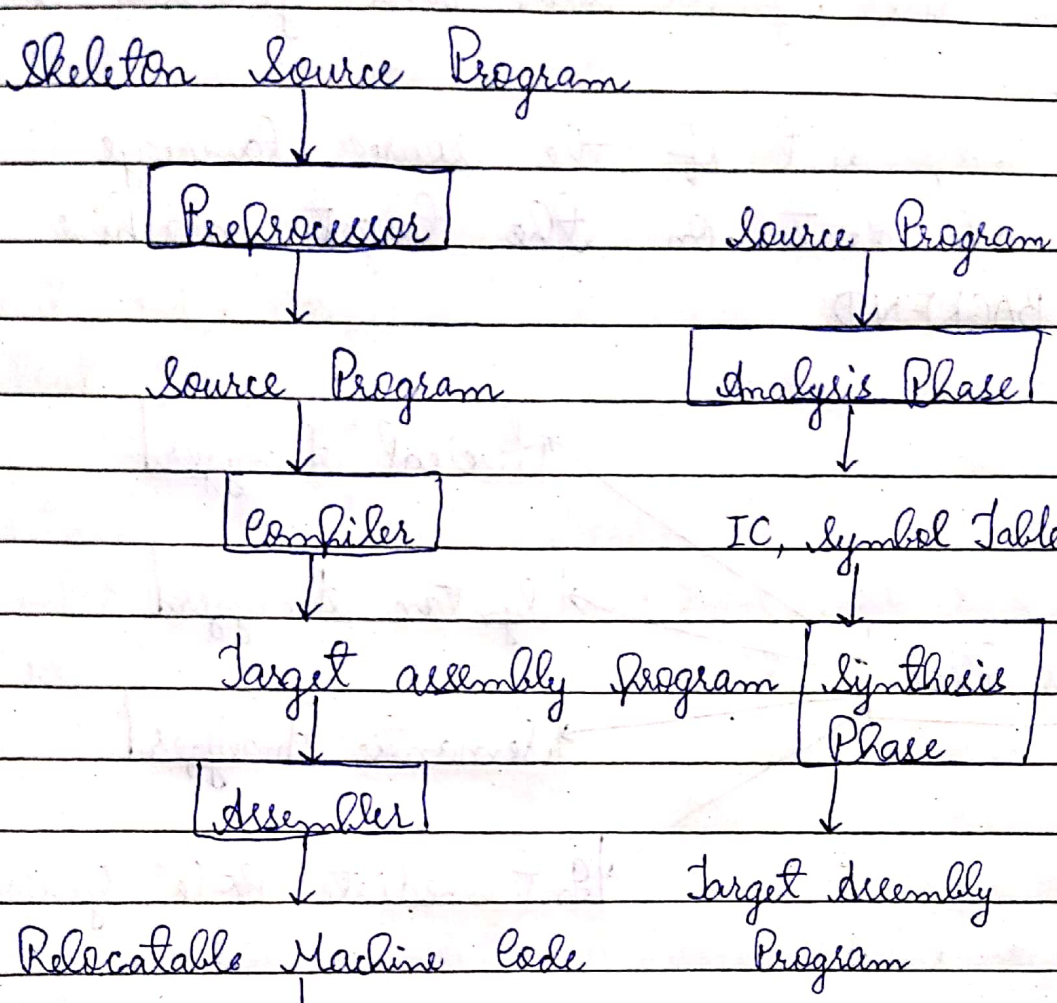
* Phases of Compiler

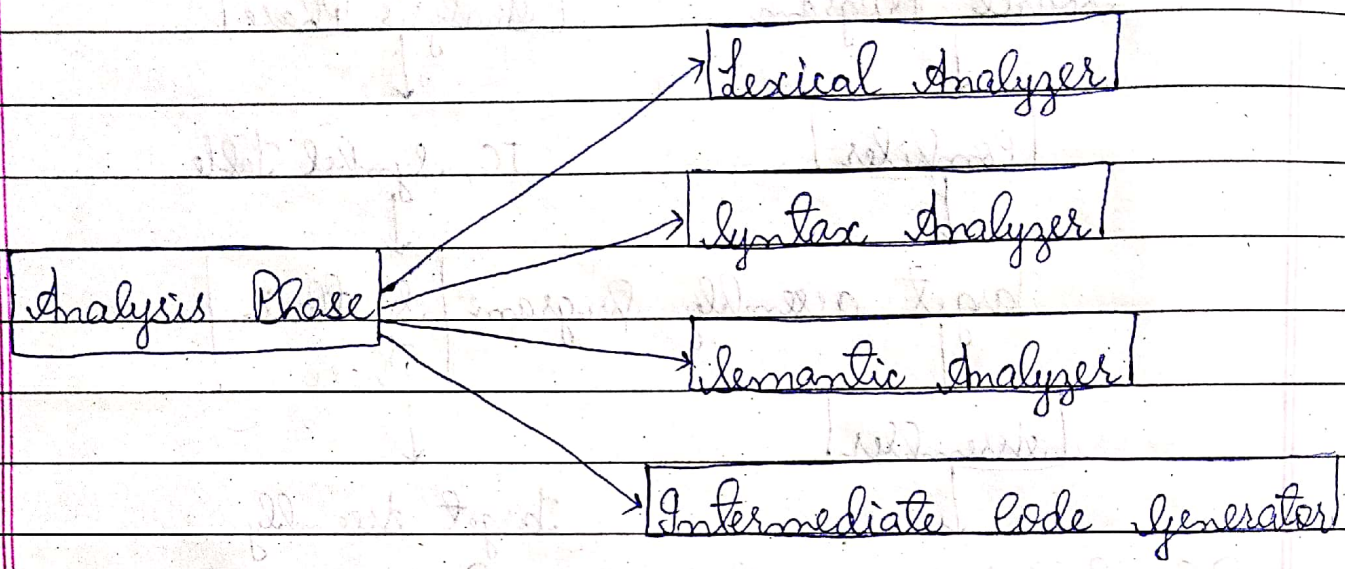


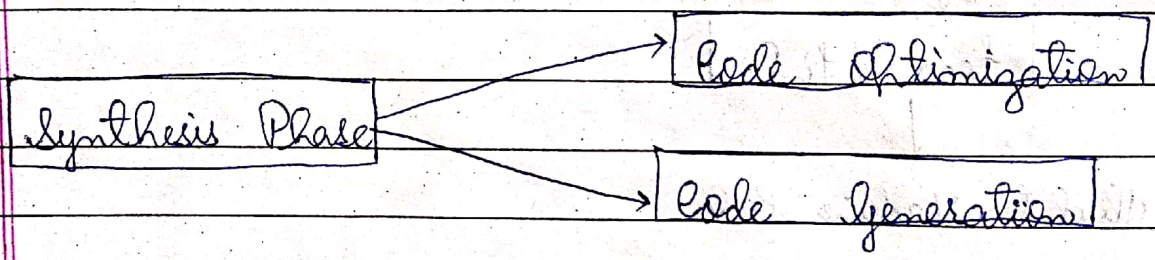
Analysis Phase:

1. Perform analysis of input code
2. Generates errors and warnings
3. Largely dependent on the source language
4. Largely independent of the target machine
5. Creates FRONT END

Synthesis Phase:

1. Takes IC, uses symbol table and generates target program
2. Largely independent of the source language
3. Largely dependent on the target machine
4. Creates BACKEND





1. Lexical Analysis:

→ Grouping into tokens (sequence of characters with meaning)
→ Identifies valid words or lexemes in source program
→ Removes white spaces, tabs, newlines
→ Passes sequence of tokens to syntax analysis phase
→ Implemented as a finite automata
→ Error generated: invalid identifier

→ Example:

(i) position = initial + rate * 60

Output of lexical phase:

<id, 1> <=> <id, 2> <+> <id, 3> <*> <number, 60>

(ii) int n1, n2;
float f;
n1 = n2 + f * 5 - n2;

Output of lexical phase:

<int> <id, 1> <id, 2> <;> <float> <id, 3> <;>
<id, 1> <=> <id, 2> <+> <id, 3> <*> <number, 5>
<-> <id, 2> <;>

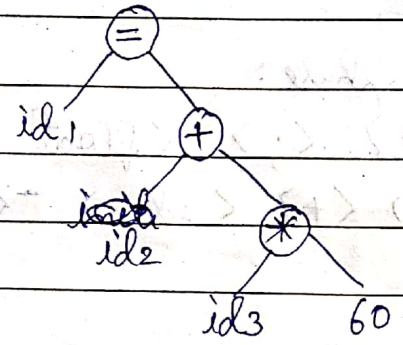
Limitations:

→ Can identify validity of lexemes (individual words) only
→ Not powerful to analyze expression or statement (can't match a pair of parenthesis)

2. Syntax Analysis (Hierarchical Analysis, Parsing)

→ Takes sequence of tokens from lexical analyzer
→ Checks syntactic correctness of expressions / blocks / statements / functions / program
→ Successful if it can identify grammar rules (CFG, CSG) for a sequence of tokens.
→ Generates a parse tree from a sequence of tokens
→ If parse tree is complete ⇒ construct is correct
→ If parse tree is incomplete ⇒ syntax error in the construct

→ Uses recursive grammar rules
→ Errors generated: missing parenthesis, missing semicolon, missing operand
Example:
(i) position = initial + rate * 60
Tokens: position (id1), '=', initial (id2), '+', rate (id3), '*', 60



3. Semantic Analysis
→ If program is syntactically correct, then check for semantic (meaning) correctness of the program
→ Semantics depends on the programming language
→ Checks for semantic errors
→ Gathers type information
→ Uses parse tree generated by the syntax phase
→ Common checks are:
(i) Type of variables and type-casting
(ii) Applicability of operators on operands - each operator has operands permitted by the source language (string operators, only int index in array)
(iii) Scope of variables and functions
(iv) Determine definition of variables and functions (function overloading)

Example:

```
int a=5, b=2;
float f;
f = a/b;
f = 5/2.0; // inbuilt - type casting
```

→ Errors:

(i) float array index
(ii) Cannot convert 'int' to 'string'
(iii) Undeclared variable

4. Intermediate code Generator

→ After semantic analysis, Intermediate code (IC) of source program will be created (low-level or machine-like IR)

→ Properties:

(i) Easy to produce
(ii) Easy to translate into target code

→ Variety of forms:

(i) Three Address code (3A code)
(ii) Postfix representation

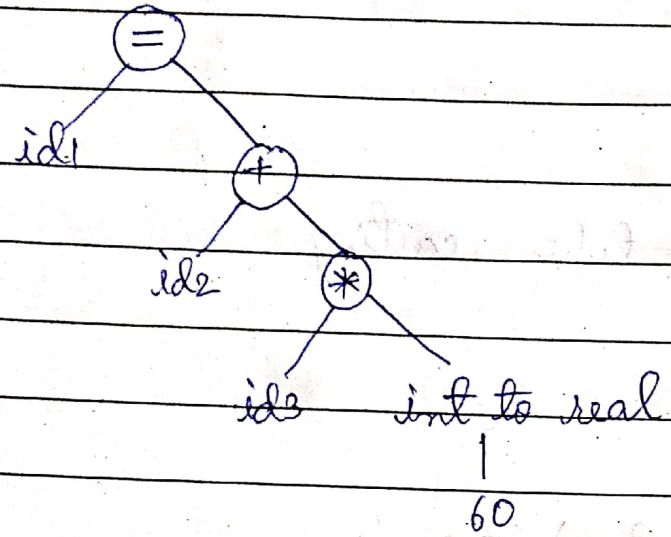
(i) Three Address code (3A code)

→ Each instruction has

(a) Maximum three operands
(b) Maximum one operator in addition to assignment

→ Compiler will decide order of evaluation
→ Compiler will generate a temp name to hold computed results

→ Example: $id_1 = id_2 + id_3 * 60$



t_1 = int to real (60)
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

5. Code Optimization
→ Improves the Intermediate code for
(i) Faster execution of machine code
(ii) Space optimization
→ Optimizing Compilers
→ Machine independent code optimization
→ Example: (i) 60.0 instead of int to real (60)
(ii) No need of t_3

$t_1 = id_3 * 60.0$
$id_1 = id_2 + t_1$

→ Removes dead code like unreachable code
→ If anything is written after return statement in function
→ Loop optimizations - major source of optimization
→ Saving even a single line in loop improves execution time significantly

→ Example,

```
b = 5;
for (i=0; i<10000; i++) {
        a = b + 100 - i * 2;
}
```

Optimized code:

```
b = 5;
c = b + 100;
for (i=0; i<10000; i++) {
        a = c - i * 2;
}
```

6. Code Generation

→ Takes an optimized Intermediate code as an input and converts it to the target code

→ The target code can be:

(i) Relocatable Machine code

(ii) Assembly code

→ Memory locations are selected for each variables of a program

→ Intermediate code is translated to a sequence of machine instructions

→ Part of backend of the compiler design

→ Target code generation depends on:

(i) Availability of machine instructions

(ii) Addressing modes

(iii) Number of registers (general and special purpose)

* Symbol Table Management:

→ Interacts with all phases - used as a reference table by all the phases

→ Compiler needs

(i) To record the identifiers (variables) used in source program

(ii) To collect information about various attributes of each identifier

(iii) Attributes for variable names - Name, type, class, size, relative offset within program, scope, value

(iv) Attributes for function names - Name, number of arguments, types of arguments, return type, method of passing each parameter

→ It is a data structure containing a record for each identifier with fields for the attributes of the identifier.

→ Lexical analyzer: detects identifier and enters in the symbol table, but can't determine all its attributes.

→ The remaining phases enter information about identifiers into symbol table and uses the entered information in different ways.

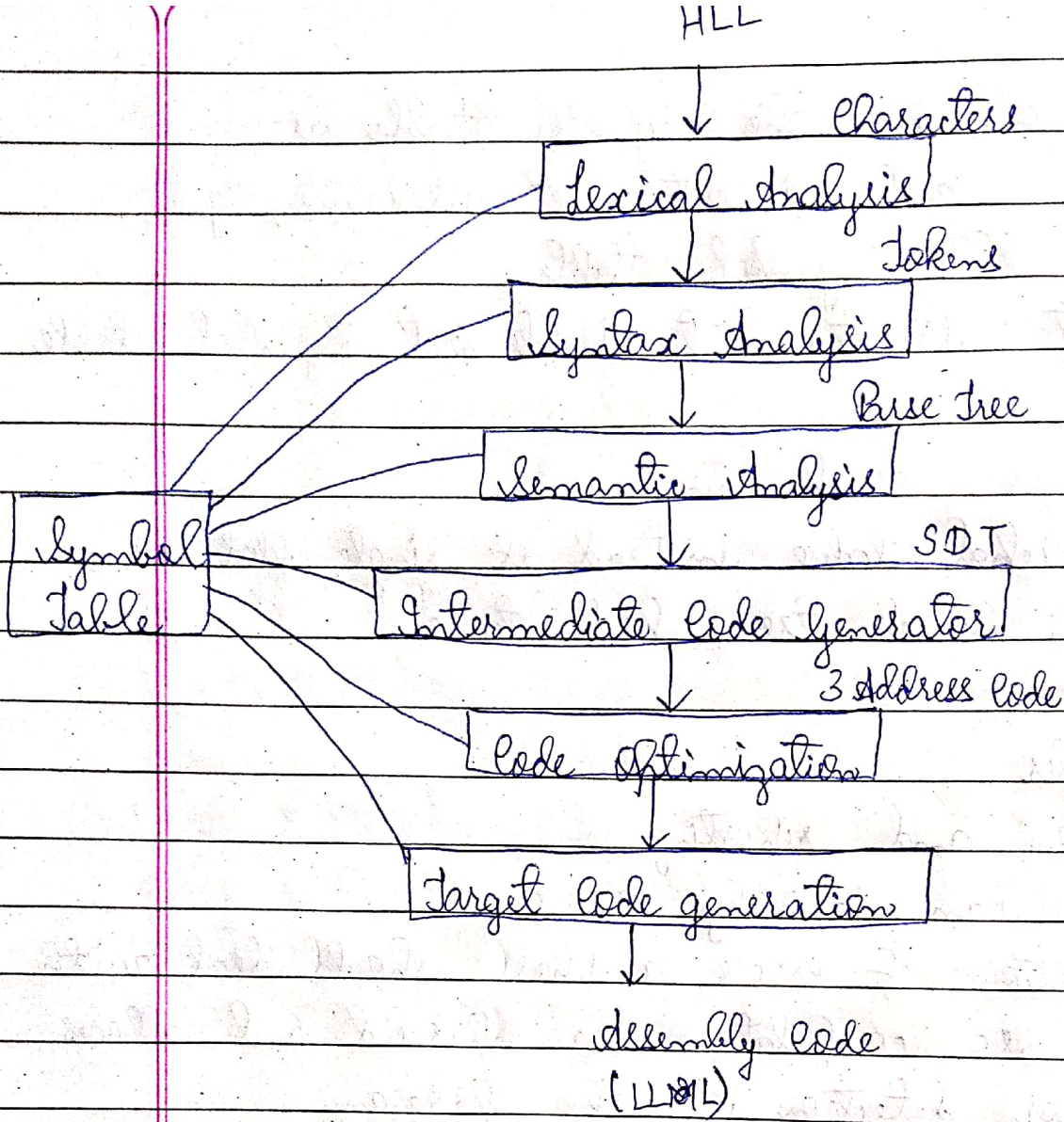
→ Semantic analyzer and Intermediate code generation needs to know type of variable.

→ Code generator: enters and uses details about the storage assigned to identifier.

→ During compilation, all phases often look up this table for definition of variables, examples Variable used is defined before that or not?

→ Hence, operations in the symbol table are allocate, free, search, insert, set attribute, get attribute in the symbol table.
→ Use the data structure to implement symbol table such that
(i) It minimizes search time
(ii) Use hierarchichal table instead of single flat one
(iii) Linear list, search trees, hash table

* Error Handler
→ Error detection and reporting
→ Each phase can identify errors
→ After detection of errors, a phase should deal with that error so compilation can proceed and allows further error detection in the program
→ Semantic error - parser can progress
→ Syntax error - parser reaches in error state
→ It indicate regarding type of error
→ Undo some processing already carried out by parser - error recovery
→ Error detection and recovery - vital role in overall operations of compiler



* Lexical Analysis (Lexer, Tokenizer, Scanner)

1. Tokenization
2. Give error message → Exceeding length
   → Unmatched string
   → Illegal character
3. Eliminate comments, white space (tab, blank space, new li

Tokens → Identifier (variables)
Tokens → Separator (brackets)
Tokens → Keyword (int, float)
Tokens → operators (+, =)
Tokens → constants
Tokens → special characters

Example,

1. int max ( )
   1 2 3 4
{
5 /* find max of a and b */ → comment
int a = 30, b = 20; 14
if (a < b) 20
return (b); 25
else
return (a); 31
} (32)

2. printf (" i = %d, & i = %x ", i ; & i);
   3 4 5 6 7 8 9 10

3. int main) (
} 5
x = y + z; 11
int x, y, z; 18
printf (" sum %d %d ", x);
{ (26)

4. main ( ) { 4
a = b + + + - - - - + + + = = ;
5 6 7 8 9 10 11 12 13 14 15
printf (" %d %d, a, b);
} 15 16 17

* Finding First() and Follow() from given grammar

First(A) contains all terminals present in first place of every string derived by A.
First(terminal) = terminal
First(ϵ) = ϵ

E.g. S → abc | def | ghi
First(S) = {a, d, g}

E.g. S → ABC | ghi | jkl (A, g, j) ⇒ ~~(a, g, j)~~ (a, b, c, g, j)
A → a | b | c (a, b, c)
B → b (b)
D → d (d)

E.g. S → ABC
A → a | b | ϵ
B → c | d | ϵ
C → e | f | ϵ
F(C) = (e, f, ϵ), F(B) = (c, d, ϵ), F(A) = (a, b, ϵ)
F(S) = F(A) = (a, b, ϵ) ∪ F(B) = (c, d) ∪ F(C) = (e, f, ϵ)
= (a, b, c, d, e, f, ϵ)

e.g. $E \to TE'$

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

$T \to FT'$

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

$F \to id \mid (E$

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

follow (A) contains set of all terminals present immediate in right of A. It never contains ϵ.
follow of start symbol is $.,

e.g. $S \to ACD$

$C \to a \mid b$

Fo (A) = first (C) = {a, b}

Fo (D) = Fo (S) = {$}

e.g. $S \to aSbS \mid bSaS \mid \epsilon$

Fo (S) = {$, b, a}

e.g. $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

Fo (A) = {a, b}, Fo (B) = {b, a}

E.g. $S \to ABC$ (A, B, C: ϵ ϵ ϵ)

$A \to DEF$

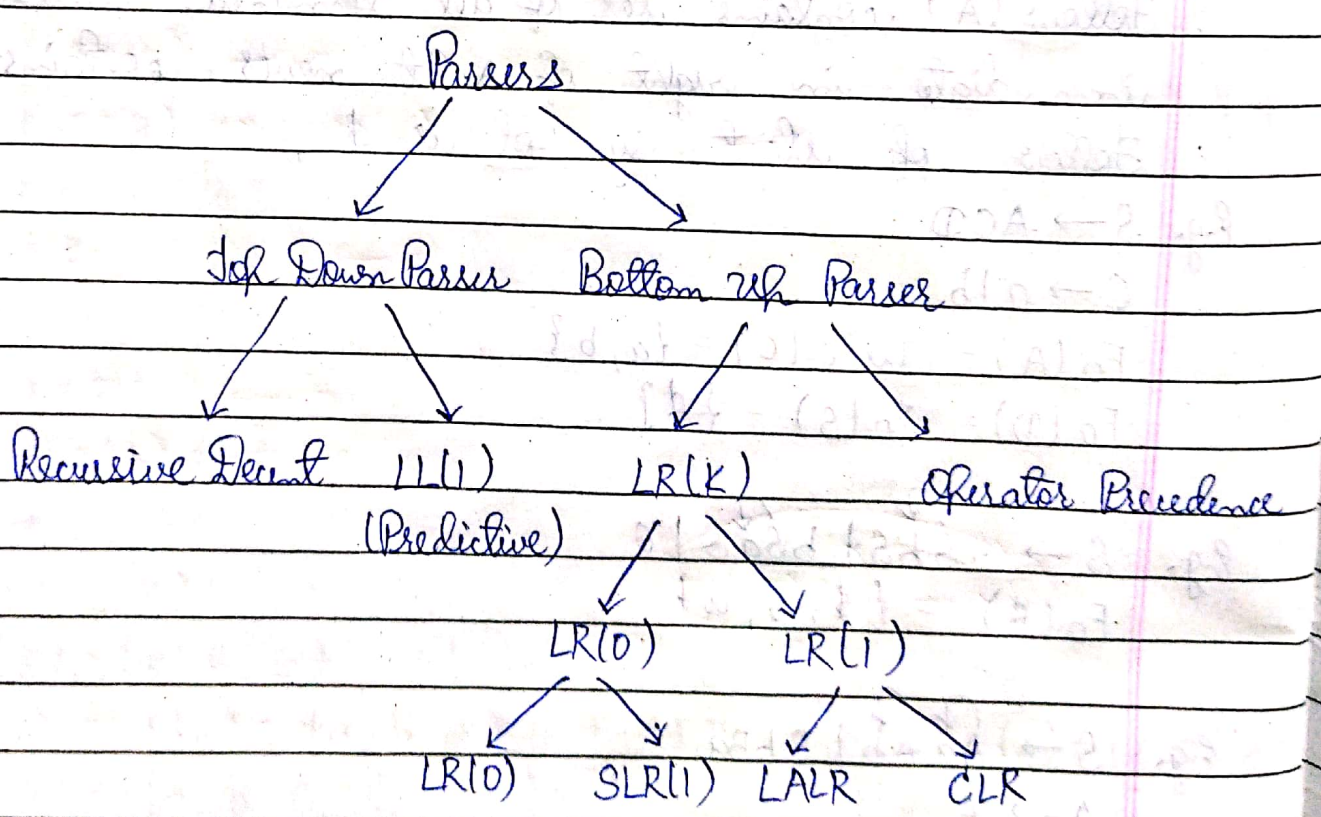
$B \to \epsilon$, $C \to \epsilon$, $D \to \epsilon$, $E \to \epsilon$, $F \to \epsilon$

Fo(A) = first(B) = first(C) = follow(S) = {$}

(first(B) $\downarrow \epsilon$, first(C) $\downarrow \epsilon$)

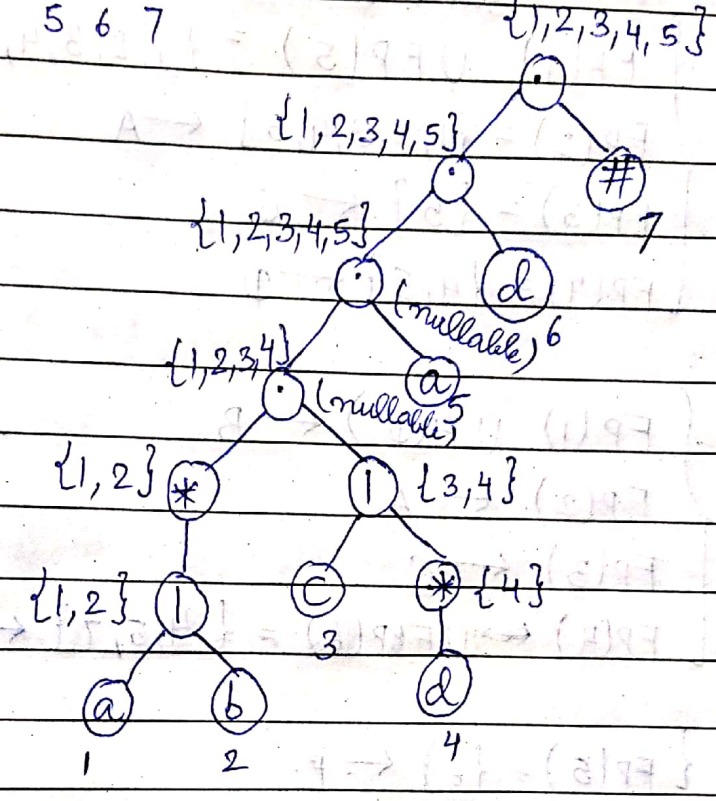
* Passing

It is a process of deriving string from a given grammar.



* RE to DFA using Syntax Tree method

Eg. $(a|b)^*.(c|d^*)^*.ad\#$
$\quad\quad 1\ 2 \quad\quad 3\ 4 \quad 5\ 6\ 7$



Union of

FIRSTPOS(1) = FIRSTPOS(child nodes)

FIRSTPOS(*.*) = FIRSTPOS(child node)

FIRSTPOS(.) = FIRSTPOS(left node)

(OR)

Union of FIRSTPOS(left node) and FIRSTPOS(right node) if left node contains * (that is nullable)

FOLLOWPOS - which symbols can follow the given symbol

FIRSTPOS(root) = {1,2,3,4,5} ← A

FOLLOWPOS(1) = {1,2,3,4,5}

FOLLOWPOS(2) = {1,2,3,4,5}

FOLLOWPOS(3) = {5}

FOLLOWPOS(4) = {4,5}

FOLLOWPOS(5) = {6}

FOLLOWPOS (6) = {7}

FOLLOWPOS (7) = ~~#~~ ∅

same variable

A
- FP(1) ∪ FP(5) = {1,2,3,4,5,6} ← B
- FP(2) = {1,2,3,4,5} ← A
- FP(3) = {5} ← C
- FP(4) = {4,5} ← D

B
- FP(1) ∪ FP(5) ← B
- FP(2) ← A
- FP(3) ← C
- FP(4) ~~←~~ ∪ FP(6) = {4,5,7} ← E

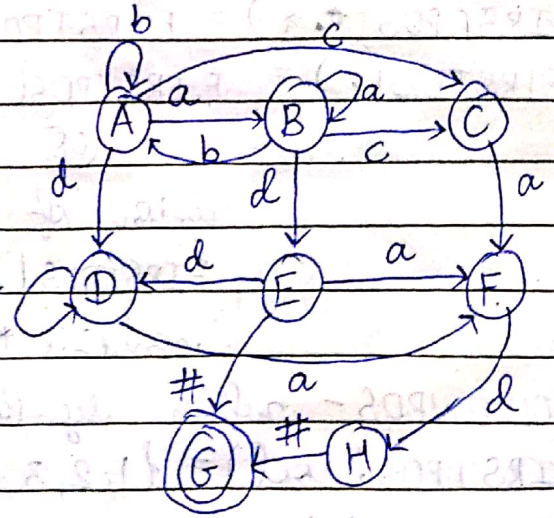
C
- FP(5) = {6} ← F

D
- FP(4) ← D
- FP(5) ← F

E
- FP(4) ← D
- FP(5) ← F
- FP(7) ~~=~~ ~~#~~ ∅ ← G

F
- FP(6) = {7} ← ~~G~~ H

G
- FP(7) ← #



* LL(1) Parser (Top - Down)

(*) How to check whether the string is LL(1) or not

E.g. $S \to (L) \mid a$

$L \to SL'$

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

$F(S) = \{(, a\}$ $\quad$ $FO(S) = \{\$,), ,\}$

$F(L) = F(S) = \{(, a\}$ $\quad$ $FO(L) = \{)\}$

$F(L') = \{\epsilon, ,\}$ $\quad$ $FO(L') = \{)\}$

Make Parsing table where rows are non-terminals and columns are terminals and $.

	(	)	a	,	$
S	$S \to (L)$	e	$S \to a$	e	e
L	$L \to SL'$	e	$L \to SL'$	e	e
L'	$L' \to \epsilon$ (FOLLOW)	e	e	$L' \to , SL'$	e

Make entry for every row of the table where terminal's first is non-terminal (column). The entry would be equation whose first is non terminal. If first contains ϵ then use follow of that non-terminal. All other entries will be e. If one cell maximum one entry then the grammar will be accepted by LL(1) Parser.

The above grammar is LL(1).

E.g. $S \rightarrow aSbS \mid bSaS \mid \epsilon$

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

$FO(S) = \{b, a, \$\}$

	a	b	$
S	$S \rightarrow aSbS$	$S \rightarrow bSaS$	
	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Here, there are two entries twice for one cell so the above grammar is not LL(1).

* How LL(1) parser works?

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

	a	b	$
S	$S \rightarrow AA$	$S \rightarrow AA$	e
A	$A \rightarrow aA$	$A \rightarrow b$	e

String : abab

Initially, there is first non-terminal on the top of stack followed by $ and string is also followed by $. So at every point, non-terminal at top of stack is replaced with the function present in row of non-terminal of table and column of terminal of string where pointer is. If top of stack and string pointer has same terminal

then stack is popped and string pointer is incremented. If there are no rules present for a given non-terminal in stack according to table then grammar is not LL(1). If top of stack becomes $ and string pointer also becomes $ then string is LL(1).

Note: while replacing non-terminal with rule, all the contents are added in reverse order.

	Stack	a b a b $	Rule
①	$ S		S → AA
	$ AA		A → aA
	$ AAa (pop & increment)		
②	$ AA		A → b
	$ Ab (pop & increment)		
	$ A		A → aA
	$ Aa (pop & increment)		
	$ A		A → b
	$ b (pop & increment)		
	$ (accepted)		

① Check table for rules in row S and column a.
② Check table for rule in row A and column b.

* Conditions for Top-Down Parser
The grammar should not contain:
- Left-Recursion
- ~~Ambiguity~~ (Left-Factoring) Non-Determinism

1. Eliminating Left Recursion

E.g. $E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to (E) \mid id$

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

$\Downarrow$

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \dots \mid \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

$E \to TE'$ $\qquad$ $T \to FT'$
$E' \to +TE' \mid -TE' \mid \epsilon$ $\qquad$ $T' \to *FT' \mid /FT' \mid \epsilon$

E.g. S → aAbA | Sae | Bba | Sdd | bS
A → a
B → b

S → abAS' | Bba S' | bSS'
S' → aeS' | ddS' | ε
A → a
B → b

2. Eliminating Left - factoring

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n$

$\Downarrow$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$

e.g. $S \rightarrow Aa | Ab | AS$

$S \rightarrow AS'$

$S' \rightarrow a | b | S$

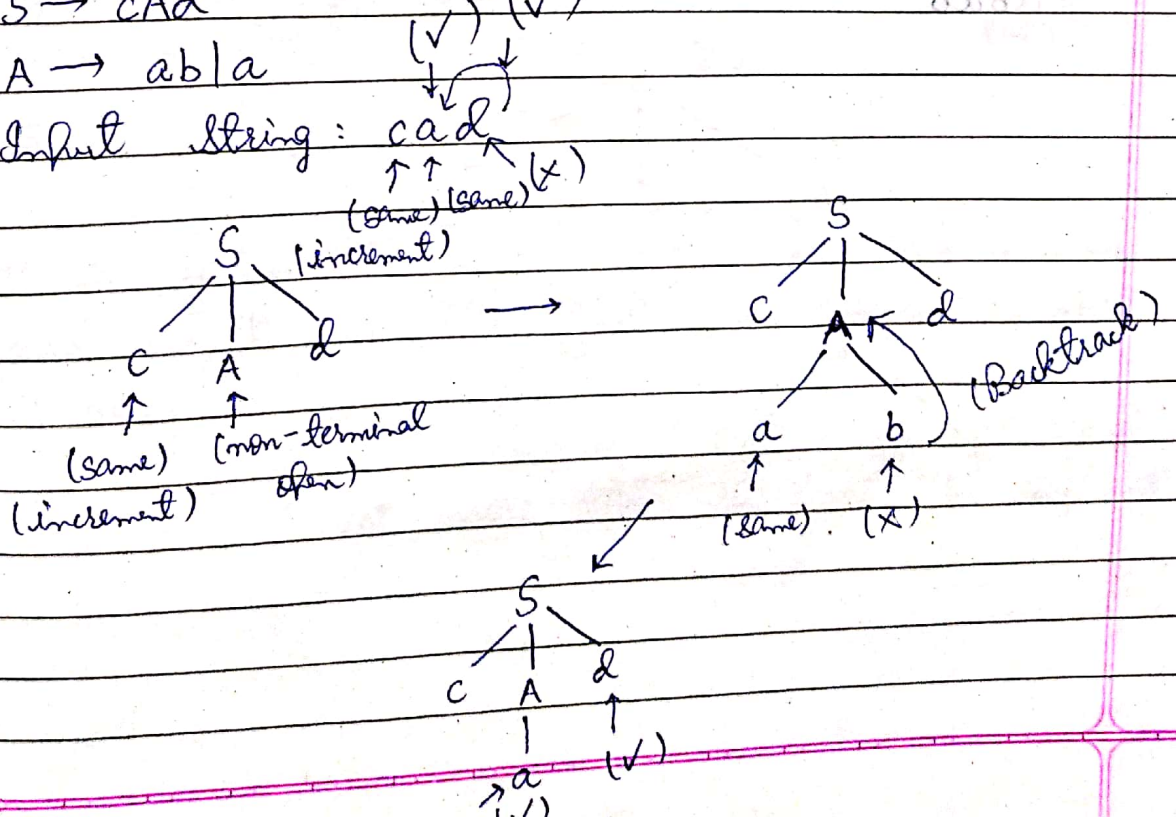
* Recursive Decent Parser (Top - Down)

A Recursive Decent Parser is a top-down parser built from a set of mutually recursive procedures where each such procedure implements one of the non-terminals of the grammar.

e.g. $S \rightarrow cAd$

$A \rightarrow ab | a$

Input string : cad



E.g. $S \rightarrow aSbS \mid bSaS \mid \epsilon$

String : abab

$S \rightarrow aSbS$

$\Rightarrow a\ aSbS\ bS$ × (Backtracking)

$\Rightarrow a\ bSaS\ bS$

$\Rightarrow ab\ aSbS\ aS\ bS$ × (Backtracking)

$\Rightarrow ab\ bSaS\ aS\ bS$ × (Backtracking)

$\Rightarrow ab\ \epsilon\ aSbS$

$\Rightarrow aba\ aSbS\ bS$ × (Backtracking)

$\Rightarrow aba\ bSaS\ bS$ × (Backtracking)

$\Rightarrow aba\ \epsilon\ bS$

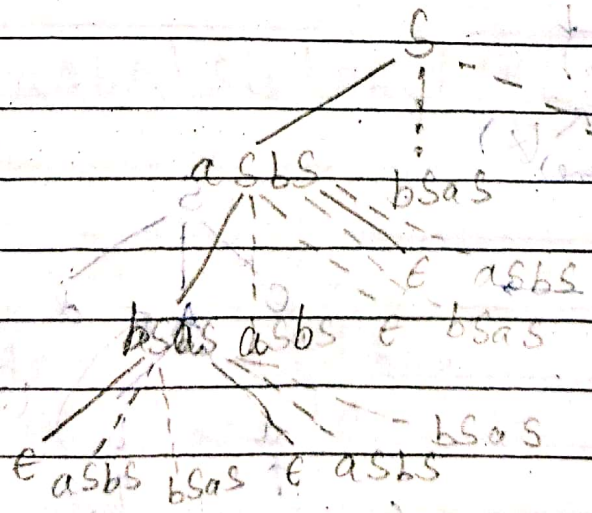
$\Rightarrow abab S$

$\Rightarrow abab aSbS$ × (Backtracking)

$\Rightarrow abab\ bSaS$ × (Backtracking)

$\Rightarrow abab\ \epsilon$

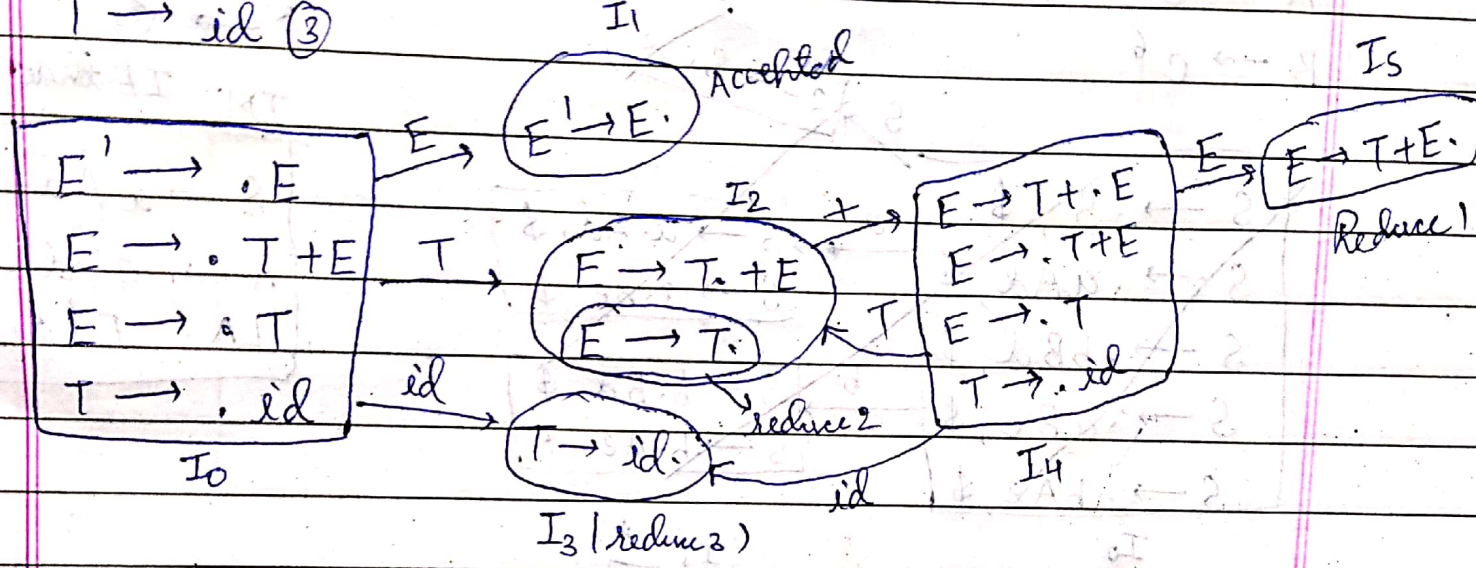
$\Rightarrow abab$



* SLR(1) (Bottom - up)

E.g. $E \to T+E$ (1) | T (2)

$T \to id$ (3)



State	id	+	$	E	T
0	S3			1	2
1			Accepted		
2		S4	r2 FO(E)		
3		r3	r3		
4	S3			5	2
5			r1		

Find follow of LHS of reduce equation

$E \to T.$

$FO(E) = \{\$\}$

$FO(T) = \{\$, +\}$

Write reduce in row having same state number and shift in the row from where equation is going.

SLR(1) ✓ (Not more than one entries in any cell)

* CLR (Bottom - up)

$S \xrightarrow{1} aAd \mid \xrightarrow{2} bBd \mid \xrightarrow{3} aBe \mid \xrightarrow{4} bAe$

$A \xrightarrow{5} c$

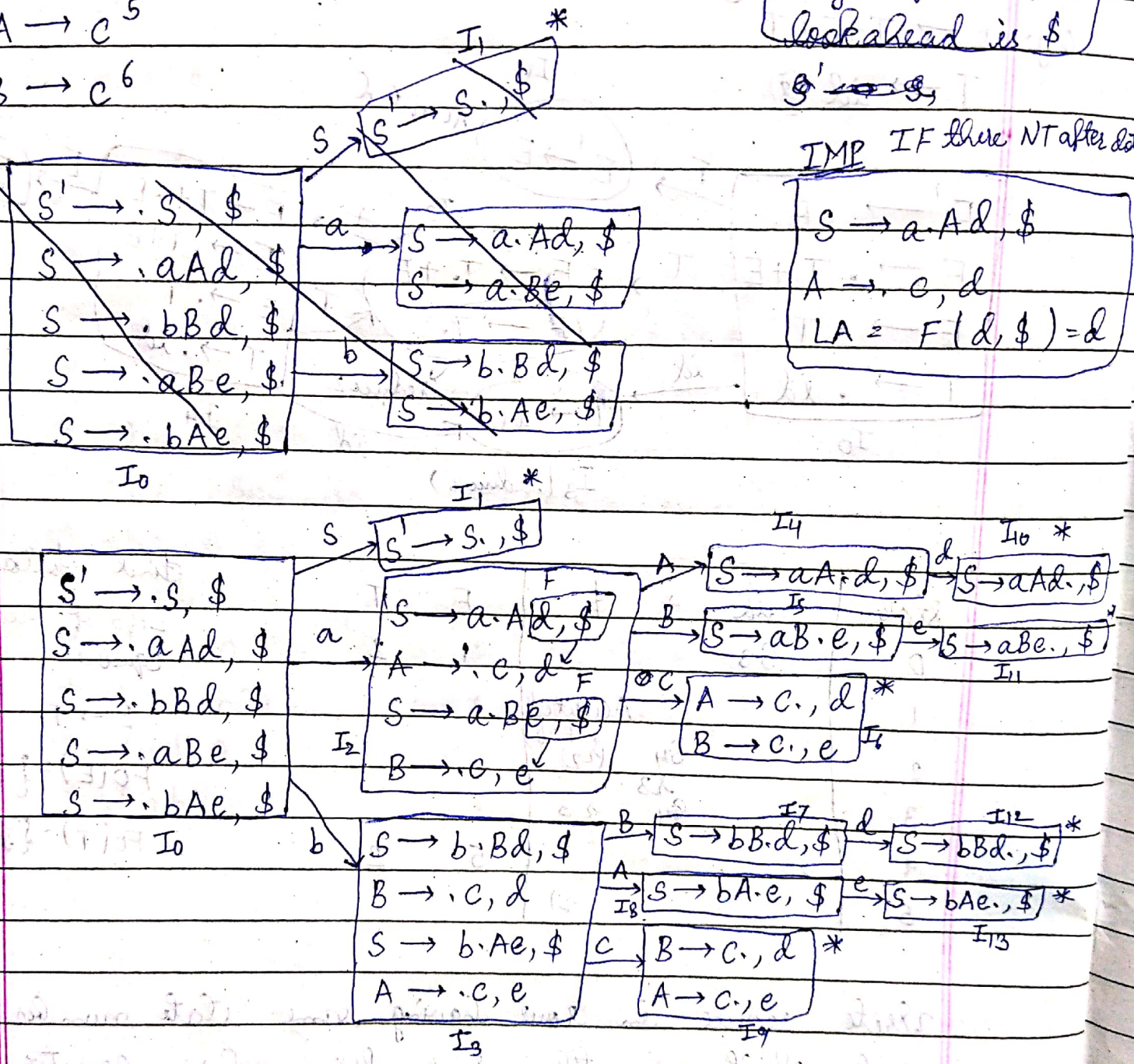
$B \xrightarrow{6} c$

IMP

By default, lookahead is $

$S' \rightarrow S$,

IMP IF there NT after dot



If state is accepted then write reduce in column of its lookahead.

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

* LALR (Bottom-up)

LALR is similar to CLR but in LALR similar states are merged.

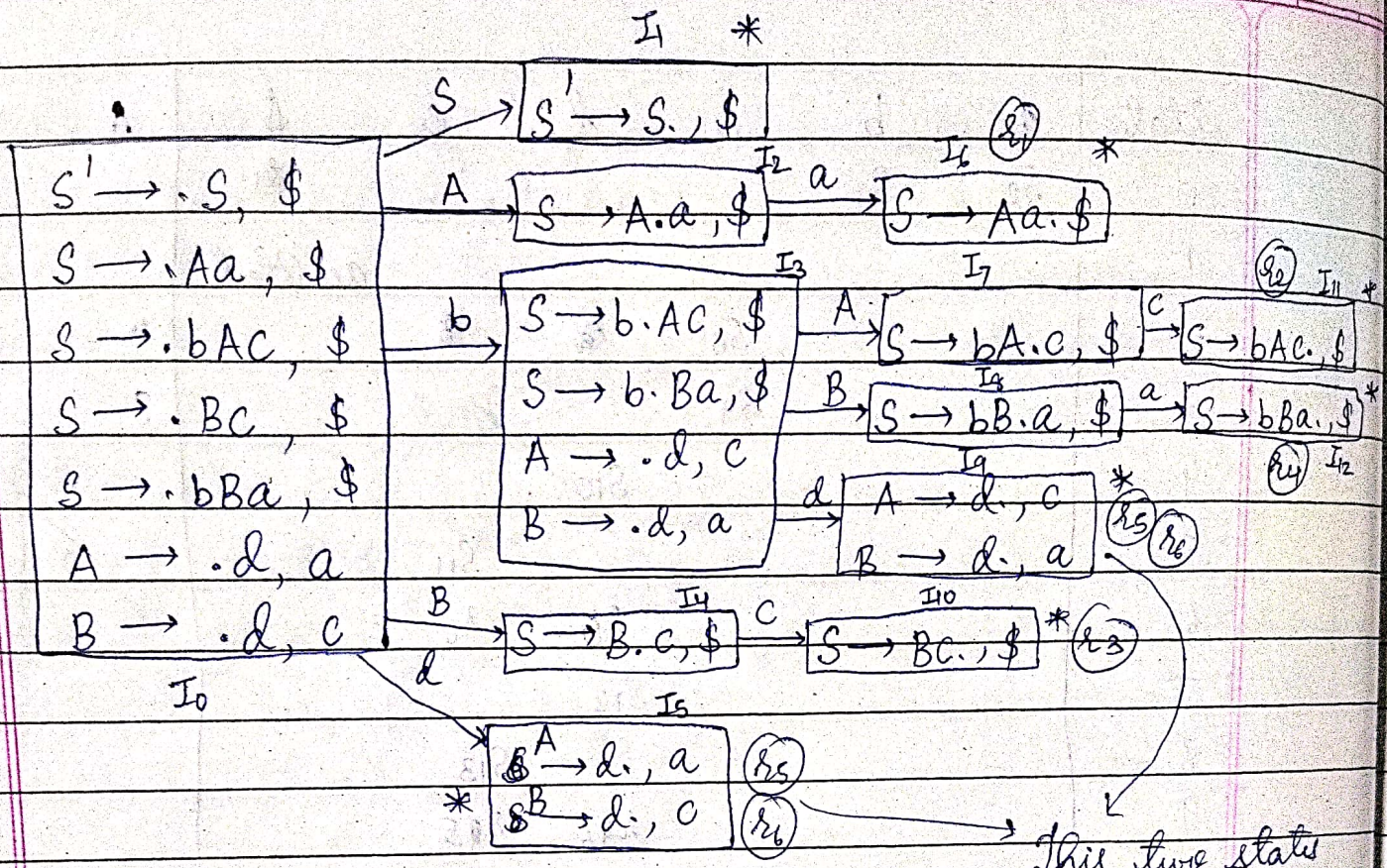
$S \rightarrow \overset{1}{A}a \mid \overset{2}{b}A\overset{3}{c} \mid \overset{4}{B}c \mid bBa$

$A \rightarrow d^5$

$B \rightarrow d^6$

In LALR, two reduces with different look aheads can be merged.

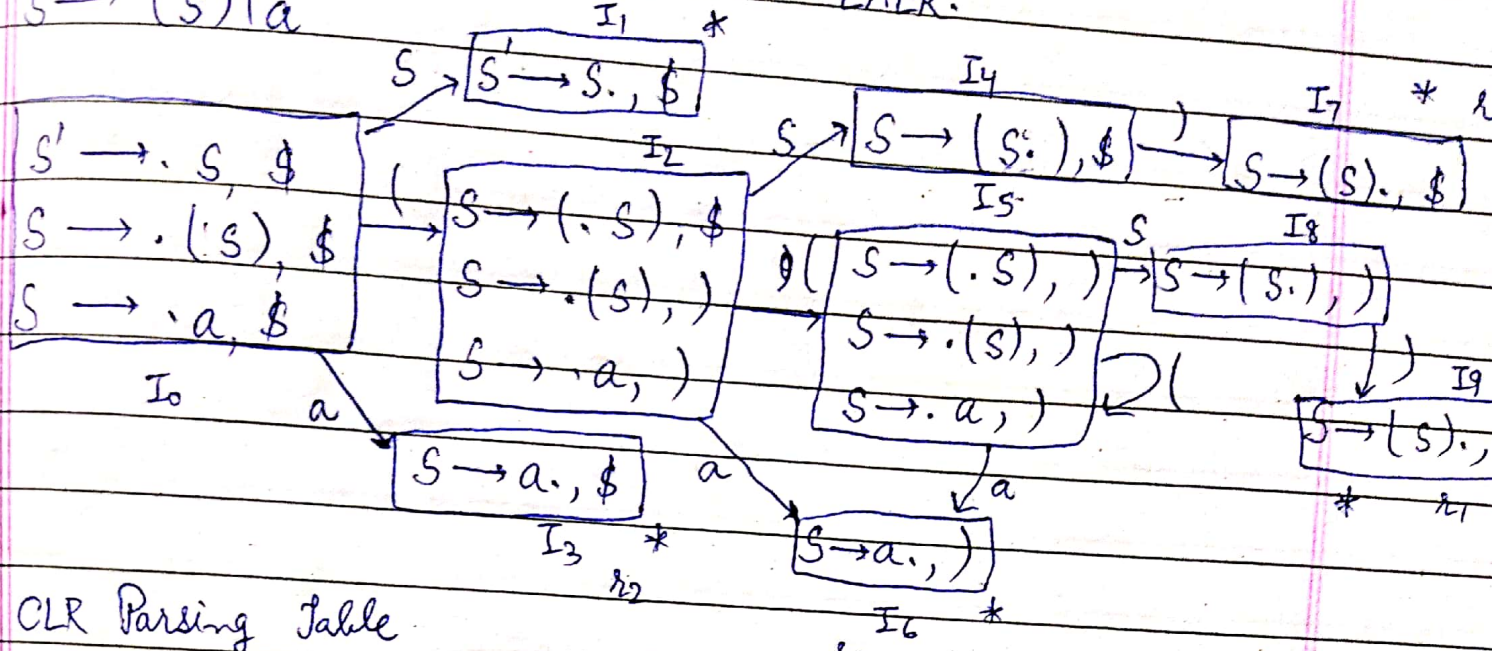
IMP Sometimes, a grammar can be CLR but is not LALR.



This two states cannot be merged as there would be multiple entries in the same column for LALR. So, the above grammar is not LALR but is CLR.

* Difference between CLR and LALR.

$S \rightarrow (S) | a$ (1) (2)



CLR Parsing Table

	(	)	a	$	S
0	S_2		S_3		1
1			accepted		
2	S_5		S_6		4
3				r_2	
4		S_8 7			
5	S_5		S_6		8
6		r_2			
7				r_1	
8		S_9			
9		r_1			

LALR Parsing Table

	(	)	a	$	S
0	S_2		S_3		1
1			accepted		
25	S_5		S_6		48
36		r_2		r_2	
48		S_{79}			
79		r_1		r_1	