+T)^*$,

$\beta \in T^*$

right recursion

$S \rightarrow \alpha S \mid \beta$

where,

$\alpha \in (V+T)^*$,

$\beta \in T^*$

- Because of the left recursion, one of the top-down parser will go to infinite loop so, we have to eliminate left recursion.
- Because of right recursion, no problem so, no need to eliminate right recursion.

⇒ Elimination of left recursion

eg.1 $E \rightarrow E+T \mid T$ $(T, T+T, T+T+T, T+T+T+T, \ldots) \Rightarrow T(+T)^*$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

⇓

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id$

eg.2

$S \rightarrow Sa|b|c|d$

$\Downarrow$

$S \rightarrow bS'|cS'|dS'$

$S \rightarrow \epsilon|aS'$

eg.3 $S \rightarrow Sa|Sb|Sc|d|e|f$

$\Downarrow$

$S \rightarrow dS'|eS'|fS'$

$S' \rightarrow \epsilon|aS'|bS'|cS'$

eg.4 $S \rightarrow Sa|Sb|Ac|d|e$

$A \rightarrow f|g|Sh$

Here, $S \rightarrow Ac$ is also a left recursion (indirect) bcz $A \rightarrow Sh$

$\Downarrow$ Substitute 'A'

$S \rightarrow Sa|Sb|d|e|fc|gc|Shc$

$\Downarrow$

$S \rightarrow dS'|eS'|fcS'|gcS'$

$S' \rightarrow \epsilon|aS'|bS'|hcS'$

- While elimination of left recursion, you have to eliminate both direct & indirect left recursion.

(30) Lect-6

⇒ Left factoring

eg.1 $S \rightarrow ab|ac|ad|b$

i/p string : ad

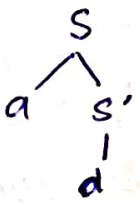
From the given grammar, to generate i/p string "ab" 1st - 3 productions are fighting bcz in i/p string "ab", 1st character i.e, 'a' is given by all 1st - 3 productions, this problem is known as left factoring.

⇓ Elimination of left Factoring (or) Left factored grammar

$S \rightarrow aS'|b$

$S' \rightarrow b|c|d$

i/p string : ad



eg.2 $S \rightarrow iEtS|iEtSeS|a|b$

$E \rightarrow d$

⇓ Elimination of left factoring

$S \rightarrow iEtSS'|a|b$

$S' \rightarrow \epsilon|eS$

$E \rightarrow d$

eg.3 $S \rightarrow a|ab|abc|abcd|e|f$

⇓ Eliminate left factoring

$S \rightarrow e|f|aS'$

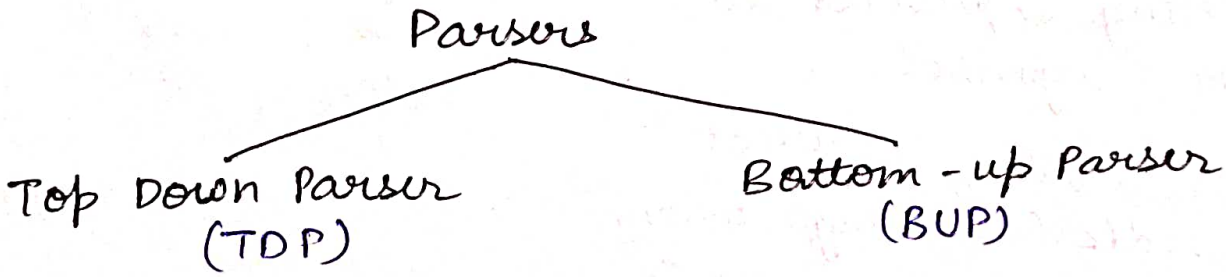
$S' \rightarrow \epsilon|bS''$

$S'' \rightarrow e|fS'''$

$S''' \rightarrow \epsilon|d$

☆ Parsers

- The process of deriving the string from the given grammar is known as derivation (or) Parsing
- Parsing is performed by Parsers.
- Depending upon how derivation is performed, parsers are of 2 types :-



⇒ How TDP works ?

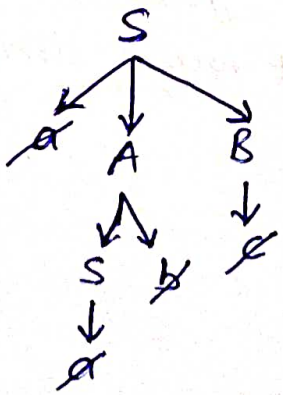
eg. $S \rightarrow aAB|a$

$A \rightarrow Sb$

$B \rightarrow c$

I/p string : $\underline{a}dd\$$

↑ lookahead symbol

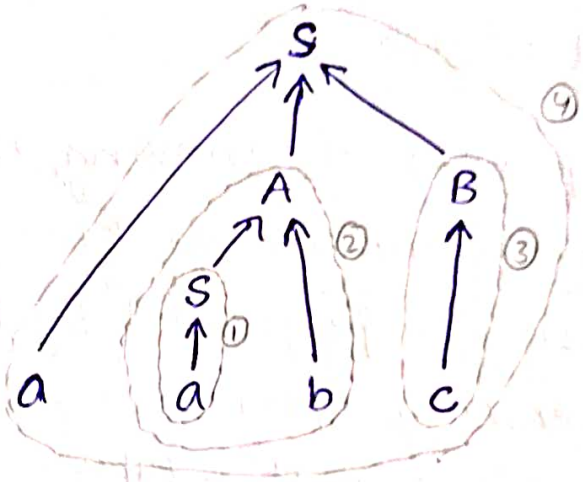


- TDP will start from start symbol & proceed to string.
- If variable contains more than one production then, selecting correct production is difficult in TDP.
- (Drawback of TDP)
- TDP follows left most derivation.

⇒ How BUP works ?

eg. S → aABb | a
A → Sb
B → c

i/p string : aabc



If we see these in reverse order, it looks like Bottom up derivation

In normal order, NO derivation bcz only terminals are there in starting, no variables are there to choose from.

- BUP will start from string & proceed to start symbol.
- Identifying correct handle (substring) whom we have to compress is very difficult in BUP.
- Bottom up parser follows reverse of right most derivation.

✯ Top down parser

- with backtracking
  - ⇓
  - Recursive descent parser
- without backtracking
  - ⇓
  - Non recursive descent parser
  - (or)
  - Imp LL(1) parser
  - (or)
  - Predictive parser

⇒ Recursive descent parser (RDP)

```
RDP(S)
{
  Select any production of S (S → x1 x2 x3 --- xk)
  for(i = 1; i ≤ k; i++)
  {
    if (xi is variable)
        RDP(xi)
    else if (xi == LAS)        → look ahead symbol
        increement i/p pointer
    else
        parsing-error (back-track)
  }
}
```

eg.1

$S \to ABC \mid DEF \mid GHI$

$A \to ab$ $\quad D \to d$ $\quad G \to g$

$B \to cd$ $\quad E \to e$ $\quad H \to h$

$C \to ef \mid gh \mid ij$ $\quad F \to f$ $\quad I \to i$

i/p string : abcdij

RDP(S)

$S \to A B C$ (x1 x2 x3)

i=1 i=2 i=3

A is terminal — B — C

RDP(A)

$A \to a b$ (x1 x2)

i=1 i=2

a ✓ b ✓

RDP(B)

$B \to cd$ (x1 x2)

i=1 i=2

c ✓ d ✓

RDP(C)

$C \to ef$ (x1 x2) $\quad C \to gh$ (x1 x2) $\quad C \to ij$ (x1 x2)

i=1 i=2 $\quad$ i=1 i=2 $\quad$ i=1 i=2

e $\quad$ g $\quad$ i ✓ j ✓

parsing-error so, backtrack $\quad$ parsing error so, backtrack

Lect-7

eg.2

$S \to aS \mid b$

i/p : aaab

RDP(S)

$S \to aS$ (x1 x2)

i=1 i=2

a ✓ S

RDP(S)

$S \to aS$ (x1 x2)

i=1 i=2

a ✓ S

RPP(S)

$S \to aS$ (x1 x2)

i=1 i=2

a

RDP(S)

$S \to aS$ (x1 x2) $\quad S \to b$

i=1 i=2 $\quad$ i=1

a $\quad$ b ✓

(Backtrack)

eg.3 $S \to Sa \mid b$

i/p : baaa

RDP(S)

$S \to Sa$ ($x_1\ x_2$)

$i=1$ $i=2$

S

RDP(S)

$S \to Sa$

$i=1$ $i=2$

S

RDP(S)

Stack overflow error (like infinite loop)

* bcz of left recursion.

eg.4

$S \to ABC \mid DEF \mid GHI$

$A \to a \mid ab \mid abc \mid abcd$

$B \to ef \mid gh \mid ij$

$C \to kl \mid mn \mid op$

$D \to d$ $G \to g$

$E \to e$ $H \to h$

$F \to f$ $I \to i$

i/p: abcdefmn

abcdefmn

RDP(S)

$S \to ABC$ ($x_1\ x_2\ x_3$)

$i=1$ $i=2$ $i=3$

A B (Backtrack) $\Rightarrow$ $S \to DEF$ ($x_1\ x_2\ x_3$) (Backtrack) $S \to GHI$ ($x_1\ x_2\ x_3$)

RDP(D)

$D \to d$ (Backtrack)

RDP(G)

$G \to g$ (Backtrack)

RDP(A)

$A \to a$ (x_1)

$i=1$

a

RDP(B)

$B \to ef$ ($x_1\ x_2$) $B \to gh$ ($x_1\ x_2$) $B \to ij$ ($x_1\ x_2$)

$i=1$ $i=2$ $i=1$ $i=2$ $i=1$ $i=2$

e (Backtrack) g (Backtrack) i (Backtrack)

Parsing error

Here, i/p string can be derived by above grammar but RDP could not do it bcz left factoring problem is there.

NOTE :-

1) In RDP, we will write recursive program for every variable. (For every variable, same recursive program is called)
2) In RDP, lot of time gets wasted in the form of backtracking.
3) If grammar contains left recursion, RDP will go to infinite loop.
4) If grammar contains left factoring, RDP may generate parsing error even if that grammar can generate the i/p string.

V. V. Imp

⇒ LL(1) - parser (or) Non recursive descent parser (or) Predictive parser

NOTE :-

1) There is no backtracking in LL(1) parser bcz it uses LL(1) parsing table.
2) To construct LL(1) parsing table, we need two functions :-
   i) First()
   ii) Follow()

- **First()**

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

eg.1

$S \rightarrow abc|def|ghi|\epsilon$

First(S) = a, d, g, ϵ = First(abc), First(def), First(ghi), First(ϵ)

= First(a), First(d), First(g), First(ϵ)

= a, d, g, ϵ

NOTE :-

First(ϵ) = ϵ

First(a) = a

where a : terminal

eg.2 $S \rightarrow ABC|DEF|GHI$

$A \rightarrow a|j|k|\epsilon$

$B \rightarrow b|l|m|\epsilon$

$C \rightarrow c|n|o|\epsilon$

$D \rightarrow d$

$E \rightarrow e$

$F \rightarrow f$

$G \rightarrow g$

$H \rightarrow h$

$I \rightarrow i$

First(S) = First(ABC), First(DEF), First(GHI)

= First(A), First(D), First(G)

= First(a), First(j), First(k), First(BC), First(d), First(g)

~~= a d g~~

= a, j, k, First(B), d, g.

= a, j, k, First(b), d, g, First(l), First(m), First(ϵ)

= a, j, k, b, d, g, l, m, First(c), First(n), First(o), First(ϵ)

= a, j, k, b, d, g, l, m, c, n, o, ϵ

eg.3 $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

Find, First() for all variables?

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

First(E) = First(TE')
= First(T)
= First(FT')
= First(F)

eg.4 $S \rightarrow (L) \mid a$

$L \rightarrow SL'$

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

	first()
S	(, a
L	(, a
L'	ϵ, ,

first(L) = first(SL')
= first(S)

eg.5 $S \rightarrow aBDh$

$B \rightarrow bC \mid \epsilon$

$C \rightarrow cC \mid \epsilon$

$D \rightarrow EF$

$E \rightarrow f \mid \epsilon$

$F \rightarrow g \mid \epsilon$

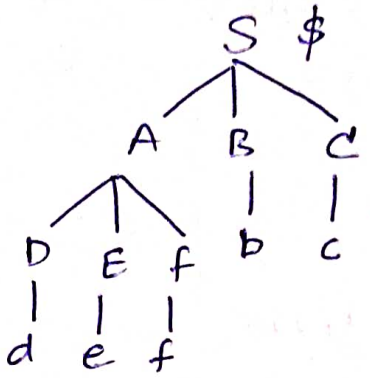
	first()
S	a
B	b, ϵ
C	c, ϵ
D	f, g, ϵ
E	f, ϵ
F	g, ϵ

- **Follow ( )**

Follow(A) contains set of all terminals present in 1st place of immediately right of A.

eg.1

S → ABC
A → DEF
B → b
C → c
D → d
E → e
F → f



I/p : defbc $

↓
'$ means string is over

Follow(D) = First(EF) = First(E) = e

Follow(F) = Follow(A) = First(BC) = First(B) = b

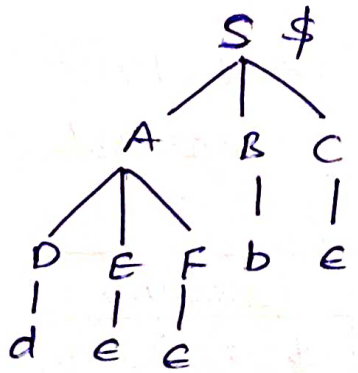
Follow(C) = Follow(S) = First($) = $

eg.2

S → ABC
A → DEF
B → b E → ∈
C → ∈ F → ∈
D → d

~~I/p : defbc $~~

I/p : db$



Follow(A) = First(BC) = First(B) = b

Follow(B) = ~~First(C) = ∈~~ = Follow(S) = ~~First($) = $~~

~~Follow(D) = First(EF) = First(E) = ∈ = First(F) = ∈~~

= Follow(A) = First(BC) = First(B) = b

NOTE :-

- Follow of any variable can never be ϵ.

→ Rules to find follow ( )

Assume A is a variable

1) If A is start symbol
then, follow (A) = $

2) B → CADE
D → d
then follow (A) = first (DE)
= first (D)
= first (d) = d.

3) B → CDA
(or)
B → CDAE
E → ϵ
then, follow (A) = follow (B)

for all variables including start symbol

eg.3 S → (L) | a
L → SL'
L' → ϵ | , SL'

Find first & follow of every variable?

	first()	follow()
S	(, a	$, , ,)
L	(, a	)
L'	ϵ, ,	)

eg.4

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

	first()	follow()
E	id, (	), $
E'	ϵ, +	$,)
T	id, (	+,), $
T'	ϵ, *	+,), $
F	id, (	*, +,), $

eg.5 Lect-8

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

Find first & follow for all variables?

	first()	follow()
S	a, b, ϵ	$, a, b

eg.6 Find first & follow for all variables?

$S \rightarrow a\ B\ D\ h$

$B \rightarrow a\ C \mid \epsilon$

$C \rightarrow c\ C \mid \epsilon$

$D \rightarrow EF$

$E \rightarrow f \mid \epsilon$

$F \rightarrow g \mid \epsilon$

	first()	follow()
S	a	$
B	a, ϵ	f, g, h
C	c, ϵ	f, g, h
D	f, g, ϵ	h
E	f, ϵ	g, h
F	g, ϵ	h

- ϵ is neither variable nor terminals.
- look ahead symbols are the symbols present in the input string.
  $ is the special look ahead symbol.
- In parsing table,
  No. of rows = No. of variables
  No. of cols = No. of terminals + 1 ($)

• LL(1) parsing table construction

Eg:1 Construct LL(1) parsing table for the following grammar:

$S \xrightarrow{①} (L) \mid a \xrightarrow{②}$

$L \xrightarrow{③} SL'$

$L' \xrightarrow{④} \epsilon \mid ,\xrightarrow{⑤} SL'$

	a	,	(	)	$
S	S→a		S→(L)		
L	L→SL'		L→SL'		
L'		L'→,SL'		L'→ϵ	

LL(1) PT

NOTE :-
In LL(1) parsing table, blank entry means parsing error
ie, given string cannot be generated by that grammar.

① $S \to (L)$

first((L)) = (

② $S \to a$

first(a) = a

③ $L \to SL'$

first(SL') = first(S) = (, a

⑤ $L' \to ,SL'$

first(,SL') = first(,) = ,

④ $L' \to \epsilon$

~~first(ϵ) = ϵ~~

follow(L') =)

NOTE:- m → LL(1) parsing table

For each production $A \to \alpha$

1) Add $A \to \alpha$ under $m[A, b]$
   $\forall b \in First(\alpha)$
2) If $First(\alpha)$ contains ϵ then,
   add $A \to \alpha$ under $m[A, c]$
   $\forall c \in follow(A)$

eg.2 construct LL(1) parsing table for:

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

follow(E') = $,)
follow(T') = +, $,)

LL(1) - PT

	id	(	)	+	*	$
E	E→TE'	E→TE'				
E'			E'→ϵ	E'→+TE'		E'→ϵ
T	T→FT'	T→FT'				
T'			T'→ϵ	T'→ϵ	T'→*FT'	T'→ϵ
F	F→id	F→(E)				

NOTE:-

- In LL(1) parsing table, if each entry contains maximum 1 production then, given grammar is LL(1) grammar.
- To verify given grammar(G) is LL(1) or not:
  (i) Draw LL(1) parsing table
  (ii) If each entry contains max 1 production then, G is LL(1) else G is not LL(1).

eg.3

Check following grammar is LL(1) or not:

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

follow(S) = $, a, b

LL(1) PT

	a	b	$
S	S→aSbS, S→ϵ	S→bSaS, S→ϵ	S→ϵ

In LL(1) PT, in some entries there are multiple productions so, given grammar is not LL(1).

Q4 Check given grammar is LL(1) or not?

$S \rightarrow \overset{①}{A}a\overset{①}{A}b \mid \overset{②}{B}b\overset{②}{B}a$

$A \rightarrow \overset{③}{\epsilon}$

$B \rightarrow \overset{④}{\epsilon}$

[To save time in exam, don't check for all variables, only check for variable which has multiple production bcz only that may make trouble.]

Soln

	a	b	$
S	S→AaAb	S→BbBa	

⇒ not complete parsing table

so, it is LL(1) grammar.

① $S \rightarrow AaAb$

first(AaAb) = ~~first(A) = ϵ~~

= first(aAb) = a

② $S \rightarrow BbBa$

first(BbBa) = ~~first(B) = ϵ~~

= first(bBa) = b

Q5 Check whether following grammar is LL(1) or not?

$S \to aBDh$
$B \to bC \mid \epsilon$
$C \to cC \mid \epsilon$
$D \to EF$
$E \to f \mid \epsilon$
$F \to g \mid \epsilon$

follow(B) = f, g, h
follow(C) = f, g, h
follow(E) = g, h
follow(F) = h

Sol: check for B, C, E, F only bcz only they have multiple productions

	a	b	c	f	g	h	$
B		②		③	③	③	
C			④	⑤	⑤	⑤	
E				⑦	⑧	⑧	
F					⑨	⑩	

⇒ not complete parsing table

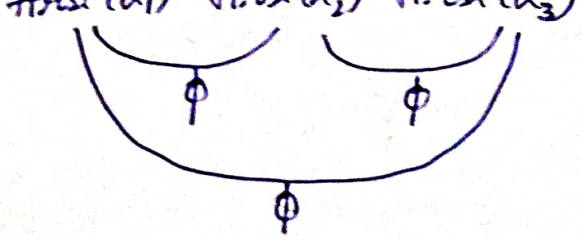
So, it is LL(1) grammar

Shortcut :-

1) A grammar G without ϵ-productions is said to be LL(1) iff

$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3$

first(α_1) first(α_2) first(α_3)



It should be pairwise disjoint

i.e., first(α_1) ∩ first(α_2) = ϕ
first(α_2) ∩ first(α_3) = ϕ
first(α_1) ∩ first(α_3) = ϕ

2) A grammar G with ε-production is LL(1) iff

$S \to \alpha_1 \mid \alpha_2 \mid \epsilon$

first(α_1) first(α_2) follow(S)

ϕ ϕ

ϕ

They should be pairwise disjoint

i.e, first(α_1) $\cap$ first(α_2) $= \phi$
first(α_2) $\cap$ follow(S) $= \phi$
first(α_1) $\cap$ follow(S) $= \phi$

Q6 Check given grammar is LL(1) or not?

$S \to E \mid a$
$E \to a$

Soln We need to check S only

$S \to E \mid a$

first(E) $= a$ — a

$a \cap a = a \neq \phi$

So, it is not LL(1)

Q7 LL(1)?

$S \to aSbS \mid bSaS \mid \epsilon$

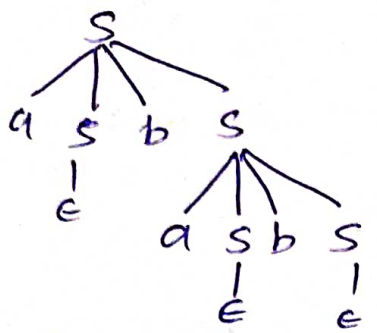
first(aSbS) $= a$ — ϕ

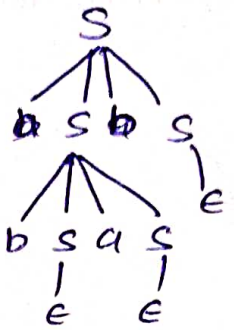
first(bSaS) $= b$ — b

follow(S) $= a, b, \$$ — a

So, it is not LL(1)

i/p : abab$





Ambiguous grammar

NOTE :-

1) Ambiguous grammar cannot be LL(1).

~~but LL(1) grammar can be ambiguous~~

2) Unambiguous grammar may or may not be LL(1).

Q8 LL(1) ?

$S \rightarrow ab \mid ac$ ⇒ contains left factoring

first(ab) = a

first(ac) = a

$\cap$

a

so, it is not LL(1)

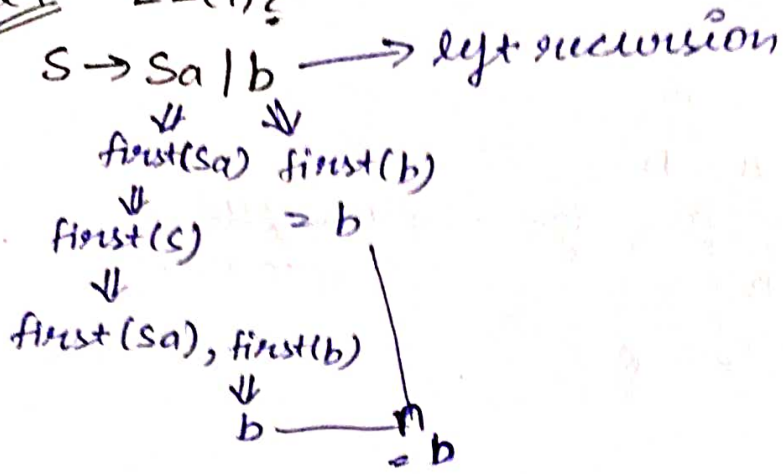
NOTE :-

1) If grammar contains left factoring then, it is not LL(1).

2) If grammar does not contain left factoring then, it may or may not be LL(1).

Q9 LL(1)?

$S \rightarrow Sa \mid b$ ⟶ left recursion



So, it is not LL(1)

NOTE :-

1) If grammar contain left recursion then, it can never be LL(1)
2) If grammar does not contain left recursion then, it may or may not be LL(1).
3) Every regular grammar need not be LL(1) bcz it may contain left factoring (or) left recursion (or) ambiguity.

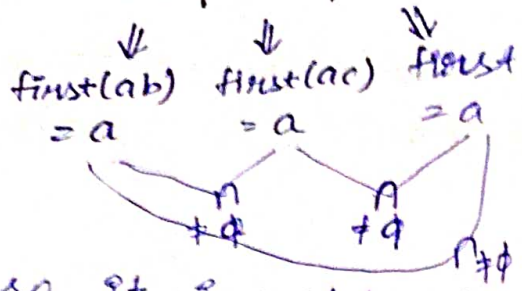
eg $S \rightarrow Sa \mid b$ is a regular grammar but not LL(1) bcz it contains left recursion.

$S \rightarrow aS \mid a$ is a regular grammar but not LL(1) bcz it contains left factoring.

$S \rightarrow Aa \mid a$ is a regular grammar but not LL(1)
$A \rightarrow a \mid \epsilon$ bcz it ~~contains~~ is ambiguous.

Q10 LL(1) ?

$S \rightarrow ab \mid ac \mid ad$



so, it is not LL(1)

but it is LL(2)

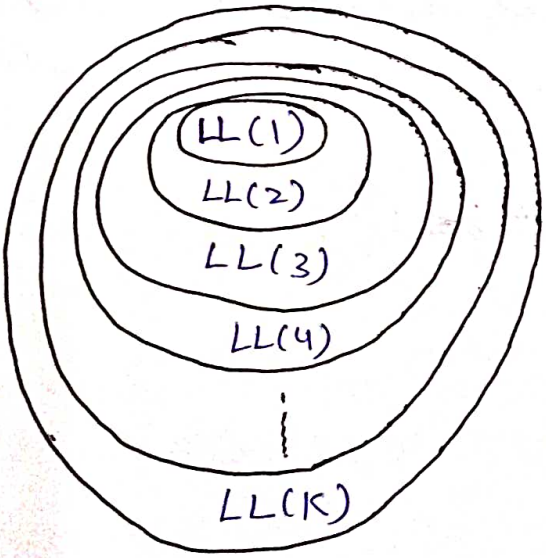
(read 2-2 symbols of each & they are pairwise disjoint)

$S \rightarrow aba \mid abb \mid abc$

LL(1) ✗

LL(2) ✗

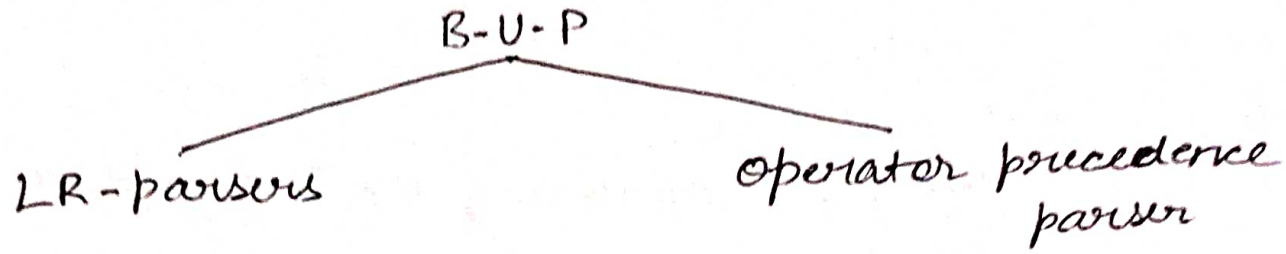
LL(3) ✓



★ Bottom up parser (B-U-P)

NOTE :-

All bottom up parsers are without backtracking bcz every bottom up parser contains parsing table.



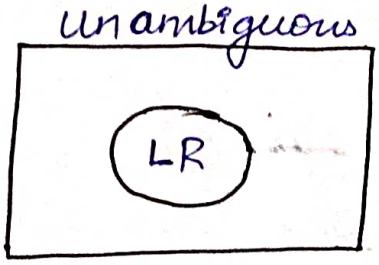
⇒ LR Parser (Shift-reduce parser)

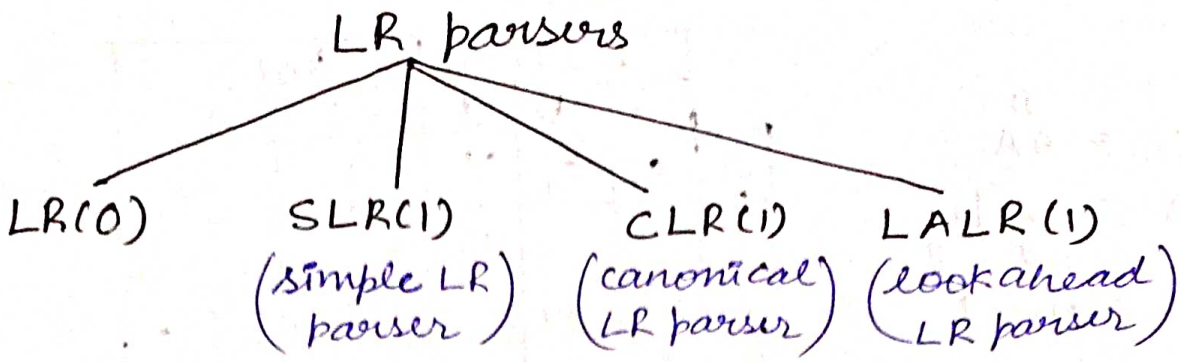
NOTE :-

1) LR parsers are applicable only to unambiguous grammar

ie, Ambiguous grammar ⇒ Not LR.
Unamabiguous grammar ⇒ may or may not be LR.

2) Every LR is unambiguous but
Every unambiguous need not be LR.





NOTE:-

1) All LR-parsers contains different parsing tables.
2) All LR-parsers contains same parsing algorithm.

→ **LR parsing algorithm:**

Let 'X' be the state no. on the top of the stack and 'a' will be the lookahead symbol.

1) If action[X,a] = S_i then Shift a && i and (Push) increcement i/p pointer.
2) If action[X,a] = r_j and j^{th} production is $\alpha \to \beta$ then pop $2*|\beta|$ symbols and replace by α.
   If X_{m-1} is the state below α then push Goto $[X_{m-1}, \alpha]$.
3) If action[X,a] = acc then, successful completion of parsing.
4) If action[X,a] = blank then, parsing error.

eg.1

$S \xrightarrow{①} AA$

$A \rightarrow aA \mid b$
(②) (③)

i/p : aabb$

Generate above string?

	Action			Goto	
	a	b	$	S	A
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

NOTE :-

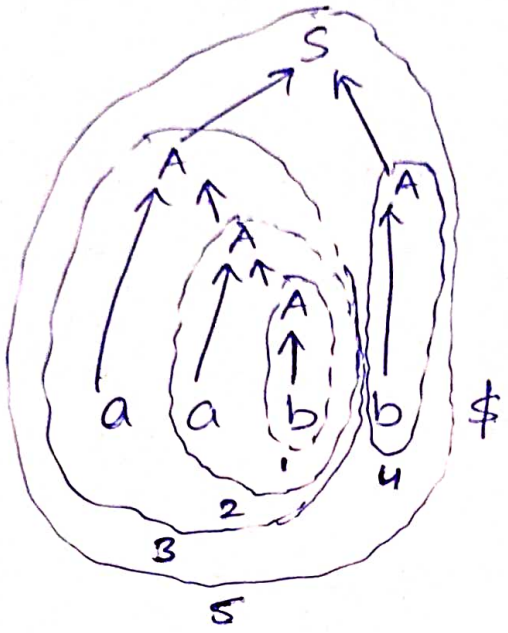
1) No. of rows in LR(0) parsing table = No. of states in the DFA of LR(0) parser.

2) No. of cols in LR(0) parsing table = No. of terminals + 1 (for $) + No. of variables
(No. of terminals + 1 (for $) : Action; No. of variables : Goto)

Sol^n i/p	Stack	Reductions
a a b b $	0 ⇒ 0a3	
a b b $	0a3 ⇒ 0a3a3	
b b $	0a3a3 ⇒ 0a3a3b4	
b $	0a3a3b4 ⇒ 0a3a3A6	1. A → b
b $	0a3a3A6 ⇒ 0a3A6	2. A → aA
b $	0a3A6 ⇒ 0A2	3. A → aA
b $	0A2 ⇒ 0A2b4	
$	0A2b4 ⇒ 0A2A5	4. A → b
$	0A2A5 ⇒ 0S1	5. S → AA

Rightmost derivation in reverse.

Now Action[1,$] = acc (so, successful completion of parsing)

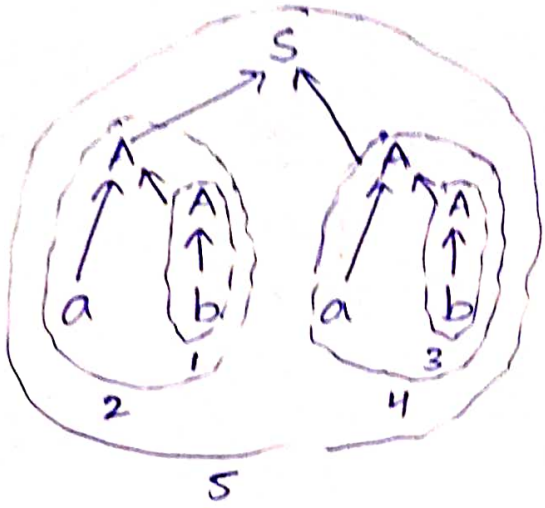


Eg.2 same grammar as previous

I/P : abab $

I/P	Stack	Reductions
↑ abab$	0 ⇒ 0a3	
~~↑~~ ab$	0a3 ⇒ 0a3b4	
a ↑ b$	0a3b4 ⇒ 0a3A6	1. A → b
a ↑ b$	0a3A6 ⇒ 0A2	2. A → aA
~~↑~~ b$	0A2 ⇒ 0A2a3	
~~↑~~ $	0A2a3 ⇒ 0A2a3b4	
$ ↑	0A2a3b4 ⇒ 0A2a3A6	3. A → b
$ ↑	0A2a3A6 ⇒ 0A2A5	4. A → aA
↑ $	0A2A5 ⇒ 0S1	5. S → AA

Now, Action [1, $] = acc so, successful completion of parsing.



- **Viable prefixes :-**
  While performing bottom up parsing, all prefixes present in stack are known as viable prefixes.

eg. 

viable prefixes :- a, ab, abc

eg.3 same grammar as previous

i/p : aabbb$

i/p	Stack	Reductions
a a b b b $	0 ⇒ 0 a 3	
a b b b $	0 a 3 ⇒ 0 a 3 a 3	
b b b $	0 a 3 a 3 ⇒ 0 a 3 a 3 b 4	
b b $	0 a 3 a 3 b 4 ⇒ 0 a 3 a 3 A 6	1. A → b
b b $	0 a 3 a 3 A 6 ⇒ 0 a 3 A 6	2. A → a A
b b $	0 a 3 A 6 ⇒ 0 A 2	3. A → a A
b b $	0 A 2 ⇒ 0 A 2 b 4	
b $	0 A 2 b 4 ⇒ 0 A 2 A 5	4. A → b
b $	0 A 2 A 5 ⇒ 0 S 1	5. S → A A

Action [1, b] = blank ⇒ parsing error

ie, This string cannot be generated by this grammar

→ LR(0) parsing table construction

To create LR(0) parsing table for the given grammar, we need two functions:

1) closure()
2) goto()

eg.1 G: S → AA
A → aA | b

G': { S' → S } Augmented production
S → AA
A → aA | b

Augmented grammar

S → .S ⇒ Augmented LR(0) item.

S → .xyz
S → x.yz
S → xy.z
S → xyz.

} LR(0) items

S → xyz. : completed LR(0) item (or) Final LR(0) item (or) Reduced LR(0) item

→ indicates production completion status

- **closure()**

eg.1 continue (grammar given above)

eg.1 closure(S → .AA)

⇓

① S → .AA
② A → .aA
.b

eg.2 closure(S → A.A)

⇓

① S → A.A
② A → .aA
.b

eg.3 closure(A → .aA)

⇓

① A → .aA

definition:

closure (I)

⇓

① add I

② If A → B·CDE is I and C → EFG is in G then, add C → .EFG to closure(I).

③ Continue 2nd step for every newly added LR(0) item.

- **goto()**

eg.1

eg.1 Goto(S → .AA, A) ← whatever is there immediately after ·
(only one symbol at a time)

⇓

① S → A·A

② A → .aA
.b

eg.2 Goto(A → .aA, a)

⇓

① A → a·A

② A → .aA
.b

eg.3 Goto(A → a·A, A)

⇓

① A → aA·

definition:

Goto(I, x) ← symbol immediately after · in I

⇓

① Add I by moving '·' after x.

② Find closure (1st step).

LR(0) parsing table construction steps :

1) I_0 = closure (Augmented LR(0) item)
2) using I_0, construct DFA.
3) convert DFA into LR(0) parsing table.

Q1 Construct LR(0) parsing table for the following grammar:

G: $S \xrightarrow{①} AA$
$A \xrightarrow{②} aA \mid \xrightarrow{③} b$

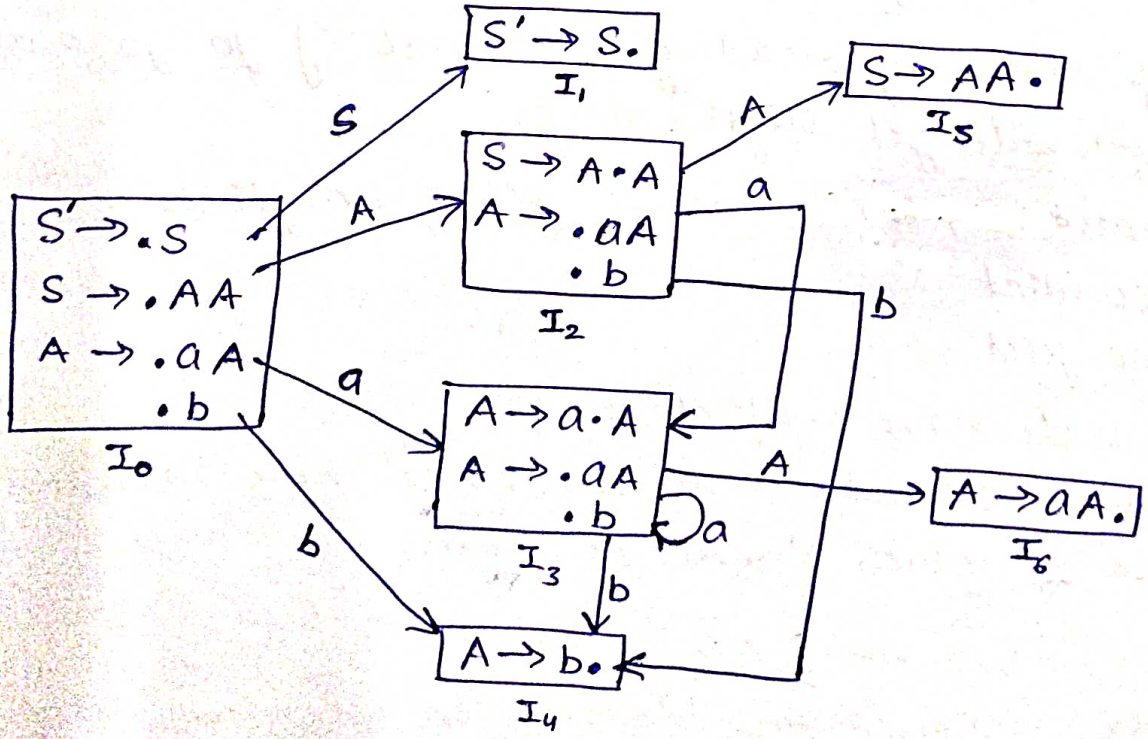
Sol

G' : $S' \to S$
$S \to AA$
$A \to aA \mid b$
} Augmented grammar

Closure $(S' \to .S)$ → Augmented LR(0) item

⇓

① $S' \to .S$
② $S \to .AA$
③ $A \to .aA \mid .b$
} I_0



LR(0) parsing table

	action			goto	
	a	b	$	S	A
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		

There is no conflict in LR(0) parsing table
so, Given grammar is LR(0)
so, Given grammar is SLR(1)

NOTE :-

- All shift entries will come bcz of non final LR(0) items but all non final LR(0) items cannot produce shift. (bcz for some non-final LR(0) item, entry will come in goto() part of table).
- On any state, by reading terminal if yo go outside then, you will get 'shift' entries.
- In LR parsers, there are 2 types of conflicts
  1) SR conflict
  2) RR conflict
- If any state have conflict then,
  1) minimum 2 productions should be there in that state
  2) Atleast 1 reduce should be there.

  ↳ This is $\Rightarrow$ not $\Leftrightarrow$

  i.e., these 2 conditions doesn't ensure that some conflict is there.

Q2 Check the following grammar is LR(0), SLR(1)?

G S → d ① A | a ② B

A → b ③ A | c ④

B → b ⑤ B | c ⑥

Sol:

Augmented grammar:

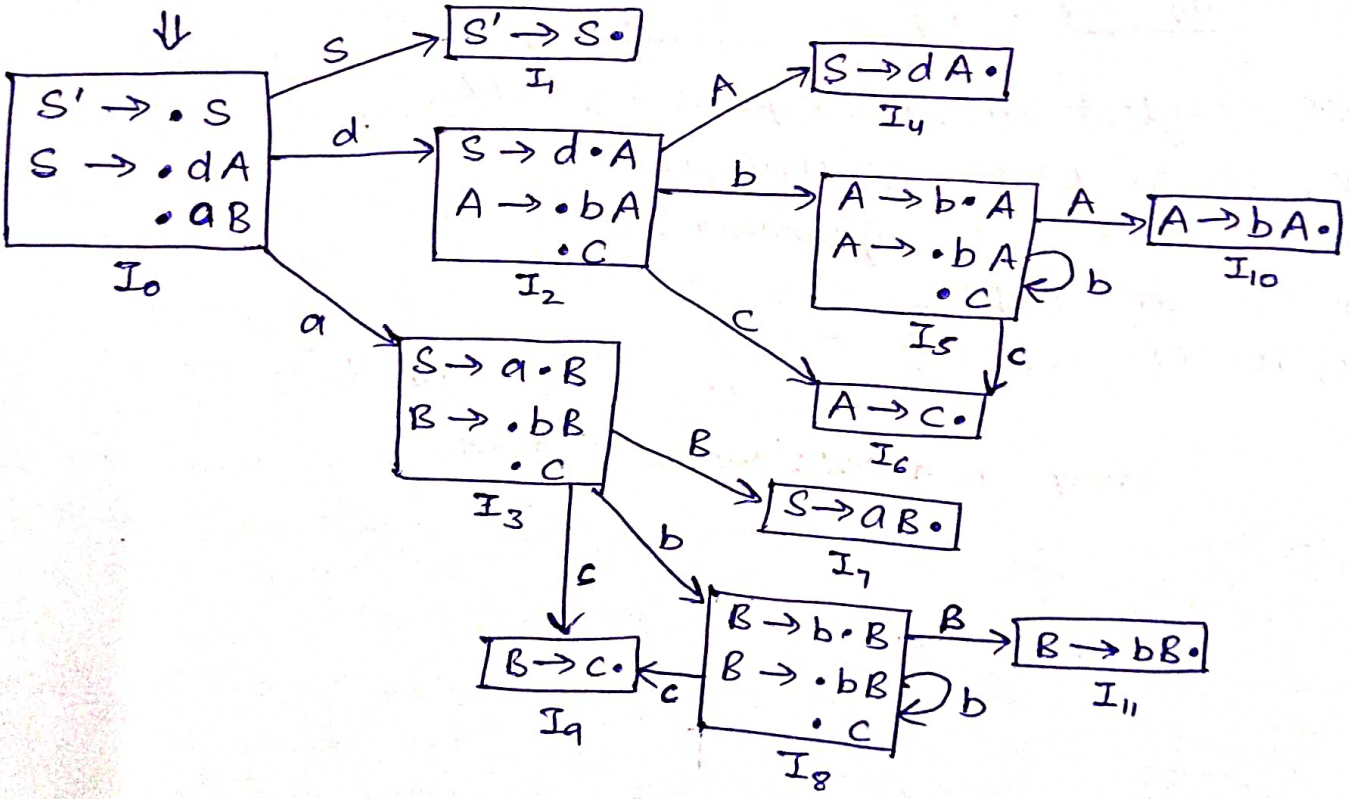
G': S' → S

S → dA | aB

A → bA | c

B → bB | c

closure (S' → .S)



DFA

LR(0) parsing table

	Action					Goto		
	a	b	c	d	$	S	A	B
0	S_3			S_2		1		
1					acc			
2		S_5	S_6				4	
3		S_8	S_9					7
4	r_1	r_1	r_1	r_1	r_1			
5		S_5	S_6				10	
6	r_4	r_4	r_4	r_4	r_4			
7	r_2	r_2	r_2	r_2	r_2			
8		S_8	S_9					11
9	r_6	r_6	r_6	r_6	r_6			
10	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5			

NO conflict in LR(0) parsing table
so, it is LR(0) grammar
so, it is SLR(1) grammar also
so, it is LALR(1) " "
so, it is CLR(1) " "

Lect-11

Q3 Check whether the following grammar is LR(0), SLR(1) & , LALR(1), CLR(1) &, LL(1)?

G: $S \to (L) \mid a$

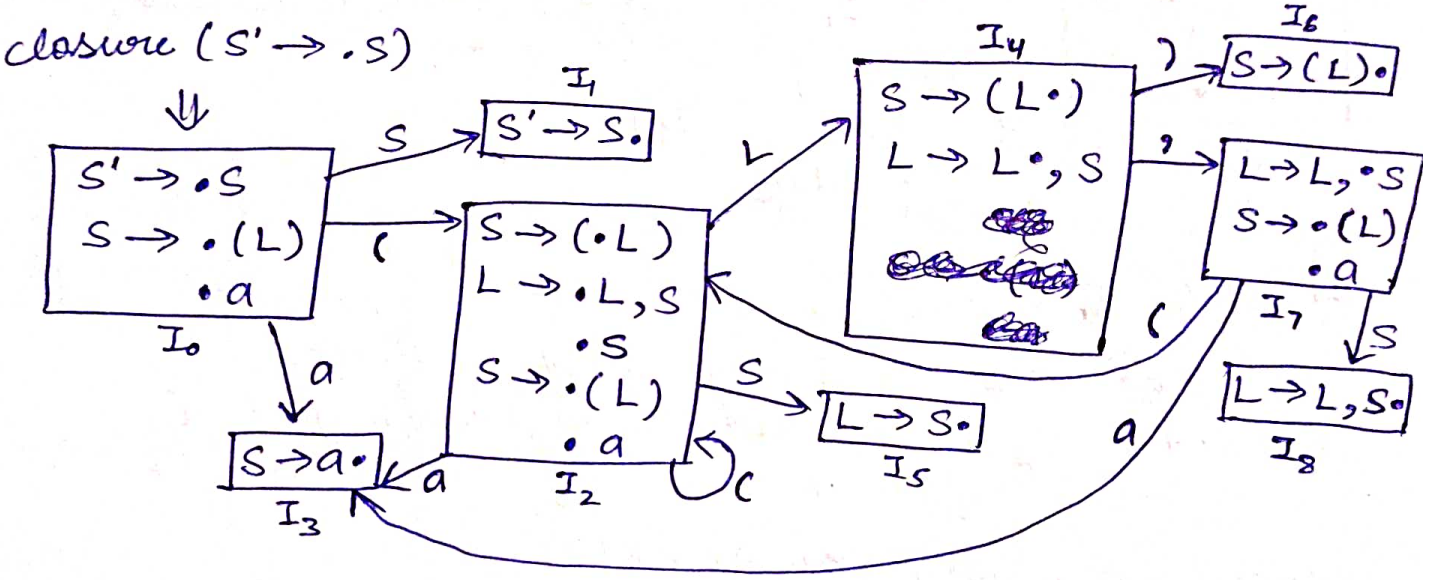
$L \to L, S \mid S$

Soln There is left recursion in the above grammar so, it is not LL(1) grammar.

G': $S' \to S$

$S \to (L) \mid a$

$L \to L, S \mid S$



Note that any of the state cannot have conflict so, it is LR(0) grammar

so, " " SLR(1) "

" " " LALR(1) "

" " " CLR(1) "

NOTE :-

- Bottom up parsers have more power as compared to top down parser.

Eg: see from Q3, given grammar is not LL(1) ie, cannot be parsed by top down parser but is LR(0), SLR(1), LALR(1), CLR(1) ie, can be parsed by bottom up parser.

Q4 $E \to E+T^{(1)} \mid T^{(2)}$

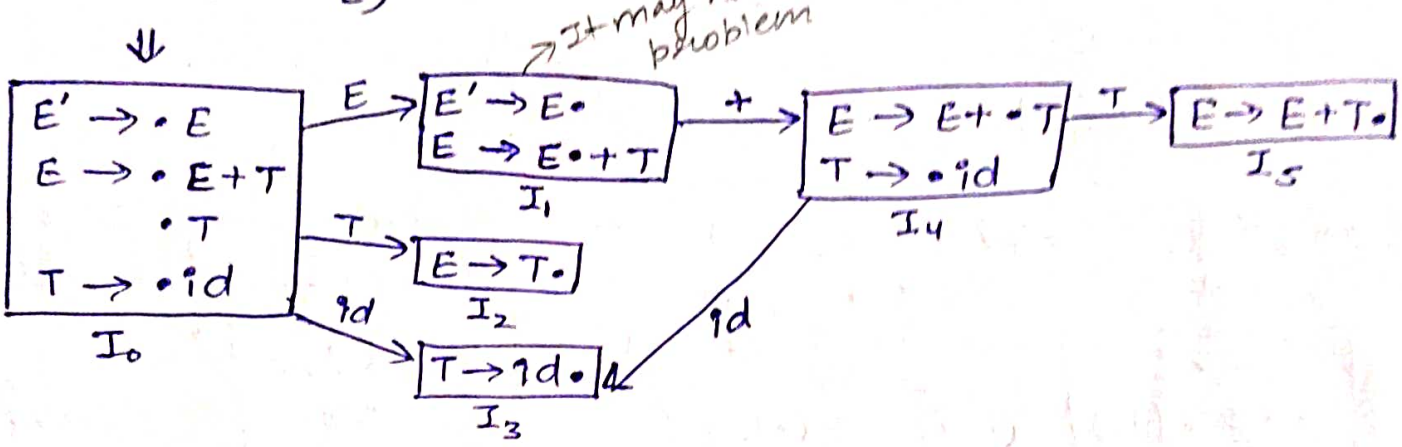
$T \to id^{(3)}$

LL(1)? LR(0)? SLR(1)? LALR(1)? CLR(1)?

Sol": It has left recursion

so, it is not LL(1) grammar

closure $(E' \to .E)$



In state I_1,

$E' \to E.$ → It is a augmented production.

$E \to E.+T$ It does not give reduce so, Neglect it,

Bcz SR or RR any conflict is not possible

Hence, given grammar is LR(0)

⇓

SLR(1)

⇓

LALR(1)

⇓

CLR(1)

only expand those states in which there is some hope of getting problem in future to save time.

eg. $S \to .abc \xrightarrow{a} S \to a.bc$

$S \to .abc$ ↓ No need to expand bcz only terminals are there after dot(.)

$S \to a.bc$ ↓ No need to expand!

so, $S \to .abc \to$ [ ]

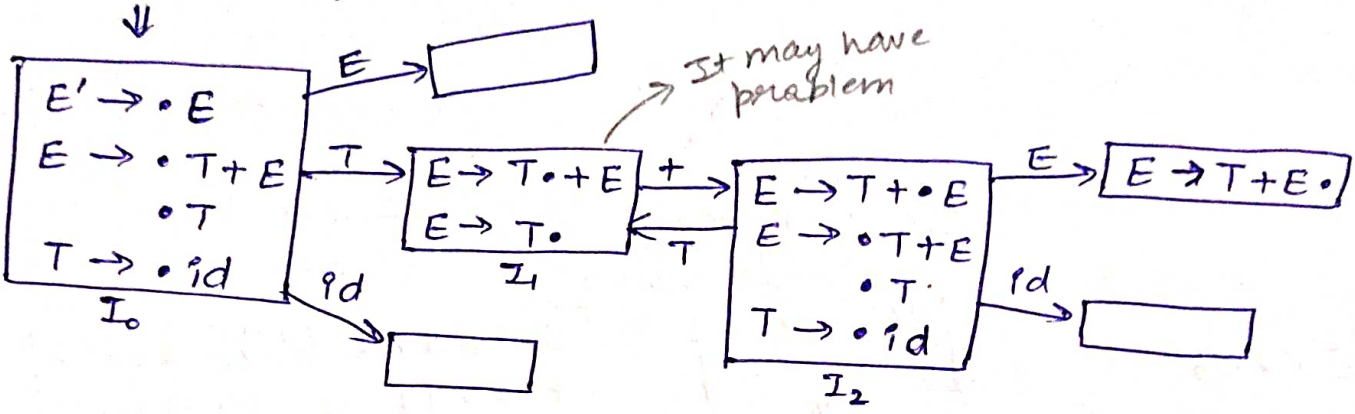
Q5 $E \xrightarrow{①} T+E \mid \xrightarrow{②} T$

$T \xrightarrow{③} id$

LL(1) ? LR(0)? SLR(1)? LALR(1)? CLR(1)?

Sol: This grammar contains left factoring so, it is not LL(1)

closure($E' \to \cdot E$)



	Action		
	+	id	$
1	S_2/r_2	r_2	r_2

LR(0) - parsing table

There is SR conflict so, it is not LR(0)

⇓ check for SLR(1)

	Action		
	+	id	$
1	S_2		r_2

SLR(1) parsing table

NO conflict

So, it is SLR(1)

⇓

LALR(1)

⇓

CLR(1)

In SLR(1), reduce entry for $E \to T\cdot$ will come in follow(E) columns only.

follow(E) = $

(64)

QC. $E \to E+T \;(1) \mid T \;(2)$

$T \to TF \;(3) \mid F \;(4)$

$F \to F* \;(5) \mid a \;(6) \mid b \;(7)$

LL(1) ?
LR(0) ?
SLR(1) ?
LALR(1) ?
CLR(1) ?

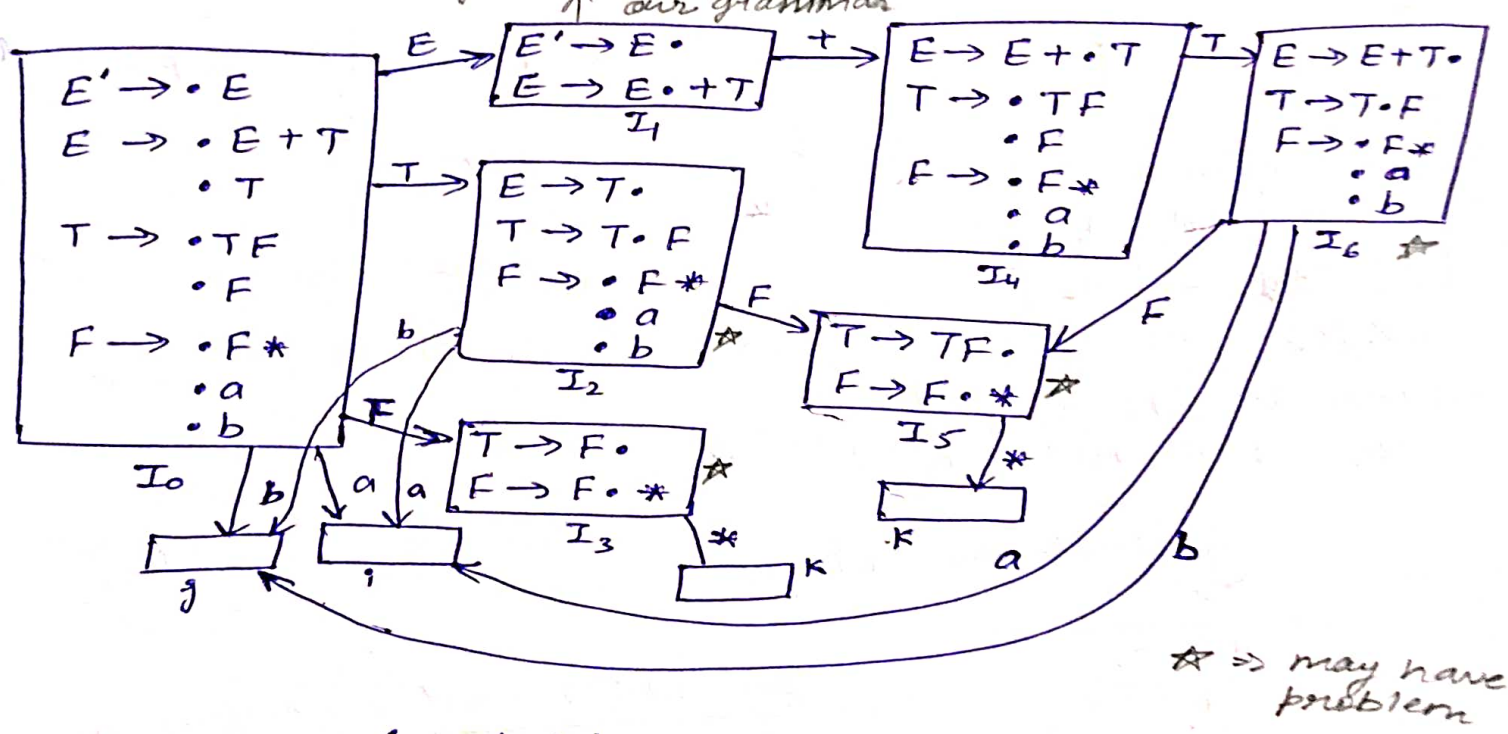
It has left recursion
so, it is not LL(1)

Remember !!

LL(1) ⇒ LR(0) ✗

LR(0) ⇒ SLR(1) ⇒ LALR(1) ⇒ CLR(1)

LL(1) ⇒ CLR(1)



☆ ⇒ may have problem

LR(0) parsing table

	action				
	+	*	a	b	$
2	r_2	r_2	r_2/S_i	r_2/S_j	r_2
3	r_4	r_4/S_K	r_4	r_4	r_4
5	r_3	r_3/S_K	r_3	r_3	r_3
6	r_1	r_1/S_K	r_1/S_i	r_1/S_j	r_1

There are SR conflict

so, it is not LR(0)

[4 inadequative states
6 SR conflicts]

⟹ check SLR(1)

SLR(1) parsing table

	action				
	+	*	a	b	$
2	r_2		S_i	S_j	r_2
3	r_4	S_K	r_4	r_4	r_4
5	r_3	S_K	r_3	r_3	r_3
6	r_1		S_i	S_j	r_1

follow(E) = +, $
follow(T) = +, $, a, b

[0 inadequative states
0 conflicts]

So, it is SLR(1) → LALR(1) → CLR(1)

- **Inadequative states:**

State which has conflict is known as Inadequative state.

Lect-12

Q7

$S \to Aa\overset{①}{A}b \mid Bb\overset{②}{B}a$

$A \to \overset{③}{\epsilon}$

$B \to \overset{④}{\epsilon}$

LL(1) ?
LR(0) ?
SLR(1) ?
LALR(1) ?
CLR(1) ?

Sol

$S \to AaAb \mid BbBa$

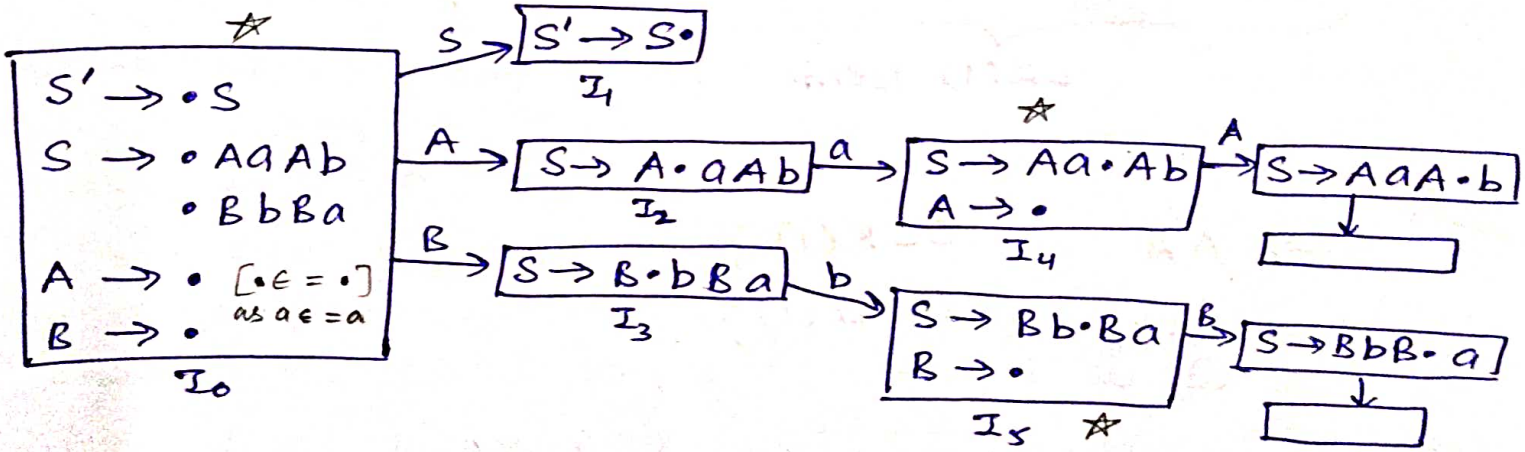
first = a

first = b

$\cap = \phi$

So, given grammar is LL(1)

So, given grammar is CLR(1)



LR(0) parsing table

	Action		
	a	b	$
0	r_3/r_4	r_3/r_4	r_3/r_4
4	r_3	r_3	r_3
5	r_4	r_4	r_4

looking like problem but don't have conflict

1 inadequative state

3 RR conflict

So, it is not LR(0) → bcz of RR conflict

Stop checking further when any conflict appears (to save time)

SLR(1) parsing table

	Action		
	a	b	$
0	$r_3/r_{4,1}$	r_3/r_4	

1. Inadequate state
2. RR conflict

So, it is not SLR(1)

follow(A) = a, b
follow(B) = a, b

- **How to check given grammar is CLR(1) & LALR(1):-**

using LR(0) items ⇒ LR(0), SLR(1) parsers
using LR(1) items ⇒ CLR(1), LALR(1) parsers

LR(1) item = LR(0) item + look ahead symbol

eg. $S \to \cdot xyz$, $a|b$

$S \to \cdot xyz$: LR(0) item
$a|b$: look ahead symbol
$S \to \cdot xyz$, $a|b$: LR(1) item

Q8

$S \to \overset{①}{A}\ A$

$A \to \underset{②}{a}A \mid \underset{③}{b}$

CLR(1) ?
LALR(1) ?

$S' \to \cdot S, \$$ → Augmented LR(1) item

- closure ($S' \to \cdot S, \$$)

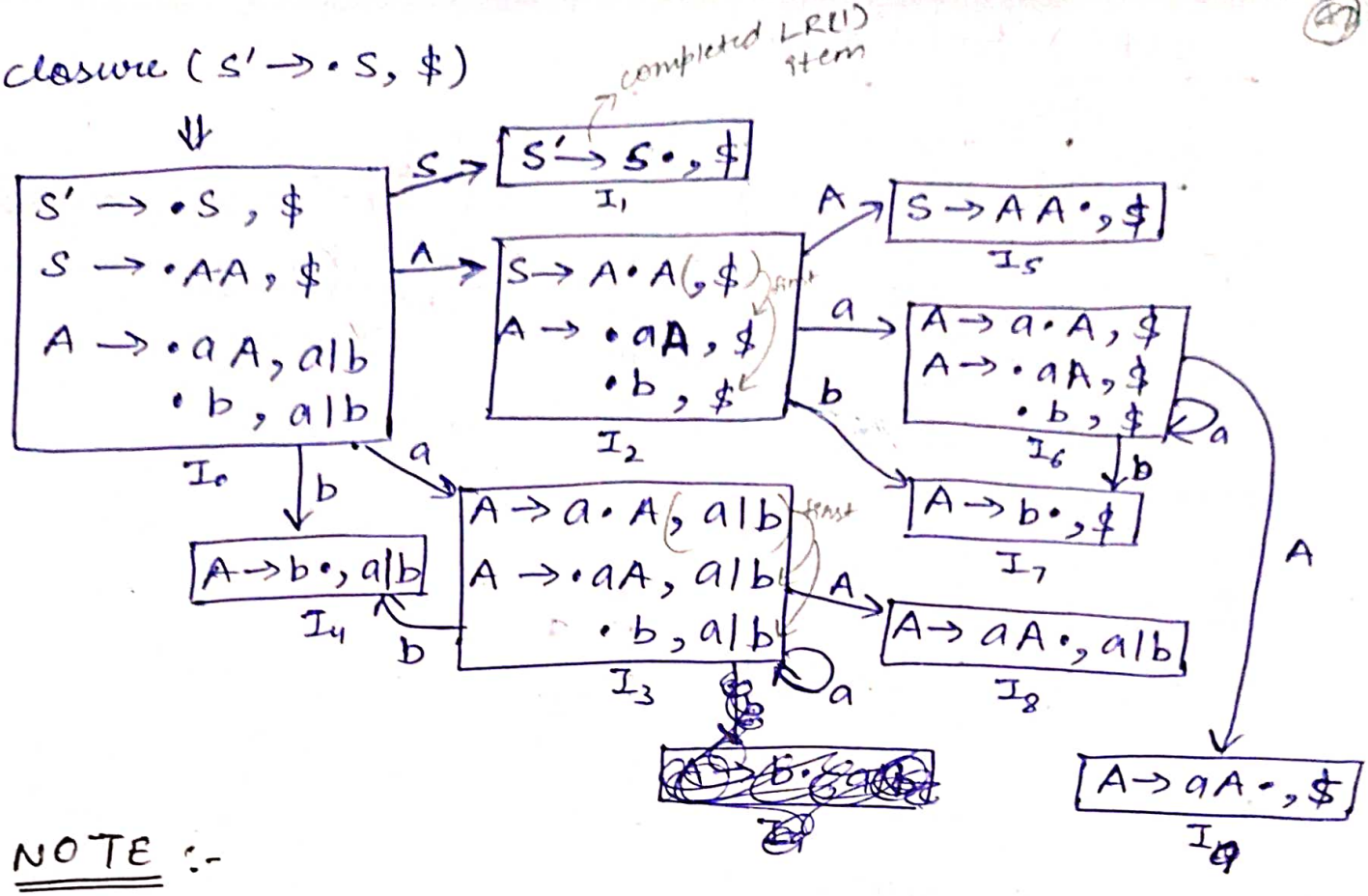
Method to find look ahead

$S' \to \cdot S, \$$ ⟹ first(, $) = $

$S \to \cdot A, \$$ ⟹ first(A, $) = first(A) = a, b

$A \to \cdot a\,A, a|b$
$\cdot b, a|b$

closure (S' → • S, $)



NOTE :-

- For the given grammar, CLR(1) has more states than LR(0) or SLR(1) or LALR(1) bcz some of states differ by look ahead symbol only in CLR(1) or LALR(1)

eg. For above grammar,

LR(0) } 7 states
SLR(1) }

CLR(1) → 10 states
LALR(1) → 7 states

- CLR(1) parser is most powerful bcz it calculates look ahead every time (while SLR(1) calculates follow only on last).
The only problem with CLR(1) is no. of states are more so, cost is more.
- To decrease the cost of CLR(1), we apply minimization on CLR(1) i.e., LALR(1).
(If 2 states differ by only lookahead, make it as single state). Minimized CLR(1) is called LALR(1)

CLR(1) parsing table

	Action			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3				8
4	γ_3	γ_3			
5			γ_1		
6	S_6	S_7			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		

WRITE reduce entries in look ahead columns only.

⇓

No conflict

⇓

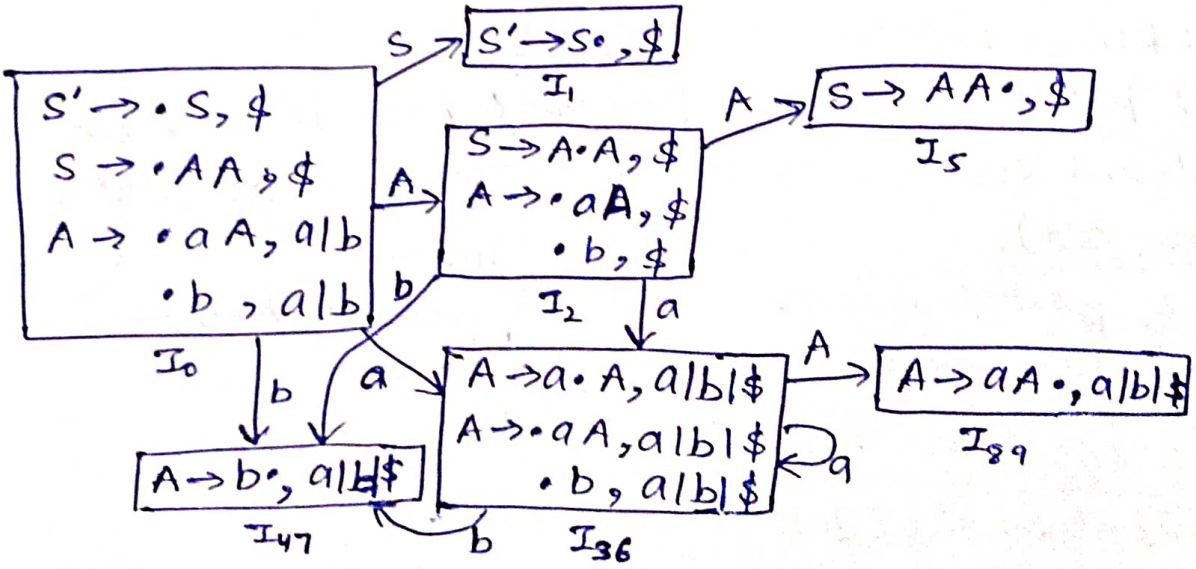
So, it is CLR(1) grammar

Apply minimization,

$I_3, I_6 \Rightarrow I_{36}$

$I_4, I_7 \Rightarrow I_{47}$

$I_8, I_9 \Rightarrow I_{89}$



Modified DFA for LALR(1)

LALR(1) parsing table

	Action			Goto	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

For a given grammars, if
n_1 ⇒ no. of states in LR(0)
n_2 ⇒ no. of states in SLR(1)
n_3 ⇒ no. of states in CLR(1)
n_4 ⇒ no. of states in LALR(1)
then,
$$n_1 = n_2 = n_4 \leq n_3$$

- Lookahead symbol can never be 'ϵ'.
  If 'ϵ' comes anywhere then, go to parent.

Q9 $S \to Aa^{①} \mid bAc^{②} \mid Bc^{③} \mid bBa^{④}$ CLR(1)? LL(1)? LALR(1)?

$A \to d^{⑤}$

$B \to d^{⑥}$

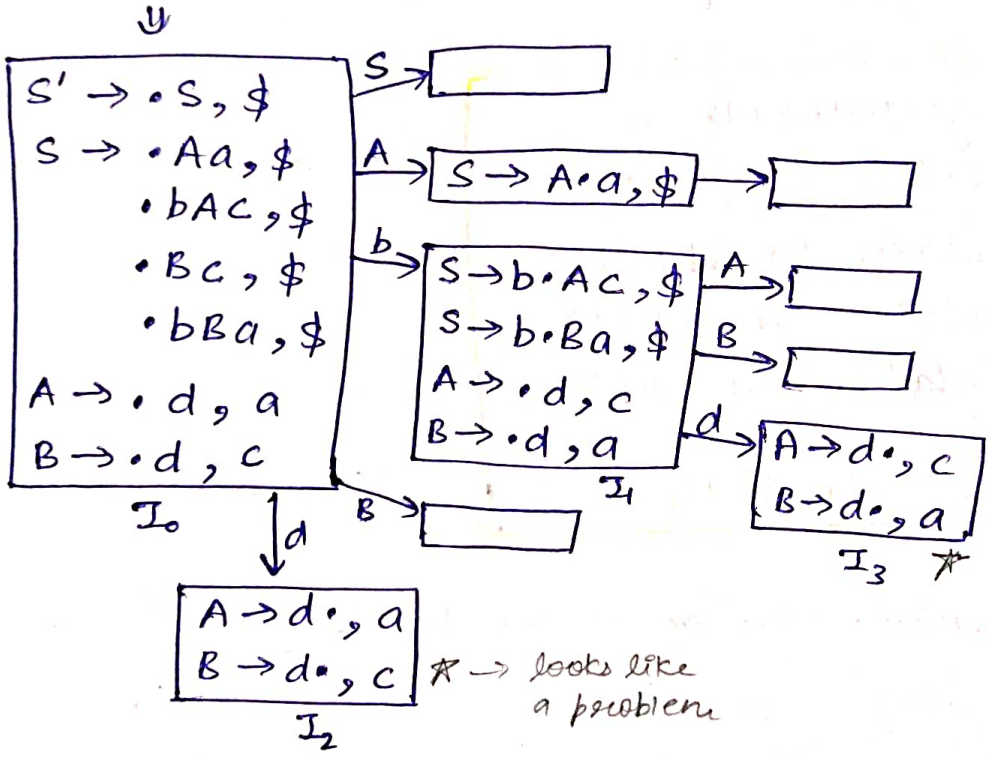
Sol: $S \to Aa \mid bAc \mid Bc \mid bBa$

It has left factoring

so, it is not LL(1) grammar

for CLR(1):

closure $(S' \to \cdot S, \$)$



Action

	a	b	c	d	$
2	r_5		r_6		
3	r_6		r_5		

CLR(1) parsing table

No conflict

so, it is CLR(1)

Note that given gramma

LR(0) ✗

SLR(1) ✗

If For a given grammar, Ques asking LR(0), SLR(1), CLR(1), LALR(1) then, only draw DFA for CLR(1) and then, check for all by determining where reduce entries will come in each

For checking LALR(1)

I_2 , I_3 $\Rightarrow$ I_{23}

	Action				
	a	b	c	d	$
23	r_5/r_6		r_5/r_6		

RR conflict

So, it is not LALR(1)

V.V Imp

NOTE :-

- If CLR(1) don't have RR conflict then, LALR(1) may contain RR conflict
- If CLR(1) don't have SR conflict then, LALR(1) also don't have SR conflict.

Q10 $S \rightarrow \overset{①}{A}aAb \mid \overset{②}{B}bBa$

$A \rightarrow \epsilon$ ③

$B \rightarrow \epsilon$ ④

LR(0)?
SLR(1)?
CLR(1)?
LALR(1)?
LL(1)?

$S \rightarrow AaAb \mid BbBa$

↓↓first ↓↓first

a b

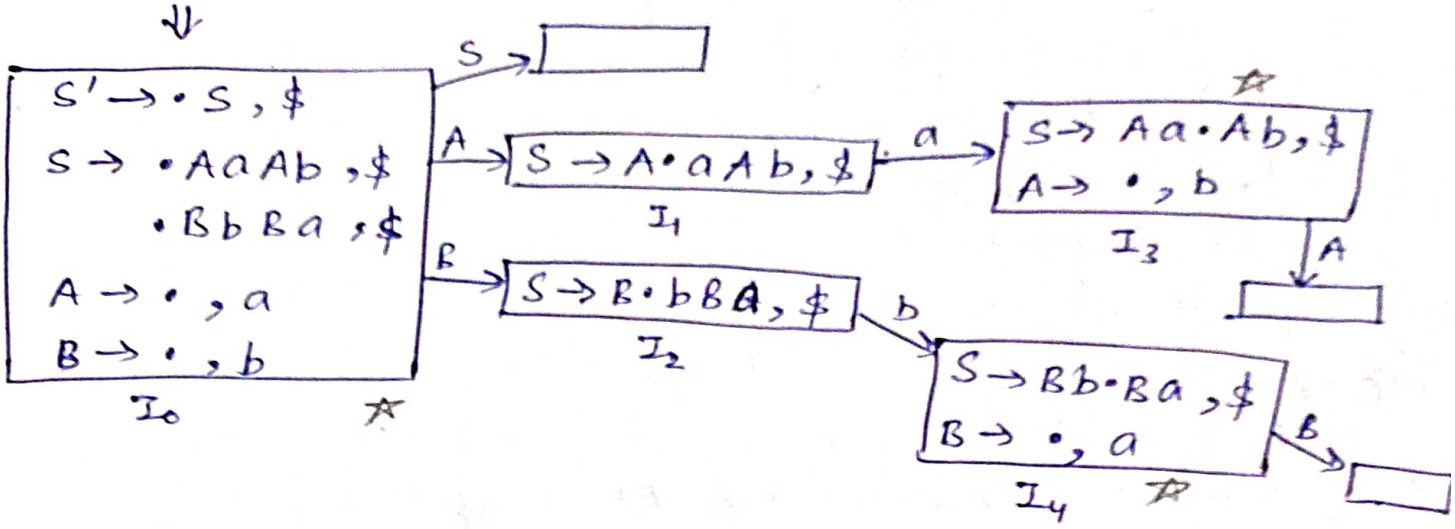
$\cap = \phi$

So, it is LL(1)

So, it is CLR(1).

But still we draw DFA for CLR(1) bcz we want to check others using that

closure ($S' \to \cdot S, \$$)



LR(0)

Action

	a	b	$
0	r_3/r_4	r_3/r_4	r_3/r_4
3	r_3	r_3	r_3
4	r_4	r_4	r_4

So, it is not LR(0)
(RR conflict)

SLR(1)

follow(A) = a, b
follow(B) = a, b

Action

	a	b	$
0	r_3/r_4	r_3/r_4	

So, it is not SLR(1)
(RR conflict)

CLR(1)

Action

	a	b	$
0	r_3	r_4	
3		r_3	
4	r_4		

So, it is CLR(1)
(No conflict)

LALR(1)

It is already minimized bcz no 2 states differ by lookahead

So, it is LALR(1) also.

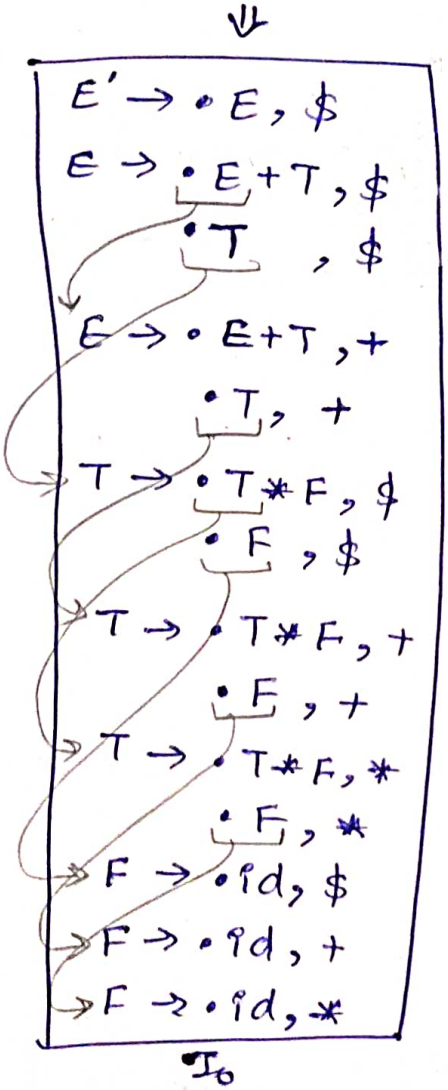
Don't bother about gaps (states we didn't expand) bcz even if they combine then also no conflict will be there

Q11 $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

construct initial 5 states of DFA of CLR(1) parser.

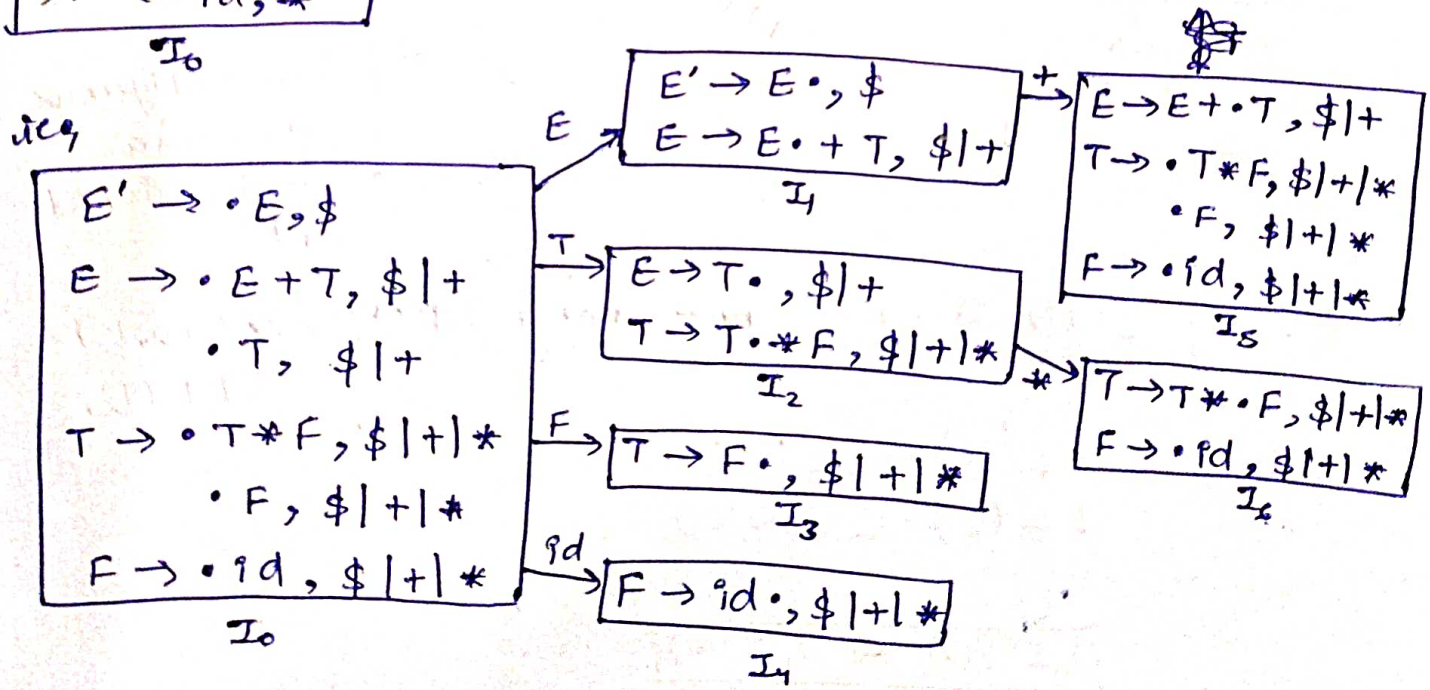
Sol

Closure $(E' \to \cdot E, \$)$



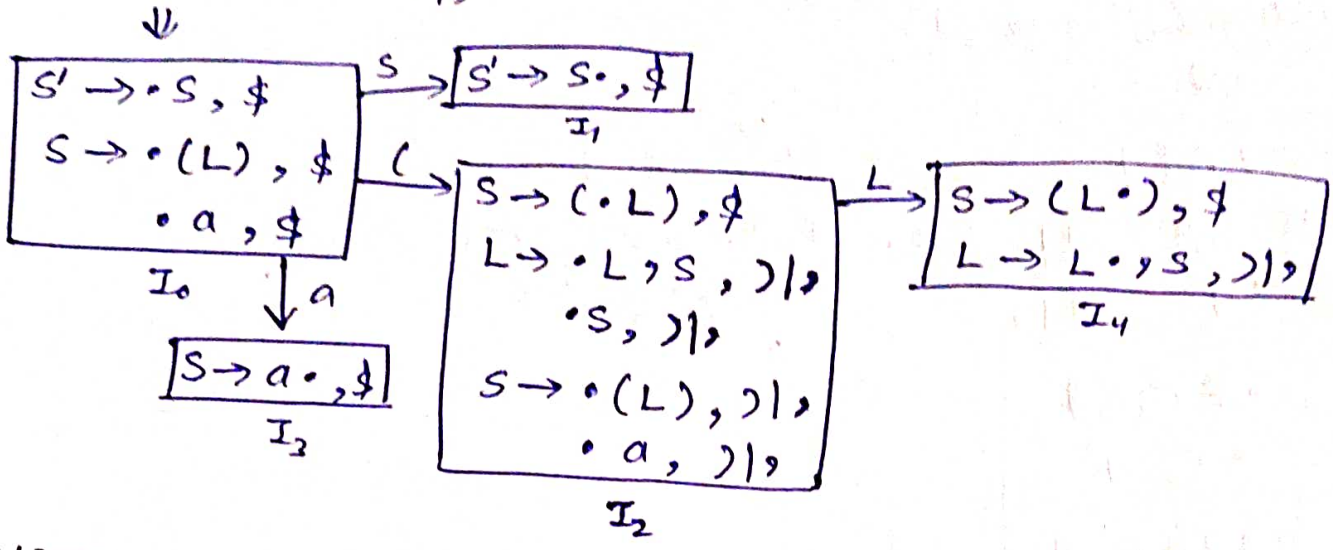
I_0

i.e., ~~E' → · E, $~~ ~~E → E+T, $|+~~ ~~· T,~~

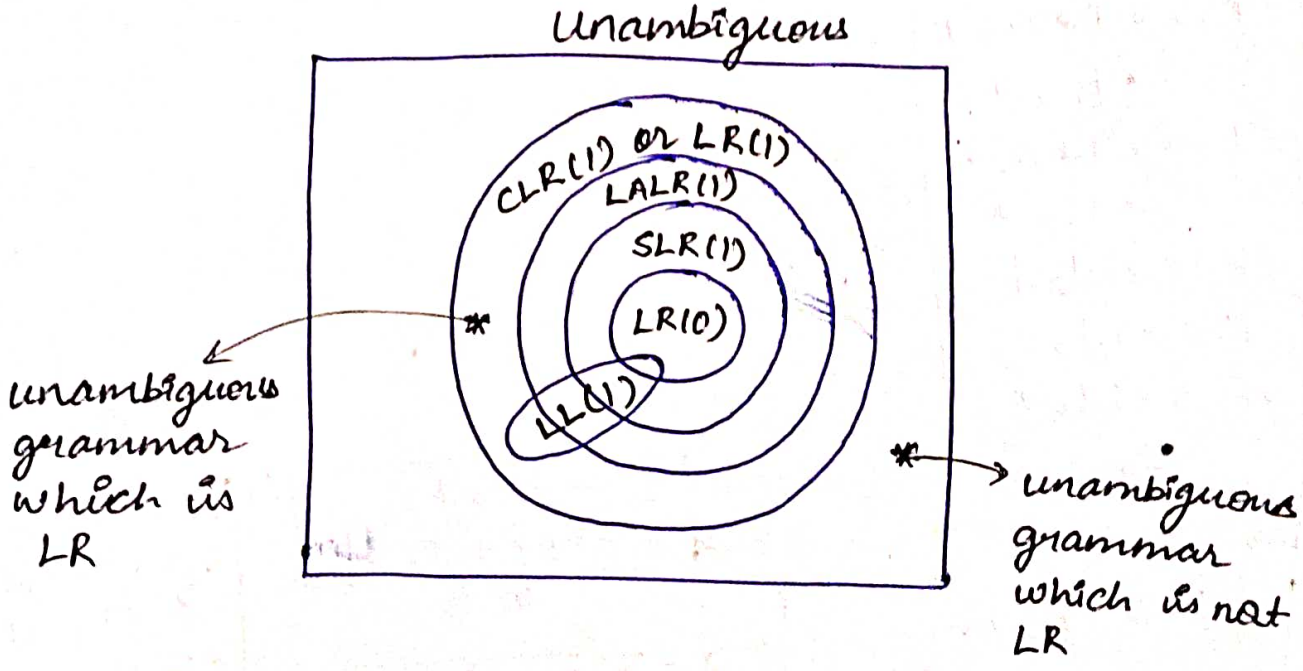


Q12 $S \to (L) \mid a$
$L \to L, S \mid S$
Construct 1st 5 states of DFA of CLR(1).

Sol
closure $(S' \to \cdot S, \$)$



NOTE :-



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

$LL(1) \subseteq LR(1)$
$LL(2) \subseteq LR(2)$
$\vdots$
$\boxed{LL(K) \subseteq LR(K)}$

⇒ Operator precedence parser

- Operator precedence parser will be applicable only for operator grammar.
- Grammar G is said to be operator grammar iff
  1) G don't have NULL productions.
  2) G don't have two adjacent variables on RHS of production.

eg.1 $E \to E+E \mid E*E \mid id$ → operator grammar ✓
Ambiguous grammar ✓
operator precedence parser ✓

eg.2
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$
operator grammar ✓
unambiguous grammar ✓
operator precedence parser ✓

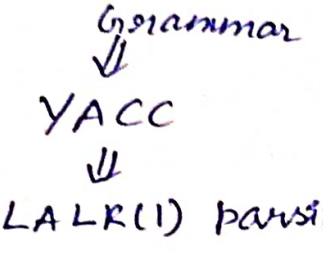
eg.3 $S \to AB$
$A \to a$
$B \to b$
unambiguous ✓
operator grammar ✗
operator precedence parser ✗

eg.4 $E \to E+E \mid E*E \mid id \mid \epsilon$
Ambiguous ✓
operator grammar ✗
operator precedence parser ✗

(16)

* YACC tool

It is a LALR(1) parser generator



If there are conflicts in LALR(1) parsing table then, YACC will resolve them.

SR conflict ⇒ replace with S (Shift)

RR conflict ⇒ replace with 1st reduce

eg. $\gamma_3 | \gamma_4$ ⇒ replace with γ_3

See 022 & 023 of WB to understand YACC better

Syntax directed translation (SDT)

Imp

Lect-14

SDT = Grammar + Semantic actions

eg.1 $E \rightarrow E_1 + E_2$ { E.val = E_1.val + E_2.val }

eg.2 $E \rightarrow E_1 * E_2$ { if(E_1.Type = E_2.Type)
E.Type = E_1.Type
else printf (Type mismatch) }

eg.3 $S \rightarrow abc$ { printf (hi) }

- SDT is new phase of compiler which is helping to all other phases of complier.

Syntax directed translation (SDT)

Imp

Lect-14

SDT = Grammar + Semantic actions

eg.1 $E \rightarrow E_1 + E_2 \{ E.val = E_1.val + E_2.val \}$

eg.2

$$E \rightarrow E_1 * E_2 \left\{ \begin{array}{l} if(E_1.Type = E_2.Type) \\ \quad E.Type = E_1.Type \\ else\ printf(Type\ mismatch) \end{array} \right\}$$

eg.3 $S \rightarrow abc \{ printf(hi) \}$

- SDT is new phase of compiler which is helping to all other phases of complier.

Attributes

⇒ In grammar, every variable has many attributes

Synthesized Attribute

Inherited attribute

eg: of synthesized attribute

$S \rightarrow ABC \{ S.val = f(A.val \mid B.val \mid C.val) \}$

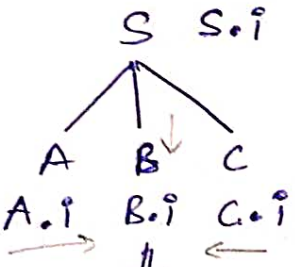
$S.val = A.val + B.val * C.val$

'val' is one of the attribute

(Parent calculated using children)

eg: of Inherited attribute

$S \rightarrow ABC \{ B.i = f(A.i \mid C.i \mid S.i) \}$



'i' is one of the attribute

$B.i = A.i + C.i * S.i$ (characteristics inherited from parents or siblings)

Synthesized attribute: At any variable, attribute value is calculated using children attribute value

Inherited attribute: At any variable, attribute value is calculated using parent, siblings attribute value.

- Depending on type of attributes used, SDT are of two types:

SDT

S-attributed definition

1. It can use synthesized attributes only. sometimes, no attribute used
2. Semantic actions will be placed on the rightmost place of right hand side of the production.
3. It uses bottom up evaluation.

L-attributed definition

1. It can use both synthesized & inherited attributes. sometimes, no attribute used but for inherited attributes, use attribute value either from parent or left sibling but not from right sibling.

   Inherited → parent ✓
   left ✓
   right ✗

2. Semantic actions will be placed anywhere on the right hand side of the production.
3. It follows depth first order, left to right evaluation

- S-attributed definition SDT $\subseteq$ L-attributed definition SDT

Q) Construct SDT to evaluate given arithmetic expression:

i/p : 2+3*4

o/p : 14

Write unambiguous grammar to get unique o/p.

Sol

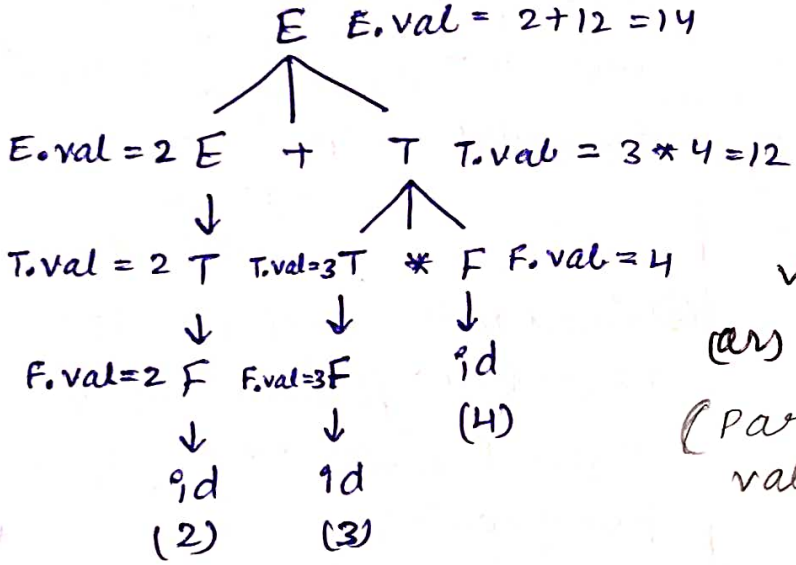
$E \rightarrow E_1 + T \quad \{E.val = E_1.val + T.val\}$ (Here, renaming of E is done as E_1)

$\quad | T \quad \{E.val = T.val\}$ (Here T is renamed as T to avoid confusion. To check whether it is renaming or separate variable in exam, see variable list)

$T \rightarrow T_1 * F \quad \{T.val = T_1.val * F.val\}$

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

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



Annonated parse tree (or) Decorated parse tree

(Parse tree with attribute value of each variable)

Note that here,

val is a synthesized attribute

so, it is a S-attributed definition

so, it is ~~also~~ L-attributed definition also.

[GATE]

Q2 common data

Consider the following SDT

$E \rightarrow E \# T \quad \{E.val = E_1.val * T.val\}$

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

$T \rightarrow T \& F \quad \{$ _____ $\}$

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

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

i) If the expression 8 # 12 & 4 # 16 & 12 # 4 & 2 is evaluated to 512, then which one of the following correctly represents the blank above?

a) $T.val = T_1.val * F.val$

b) $T.val = T_1.val - F.val$

c) $T.val = T_1.val + F.val$

d) None

From grammar, we get to know

priority : & > #

meaning : & → ?

→ *

Associativity : & → L-R

→ L-R

8 # 12 & 4 # 16 & 12 # 4 & 2

a) $(((8 * (12 * 4)) * (16 * 12)) * (4 * 2)) \neq 512$

b) $(((8 * (12-4)) * (16-12)) * (4-2))$ → we have put brackets acc to precedence & associativity for our convinience

$= (((8 * 8) * 4) * 2) = 512$

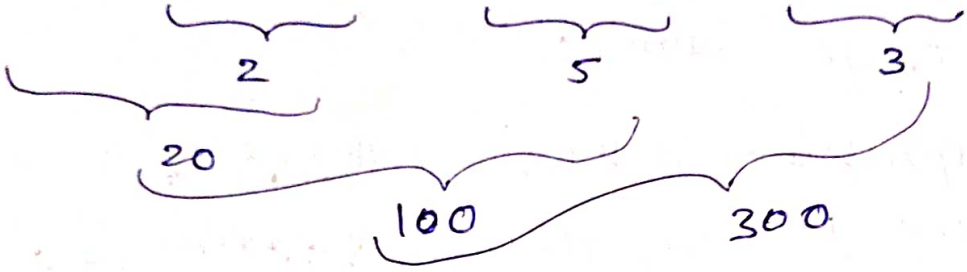
so, (b) is correct

9) The expression 10 # 8 & 6 # 9 & 4 # 5 & 2 will be evaluated to

a) 300 b) 12740 c) 400 d) None

10 # 8 & 6 # 9 & 4 # 5 & 2

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



Q3 Construct SDT to convert the given infix expression to postfix?

i/p : a + b * c (Infix)

o/p : a b c * + (Postfix)

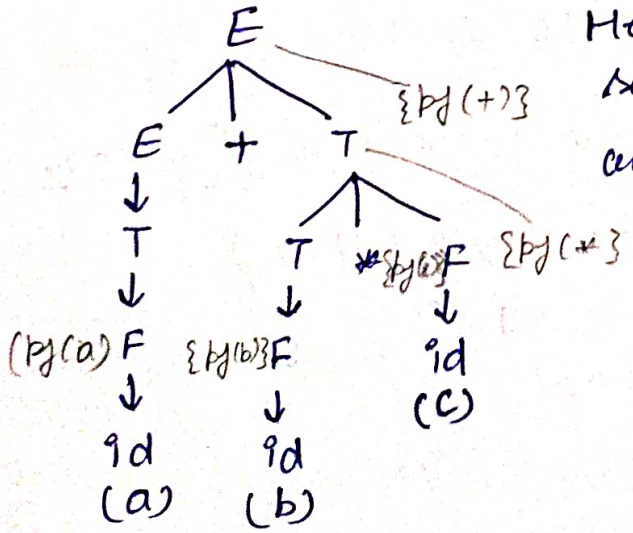
we know, Operands order doesn't change in infix, prefix, postfix expression

Sol^n

E → E + T {pf(+)}
| T (No semantic action req.)

T → T * F {pf(*)} → here, it will execute only after T, & F are over so, postfix
| F (No semantic action req.)

F → id {pf(id)}



Here, no attribute needed so, it is L-attributed defn and also S-attributed defn.

Q4 Construct SDT to convert given infix to prefix expression.

i/p : Infix : a + b * c

o/p : Prefix : + a * bc

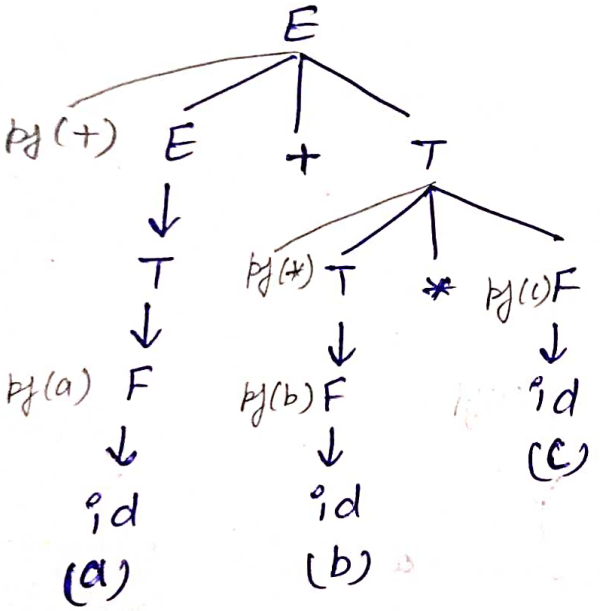
$E \rightarrow \{pf(+)\}\ E + T$ | T

$T \rightarrow \{pf(*)\}\ T * F$ | F

$F \rightarrow id\ \{pf(id)\}$

→ possible bcz operators are fixed in a lang. so we can print them before also

here $F \rightarrow \{pf(id)\}\ id$ NOT possible bcz we cannot print id before reading it



Here, semantic action are not always place in right most place of RHS of production

so, it is not S-attributed definition.

It is L-attributed dyn

→ For i/p : infix : a + b * c

o/p : infix : a + b * c

$E \rightarrow E\ \{pf(+)\}\ +\ T$ | T

$T \rightarrow T\ \{pf(*)\}\ *\ f$ | T

$F \rightarrow id\ \{pf(id)\}$

(or)

$E \rightarrow E\ +\ \{pf(+)\}\ T$ | T

$T \rightarrow T\ *\ \{pf(*)\}\ F$ | F

$F \rightarrow id\ \{pf(id)\}$

Lect-15

Q5 Consider the following SDT

$S \rightarrow AS$ {printf(1)}

$S \rightarrow AB$ {printf(2)}

$A \rightarrow a$ {printf(3)}

$B \rightarrow bC$ {printf(4)}

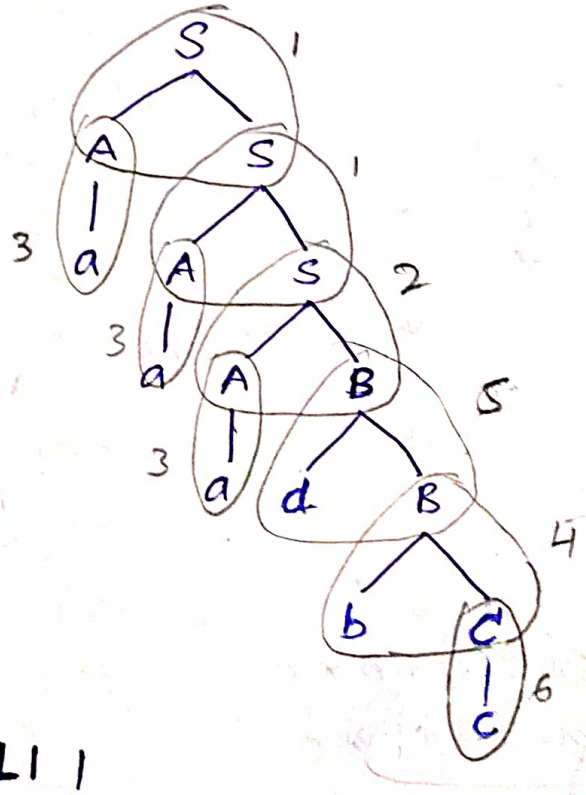
$B \rightarrow dB$ {printf(5)}

$C \rightarrow c$ {printf(6)}

i/p : aaadbc

o/p : ?

Soln



o/p:

3 3 3 6 4 5 2 1 1

Q6 Consider the following SDT:

$S \rightarrow T \{prf(*)\}\ R \{prf(+)\}$

$R \rightarrow +T\ R \{prf(*)\}$

$\quad\quad | \in \{prf(*)\}$

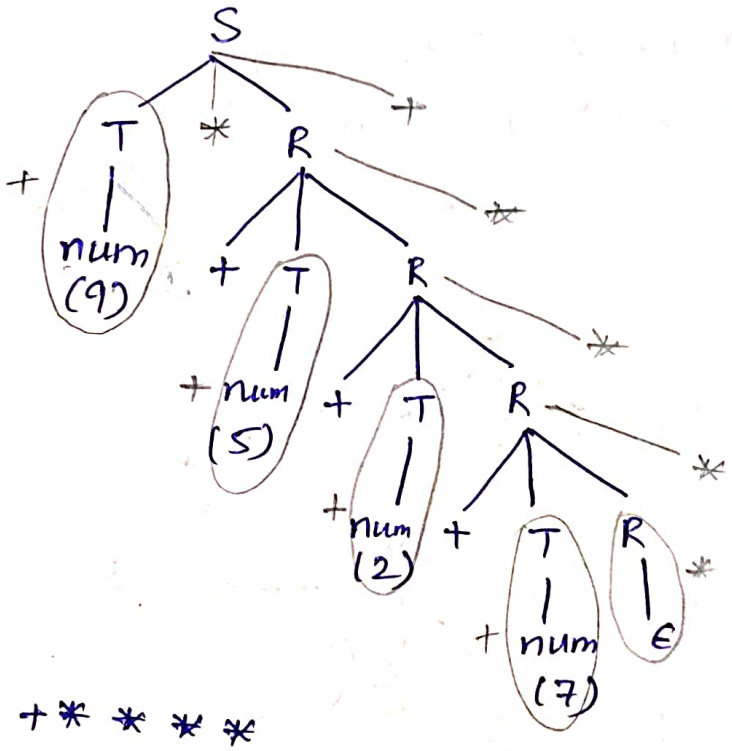
$T \rightarrow num \{prf(+)\}$

It is L-attributed but not S-attributed

I/p: 9+5+2+7

O/p: ?

Sol



O/p: + * + + + * * * *

Q7 Construct SDT to convert binary to decimal

i/p: 1101 i/p: 1101.101

o/p: 13 o/p: 13.625

Sol:

$S \rightarrow L \cdot L \quad \{S.DV = L_1.DV + \frac{L_2.DV}{2^{L_2.nb}}\} \{pf(S.DV)\}$

$\quad | L \quad \{S.DV = L.DV\} \{pf(S.DV)\}$

$L \rightarrow L B \quad \{L.DV = 2 * L_1.DV + B.DV\} \{L.nb = L_1.nb + B.nb\}$

$\quad | B \quad \{L.DV = B.DV\} \{L.nb = B.nb\}$

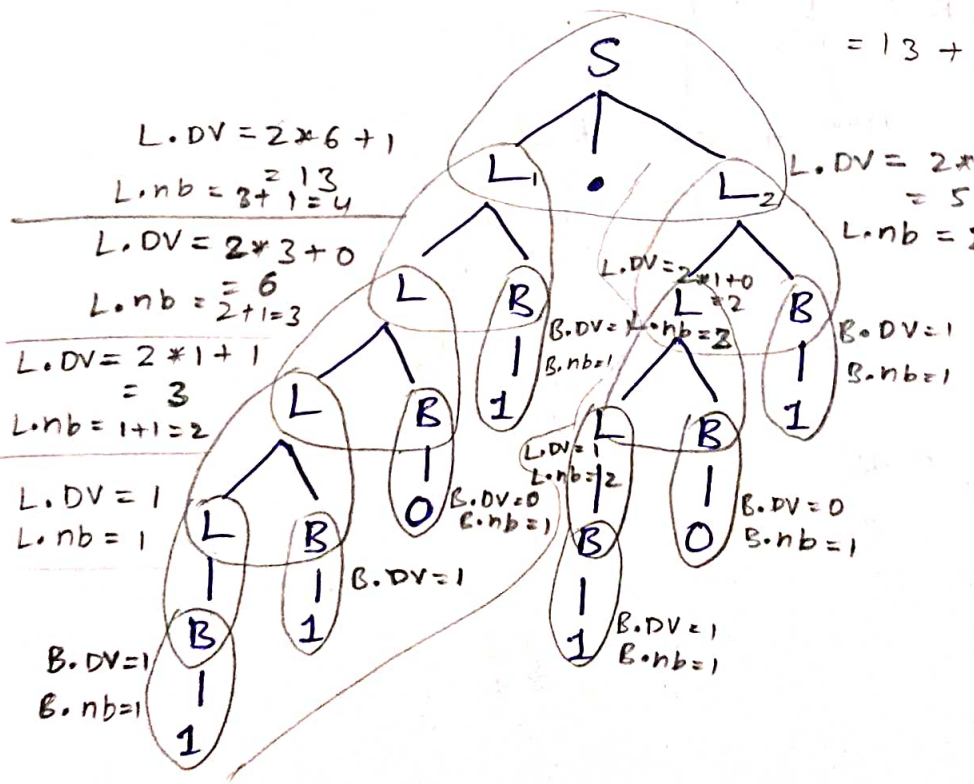
$B \rightarrow 0 \quad \{B.DV = 0\} \{B.nb = 1\}$

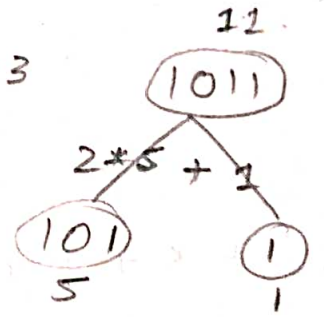
$\quad | 1 \quad \{B.DV = 1\} \{B.nb = 1\}$

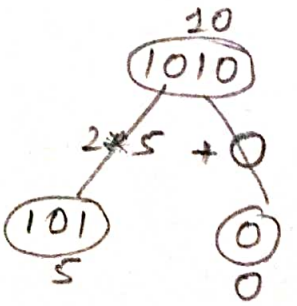
$S.DV = L_1.DV + \frac{L_2.DV}{2^{L_2.nb}}$

$= 13 + \frac{5}{2^3} = 13 + \frac{5}{8} = 13 + 0.625$

$= 13.625$





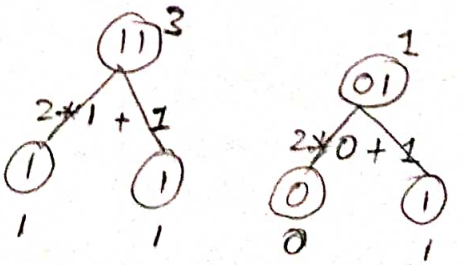


(.01) $\Rightarrow \frac{1}{2^2} = \frac{1}{4} = 0.25$

.25

(.110) $\Rightarrow \frac{6}{2^3} = \frac{6}{8} = \frac{3}{4} = 0.75$

.75



Here, 2 attributes are used

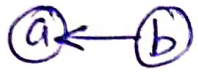
DV and nb ⇒ both are synthesized attributes

So, it is S-attributed dfn

so, it is L-attributed dfn also.

a = b

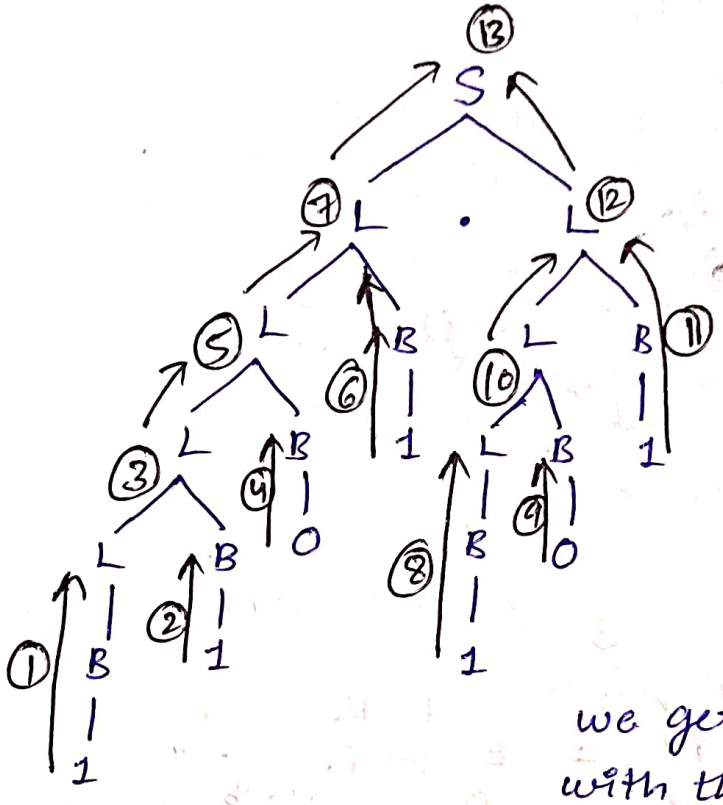
↓



means b depends on a

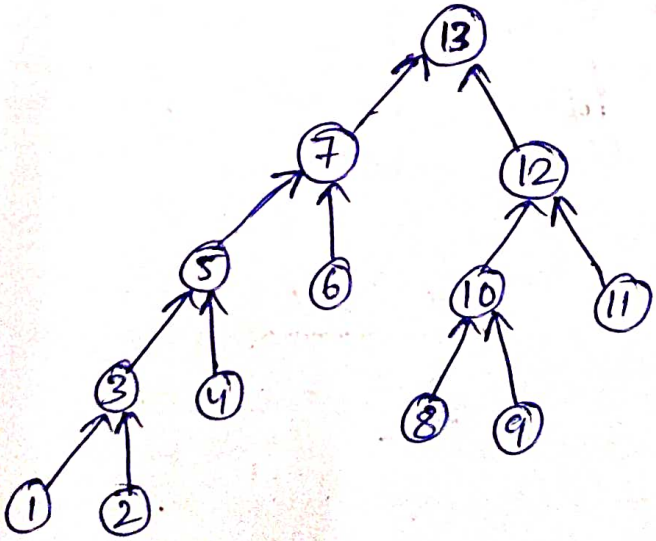
a = b + c

a depends on b and c



we get dependencies with the help of semantic actions

dependency graph :-



Topological sort :

(possible execn order)

4, 2, 1, 3, 5, 6, 7, 11, 8, 9, 10, 12, 13

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

(close to programming)

Q8 Construct SDT to generate intermediate code for the given arithmetic expression

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

o/p : $t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

Sol:

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

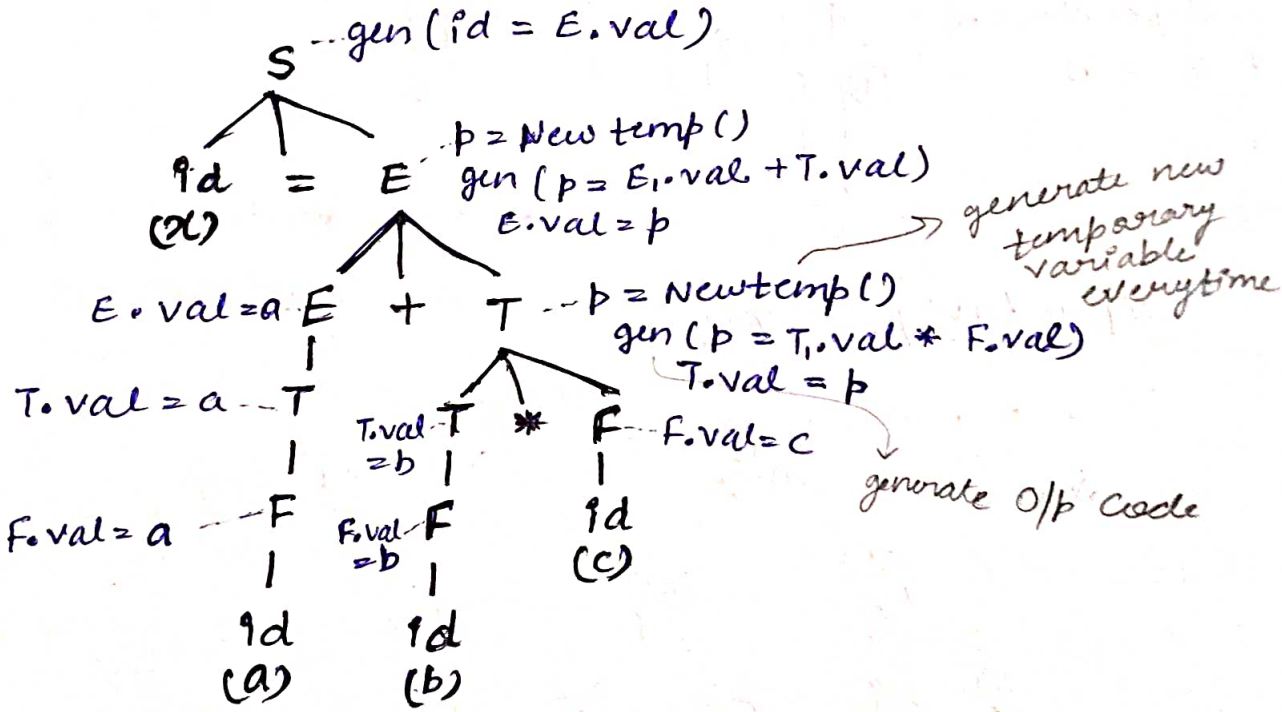
$E \rightarrow E + T \left\{ \begin{array}{l} p = Newtemp() \\ gen(p = E_1.val + T.val) \\ E.val = p \end{array} \right\}$

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

$T \rightarrow T * F \left\{ \begin{array}{l} p = Newtemp() \\ gen(p = T_1.val * F.val) \\ T.val = p \end{array} \right\}$

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

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



Q9 [GATE] consider the following SDT (assuming left to right associativity)

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

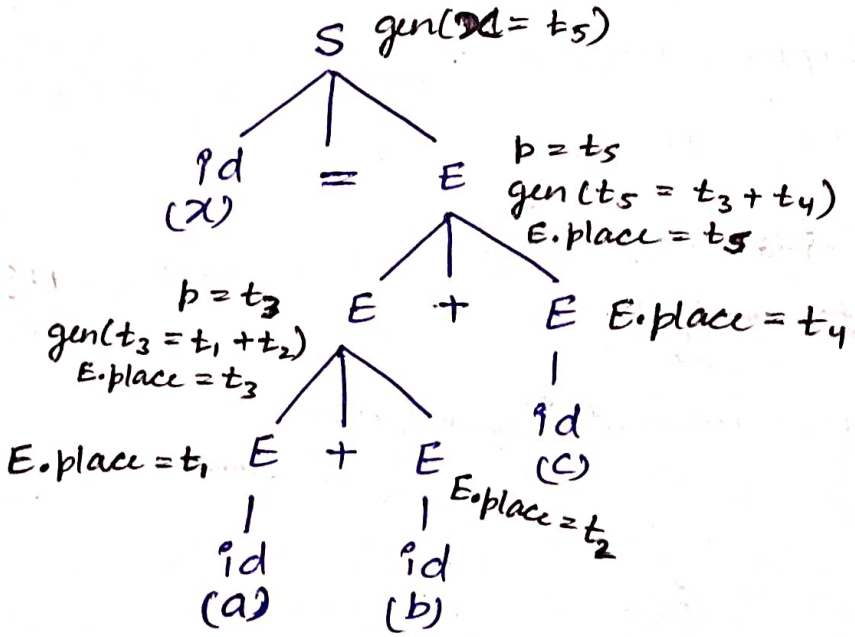
$E \rightarrow E_1 + E_2 \quad \{ p = Newtemp(); \; gen(p = E_1.place + E_2.place); \; E.place = p; \}$

$E = id \quad \{ p = Newtemp(); \; E.place = p; \}$

i/p : $x = a + b + c;$

o/p : ?

Sol:



o/p: $t_3 = t_1 + t_2.$

$t_5 = t_3 + t_4$

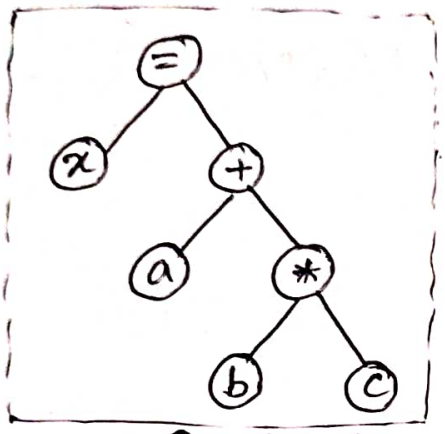
$x = t_5$

(94)

Q10 Construct SDT to create syntax tree for the given arithmetic expression:

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

o/p :



Syntax tree

We know, in syntax tree

leaf nodes ⇒ operands
internal nodes ⇒ operators

$S \rightarrow F = E \{ S.ptr = mknode(F.ptr, =, E.ptr) \; return(S.ptr) \}$

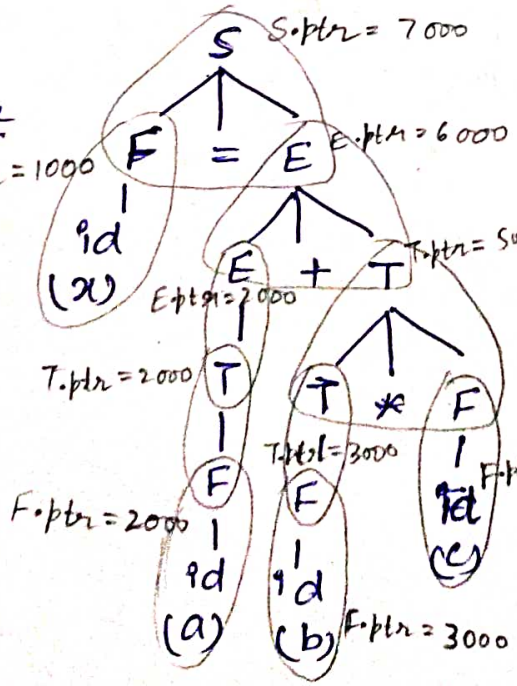
$E \rightarrow E + T \{E.ptr = mknode(E_1.ptr, +, T.ptr)\}$
$\quad | T \{E.ptr = T.ptr\}$

$T \rightarrow T * F \{T.ptr = mknode(T_1.ptr, *, F.ptr)\}$
$\quad | F \{T.ptr = F.ptr\}$

$F \rightarrow id \{F.ptr = mknode(N, id, N)\}$

(mknode: make node; N: Null)

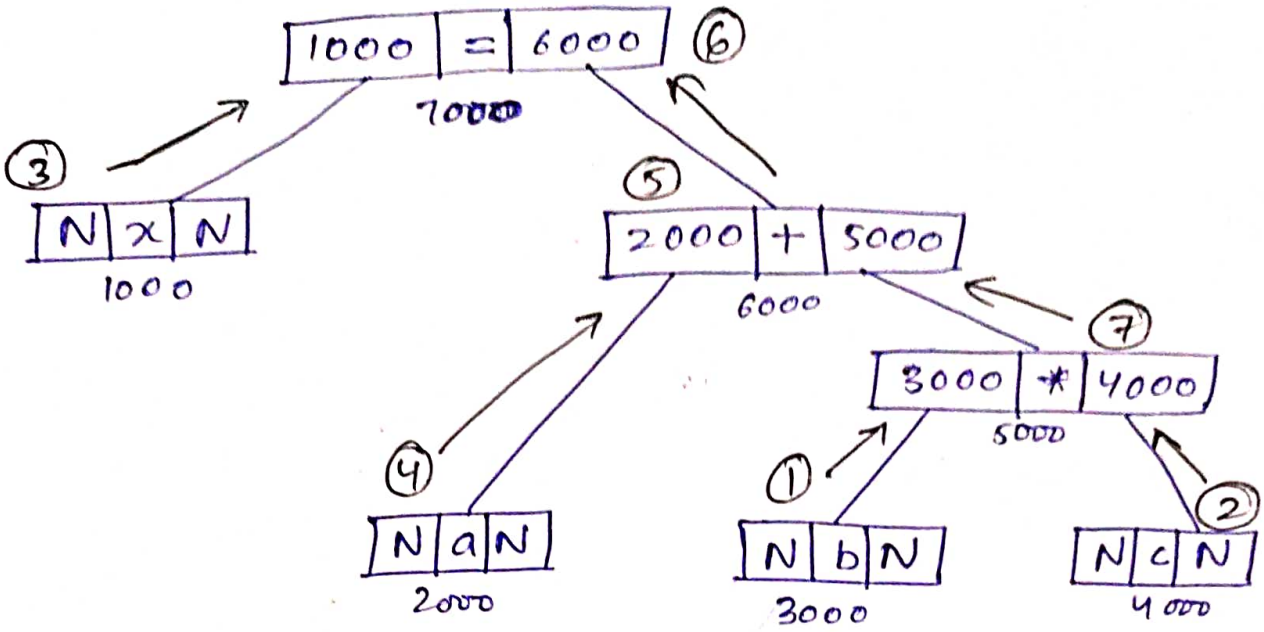
Parse tree



Parse tree

leaf node ⇒ terminals
internal node ⇒ variable

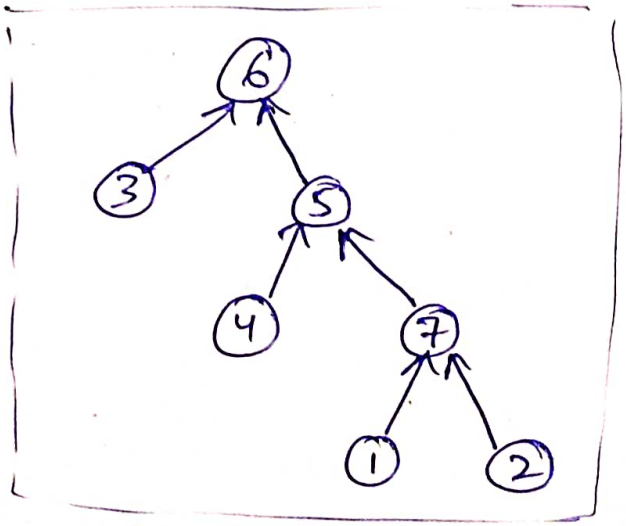
Syntax tree is compressed form of parse tree



Syntax tree

Here, ptr is synthesized attribute

so, SDT is S-attributed dyn & L-attributed dyn also.



dependency graph

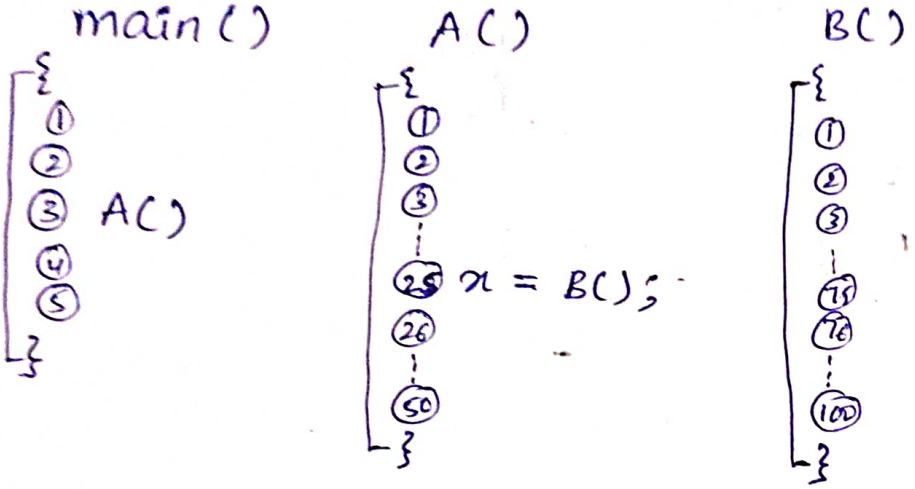
⇒ directed acyclic graph
so, topological sort can be applied

Topological sort : (possible exec^n order)

① 3, 1, 2, 7, 4, 5, 6

② 3, 4, 1, 2, 7, 5, 6 (close to programming)

Runtime Environment

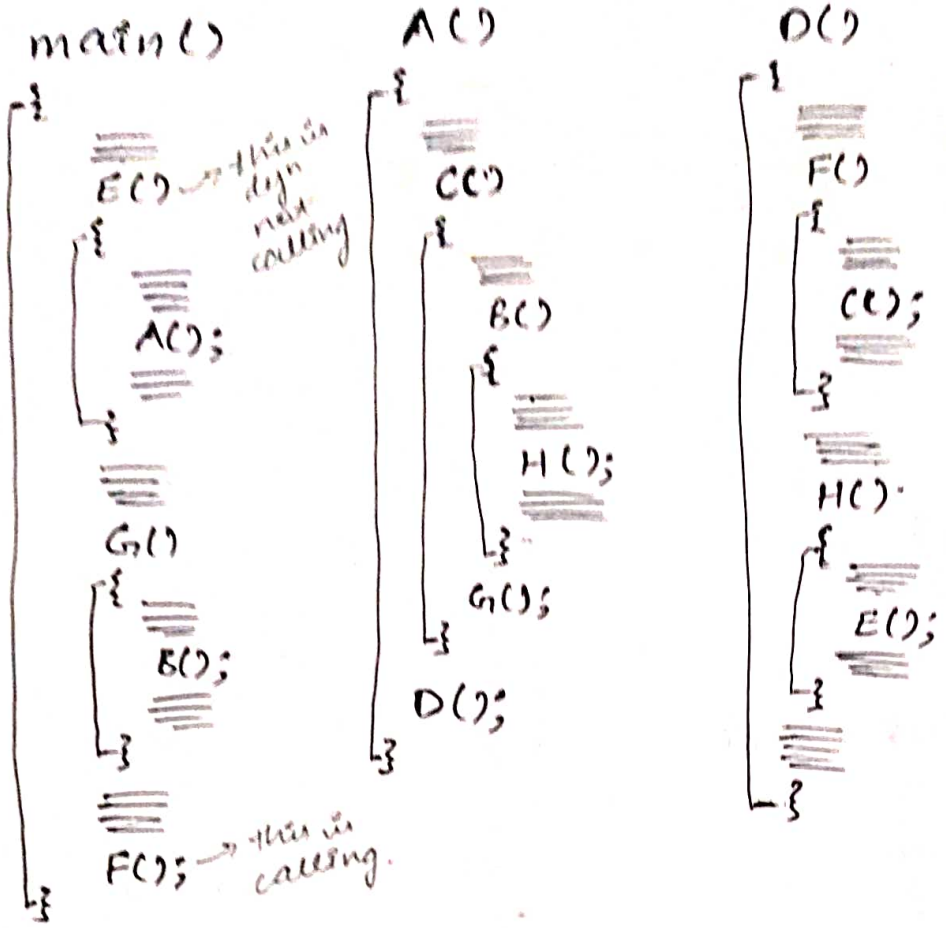


- Whenever a function is called, its activation records is pushed into the stack.
- When a function is executing, its activation record keep on updating.
- When a function terminates, its activation record is popped out of the stack.
- Activation record contains :
  1) Local variables
  2) temporary variables
  3) Return address
  4) Machine status such as registers, program counters etc., before the procedure is called
  5) control link
  6) Access link [where you accessed funcn code] ⇒ stores the info" of data which is outside local scope
  7) Actual parameters

If any of the above is not available then, NULL will be stored in place of that.

(100)

Q [GATE]



Give control links & access links for each funcn.

Sol

Access links	funcn	control links
D()	H()	E()
main()	G()	B()
D()	F()	C()
main()	E()	A()
Null	D()	Null
A()	C()	G()
C()	B()	H()
Null	A()	D()
Null	main	F()

↓ where we accessed code of these funcns

If they are independent, store NULL

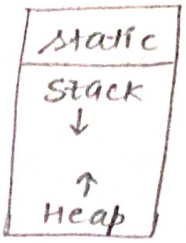
↓ funcn which are called while execution of these funcns

If no funcn is called inside them, store NULL

- Using access link, we can access non-local data.

☆ Imp Storage allocation strategies

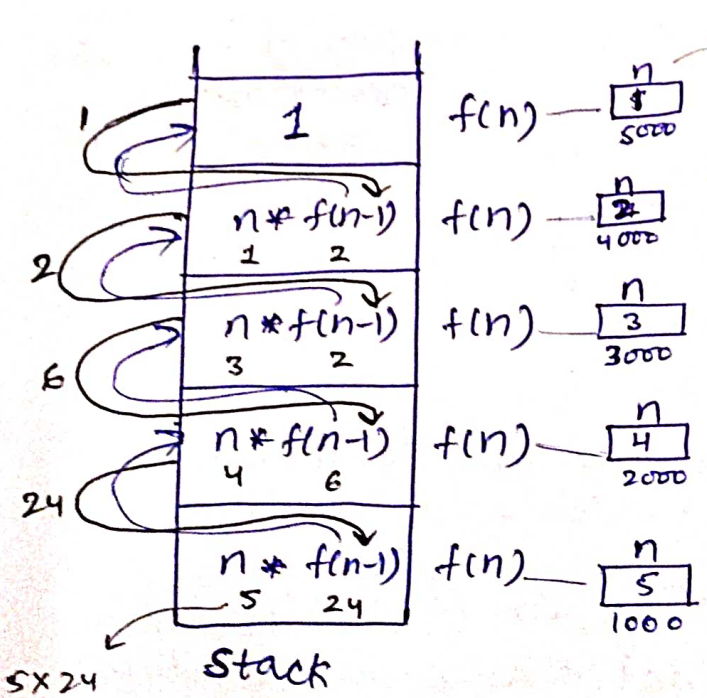
1) Static storage allocation
2) Stack storage allocation
3) Heap storage allocation



eg.

```
fact(int n)
{
  if (n<=1)
  return 1;
  else
  return (n* fact(n-1));
}
```

i/p : n = 5
o/p : ?

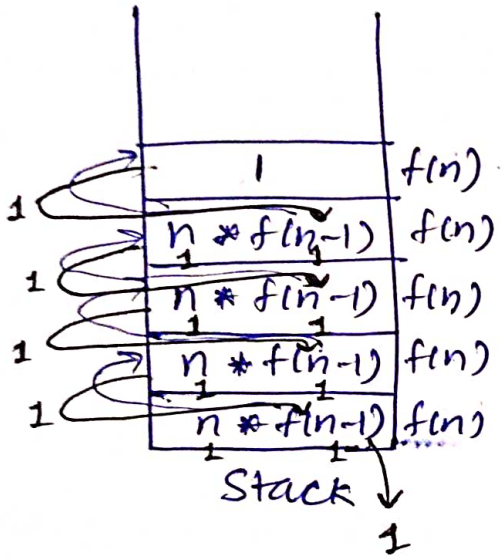


Stack storage allocation

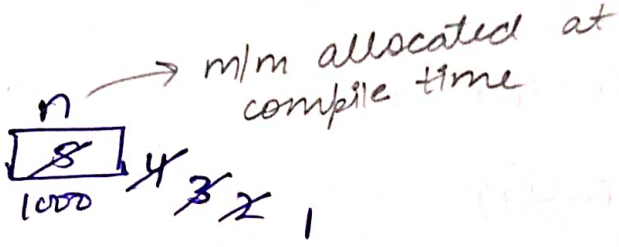
5 x 24 = 120

so, o/p : 120

```
fact(static int n)
{
  if(n<=1)
  return 1;
  else
  return (n * fact(n-1));
}
```



o/p : 1 ✗ wrong



Static storage allocation

So, Don't use static keyword in parameters of recursive program

→ **Static storage allocation :**

1) For static variable, memory is created at compile time, memory will be allocated only once.
2) For static variable, memory will be allocated in static area.
3) For static variable, binding is done at compile time. Its binding will not change at runtime.

Drawbacks of static storage allocation:

1) Recursion is not supported.
2) Dynamic data structures are not supported.
3) Size of the data must be known at compile time.

→ Stack Storage allocation

1) When a function is called, its activation record will be pushed inside stack.
2) When a function is completed, its execution then its activation record is popped out of the stack.
3) Recursion is supported.
4) Dynamic data structures are not supported.
5) Local variables belong to new activation record only always. (only top value can be accessed from stack)
6) Once func^n completes its execution then it will be popped out of stack, that means if it is needed later then, we cannot get it.

→ Heap Storage allocation

1) Allocation && deallocation will be done at anytime based on user requirement.

	C	C++	JAVA
Allocation ⇒	malloc	new	new
Deallocation ⇒	free	delete	Automatically by garbage collector

2) Recursion is supported.
3) Dynamic data structures are supported.

Intermediate code generation

eg: $x = a + b * c$

$\Downarrow$

$t_1 = b * c$
$t_2 = a + t_1$
$x = t_2$

} Intermediate code

Advantage of Intermediate code:
It is machine independent code.
So, portability exist.

⇒ **Representations of Intermediate code**

- Linear form
  - 3 address code
  - Postfix notation
- Tree form
  - Syntax tree
  - DAG (directed acyclic graph)

eg: $x = (a + (b * c)) / (a - (b * c))$

(a: ②, +: ①, (b*c): ⑤, a: ④, -: ③, (b*c): ⑤, /: ⑤) → exec sequence

3-address code: (maximum 3-address)

$t_1 = b * c$
$t_2 = a + t_1$
$t_3 = b * c$
$t_4 = a - t_3$
$t_5 = t_2 / t_4$
$x = t_5$

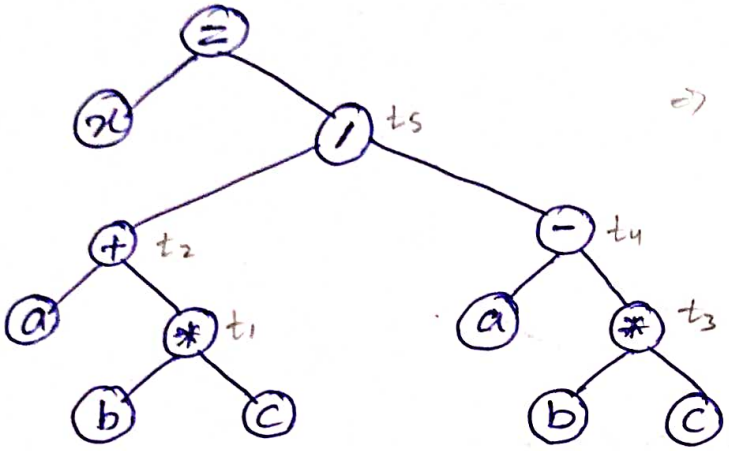
here, we calculated b*c 2 times bcz it is not minimized
but if ques ask for minimum no. of temporary variables then, do minimizations

Postfix Notation :

$a = b \Rightarrow ab =$
(infix) (postfix)

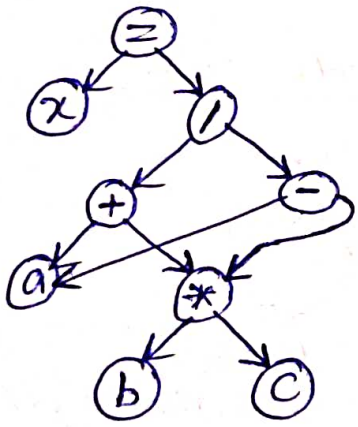
$x\ a\ b\ c\ *\ +\ a\ b\ c\ *\ -\ /\ =$
$t_1\ t_2\ \ t_3\ t_4\ \ t_5$

Syntax tree :



⇒ Here also, directions are there but we have not represented it

DAG : [Eliminating common sub expression]



In DAG,
no. of leaf nodes = no. of distinct operands

eg.2 $x = ((a+a) * (a+a)) / ((a+a) * (a+a))$

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

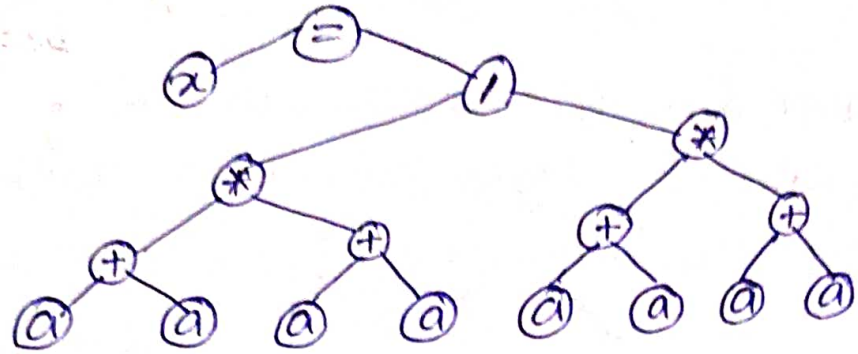
3-address code :

$t_1 = a + a$
$t_2 = a + a$
$t_3 = t_1 * t_2$
$t_4 = a + a$
$t_5 = a + a$
$t_6 = t_4 * t_5$
$t_7 = t_3 / t_6$
$x = t_7$

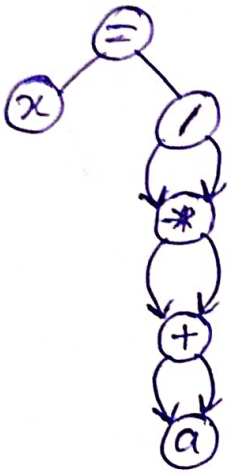
Postfix :

$x\ a\ a\ +\ a\ a\ +\ *\ a\ a\ +\ a\ a\ +\ *\ /\ =$

Syntax tree

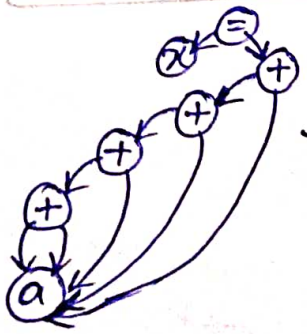


DAG :



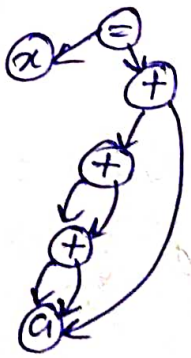
eg3 DAG? (according to C language)

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



→ acc. to C language

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



→ same meaning but it is not according to C language

Q4 Consider the following intermediate code given below:

```
a = b+c
c = a+d
d = b+c
e = d-b
a = e+b
```

What are the minimum no. of nodes & edges present in DAG for the above code?

Sol

$a = e + b$

$\Downarrow$

$a = (d - b) + b$

$\Downarrow$

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

$\Downarrow$

$a = ((b + (a+d)) - b) + b$

$\Downarrow$

$a = ((b + ((b+c)+d)) - b) + b$ → minimization bcz we need min. no. of nodes & edges

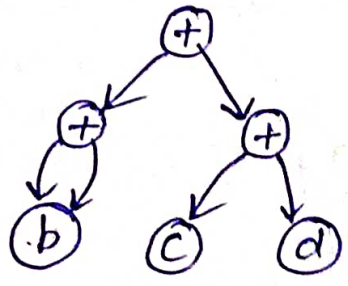
$\Downarrow$

$a = ((b+c)+d) + b$ ⟹ DAG

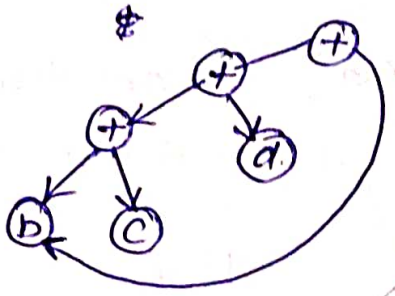
$\Downarrow$

$a = b + b + c + d$

$\Downarrow$ DAG



6 nodes
6 edges

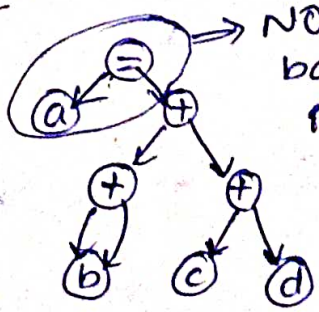


6 nodes
6 edges

(Take minimum) both have same no. of nodes & edges.

Hence, min. no. of nodes & edges required = 6, 6

Note that



NO need to write this bcz give code is intermediate code not expression. (bcz here a is also a temporary variable only)

Mention assignment operator (=) in DAG only for expression, not for intermediate code.

⇒ Types of 3-address code (max. 3 addresses)

3-address-code types:

1) $x = y \text{ op } z$
2) $x = \text{op } y$
3) $x = y$
4) goto A
5) if $(x < y)$ goto A
6) $x = A[i]$, ⇒ It has 3 addresses (x, A, i)
   $A[i] = x$
7) ~~$x = A[i][j]$~~ ⇒ It has 4 addresses
7) $x = *y$, $*y = x$ ⇒ here 3 addresses are there (x addr, y addr, content of y)
8) $x = \&y$
9) ~~$x = fun(a, b)$~~ ⇒ It has 4 addresses (x, fun, a, b)

Q1 Write 3-address code for the following expression:

if $a < b$ and $c > d$ then $e = 1$ else $e = 0$

here, 5 addresses are there a, b, c, d, e

Sol

① if $a < b$ goto ④
② $e = 0$
③ goto ⑦
④ if $c > d$ goto ⑥
⑤ goto ②
⑥ $e = 1$
⑦ Halt

Q2 Write 3-address code for for-loop in C lang.

```
for (i = 7 ; i ≤ 1000 ; i++)
{
    x = a + b * c - d;
}
```

Sol:

① $i = 7$
② if $i \leq 1000$ goto ④
③ goto ⑩
④ $t_1 = b * c$
⑤ $t_2 = a + t_1$
⑥ $t_3 = t_2 - d$
⑦ $x = t_3$
⑧ $i = i + 1$
⑨ goto ②
⑩ Halt

Backpatching :-
(while writing intermediate code, goto addresses are filled later after writing remaining code. This is called Backpatching).

Q3 Write 3-address code for switch-case in C language.

```
i = 5;
switch (i)
{
    case 1:
        x1 = a1 + b1 * c1;
        break;
    case 2:
        x2 = a2 + b2 * c2;
        break;
    default:
        x3 = a3 + b3 * c3;
}
```

Sol:

① $i = 5$
② if ($i == 1$) goto (100)
③ if ($i == 2$) goto (104)
④ $t_1 = b_3 * c_3$
⑤ $t_2 = a_3 + t_1$
⑥ $x_3 = t_2$
⑦ Halt

(100) $t_1 = b_1 * c_1$
(101) $t_2 = a_1 + t_1$
(102) $x_1 = t_2$
(103) goto ⑦
(104) $t_1 = b_2 * c_2$
(105) $t_2 = a_2 + t_1$
(106) $x_2 = t_2$
(107) goto ⑦

Q4 Write 3 address code for the following statement in C language:

int A[20][30], x
x = A[i][j] ⟶ write code for this statement only

Sol: Array element size = Integer size = 2B
Row major order, ~~Location~~

$Loc(A[i][j]) = A + [(i * 30 + j) * c]$ (c: element size)

① $t_1 = i * 30$
② $t_2 = j$
③ $t_3 = t_1 + t_2$
④ $t_4 = t_3 * 2$
⑤ $t_5 = A + t_4$
⑥ $x = *t_5$

Q5 [GATE] The least number of temporary variables required to create 3-address code in SSA form of the expression

$q + r/3 + s - t * 5 + u * v / w$ is _____

SSA ⇒ Static single assignment : Each address has single assignment

It is 3 address code in which same

$a = 2$
$a = 3$ Not possible in SSA

variable cannot have multiple meanings (values)

Sol 3-address code :

① $r = r / 3$
② $t = t * 5$
③ $u = u * v$
④ $w = u / w$
⑤ $q = q + r$
⑥ $s = q + s$
⑦ $t = s - t$
⑧ $w = t + w$

$q + \underbrace{r/3}_{r} + s - \underbrace{t * 5}_{t} + \underbrace{u * v / w}_{u \quad w}$

q, s, t, w

so, NO temp variables required

Total variables = 7 (q, r, s, t, u, w)

SSA

① $r_1 = r / 3$
② $t_1 = t * 5$
③ $u_1 = u * v$
④ $w_1 = u_1 / w$
⑤ $q_1 = q + r_1$
⑥ $s_1 = q_1 + s$
⑦ $t_2 = s_1 - t_1$
⑧ $w_2 = t_2 + w_1$

so, 8 temp variables required ($r_1, t_1, u_1, w_1, q_1, s_1, t_2, w_2$)

Total variables = $8 + 7 = 15$

Q6 Consider the following code:

$t_1 = a * a$
$t_2 = a * b$
$t_3 = t_1 * t_2$
$t_4 = t_3 + t_2$

what is the minimum no. of total variables required to represent above code in 3 address code? and SSA

Sol:

$t_4 = t_3 + t_2$

$= (t_1 * t_2) + t_2$

$= (t_1 * (a * b)) + (a * b)$

$t_4 = ((a * a) * (a * b)) + (a * b)$

3 address code:

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

$((\underbrace{(a * a)}_{a} * \underbrace{(a+b)}_{b}) + \underbrace{(a*b)}_{b})$ — outer a

No. of New temp variables = 0
Total temp variables = 2 (a, b)

SSA

$b_1 = a * b$
$a_1 = a * a$
$a_2 = a_1 * b_1$
$a_3 = a_2 + b_1$

No. of new temp variables = 4
Total temp variables = 6

Q7 Consider the following intermediate code in 3-address code form

$p = a - b$
$q = p * c$
$p = u * v$
$q = p + q$

which is the correct SSA form of the above?

a) $p_1 = a - b$
$q_1 = u * v$
$p_1 = u * v$
$q_1 = p_1 + q_1$

b) $p_3 = a - b$
$q_4 = p_3 * c$
$p_4 = u * v$
$q_5 = p_4 * q_4$

c) $p_1 = a - b$
$q_1 = p_2 * c$
$p_3 = u * v$
$q_2 = p_4 * q_3$

d) $p_1 = a - b$
$q_1 = p * c$
$p_2 = u * v$
$q_2 = p + q$

⇒ Implementations of 3-address code

how 3-address code are stored inside computer

1) Quadruples
2) Triples
3) Indirect triples

eg. $a = b * -c + b * -c$

(unary minus)

Quadruples

S.No.	operator	opnd 1	opnd 2	result
1	-	c		t_1
2	*	b	t_1	t_2
3	-	c		t_3
4	*	b	t_3	t_4
5	+	t_2	t_4	t_5
6	=	t_5		a

Triples

S.No.	operator	opnd 1	opnd 2
1	-	c	
2	*	b	(1)
3	-	c	
4	*	b	(3)
5	+	(2)	(4)
6	=	a	(5)

Indirect triples

S.No.	operator	opnd 1	opnd 2
1	-	c	
2	*	b	(1)
3	-	c	
4	*	b	(3)
5	+	(2)	(4)
6	=	a	(5)

Array of pointers

(1)	⇒ (14)
(2)	⇒ (5000)
(3)	⇒ (9)
(4)	
(5)	
(6)	

In Indirect triple, after completion of triple just move necessary things to necessary location

⇒ Control flow graph

Purpose of control flow graph is to check whether cycle exist or not in given 3-address code.

→ How to find leaders?

1) First statement is a leader.
2) Target of goto is a leader.
3) Next statement after goto is a leader.

→ Basic blocks?

definition: It is set of 3-address statements where control enters at the begin and leaves at the end. In between basic block, there is no-halt (or) no-jump.

→ How to find basic blocks?

From one leader upto just before next leader, one basic block is there.

- No. of basic blocks = No. of leaders

eg: header

L_1 ① $x = a+b$
② $t_1 = x+c$ — B_1

L_2 ③ $t_2 = a+t_1$
④ if $t_2 > t_1$ goto 9 — B_2

L_7 ⑤ $t_3 = t_1 + t_2$
⑥ $t_5 = t_3 - t_2$
⑦ goto 12 — B_3

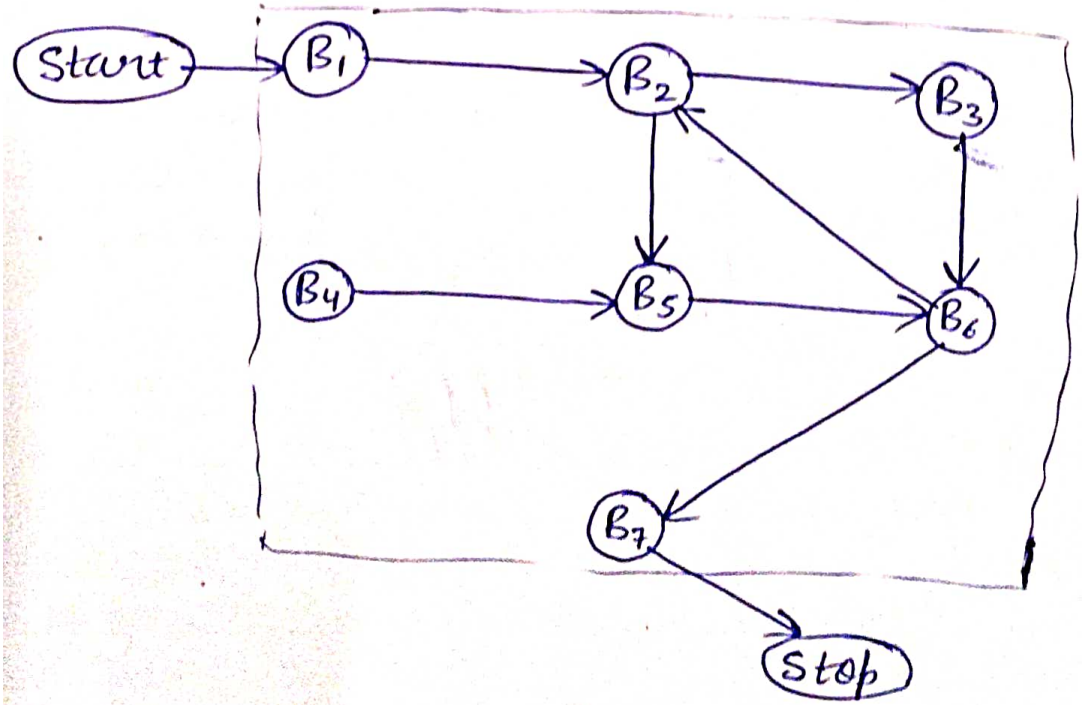
L_5 ⑧ $t_6 = t_5 / t_2$ — B_4

L_3 ⑨ $t_3 = t_1 - t_2$
⑩ $a = b+c$
⑪ $d = b-c$ — B_5

L_4 ⑫ $t_6 = t_4 + t_9$
⑬ if $t_6 > t_7$ goto 3 — B_6

L_6 ⑭ $t_4 = t_7 + t_5$ — B_7

control flow graph (CFG)



In this CFG,
there are 7+2 = 9 nodes and 10 edges

DFS : S B_1 B_2 B_5 B_6 ... cycle found

Code Optimization

machine independent optimization

1) Loop invarient
2) Loop unrolling
3) Loop Jamming
4) Induction variable elimination

(1, 2, 3, 4 — for loops only)

5) Strength reduction
6) Algebraic Simplification
7) Constant folding
8) Copy propagation
9) Deadcode elimination
10) common subexpression elimination
11) Constant propagation

⇓

done on intermediate code

machine dependent optimization

1) Peephole optimization
   (i) Redundant load and store instn elimination
   (ii) use of machine idioms.
   (iii) Strength reduction
   (iv) Algebraic Simplification
   (v) Removal of unreachable code
   (vi) Flow control optimization
   (vii) Constant folding
2) Register allocation

⇓

done on target code

☆ Machine Independent optimization

⇒ Loop Invarient (Code motion)

Eg:

```
z = 2 , x = 10, y = 20, a = 5
for (i=1 ; i<=n ; i++)
{
    z = x + y * a;      ⇒ not changing in loop
    bf(hi);               so, move it out of loop
}                         (either before or after)
```

⇓

```
z = 2 , x = 10, y = 20, a = 5
for (i=1 ; i<=n ; i++)
{
    bf(hi);
}
z = x + y * a;
```

⇒ Loop unrolling (or) decreasing comparisons

Eg:

```
for (i = 1; i<=n; i++)
{
    a[i] = i;
}
```

⇓

```
for(i=1 ; i<=n ; i++)
{
    a[i] = i;
    i++;
    a[i] = i;
}
```

⇒ Here, no. of comparisons $i \le n$ are reduced to half.

⇒ Loop Jamming (or loop combine

a=1, b=2

eg: for (i=1; i≤n; i++)
{
a = a+b;
}

x=10, y=15, z=20

for (i=1; i≤n; i++)
{
x += y+z;
}

⇓

a=1, b=2
x=10, y=15, z=20
for (i=1; i≤n; i++)
{
a = a+b;
x += y+z;
}

⇒ Here no. of comparisons & incrementation are reduced

⇒ Induction variable elimination :

eg x=10, y=20, j=4, i=1
while (i≤100)
{
x += y*j;
i = i+1;
j = j+4;
}

⇓

x=10, y=20, j=4
while (j≤400)
{
x += y*j;
j = j+4;
}

⇒ **Strength reduction** : replace costly operator by less cost operator

eg $x^2 \Rightarrow x * x$

$16 * x \Rightarrow$ 4 times left shift

$2 * i \Rightarrow i + i$

⇒ **Algebraic simplification**

eg:

$x + 0 \Rightarrow x$	$x - 0 \Rightarrow x$
$0 + x \Rightarrow x$	b || true $\Rightarrow$ true
$1 * x \Rightarrow x$	b || false $\Rightarrow$ b
$x * 1 \Rightarrow x$	b && true $\Rightarrow$ b
$0 / x \Rightarrow 0$	b && false $\Rightarrow$ false

⇒ **constant folding**

eg: $10 + 2 * 3 \Rightarrow 16$

if $(10 < 20)$ goto L_1 else goto L_2 $\Rightarrow$ goto L_1

⇒ **copy propogation**

eg:

$t_2 = t_1$
$t_3 = t_2 * t_1$
$t_4 = t_3$
$t_5 = t_3 * t_2$
$a1 = t_5 * t_4$

$\Longrightarrow$

$t_3 = t_1 * t_1$
$t_5 = t_3 * t_1$
$a1 = t_5 * t_3$

⇒ **Constant propogation**

eg: $P_i = 360$, $y = 180$

$x = P_i / y$

$\Downarrow$ ← using constant propogation solution

$x = 360 / 180 \Longrightarrow x = 2$ (using constant folding)

⇒ dead code elimination

eg.1
$t_2 = 10$, $t_3 = 30$ } dead code
$t_1 = t_2 * t_3$ }
$t_5 = 60$, $t_6 = 70$
$t_1 = t_5 * t_6$
pf(t_1)

⟹

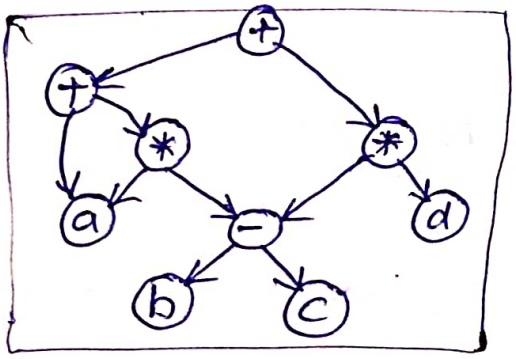
$t_5 = 60$, $t_6 = 70$
$t_1 = t_5 * t_6$
pf(t_1)

eg.2
t_1 = display() } ⇒ we cannot delete this blindly bcz it may be possible some important work is happening in display() func^n
$t_1 = 20$
pf(t_1)

So, No dead code elimination in this

⇒ common subexpression elimination (using DAG)

eg. $a + a * (b-c) + (b-c) * d$



$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = t_1 * d$
$t_4 = a + t_2$
$t_5 = t_4 + t_3$

★ Machine dependent optimization

⇒ Peephole optimization

i) Redundant load & store instrn elimination :

eg.1
```
Load  X, R0
Store R0, X   ⇒  Load X, R0
```

eg.2
```
Load  X, R0
Load  X, R0   ⇒  Load X, R0
```

eg.3
```
Store R0, X   ⇒  Store R0, X
Load  X, R0
```

eg.4
```
Store R0, X   ⇒  Store R0, X
Store R0, X
```

ii) Use of machine idioms : (instrn)

eg.
```
MOV  i, R0
MOV  1, R1   ⇒  INC i
ADD  R0, R1
MOV  R0, i
```

iii) Strength reduction

eg:
- 2 * i ⇒ left shift
- 9/2 ⇒ right shift

iv) Algebraic simplification

same as previous

(v) Removal of unreachable code :

eg:

```
#define debug 0
if (debug)
    pf (hi)
if (!debug)
    pf (bye)
```

unreachable code ⟹

```
#define debug 0
if (!debug)
    pf (bye)
```

(vi) Flow control optimization :

eg:

```
L1 : goto L2
     :
L2 : goto L3
     :
L3 : goto L4
     :
L4 :
```

⟹

```
L1 : goto L4
     :
L4 :
```

(vii) Constant folding

same as previous

⇒ Register allocation

1) Assign multiple variables to single register without changing the program behaviour.

2) X is live variable at statement S_i iff
   i) There is statement at S_j reads x.
   ii) There is a path from S_i to S_j.
   iii) No new defination to x before S_j. here, $j=i$ is possible

3) 2 temporary variables that are live can't be allocated in the same register simultaneously.

4) Less registers, more variables so, we have to keep some variables in m/m temporarily, it is called Spilling.
(main m/m)

(126) eg:

① $x = a+b$

② $y = d+c$

③ $x = x+b$

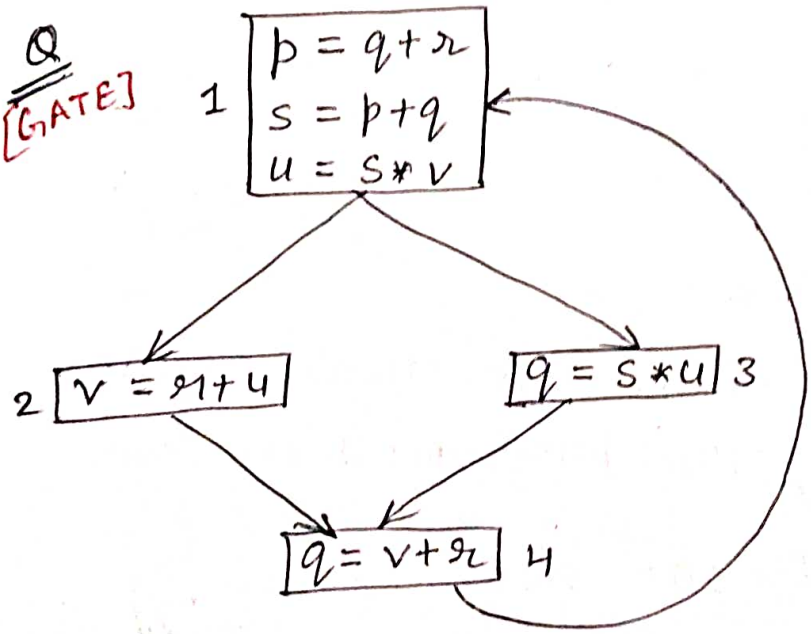
④ $a = b+d$

⑤ $x = a+b+c$ → live

→ dead (not live)

	a	b	c	d	x	y
1	L	L	L	L	D	D
2	D	L	L	L	L	D
3	D	L	L	L	L	D
4	D	L	L	L	D	D
5	L	L	L	D	D	D

Q [GATE]



What are the live variables at basic blocks 2 & 3? (Statement)

	p	q	r	s	u	v
2	D	D	L	D	L	D
3	D	D	L	L	L	L

Both statement 2 & 3 have live r & u.

Q
```
a = 1
b = 10
c = 20
d = a+b
e = c+d
f = c+e
b = c+e
e = b+f
d = 5+e
return (d+f)
```

Assuming all operations take their operands from registers, what is the minimum number of registers needed to execute this program without spilling?

Sol

$a = 1$ (R_1)

$b = 10$ (R_2)

$c = 20$ (R_3)

$d = a + b$ (R_1) (R_1) (R_2)

$e = c + d$ (R_1) (R_3) (R_1)

$f = c + e$ (R_2) (R_3) (R_1)

$b = c + e$ (R_1) (R_3) (R_1)

$e = b + f$ (R_3) (R_1) (R_2)

$d = 5 + e$ (R_1) (R_1) (R_3)

If variable is live ⇒ don't replace it with another value

If variable is not live ⇒ can replace it with another value

So, Minimum No. of registers needed = 3 (R_1, R_2, R_3)

★ CFG minimization

Steps:-

1) Eliminate null productions
2) Eliminate unit productions
3) Eliminate useless symbols
   a) Eliminate unreachable variables.
   b) Eliminate unnecessary productions.

These are done in this order only

eg.

$S \rightarrow AB$
$A \rightarrow \epsilon / a$
$B \rightarrow \epsilon | b$

① Eliminate ϵ productions

Replace A, B by ϵ bcz $A \rightarrow \epsilon$, $B \rightarrow \epsilon$

$S \rightarrow AB | B | A$
$A \rightarrow a$
$B \rightarrow b$

② Eliminate unit productions

$S \rightarrow AB | b | a$
$A \rightarrow a$
$B \rightarrow b$

③ Eliminate unreachable variable

Here, NO unreachable variable

④ Eliminate unnecessary productions

Here, NO unnecessary production.

So, minimized CFG is:

$S \rightarrow AB | b | a$
$A \rightarrow a$
$B \rightarrow b$

Q17 & 18 of WB

$S \rightarrow AB \mid CA$

$B \rightarrow BC \mid AB$ → here recursion is there but no termination so, these are useless production

$A \rightarrow a$

$C \rightarrow aB \mid b$

So, remove B variable productions from everywhere

① NO ∈ production

② NO unit production

③ NO unreachable variable

④

~~$S \rightarrow AB \mid CA$~~

~~$B \rightarrow BC \mid AB$~~

$A \rightarrow a$

~~$C \rightarrow aB \mid b$~~

$\Rightarrow$

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$