• Compiler Design

Weightage: (5-8 marks)

textbook : Dragon edition.

1. Introduction.
2. Lexical Analysis.
*** 3. Syntax Analysis. (3-4 mrks).
4. Syntax Directed Translations (SDTs).
5. Intermediate Code & Code optimization.
** .

→ Compiler :- Translator.

HLL (C code). → [Compiler] → [Assembly code] m/c code Object code.

↓ Compilation Errors.

- → It is translator.
- → It is program.
- → It is executable.

High level program (m/c independent code) → [Compiler. (Executable code)] (m/c dependent code) → Low-level program. (m/c dependent code).

Portable.	Not-Portable
m/c Independent.	m/c dependent.
ex:- HLL, Intermediate codes.	ex:- Compiler, Assembly code, m/c code.

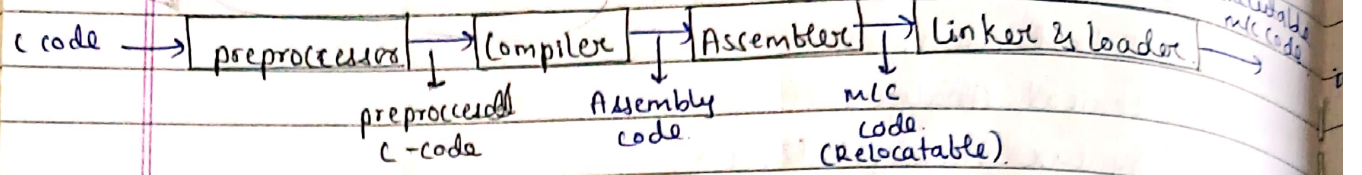
• Translator

Source. (Input) → [Translator.] → Target (output).

Page No.
Date

1:10:20

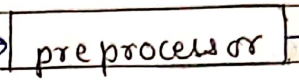
- C language Translation -



Preprocessor -

#include<....>
#define<....>

#define MAX 10
int a=MAX;

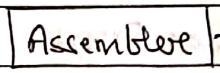


int a=10;

Assembler -

x=10;

Assembly code.
MOV A, 10

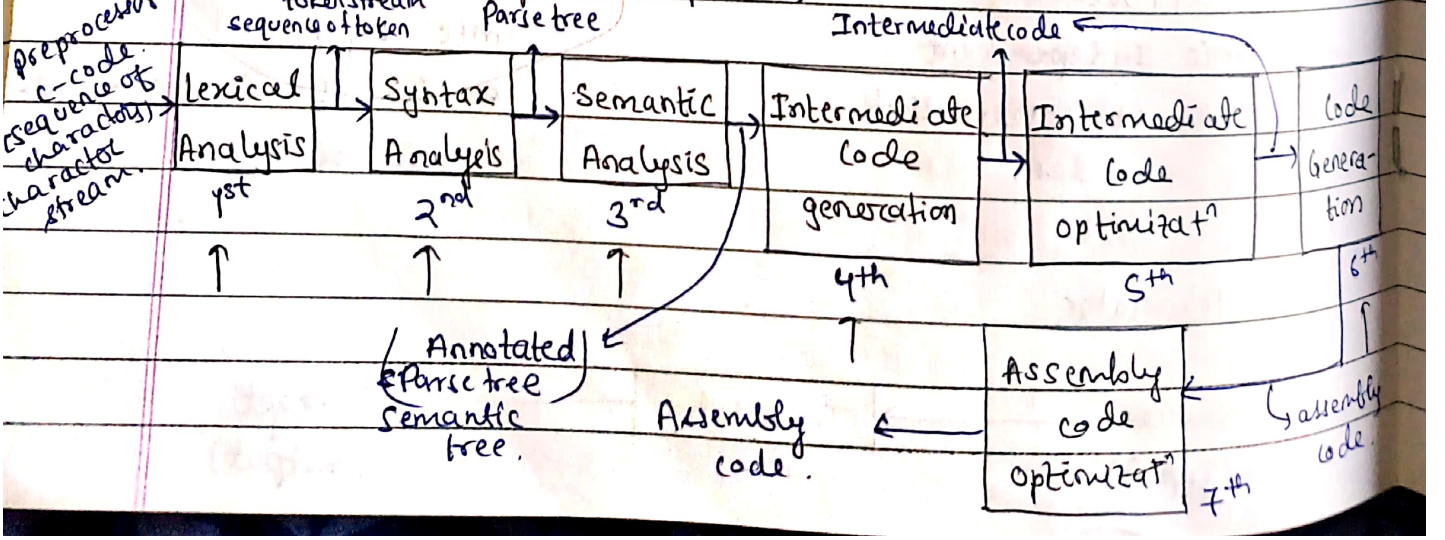


Relocatable m/c code.

Linker - To resolve all external references
(all external files)
(all external variables & functions)

Loader - It performs "relocation"
↳ Altering address of code/Data

* Phases of a compiler - (7 phases)



m/c independent phases:
- Lexical analysis
- Syntax analysis
- Semantic analysis

} Analyzers (code not changing just analyzing)

- Inter. code generator.
- Inter. code optimizat^n

m/c dependent:
- Code Generation.
- Assembly code optimizat^n

} Synthesizers (generating code, changing it)

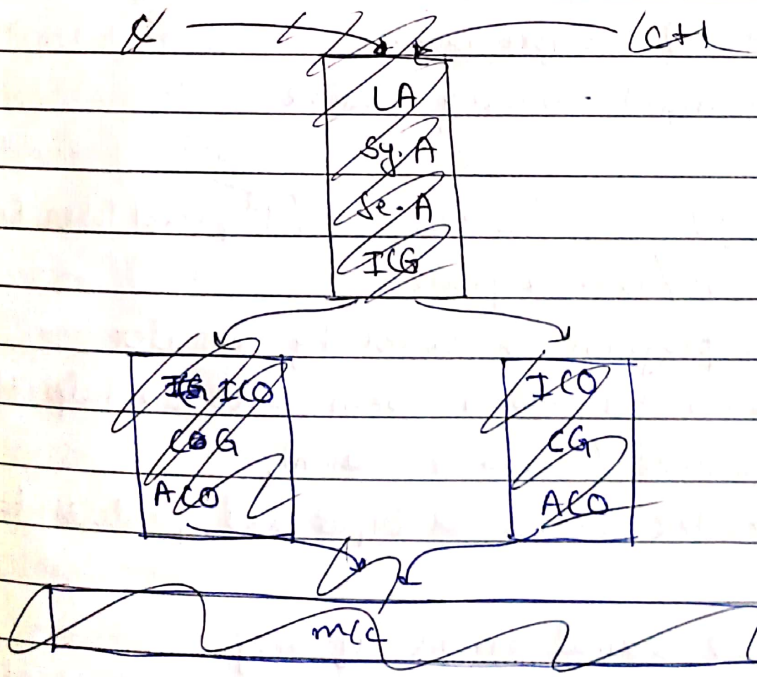
C code - C
I code - Intermediate
→ m/c independent.

A code - Assembly
M code - m/c.
E code - Executable.
→ m/c dependent.

- ALL 7 phases can interact with 1 data structure i.e. Symbol Table.

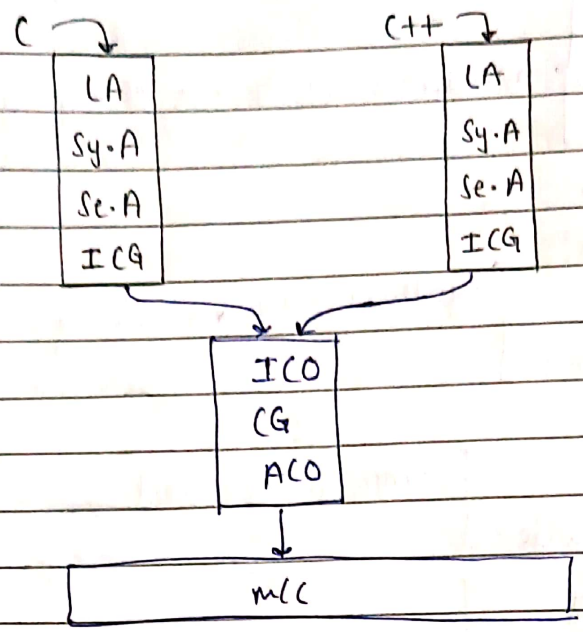
→ Symbol table is used in all the phases, throughout the process of compilation.

Cost of one compiler = 7 (i.e. 7 phases)



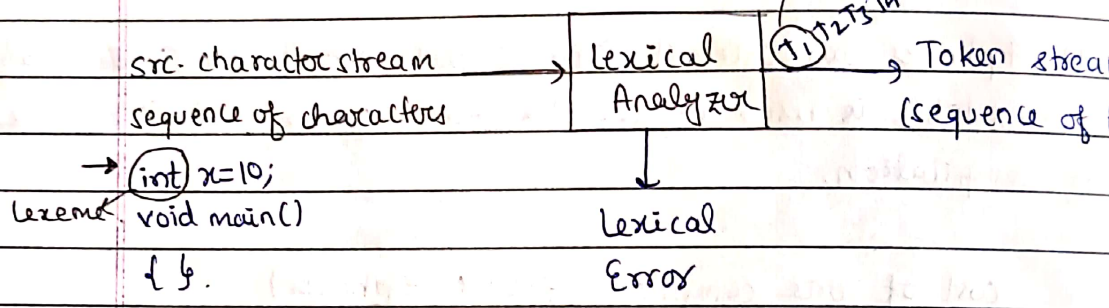
"File no need to have main() for compilation"

- Execution begins from ~~begin~~ main().
- Compilation begins from 1st character of file & only ends at the last character of file.



= 11 phases.

★ Lexical Analysis -



- Compilation (static) - Before runtime
- Execution (Dynamic). - During runtime.

- Lexical Analysis / Scanner / Linear phase / 1st phase / Token Recognizer / token generator / Lexical phase.
  - It scans whole program character by character.
  - It groups characters into a "Token" with the help of "longest prefix Rule" (Maximal Munch).
  - It takes character stream as input and produces token stream as output.
  - It may produce lexical errors if any.
  - It recognizes tokens, whitespaces, and comments & also newline character
  - It produces only tokens.

* It may take help of parser to separate tokens.
* It also uses/creates symbol table.

- symbol table-

Literal table
10

```
void fun(){
    int x = 10;      ↳ literal.
    char y;
}
```

Lexeme	token	Line	type	← symbol table.
fun	id_1	1		
x	id_2	2	int	
y	id_3	3	char.	

* It is a data structure
* It stores attribute information of each identifier.

1) Lexeme : It's smallest unit of program. (logic) int x;
2) Token : It represents lexemes. (Internal representation of lexeme.)
3) Types of token.
4) Finding tokens.
5) Finding lexical Errors.

Lexeme	token.
int	T_1
x	T_2
;	T_3

- Types of Tokens - (KISCO)
1). Identifiers
2). Keywords.
3). Operators
4). Literals/constants.
5). Special.

1) Identifier - Name of variable/function.
i). Contains only letters, digits, underscore.
ii). Should not start with digit.
iii). should not be a keyword.
ex. x, x123, Int, int1 ✓
1x23X, int X

→ 32 keywords [Reserved words]

3) Keywords : int, float, char, double, short, long, void, auto, static, register, extern, signed, unsigned, const, volatile, struct, union, enum, typedef, if, else, switch, case, default, break, while, do, for, continue, return, goto, sizeof.

3) Operators : i) Arithmetic (→ +, -, *, /, %)
ii) Relational → (<, <=, >, >=)
iii) Logical → (&&, ||, !)
iv) Bitwise → (&, |, ~, ^, <<, >>)
v) Unary → (+, -, ++, --, &, *, (type), !, ~, sizeof)
vi) Assignment → (=, +=, -=, /=, *=, %=, <<=, >>=)
vii) Special operators → (•, ,, →)

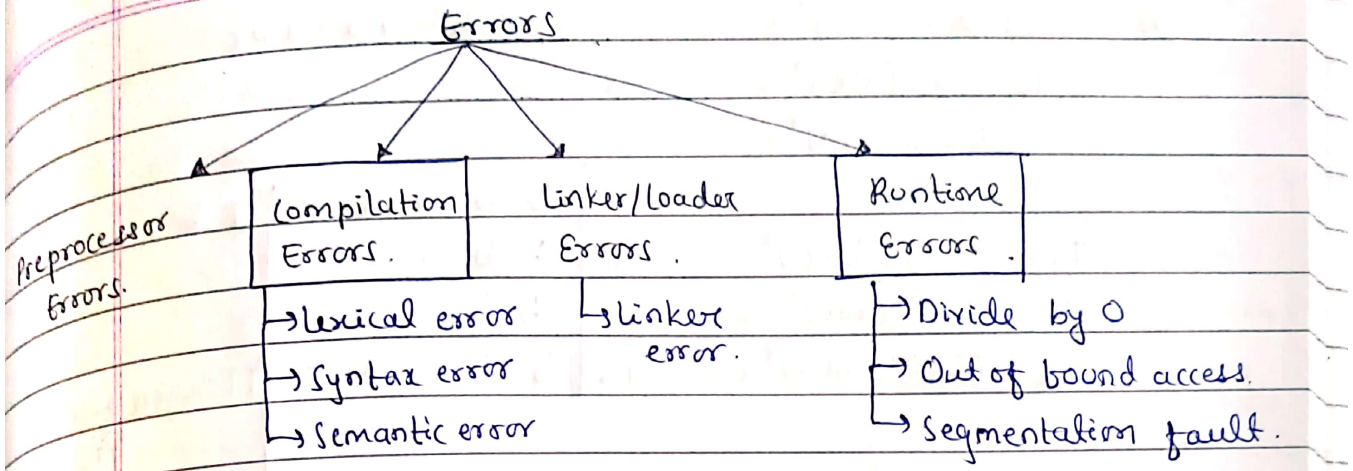
4) Literals/constants :
Integer: → Decimal : 2391 / Hexa : 0x2ad3 / → octal : 0237 (0 to 7)
Character: 'a' '\n' 'A'
Real : (float/double) : 23.64f, 23.64, 10e-2.
String : "gate"

5) Special tokens : [,], (,), {, }, ;

NOTE:- Whitespaces & comments are not tokens.

* find the tokens -

① int x=10; ⑤ {No error}
② int x,y; ⑤ {No error}
③ int x=y=z; ⑦ {semantic error, coz y,z not declared} → compilation error.
④ int float,char; ⑤ {Syntax error → Compilation error}
⑤ x = "gate"; ④ {semantic error → x not declared}
⑥ int x="gate"; ⑤ {Semantic error} coz assigning addr. to int, size of addr. can be greater than size of integer.



(7) int *p; (4) No error

(8) int *p=1000; (6) {semantic error → p expecting address, u are giving integer}

(9) int *p=(int*)1000; (10) {No error}

(10) void main(){

printf("x=%d",x); (13) {semantic error → x not declared}

}

(11) ~~void~~ ~~x=a++++++=**b;~~ x=a++++++==**b; (11)

(12) int ***p; (5)

(13) x++; (2) (3).

(14) int 12xy; → lexical error {no meaning of 12x..}

(15) int x=029; → 0-octal, 2-octal, 9-not octal, so lexical error.

(16) int x=0XAX23; AX23 is not hexa coz of X, so lexical error.

(17) in t x; (4) tokens

(18) intx; (2) tokens

(19) in /*comment*/t x; in t x; (4) tokens.

(20) in/*comment t x; lexical error → comment not ended.

(21) int(/*comment*/*t x; } int a; (3) tokens → No error.

(22) int /*xyz*/a;

(23) x===+=<<=>>===y; (9) tokens.

GATE-2018

(23) T_1: a ? (b|c)* a i/p = b b a a c a b c.

T_2: b ? (a|c)* b

T_3: c ? (b|a)* c.

$T_1 = (\varepsilon + a)(b+c)^* a$.

$T_2 = (\varepsilon + b)(a+c)^* b$

$T_3 = (\varepsilon + c)(b+a)^* c$.

	b b a a c	a b c
T_1	b b a.	
T_2	b b.	
T_3	b b a a c	} Longest from (T_3)

bb aac abc i/p → | T_1 / T_2 / T_3 } in LA | → T_3 T_3

	a b c.	
T_1	a	
T_2	a b	
T_3	a b c	(T_3).

(24) $T_1 = a^* b c$ i/p = a a b c b c b b a c

$T_2 = b^* a c$

T_1	a a b c	b c	b
T_2	a	b	b b a c.

o/p → T_1 T_1 T_2

Q Which of the following is used in lexical Analyzer?

(A) Regular expression

(B) Finite Automata.

(C) Regular Grammar.

(D) None.

(A) (B), (C)

* Syntax Analysis :-

- Basics
  - Syntax error
  - CFG → Types, elimination of left Recursion.
  - Left factoring.
  - first & follow computation (*)
- Topics
  - Top-down parser : LL(1)
  - Bottom-up parser
    - LR parsers
    - Operator precedence parser

Find syntax Errors :-

(1) int x; — No syntax error & No Error.

(2) int x, float y; — syntax error (its expecting identifier after ,)

(3) x, y; — No syntax error but semantic error (x, y not declared)

(4) x==y; — No syntax error but semantic error (x, y not declared)

(5) if(); — syntax error, condition/expression missing

(6) while(); — syntax error, condition/expression missing

(7) for(;;); — No syntax error, No semantic error, ∞ loop.

*** (8) x y; — No syntax error, 2 Semantic error exist.

(9) if(23); — No error.

(10) typedef int x; — ≅ int y; No error at all.
x y;

Q. (a-2022)
int main() {
Integer x;
return 0;
}

(A) lexical Analysis
(B) Syntax Analysis
(C) semantic Analysis.
(D) m/c dependent optimizes. (C)

(12) void main(— → Syntax Error.
{
}

(13) for(x,y,z); — → syntax error, for has fixed syntax (missed ;)

*** (14) ~~fro(x,y,z);~~
void main(){ — → semantic error, fro not defined.
fro(2,3,4);
}

(15) void main(){
for(2;3;4); — ~~→ syntax err~~ No error, runtime - ∞ loop.
}

(16)
```
#include <stdio.h>
void main() {
    fro(2,3,4);
}
```
Linker error
fro Not in #stdio.h.

(17)
```
#include <p2.c>
void main() {
    fro(2,3,4);
}
```
compile later.

P2.C file
```
void fro(int x, int y, int z) {
    int a;
    a=x+y+z;
}
```
1st compile this.

No error.

☺

★ Context Free Grammar:-

Grammar (G) = (V, T, P, S). ↳ start symbol.

↳ To represent a language.

↳ It is set of rules. P → set of productions.

T → ~~terminals~~ set of terminals

set of characters (lexical analysis).

set of tokens (syntax analysis)

V → set of variables (Non-terminal)

CFG = (V, T, P, S).
V → Any
V → (VUT)*

• Why CFG?

→ It is most suitable grammar to represent syntax of most programming languages.

```
void main()
{
    int x;
}.
```

Program → R main() B

R → void | int

B → { D; }

D → T Id

T → int | float.

V={program, R, B, D, T

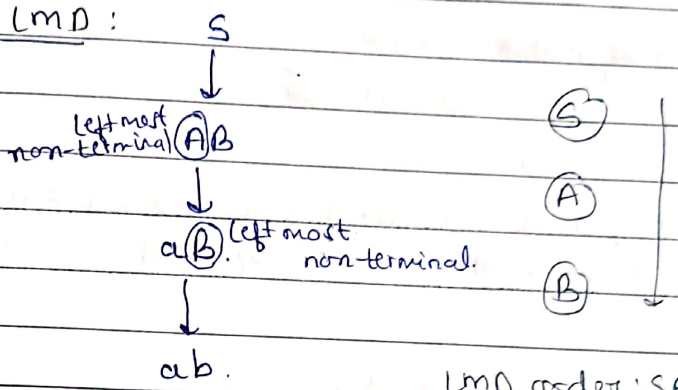
T={main, (,), void, int, ;, id, float}

- **Derivations of a string:-**

1) Left most Derivation (LMD). } Linear.
2) Right most Derivation (RMD). }
3) Parse tree. } → Non linear.
   ↳ Derivation tree.

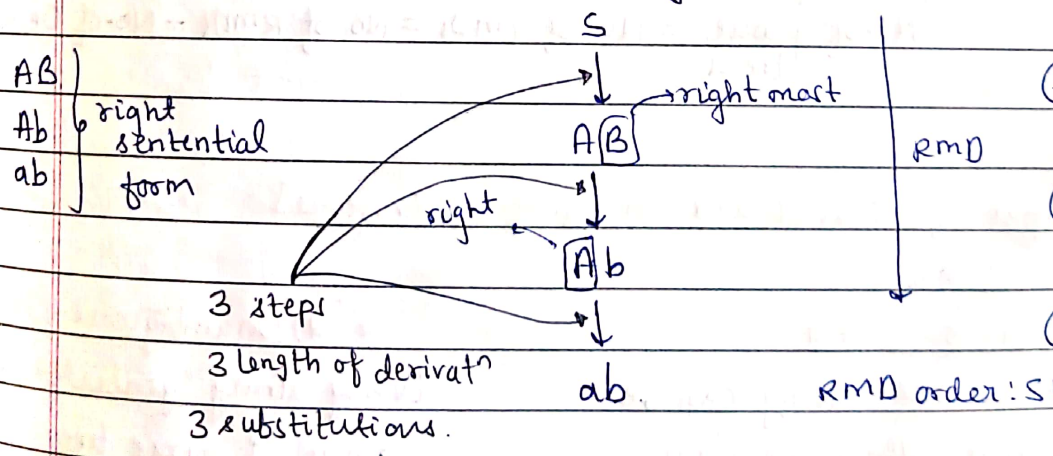
$S \to AB$
$A \to a$
$B \to b$.

LMD :



LMD order : SAB.

- **Sentential form:** Any sequence of symbols derived from START.
  S, AB, aB, ab } Left sentential form.
- **LMD:** In every sentential form, left most non-terminal is substituted to derive a string.
- **RMD:** In every sentential form, right most non-terminal is substituted to derive a string.

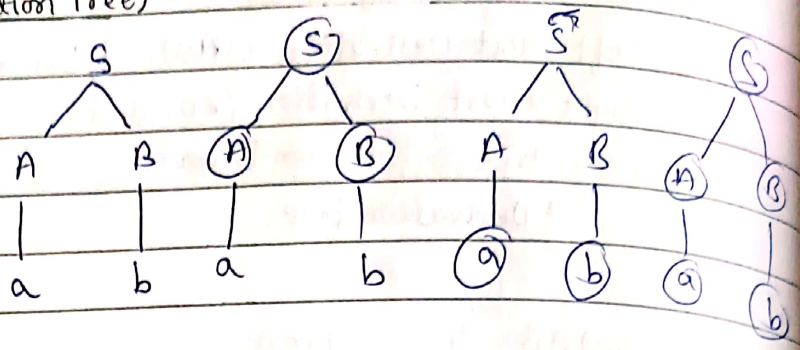


RMD order : SBA.

→ for a string : LMD & RMD need not be same.

- **Parse Tree (Derivation Tree) -**

$S \to AB$
$A \to a$
$B \to b$

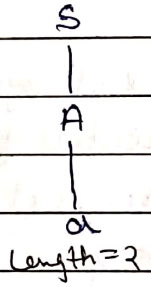


leaf nodes - a, b
non leaf nodes - A, B, S.
root - S.

NOTE
- Length of derivation = No. of substitutions in LMD/RMD = no. of non-leaf nodes in parse tree = No. of steps

No. of Parse trees?

$S \to A|a$
$A \to a|b.$
$w = a$



2 Parse trees
2 LMDs
2 RMDs.

Length = 1.
Length = 2

★ For any string:

No. of parse trees = No. of LMDs = No. of RMDs = No. of Derivations

- **Types of CFGs [Based on no. of derivations]**

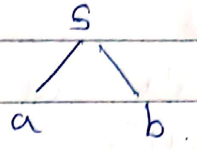
1) Ambiguous CFG.
Atleast one string can be derived with more than one parse tree.

2) Unambiguous CFG
Every string generated has only 1 parse tree.
(1 derivation)
(1 LMD)
(1 RMD).

① $S \to ab$.

ab : 1 Parse tree

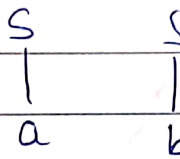
Unambiguous CFG.



② $S \to a \mid b$.

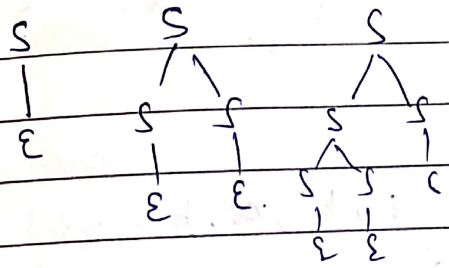
$a \to 1$ PT

$b \to 1$ PT



unambiguous

③ $S \to SS \mid \varepsilon$

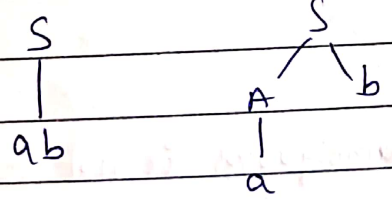


---- infinite ways to generate ε.

∴ ~~Un~~Ambiguous.

④ $S \to Ab \mid ab \mid \varepsilon$ ε, ab, b, bb.

$A \to \varepsilon \mid a \mid b$.

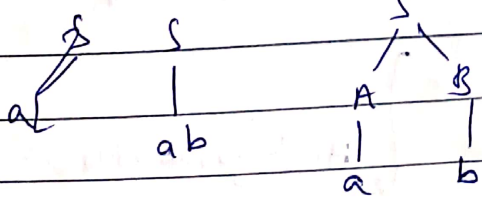


→ ab ke liye 2 parse trees, so its Ambiguous.

⑤ $S \to AB \mid ab$ ab, a^+b^+

$A \to Aa \mid a$ (a^+)

$B \to Bb \mid b$. (b^+)

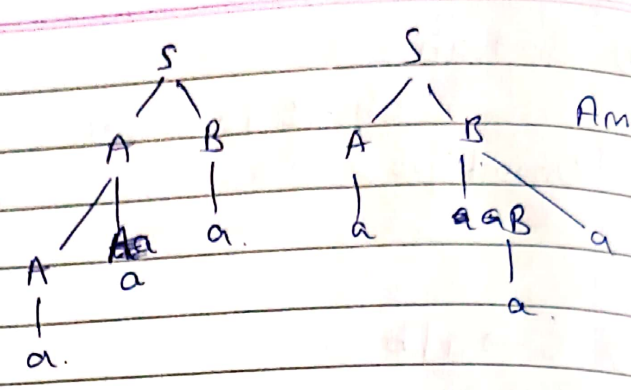


ambiguous.

⑥ $S \rightarrow AB$

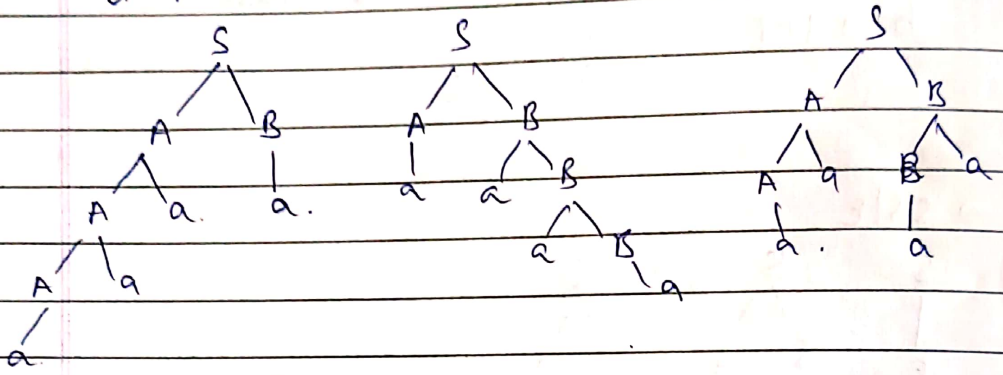
$A \rightarrow Aa|a$ (a^+)

$B \rightarrow Ba|a$ (a^+)



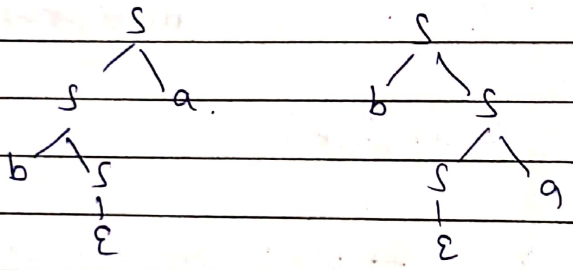
Ambiguous

a^4 ?



→ 3 parse trees

⑦ $S \rightarrow Sa|bS|\varepsilon$ $\quad \varepsilon, b^+, a^+, b \ldots$



ambiguous.

NOTE:- Checking Ambiguous & Unambiguous is not having algorithm.

$S \rightarrow Sa|bS|\varepsilon$ (right recursion, left recursion)

→ If same non terminal is having both Left & Right recursion, then CFG is definitely AMBIGOUS.

⑨ $S \rightarrow SaS|\varepsilon$

⑩ $S \rightarrow SS|a$.

⑪ $S \rightarrow Ab$

$A \rightarrow Aa|bA|c$

} Ambiguous CFG.

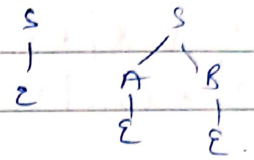
(12) $S \to (S) \mid \varepsilon$ → unambi.

(13) $S \to aSb \mid \varepsilon$ → unambi

(14) $S \to AB \mid \varepsilon$ → ambiguous

$A \to aA \mid \varepsilon$

$B \to bB \mid \varepsilon$



(15) $S \to AB$ a^+b^+

$A \to aA \mid \varepsilon$ ↳ unambi.

$B \to bB \mid \varepsilon$

(16) $E \to E+E \mid a$ → Ambiguous (by shortcut)

(17) $E \to E+E \mid E*E \mid E-E \mid E/E \mid a$.

* Elimination of Left Recursion -
  - ↳ Top down parser not possible if CFG has Left Recursion.
  - ↳ Conversion from Left recursion to right recursion.

(1) $S \to Sa \mid b$ → $L = ba^*$

Left rec. / non left rec

$S \to bX$

$X \to aX \mid \varepsilon$ } Right recursion.

$S \to bX$ OR $S \to bX \mid b$

$X \to aX \mid \varepsilon$ $X \to aX \mid a$.

(2) $S \to Sa \mid Sb \mid cd \mid es$

Left rec. non left

$S \to cX \mid dX \mid esX$	} Right recursion.
$X \to aX \mid bX \mid \varepsilon$	

NOTE:- $A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_k \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

k Left rec. term. n non left rec. terms.

~~$A \to \beta_1 \mid \beta_2$~~	$A \to \beta_1 X \mid \beta_2 X \mid \dots \mid \beta_n X$
	$X \to \alpha_1 X \mid \alpha_2 X \mid \dots \mid \alpha_k X \mid \varepsilon$

③ $S \to Sab \mid cdS \mid Se \mid f \mid gh$

$S \to cdSX \mid fX \mid ghX$
$X \to abX \mid eX \mid \varepsilon$

⑤ ~~$S \to Sa \mid Abc \mid efg$~~
~~$A \to Sb \mid f$~~

④ $S \to \underbrace{Sa}_{\text{direct left rec.}} \mid \underbrace{Abc}_{\text{indirect left rec.}} \mid efg$
$A \to \underbrace{Sb}_{\text{indirect left rec.}} \mid f$

Direct Left Rec. ⇓ Visible.

Indirect Left Rec. ⇓ Substitute using Algo (only 1st place in RHS) ⇓ Direct Left Rec.

	Algo1	Algo2	Algo3
S →	S	S	A
A →	A	A	B
B →	B	C	S
C →	C	B	C

we follow this one

4! algorithm. 2 answers.

Algo1.

S step1: Solve S using direct left Recursion elimination.
A step2: Substitute S in A rules then apply Direct left Rec. elimination
B step3: Substitute both S & A in B rules recursively, then do direct left rec.
C step4: Substitute S, A, & B in C productions recursively then apply direct left rec. elimination.

⑥ $S \to Sa \mid Abc \mid efg$
$A \to Sb \mid f$

step 1: choose S,

$S \to Sa \mid Abc \mid efg$

Left rec. — Assume non-left rec.

$S \to Abc X \mid efg X$

$X \to aX \mid \varepsilon$

step 2: choose A,

$A \to S b S \mid f.$

↓ substitute S in 1st place

$A \to Abc X b S \mid efg X b S \mid f.$

↓ Apply direct left rec. elimination.

$A \to efg X b S Y \mid fY$

$Y \to bcXbsY \mid \varepsilon.$

$X \to aX \mid \varepsilon.$

final ans. → $S \to AbcX \mid efgX$

$A \to efgXb SY \mid fY$

$X \to aX \mid \varepsilon$

$Y \to bcXbSY \mid \varepsilon$

(5) $E \to E+T \mid a$

$T \to T*F \mid b$

$F \to c$

$E \to E+T \mid a$

↓

$E \to aX$

~~$X \to +TY \mid \varepsilon$~~ $X \to +TX \mid \varepsilon.$

$T \to T*F \mid b.$

$T \to bY$

$Y \to *FY \mid \varepsilon.$

final ans :- $E \to aX$

$X \to +TX \mid \varepsilon$

$T \to bY$

$Y \to *FY \mid \varepsilon.$

$F \to c.$

⑥ $S \to Ab$
$A \to Sa \mid Ad \mid f$
$B \to Sg \mid Ah \mid e$

$S \to Ab$
$A \to Aba \mid Ad \mid f$
$A \to fX$
$X \to baX \mid dX \mid \varepsilon$

$B \to Abg \mid Ahle$
$B \to fXbg \mid fXhle$

find ans. $S \to Ab$
$A \to fX$
$X \to baX \mid dX \mid \varepsilon$
$B \to fXbg \mid fXhle$

★ Left factoring -

Unambiguous CFG
↓ Left factoring
Left factored CFG. [free from common prefixes]

① $S \to ab \mid ac$ — common prefixes.

$S \to aX$
$X \to b \mid c$ — Left factored CFG.

② $S \to a \mid ab \mid abc \mid d$ — common.

$S \to aX \mid d$
$X \to b \mid bc \mid \varepsilon$ — common.

$X \to bY \mid \varepsilon$
$Y \to c \mid \varepsilon$

$S \to aX \mid d$
$X \to bY \mid \varepsilon$
$Y \to c \mid \varepsilon$

③. $S \to Ab \mid acA \mid d$ — Indirect common prefix
$A \to ab \mid f$.

$S \to Ab \mid acA \mid d$.
↳ non terminal, substitute.

$S \to abb \mid afb \mid acA \mid d$.
common.

$S \to aX \mid fb \mid d$
$X \to bb \mid cA$.
$A \to ab \mid f$.

④ final ans.

④ $S \to Sab \mid bA$
$A \to Bc \mid deB$
$B \to f \mid d$

$S \to bAX$
$X \to abX \mid \varepsilon$
$A \to Bc \mid deB$
$B \to f \mid d$.

$S \to bAX$
$X \to abX \mid \varepsilon$
$A \to fc \mid dc \mid deB$
$A \to fc \mid d\gamma$
$\gamma \to c \mid eB$

firstly eliminate left Rec.
$S \to bAX$
$X \to abX \mid \varepsilon$.
$A \to deB\gamma \mid \varepsilon$
$\gamma \to A \mid \varepsilon \gamma \mid \varepsilon$.
$B \to f \mid d$.
, $B \to fld$.

- Left factoring Algorithm

unambiguous CFG
↓ step1: Eliminate left Rec.
Non-left Recursive CFG
↓ substitute non terminals if they appear in 1st place.
CFG which is free from "non-terminal that appears in 1st place"
↓ Eliminate direct common prefixes.
Left factored CFG.

* First Set & Follow Set Computation -

$X \to a Y b \mid f Y g \mid e \mid \varepsilon$

First(X) = Set of terminals (~~other~~ including ε) where every terminal is derived as 1st symbol from X.
= {t | X ⇒ tα, where t is terminal or ε}

FIRST(X) = {a, f, e, ε}

follow(Y) = Set of terminals ~~(including ε)~~ where every terminal is derived as 1st symbol after Y.

FOLLOW(Y) = {b, g}.

FIRST(A)
Look at all A productions.

FOLLOW(A)
Look at whole CFG where RHS has A.

A → ____

NOTE:- FIRST(X) may contain ε, FOLLOW(X) never contain ε.

① $S \to a \mid \varepsilon \mid bcd$
FIRST(S) = {a, ε, b}

② $S \to Aab \mid c$ FIRST(S) = {c, d}
$A \to d$ FIRST(A) = d

③ $S \to Aabc$ FIRST(S) = {c, a}
$A \to \varepsilon$ FIRST(A) = {ε}

④ $S \to A$ FIRST(S) = {ε}
$A \to \varepsilon$ FIRST(A) = {ε}

(5) $S \to AB$, $A \to \varepsilon$, $B \to a$

FIRST(S) = {a}
FIRST(A) = {ε}
FIRST(B) = {a}

(6) $S \to AB$, $A \to ab \mid \varepsilon$, $B \to cd \mid \varepsilon$

FIRST(S) = {a, c, ε}.
FIRST(A) = {a, ε}
FIRST(B) = {c, ε}

- **FOLLOW set :-**

(1) $S \to a$ — follow(S) = {$}.

* If S is start symbol, then $ ∈ follow(S)	S$ → special end terminal.

(2) $S \to Sa \mid Sb \mid c$ — follow(S) = {a, b, $}.

(3) $S \to Sac \mid Sbd \mid e$ — follow(S) = {a, b, $}

(4) $S \to aA \mid bSd \mid c$ — follow(S) = {$, d, g}

$A \to Ae \mid Sg \mid f$ — follow(A) = {e, g, d, $} = {e} ∪ follow(S)

$S \to a\boxed{A}$ — $S \to \alpha$, $a\boxed{A}$?

NOTE :- follow(A) = we should compute follow(S)

If $X \to \alpha\boxed{Y}$ then follow(Y) = follow(X)

(5) $S \to AB$, $A \to a$, $B \to b$

follow(S) = {$}
follow(A) = {b}
follow(B) = {$}

⑥ $S \to AB$ Follow(S) = {$}
$A \to a$ follow(A) = {$} = Follow(S)
$B \to \varepsilon$ follow(B) = {$} = Follow(S)

⑦ $S \to AB$ follow(S) = {$}
$A \to a$ Follow(A) = {$, b}
$B \to bc \mid \varepsilon$ Follow(B) = {$}

★ Compute first & follow -

① $S \to SS \mid (S) \mid a$ FIRST(S) = {a, (, ε}, ~~follow(S) = {$,), a}~~
FOLLOW(S) = {$, (, a,)}

② $E \to E+E \mid E*E \mid a$ FIRST(E) = {a}
FOLLOW(E) = {+, *, $}

③ $E \to E+T \mid a$ FIRST(E) = {a} follow(E) = {+, $,)}
$T \to F*T \mid b$ FIRST(T) = {b, (, c} follow(T) = {$, +,)}
$F \to (E) \mid c$ FIRST(F) = {(, c} follow(F) = {$, *}

④ $S \to ABC$ FIRST(S) = {ε, a, e, g} follow(S) = {$}
$A \to ab \mid \varepsilon$ FIRST(A) = {ε, a} Follow(A) = {$, e, g}
$B \to ef \mid \varepsilon$ FIRST(B) = {ε, e} follow(B) = {$, g}
$C \to gh \mid \varepsilon$ FIRST(C) = {ε, g} follow(C) = {$}

★ LL(1) Parser :-
- one token prediction.
- Left most derivation (LMD)
- Left to Right scan.

→ Top-down parser.
→ Predictive parser.
→ ~~Recursive Descent~~ (RD)

→ Recursive Descent OR Non-recursive Descent

writing procedures for each production.

ex:- S → A

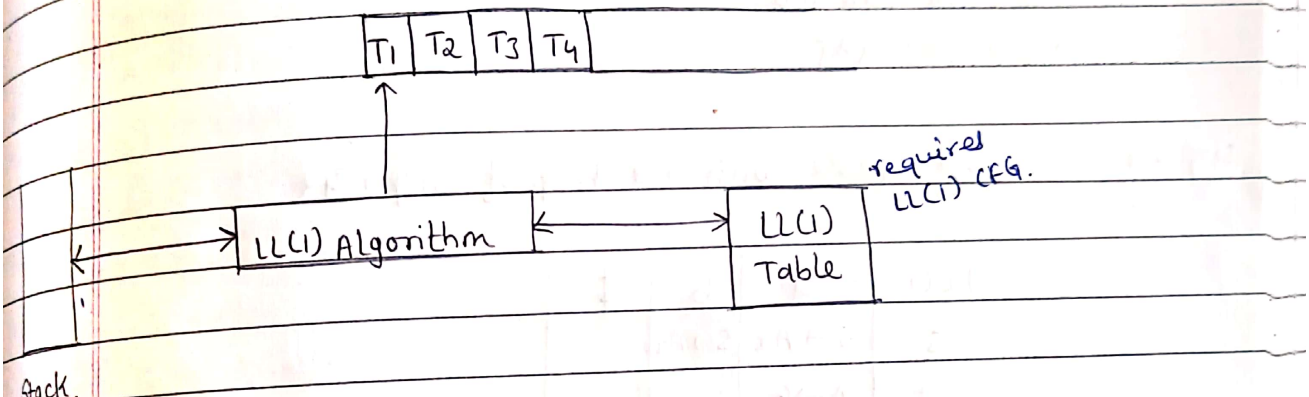
```
S()
{
  ch=='a'
}
```

table implementation

ex:- S → a.

	a
S	S → a.

→ LL(1) parser configuration-



• LL(1) CFG-

→ How to write LL(1) CFG?

Step 1: Take unambiguous CFG.
↓ Eliminate left recursion.

Step 2: Unambiguous & Non Left Recursion CFG.
↓ Apply left factoring.

Step 3: Unambiguous, Non left Recursion, & left factored CFG.
↓ using first & follow sets, construct LL(1) Table.

Step 4: If no multiple productions in the same entry of table then CFG is LL(1).

- How to construct LL(1) Table?

(1) $S \to Aa$

$A \to b \mid \varepsilon$

step 1: Compute FIRST set for every non-terminal.

FIRST(S) = {a, b}

FIRST(A) = {ε, b}

→ Because of this ε, we require follow(A).

step 2: ~~follow(A) = {$, a}~~.

If any FIRST set contain ε then compute follow set only for that non-terminal.

follow(A) = {a}.

step 3: construct table with the help of step 1 & 2

LL(1)	a	b	$
S	S → Aa	S → Aa	
A	A → ε	A → b	

$|V| \times |T| + 1|$

	a
S	fill this entry with an production of S, where they derive 'a' as 1st symbol. ex:- S → a, S → ab

→ for S → a | ab | ab | ab

S	we will have only S productions
A	we will have only A productions
B	we will have only B productions.

when we look at follow set.

	t
S	Fill null productions of S. ex :- S → AB, S → ε

S → AB | ε

A → ε

B → ε

* Shortcut

S → Aa

A → b | ε.

{b} ∩ fo(A)

{b} ∩ {a} = ϕ

→ LL(1) ✓.

i) X → α1 | α2

If Fi(α1) ∩ Fi(α2) ≠ ϕ then not LL(1).

ii) X → α | ε

If Fi(α) ∩ Fo(X) ≠ ϕ then not LL(1).

② S → aSb | ε

fi(aSb) ∩ fo(S)

a ∩ {b, $} = ϕ → LL(1) ✓.

First(S) = {a, ε}.

follow(S) = {b, $}.

	a	b	$
S	S → aSb	S → ε	S → ε

③ S → Sa | b.

* Every LL(1) CFG is -

I) Unambiguous.

II) Non Left Recursive.

III) Left factored.

→ Given CFG is having Left recursion, so not LL(1) ✓.

* If CFG is unambiguous, Non left Recursive, left factored then CFG need not be LL(1).

④ $S \to aSa|\varepsilon$

$fi(aSa) \cap Fo(S)$

$a \cap \{a, \$\} \neq \phi$

$a \neq \phi$ ↳ Not LL(1) even if its unambi, Non left rec, Left factored

$Fi(S) = \{a, \varepsilon\}$

$fo(S) = \{a, \$\}$

	a	$
S	$S \to aSa$	$S \to \varepsilon$
	$S \to \varepsilon$	

- Identify CFG is LL(1) or not.

① $S \to a$. LL(1).

② $S \to SS|\varepsilon$. $\neq$ LL(1)

③ $S \to a|ab$. $\neq$ LL(1) (cos common prefix)

④ $S \to Sab|\varepsilon$ $\neq$ LL(1)

⑤ $S \to Ab|Ba$ $Fi(Ab) \cap Fi(Ba)$

$A \to ab$ $a \cap a \neq \phi \to \neq LL(1)$

$B \to ac$.

⑥ $S \to Aa|Bb|acd|\varepsilon$ $Fi(Aa) \cap Fi(Bb)$

$A \to d$ $d \cap a = \phi$.

$B \to ab$ $Fi(Aa) \cap Fi(acd)$

$Fi(Bb) \cap Fi(acd)$ $d \cap a \neq \phi$.

$a \cap a \neq \phi$

↳ $\neq$ LL(1)

- LL(1) parsing Algorithm -

$S \to AB$ $\quad fi(S) = \{\$\}$

$A \to aA|\varepsilon$ $\quad fi(S) = \{a, b, a\}$

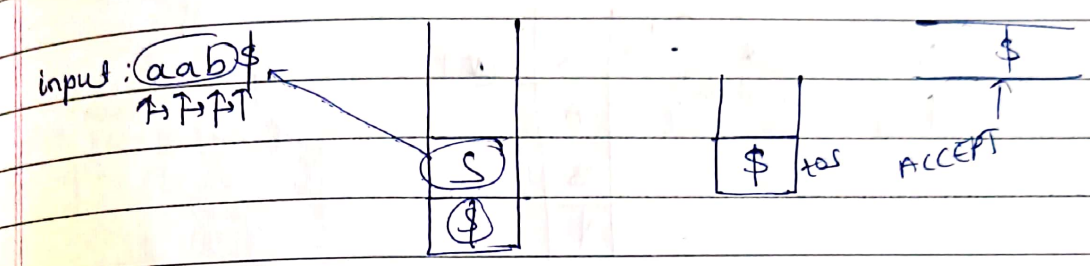
$B \to b$ $\quad fi(A) = \{\varepsilon, a\}$

$fi(B) = \{b\}$.

$fo(A) = \{\$, b\}$

M	a	b	$
S	S→AB	S→AB	
A	A→aA	A→ε	
B		B→b	

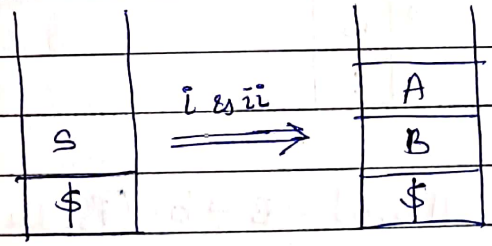
(M: STACK; input)



I) a a b $ $\quad M[S, a] = S \to AB$ $\quad$ {PREDICTION (substitution)

i) POP LHS.

ii) PUSH RHS in reverse order.

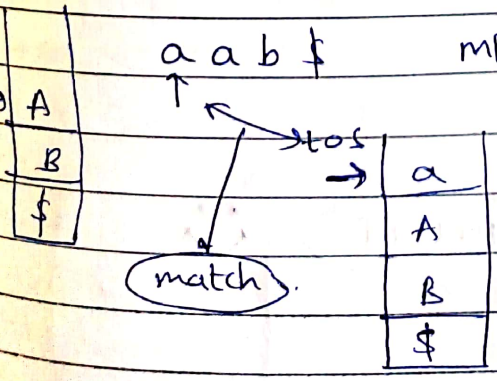


- $M[X, t] : X \to \alpha$

i) POP X (LHS)

ii) Push α (RHS) in reverse order.

II) a a b $ $\quad M[A, a] : A \to aA$.



i) Pop A

ii) Push aA in reverse order.

tos	t
	;
	;
	;
	;

t
↑ i/p.

(tos == i/p) ≠ $

i) Pop t.

ii) Increment i/p pointer.

(IV)

A
B
$

a b $
↑

match.

a
A
B
$

(V)

M[A, a] = A → aA (Prediction)

(VI)

a
A
B
$

b $
↑

M[A, b] = A → ε (Prediction)

i) POP A

ii) PUSH ε
↳ push nothing.

B
$

(VII)

B
$

b $
↑

match

b
$

M[B, b] = B → b (Prediction)

(VIII)

$

$
↑

match.

ACCEPT

S
⇓
AB
⇓
aAB
⇓
aaAB
⇓
aaB
⇓
aab.

LMD order: S → AB → aAB → aaAB → aaB → aab

I). T-conflict [Terminal conflict]

If tos is terminal
and not matching with i/p
⇓
T-conflict.

II) N-conflict [Non-terminal conflict]

	terminal
→ Non ter.	

If tos is non-terminal, and there is no production in table
⇓
N-conflict.

exL	a
B	
:	
:	

⇒ N-conflict.

	a
B	Blank.

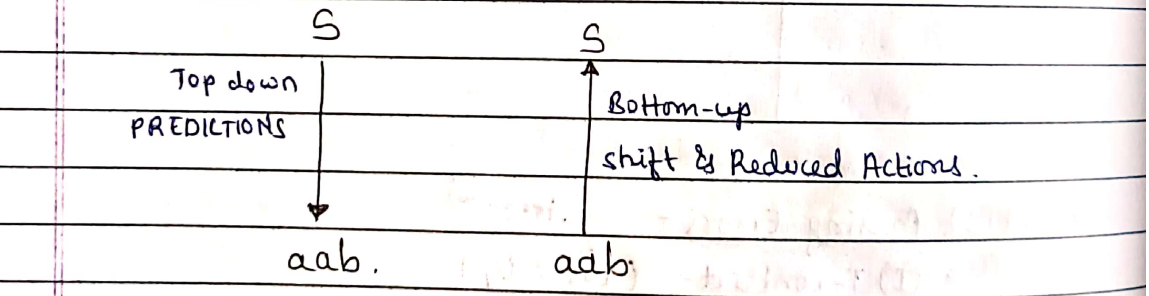
- LL(1) Parsing Errors -
  - ↳ (I) T-conflict. (tos ≠ i/p) (terminal)
  - (II) N-conflict. (M[tos, i/p] is blank).

	LL(0)	LL(1)	LL(2)	LL(3)	. . . . .
1)	S→aa	✓	✓	✓	
2)	S→aA A→b	✓	✓	✓	
3)	S→aA A→bB B→c	✓	✓	✓	

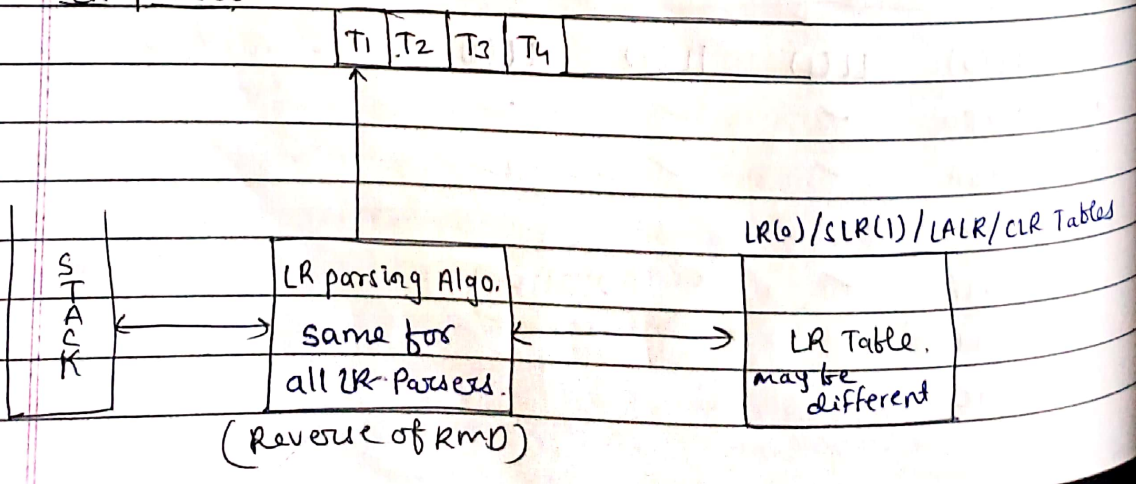
	LL(0)	LL(1)	LL(2)	LL(3)
1) S → a|b	X	✓	✓	✓
2) S → a|@b	X	X	✓	✓
3) S → a	✓	✓	✓	✓
4) S → ab|abc|ef	X	X	X	✓

Bottom-up parsers – (SR Parsers) (Shift, Reduce)

- **LR parsers.**
  - LR(0) parser.
  - (S)LR / SLR(1) — Simple
  - (LA)LR / LALR(1) — look ahead
  - (C)LR / LR(1) — canonical
  - LR
    - RMD in reverse.
    - Left to right scan.
- **Operator Precedence Parser**
  - for precedence relations



- LR parser



LR(0) Parsing DFA
LR(0) Grammar Identification.
SLR(1) Grammar Identification.
} LR(0) items required.

LR(1) parsing DFA.
LR(1) Grammar Identification.
LALR(1) parsing DFA.
LALR(1) Grammar Identification.
} LR(1) items required.

LR(0) Table → SLR Table → LALR Table → CLR Table.
LR parsing Algo.

- Basics
  - → Augmented CFG.
  - → Item
  - → Types of Items.
  - → Closure() and goto() } ⇒ Helps to construct DFA.

1) Augmented CFG -

S → Sa | Sb | c follow(S) = {a, b, $}

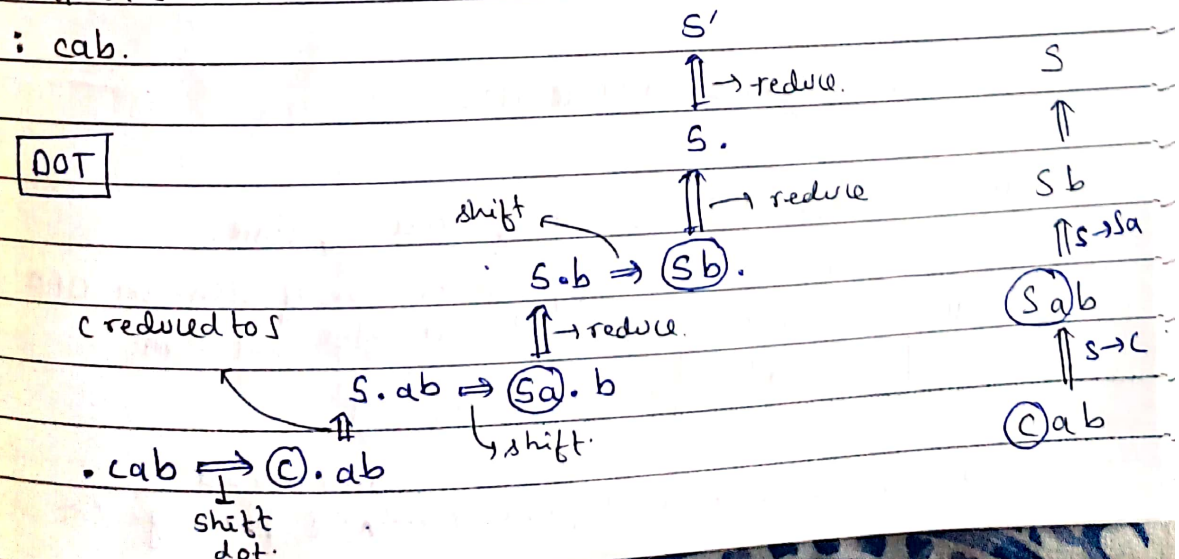
CFG → append S' → S

S' → S
S → Sa | Sb | c

Augmented CFG.

S → Sa | Sb | c
I/p : cab.

DOT



- Why we need a DOT?

→ To perform shift & Reduced Action.

→ To keep track of program.

- Item. (3 types)

↳ It is a production which has a dot.

$S \to AaB$.

Production.

$S \to \cdot AaB$ (non-ter.) → State Item

$S \to A \cdot aB$ (ter.) → Items. → Shift Item

$S \to Aa \cdot B$ (non ter) → State Item.

$S \to AaB \cdot$ → Reduced Item. (completed).

- Why we need a symbol which is immediately after DOT?

i.e. $X \to \alpha \cdot [\text{symbol}] \beta$.

Items.

Shift Item.	State Item.	Reduced Item.
$X \to \alpha \cdot \square\beta$	$X \to \alpha \cdot \square\beta$	$X \to \alpha \cdot$
↓ terminal.	↓ non terminal	
ex:- $S \to \cdot a$	ex:- $S \to \cdot A$	ex:- $S \to a \cdot$
$S \to A \cdot aB$	$S \to a \cdot AB$	$S \to A \cdot$
		$S \to aA \cdot$

$S \to ab \cdot Ac$ → State item.

★ closure(): → It is "set of items" → state.

↳ It should be applied on every item in DFA.

II) If Nonterminal present just after dot then.

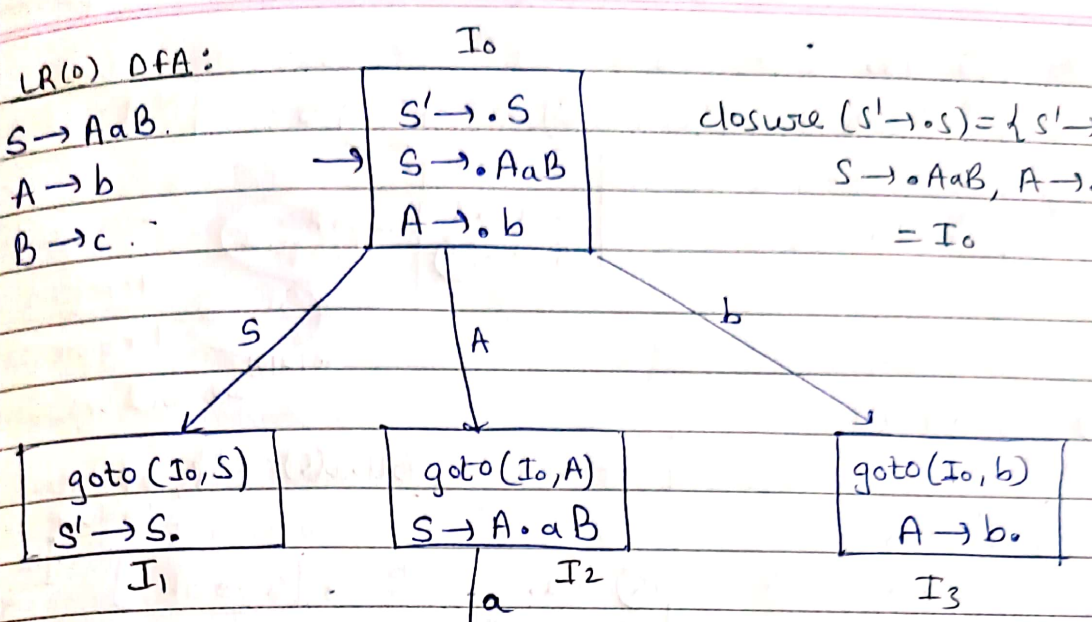
[ ] → [ ] • Nonterminal [ ]

X

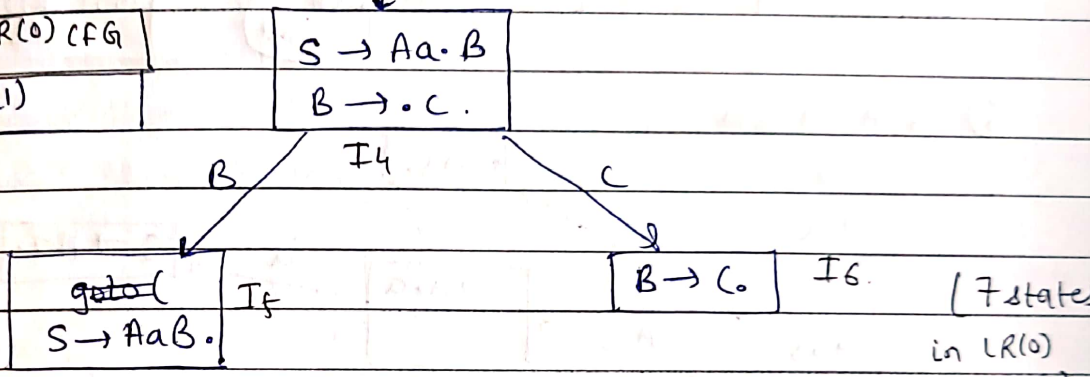
we should add all X productions in same set by placing dot at begin.

• LR(0) DFA:

S → AaB.
A → b
B → c.

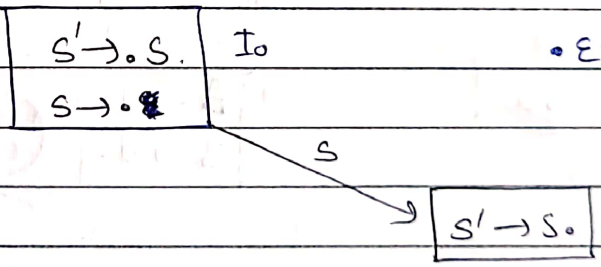


* It is LR(0) CFG so SLR(1)



(7 states in LR(0) DFA).

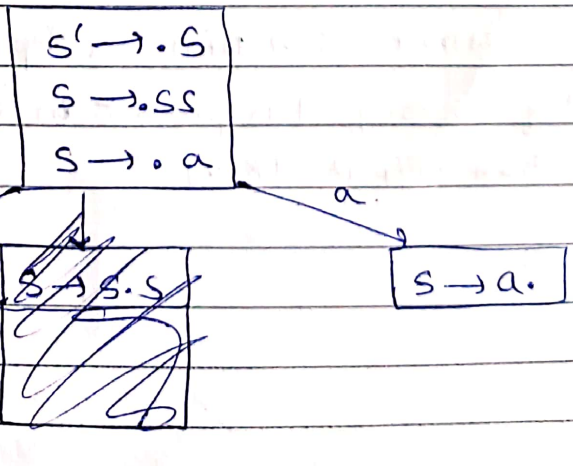
2) S → ε.



• ε = ε. = •

* It is LR(0) LFG so, SLR(1)

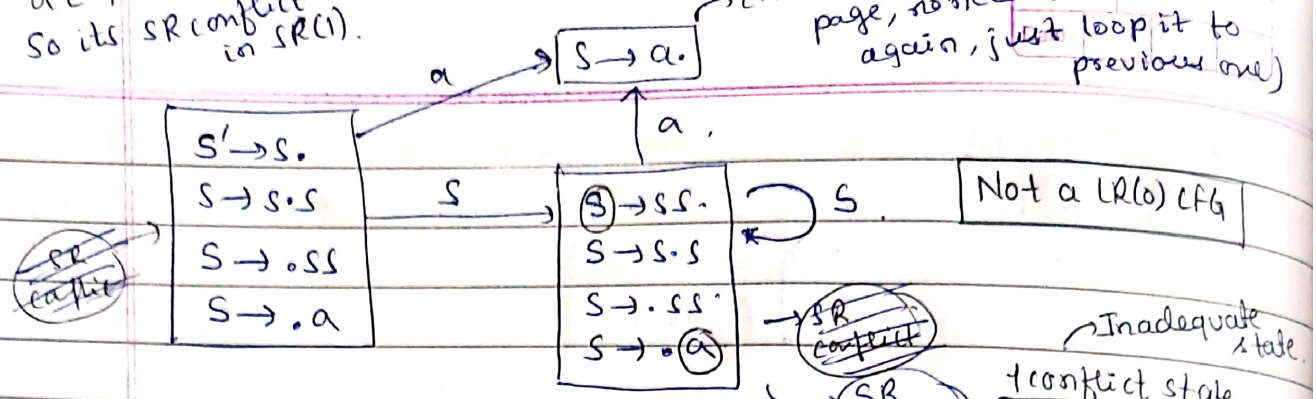
3) S → SS | a.



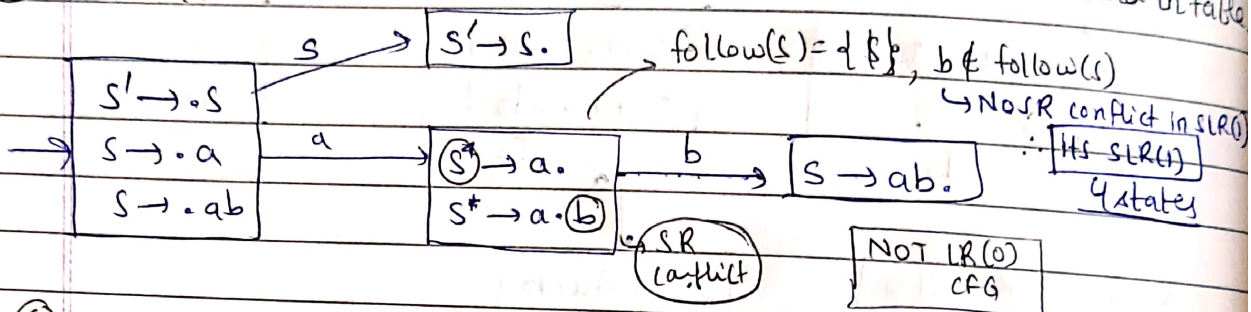
SR conflict

follow(S) = {$, a}
a ∈ follow(S)
So its SR conflict in SR(1).

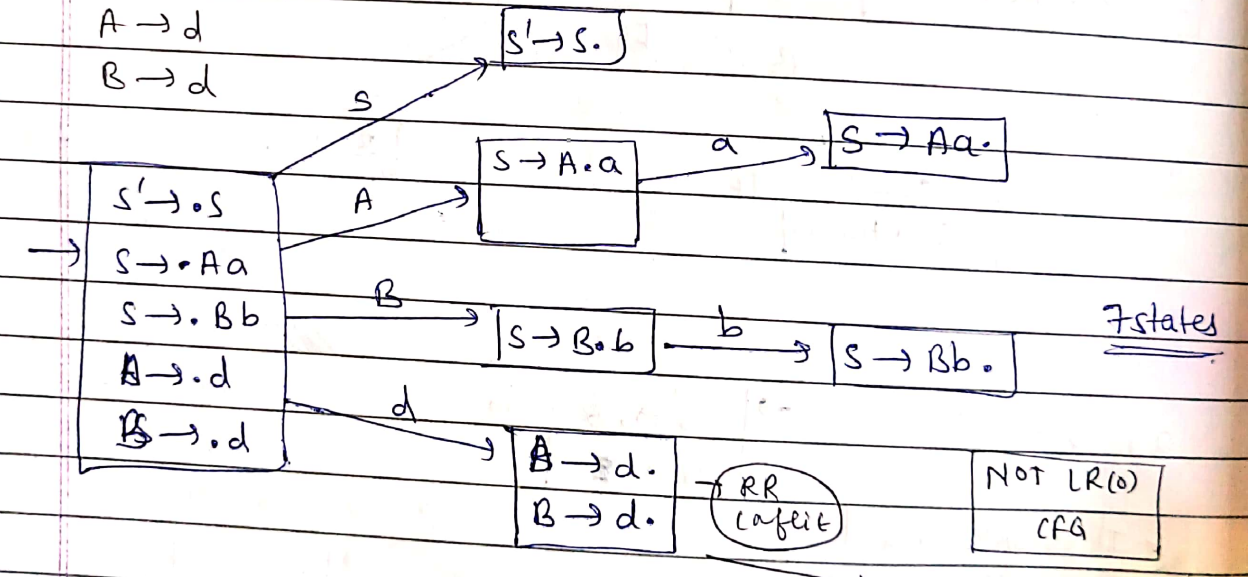
∴ It is not SLR(1) CFG.



(5) S → a|ab



(6) S → Aa|Bb
A → d
B → d



follow(A) = {$, a}
follow(B) = {$, b}
fol(A) ∩ fol(B) = ϕ
No RR conflict | its SLR

- How to check CFG is LR(0) or not?
  - → construct LR(0) DFA. (step 1)
  - → If no conflict is present in DFA, (step 2) then CFG is LR(0)

SR conflict in LR(0).

Shift item
Reduced item
⋮

If shift item & reduced item are present in same state then it produces SR conflict.

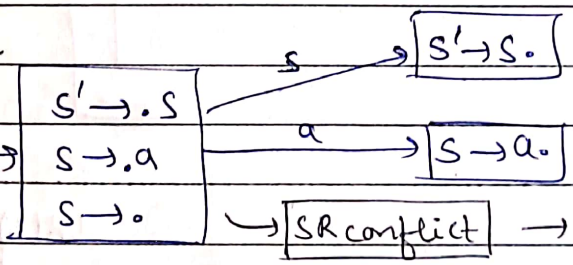
RR conflict in LR(0)

Reduced item$_1$
Reduced item$_2$
⋮

If 2 reduced items are present in same state then it produces RR conflict(s).

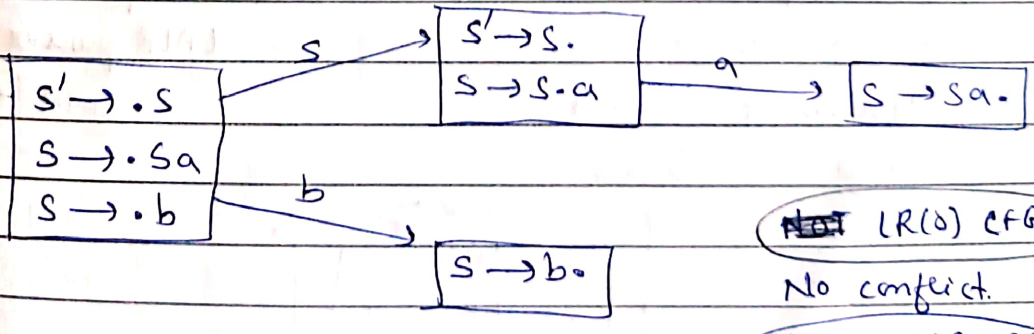
NOTE:- (I) If state has single item then it never produces conflicts.
(II) State item will not participate in any conflict.
*** (III) Acceptance item ($S' \xrightarrow{S:} S$) not participates in any conflict.
(IV) If state is not having reduced item then it never produces any conflict.

(7) $S \to a \mid \varepsilon$



Fol(S) = {$}
a ∉ fol(S) → No conflict
So its SLR(1)
→ Not LR(0) ~~conflict~~ CFG.

(8) $S \to Sa \mid b$



~~NOT~~ LR(0) CFG
No conflict.
its SLR(1) CFG

- How to check CFG is SLR(1) or Not?
  - step 1: Construct LR(0) DFA.
  - step 2: If no SLR(1) conflicts in DFA then CFG is SLR.

SR conflict in SLR(1)

shift item $X \to \alpha \cdot t \beta$

Reduced item $Y \to \alpha \cdot$

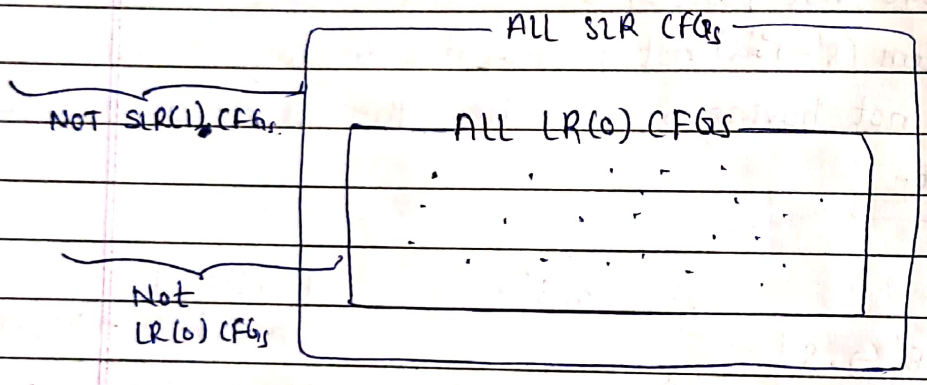
If $t \in$ follow(Y)
then SR conflict

RR conflict in SLR(1)

Red_1 $X \to \alpha \cdot$

Red_2 $Y \to \alpha \cdot$

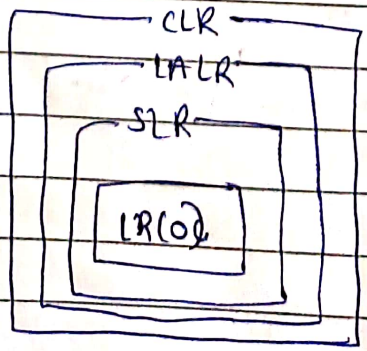
If follow(X) $\cap$ follow(Y) $\neq \phi$
then RR conflict.



Every LR(0) CFG is SLR.

LR(0) Items → LR(0) parser.
- SLR(1) parser.
  - follow set computed based on whole CFG

LR(1) Items → CLR parser [LRC1) parser]
LALR parser



* CLR & LALR -

- How to check given CFG is CLR or not?

→ step1: ~~the~~ construct CLR DFA. CLR aka. LR(1)

step2: Check SR & RR conflicts.

If no conflicts, then CFG is CLR.

- How to construct CLR(1) parsing DFA?

↳ use LR(1) items.

LR(0) item

$X \to \alpha \cdot \beta$

LR(1) Item

$X \to \alpha \cdot \beta , \gamma$

Actual Rule. ↳ look-aheads.

- How to compute look-a-heads of Y in LR(1) item?

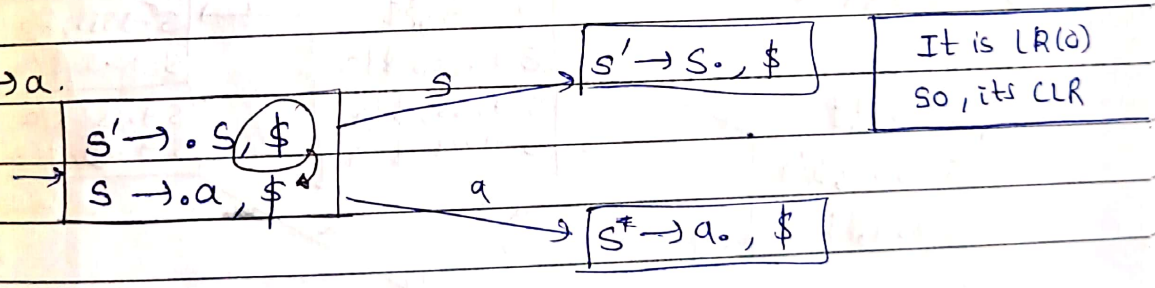
~~$X \to \alpha \cdot \beta$~~ $X \to \alpha \cdot Y\beta, L_1$ → First(βL_1)

$Y \to$ [ ], look-a-heads of Y.

$Y \to$

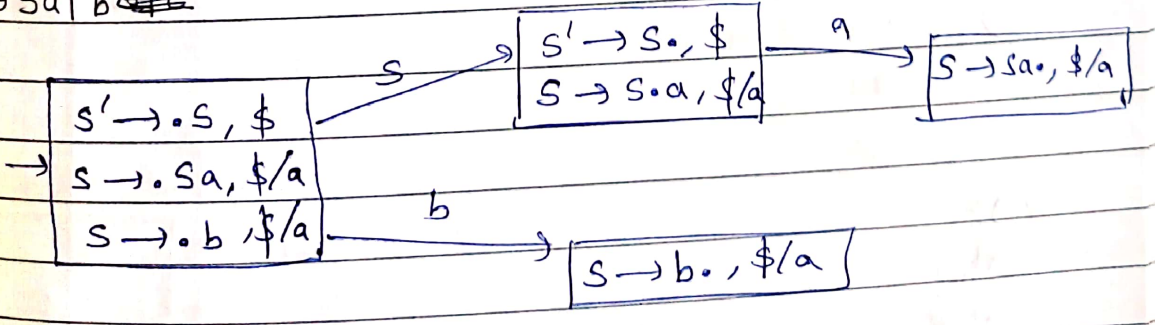
$Y \to$

① $S \to a$.



It is LR(0)
So, its CLR

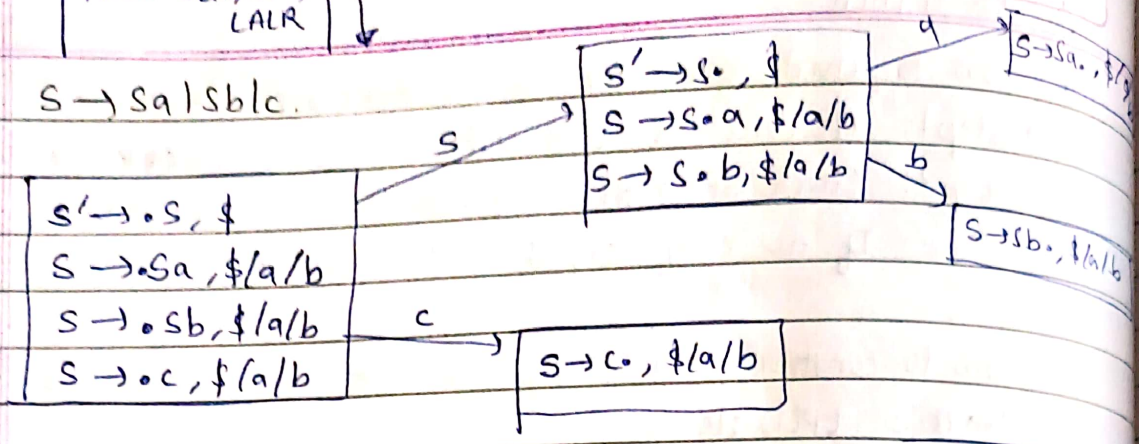
② $S \to Sa \mid b$



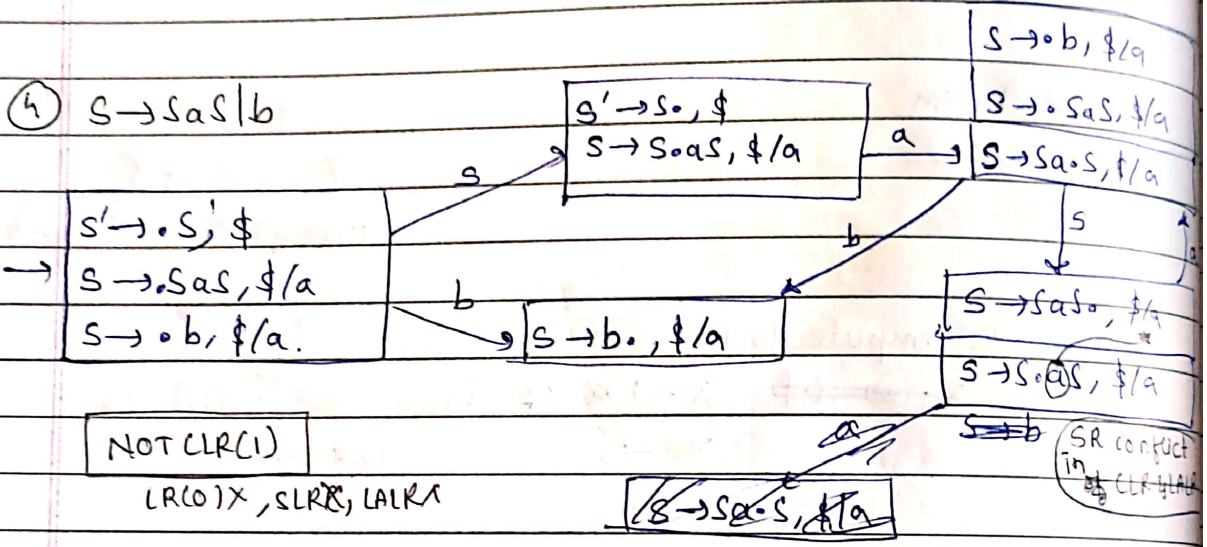
it is LR(0)
So its CLR, LALR, SLR

LR(0)
Thus SLR, CLR, LALR

③ $S \to Sa \mid Sb \mid c$.



④ $S \to SaS \mid b$

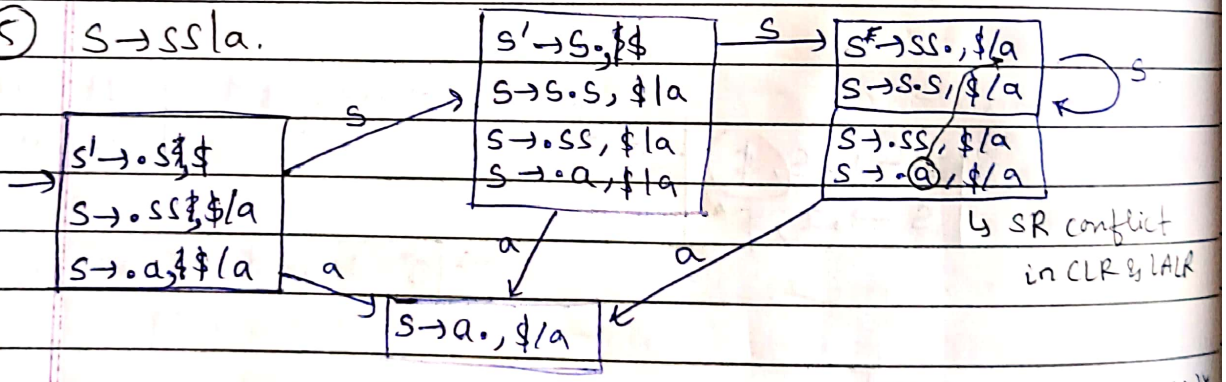


NOT CLR(1)

LR(0) ✗, SLR ✗, LALR ✓

SR conflict in CLR & LALR

⑤ $S \to SS \mid a$.



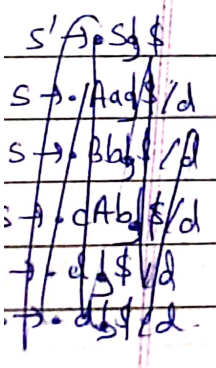
SR conflict in CLR & LALR

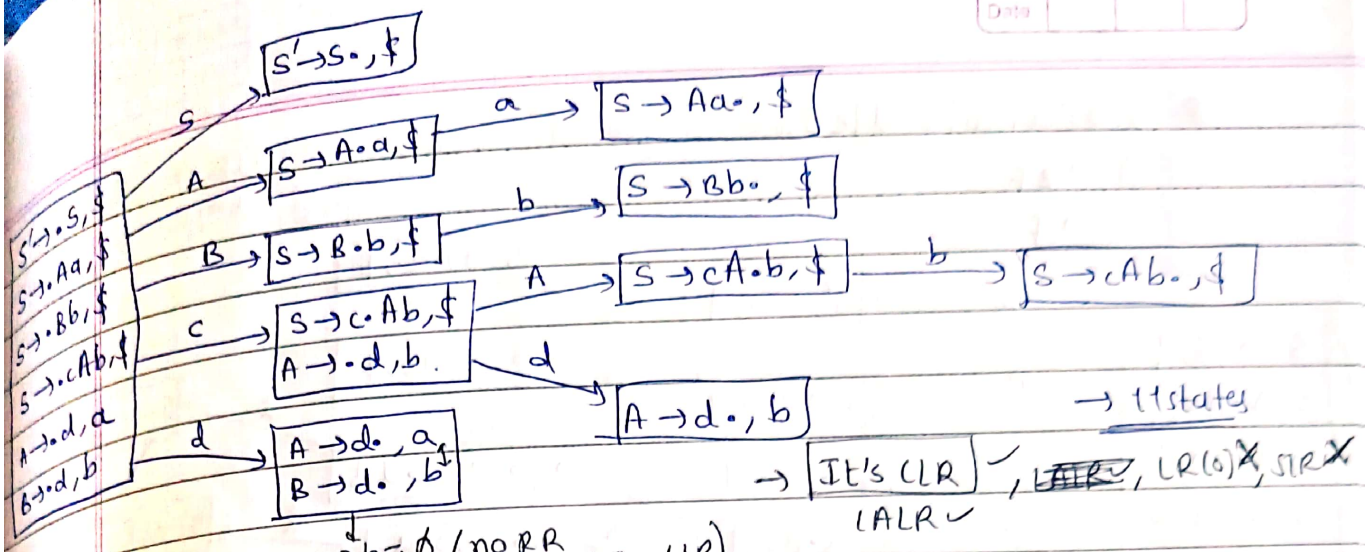
NOT CLR ✗ LALR ✗, LR(0) ✗ SLR ✗

⑥ $S \to Aa \mid Bb \mid cAb$

$A \to d$

$B \to d$





anb = ϕ (no RR conflict in CLR)

→ 11 states

→ It's CLR ✓, ~~LALR~~, LR(0) ✗, SLR ✗

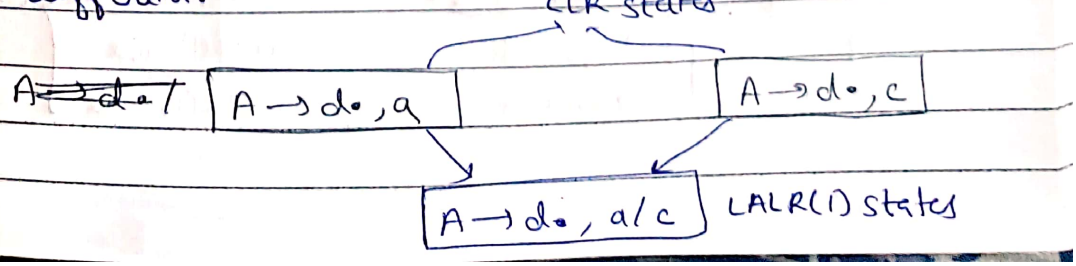
LALR ✓

conflicts	LR(0).	SLR.	CLR & LALR.
SR Conflict	$X \to \alpha . t \beta$ $Y \to \alpha .$ ⋮ state has both shift & reduced items.	If $t \in$ follow(Y) then SR conflict	$X \to \alpha . t \beta$, L_1 $Y \to \alpha .$, L_2 ⋮ If $t \in L_2$, then SR conflict.
RR Conflict	$X \to \alpha .$ $Y \to \alpha .$ ⋮ state has ~~both more~~ min. 2 reduced items.	If fo(X) ∩ fo(Y) is not empty then RR conflict	$X \to \alpha .$, L_1 $Y \to \alpha .$, L_2 ⋮ If $L_1 \cap L_2 \neq \phi$ then RR conflict

- How to check given CFG is LALR(1) or not?

  Step 1 :→ construct CLR(1) DFA.

  Step 2 :→ We can construct LALR(1) DFA by merging states if they have same items where look-a-heads may be different.



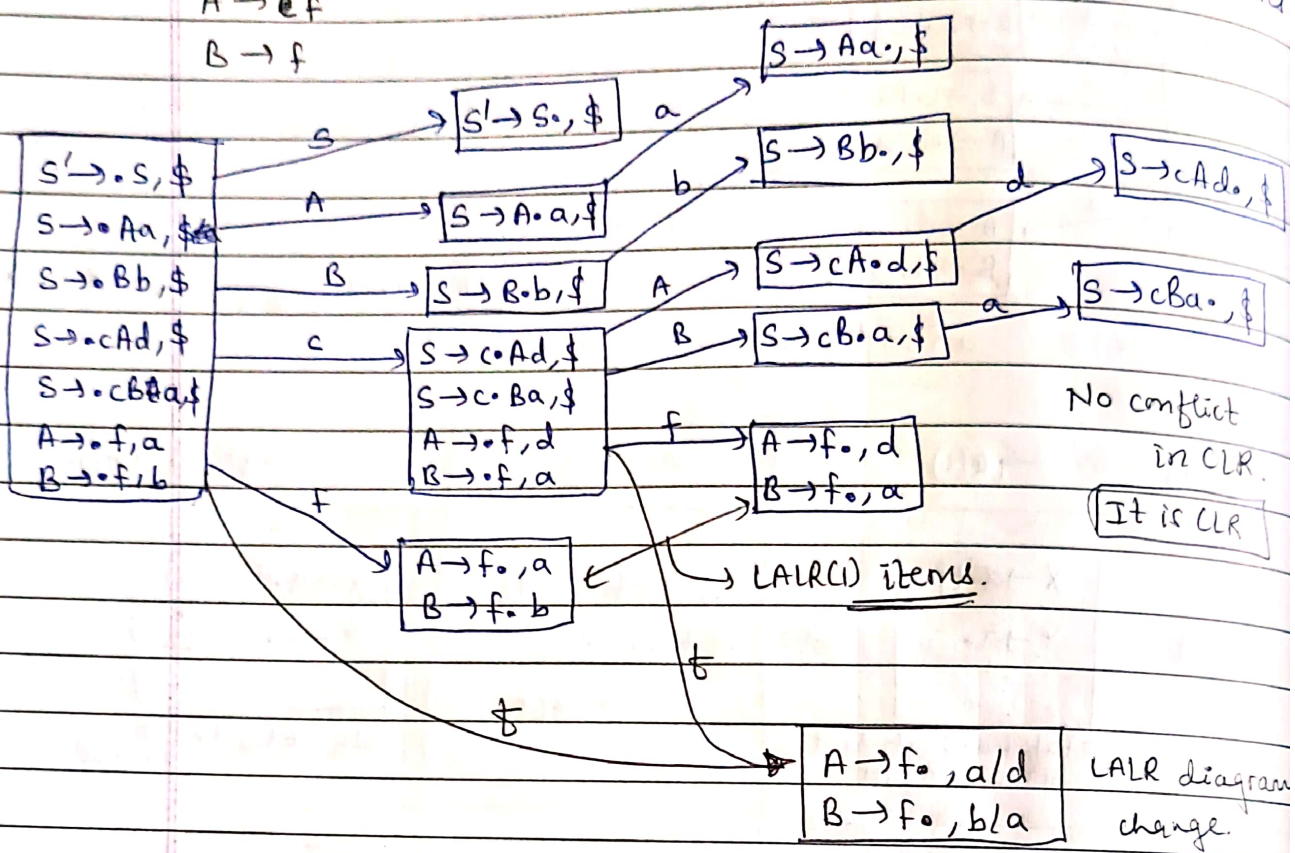
Page No.
Date

CLR → 13 states
LALR → 12 states

Q. $S \to Aa \mid Bb \mid cAd \mid cBa$.

$A \to ef$

$B \to f$



No conflict in CLR.

It is CLR

↳ LALR(1) items.

LALR diagram change.

$\{a, d\} \cap \{b, a\} \neq \phi$

$a \neq \phi$

Not a LALR ← ↳ RR conflict in LALR.

★ No. of states:

$n(LR(0)) = n(SLR) = n(LALR) \leq n(CLR)$

★ Power: LR(0) < SLR < LALR < CLR

less powerful.

more powerful.

● Entries in L(R) Table:

→ Production.

→ Blank.

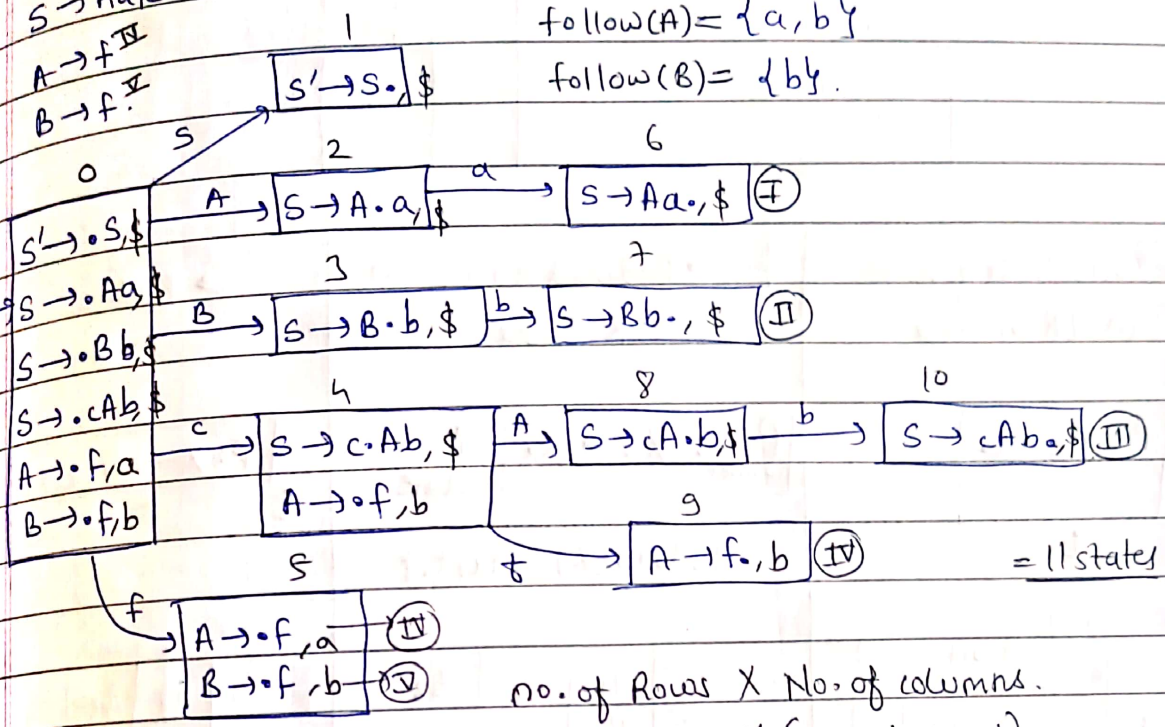
LR(0) Table
SLR(1) Table
LALR(1) Table
CLR(1) Table

Entries → Shift entry
→ Reduced entry
→ Go to entry.
→ Acceptance entry
→ Blank entry.

- LR(0) Table construction -

$S \to \overset{I}{A}a \mid \overset{II}{B}b \mid \overset{III}{c}Ab$.
$A \to f$ (IV)
$B \to f$ (V)

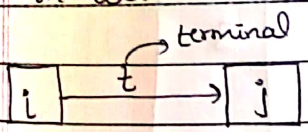
follow(S) = {$}
follow(A) = {a, b}
follow(B) = {b}.



= 11 states

no. of Rows X No. of columns.
no. of states X (|T| + 1 + |V|).
(1 → action, |V| → goto)

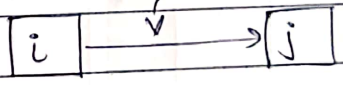
- Rule for shift Entry in all LR Tables:



	action col^m
	t
i	s_j

How many shift entries in table?
→ (6).
= no. of terminal transition in DFA.

- Rule for goto entry in all LR Tables:



	goto col^m
	X
i	j

How many goto entries in table?
→ (4)
= no. of non-terminal transition in DFA.

- Acceptance Entry Rule in all LR Tables :-

		$
i	i	Accept.
$S' \rightarrow S.$		
⋮		

- Reduced Entry Rule in LR(0) Table:

$P: X \rightarrow \alpha.$
⋮
i

- Reduced Entry in SLR Table.

P : $X \rightarrow \alpha.$	i	
number	state	
follow(X) = {t_1, t_3}		

- Reduced Entry Rule in LALR & CLR

$P: X \rightarrow \alpha., t_1/t_2$
i

	ACTION				
i	R_p	R_p	R_p	R_p	R_p

	t_1		t_3
i	R_p		R_p

	t_1	t_2
i	R_p	R_p

- LR(0) Table construction –

	← ACTION TABLE →					← GOTO TABLE →		
	a	b	c	f	$	S	A	B
0			S_4	S_5		1	2	3
1					Accept			
2	S_6							
3		S_7						
4				S_9			8	
5	RII RI	RIII RI	RIII RI	RIII RI	RIII RI			
6	RI	RI	RI	RI	RI			
7	RII	RII	RII	RII	RII			
8		S_{10}						
9	R_{IV}	R_{IV}	R_{IV}	R_{IV}	R_{IV}			
10	RIII	RIII	RIII	RIII	RIII			

- SLR(1) Table construction -

	a	b	c	f.	$	S	A	B
0			S_4	S_5		1	2	3
1					Accept			
2	S_6							
3		S_7						
4				S_9			8	
5	RIV	R IV R V						
6				RI				
7				RII				
8		S_{10}						
9	RIV	R IV						
10				RIII.				

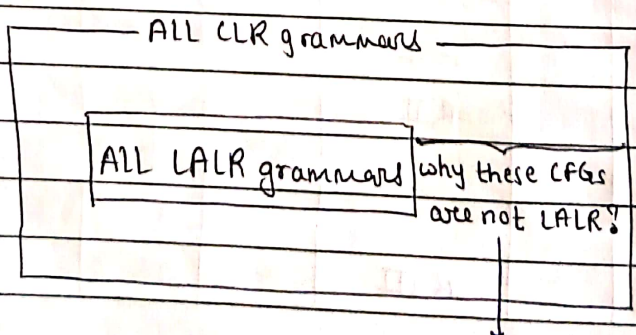
- LALR(1) Table construction-
& CLR(1)

	a	b	c	f	$	S	A	B
0			S_4	S_5		1	2	3
1					Accept			
2	S_6							
3		S_7						
4				S_9			8	
5	RIV	RV.						
6					R I			
7					RII			
8		S_{10}						
9		RIV						
10					RIII			

No. of states	#states(LR(0)) = #states(SLR) = #states(LALR) ≤ #states(CLR)
Table size	|LR(0) Table| = |SLR Table| = |LALR Table| ≤ |CLR Table|
No. of shift entries	[no. of shift entries in LR(0), SLR, LALR are same] ≤ [no. of shift entries in CLR]
No. of goto entries	[no. of goto entries in LR(0), SLR, LALR are same] ≤ [no. of goto entries in CLR]
No. of reduced entries	# reduced entries in LR(0)] ≥ [in SLR] ≥ [in LALR] ≥ [in CLR]
No. of conflicts.	in LR(0) ≥ in SLR ≥ in LALR ≥ in CLR

NOTE:- If CFG is CLR and containing states to get LALR then RR conflicts possible, SR conflict is impossible

$X \to \alpha\bullet, L_1$ (a)	&	$X \to \alpha\bullet, L_3$ (b)	no SR conflict	In CLR &	no SR conflict
$Y \to \alpha\bullet, L_2$ (b)		$Y \to \alpha\bullet, L_4$ (c)	$X \to \alpha\bullet t\beta, L_1$; $Y \to \alpha\bullet, L_2$		$X \to \alpha\bullet t\beta, L_3$; $Y \to \alpha\bullet, L_4$
No RR conflict		No RR conflict	$t \notin L_2$		$t \notin L_4$
combine to get LALR.			combine to get LALR.		
$X \to \alpha\bullet, L_1 \cup L_3$ {a,b} ; $Y \to \alpha\bullet, L_2 \cup L_4$ {b,c}	possibility of RR conflict		$X \to \alpha\bullet t\beta, L_1 \cup L_3$; $Y \to \alpha\bullet, L_2 \cup L_4$	impossible to get SR conflict.	
			$t \notin L_2 \cup L_4$		



If we construct CLR, no conflicts.

If we construct LALR, RR conflicts exi

input: fa. Table: LALR

step1:

Input
f | a | $

0

M[0,f] = S_5.

step 2:

a | $

~~s~~
~~f~~
0

M[5,a] : R_{IV}

A → f.

i) Pop 2*|f|

ii) Push A

iii) Push M[0,A] = 2.

step3:

~~2~~
A
0

a | $

m[2,a] = S_6.

step4:

~~6~~
~~a~~
~~2~~
~~A~~
0

$

m[6,$] = R_I

S → Aa.

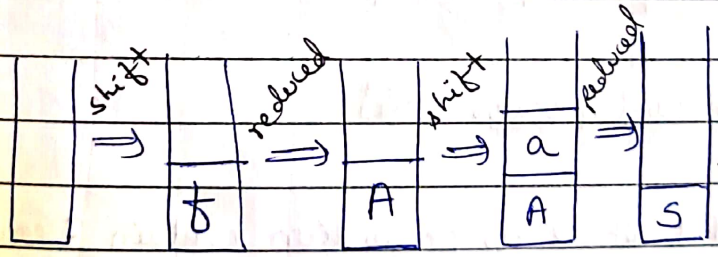
step5:

1
S
0

$

m[1,$]

ACCEPT



Reverse of RMD

S

Aa

fa.

S

Aa RMD.

fa

• fa ⟹ (shift) f•a ⟹ (Reduced) A•a ⟹ (shift) Aa• ⟹ (Reduced) S•

I) Every LL(1) is LR(1), → CLR.
II) Every LR(0) is SLR
III) Every SLR is LALR.
IV) Every LALR is CLR, → LR(1).
V) Every ambiguous CFG is not CLR.
VI) Every not CLR is not LALR.
VII) Every not LALR is not SLR.
VIII) Every not SLR is not LR(0).
IX) Every LR(0), SLR, CLR, LALR, LL(1) } is unambiguous.

* Operator precedence parsing -
  - → operator Grammar.
  - → operator precedence Relations.
  - → How to compute operator precedences.

→ Operator Grammar [operator CFG]
  - → It is CFG where no production contain 2 consecutive non terminals. and no null production in CFG.

① $S \to a$ OG.

② $E \to E+E$ OG. $E \to a$ OG

③ $E \to EE$ XOG. $E \to a$ XOG.

④ $E \to E+E \mid \varepsilon$ not OG.

⑤ $E \to EE \mid a \mid \varepsilon$ Not OG.

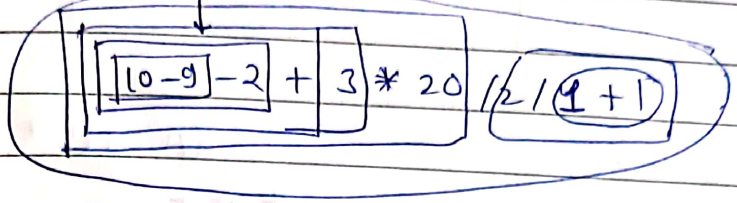
→ Operator precedence Relations :-
1) Highest precedence
2) lowest precedence.
3) Equal precedence } Associativity.
   - → (i) Left Associativity.
   - (ii) Right Associativity

①

	order	Associativity
-	Highest	Left-to-Right
+, *		Left-to-Right
/	lowest.	Right-to-left

Consider this precedence table.

Find o/p for 10-9-2+3*20/2/4+1



$= 40/1 = 40$.

② - is highest, * is lowest

- follows left associative, Remaining follows Right to left.

Find output for 3*2+3-1+4-2-1.

-	left to Right
+	Right to left
*	Right to left

3*2+(3-1)+4+2-1

3*2+2+2-1

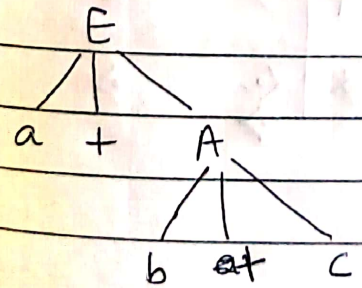
3*2+2+1

~~6+2+1~~ ~~9~~

3*2+3

3*5 = 15

③



a + b + c

Left Right

→ Left (+) has less precedence than Right (+).

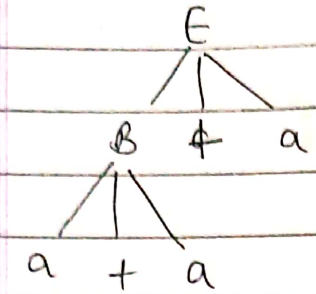
→ + is right associative.

	Right
Left	+
+	←

+ < +

Left Right

④



((a + a) + a)

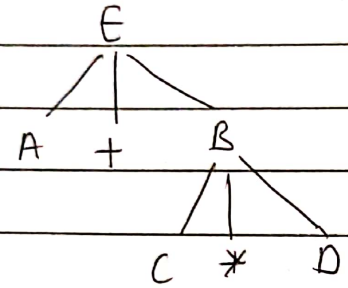
Left ⇒ Right. [+ ⇒ +]

Left to right associativity.

	+
+	⇒

	Right
Left	

⑤



(A + (C * D))

~~* ⇒ +~~

highest → (*), lowest → (+)

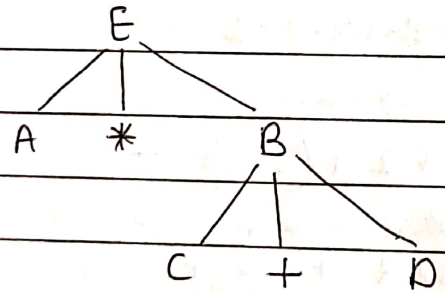
left right

I) + ⇐ * ✓

II) * ⇒ + ✗

left right

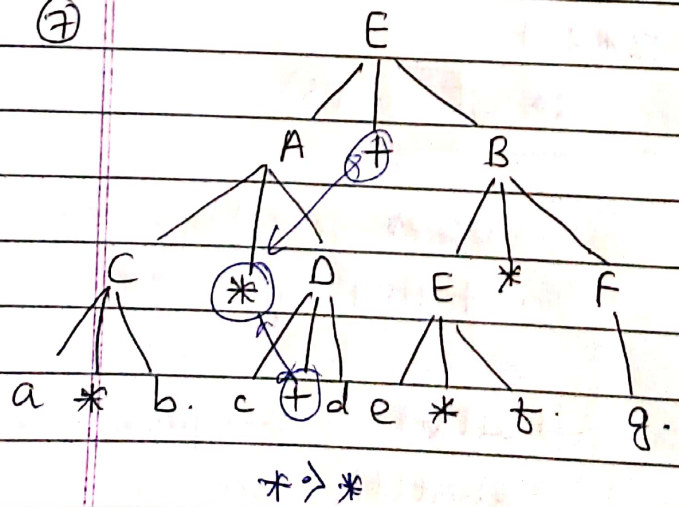
⑥



(A * (C + D))

* ⇐ +

⑦

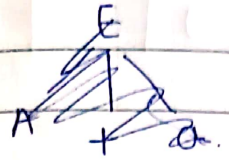


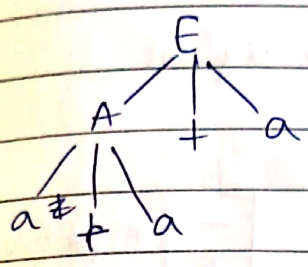
* ⇒ *

		Right	
		+	*
Left	+	⇒	⇐
	*	⇐	⇒

(8) $E \to A + \overset{a}{B} \mid a$

$A \to a + a$



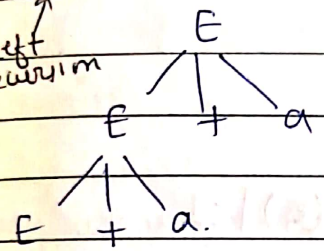


+ > +

+ is left Associative.

(9) $E \to E + a \mid a$

left recursion



+ > +

Left associative.

- + is left Associated because + associated with left Recursion.

(10) $E \to a * E \mid a$ → Right associative coz Right recursion.

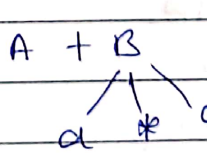
(11) $E \to a * A \mid a$ → * is Right associative

$A \to a * a$

(12) $S \to A + B \mid a$ $a + a * a.$

$B \to A \to a$

$B \to a * a.$



	+	*
+		<
*		>

(13) $E \to E + T \mid a$

$T \to F * T \mid b$

$F \to c.$

	+	*
+	① >	② <
*	③ >	④ <

Right. Left.

① + is left Associative. → coz left associative.

② + < *

③ * > +

④ * is right Associative.
↳ coz right recursion.

(14) $E \rightarrow (E) \mid a$.

	(	)
(		$\doteq$
)		

$(\doteq)$

Left Right

(15) $E \rightarrow E+E \mid E*E \mid id$

If - is right associative & - is highest
* is left associative then
find o/p for $5-2-1*3$

-	Right to left
*	left to right

$5-2-1*3$

$5-1*3$

4 → (12) and.

- $E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid E\%E \mid (E) \mid id$.
  It will generate all arithmetic expressions.

NOTE:- Operator precedence Relations can be computed using -
1) Parse tree.
2) Operator precedence table.
3) Unambiguous operator Grammar.
4) Information provided in English statements.

- Semantic Analysis
  ↳ It uses SDT to perform type checking, function compatibility, variable declaration etc.

Vs.

Syntax Directed Translation
↳ It is more powerful program or tool that can be used for
1) Semantic Analysis
2) Parse tree Generation.
3) Intermediate code.
4). Evaluations of expressions
5). Conversions.

- SDT

SDT = Syntax + translation.
(structure)

- → Semantic.
- → Evaluations.
- → Actions.
- → Specific Requirement.
- → Program.

= CFG + Translations
{ .... }

ex:-

$S \rightarrow AB$	$\{S.x = A.x * 2\}$
$A \rightarrow a$	$\{A.x = a.val + 100\}$
$B \rightarrow a$	$\{B.x = 1\}$.

Syntax. → translation.

→ Every production has 0 or more translations

$S \rightarrow AB$ $\{A.y = S.x + B.z\}$ (compute: A.y; use: S.x, B.z)

syntax rule. translation

- → Computation
- → Attributes.

S (Parent) of A

A | B (right sibling) of A.

x, y, z } Attributes.

- Attributes -

↳ ① Inherited Attribute.

$S \rightarrow A\,a\,B$

$\{B.val = S.val * A.val\}$

computation.

→ Computation depends on parent/siblings.

② Synthesized Attribute.

$S \rightarrow A\,a\,B$

$\{S.val = A.val + a.val\}$

computation.

→ Computation depends on children.

→ Find Attribute type -

① $S \rightarrow a$ $\{S.x = a.val\}$ (x is synthesized)

② $S \rightarrow Sa$ $\{S_1.x = S.x + 1;\ S.y = S_1.y * 2\}$ x → Inherited
$S \rightarrow b$ $\{S.x = b.val - 1;\ S.y = S.x + 9\}$ x → synthesis.

} x is neither inherited nor synthesis.

y → synthesized } y is synthesized attribute.
y → synthesized }

③ E → E1 + E2 {E.x = E1.x + E2.x}
E → a {E.x = a.val}.
x is synthesized attribute.

SDT is S-attributed & L-attributed.

④ D → T I ; {I.type = T.type}
T → int {T.type = int}
I → a { }.
This SDT is L-attributed. but not S-attributed.

Attribute type is neither inherited nor synthesized.

SDTs

Specifications
for writing SDTs
by using "Definitions"

Evaluations
For translating SDTs

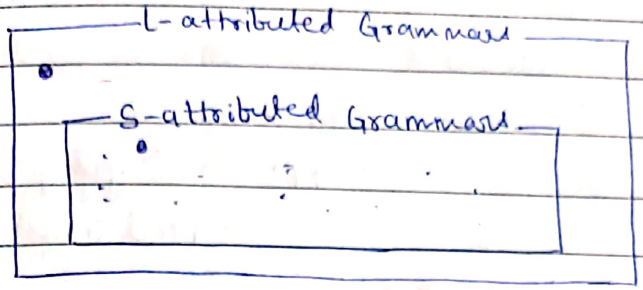
→ Definitions of SDTs :- (L-attributed definitions)
① L-attributed Grammar [L-attributed SDT]
② S-attributed Grammar [S-attributed SDT]
(S-attributed definitions)

① L-attributed SDT.
→ Computation depends on parent / left siblings / children
S → a {S.val = a.num}
S → AB {B.val = S.val}
{A.val = S.val}
{B.x = S.x + A.y}
{S.x = A.x + B.x}

NOTE:- Except right siblings, it can depend on any.
→ Translations can appear anywhere. S → {...} a

② S-attribute SDT
→ Computation depends on only children.
[It uses synthesized attributes]
→ Translation must appear only at end of production.
S → a {...}

→ Every S-attributed SDT is always L-attributed.



- $E \to E_1 \{E.val = E_1.val + E_2.val\} + E_2$.
  $E \to a \{E.val = a.num\}$.
  ↳ not at the end of production.

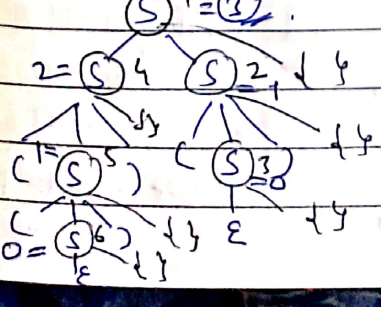
↳ SDT is not S-attributed
SDT is L-attributed.

- L-attributed Grammar -
  Computation can depend on parent / left sibling / children.
  ↳ evaluated in top-down approach
  ↳ evaluated in bottom-up approach.

- S-attributed Grammar -
  Computation depends only on children.
  ↳ (Bottom up approach)

(1) $S \to SS_2 \{S.x = S_1.x + S_2.x\}$
$S \to (S_1) \{S.x = S_1.x + 1\}$
$S \to \varepsilon \{S.x = 0\}$

(Q1) What is attribute type? (x) → x is synthesized attribute.
(Q2) What is SDT definition? → Both S-attributed & L-attributed
(Q3) Find attribute value at root for (()())().
If we use S-attributed evaluation: (Bottom up parsing)
(Reverse of RMD)



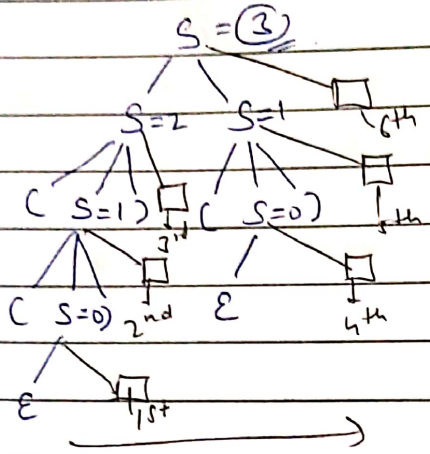
RMD Numbering: 1st 2nd 3rd 4th 5th 6th
Reverse.

ans. 3.

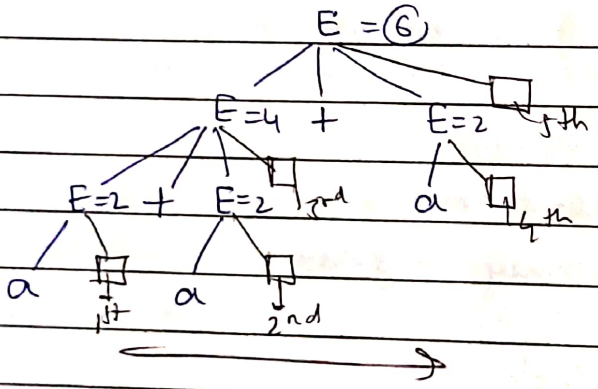
Q4) what is functionality of this SDT? ⇒ computing no. of balanced parentheses

→ If we use L-attributed evaluation:
(Depth first, Left to Right).
(Inorder approach) (Topological order).

Find translations order from Left to Right.

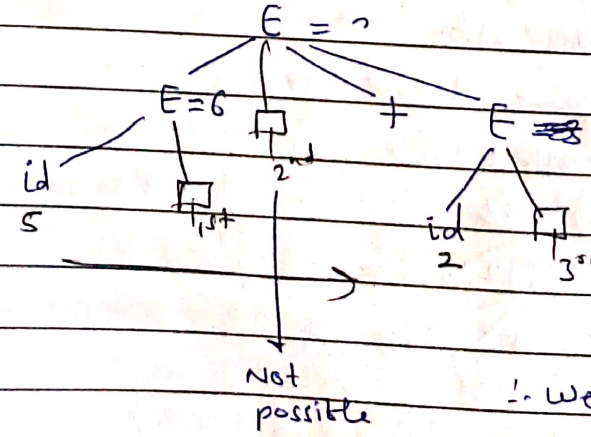


2). $E \rightarrow E_1 + E_2 \{E.x = E_1.x + E_2.x\}$.
$E \rightarrow a \{E.x = 2\}$, Find translation for a+a+a.



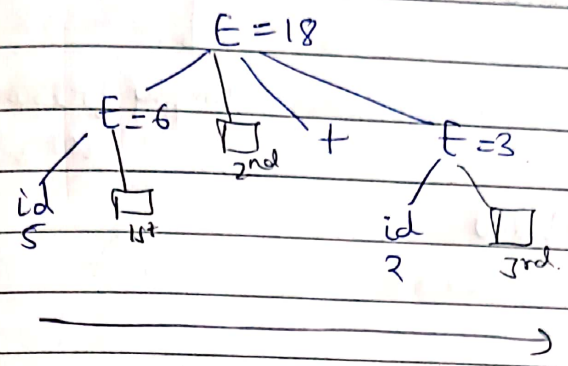
ans. 6.

3). $E \rightarrow E_1 \{E.x = E_1.x - E_2.x\} + E_2$
$E \rightarrow id \{E.x = id.val + 1\}$.
Find translation for ~~id + id~~. 5+2.

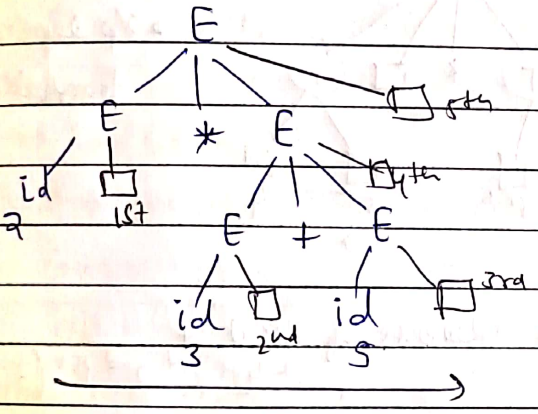


∴ We can't translate.

④ $E \to E_1 \{E.x = E_1.x * 3\} + E_2$.
$E \to id \{E.x = id.val + 1\}$.
Find translation for 5+2.



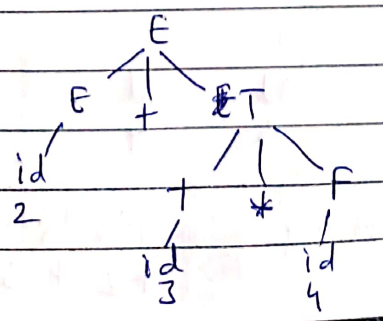
⑤ $E \to E + E \{print +\}$ + is high precedence.
$E \to E * E \{print *\}$ + and * are left Associate.
$E \to id \{print\ id.val\}$
Find o/p for 2*3+5.



o/p: 2 3 * 5 +

⑥ ~~$E \to E + T \{p$~~
$E \to \{print +\} E + T$ $E \to {}^{r}E + T$
$E \to id \{print\ id.val\}$ $E \to id^{r}$
$T \to T * \{print *\} F$. $T \to T * {}^{r}F$
$T \to id \{print\ id.val\}$ $T \to id^{r}$
$F \to \{print\ id.val\}\ id$ $F \to {}^{r}id$

Q1) Find o/p for 2+3*4
Q2) Find o/p for 2+3*4 using LR parsing
Q3) Find o/p for 2+3*4 using RMD.



Q4). Find o/p for 2+3*4 using TDP(LMD)(LL(1)).
Q5). Find o/p for 2+3*4 using reverse of LMD.
Q6). Find o/p for 2+3*4 using reverse of L-attributed order.

SDT:

Attributes computation exist.

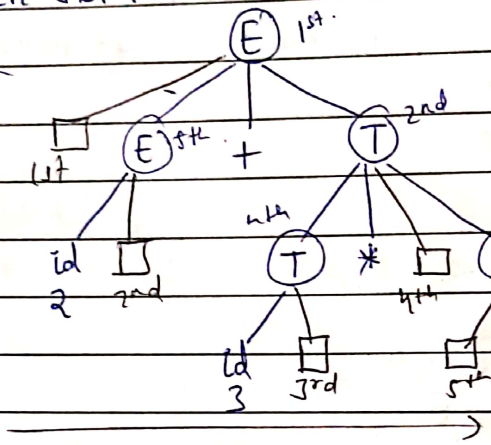
No attributes are computed.

simply follow order that is mentioned in question

L-attributed ⇐ S-attributed.

always follow this

Q1): Given SDT is L-attributed,



o/p: +2 3 * 4.

- o/p depends on translation position.

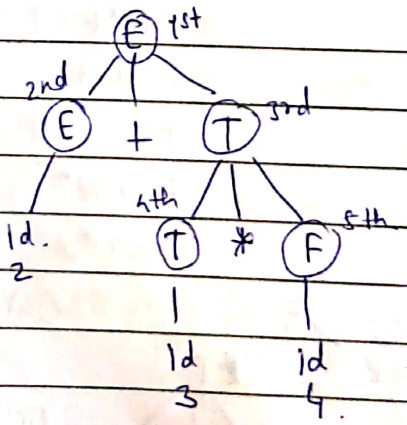
Q2) LR parsing → bottom-up → Reverse of RMD → 5th 4th 3rd 2nd 1st (2 3 4 * +).

Q3) RMD order: 1st 2nd 3rd 4th 5th

+ * 4 3 2.

→ o/p only depends on non terminal

Q4)



E

⇓

E + T

⇓

id + T

⇓

id + T * F.

~~LRD~~

o/p: LRD numbering order: 1st 2nd 3rd 4th 5th

+2 * 3 4.

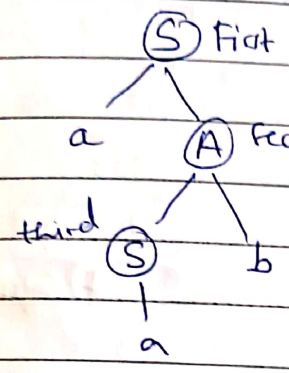
(85) Reverse of IMP : 43 * 2 +

(86) 4 * 32 +] → Reverse of L-attributed o/p.

(7) S → a A {print 1}
S → a {print 2}
A → Sb {print 3}

I/p = aab

What is o/p using bottom up parser?
reverse of RMD



Reverse of : Third Second First
RMD 2 3 1

(8) E → E1 + E2 {E.x = E1.x (+) E2.x + (1)}
E → E1 * E2 {E.x = E1.x (*) E2.x + (1)}.
E → id {E.x = (0)}
What is functionality of SDT?
→ It counts no. of operators

- Intermediate code. → I.C.
  - → Postfix code.
  - → 3 address code.
  - → SSA code.
  - → Syntax tree
  - → DAG
  - → Control flow graph

Q). What is Intermediate code?
HLL —ICG (Easy)→ I.C. —CG (Easy)→ Ass. Code.
→ It's a machine independent code
→ Portable.

Analyzed code (Lexically, Syntactically, Semantically verified) → ICG → Intermediate code.

ICG → Inter. code Generator
CG → Code Generator.

ex:- x = y + z/c; → t1 = z/c
t2 = y + t1
x = t2

→ Intermediate code has nearest relation to assembly code. ← easily convertable to assembly code.

★ Intermediate Code Representations :-

I) Linear → Postfix
→ 3AC/TAC (Three Address Code)
→ SSA (static single Assignment)

II) Non linear → Syntax Tree.
→ DAG (Directed Acyclic Graph)
→ CFG (control flow graph)

$x = a+a+a+a$

$x = (((a+a)+a)+a)$

I) Postfix code

$x\ aa+a+a+=$

*** II) Intermediate code (TAC/3AC) -

Every instruction has atmost 3 address.

$t_1 = a+a$
$t_2 = a+a$
$x = t_1 + t_2$

OR

$t_1 = a+a$
$x = t_1 + t_1$

OR

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

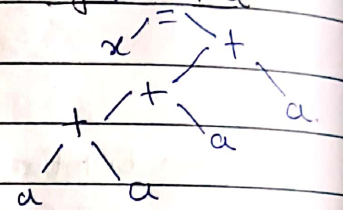
SSA SSA Not SSA

III) SSA code -

→ Every variable is having single assignment.

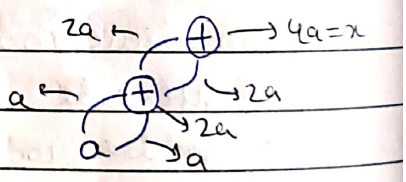
→ It is 3 address code but, every var. is having single assignment.

I) Syntax Tree



Leaf : operand,
non-leaf : operator.

II) DAG.



Eliminates common subexpressions

→ 3 nodes, 4 edges

ex(1) $x = a * b$.

postfix : $x\ ab*=$

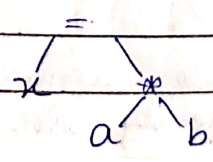
3AC : $x = a*b$.

OR $a = a*b$
$b = a*b$.

→ a has 2 def^n, b has 2 def^n → ∴ they are not SSA code.

SSA : $x = a*b$.

Syntax tree:



DAG

② a + a + a.

Postfix : xaa+a+=

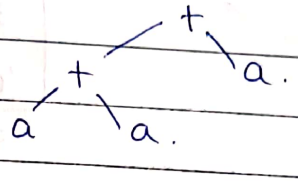
3AC : $t_1 = a + a$ (OR) $t_1 = a + a$ (OR) $t_1 = a + a$

$t_1 = t_1 + a$ $a = t_1 + a$ $t_2 = t_1 + a$.

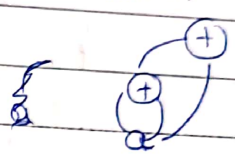
2 variables. Best 3AC. 3 variables

SSA : not SSA / not SSA, SSA ✓

syntax tree :



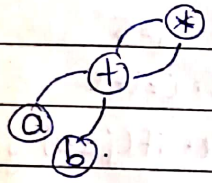
DAG.



3 nodes, 4 edges.

③ Find no. of nodes & edges in DAG for,

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



4 nodes, 4 edges.

④ Find no. of min. variables in 3AC for $x = (a+b) * (a+b)$

~~$t_1 = a + b$~~ $a = a + b$.

~~$t_2 = t_1 * t_1$~~. $a = a * a$. ← 3 variables (minimum).

(OR)

$b = a + b$

$b = b * b$

SSA code : $t_1 = a + b$ ← 4 variables (minimum)

$t_2 = t_1 * t_2$.

⑤ $x = a + b + c - a$? How many min. variables in 3AC ?

~~$a =$~~ $b = a + b$. $x = b + c$.

$b = b - a$. $b = b + c$ → 2 variables.

$b = b + c$.

(7) (1) $x = (a*b) + b + (a*c)$. How many min. variables in 3AC?

2nd independent. 1st

→ $t_1 = a*b$
$t_1 = t_1 + b$
$c = a*c$
$t_1 = t_1 + c$

4 variables.

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

(1) for(i=0; i<n; i++){
x = x+i;
}

High level.

1. i=0
2. if(i≥n) goto 5
3. x = x+i
4. i = i+1, goto 2
5. exit

3AC

(2)
```
for(i=0, j=5; i++, j>0; j++){
    x = x+i;
    y = y*j;
}
a = x+y;
```

1. i=0
2. j=5
3. i=i+1
4. if(j≤0) goto 8
5. x=x+i
6. y=y*j
7. j=j+1, goto 3
8. a=x+y;

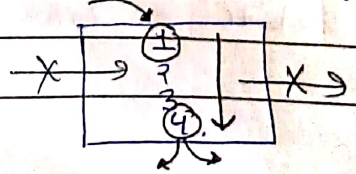
3AC

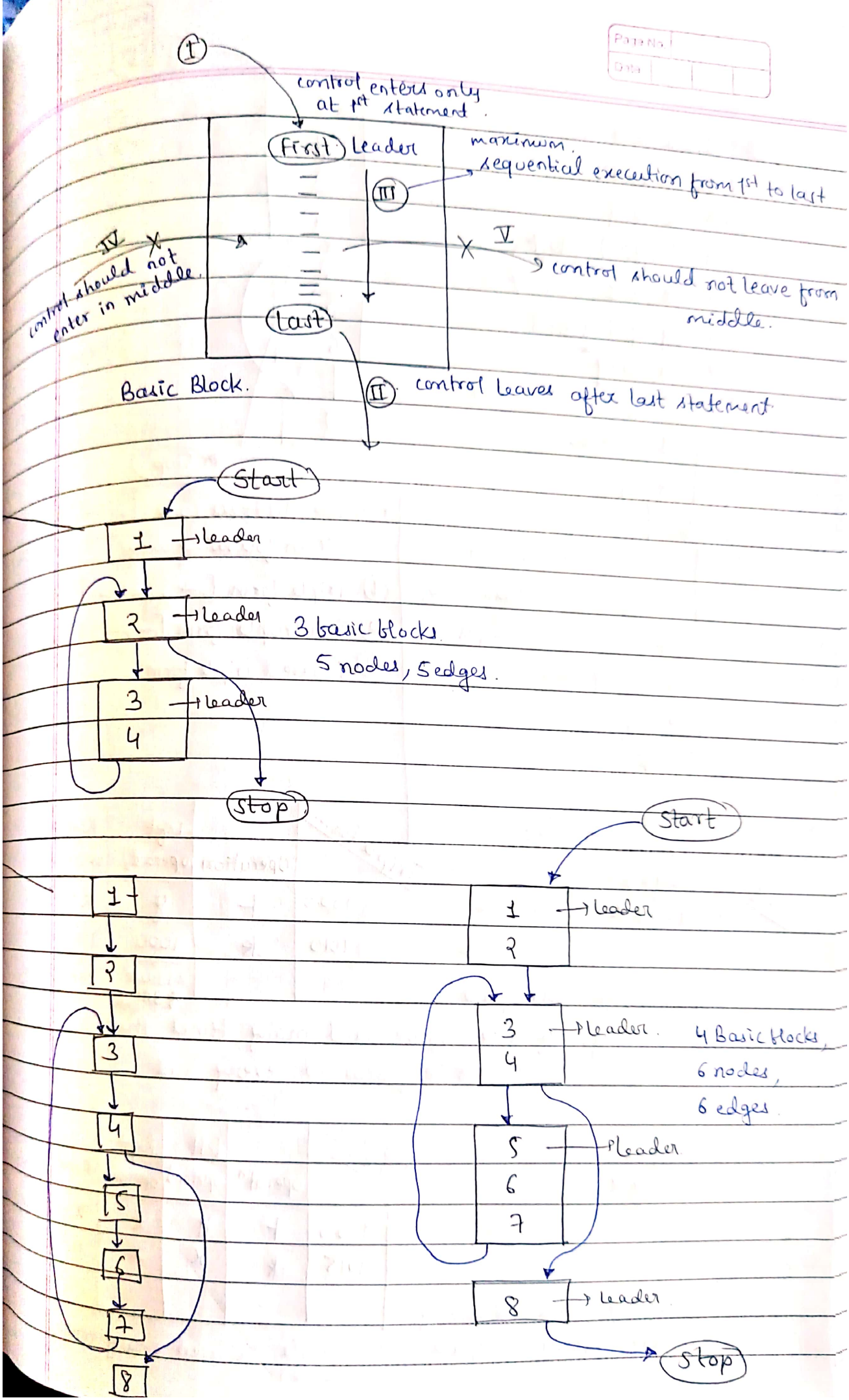
★ Control Flow Graph (CFG)-

- It is "Flow Graph"
- collection of nodes & edges
- collection of "basic blocks".

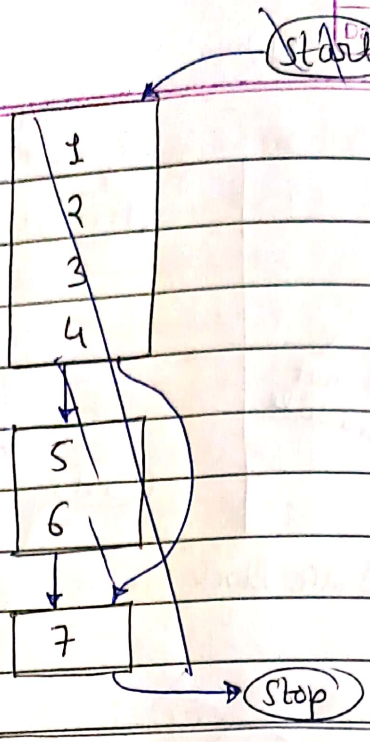
→ What is Basic Block? (sequence)

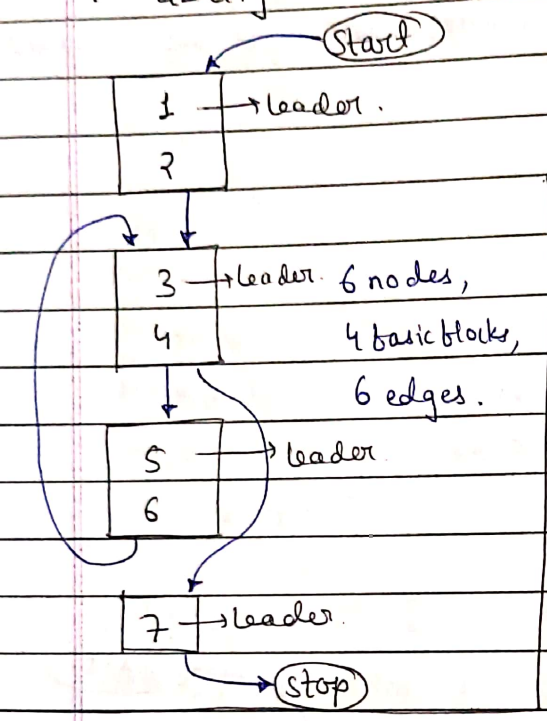
↳ It is maximum set of statements where control enters only at 1st statement and only leaves after last statement.





3. 1. x=a+1
2. y=x*2
3. z=b-y
4. if(y>z) goto 7
5. x=x-1
6. y=y-1 goto 3
7. a=a+y





6 nodes,
4 basic blocks,
6 edges.

- Three Address code: [we can store 3AC in following ways]
  1. Triple form. [—, —, —]
  2. Quadruple form. [—, —, —, —]
  3. Indirect triple form.

3AC:
x=y+z.
a=x*x.

Triple form

	Operation	Left operand,	Right operand$_2$
1000	+	y	z.
1010	*	1000	1000
		Address of 3AC code	Address of 3AC code

Advantage:- less space
Disadvantage:- If any lvalue used many times then it has to be calculated every time.

3AC:
x=y+z.
a=x*x.

Quadruple form

	Operatn	Left operand	Right operand	Result
1000	+	y	z	x
1015	*	x	x	a.

Advantage: Saves time if any variable (lvalue) is used many times.
Disadvantge: more space.

3AC:
x = y + z
a = x * x.

Indirect table

5000	1000
5010	1010.

Indirect triple form

	operatn	left operand	Right operand
1000	+	y	z.
1010	*	5000	5000

↳ address of address of 3AC code.

- **Code Generator -**

Intermediate code (m/c independent).
x = x + y
x & y are variables.

→ Code Generator (m/c dependent phase.)

→ Assembly code (m/c dependent)
ADD R1, [1000]
x & y are registers/ memory addresses.

x = z + 15

→ CG → ADD R, 15

Assembly instruction.

- Type of instruction.
- Type of operation.
- Type of addressing mode.
- operands: Data.
- operands: Memory Address / Register.

(depends on m/c architecture.)

Code Generator
↳ Register Allocation
↳ Graph coloring
↳ To allocate min. no. of registers.

- we have only 1 register. How many min. memory spills required?

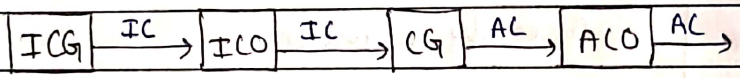
```
a = a + b
b = a * c.
```

3AC.

```
R ← a
mem1 ← b
R ← R + mem1
mem1 ← c
R ← R * mem1
```

} one memory spill.

- Code optimization [CO].
  - → what is CO?
  - → CO Techniques
  - → Data flow Analysis.
    - → Live variable analysis
    - → Reaching Definition Analysis.



~~Code~~ Code Optimization?

→ It may save time/space.

$x = y * 2$ ⟶ $x = y + y$

costlier — cheaper.

code optimization-

- At High Level. — In IDEs, In text editors
- At Intermediate Level. — In compiler.
- At assembly level. — In compiler.

Code optimization.

- → At Statement level.
- → At Basic Block level. } local optimization.
- → At loop level.
- → Intra-procedural.
- → Inter-procedural.] → global optimization.

→ Code optimisation techniques -

i) Constant folding -

$x = \underbrace{2*3}_{\text{folding.}} + y \implies x = 6 + y.$

ii) Cancellations -

$x = \cancel{a} + b*c - \cancel{a} \implies x = b*c.$

iii) Identity -

$x = y + 0 - z. \implies x = y - z$

$a = b * 1 + e \implies x = b + e.$

iv) Strength Reduction -

$\underbrace{x = y*2}_{\text{costlier.}} \implies \underbrace{x = y << 1. \text{ (OR) } x = y + y.}_{\text{cheaper.}}$

v) Common sub-expression & elimination -

↳ we can use DAG.

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

$\Downarrow$

$t_1 = a*b.$

$x = t_1 + t_1 - c.$

vi) Copy propagation (constant/variable)

$x = 20$ → constant

$y = x + a.$ constant propagation → $\boxed{y = 20 + a}$

~~ADD~~ $x = b$ variable

$y = x + a.$ propagation → $\boxed{y = b + a}$

⑦ Dead Code Elimination -

```
x = a+b
y = a*b        (dead code)
z = x+c
print(z).
```

⟹

```
x = a+b
z = x+c
print(z).
```

⑧ Loop optimizations

- → Code motion.
- → Induction variables Elimination.
- → Loop merging.
- → Loop unrolling.

- Code motion -

```
for(i=0; i<n; i++){
    x = a+i;
    y = c*b;        loop invariant code.
}
```

```
for(i=0; i<n; i++){
    x = a+i;
}
y = c*b;
```

- Induction Variables Elimination:-

Variables		Induction variables	
i	j=0;	(i)	for(i=0; i<n; i++){
j	for(i=0; i<n; i++){	~~(j)~~ (j)	x = x+i;
n	x = x+i;	x	y = y*i;
x	y = y*j;	y	z = x-a;
y	z = x-a;	z.	}
a.	j++;		
	}		

- **Loop Merge / Loop Combine / Loop fusion:**

```
for(i=0; i<n; i++){
    A[i]=i+1;
}
for(j=0; j<n; j++){
    B[j]=j*3;
}
```

⇒

```
for (i=0; i<n; i++){
    A[i]=i+1;
    B[i]=i*3;
}
```

- **Loop unrolling:**

```
for(i=0; i<=4n; i++){
    printf("a");
}
```

Double → half.

```
for (i=0; i<=2n; i++){
    printf("a");
    printf("a");
}
```

```
for (i=0; i<=n; i++){
    printf("a");
    printf("a");
    printf("a");
    printf("a");
}
```

* **Data Flow Analysis -**

- (i) Forward Analysis.
  - Reaching Definitions
  - Available Expressions.
- (ii) Backward Analysis.
  - Live variable Analysis.

- Data Flow Analysis -

→ What is Live variable?

1. $x = a + b$
2. $y = c * d$
3. $z = x - c$
4. $w = y * z$

Is x live at statement 2?
Ans. Yes.

S_i x is live.

(I) S_j uses x. (reads)

(II) path exist from S_i to S_j.

(III) There should be no write into x before S_j reads x.

- x is live variable at statement S_i iff

(Read) I) Some statement S_j reads x.

(Path) II) Path exist from S_i to S_j.

(No write) III) No write into x in between S_i & S_j before reading x.

Q.1. Find live variables at statement 1, 2, 3, 4.

1. $x = a + b$
2. $y = x * c$
3. $z = x + y$
4. $a = b * z$.

← a ✓ b ✓ c ✓ x y z.
1→1 1→2 1→2 X X X.
1→2→3→4.

(a, b, c) live at (1).

a b ✓ c ✓ x ✓ y z.
X 2→3→4 2→2 1→2, 2→3 X X

(b, c, x) live at (2)

a b ✓ c x ✓ y ✓ z.
X 3→4 X 3→3 3→3 X

(x, b, y) live at (3).

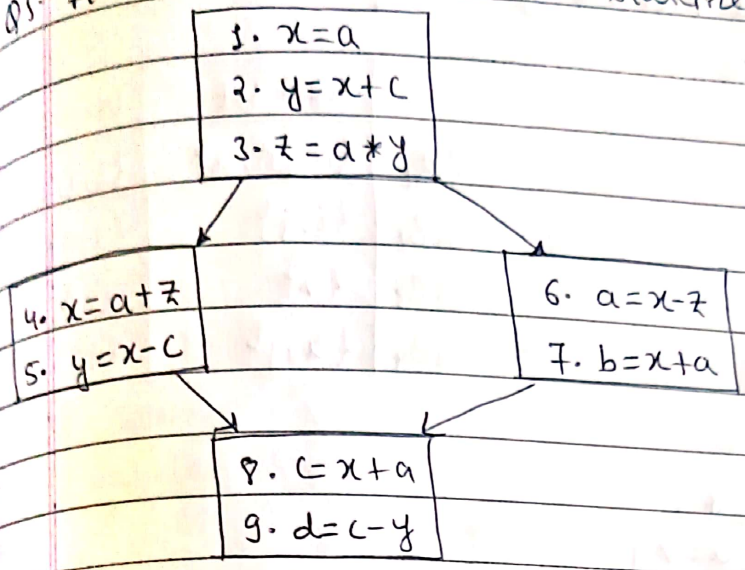
a b c x y z.
X ✓ X X X ✓

(b, z) live at (4)

c = x - c ← Is c live?

Yes.

Qs: find Live variables at all statement?



1. a b c d x y z
✓ X ✓ X X X X
(a, c) Live at (1) ✓

2. a b c d x y z
✓ X ✓ X ✓ X X
(a, c, x) Live at (2) ✓

3. a b c d x y z
✓ X ✓ X ✓ ✓ X
(a, c, x, y) Live at (3) ✓

4. a b c d x y z
✓ X ✓ X X X ✓
(a, c, z) Live at (4). ✓

6. a b c d x y z
X X X X ✓ ✓ ✓
✓ (x, y, z) Live at (6)

5. a b c d x y z
✓ X ✓ X ✓ X X
(a, c, x) Live at (5) ✓

7. a b c d x y z
✓ X X X ✓ ✓ X
✓ (a, x, y) Live at (7)

8. a b c d x y z
✓ X X X ✓ ✓ X
(a, x, y) Live at (8) ✓

9. a b c d x y z
X X ✓ X X ✓ X
✓ (c, y) Live at (5)

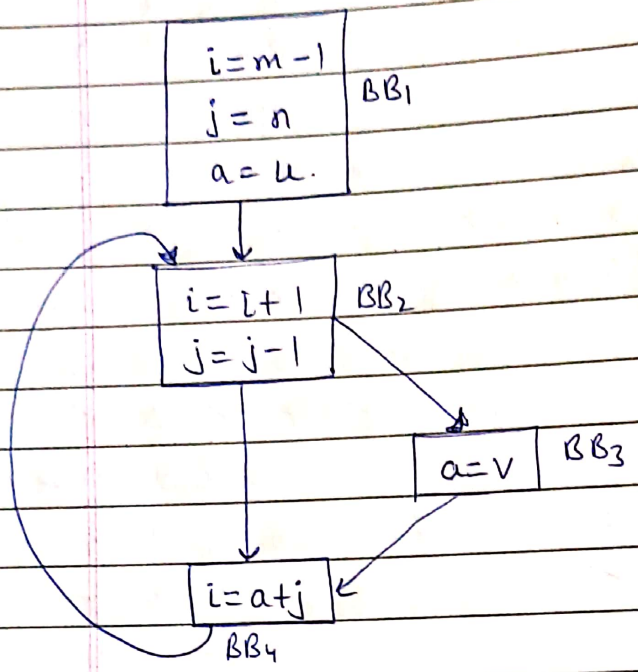
* GEN Set & KILL Set for Basic Block:
(use) (definition)

Basic Block	
	(x) = a+b
	(c) = x-c
	(z) = x*y

$KILL_K$ = {x, c, z}
= Set of variables which are defined in Basic Block
= {v | v is defined in BB}

GEN_K = {a, b, c, y}
= {set of variables in Basic block where every variable used at some statement in Basic Block (BB) but no write into it before read}

- Find GEN set and KILL set for every Basic Block.

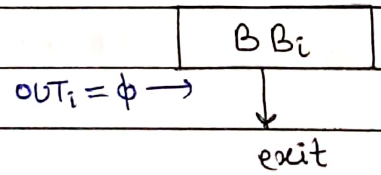


	GEN set	KILL Set.
BB_1	{m, n, u}	{i, j, a}
BB_2	{i, j}	{i, j}
BB_3	{v}	{a}
BB_4	{a, j}	{i}

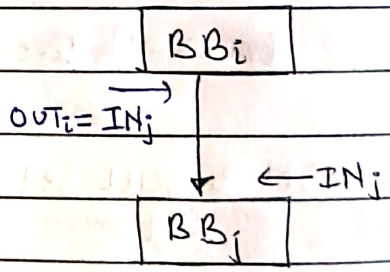
- IN set & OUT set for Basic Blocks!

→ How to compute OUT set for BB_i?

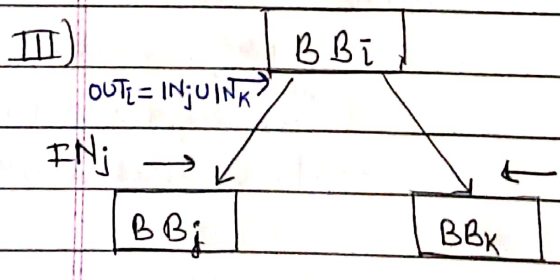
I).



II).



III)

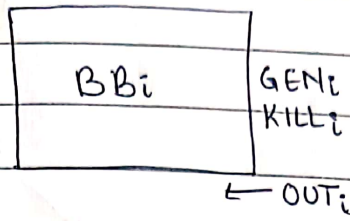


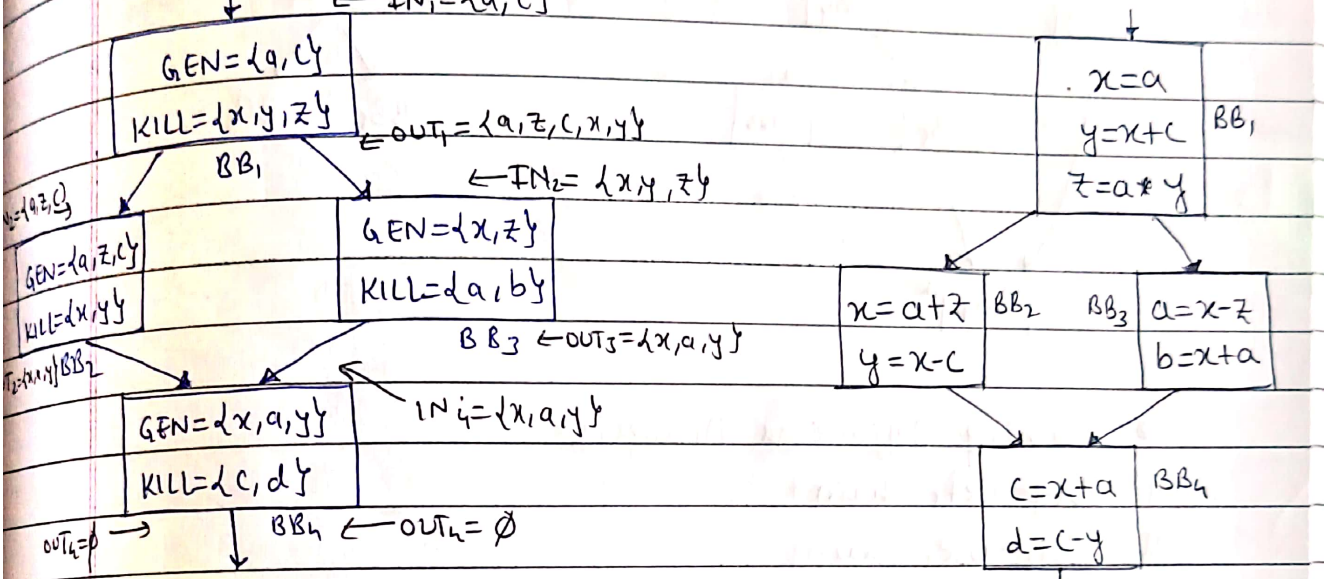
$OUT_i = IN_j \cup IN_K$.

How to compute IN set for BBi ?

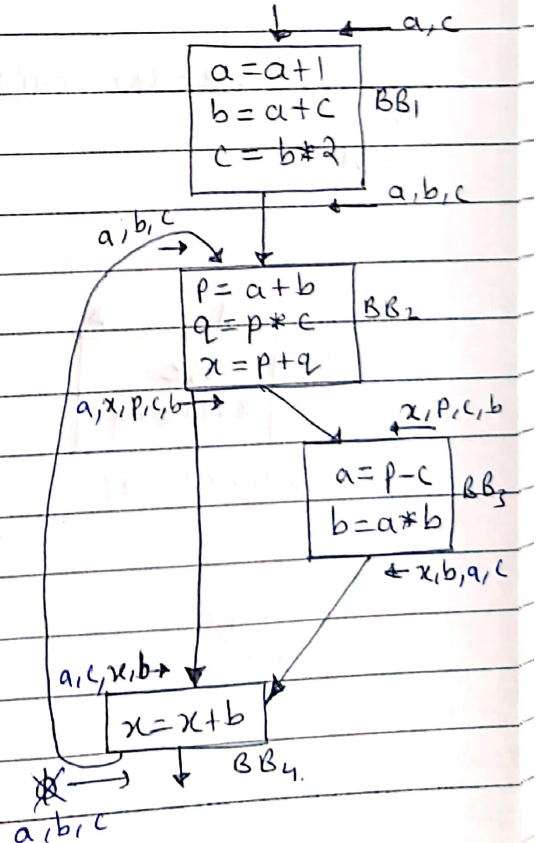
$IN_i = (OUT_i - KILL_i) \cup GEN_i$

↳ shortcut OKG GOD.

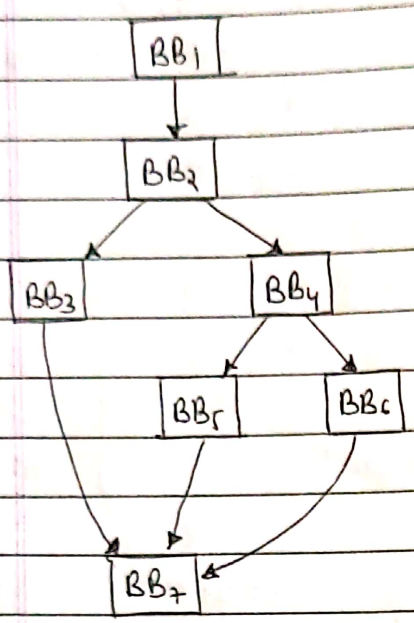




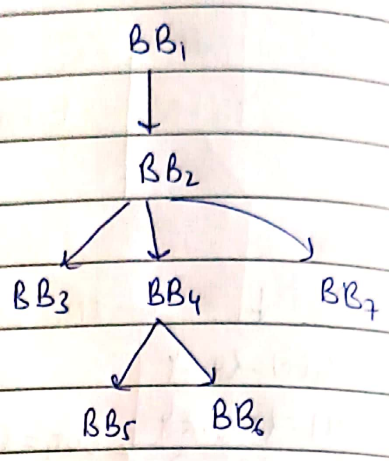
	GEN Set	KILL Set	OUT Set	IN Set
BB1	a, c	a, b, c	a, b, c	a, c
BB2	a, b, c	P, a, x	i) x, P, c, b ii) a, x, P, c, b	i) a, b, c ii) a, c, b
BB3	P, c, b	a, b	i) x, b ii) x, b, a, c	i) x, P, c, b ii) x, P, c, b
BB4	x, b	x	i) ϕ ii) a, b, c	i) x, b ii) a, c, x, b



→ Control Flow Graph-



Dominator Tree

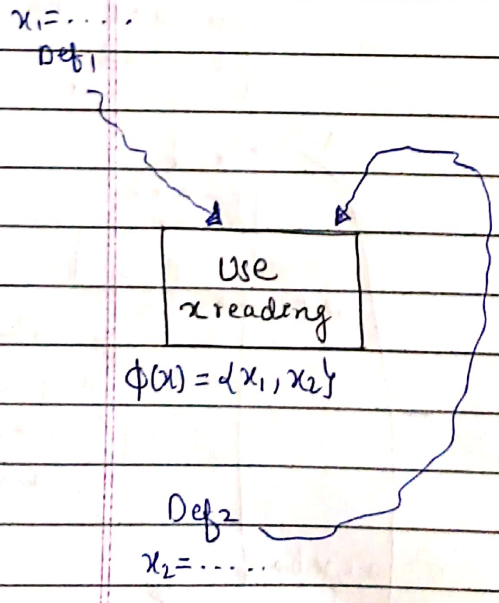


- Reaching Definitions Analysis-

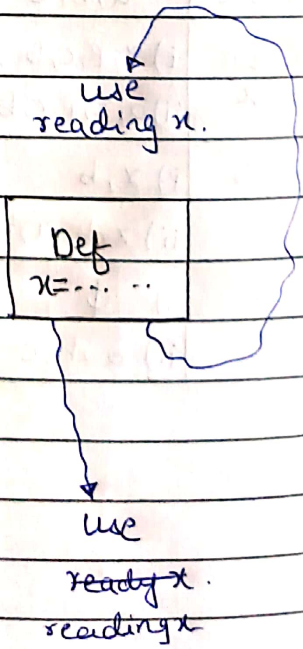
I) Use - Def, chain

II) Def - use, chain

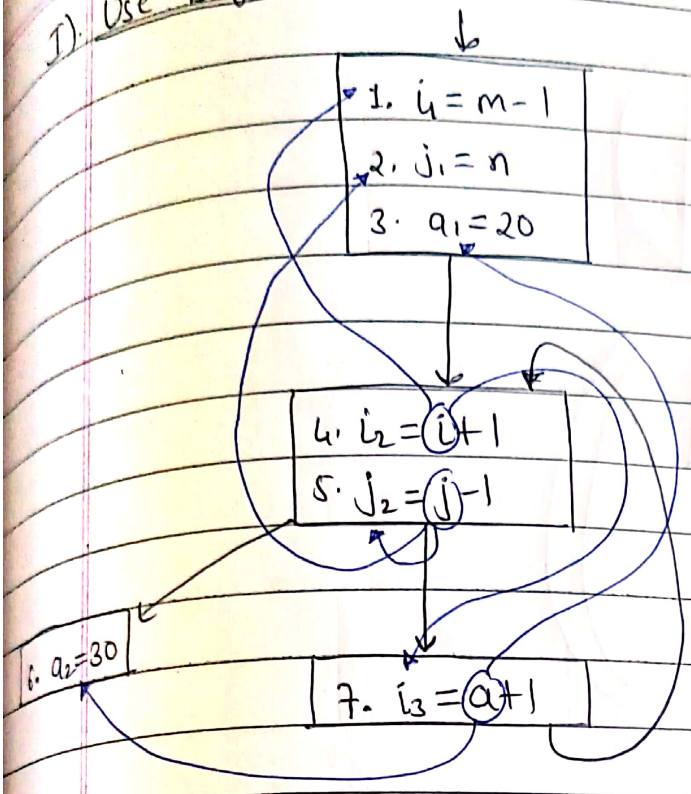
- Use-def chain



- Def-use chain



I). Use-Def chains-



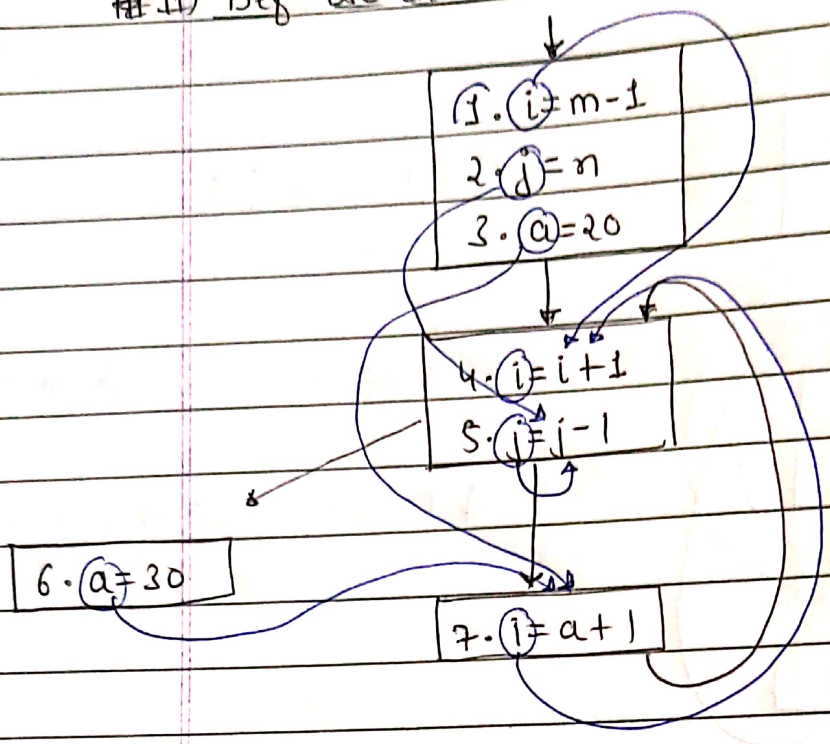
$\phi(j) = \{j_1, j_2\}$ at statement 5.

$\phi(i) = \{i_1, i_3\}$ at statement 4.

$\phi(a) = \{a_1, a_2\}$ at statement 7.

statement	USE	Definitions.
1	m	–
2	n	–
4	i	1, 7
5	j	2, 5
7	a	3, 6.

II) Def-Use chain



Statement	Def	Uses
1	i	4
2	j	5
3	a	7
4	i	x
5	j	5
6	a	7
7	i	4