LECTURE NOTES
On

Compiler Design(CD)

B. Tech,6th Semester, CSE

Prepared by:

Ms. Monali Patel

Asst. Professor

Computer Science & Engineering

Vikash Institute of Technology, Bargarh

(Approved by AICTE, New Delhi & Affiliated to BPUT, Odisha)
Barahaguda Canal Chowk, Bargarh, Odisha-768040
www.vitbargarh.ac.in

DISCLAIMER

- This document does not claim any originality and cannot be used as a substitute for prescribed textbooks.

- The information presented here is merely a collection by Monali Patel with the inputs of students for their respective teaching assignments as an additional tool for the teaching- learning process.

- Various sources as mentioned at the reference of the document as well as freely available materials from internet were consulted for preparing this document.

- Further, this document is not intended to be used for commercial purpose and the authors are not accountable for any issues, legal or otherwise, arising out of use of this document.

- The author makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaims any implied warranties of merchantability or fitness for a particular purpose.

******

COURSE CONTENT

B. Tech, 6th Semester CSE

Module I: (10 hours)

Introduction: Overview and Phases of compilation. Lexical Analysis: Non-Deterministic and Deterministic Finite Automata (NFA & DFA), Regular grammar, Regular expressions and Regular languages, Design of a Lexical Analyzer as a DFA, Lexical Analyzer generator. Syntax Analysis: Role of
a Parser, Context free grammars and Context free languages, Parse trees and derivations, Ambiguous grammar. Top Down Parsing: Recursive descent parsing, LL (1) grammars, Non-recursive
Predictive Parsing, Error reporting and Recovery. Bottom Up Parsing: Handle pruning and shift reduces Parsing, SLR parsers and construction or SLR parsing tables, LR(1) parsers and construction of LR(1) parsing tables, LALR parsers and construction of efficient LALR parsing tables, Parsing using
Ambiguous grammars, Error reporting and Recovery, Parser generator

Module II: (6 hours)

Intermediate Code Generation: DAG for expressions, Three address codes - Quadruples and Triples, Types and declarations, Translation of Expressions, Array references, Type checking and Conversions, Translation of Boolean expressions and control flow statements, Back Patching, Intermediate Code Generation for Procedures.

Module III: (10 hours)

Code Generation: Factors involved, Registers allocation, Simple code generation using STACK Allocation, Basic blocks and flow graphs, Simple code generation using flow graphs. CodeOptimization: Objective, Peephole Optimization, and Concepts of Elimination of local common sub-
expressions, Redundant and un-reachable codes, Basics of flow of control optimization.

Module IV: (10 hours)

Run Time Environment: Storage Organizations, Static and Dynamic Storage Allocations, STACK Allocation, Handlings of activation records for calling sequences. Syntax Directed Translation: Syntax Directed Definitions (SDD), Inherited and Synthesized Attributes, Dependency graphs, Evaluation orders for SDD, Semantic rules, Application of Syntax Directed Translation. Symbol Table: Structure and features of symbol tables, symbol attributes and scopes.

****

REFERENCES

B. Tech, 6th Semester CSE

Compiler Design(CD)

Books:

[1] Compilers – Principles, Techniques and Tools, A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, 2nd Ed., Pearson. 2007

[2] Advanced Compiler Design & Implementation, S. S. Muchnick, Morgan Kaufmann, 1997

[3] Modern Compiler Design, D. Galles, 1st Ed., Pearson Education,2004

Digital Learning Resources:

Course Name: Compiler Design

Course Link: https://onlinecourses.nptel.ac.in/noc21_cs07/preview

Course Instructor: Prof. Santanu Chattopadhyay,

****

Introduction to compilers & its phases

A compiler is a program takes a program written in a source language & translate it into an equivalent program in a target language.

→ The source language is a HLL & target language is machine language.

Necessity of compiler

→ Techniques used in a lexical analyzer, can be used in text editor, info. retrieval system & pattern recognition program

→ Techniques used in parser can be used in query processing (SQL)

→ Most of the techniques used in compiler design can be used in NLP.

Properties

a) correctness

i) correct o/p in execution

ii) it should report errors

iii) correctly report if the programmer is not following syntax

b) Efficiency

c) compile time & execution

d) Debugging / Usability

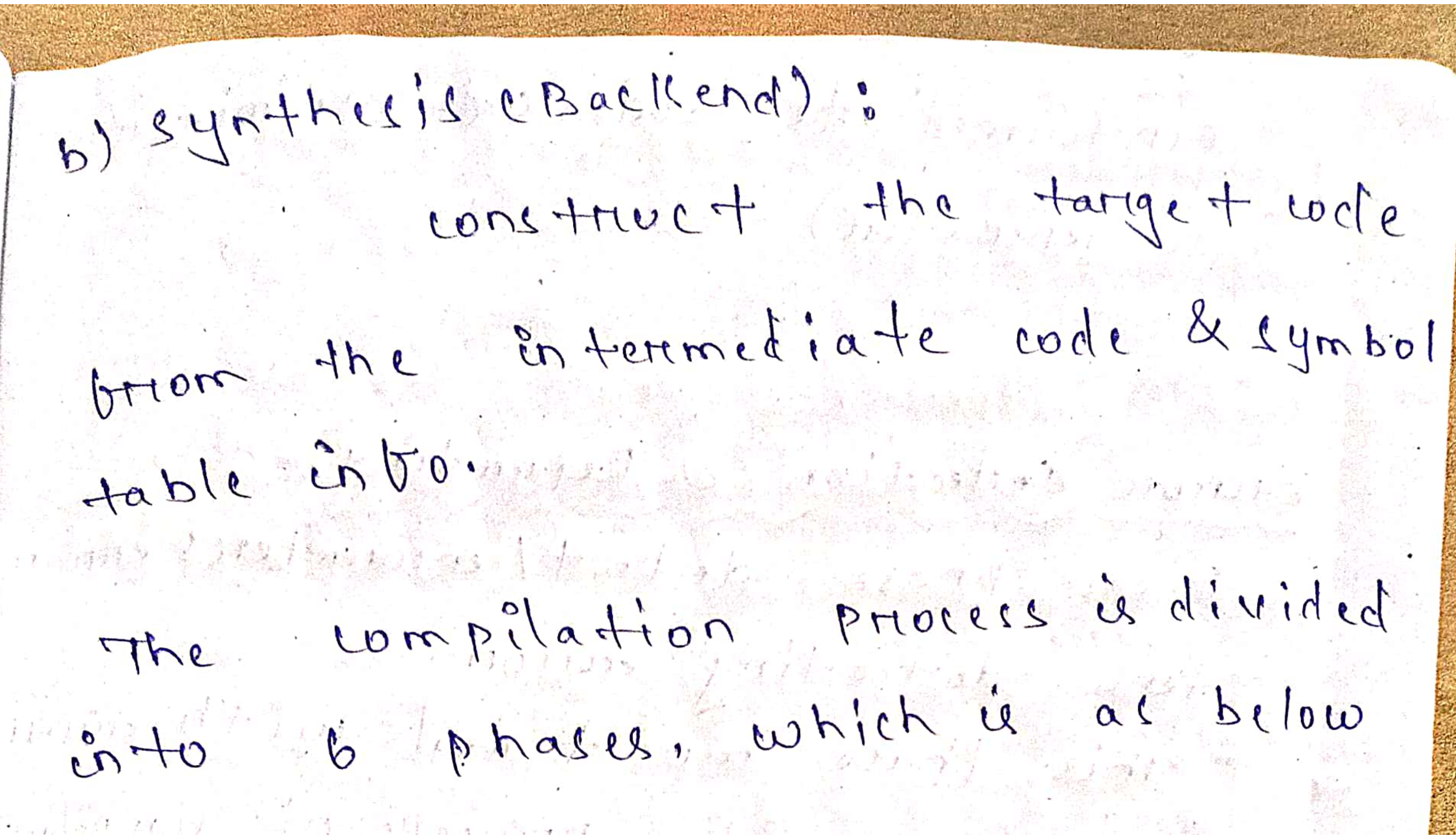
Compilere	Interepretere
→ It translates the whole program at a time.	→ It translate statement by statement
→ compiler is faster.	→ It is slowere
→ Debugging is not easy.	→ Debugging is easy
→ compiler are not portable.	→ Interepreter are portable.
→ eg c, c++, etc	→ JS, pyton, Ruby etc

→ The design procedure of compilere is basically divided into 2 parts

a) Analysis (Front end) :

splits the source program into pieces & puts grammatical structure on them

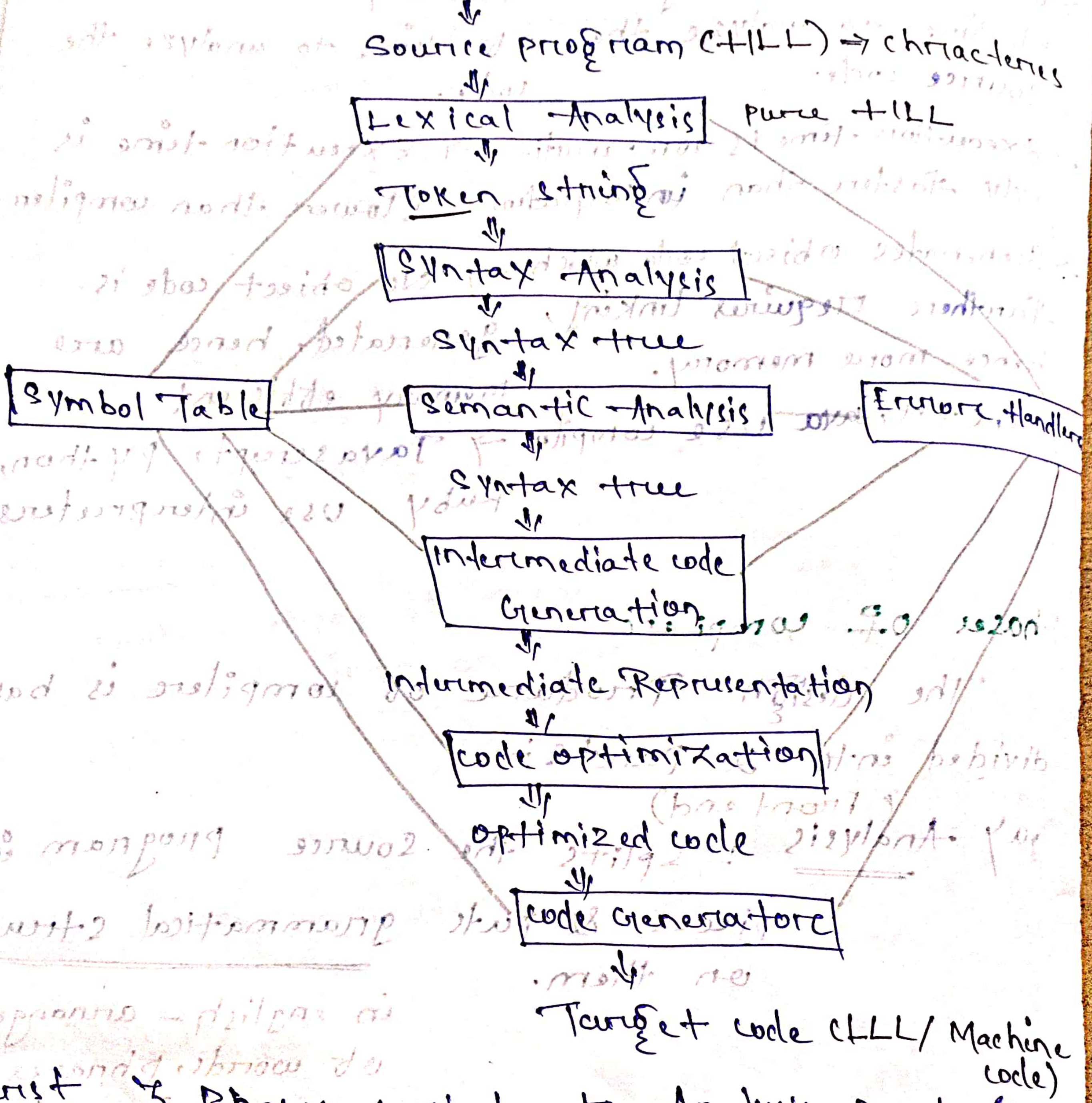
(Grammare $G = \{V, T, P, S\}$



b) synthesis (Backend) :

construct the target code from the intermediate code & symbol table info.

The compilation process is divided into 6 phases, which is as below



→ First 3 Phases consist of Analysis Part & last 3 ~~Part~~ phases are included in synthesis part.

1) **Lexical Analysis :⇒**

Lexical Analyzer Scan the source code & divide into tokens i.e input to source code & o/p Stream of token.

→ Tokens are represented as,

< tokenname, attribute value>

eg:⇒ Value = x + y * 10

The sequence of token string is Represented after g Lexical Analysis phase is -

<id 1> <=> <id 2> <+> <id 3> <*> <10>

⇓id ⇓opr ⇓id ⇓opr ⇓id ⇓opr ⇓constant.

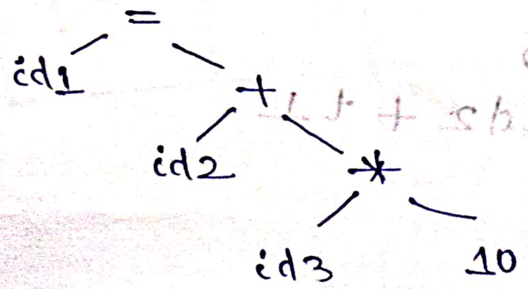
2) **Syntax Analysis :**

It verifies the grammatical mistake of the source code. To verify the syntax of the source code the lang. must be defined by C.F.G

→ syntax analyzer take the stream of ~~source~~ tokens as i/p & generates the Parse tree.

→ The parse tree/syntax tree is a tree in which each interior node represents an operator & the children node represents operands

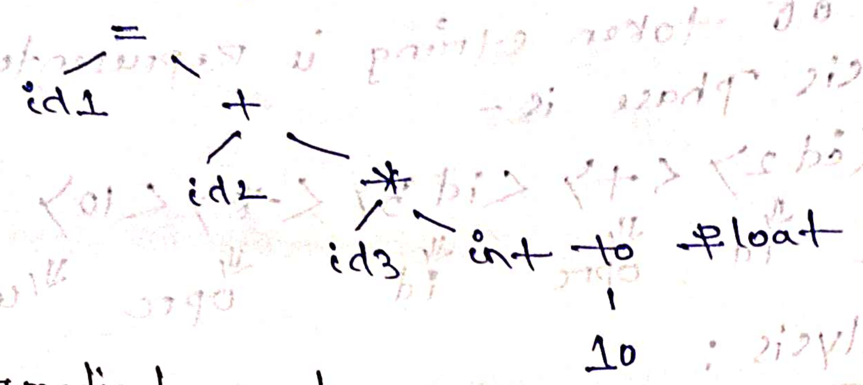
~~xx~~ <id1> <=> <id2> <+> <id3> <*> <10>



3) Semantic Analysis

→ The semantic analyzer verify the meaning of each & every sentence by performing type check.

eg ⇒ If an integer number is operated upon a floating point no. then it will convert integer to floating point.



4) Intermediate code Generation.

The source code is converted into intermediate code to make code generation process simple & easy.

→ This phases produces ~~3 co~~ 3 address code.

eg :- t1 = int to float (10)

t2 = id3 * t1

t3 = id2 + t2

5) code optimization

It optimize the intermediate code.

t1 = id3 * 10.0

id1 = ~~id2 *~~ id2 + t1

ⓐ

b) Code Generation

It takes input the intermediate code & generates target code.

eg :-

```
LOAD    R2, id3
MUL     R2, #10.0
LOAD    R1, id2
ADD     R1, R2
STORE   id1, R1
```

~~Lexical Analysis~~ :-

Symbol Table

→ It is a data structure which contains record for each identifier & fields for the attributes of the identifier.

→ This DS helps ~~to find~~ the info. for each identifiers & tokens present in the program.

→ Each phase of the compiler refers to symbol table to get info. about the identifiers & tokens.

id1	value
id2	value
id3	value
⋮	

Q :- ⇒ ~~Sum = old sum + Rate *~~

a = b + c * 50

↓

Lexical Analyzer

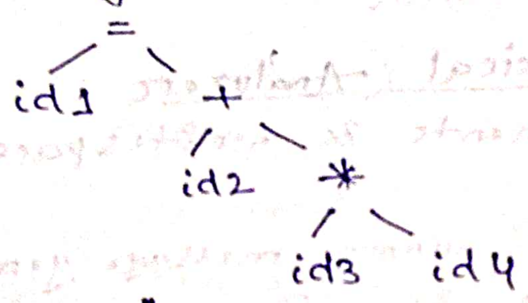
⇓

id1 = id2 + id3 * 50 (id4)

⇓

Syntax Analyzer

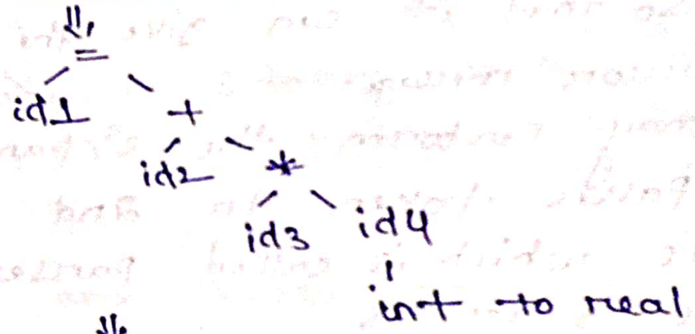
⇓



⇓

Semantic Analyzer

⇓



⇓

Intermediate code Generator

⇓

t1 = int to real
t2 = id3 * t1
t3 = id2 + t2
~~t3 = id~~ id1 = t3

⇓

Code optimization

⇓

t1 = id3 * 50.0
id1 = id2 + t1

⇓

Code Generation

⇓

move id3, R2
mul #50·6, R2
move id2, R2
add R2,R1
move R1,id1

Errorr detection & Reporting

Each phase detect/encounters errorr after detecting errorr.

→ This Phase must deal with errorr to continue with process of compila

→ The following are some errorr encountered in each phas.

i) Lexical Analyzer : Miss spell token

ii) Semantic " : Type mismatch

iii) Syntax Analyzer : Missing Parenthesis, less no. of operands.

iv) Intermediate code generation : In compatib operands for an operand.

v) code optimization : unreachable statement

vi) code Generation : Memory restriction to store a variable.

Deterministic Finite Automata (DFA)

→ DFA refers to uniqueness of the computation.

→ The FA are called deterministic if the machine is read an input string 1 symbol at a time.

→ In DFA there is only one path for specific input from the current state to next state.

→ DFA doesn't allow the null move i.e DFA can not change state without any i/p character.

Formal definition of DFA

A DFA is a collection of 5 tuples

Q: finite set of state

Σ: finite set of input alphabet.

q_0: initial state

F: final state.

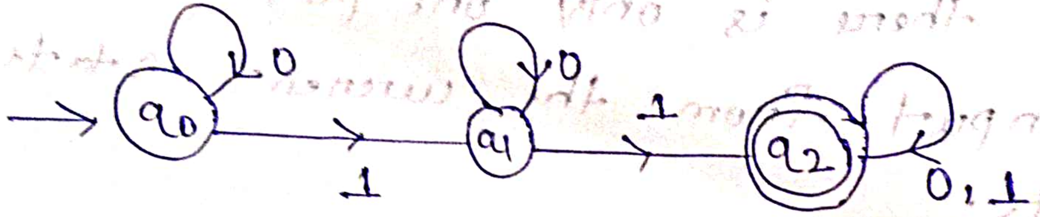
δ: Transition function

Graphical Representation of DFA

A DFA can be represented by graphs called state diagram

1. The state is represented by vertices
2. The arc labeled with an i/p character show transitions
3. The initial state is marked with an arrow
4. The final state is denoted by double circle

eg



$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

$q_0 = \{q_0\}$

$F = \{q_2\}$

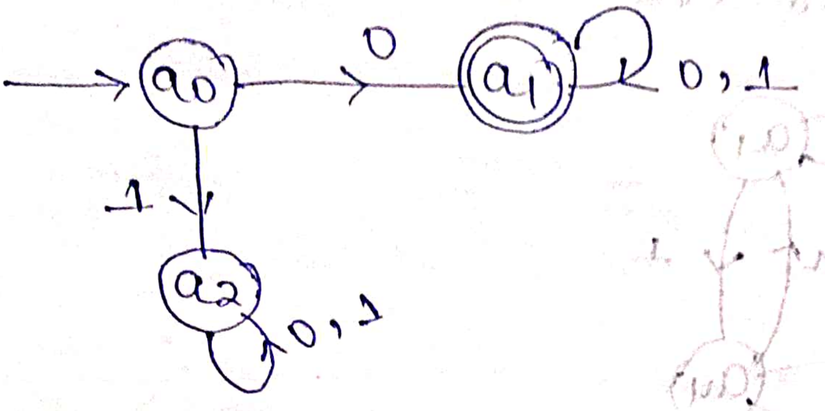
$\delta(q_0, 0) = q_0$ $\delta(q_1, 0) = q_1$ $\delta(q_2, 0) = q_2$

$\delta(q_0, 1) = q_1$ $\delta(q_1, 1) = q_2$ $\delta(q_2, 1) = q_2$

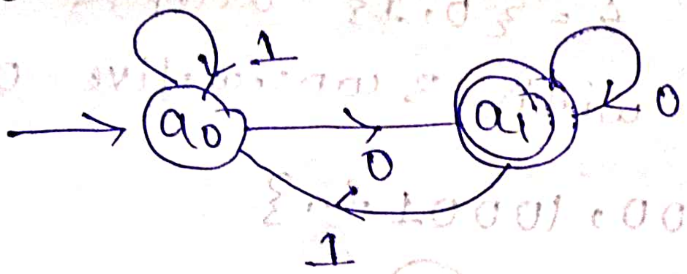
Transition table

P.S	0	1
→ q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_2

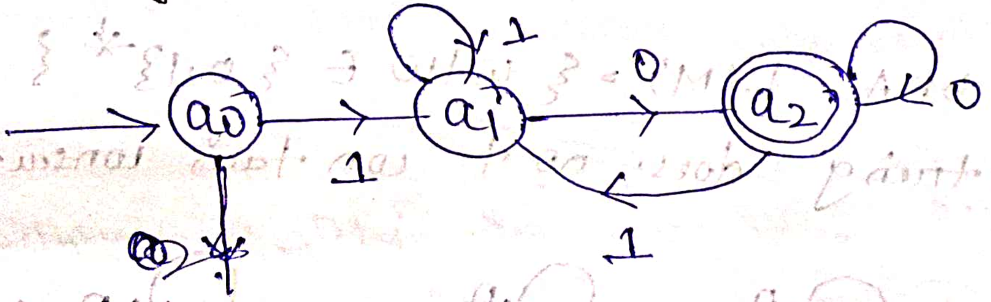
Q: DFA with $\Sigma = \{0, 1\}$ accepts all starting with 0



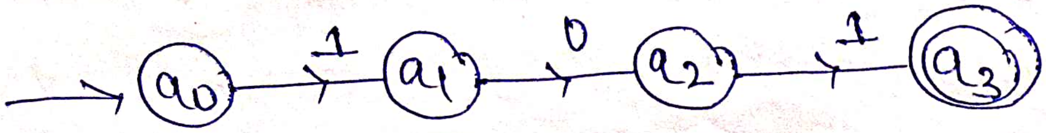
Q DFA with $\Sigma = \{0, 1\}$ accepts all ending with 0



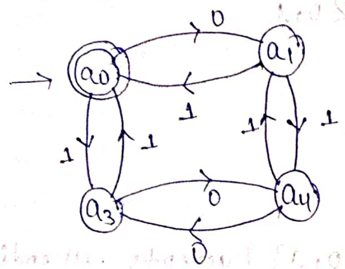
Q: Design a FA with $\Sigma = \{0, 1\}$ accepts those string which starts with 1 and ends with 0



Q: Design a FA with $\Sigma = \{0, 1\}$ accepts the only i/p 101

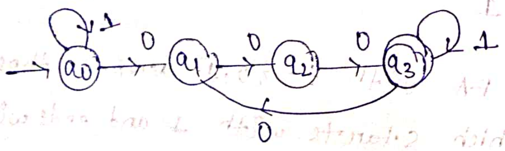


Q. Design FA with Σ = {0,1} accepts even no. of 0's and even no' of 1's

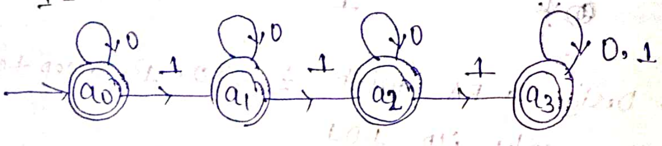


Q. Design FA with Σ = {0,1} accepts the set of all strings with 3 consecutive 0's

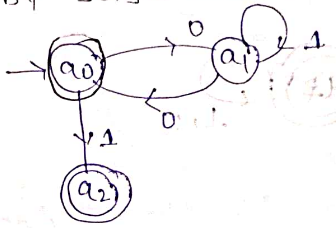
L = {000, 0001, 1000, 10001 ...}



Q. Design a DFA L(M) = {w | w ∈ {0,1}* } & w is a string does not contain consecutive 1's

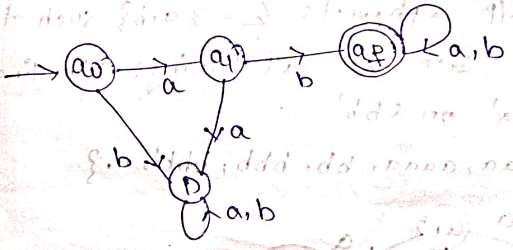


Q: Design a FA with Σ = {0,1} accepts the strings with an even no. of 0's followed by single 1

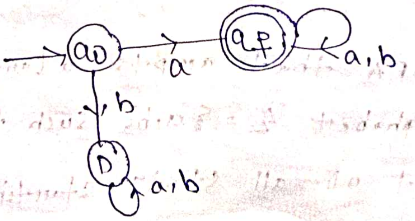


Q. Draw a DFA for the language accepting strings starting with "ab" over Σ = {a,b}

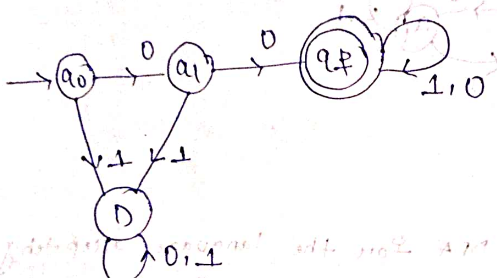
L = {ab, aba, abab ...}



Q. Draw a DFA for language accepting strings starting with 'a' over input {a,b}

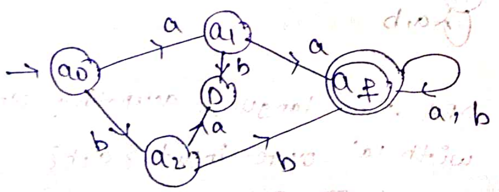


Q Draw a DFA that accepts a language L over i/p alphabet {0,1} such that L is the set of all string, starting with '00'

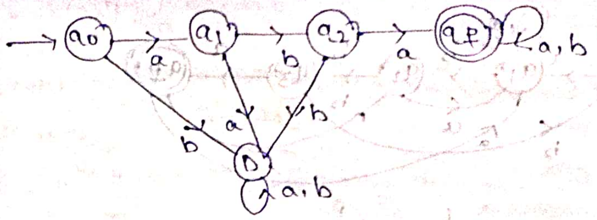


Q construct a DFA that accepts a langu L over i/p alphabets Σ = {a,b} such tha L is the set of all strings starting with 'aa' or 'bb'

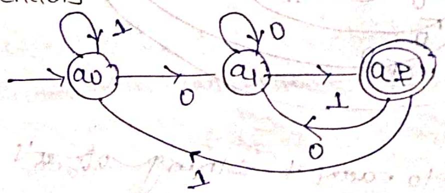
L = {aa, aaa, aaaa, bb, bbb, bbbb ...}



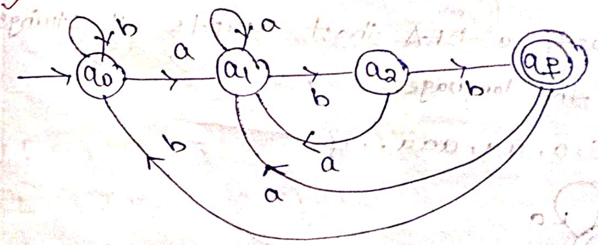
Q construct a DFA that accepts a Language L over i/p alphabet Σ = {a,b} such that L is the set of all string starting with 'aba'



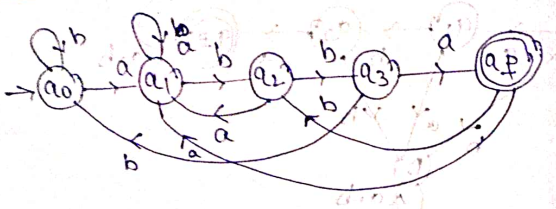
Q ending with '01' over Σ = {0,1}



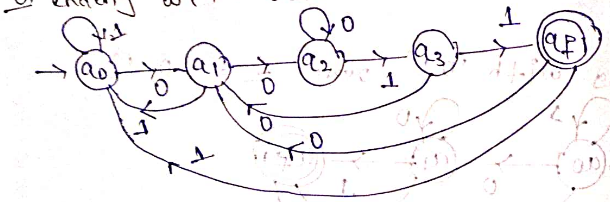
Q ending with 'abb'



Q Draw a DFA for the language accepting strings ending with 'abba' over i/p alphabet Σ = {a,b}



Q ending with '0011'

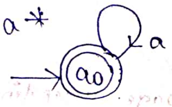


Q ~~obtain DFA to accept string of a's & b's having exactly one a~~

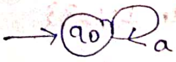
Q 1) construct a DFA that accepts all strings from the language

L = { ε, a, aa, aaa . . . }

a*



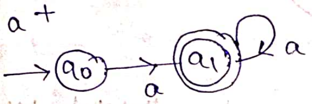
2) construct a DFA that accepts all strings from the language L = { }



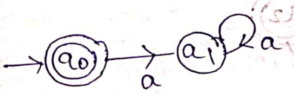
3) construct a DFA that accepts all strings from the language

L = { a, aa, aaa . . . }

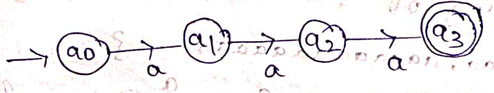
a +



4) construct a DFA that accepts all strings from the language L = { ε }

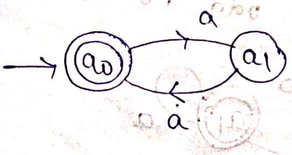


5) L = { aaa }

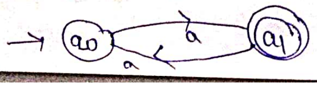


6) L = { string with even size }

L = { ε, aa, aaaa, aaaaaa . . }

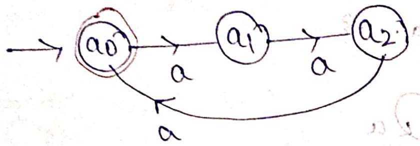


7) L = { string with odd size }



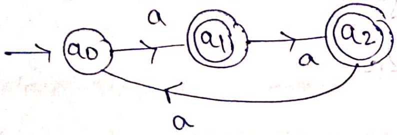
8) L = {strings of size divisible by 3}

L = {ε, aaa, aaaaaa, aaaaaaaaa . . }



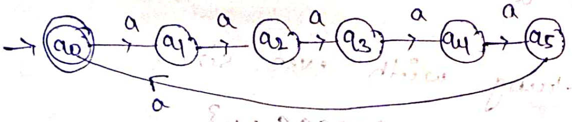
9) L = {string of size not divisible by 3}

L = {a, aa, aaaa, aaaaa . . . }

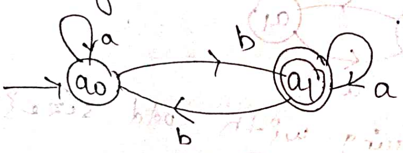


10) L = {string of size divisible by 6}

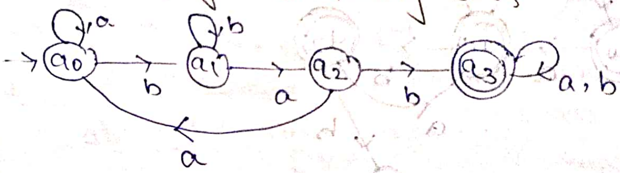
L = {ε, aaaaaa, aaaaaaaaaaaa, . . . }



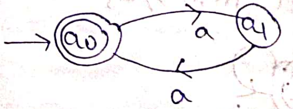
11) L = {string with odd no. of b's}



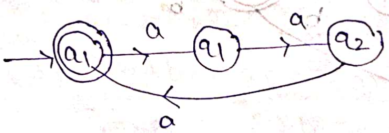
12) L = {strings containing bab}



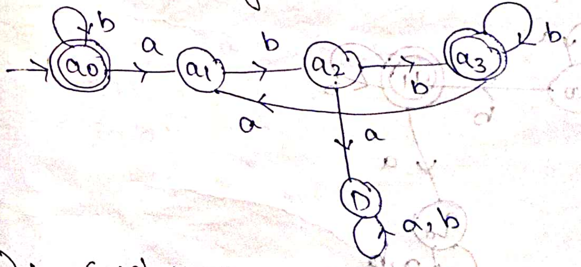
13) L = {string with size multiple of 2 or 3}



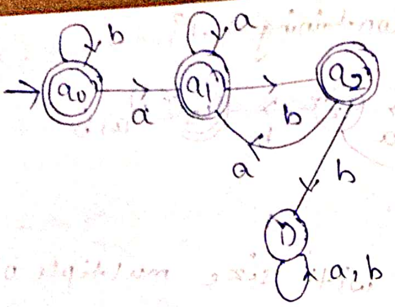
(multiple of 2)



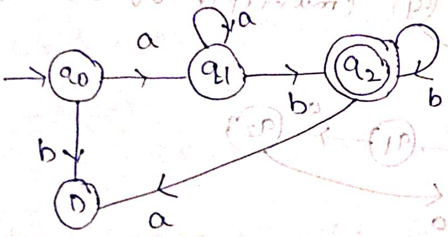
14) L = {w| every a in w is followed by bb}



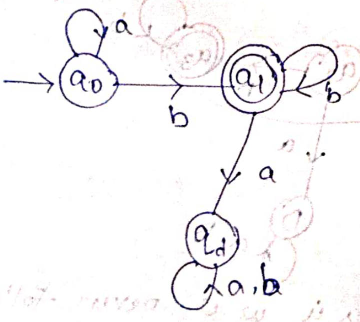
15) L = {w| every a is w is never followed by bb}



16) $L = \{ w | w = a^n b^m \text{ for, } m,n \geq 1 \}$



17) $L = \{ w | w = a^n b^m \text{ for } m,n \geq 0 \}$



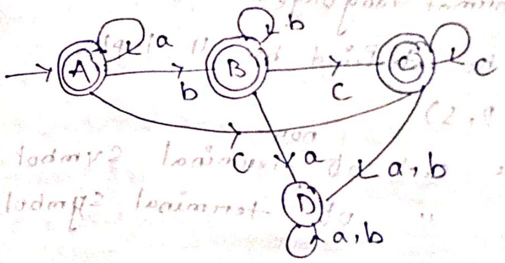
18) $L = \{ w | w = a^m b^n c^p \text{ for } m,n,p \geq 0 \}$

~~L = { abc, aabc, abbc, abcc ... }~~

$L = \{ \epsilon, a, aa, aaa, \ldots, b, bb, bbb \ldots, c, cc, ccc \ldots,$

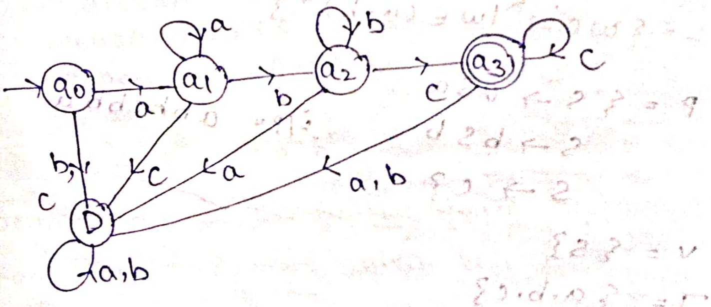
$abc, aabc, abbc, abcc \ldots \}$

~~a¢b~~



19) $L = \{ w | w = a^m b^n c^p \text{ for } m,n,p \geq 1 \}$

$L = \{ abc, aabc, abbc, abcc, a~~b~~cb \ldots \}$



Minimization of DFA

1)
- Equivalence thm
- My-hill Nerode thm

1)

	a	b
→ a_0	a_1	a_2
a_1	a_1	a_3
a_2	a_1	a_2
a_3	a_1	* a_4
* a_4	a_1	a_2

a) Equivalence thm

$\pi(Q_1, Q_2) = \{a_0, a_1, a_2, a_3\}\{a_4\}$

	a	b
a_0	a_1	a_2
a_1	a_1	a_3

$a_0 \equiv a_1$

	a	b
a_0	a_1	a_2
a_2	a_1	a_2

$a_0 \equiv a_2$

	a	b
a_0	a_1	a_2
a_3	a_1	* a_4

$a_0 \not\equiv a_3$

$\pi(Q_1, Q_2) = \{a_0, a_1, a_2\}, \{a_3\}, \{a_4\}$

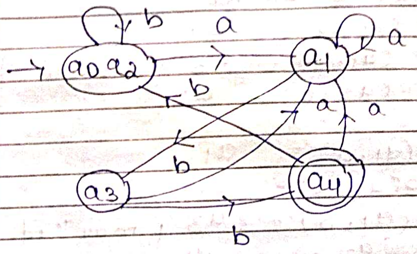
	a	b
a_0	a_1	a_2
a_1	a_1	a_3

$a_0 \not\equiv q_1$

	a	b
a_0	q_1	a_2
a_2	a_1	a_2

$a_0 \equiv a_2$

$\pi(Q_1, Q_2) = \{a_0\, a_2\}, \{a_1\}, \{a_3\}, \{a_4\}$



b) My Hill Nerode thm

a_0	a_0			
a_1	✓	a_1		
a_2		✓	a_2	
a_3	✓	✓	✓	a_3
a_4	✓	✓	✓	✓

$(a_0, a_1) = (a_0, a) = a_1$
✓
$(a_1, a) = a_1$
$(a_0, b) = a_2$ | marked
$(a_1, b) = a_3$

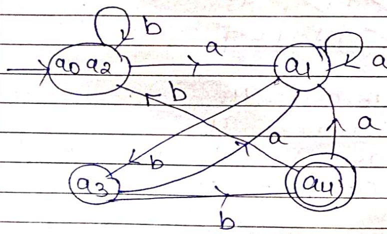
$(a_0, a_2) = (a_0, a) = a_1$
$(a_2, a) = a_1$
$(a_0, b) = a_2$
$(a_2, b) = a_2$

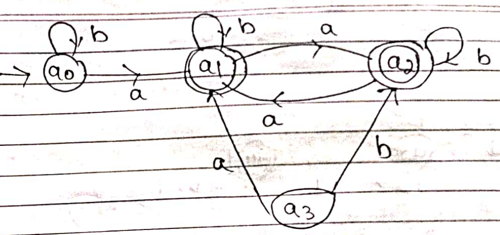
$(a_0, a_3) = (a_0, a) = a_1$
$(a_3, a) = a_1$
$(a_0, b) = a_2$ | marked
$(a_3, b) = a_4$ |
✓

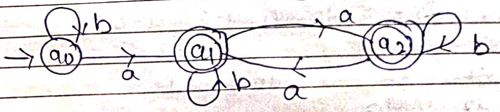
$(a_1, a_2) = (a_1, a) = a_1$
$(a_2, a) = a_1$
$(a_1, b) = a_3$ | marked
$(a_2, b) = a_2$
✓

$(a_1, a_3) = (a_1, a) = a_1$
$(a_3, a) = a_1$
$(a_1, b) = a_3$ | marked
$(a_3, b) = a_4$ |
✓

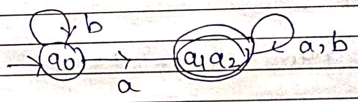
$(a_2, a_3) = (a_2, a) = a_1$
$(a_3, a) = a_1$
$(a_2, b) = a_2$ | marked
$(a_3, b) = a_4$ |
✓

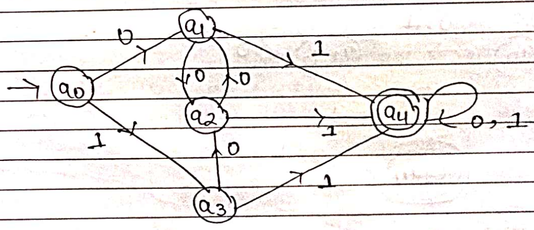


2)




a) Equivalence th^m
$\pi(Q_1, Q_2) = \{a_0\}, \{a_1, a_2\}$

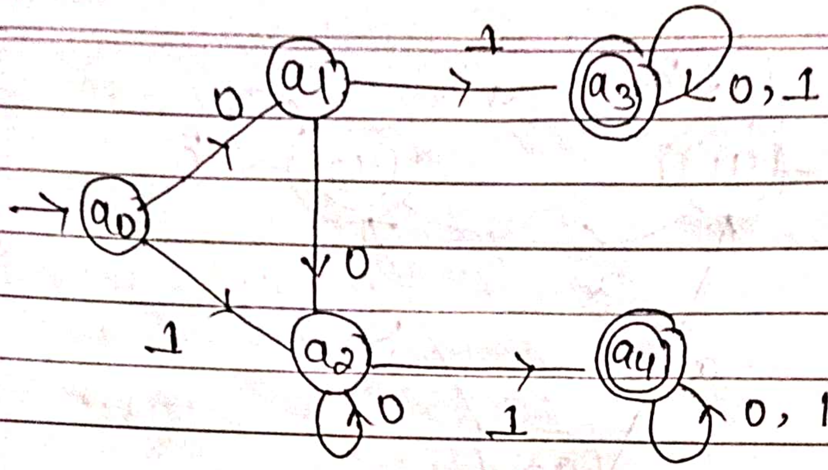


3)


~~$\pi(Q_1, Q_2) = \{a_0, a_1, a_2, a_3\}\{a_4\}$~~

→ Table filling Method / My hill Nerode th^m

4)

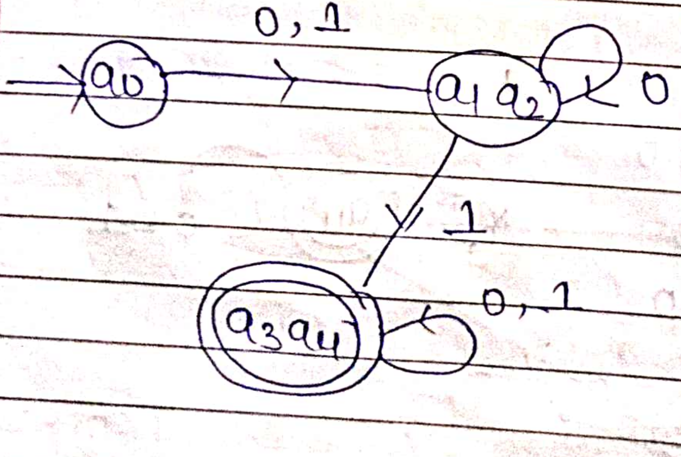


$\pi(Q_1, Q_2) = \{a_0, a_1, a_2\} \{a_3 a_4\}$

	0	1	
a_0	a_1	a_2	$a_0 \not\equiv a_1$
a_1	a_2	a_3	

	a	b	
a_0	a_1	a_2	$a_0 \not\equiv a_2$
a_2	a_2	a_4	

$\pi(Q_1, Q_2) = \{a_0\} \{a_1 a_2\} \{a_3 a_4\}$



Non- Deteremini-stic Finite Automata (NFA

NFA is defined by quin-tuple
$(Q, \Sigma, \delta, q_0, F)$

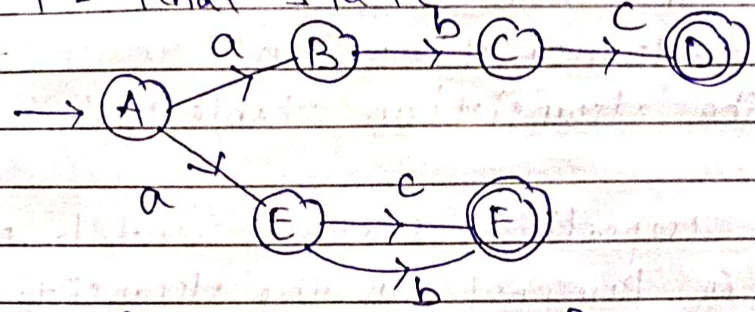
Q = Finite set of state

Σ = input alphabet

δ = transition function

q_0 = initial state

F = Final state



$Q = \{A, B, C, D, E, F\}$

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

$\delta = \delta(A, a) = B, E \quad \delta(B, b) = C$

$\delta(C, c) = D$

$\delta(E, c) = F$

$\delta(E, b) = F$

$q_0 = A$

$F = \{D, F\}$

Converting NFA to DFA

Step-1

→ Let Q' be a new set of state of the DFA. Q' is null in starting

→ Let T' be a new transition ~~state~~ table

Step-2

→ Add start state of the NFA to Q'

→ Add transitions of the start state to the transition table T'

→ If start state makes transition to multiple states for some i/p alphabet.

Step-3

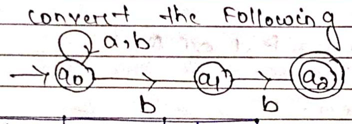
If any new state is present in the transition table T'

- Add the new state in Q'
- Add transitions of that states in the transition table T'

Step-4

Keep repeating step-3 until no new state is present in the transition table T'

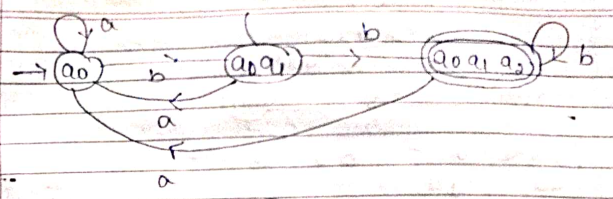
Q.1) convert the Following NFA to DFA



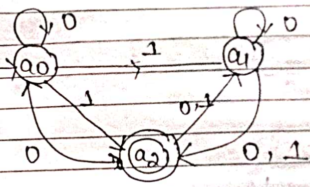
state	a	b
$\rightarrow a_0$	a_0	$\{a_0, a_1\}$
a_1	—	$*a_2$
$* a_2$	—	—

state	a	b
$\rightarrow a_0$	a_0	$\{a_0 a_1\}$
$\{a_0, a_1\}$	a_0	$\{a_0 a_1 a_2\}$

	a	b
a_0	a_0	$\{a_0 a_1\}$
$a_0 a_1$	a_0	$\{a_0 a_1 a_2\}$
$* a_0 a_1 a_2$	a_0	$\{a_0 a_1 a_2\}$



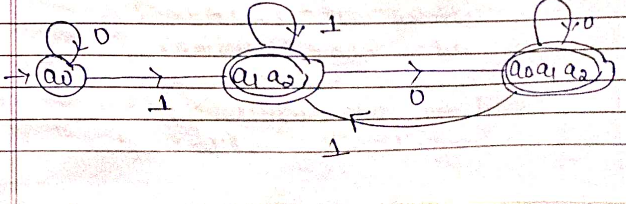
2)



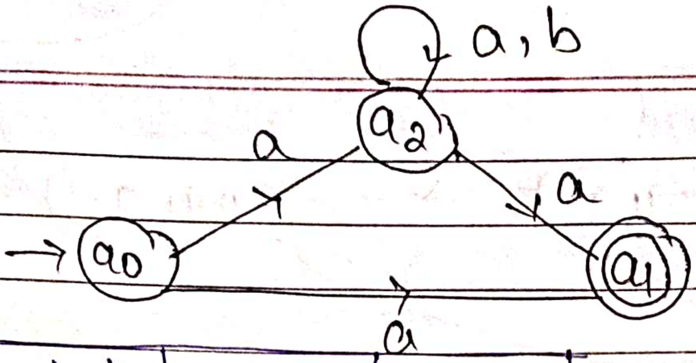
state	0	1
a_0	a_0	$a_1 a_2$
a_1	$a_1 a_2$	$*a_2$
$*a_2$	$a_0 a_1$	a_1

state	0	1
a_0	a_0	$a_1 a_2$
$a_1 a_2$	$a_0 a_1 a_2$	$a_1 a_2$

state	0	1
a_0	a_0	$a_1 a_2$
$a_1 a_2$	$a_0 a_1 a_2$	$a_1 a_2$
$a_0 a_1 a_2$	$a_0 a_1 a_2$	$a_1 a_2$



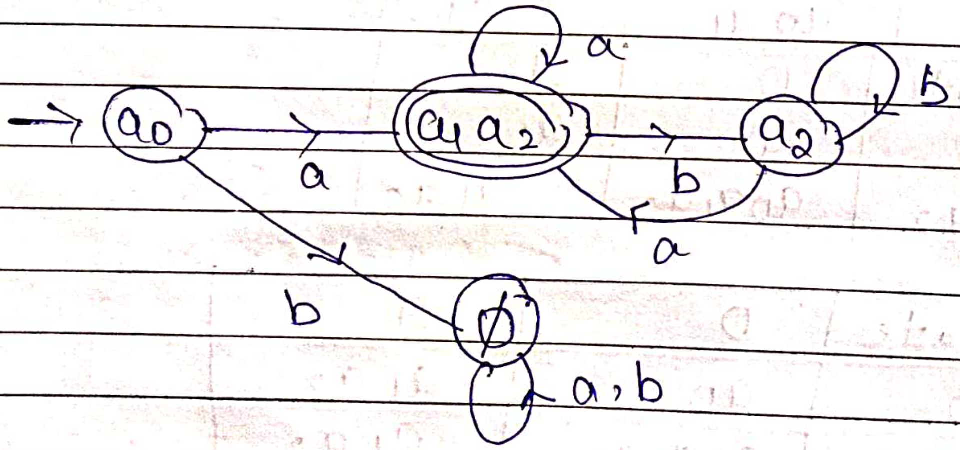
3)



State	a	b
a0	a1 a2	—
a1	—	—
a2	a1 a2	a2

State	a	b
a0	a1 a2	—
a1 a2	a1 a2	a2

	a	b
a0	a1 a2	—
a1 a2	a1 a2	a2
a2	a1 a2	a2



Regulare Grammare

→ It is type 3 grammare recognized using finite automata formal definition

→ Regulare Grammare generates regulare language.

→ They have a single non-terminal on LHS & a RHS consisting of a single terminal followed by a non-terminal.

(or)

→ The LHS must contain a non-terminal, & RHS must contain atmost 1 non terminal

eg $A \rightarrow aB$
$A \rightarrow a$
$A \rightarrow Ba$

Types of Regulare Grammare

a) Left Lineare Grammare

In LLG, the prod^n are in the form of $A \rightarrow Ba$ or $A \rightarrow a$.

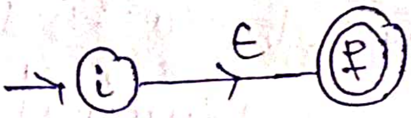
b) Right Lineare Grammare

In RLG, the prod^n are in the form of $A \rightarrow aB$ or $A \rightarrow a$

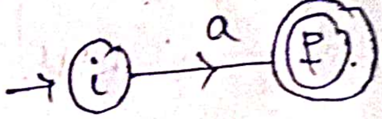
Converting RE to E-NFA (Thomson construction)

→ This gurantees that the resultin NFA will have exactly one final s. and one start(initial state).

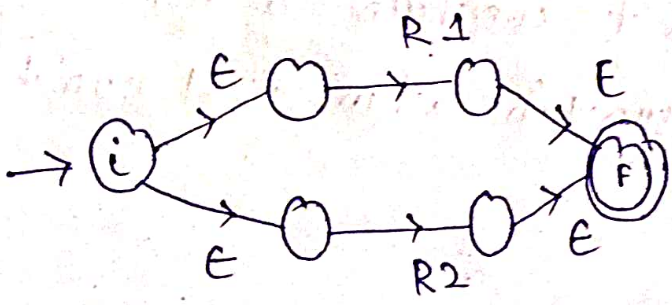
1. ε



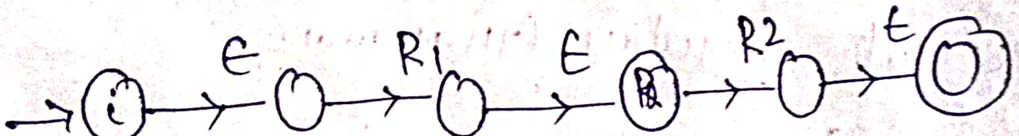
2. a



3. R1 / R2

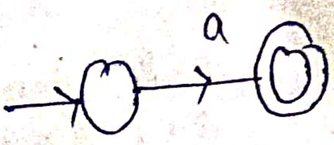


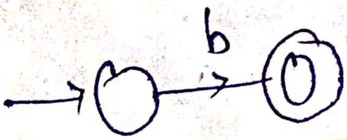
4. R1 · R2



Q (a/b)* a

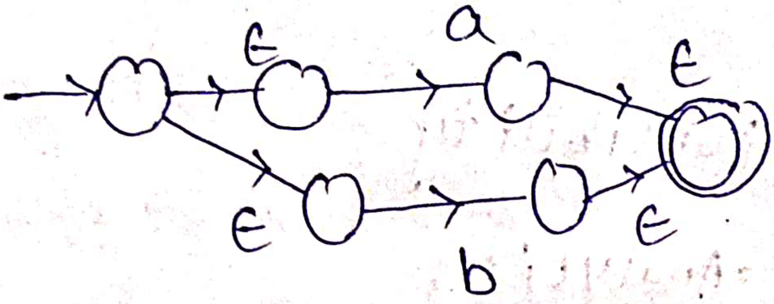
Step - 1





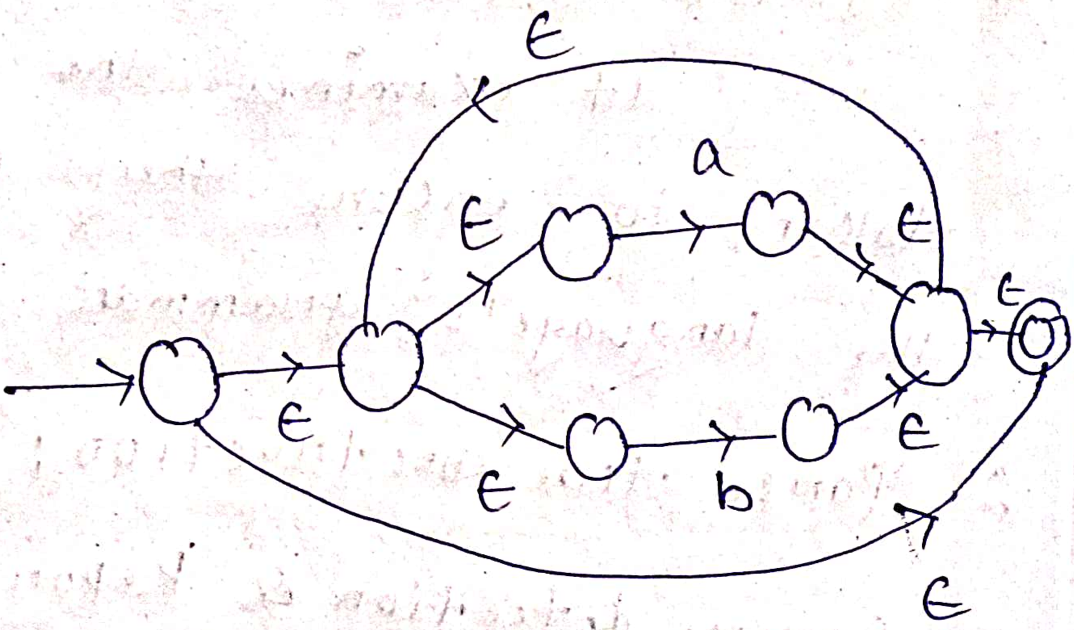
step-2

(a/b)



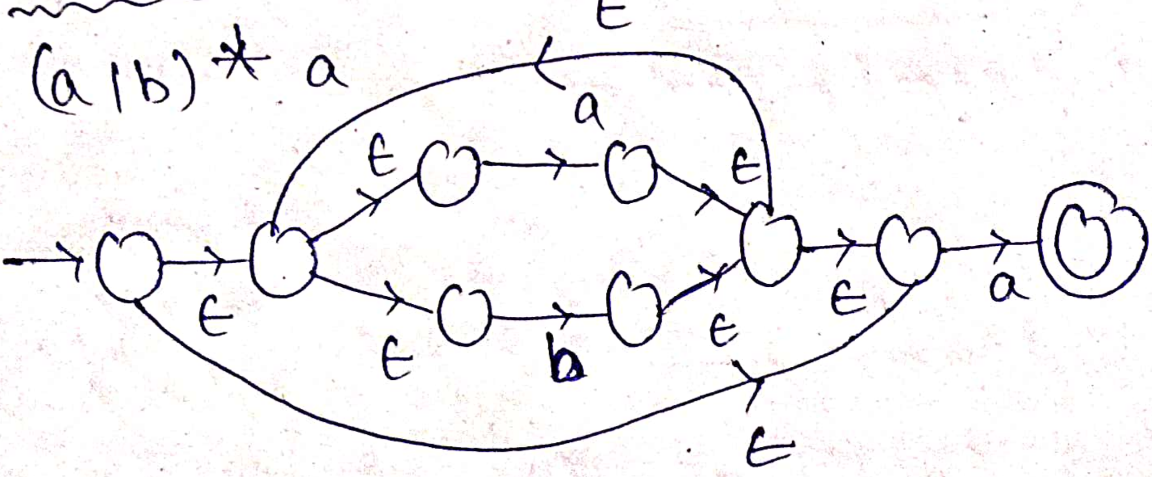
step-3

(a|b)*



step-4

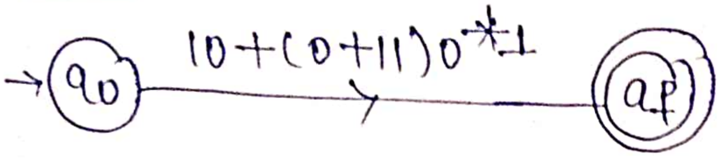
(a|b)* a



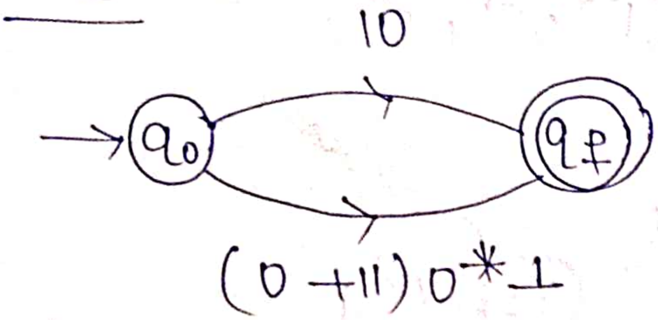
Finite Automata to Regular Expression

Q Design a FA from given RE 10+(0+11)0*1

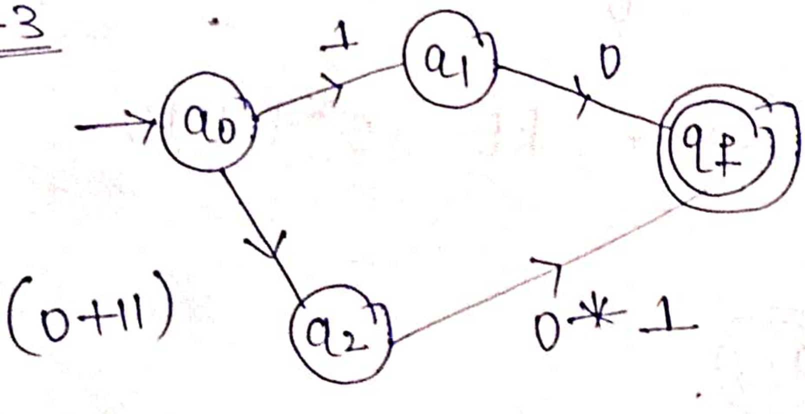
S-1



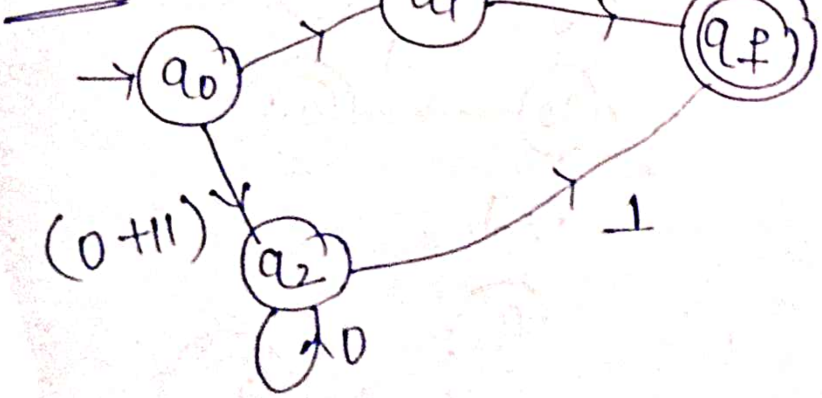
S-2



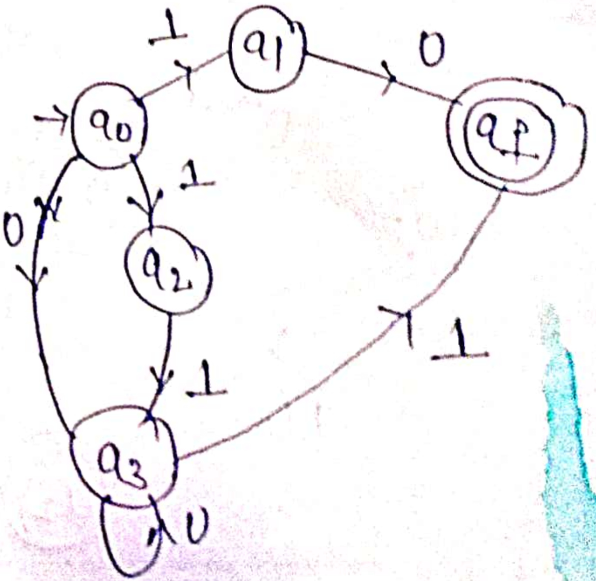
S-3



S-4

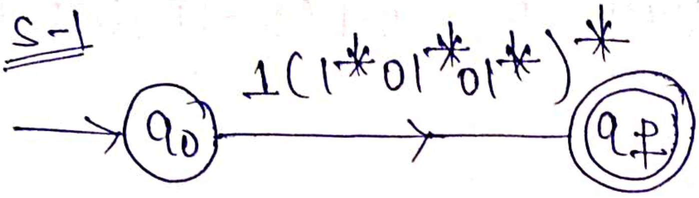


S-5

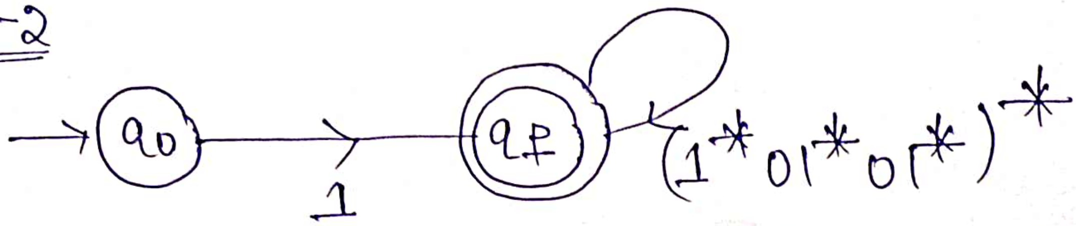


Q Design a NFA from RE $1(1^*01^*01^*)^*$

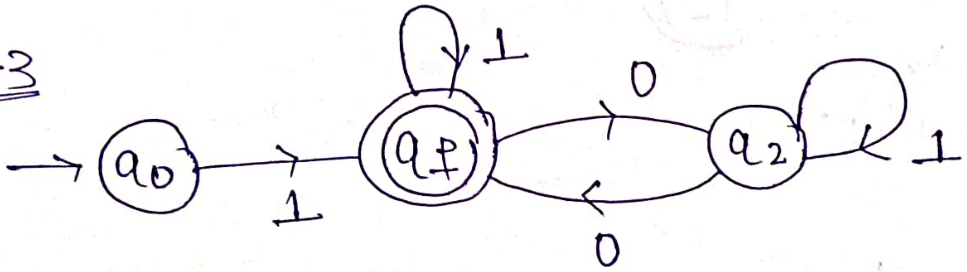
S-1



S-2

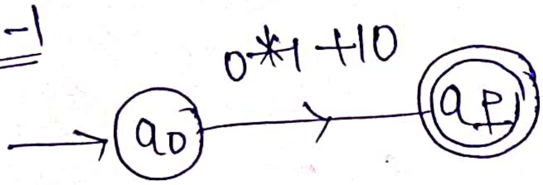


S-3

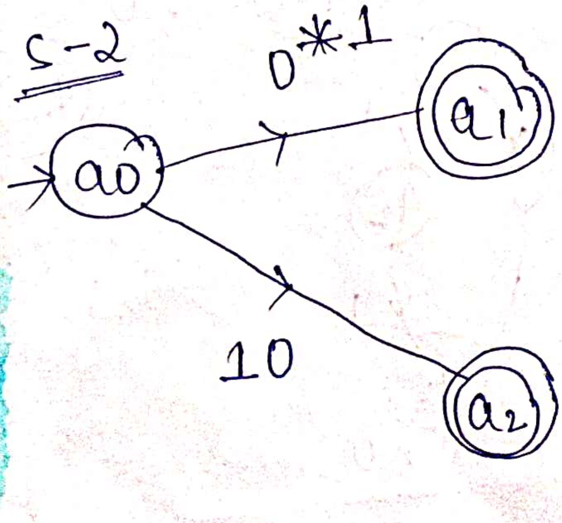


Q construct FA for RE 0^*1+10

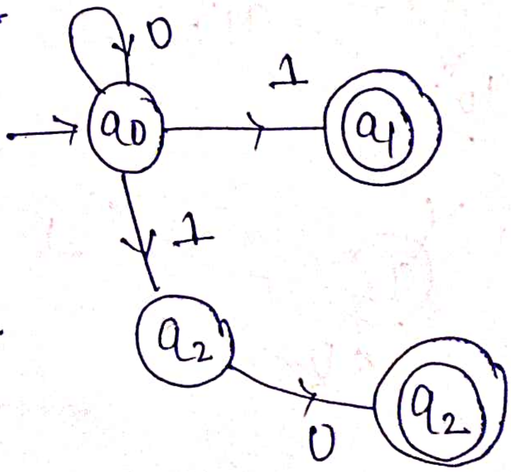
S-1



S-2



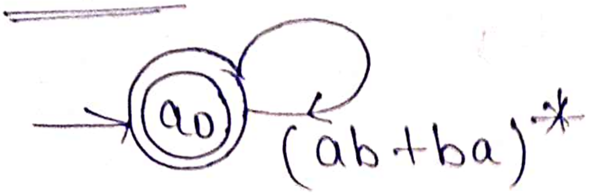
S-3



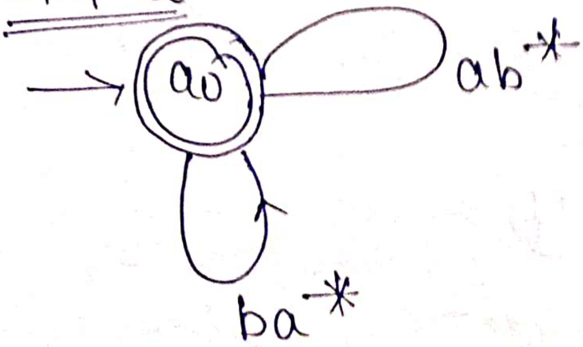
②

Q construct F.A forc RE $(ab+ba)^*$

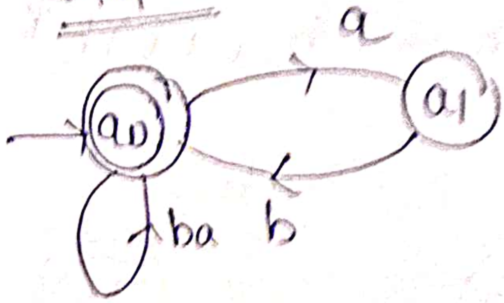
Step-1



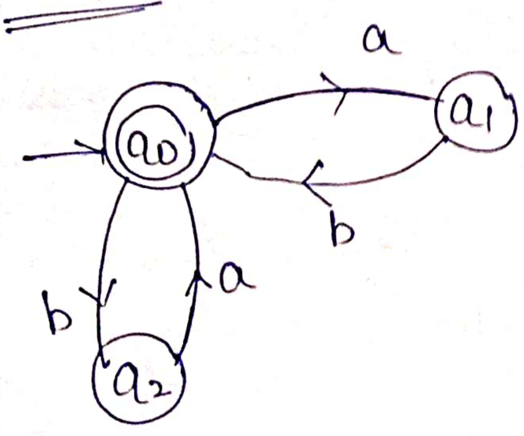
Step-2



Step-3

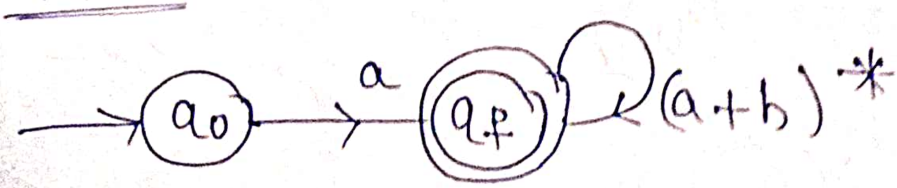


Step-4

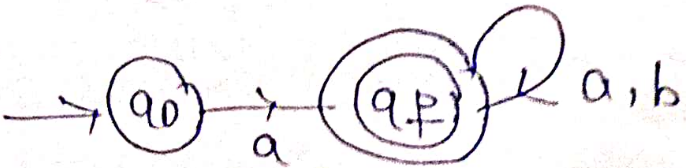


Q construct F.A forc RE $a(a+b)^*$

Step-1



Step-2

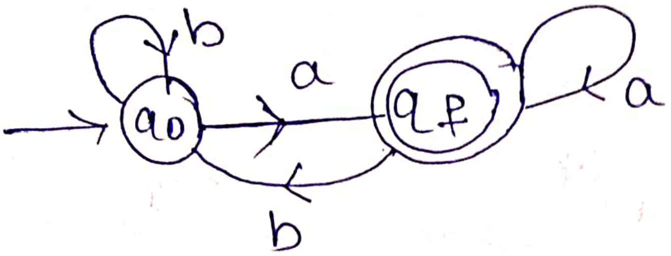


$(a^* + b^*) \cdot a$

$a^*a + bba$

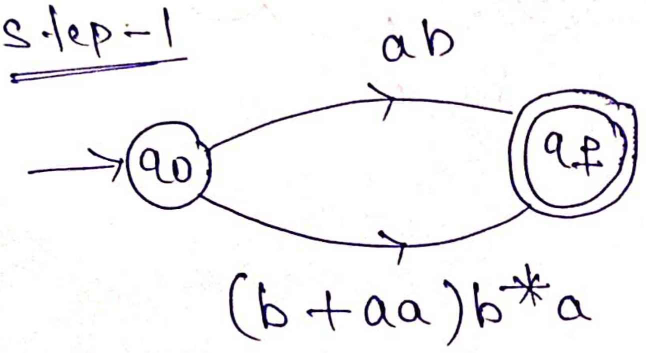
Q: $(a+b)^* a$

step-1

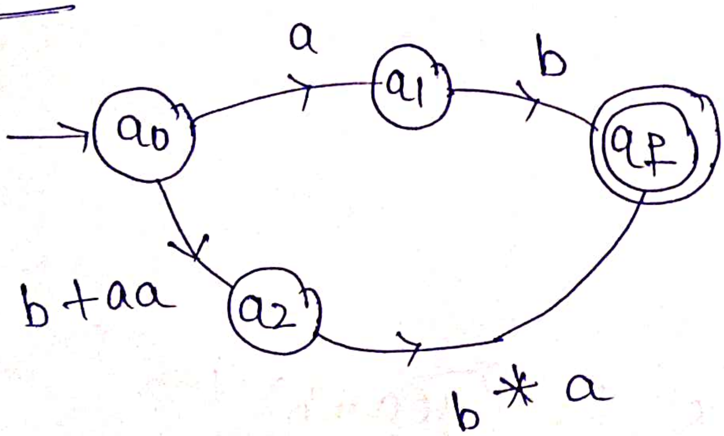


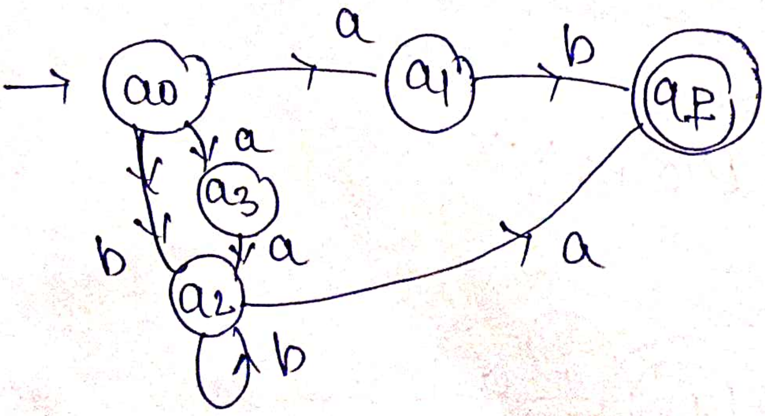
Q: $[ab + (b+aa)b^* a]$

step-1



step-2



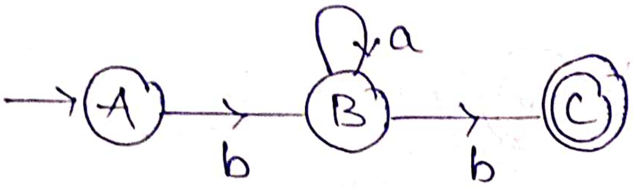


3

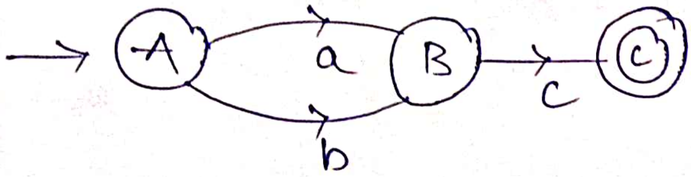
~~Q: $(b/ab^*ab^*)^*$~~

Q: ba^*b {bb, bab, baaab ...}

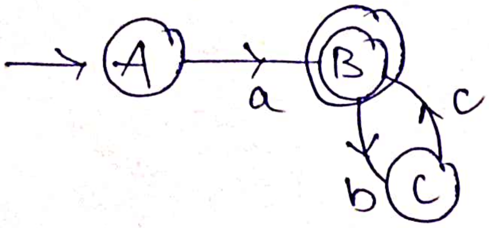
Step – 1



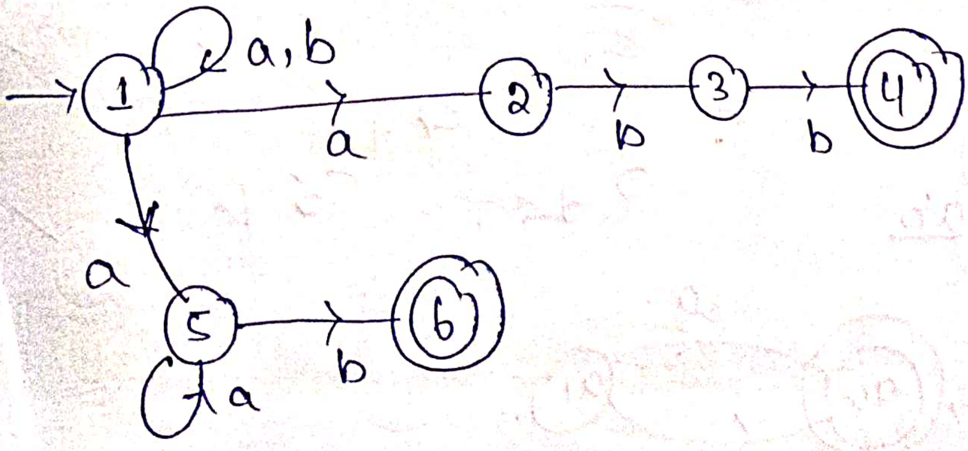
Q: $(a+b)\cdot c$ {ac, bc}



Q: $a(bc)^*$ {a, abc, abcbc ...}



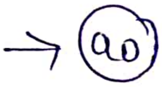
Q $(a|b)^*(abb|a^{+}b)$



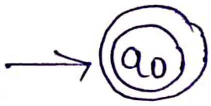
Finite Automata from ~~to~~ RE

Q:

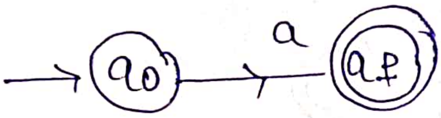
(1) ϕ



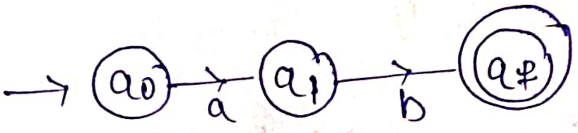
(2) ε



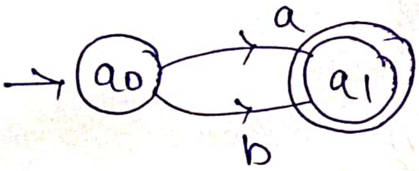
(3) a



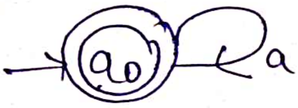
(4) a ~~•~~ b



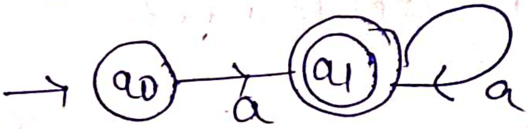
(5) a+b



(6) a*

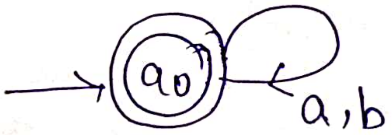


(7) a+

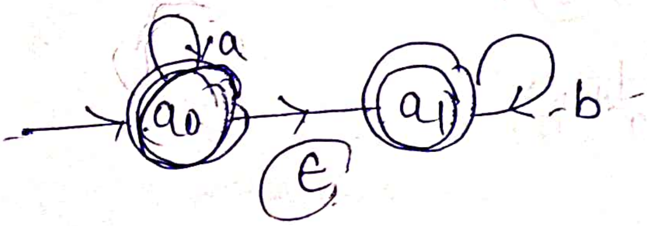


ϵ, a, aa

(8) (a+b)*



(9) a*b*



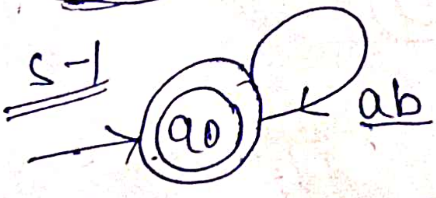
1.10 = 10

1.ε = 1

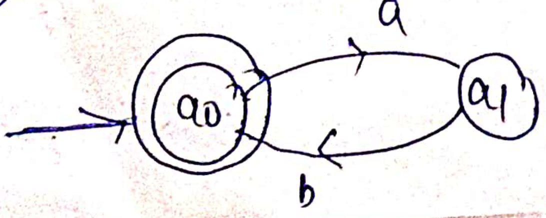
a ~~*~~ εb ~~*~~

(10) (ab)*

S-1

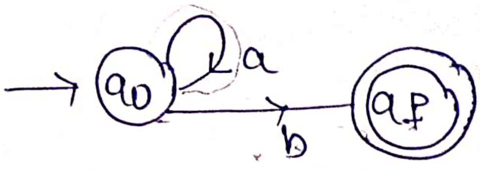


S-2

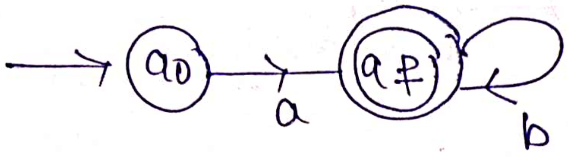


(4)

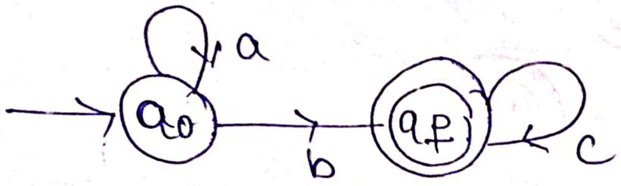
(11) a^*b



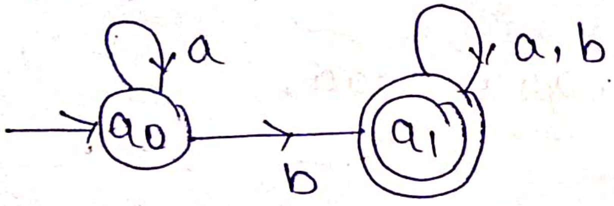
(12) ab^*



(13) a^*bc^*

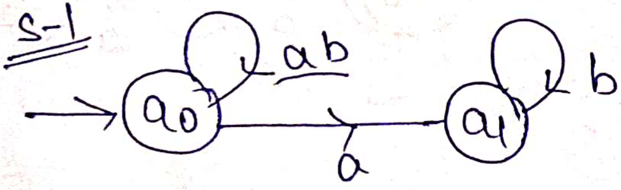


(14) $a^*b(a+b)^*$

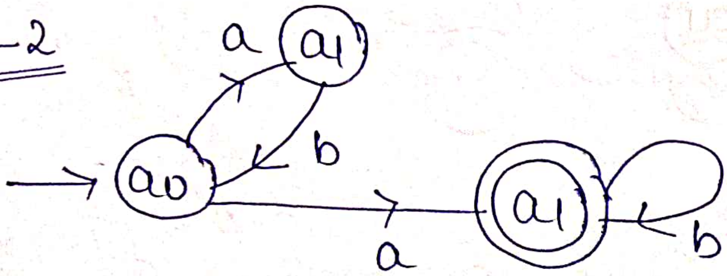


(15) $(ab)^*ab^*$

S-1

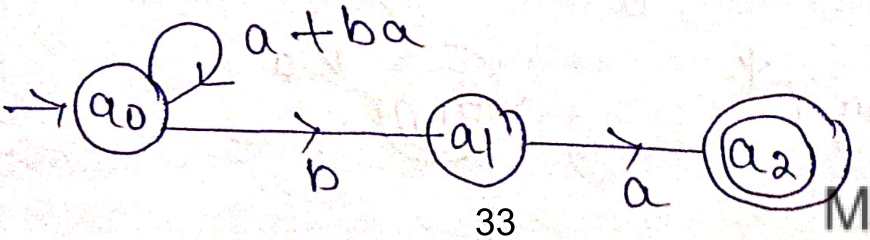


S-2



(16) $(a+ba)^*ba$

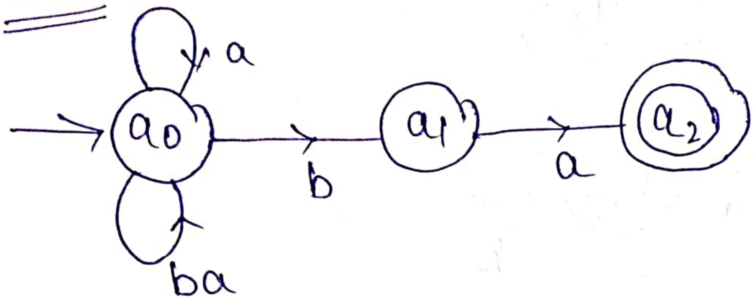
S-1



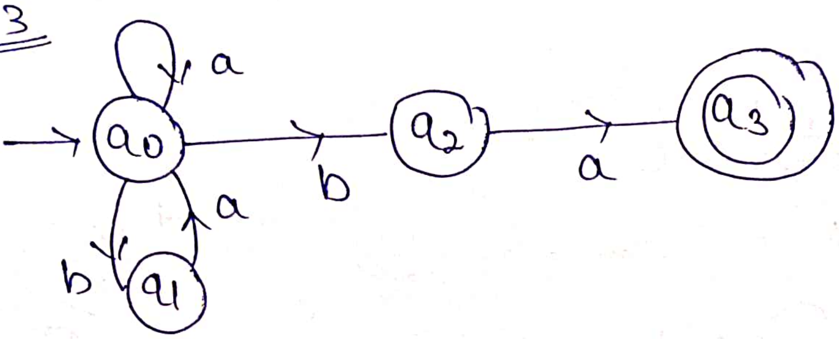
$a^*b = (\epsilon, a, aa)b$

b, ab

S-2

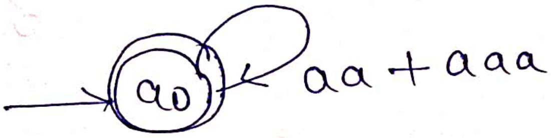


S-3

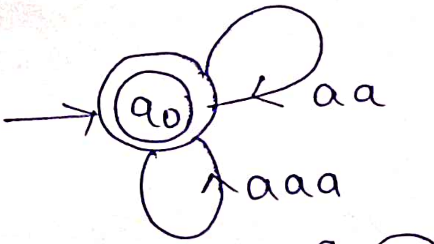


(17) (aa+aaa)* {ε, aa, aaa, aaaa,

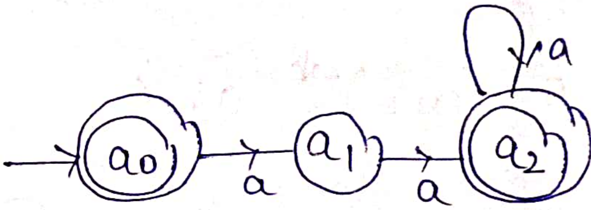
S-1



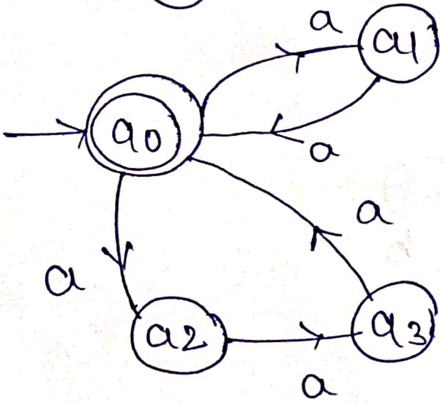
S-2



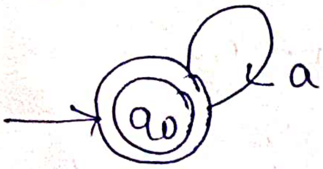
OR



S-3



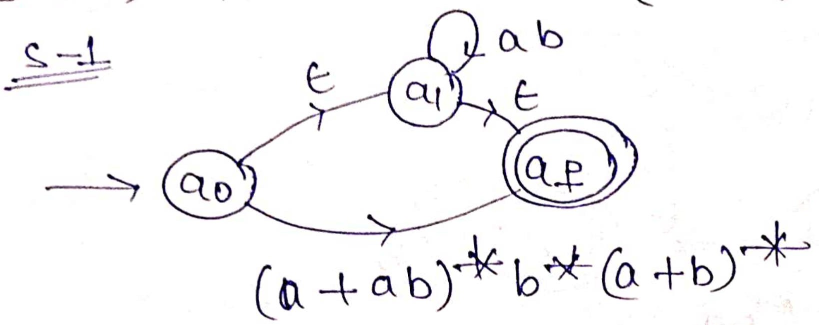
(18) (a+aaaa)*



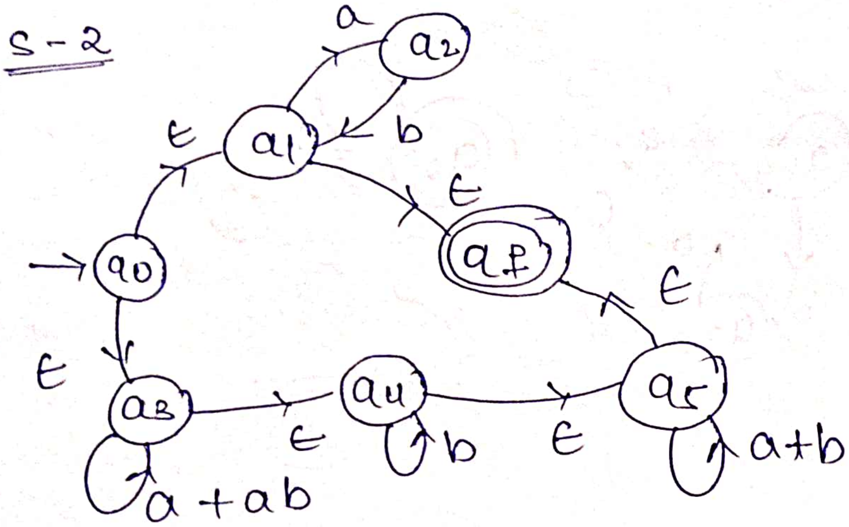
5

(19) $(ab)^* + (a+ab)^* b^* (a+b)^*$

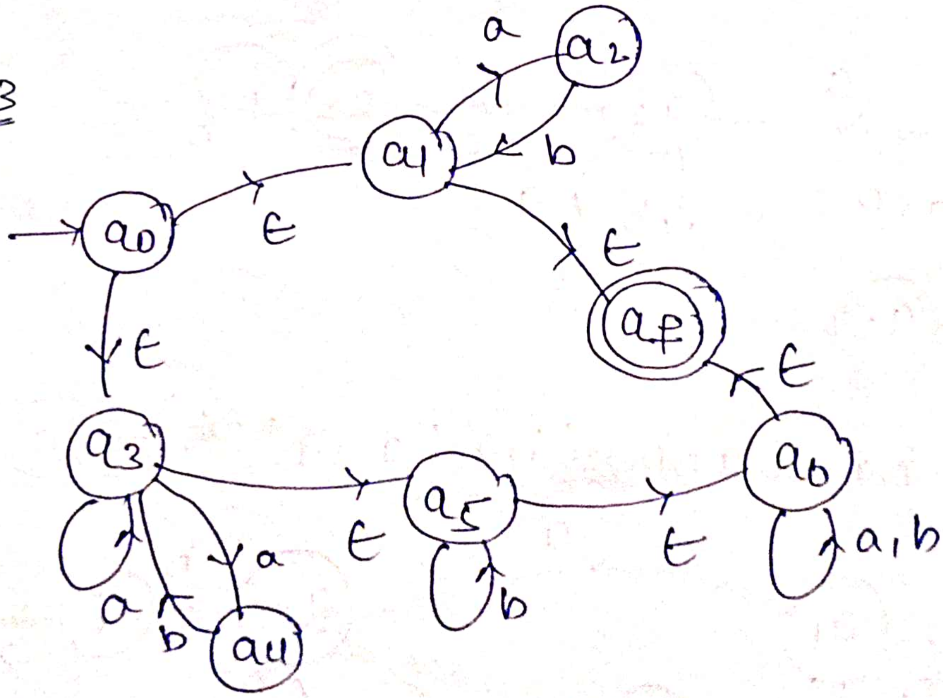
S-1



S-2

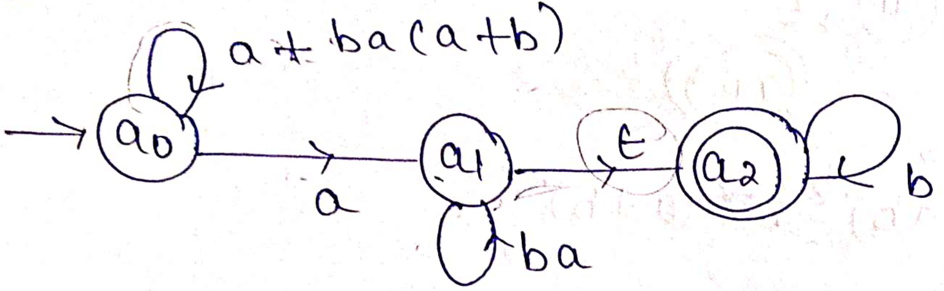


S-3

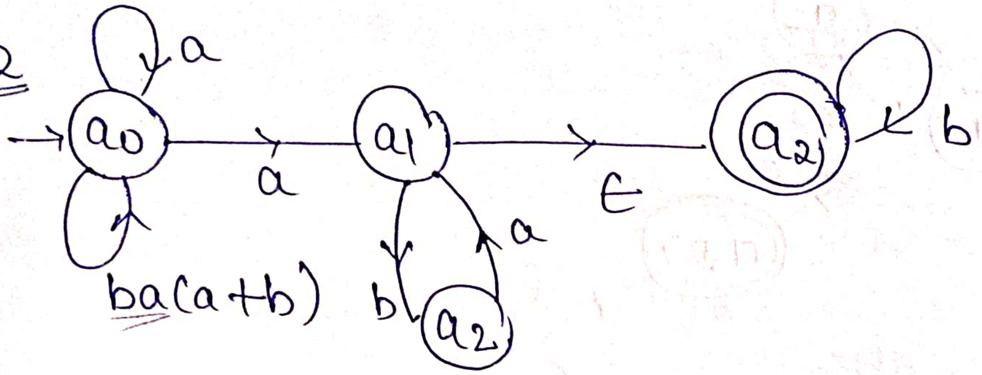


(20) $[a + ba(a+b)]^* a(ba)^* b^*$

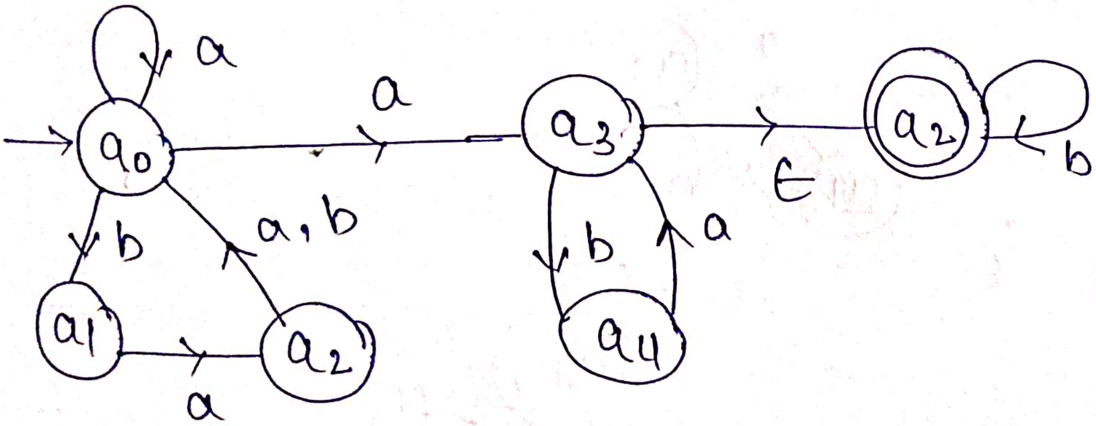
S-1



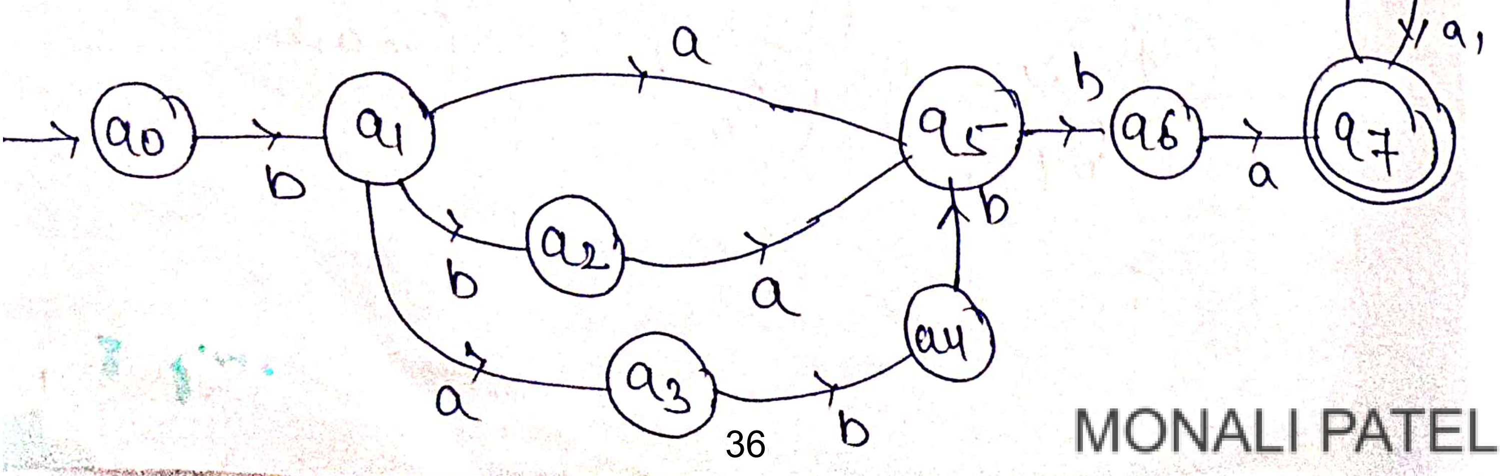
S-2



S-3



$b(a + ba + abb)(ba(a+b)^*)$

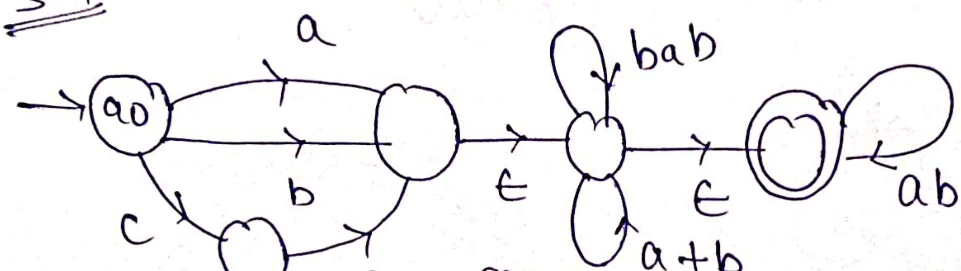


(22) $(a+b+ca)((bab)^* + (a+b)^*)^* (ab)^*$

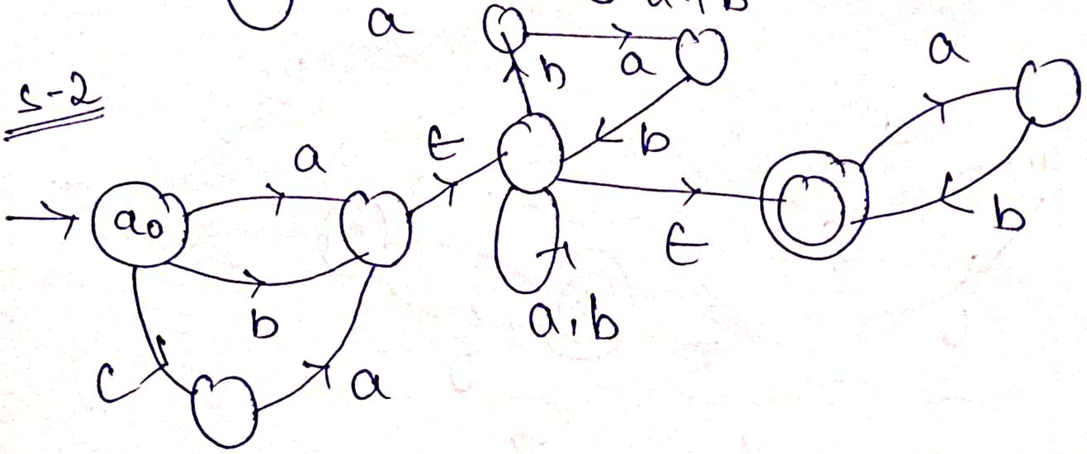
$(a^* + b^*)^* = (a+b)^*$

$= (a+b+ca)(bab + (a+b))^* (ab)^*$

S-1

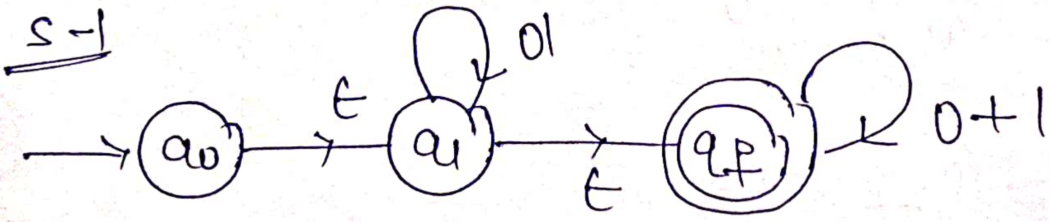


S-2

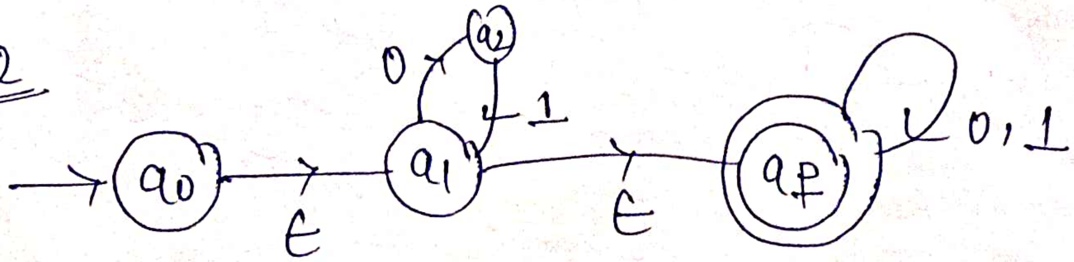


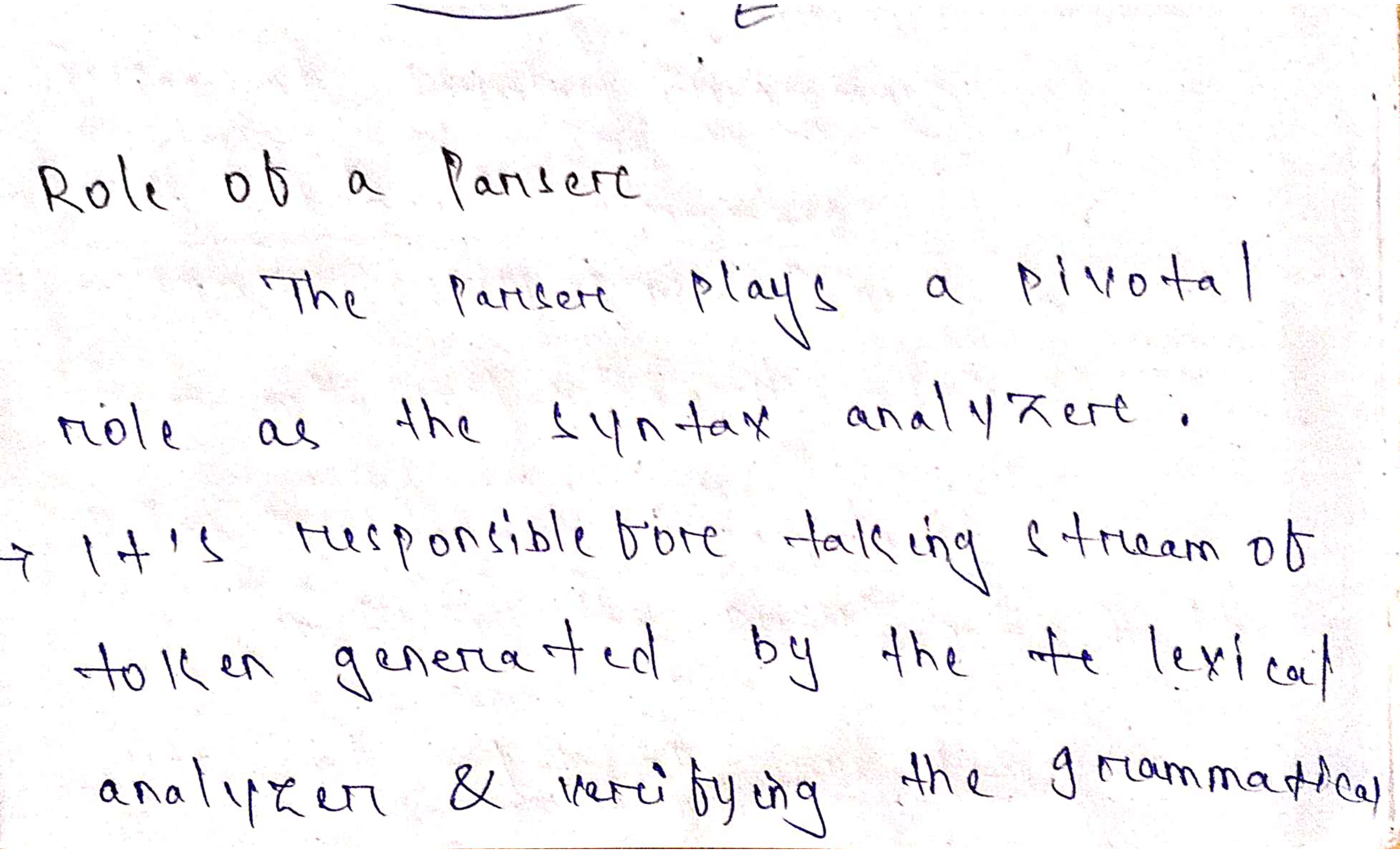
(23) $01^* (0+1)^*$

S-1



S-2

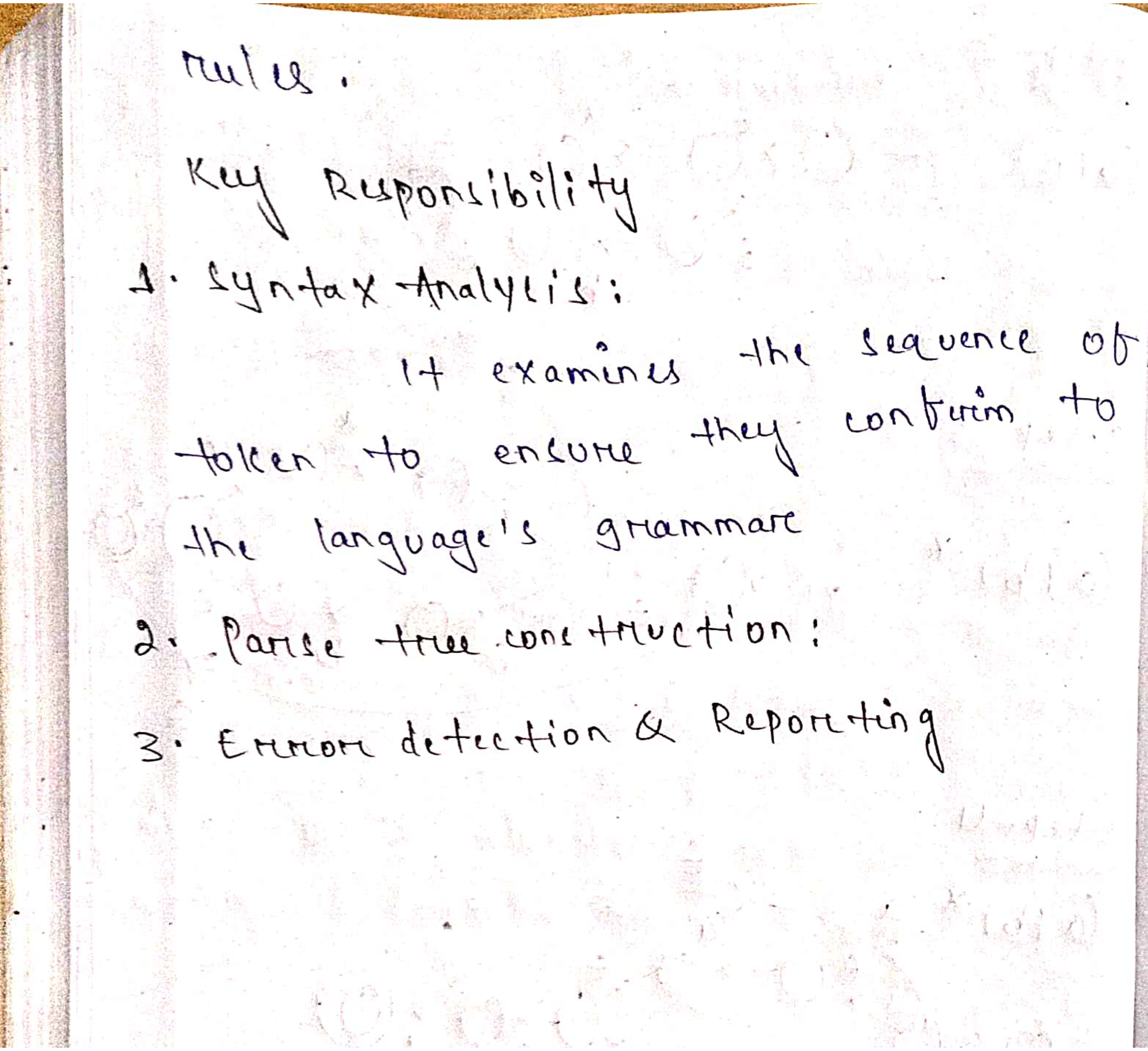




Role of a Parser

The parser plays a pivotal role as the syntax analyzer.

→ It's responsible for taking stream of token generated by the ~~te~~ lexical analyzer & verifying the grammatical



rules.

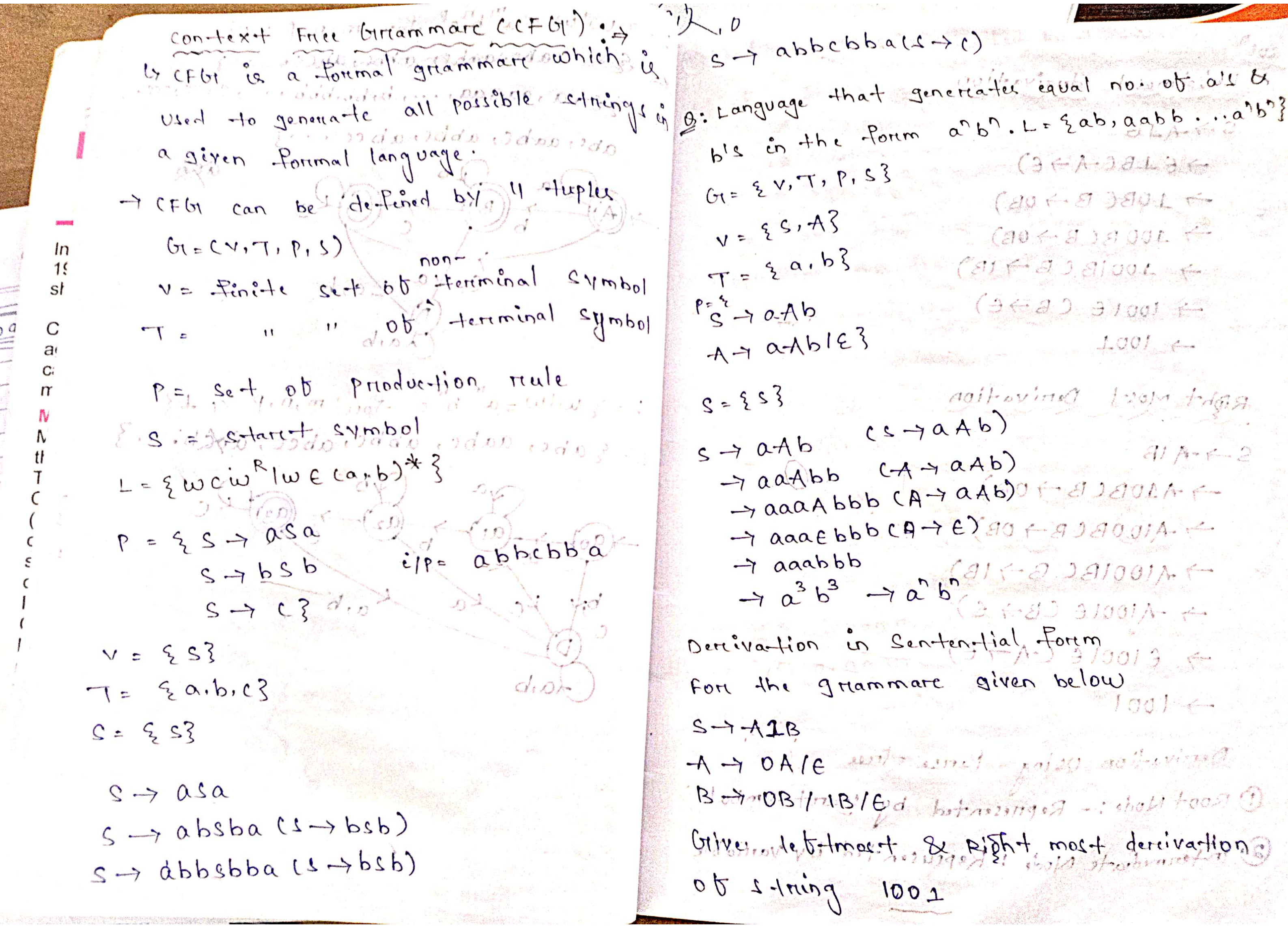
Key Responsibility

1. Syntax Analysis:

It examines the sequence of token to ensure they conform to the language's grammar

2. Parse tree construction:

3. Error detection & Reporting



Context Free Grammare (CFG) :→

↳ CFG is a formal grammare which is used to generate all possible strings in a given formal language.

→ CFG can be defined by 4 tuples

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

V = finite set of non-terminal symbol

T = " " of terminal symbol

P = set of production rule

S = start symbol

$L = \{ wcw^R \mid w \in (a,b)^* \}$

$P = \{ S \to aSa$
$S \to bSb$
$S \to c \}$

i/p = abbcbba

$V = \{S\}$
$T = \{a, b, c\}$
$S = \{S\}$

$S \to aSa$
$S \to absba \; (S \to bSb)$
$S \to abbsbba \; (S \to bSb)$
$S \to abbcbba \; (S \to c)$

Q: Language that generates equal no. of a's & b's in the form $a^n b^n$. $L = \{ab, aabb \ldots a^n b^n\}$

$G = \{V, T, P, S\}$
$V = \{S, A\}$
$T = \{a, b\}$
$P = \{ S \to aAb$
$A \to aAb / \epsilon \}$

$S = \{S\}$

$S \to aAb \quad (S \to aAb)$
$\to aaAbb \quad (A \to aAb)$
$\to aaaAbbb \; (A \to aAb)$
$\to aaa\epsilon bbb \; (A \to \epsilon)$
$\to aaabbb$
$\to a^3 b^3 \to a^n b^n$

Derivation in Sentential form for the grammare given below

$S \to A1B$
$A \to 0A / \epsilon$
$B \to 0B / 1B / \epsilon$

Give leftmost & Right most derivation of string 1001

Sol^n

Leftmost derivation

$S \rightarrow A1B$

$\rightarrow \epsilon 1B \; (A \rightarrow \epsilon)$

$\rightarrow 10B \; (B \rightarrow 0B)$

$\rightarrow 100B \; (B \rightarrow 0B)$

$\rightarrow 1001B \; (B \rightarrow 1B)$

$\rightarrow 1001\epsilon \; (B \rightarrow \epsilon)$

$\rightarrow 1001$

Right Most Derivation

$S \rightarrow A1B$

$\rightarrow A10B \; (B \rightarrow 0B)$

$\rightarrow A100B \; (B \rightarrow 0B)$

$\rightarrow A1001B \; (B \rightarrow 1B)$

$\rightarrow A1001\epsilon \; (B \rightarrow \epsilon)$

$\rightarrow \epsilon 1001\epsilon \; (A \rightarrow \epsilon)$

$\rightarrow 1001$

Derivation Using Parse tree

① Root Node :- Represented by start symbol

② Intermediate Node : Represented by Variable

③ Leaf Node ; Represented by terminal or ε

eg. For given grammar below

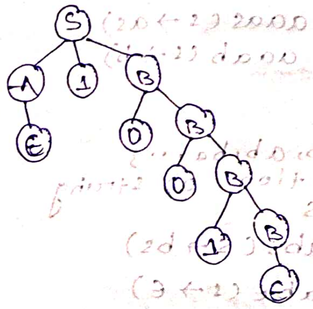
$S \rightarrow A1B$

$A \rightarrow 0A \mid \epsilon$

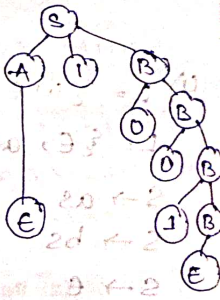
$B \rightarrow 0B \mid 1B \mid \epsilon$

Give parse tree for leftmost & rightmost derivation of the string 1001

Leftmost derivation



Rightmost derivation



Writing Grammar for a Language

① $L = \{\epsilon, a, aa, \ldots\}$ a^*

$S \rightarrow aS$

$S \rightarrow \epsilon$

Generating String

$S \rightarrow aS$

$\rightarrow aaS$

$\rightarrow aaaS$

$\rightarrow aaa\epsilon$

$\rightarrow aaa$

② $L = \{a, aa, aaa \ldots\}$

$S \to aS$

$S \to a$

Generation of String

$S \to aS$

$\to aa\,(S \to a)$

③ $L = \{b, ab, aab, aaab \ldots\}$

$S \to aS$

$S \to b$

Generation of String

$S \to aS$

$S \to aaS\,(S \to aS)$

$\to aaaS\,(S \to aS)$

$\to aaab\,(S \to b)$

4) $L = \{w \in \{a,b\}^*\}$

$L = \{\epsilon, a, b, aa, bb, ab, ba \ldots\}$

$S \to aS$

$S \to bS$

$S \to \epsilon$

Generation of String

$S \to aS$

$S \to abS\,(S \to bS)$

$S \to ab\epsilon\,(S \to \epsilon)$

$S \to ab$

5) $L = \{a^n b^n \mid n \geq 0\}$

$L = \{\epsilon, ab, aabb, aaabbb \ldots a^n b^n \ldots\}$

$S \to aSb$

~~$S \to ab$~~

$S \to \epsilon$

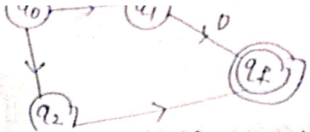
Generation of String

$S \to aSb$

$\to aaSbb$

$\to aaaSbbb$

$\to aaa\epsilon bbb\,(S \to \epsilon)$



6) $L = \{a^n b^n \mid n \geq 1\}$

$L = \{ab, aabb, aaabbb \ldots a^n b^n \ldots\}$

$S \to aSb$

$S \to ab$

Generation of String

$S \to aSb$

$\to aaSbb\,(S \to aSb)$

$\to aaaSbbb\,(S \to aSb)$

$\to aaa\epsilon bbb\,(S \to \epsilon)$

$\to aaabbb$

7) $L = \{w \in \{a,b\}^* \mid w$ is palindrome of oddlength$\}$

$\to babab \leftarrow$

$S \to aSa$

$S \to bSb$

$S \to a$

$S \to b$

Generation of String

$S \to aSa$

$\to abSba\,(S \to bSb)$

$\to abaSaba\,(S \to aSa)$

$\to abababa\,(S \to b)$

8) $L = \{w \in \{a,b\}^* \mid w$ is palindrome of evenlength$\}$

$S \to aSa$

$S \to bSb$

$S \to \epsilon$

Generation of String

$S \to aSa$

$\to abSba\,(S \to bSb)$

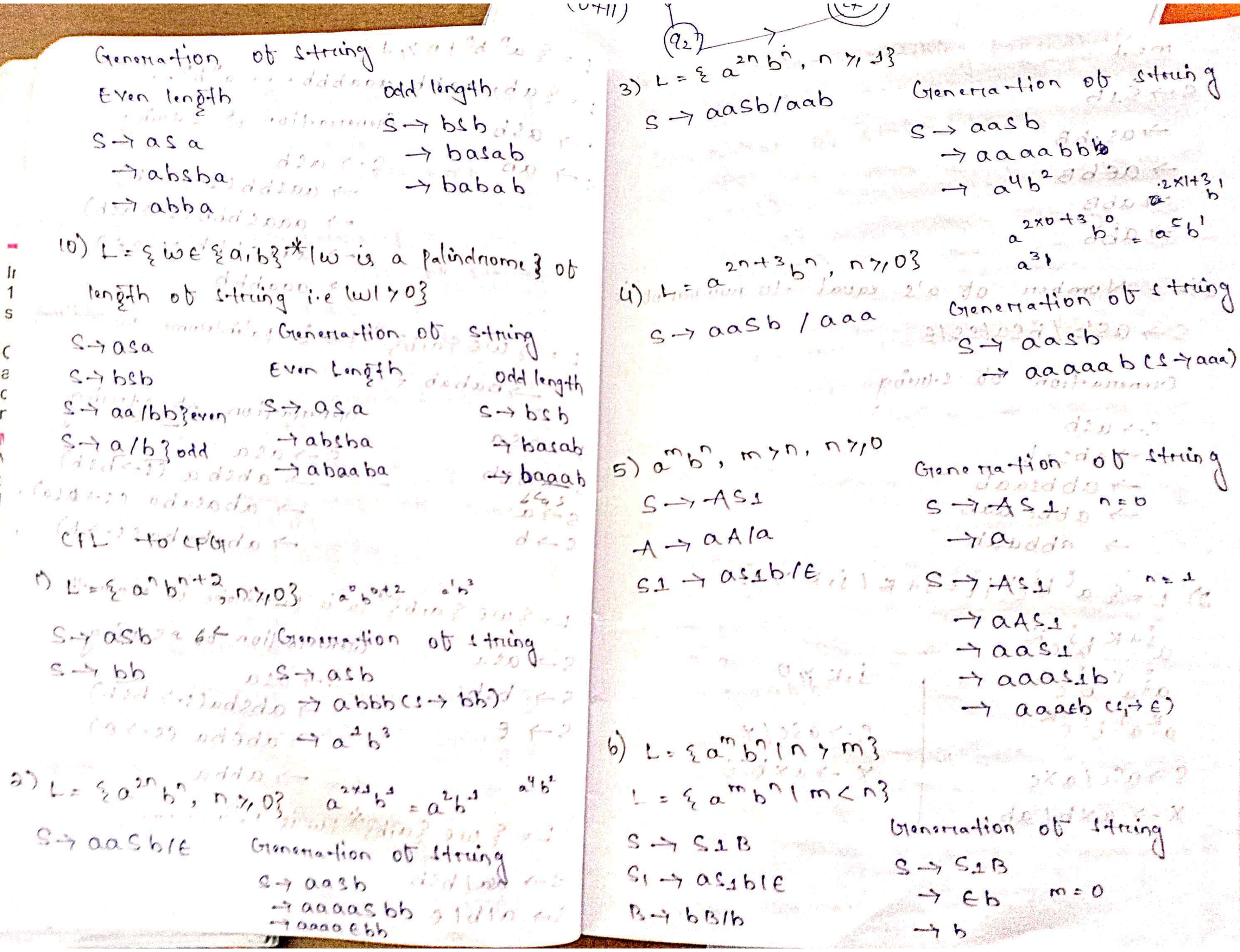
$\to ab\epsilon ba\,(S \to \epsilon)$

$\to abba$

9) $L = \{w \in \{a,b\}^* \mid w$ is Palindrome$\}$

$S \to aSa \mid bSb$

$S \to a \mid b \mid \epsilon$



Generation of string

Even length

$S \to aSa$
$\to abSba$
$\to abba$

Odd length

$S \to bSb$
$\to baSab$
$\to babab$

10) $L = \{ w \in \{a,b\}^* \mid w$ is a palindrome$\}$ of length of string i.e $|w| \geq 0 \}$

$S \to aSa$
$S \to bSb$
$S \to aa/bb \}$ even
$S \to a/b \}$ odd

Generation of string

Even Length

$S \to aSa$
$\to abSba$
$\to abaaba$

Odd length

$S \to bSb$
$\to baSab$
$\to baaab$

CFL to CFG

1) $L = \{ a^n b^{n+2}, n \geq 0 \}$ $a^0 b^{0+2}$ $a^1 b^3$

$S \to aSb$
$S \to bb$

Generation of string

$S \to aSb$
$\to abbb (S \to bb)$
$\to a^1 b^3$

2) $L = \{ a^{2n} b^n, n \geq 0 \}$ $a^{2\times1} b^1 = a^2 b^1$ $a^4 b^2$

$S \to aaSb/\epsilon$

Generation of string

$S \to aaSb$
$\to aaaaSbb$
$\to aaaa\epsilon bb$

3) $L = \{ a^{2n} b^n, n \geq 1 \}$

$S \to aaSb/aab$

Generation of string

$S \to aaSb$
$\to aaaabb$
$\to a^4 b^2$

4) $L = a^{2n+3} b^n, n \geq 0 \}$

$a^{2\times1+3} b^1$
$a^{2\times0+3} b^0 = a^5 b^1$
a^3

$S \to aaSb / aaa$

Generation of string

$S \to aaSb$
$\to aaaaab (S \to aaa)$

5) $a^m b^n, m > n, n \geq 0$

$S \to AS_1$
$A \to aA/a$
$S_1 \to aS_1b/\epsilon$

Generation of string

$S \to AS_1$ $n=0$
$\to a$

$S \to AS_1$ $n=1$
$\to aAS_1$
$\to aaS_1$
$\to aaaS_1b$
$\to aaa\epsilon b (S_1 \to \epsilon)$

6) $L = \{ a^m b^n \mid n > m \}$

$L = \{ a^m b^n \mid m < n \}$

$S \to S_1B$
$S_1 \to aS_1b/\epsilon$
$B \to bB/b$

Generation of string

$S \to S_1B$
$\to \epsilon b$ $m=0$
$\to b$

$m=1$

$S \rightarrow S_1B$

$\rightarrow aS_1bB$

$\rightarrow a\epsilon bB$

$\rightarrow abB$

$\rightarrow abb$

1) L = Number of a's equal to number of b's

$S \rightarrow aSb / bSa / SS / \epsilon$

Generation of string

$S \rightarrow aSb$

$\rightarrow abSab$

$\rightarrow abbSaab$

$\rightarrow abb\epsilon aab$

$\rightarrow abbaab$

2) $L = \{a^i b^j c^k \mid i = j+k \,/\, j,k \geq 1\}$

$a^{j+k} b^j c^k$

$a^j a^k b^j c^k$

$a^k a^j b^j c^k$

$S \rightarrow aSc / aXc$

$X \rightarrow aXb / ab$

$j,k \geq 0$

$S \rightarrow aSc / X$

$X \rightarrow aXb / \epsilon$

3) Give CFG for matching parenthesis

$S \rightarrow (S) / SS / \epsilon$

Generation of string

$S \rightarrow (S)$

$\rightarrow (SS)$

$\rightarrow ((S)S)$

$\rightarrow (()(S))$

$\rightarrow (()())$

$()$

4) Give CFG for $L = \{a^n b^m \mid n \neq m\}$

$S \rightarrow aSb / X / Y$

$X \rightarrow aX / a$

$Y \rightarrow bY / b$

Generation of string

$S \rightarrow aSb$

$\rightarrow aaSbb$

$\rightarrow aaXbb$

$\rightarrow aaaXbb$

$\rightarrow aaaabb$

Derivation

It is a Sequence of Production rules, It is used to get the input String through the prodⁿ rule.

→ During Parsing, we have to take 2 decision

a) Decide the non-terminal which is to be replaced

b) Decide the Production rule by which the non-terminal will be replaced

We have 2 options to decide which non-terminal to be placed

1) Left-most Derivation

The input is Scanned & replaced with the prodⁿ rule from left to right.

(Read the input String from left to right)

eg: E = E + E

E = E - E

E = a|b input a-b+a

Ans: E = E + E

= E - E + E

= a - E + E

= a - b + E

= a - b + a

2) Rightmost Derivation:

The input is Scanned & replaced with the Production rule from right to left.

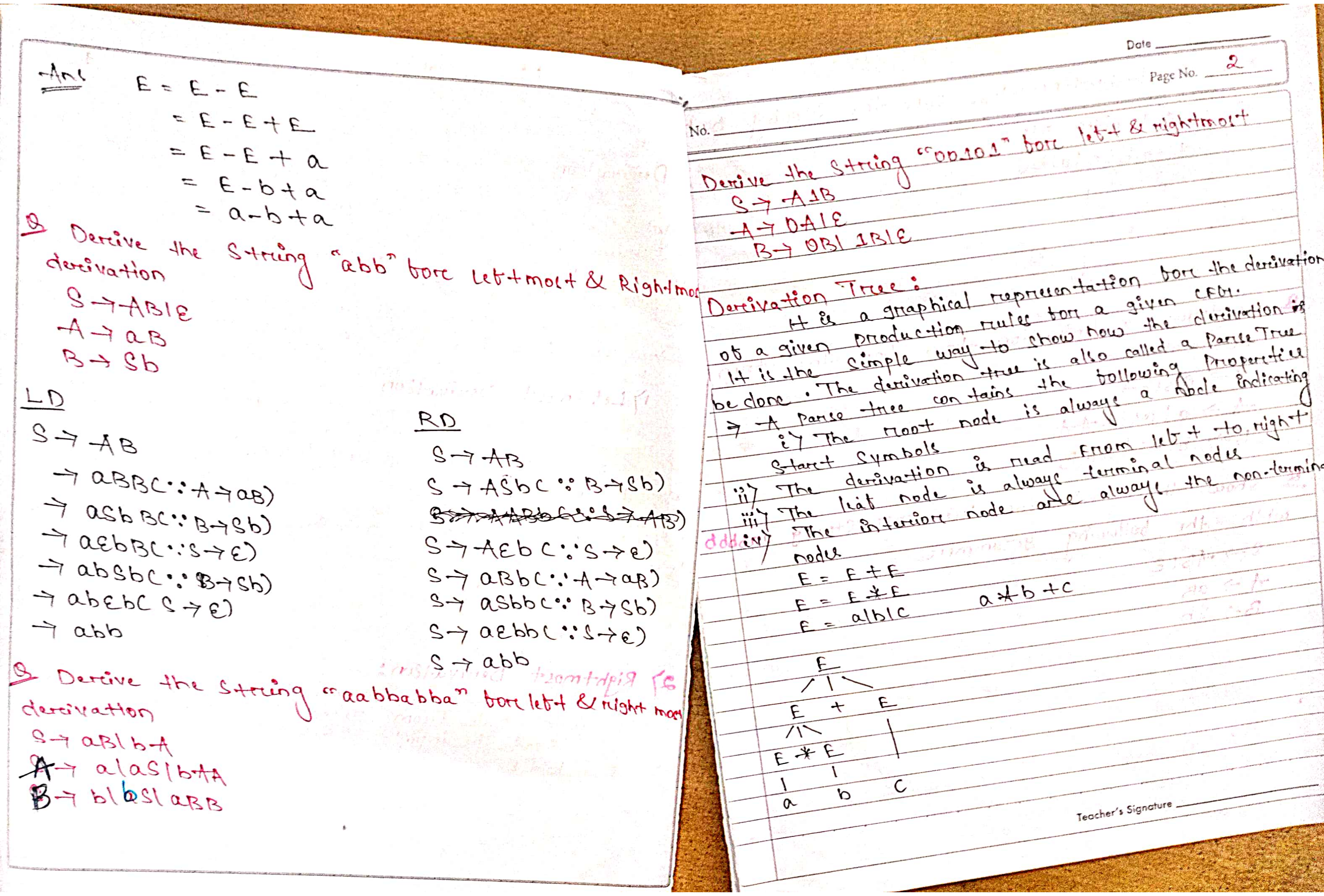
(Read the input String from right to left)

eg E = E + E

E = E - E

E = a|b input a-b+a

Teacher's Signature ____________



Ans

E = E - E

= E - E + E

= E - E + a

= E - b + a

= a - b + a

Q Derive the String "abb" for leftmost & Rightmost derivation

S → AB | ε

A → aB

B → Sb

LD

S → AB

→ aBBC ∵ A → aB)

→ aSbBC ∵ B → Sb)

→ aεbBC ∵ S → ε)

→ abSbC ∵ B → Sb)

→ abεbC S → ε)

→ abb

RD

S → AB

S → ASbC ∵ B → Sb)

~~S → AABbC ∵ S → AB)~~

S → AεbC ∵ S → ε)

S → aBbC ∵ A → aB)

S → aSbbC ∵ B → Sb)

S → aεbbC ∵ S → ε)

S → abb

Q Derive the String "aabbabba" for left & right most derivation

S → aB | bA

~~A~~ → a | aS | bAA

B → b | bS | aBB

Date

Page No. 2

No.

Derive the String "00101" for left & rightmost

S → A1B

A → 0A | ε

B → 0B | 1B | ε

Derivation Tree:

It is a graphical representation for the derivation of a given production rules for a given CFG. It is the simple way to show how the derivation is be done. The derivation tree is also called a Parse Tree

→ A parse tree contains the following properties

i) The root node is always a node indicating start symbols

ii) The derivation is read from left to right

iii) The leaf node is always terminal nodes

iv) The interior node are always the non-terminal nodes

E = E + E

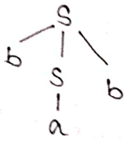
E = E * E

E = a | b | c

a * b + c

Teacher's Signature

Draw a derivation tree for the string ba from CFG given by
S → bSb|a|b



Q construct a derivation tree for the string aabbabba
S → aB|bA
A → a|aS|bAA
B → b|bS|aBB

Q Show the derivation tree for string aabbbb with the following grammar
S → AB|ε
A → aB
B → Sb

Date

Page No. 13

No.

Ambiguity in Grammar

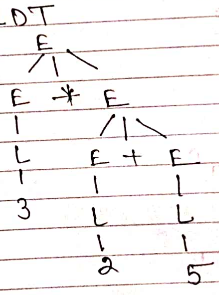
A grammar is said to be ambiguous if there exist more than 1 parse tree for the given input string.

if the grammar is not ambiguous, then it is called unambiguous

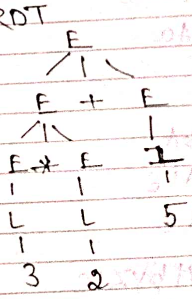
if the grammar has ambiguity, then it is not good for compiler construction, No method can automatically detect & remove the ambiguity by re-writing the whole grammar

Q E → E+E|L
E → E*E
E → (E)
L → ε|0|1|2|...|9   String = 3*2+5

LDT

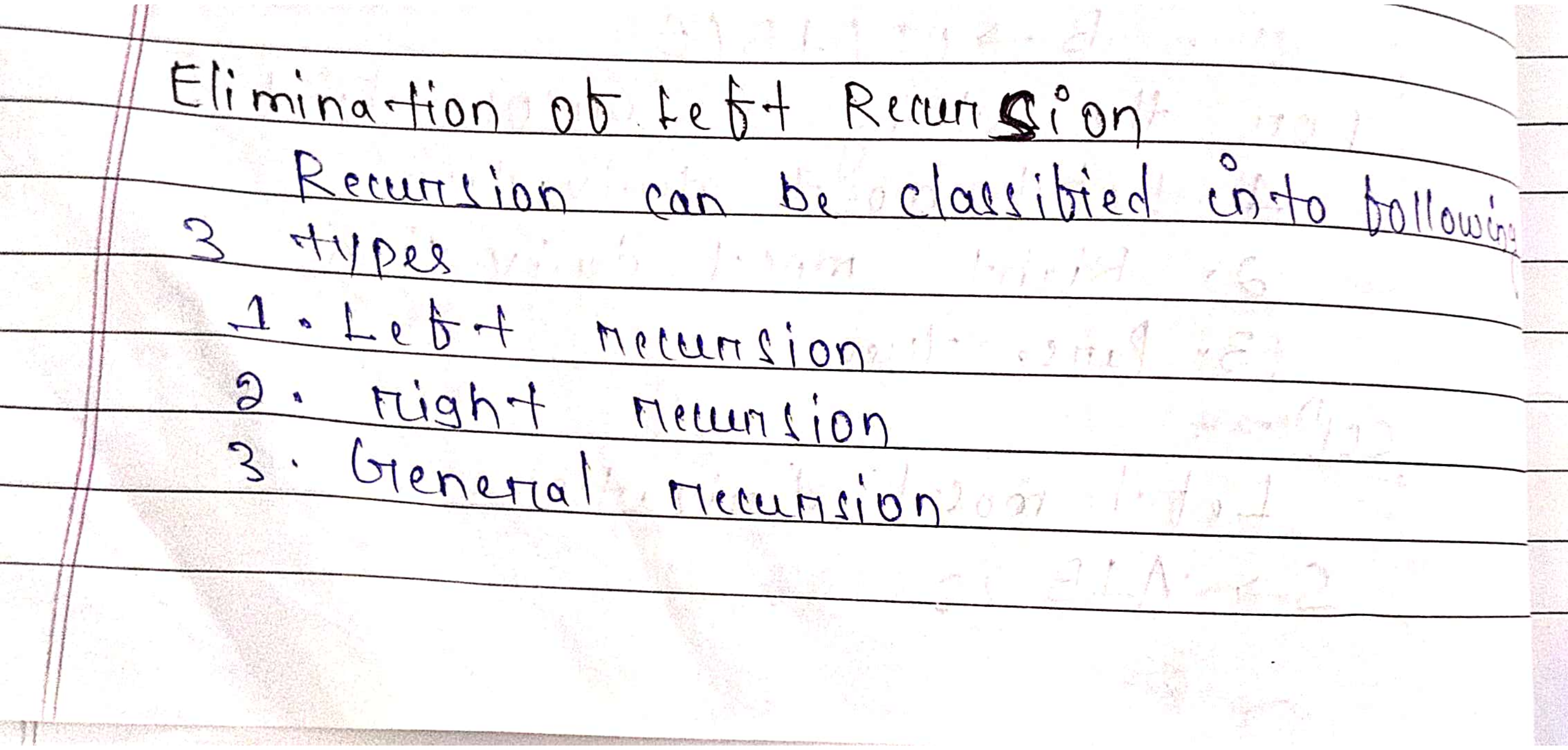


RDT



It is a ambiguous Grammar

Teacher's Signature



Elimination of Left Recursion

Recursion can be classified into following 3 types

1. Left recursion
2. Right recursion
3. General recursion

Date
Page

1. Left recursion

A Production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.

$S \rightarrow Sa / \epsilon$

→ Left recursion is considered to be a problematic situation for top down. therefore it has to be eliminated.

Elimination of Left recursion

It can be eliminated by converting the grammar into right recursive grammare.

if, $A \rightarrow A\alpha / \beta$

then, $A \rightarrow \beta A'$

$A' = \alpha A' / \epsilon$

2. Right recursion.

A prodn of grammar is said to have right recursion if the right most variable of its RHS is same as its LHS

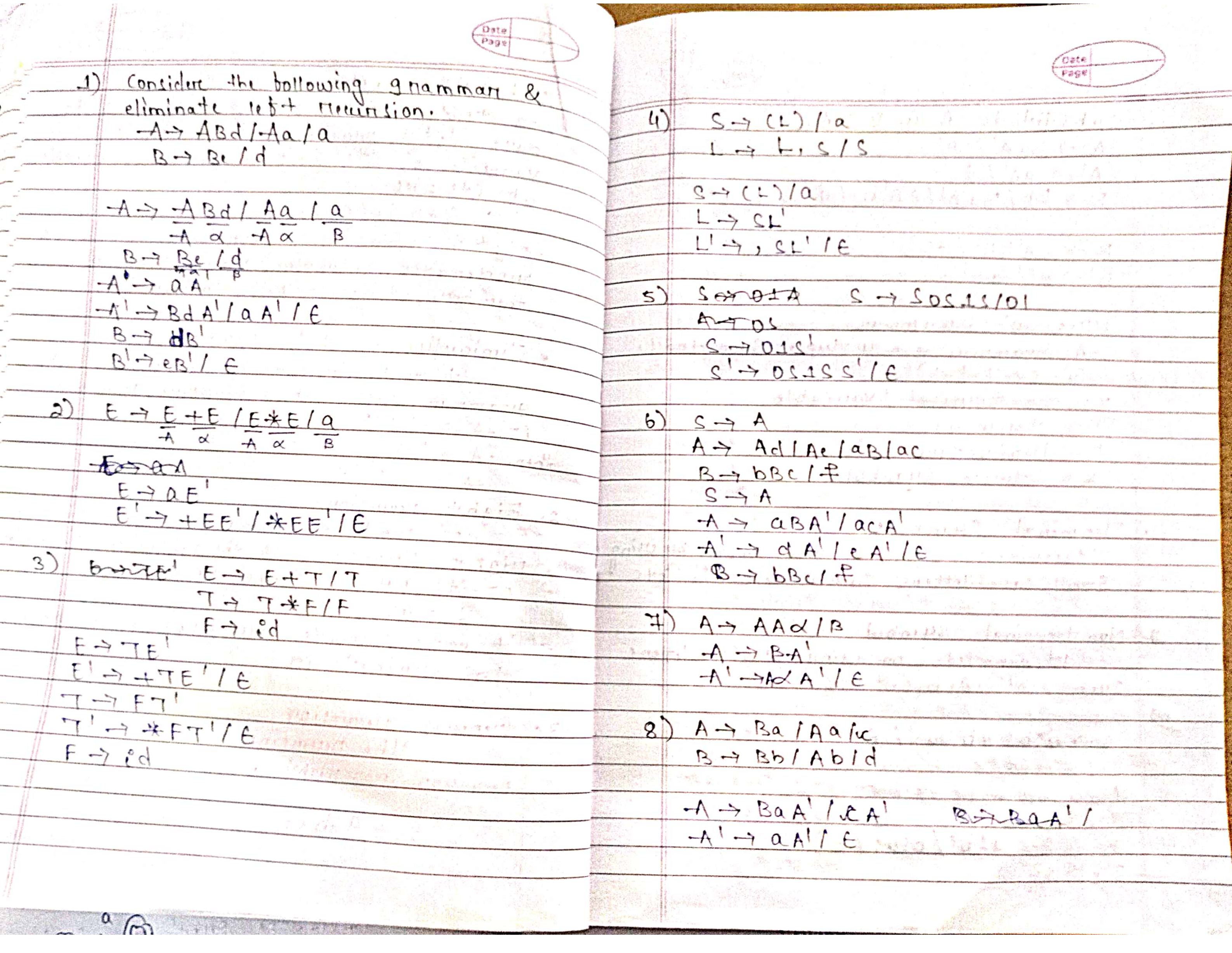
$S \rightarrow aS / \epsilon$

→ It does not create any problem so, no need to eliminate it.

3. General recursion

The recursion which is neither left recursion nor right recursion.

$S \rightarrow aSb / \epsilon$



1) Consider the following grammar & eliminate left recursion.

$A \to ABd / Aa / a$

$B \to Be / d$

$A \to \underbrace{A}_{A}\underbrace{Bd}_{\alpha} / \underbrace{A}_{A}\underbrace{a}_{\alpha} / \underbrace{a}_{\beta}$

$B \to \underbrace{Be}_{\alpha} / \underbrace{d}_{\beta}$

$A' \to aA'$

$A' \to BdA' / aA' / \epsilon$

$B \to dB'$

$B' \to eB' / \epsilon$

2) $E \to \underbrace{E}_{A}\underbrace{+E}_{\alpha} / \underbrace{E}_{A}\underbrace{*E}_{\alpha} / \underbrace{a}_{B}$

~~$E \to aA$~~

$E \to aE'$

$E' \to +EE' / *EE' / \epsilon$

3) ~~$E \to TE'$~~ $E \to E+T / T$

$T \to T*F / F$

$F \to id$

$E \to TE'$

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

$T \to FT'$

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

$F \to id$

4) $S \to (L) / a$

$L \to L,S / S$

$S \to (L) / a$

$L \to SL'$

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

5) ~~$S \to 01A$~~ $S \to S0S1S / 01$

~~$A \to 0S$~~

$S \to 01S'$

$S' \to 0S1SS' / \epsilon$

6) $S \to A$

$A \to Ad / Ae / aB / ac$

$B \to bBc / f$

$S \to A$

$A \to aBA' / acA'$

$A' \to dA' / eA' / \epsilon$

$B \to bBc / f$

7) $A \to AA\alpha / B$

$A \to BA'$

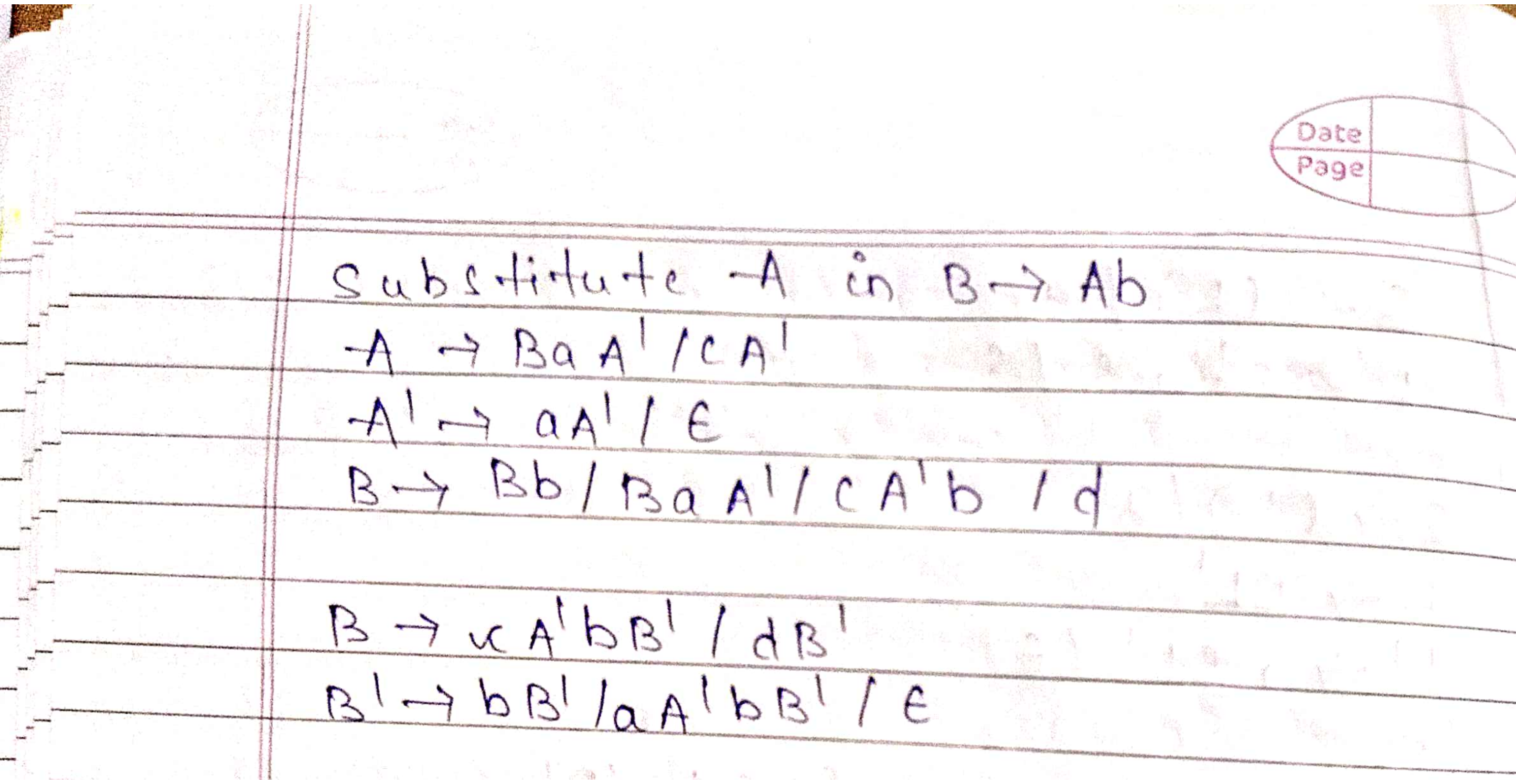
$A' \to A\alpha A' / \epsilon$

8) $A \to Ba / Aa / c$

$B \to Bb / Ab / d$

$A \to BaA' / cA'$ ~~$B \to BaA' /$~~

$A' \to aA' / \epsilon$



Date
Page

Substitute A in $B \rightarrow Ab$

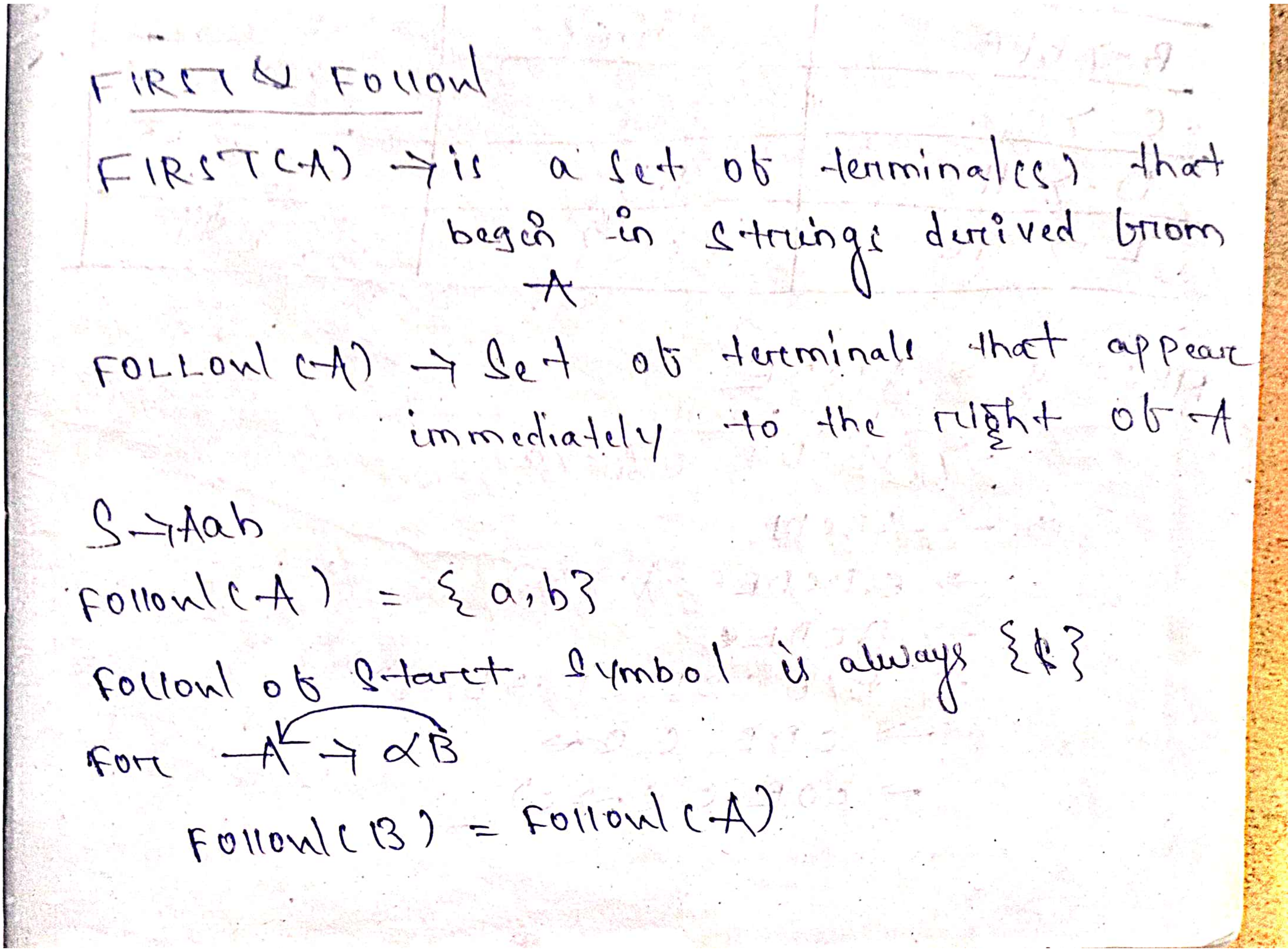
$A \rightarrow BaA' / cA'$

$A' \rightarrow aA' / \epsilon$

$B \rightarrow Bb / BaA' / cA'b / d$

$B \rightarrow cA'bB' / dB'$

$B' \rightarrow bB' / aA'bB' / \epsilon$



FIRST & Follow

FIRST(A) → is a set of terminal(s) that begin in strings derived from A

FOLLOW(A) → Set of terminals that appear immediately to the right of A

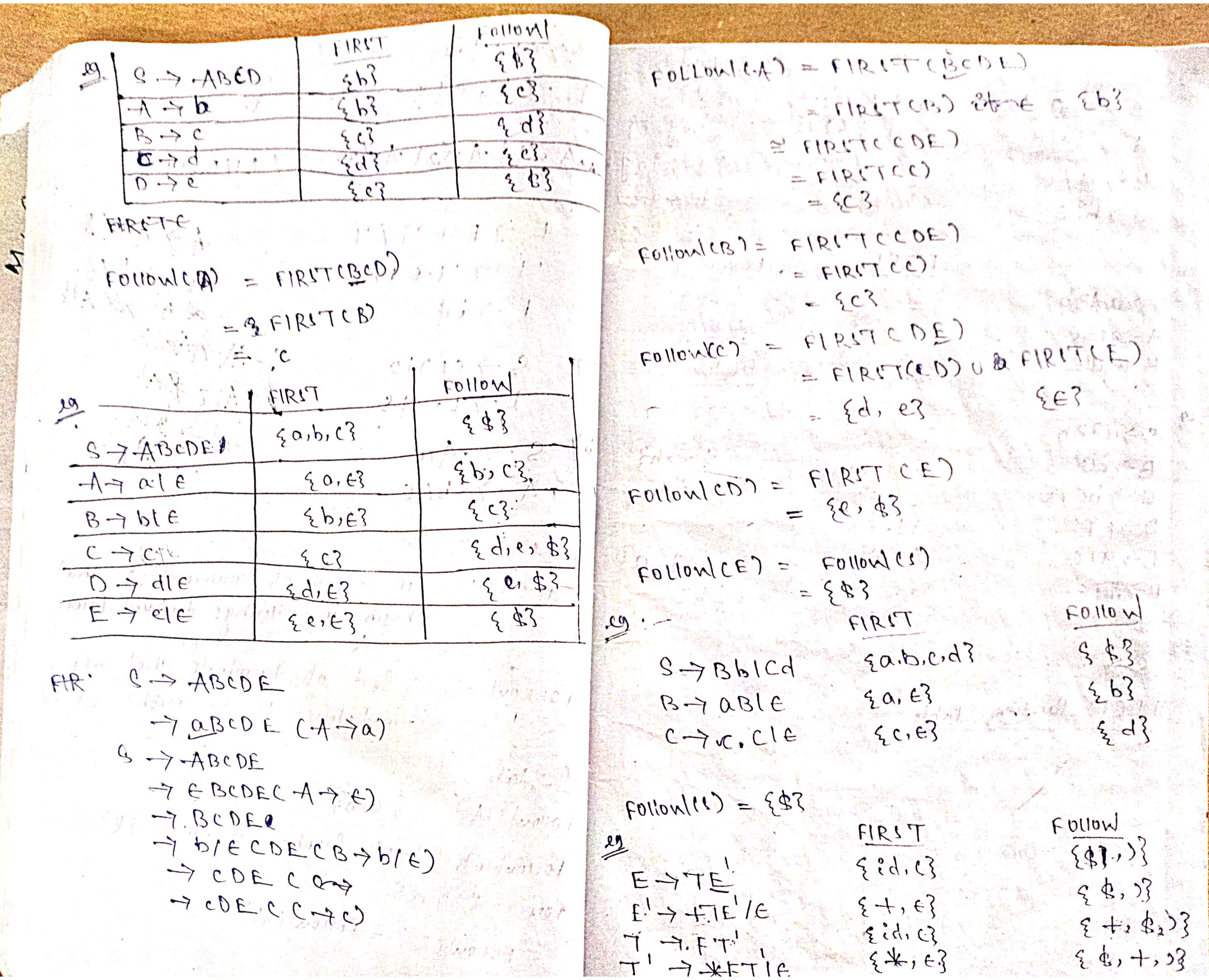
$S \rightarrow Aab$

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

Follow of Start symbol is always {\$}

For $A \rightarrow \alpha B$

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



eg

	FIRST	Follow
$S \to ABCD$	{b}	{$}
$A \to b$	{b}	{c}
$B \to c$	{c}	{d}
$C \to d$	{d}	{e}
$D \to e$	{e}	{$}

~~FIRST~~

Follow(A) = FIRST(BCD)

= FIRST(B)

= c

eg

	FIRST	Follow
$S \to ABCDE$	{a, b, c}	{$}
$A \to a \mid \epsilon$	{a, ε}	{b, c}
$B \to b \mid \epsilon$	{b, ε}	{c}
$C \to c$	{c}	{d, e, $}
$D \to d \mid \epsilon$	{d, ε}	{e, $}
$E \to e \mid \epsilon$	{e, ε}	{$}

FIR: $S \to ABCDE$

$\to aBCDE$ $(A \to a)$

$S \to ABCDE$

$\to \epsilon BCDE$ $(A \to \epsilon)$

$\to BCDE$

$\to b/\epsilon CDE$ $(B \to b/\epsilon)$

$\to CDE$ $(C \to$

$\to cDE$ $(C \to c)$

FOLLOW(A) = FIRST(BCDE)

= FIRST(B) ~~if~~ ε = {b}

= FIRST(CDE)

= FIRST(C)

= {c}

Follow(B) = FIRST(CDE)

= FIRST(C)

= {c}

Follow(C) = FIRST(DE)

= FIRST(D) ∪ FIRST(E)

= {d, e} {ε}

Follow(D) = FIRST(E)

= {e, $}

Follow(E) = Follow(S)

= {$}

eg :-

	FIRST	Follow
$S \to Bb \mid Cd$	{a, b, c, d}	{$}
$B \to aB \mid \epsilon$	{a, ε}	{b}
$C \to c.C \mid \epsilon$	{c, ε}	{d}

Follow(S) = {$}

eg

	FIRST	Follow
$E \to TE'$	{id, c}	{$,)}
$E' \to +TE' / \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{id, c}	{+, $,)}
$T' \to *FT' \mid \epsilon$	{*, ε}	{$, +,)}

eg:-

	FIRST	Follow
$S \to ACB \mid cbB \mid Ba$	{d, g, h, b, a}	{ $ }
$A \to da \mid BC$	{d, g, ~~ε~~ h}	{ h, g, $ }
$B \to g / \epsilon$	{g, ε}	{ $, a } h, g ~~{ [illegible]~~
$C \to h \mid \epsilon$	{h, ε}	{ g, b, $, h }

eg

	FIRST	FOLLOW
$S \to aABb$	{a}	{ $ }
$A \to c / \epsilon$	{c, ε}	{d, b}
$B \to d \mid \epsilon$	{d, ε}	{ b }

eg

	FIRST	Follow
$S \to aBDh$	{a}	{ $ }
$B \to cC$	{c}	{ g, f, h }
$C \to bC \mid \epsilon$	{b, ε}	{ g, f, h }
$D \to EF$	{g, f, ε}	{ h }
$E \to g \mid \epsilon$	{g, ε}	{ f, g } h }
$F \to f \mid \epsilon$	{f, ε}	{ h }

eg →

LL(1) Parsing Table

LL(1) Parsing Table

	FIRST	Follow
$E \to TE'$	{id, (}	{ $,) }
$E' \to +TE' \mid \epsilon$	{+, ε}	{ $,) }
$T \to FT'$	{id, (}	{ +, $,) }
$T' \to *FT' \mid \epsilon$	{*, ε}	{ +, $,) }
$F \to id \mid (E)$	{id, (}	{ *, +, $,) }

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

$E' \to \epsilon$

Follow of E' = {$,)}

eg

	FIRST	Follow
$S \to AaAb \mid BbBa$	{a, b}	{ $ }
$A \to \epsilon$	{ ε }	{a, b}
$B \to \epsilon$	{ ε }	{a, b}

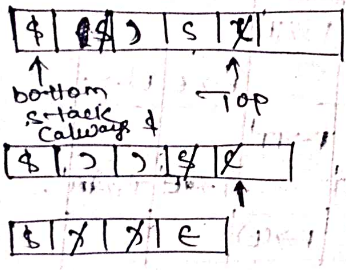
	a	b	$
S	S→AaAb	S→BbBa	
A	A→ε	A→ε	
B	B→ε	B→ε	

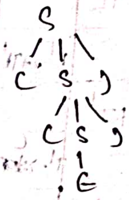
$S \to (S) \mid \epsilon$ FIRST $\{(, \epsilon\}$ FOLLOW $\{\$,)\}$

	(	)	$
S	S→(S)	S→ε	S→ε

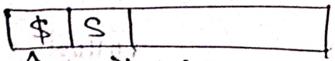
string to be generated

(())$ Reverse order





(())$

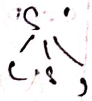


bottom of stack   initially top of the stack will be Start symbol

S on seeing (

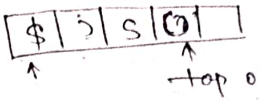
we use the prodn S→(S)

So tree looks like



Now replace S with RHS such that leftmost symbol appear on top

i.e

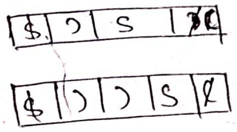


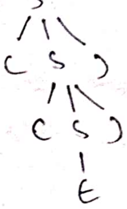
(())

top of the stack

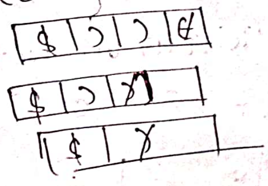
Now, Top of stack is open brace & input is open brance

So, pop off





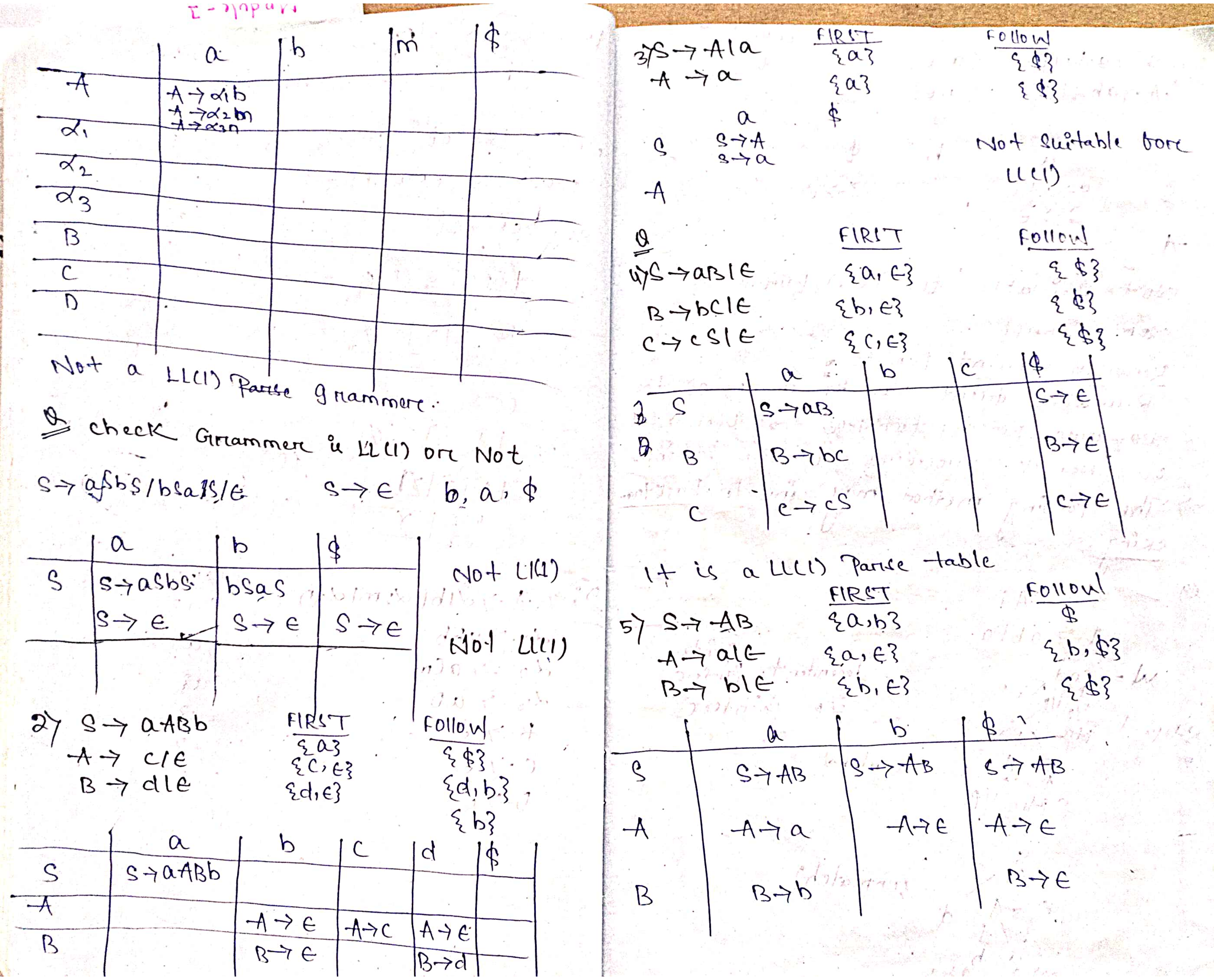
((



(()

(())

eg:→	FIRST
A→α1b|α2m|α3n	{a}
α1→aBb	{a}
α2→aCm	{a}
α3→aD	{a}
B→ε	{ε}
C→ε	{ε}
D→ε	{ε}



	a	b	m	$
A	$A \to \alpha_1 b$ $A \to \alpha_2 m$ $A \to \alpha_3 n$			
α_1				
α_2				
α_3				
B				
C				
D				

Not a LL(1) Parse grammer.

Q check Grammer is LL(1) or Not

$S \to aSbS / bSaS / \epsilon$ $S \to \epsilon$ b, a, $

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

Not LL(1)

Not LL(1)

2)

	FIRST	Follow
$S \to aABb$	{a}	{$}
$A \to c / \epsilon$	{c, ϵ}	{d, b}
$B \to d / \epsilon$	{d, ϵ}	{b}

	a	b	c	d	$
S	$S \to aABb$				
A		$A \to \epsilon$	$A \to c$	$A \to \epsilon$	
B		$B \to \epsilon$		$B \to d$	

3)

	FIRST	Follow
$S \to A / a$	{a}	{$}
$A \to a$	{a}	{$}

	a	$
S	$S \to A$ $S \to a$	
A		

Not suitable fore LL(1)

Q

4)

	FIRST	Follow
$S \to aB / \epsilon$	{a, ϵ}	{$}
$B \to bC / \epsilon$	{b, ϵ}	{$}
$C \to cS / \epsilon$	{c, ϵ}	{$}

	a	b	c	$
S	$S \to aB$			$S \to \epsilon$
B	$B \to bc$			$B \to \epsilon$
C	$c \to cS$			$c \to \epsilon$

It is a LL(1) Parse table

5)

	FIRST	Followl
$S \to AB$	{a, b}	$
$A \to a / \epsilon$	{a, ϵ}	{b, $}
$B \to b / \epsilon$	{b, ϵ}	{$}

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

		FIRST	Follow.
b)	$S \to aAa \mid \epsilon$	$\{a, \epsilon\}$	$\{\$, a\}$
	$A \to abS \mid \epsilon$	$\{a, \epsilon\}$	$\{a\}$

	a	b	$
S	$S \to aAa$ $S \to \epsilon$		$S \to \epsilon$
A			

Not suitable for LL(1) parsing

~~Recursi~~

Recursive Decent Parser:

Recursive decent parsing is one of the top-down parsing technique that uses a set of recursive procedure to scan its input.

→ The parsing method may include Backtracking.

eg:- $S \to cAd$

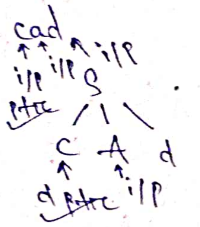
$A \to ab \mid a$

W = cad

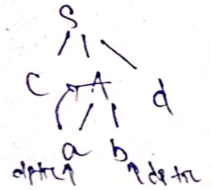
decendent pointer ~~parser~~.

i/p pointer

Step-1

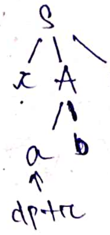


(match)

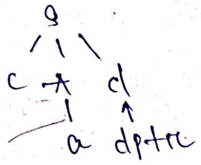


(unmatch)

Backtracking



(match)



(match)

(cad)

Q $A \to abC \mid aBd \mid aAD$

$B \to bB \mid \epsilon$

$C \to d \mid \epsilon$

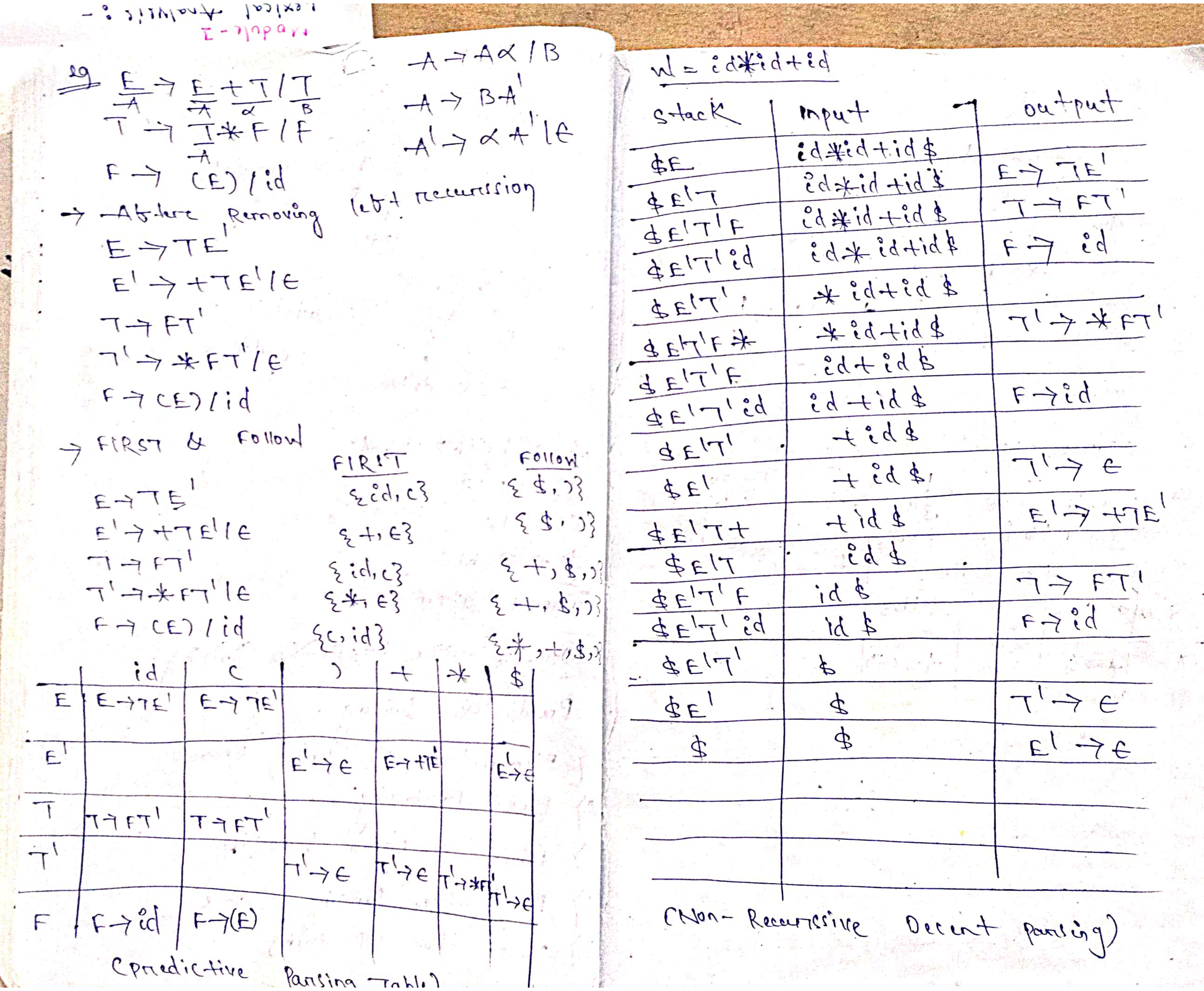
$D \to a \mid b \mid \epsilon$

input given = aaba

Non-recursive Predictive Parsing:-

Predictive Parsing

1) Elimination of Left Recursion
2) Left Factoring
3) First & Follow function
4) Predictive Parsing Table
5) Parse the i/p String



eg $\underset{A}{\underline{E}} \rightarrow \underset{A}{\underline{E}}\underset{\alpha}{\underline{+T}} / \underset{B}{\underline{T}}$

$T \rightarrow \underset{A}{\underline{T}} * F / F$

$F \rightarrow (E) / id$

$A \rightarrow A\alpha / B$

$A \rightarrow BA'$

$A' \rightarrow \alpha A' / \epsilon$

→ After Removing left recurrsion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

→ FIRST & Follow

	FIRST	Follow
$E \rightarrow TE'$	{id, (}	{ \$,)}
$E' \rightarrow +TE' / \epsilon$	{+, ε}	{ \$,)}
$T \rightarrow FT'$	{id, (}	{+, \$,)}
$T' \rightarrow *FT' / \epsilon$	{*, ε}	{+, \$,)}
$F \rightarrow (E) / id$	{(, id}	{*, +, \$,)}

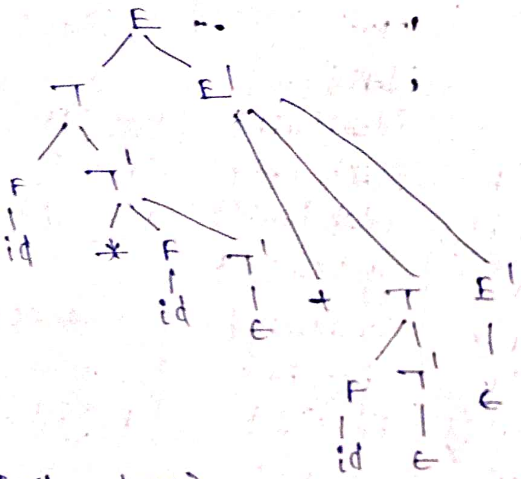
	id	(	)	+	*	\$
E	$E \rightarrow TE'$	$E \rightarrow TE'$				
E'			$E' \rightarrow \epsilon$	$E \rightarrow +TE'$		$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	$T \rightarrow FT'$				
T'			$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \epsilon$
F	$F \rightarrow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

(Predictive Parsing Table)

w = id*id+id

stack	input	output
\$E	id*id+id\$	
\$E'T	id*id+id\$	$E \rightarrow TE'$
\$E'T'F	id*id+id\$	$T \rightarrow FT'$
\$E'T'id	id*id+id\$	$F \rightarrow id$
\$E'T'	*id+id\$	
\$E'T'F*	*id+id\$	$T' \rightarrow *FT'$
\$E'T'F	id+id\$	
\$E'T'id	id+id\$	$F \rightarrow id$
\$E'T'	+id\$	
\$E'	+id\$	$T' \rightarrow \epsilon$
\$E'T+	+id\$	$E' \rightarrow +TE'$
\$E'T	id\$	
\$E'T'F	id\$	$T \rightarrow FT'$
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

(Non-Recurrssive Decent parsing)



(Parse tree)

eg S → AbS | e | ε
A → a | cAd

	FIRST	Follow
S → AbS | e | ε	{a, c, e, ε}	{b}
A → a | cAd	{a, c}	{b, d}

	a	b	c	d	e	$
S	S→AbS		S→AbS		S→e	S→ε
A	A→a		A→cAD			

eg- check LL(1) Parser for the following Grammer

S → aABb
A → c / ε
B → d | ε

Ans

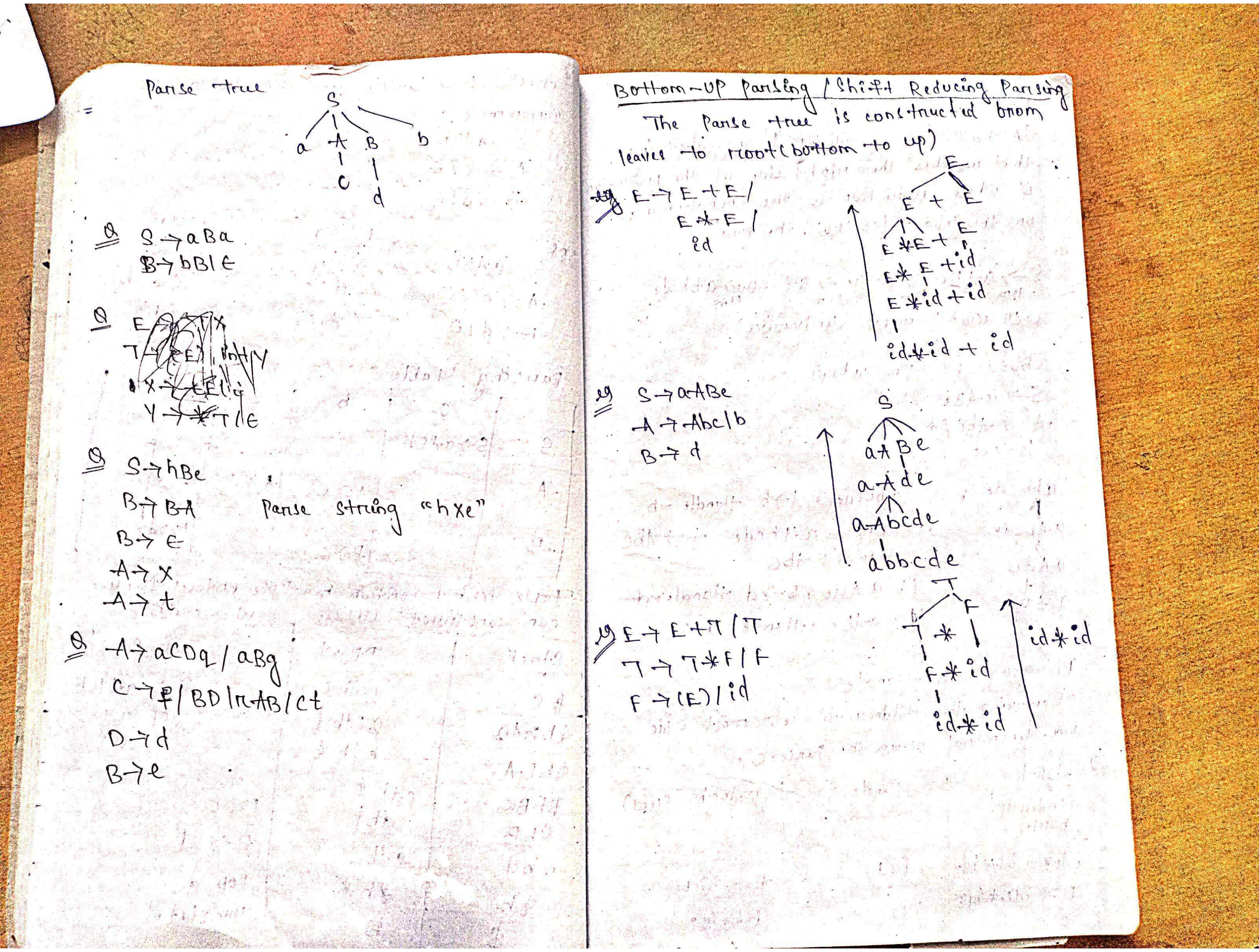
	FIRST	Follow
S → aABb	{a}	{$}
A → c / ε	{c, ε}	{d, b}
B → d | ε	{d, ε}	{b}

Parsing Table

	a	b	c	d	$
S	S→aABb				
A		A→ε	A→c	A→ε	
B		B→ε		B→d	

Table doesn't contain multiple entries so, we can construct LL(1) w = acdb

Stack	input	O/P
$ S	acdb $	S → aABb
$bBAa	acdb $	pop a
$bBA	cdb $	
$bBc	cdb $	A → c
$bB	db $	pop c
$bd	db $	B → d
$b	b $	pop d
$	$	completed



Parse tree

S
a
A
B
b
c
d

Q S → aBa
B → bB | ε

Q ~~E → T X~~
~~T → (E) | int | Y~~
~~X → + E | ε~~
~~Y → * T | ε~~

Q S → hBe
B → BA Parse string "hxe"
B → ε
A → x
A → t

Q A → aCDq / aBg
C → f / BD | rAB / ct
D → d
B → e

Bottom-Up Parsing / Shift Reducing Parsing

The Parse tree is constructed from leaves to root (bottom to up)

eg E → E + E /
E * E /
id

E
E + E
E * E + E
E * E + id
E * id + id
id * id + id

eg S → aABe
A → Abc | b
B → d

S
aABe
aAde
aAbcde
abbcde

eg E → E + T / T
T → T * F / F
F → (E) / id

T
T * F
F * id
id * id
id * id

Handles

A handles of a string is a substring that matches the right side of the production & whose reduction to the non terminal on the left side of the prod^n

eg

$S \xrightarrow[rm]{} aABe \xrightarrow[rm]{} aAde \xrightarrow[rm]{} aAbcde \xrightarrow[rm]{} abbcde$

(All right most derivative)

~~abbcde : Y = abbcde~~

$S \to aABe$
$A \to Abc / b$
$B \to d$

abbcde : Y = abbcde , A→b Handle = b
aAbcde   Y = RHS = aAbcde A→Abc
    Handle : Abc
aAde   Y : aAde , B→d Handle = d
aABe   Y = aABe , Handle = aABe
S

Pruning the Handle
Removing the children of left most side non-terminal from the parse tree.

eg

Right Sentential Form	Handle	Reducing Prod^n
$id_1 + id_2 * id_3$	id1	E → id 1
E + id2 * id3	id2	E → id2
E + E * id3	id3	E → id3
E + E * E	E + E	E → E + E
E * E	E * E	E → E * E
$\textcircled{E}$		

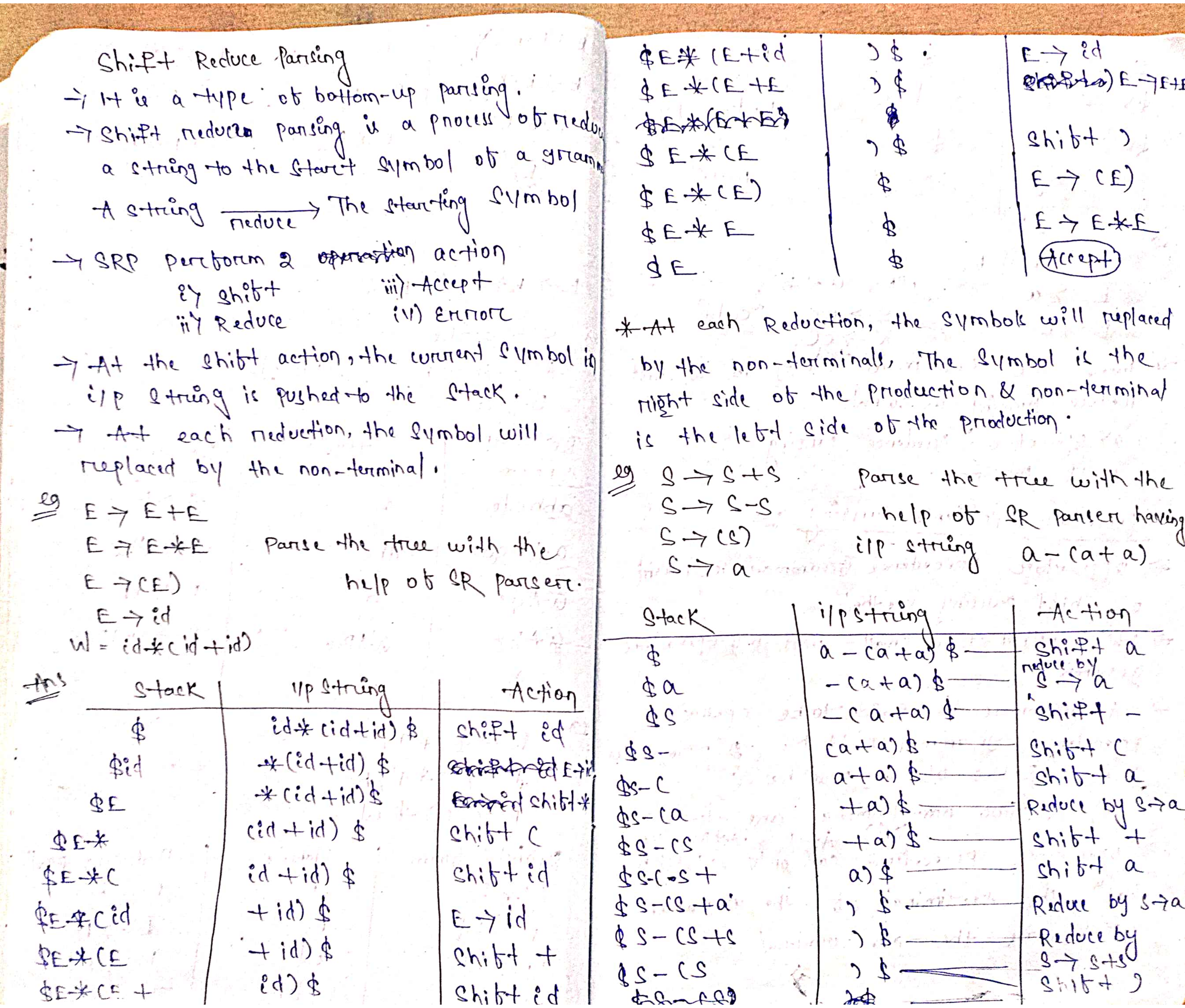
E → E + E / E * E / id

eg $S \to aABe$
$A \to Abc / b$ abbcde
$B \to d$

Right sen. Form	Handle	Reducing Prod^n
a$\underline{b}$bcde	b	A → b
a$\underline{Abc}$de	Abc	A → Abc
aAde	d	B → d
aABe	aABe	S → aABe
$\textcircled{S}$		

eg $E \to E + T / T$
$T \to T * F / F$
$F \to (E) * id$

Right s. form	Handle	Reducing prod^n
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F



Shift Reduce Parsing

→ It is a type of bottom-up parsing.

→ Shift ~~reduce~~ reduce parsing is a process of reducing a string to the start symbol of a grammar.

A string $\xrightarrow[\text{reduce}]{}$ The starting symbol

→ SRP perform 2 ~~operation~~ action

i) shift
ii) Reduce
iii) Accept
(iv) Error

→ At the shift action, the current symbol in i/p string is pushed to the stack.

→ ~~At~~ each reduction, the symbol will replaced by the non-terminal.

eg

E → E+E
E → E*E
E → (E)
E → id

Parse the tree with the help of SR parser.

W = id*(id+id)

Ans

Stack	I/p String	Action
$	id*(id+id) $	shift id
$id	*(id+id) $	~~Shift red~~ E→id
$E	*(id+id) $	~~Reduced~~ shift *
$E*	(id+id) $	shift (
$E*(	id+id) $	shift id
$E*(id	+id) $	E → id
$E*(E	+id) $	shift +
$E*(E+	id) $	Shift id
$E*(E+id	) $	E → id
$E*(E+E	) $	~~Shift~~ E→E+E
~~$E*(E+E)~~	~~$~~	
$E*(E	) $	shift)
$E*(E)	$	E → (E)
$E*E	$	E → E*E
$E	$	Accept

* At each Reduction, the symbols will replaced by the non-terminals, The symbol is the right side of the production & non-terminal is the left side of the production.

eg

S → S+S
S → S-S
S → (S)
S → a

Parse the tree with the help of SR parser having i/p string a-(a+a)

Stack	i/p string	Action
$	a-(a+a) $	Shift a
$a	-(a+a) $	reduce by S → a
$S	-(a+a) $	shift -
$S-	(a+a) $	Shift (
$S-(	a+a) $	Shift a
$S-(a	+a) $	Reduce by S→a
$S-(S	+a) $	Shift +
$S-(S+	a) $	Shift a
$S-(S+a	) $	Reduce by S→a
$S-(S+S	) $	Reduce by S→S+S
$S-(S	) $	Shift)
~~$S-(S)~~	~~$~~	

$S-(S)	$	Reduce by S→(S)
$S-S	$	Reduce by S→ S-S
$S	$	accept

eg E→E+E
E→a
E→b
i/p = a+b

eg S→CC "ccdd"
C→cC
C→d

→ There are two main categories to shift reduce parsing as follows
a) operator - predence parsing
b) LR parser

operator Predence parsing :→

→ operator precedence grammar is a kind of shift parsing method.

→ It is applied to a small class of operator grammer.

→ A grammer is said to be operator precedence grammer if it has 2 properties
i) No. R.H.S of any prodn has ε
ii) No two non-terminal are adjacent. ~~(AB)~~, A+B, A*B, A/B

→ Operator precedence can only establish betn the terminal of the grammer & ignore the non-terminal

→ There are 3 operator predence relation
i) a ·> b - Terminal a has the higher precedence than terminal b
ii) a <· b - Terminal a has the lower precedence than terminal b
iii) a ≐ b - Terminal a & b both have same precedence.

eg

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	X
)	·>	·>	X	·>	X	·>
id	·>	·>	X <·	·>	X	·>
$	<·	<·		X	<·	X

eg S → SAS / id
A → aSa / a

S → SaSaS / SaS / id
A → aSa / a

Q What do you mean by OPG with the help of following grammer parse the i/p = id+id*id
T → T+T | T*T | id

Steps to solve

1) check OPG or NOT
2) operator precedence Relation table
3) parse the given string

T → T+T / T * T / id

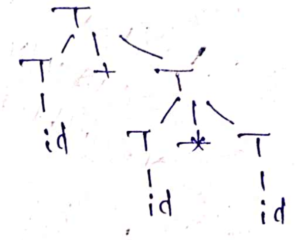
	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>	—	·>
$	<·	<·	<·	A

Basics
(identifier)
id, a, b, c = High
$ = low
+ ·> +
* ·> *
id ≠ id
$ A $

Parse the Given Tree

Stack	Relation	Input	comment
$	~~id+id*id$~~		
$	<·	id+id*id$	shift id
$id	·>	+id*id $	~~$~~ Reduce T→id
$T	<·	+ id*id $	shift +
$T+	<·	id*id $	Shift id
$T+id	·>	* id $	Reduce T→id
$ T+T	<·	*id $	shift *
$T+T*	<·	id $	shift id
$T+T* id	·>	$	Reduce T→id
$T+T*T	·>	$	Reduce T→T*T
$T+T	·>	$	Reduce T→T+T
$ T	A	$	

Parse Tree



eg E → E+T/T
T → T* V/V
V → a/b/c/d
a+b*c*d

Operator Precedence Relation Table

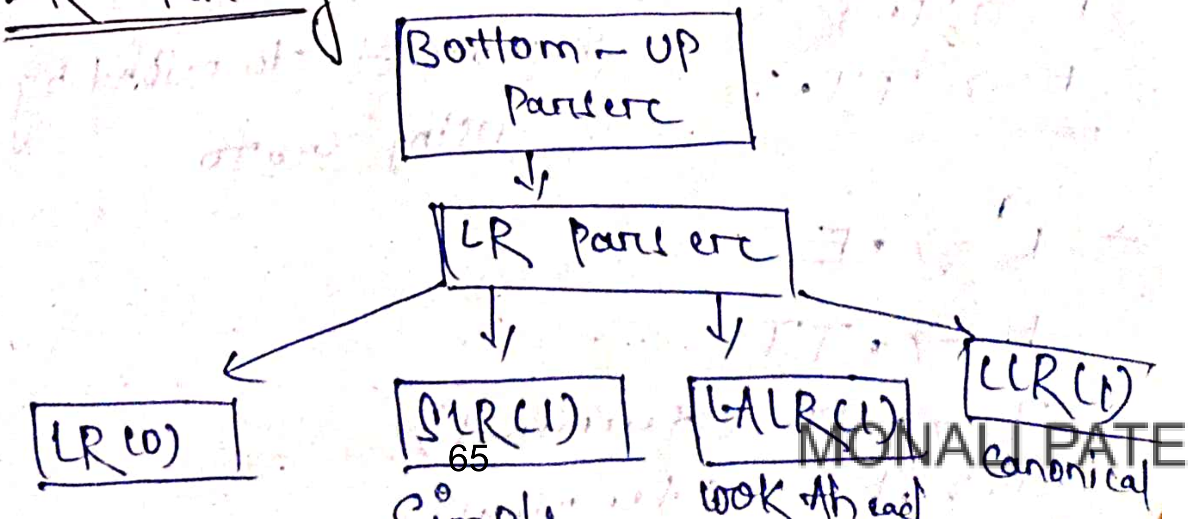
	+	*	a	b	c	d	$
+	·>	<·	<	<	<	<	>
*	>	>	<	<	<	<	>
a	>	>	–	–	–	–	>
b	>	>	–	–	–	–	>
c	>	>	–	–	–	–	>
d	>	>	–	–	–	–	>
$	<·	<	<·	<	<	<	A

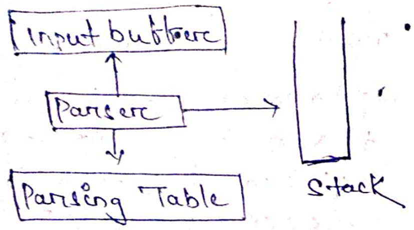
SRIC

Stack	R	input	comment
$	<	a+b*c*d$	shift a
$a	>	+b*c*d$	Reduce V→a
$V	<	+b*c*d$	shift +
$V+	<	b*c*d$	shift b
$V+b	>	*c*d $	Reduce V→b
$V+V	<	*c*d$	shift *
$V+V*	<	c*d$	shift c
$V+V*c	>	*d $	Reduce V→c
$V+V*V	>	*d $	Reduce T→V
$V+V*T	>	*d $	Reduce E→T
$V+V*E	>	*d $	No Prodn

* No production found for reduction operation
* Since parsing process failed to complete the given input can not be parsed by the given grammar OPP. method

2) LR Parsing





→ LR(0) items – LR(0), SLR(1) E → ·aA

⇒ LR(1) items – LALR(1), CLR(1) E → ·aA, ~~$~~ a|

① LR(0) items – Add agumented prod^n^

eg E → TT a) LR(0) Parsere

T → aT/b

– Add Agumented Prod^n^

– Agumented Grammere

E' → E

E → TT (①)

T → aT ② / b ③

construct LR(0) item

E' → ·E immediate right to . Non-terminal

~~E → ·TT~~

~~T → ·aT/b~~

E → T·T

E → TT·

* move it to right by using Goto

→ E' → ·E

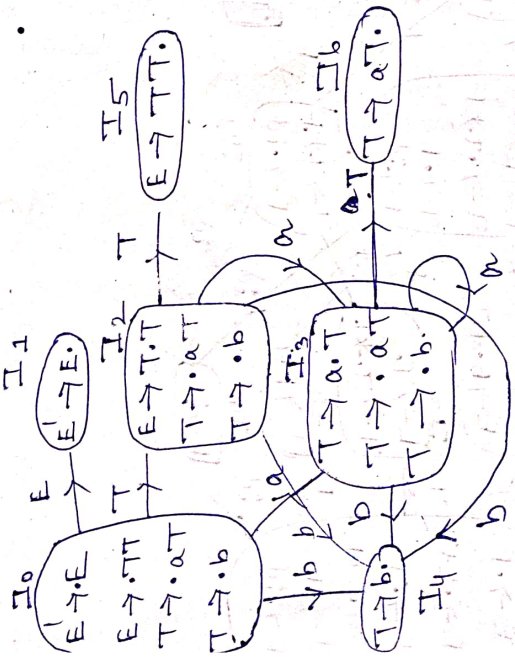
E → ·TT

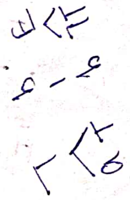
T → ·aT (terminal)

T → ·b (terminal)

	Action → terminal			Goto → non terminal	
	a	b	$	E	T
0	S3	S4		1	
1			accept		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

Shift
Reduce





b) SLR(1) Parser

$E' \to \cdot E$

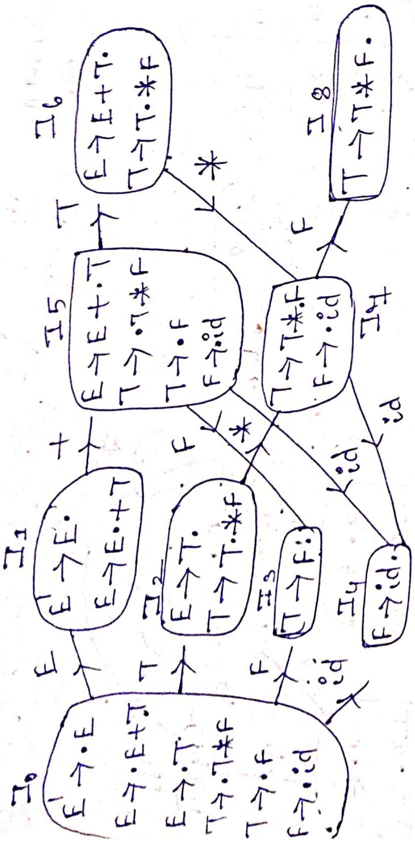
$E \to \cdot E+T$ ①

$E \to \cdot T$ ②

$T \to \cdot T*F$ ③

$T \to \cdot F$ ④

$F \to \cdot id$ ⑤



	Action				Goto		
	id	+	*	$	E	T	F
0	S4				1	2	3
1		S5		accept			
2		r2	S7	r2			
3		r4	r9	r4			
4		r5	r5	r5			
5	S4					6	3
6		r1 ~~S5~~	S7	r1			
7	S4						8
8		r3	r3	r3			

Follow

$E \to \{+, \$\}$

$T \to \{+, \$, *\}$

$F \to \{*, +, \$\}$

$E \to T\cdot$

$E \to E+T\cdot$ ⑥ ①

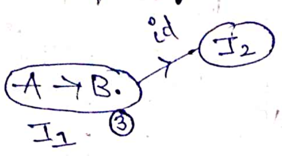
I_8 $T \to T*F\cdot$ ③

LR(0) conflict

→ There are both shift & reduce in the same item (shift - reduce - S/R)

→ There are two reduce actions in the same items ("reduce-reduce", R/R)

LR(0)



① SR conflict

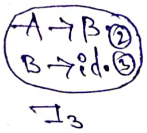
Action

	id	+	*
1	S2/r3	r3	r3

② RR

Action

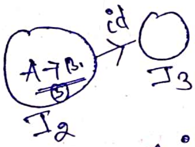
	t1	t2	t3	t4
3	r2/r3	r2/r3	r2/r3	r2/r3



SLR(1)

① S-R ② R-R

①



Action

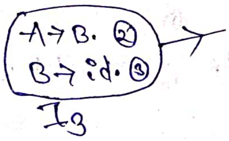
	id	+	*
2	S3/r3		

S-R conflict

follow A = { id }

	t1	t2	t3
3	r2	r2/r3	r3

R-R conflict



follow of A = { . t1, t2 }

follow of B = { t2, t3 }

2) LR(1) item

↳ LR(0) item + lookahead

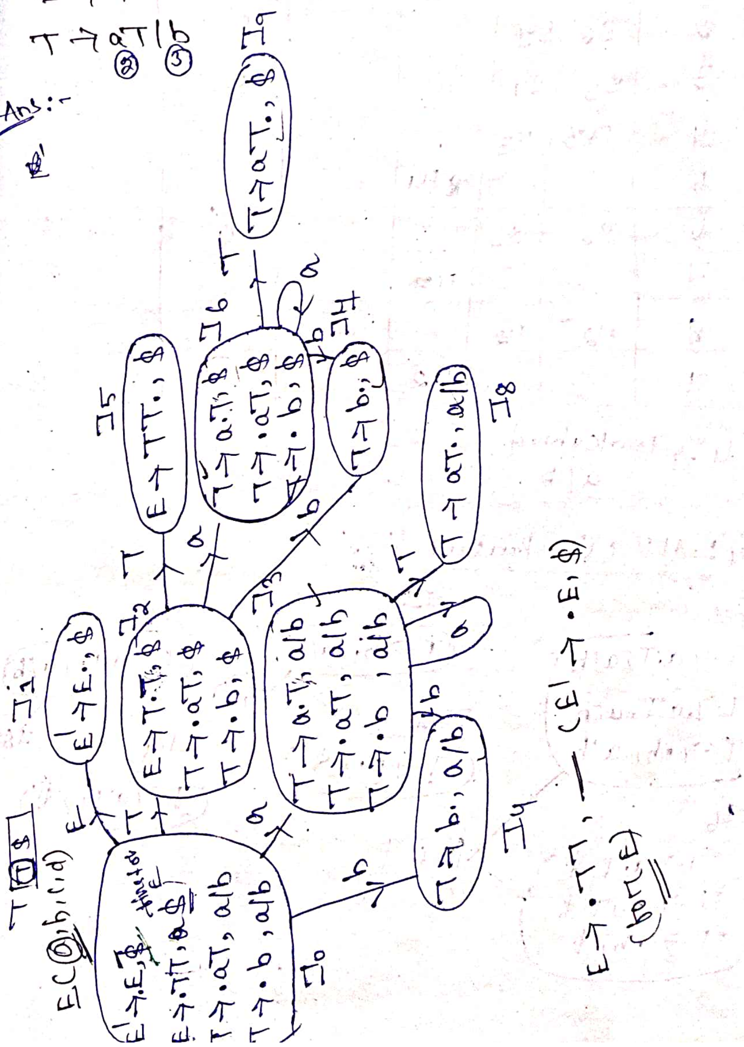
$E' \to E$

$E \to TT$ ①

$T \to aT \mid b$ ② ③

a) CLR(1) Parser

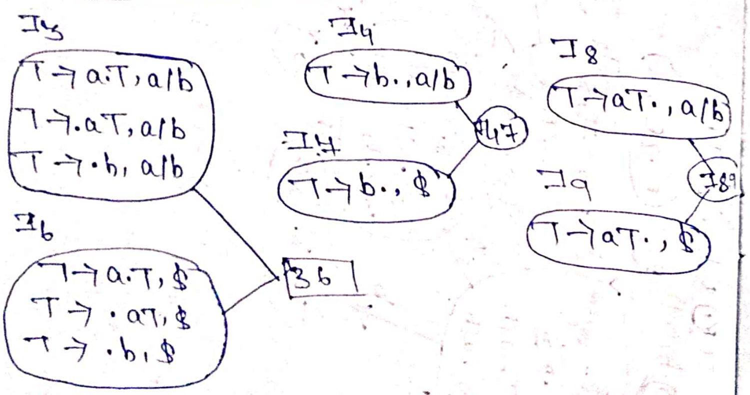
Ans:-



	Action			Go-to	
	a	b	$	E	T
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

4 → lookahead a/b

b) LALR(1) Parsere



	Action			Goto	
	a	b	$	E	T
0	S_{36}	S_{47}		1	2
1			Accept		[illegible]
2	S_{36}	S_{47}			85
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
~~36~~	~~S_{36}~~	~~S_{47}~~			~~89~~
~~47~~			~~r_3~~		
89	r_2	r_2	r_2		
~~89~~			~~r_2~~		

S → State

r → Reduce (prodⁿ no. in respective grammer)

LR(0) → SLR(1) → LALR(1) CLR(1)

eg S → AA

A → aA

A → b

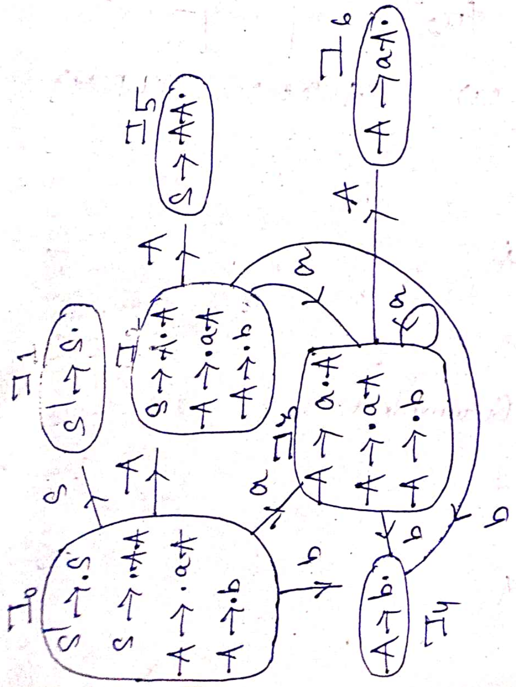
Agumented Grammere

S' → ·S

S → ·AA

A → ·aA

A → ·b



eg $E \to E+T$

$E \to T$

$T \to (E)$

$T \to id$

LALR

eg $S \to CC$

$C \to cC$

$C \to d$

~~Handling of Ambiguous~~

Handling of Ambiguous Grammer in LR Parsing

The ambiguous grammer LR parsing table contains

→ Shift / Reduce conflict

→ Reduce / Reduce conflict & it can be resolved by using following method

i) The S-R conflict in the parsing table is by favouring the shift action over reduced action (Give priority to shift action).

ii) The R-R conflict in the parsing table is resolved by favouring first reduce action over Second reduce action.

eg Resolve the conflict from the following expression grammer

$E \to E+E$

$E \to E*E$

Agumented Grammere

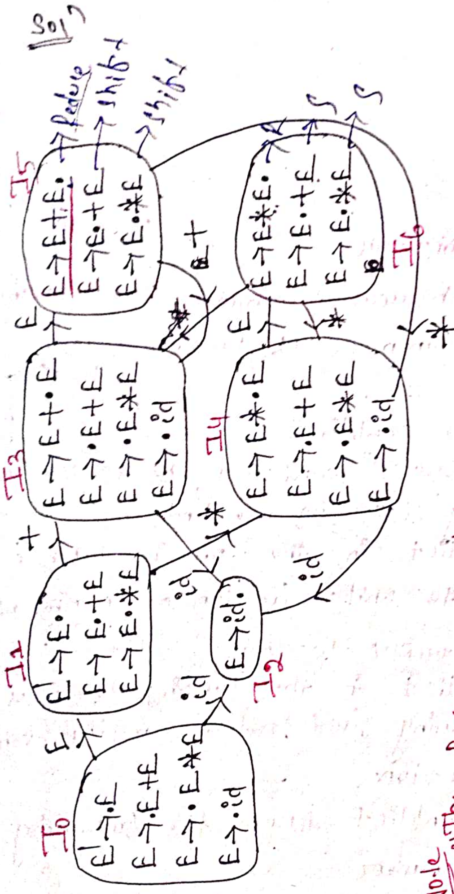
$E' \to E$

$E \to E+E$ ①

$E \to E*E$ ②

$E \to id$ ③

Sol^n



Note
i) The Parser generated tool Yacc also resolves the conflict of the above manner.

ii) If the grammar is expression grammar than the conflict S/R & R/R are Resolved based on the Precedence of the operation.

construction of SLR Parsing table

	Action				Go-to
	id	+	*	$	E
0	S_2				1
1		S_3	S_4		
2		r_3	r_3		
3	S_2				5
4	S_2				6
5		r_1 S_3/r_1	S_6 S_6/r_1		
6		r_2 S_3/r_2	r_2 S_4/r_2		

conflict

① Resolving conflict of I5 state on ~~*~~ +
id + id + id

② Resolving conflict of I5 on state *
id + id * id

③ Resolving conflict of I6 on state *
id * id * id

④ Resolving conflict of I6 on state +
id * id + id

DAG(Direct Acyclic Graph)

→ The directed acyclic graph is used to represent the structure of basic blo to visualize the flow of values betn ba blocks & to provide optimization techniques in the basic blocks.

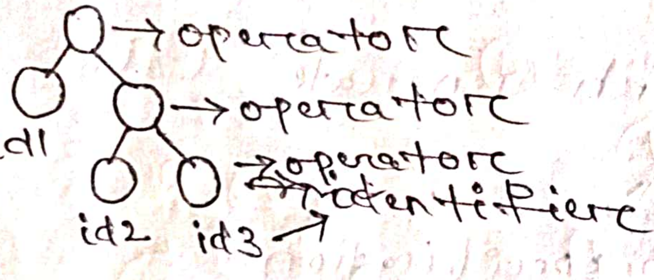
→ DAG is a 3 address code i.e generated as the result of an intermediate code generation (ICG)

Chrracteristics of DAG

i) Leaves have a unique identifierr

ii) Interior nodes are labelled with operator symbol

iii) nodes are given a string of identifiers use as labels for string computed values.

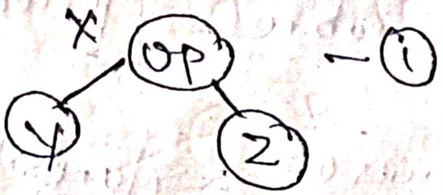


case-i x = y op z

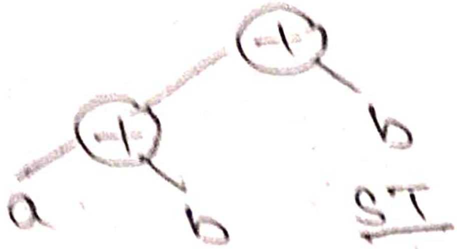
case-ii x = op y

case-iii

x=y ~~xorxy~~

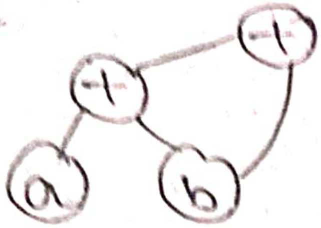


eg $(a+b)+b$



ST (Syntax tree)

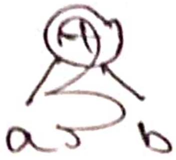
DAG



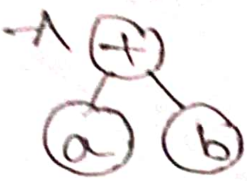
eg $A = a+b$

$B = A+C$

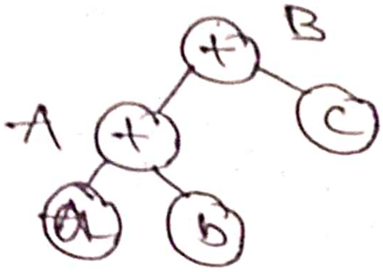
$Z = A*B$



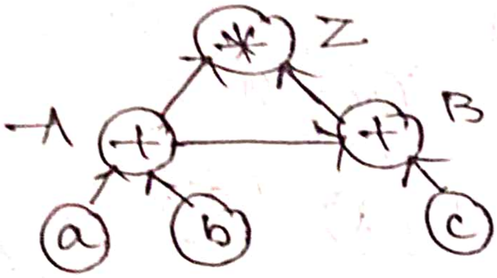
①



②



③



DAG

eg $P = a+b$

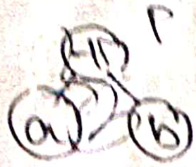
$Q = a-b$

$R = P*Q$

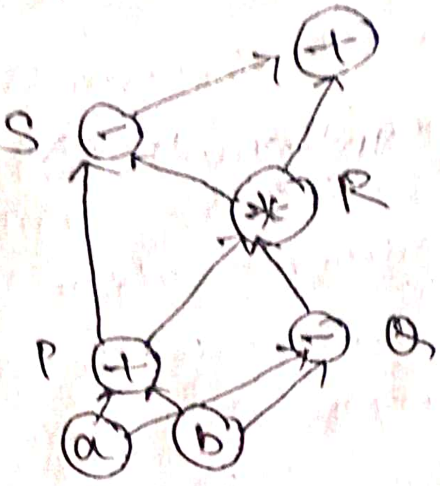
$S = P-R$

$T = S+R$

①



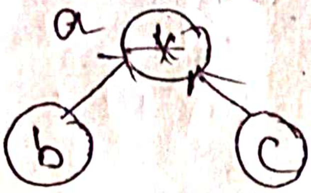
②



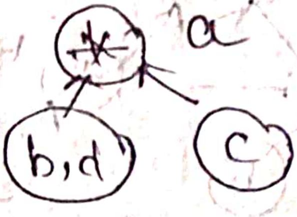
eg

a = b*c
d = b
e = d*c
b = e
f = b+c
g = f+d

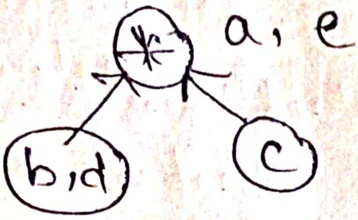
①



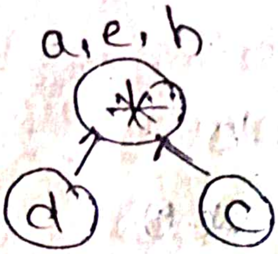
②



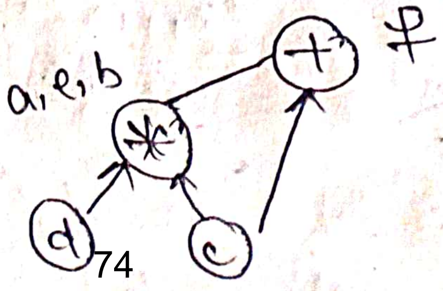
③



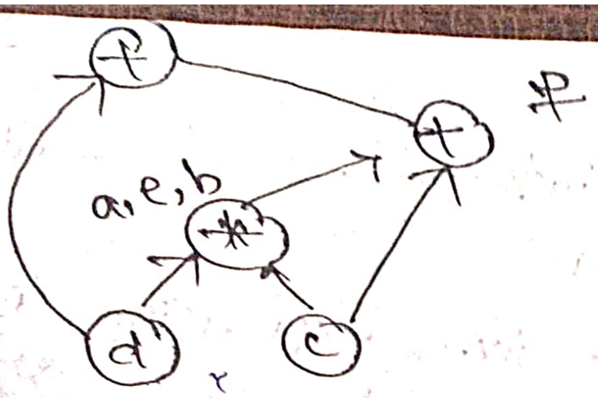
④



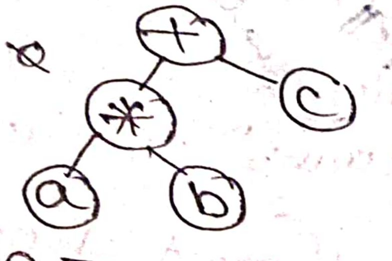
⑤



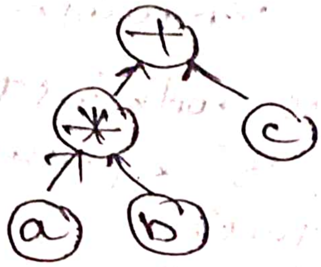
⑥



eg a*b+c

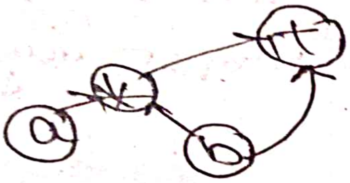


ST

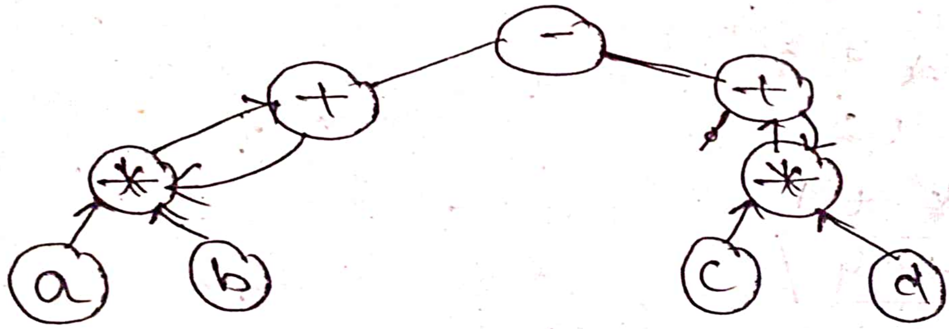


CDAG

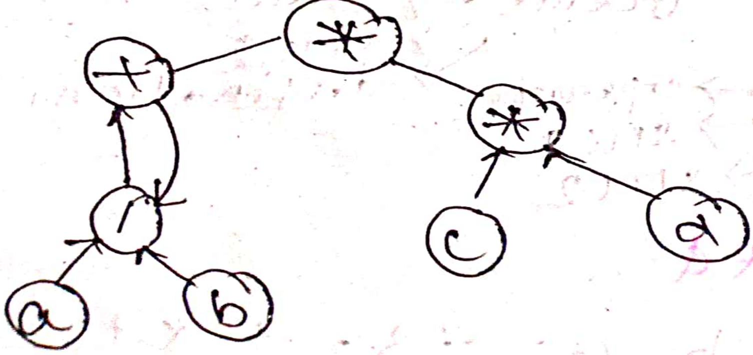
eg a*b+b



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



(a/b) + (a/b) * (c*d)



Three address code

* It is an intermediate code. It is used by th optimizing compliers.

* In 3 address code, the given expression is broken down into several separate instructions. i.e translate into Assembly language.

→ Each 3 address code instruction has at most 3 operand (comb^n of assignm a binary operator)

→ In 3 address code, there is at most 1 operator on the right side of an instruction

eg $x + y * z$ t1 & t2 are compiler generated temporary name

$t1 = y * z$

$t2 = x + t1$

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

$t1 = b - c$

$t2 = a * t1$

$t3 = a + t2$

$t4 = d * t1$

$\boxed{t5 = t3 + t4}$

Types

① Quadruples → 4 fields → operator
→ source 1
→ src2
→ destination

② Triples → 3 fields → operator
→ src1
→ src2

① Quadruples

Q:→ $a = -b * (c + d)$

TAC

$t1 = -b$ ① $t3 = t1 * t2$ ③

$t2 = c + d$ ② $\boxed{a = t3}$

	operator	Source 1	Source 2	Destination
(0)	−	b	—	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	=	t3		a

② Triples

eg $a = -b * c + d$

TAC

$t1 = -b$
$t2 = c + d$
$t3 = t1 * t2$
$a = t3$

	operator	Source 1	Source 2
(0)	−	b	—
(1)	+	c	d
(2)	*	0	1
(3)	=	2	

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

$t1 = -c$
$t2 = b * t1$
$t3 = -c$
$t4 = b * t3$
$t5 = t2 + t4$
$a = t5$

1) <u>Quadrouples</u>

	operator	src1	src2	destination
(0)	-	c	—	t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t3	t4	t5
(5)	=	t5		a

2) <u>Triples</u>

	operator	src1	src2
(0)	-	c	
(1)	*	b	(0)
2)	-	c	
3)	*	(1)	(2)

Types & Declaration

The appl^n of types can be grouped undere checking & translation

1) Type checking

uses logical rules to reason about the behaviorr of a program at run time, specially it ensurees that the type of the operands match the type expected by an operatorr

eg int * float → (Not possible)

2) Translation Appl^n from the type of a name, a compilerr can determine the storage that will needed forr that name at run time

1. Type Expression

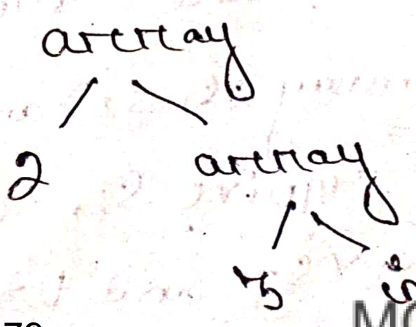
Types have structure, which we shall represent using type expressions. → int | boolo float

→ A type expression is eithere a basic type or is formed by applying an operatorr called a type constructorr to a type expression.

eg int[2][3]

array (2, array (3), integerr))

Operatorr takes 2 parameterees, a no. & a type.



→ A basic type is a type ex

basic types for a language include boolean, char, integer & void. int abc;

→ A type name is type expression

→ A type expression can be formed by applying the array type constructor to a no. & a type expression. a[10]; or int a[10];

→ A record is a dc with name fields.

→ A type expression can be formed by applying the record type constructor to the field name & their types.

→ A type expression can be formed by applying the record type constructor to the field name & their types.

→ A type expression can be formed by using the type constructor $s \rightarrow t$

2. Type Equivalance

Two expressions are structurally equivalent if there are two expression of same basic type or formed by ~~using~~ applying same constructor.

Structural Equivalance Algo.

```
if( s and t are same basic types) then return true
elseif( s = array (s1, s2) and t = array ( t1, t2) then
    return sequiv( s1, t1) & sequiv( s2, t2))
elseif( s = s1 x s2 and t = t1 x t2) then return
    sequiv( s1, t1) and sequiv ( s2, t2)
```

SDT(Syntax Directed Translation)

→ SDT refers to a method of compiler implementation where Source language translation is completely driven by the parser

→ In SDT, along with the grammer we associate some informal notations & these notations are called as Semantic rule.

* Grammer + Semantic rule = SDT

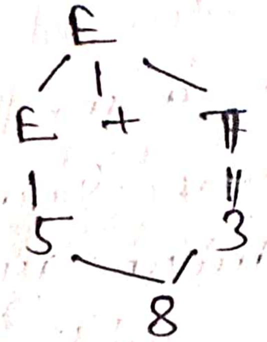
→ In SDT, every nonterminal can get 1 or more than 1 attribute or Sometime 0 attribute depending on the type of the attribute. (value of these attribute is evaluated by the Semantic rules associated with the prodⁿ rule).

→ In the Semantic rule, attribute is VAL & an attribute may hold anything like a string, a no., a memory location & a complex record

→ In SDT, whenever a construct encounter in the PL then it is translated according to the Semantic rules define in that particular PL.

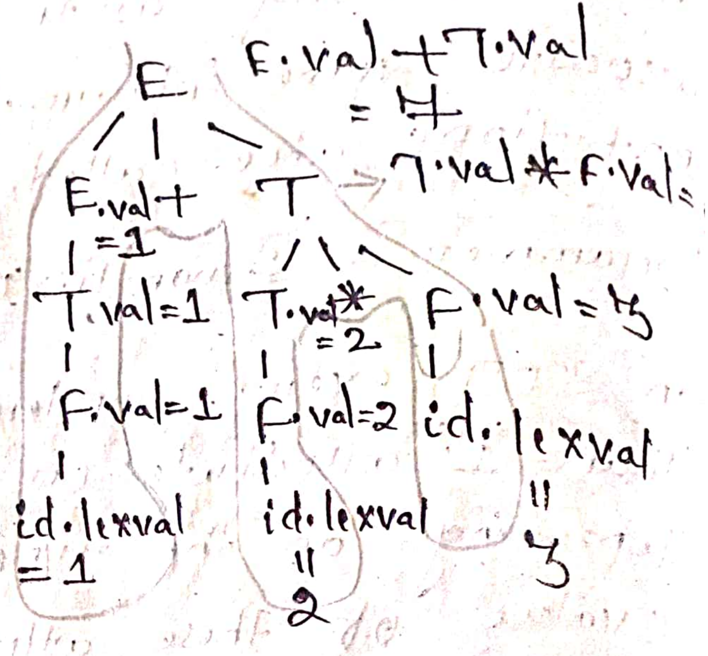
Production	Semantic Rules
E → E+T	{E.val = E.val + T.val}
E → T	{E.val = T.val}
T → T * F	{T.val = T.val * F.val}
T → F	{T.val = F.val}
~~T→(F)~~	~~T.val~~
F → id	{F.val = id.lexval}

E → E + T



eg 1 + 2 * 3
1 + 6 = 7

Top to bottom
left to right



eg S → S # A / A
A → A & B / B
B → id

5 # 3 & 4

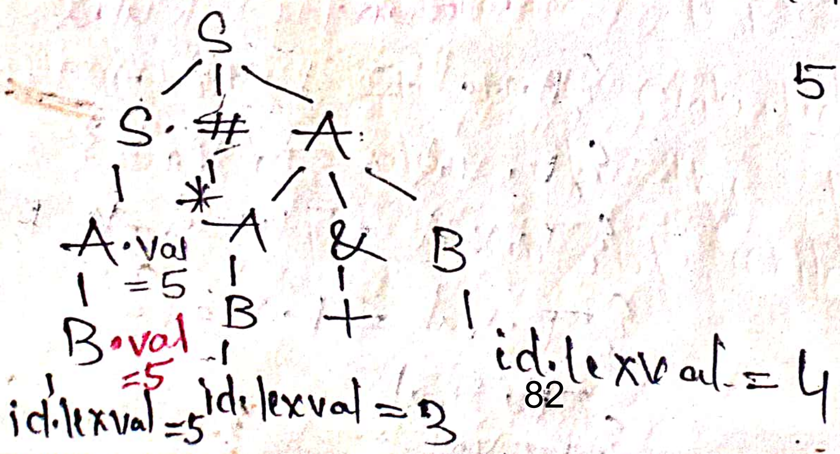
Sol

Production	Semantic Rules

S → S # A / A	S.val = S.val * A.val
	S.val = A.val

A → A & B / B	A.val = A.val + B.val
	A.val = B.val

B → id	B.val = id.lexval



~~Types of~~ SDT

→

Types of attribute

Attribute may be of two types -

→ i) Synthesized

→ ii) Inherited

i) Synthesized Attributes

→ It is an attribute of the non terminal on the LHS. of production

→ It represents info. i.e being passed up parse tree. The attribute can take value only from the children (variable in the RHS of the production)

eg A → BC (A depend on B & C

ii) Inherited attribute

→ An attribute of a Nonterminal on the RHS of the prodⁿ

→ The attribute can take value either from its parent or from siblings.

Types of SDT

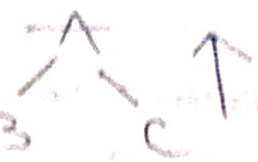
i) S- attributed SDT

→ If an SDT uses only synthesized attributes it is called S- attributed SDT

→ S- attributed SDTs are evaluated in bottom-up parsing as the value of the parent nodes depends upon the value of child node.

$A \rightarrow BCD$ { $A.S = B.S$, $A.S = C.S$,
$A.S = D.S$ }

eg $A \rightarrow BC$
$B \rightarrow C$

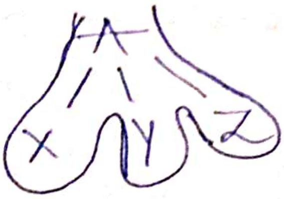


2) L-attributed SDT

→ If an SDT uses both synthesized & inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed.

→ L-attributed SDTs are evaluated by Depth-first & left-to right parsing method.

$A \rightarrow XYZ$ { ~~A.S~~ $Y.S = A.S$, $Y.S = X.S$ }



eg $A \rightarrow PQ$ & $A \rightarrow XY$

L attribute L attribute

Rule1 { $P.i = \underline{A_i} + 2$, $Q_i = \underline{P_i + A_i}$ &
$A.S = \underline{P.S} + Q.S$ }

L.attribute

✗ Rule 2 { $X_i = \underset{L}{A_i} + \underset{✗}{\underline{Y.S}}$ & $Y.i = X.S + A_i$ }

Arrray Referrences

Arrray :- An arrray is an indexed collection of data elements of the same type. (Homogeneous)

→ Indexed means that the arrray elements are numbered (starting at 0) a[0]

→ The restriction of the same type is an important one, becoz arrray are stored in consecutive memory cells.

→ Arrray can be 1D, 2D or K-dimension

1D arrray Address calculation

Address of an element of an arrray say A[I] is calculated as:

$$\boxed{B + W * (I - LB)}$$

eg :-

Address	1100	1104	1108	1112	1116	1120
Element	15	7	11	44	93	20
index	0	1	2	3	4	5

where,

B = Base address

W = Storage size of one element (in byte)

I = Sub Script of element whose address to be found

LB = Lowerr limit C

eg Give an address of an arrray B[1300...1900] as 1020 & size of each element is 2 byte in the memory. Find the address ob B[1700]

Ans B = 1020
LB = 1300
W = 2
I = 1700

Address of A[I] = B+W * (I-LB)
= 1020 + 2 * (1700-1300)
= 1020+2 * 400
= 1020+ 800
= 1820

Three address code for 1D array
int a[i]

A[I] = B+W* (I-LB)
A[I] = B+W * I - W*LB
= B- W*LB- W * I   LB = 0, by default
= B- 0 - W * I
= B+ 4 * I ← TAC (W= 4]

3 address code
T1= address A (Base address)
T2 = 4*I
T3 = T2[T1]

2D array

column index → / Row index ↓

	0	1	2	3
0	8	6	5	4
1	2	1	9	7
2	3	6	4	2

2-D array

Row-major

8	6	5	4	2	1	9	7	3	6	4	2
R0				R1				R2			

8	2	3	6	1	6	5	9	4	4	7	2
C0			C1								

Address calculation : in 2D array

Row major system

Address of A[I][J] = B+W * [N * (I-Lr)+(J-

Column major system

Address of A[I][J] = B+W * [(I-Lr]+ M*(J-

Where,

B = Base address

I = Row subscript of element whose address is to be found.

J = column subscript of element whose address is to be found

W = Storage size of one element (in byte)

Lr = lower limit of row/start row index of matrix (0)

Lc = lower limit of column

M = No. of row the given matrix

N = No. of column of the given matrix

Q Find the address of a[2][3] if array is arranged row wise & column wise. Given W=4, M=10, N=10

Rowwise = A[I][J] = B+W * [N * (I-Lr)+(J-Lc)]

A = B+4 * [10 * (2-0)+(3-0)]

= B+4 * [23]

= B+92 = 92

if no given then B=0.

column wise = A[I][J] = B+W * [(I-Lr) +M(J-Lc)]

= B+4*[(2-0) +10(3-0)]

= B * 4 * 32

= B+128 = 128

Translation of Boolean Expression

(or)

Generating 3 address code for Boolean expression (or)

control flow Translation with Boolean expression

- → short circuit or Jumping code
- → without generating code for the boolean operators
- → without evaluating the entire expressio

eg E1 or E2 E1 and E2

```
if a<b
    go to E.true
else
    go to E.false
```

E → E1 or E2 | { E1.true = E.true ;
E1.false = newlabel ;
E2.true = E.true ;
E2.false = E.false ;
E.code = E1.code || gen (E1.false) ||
E2.code }

E → E1 and E2 {E1. true = newlabel;
E1. falle = E. false;
E2 .. true = E. true;
E2. falce = E. false;
E. code = t1. code || gen(E. true:)||
E2. code }

E → not E1 {E1 .. true = E. false;
E1. false = E. true;
E. code = E1. code }

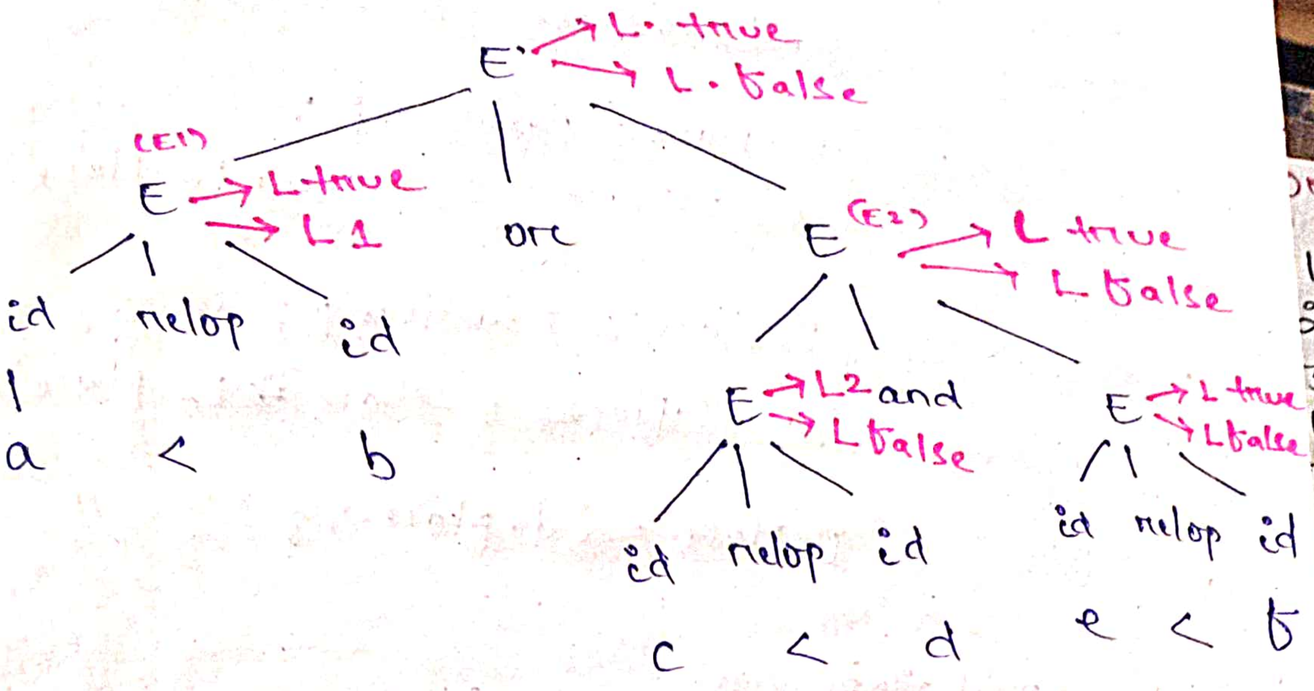
E → (E1) {E1. true = E. true;
E1. false = E. false;
E. code = E1. code }

E → id1 relop id2 E. code = gen(if id1. place relop
id2 go to E. true) ||
gen('go to E. false')}

E → true E. code = gen('go to' E. true:)

E → false E. code = gen('go to' E. false')

Q a<b or c<d and e<f



Three address code

if a<b go to L.true

go to L1

L1: c<d go to L2

go to L false

L2: if e<f go to L.true

go to L false

SDT to Produce Three address code

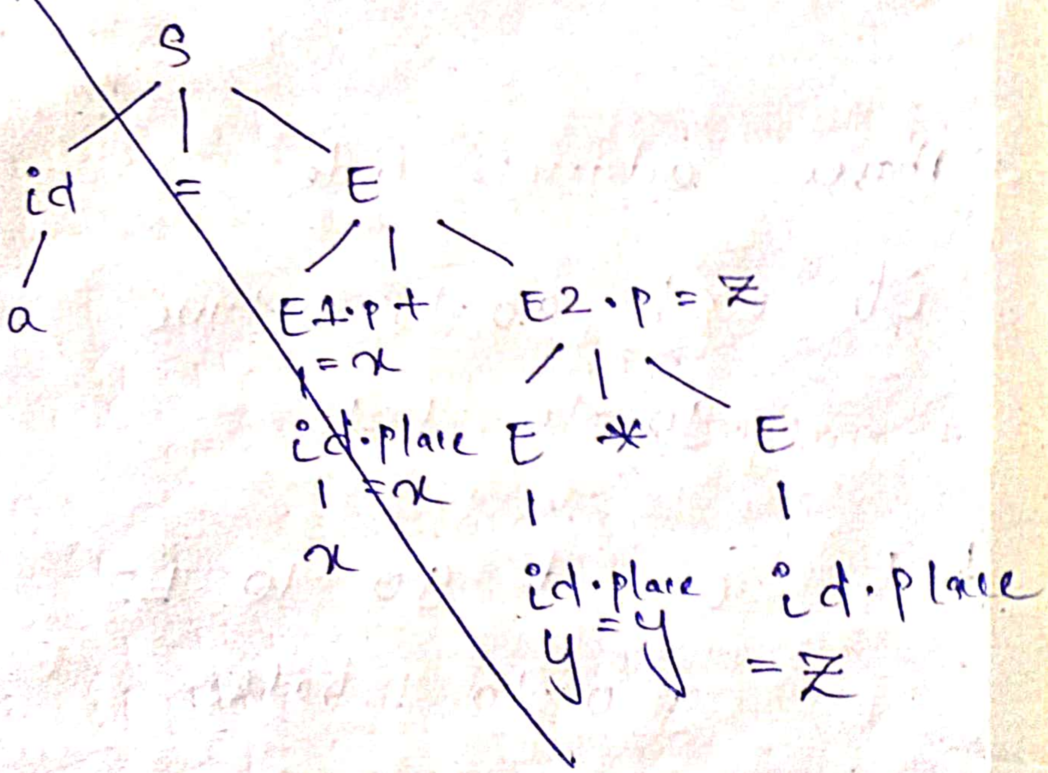
S → id = E {gen(id.place = E.place);}

E → E1 + E2 {E.place = newtemp;
gen(E.place = E1.place + E2.place);}

E → E1 * E2 {E.place = newtemp;
gen(E.place = E1.place * E2.place);}

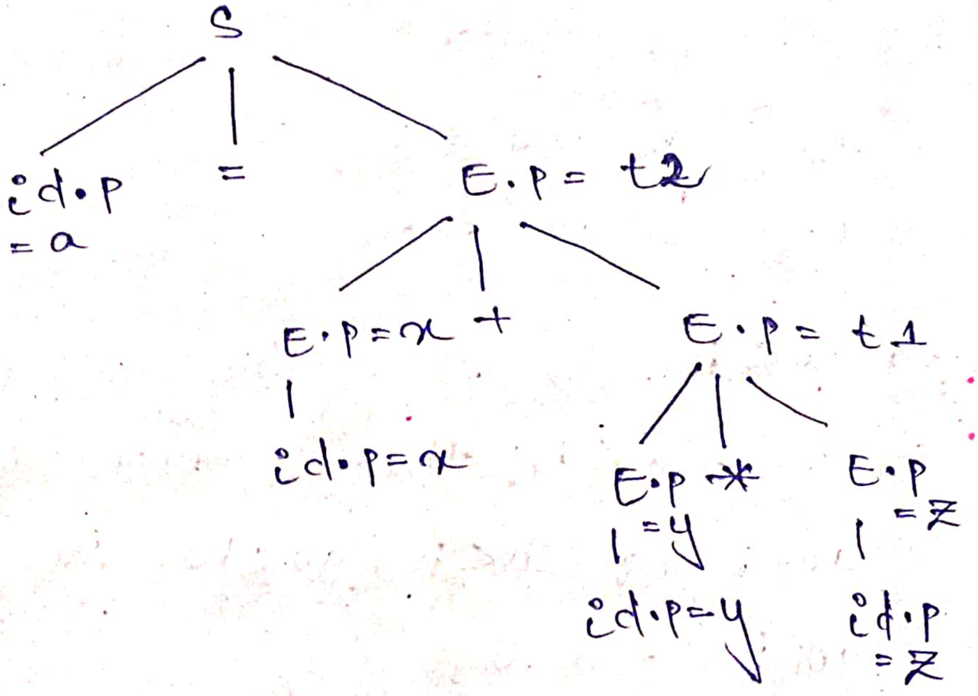
E → id {E.place = id.place;}

a = x + y * z



Output

$a = x + y * z$



output

$t1 = y * z$

$t2 = x + t1$

$a = t2$

Boolean Expression

if E then S

if E then S1 else S2

while E do S

* and, or, not   not > and > or

Translating Boolean Expression

1. Numerical representation → true (1)
   → false (0)

2. Flow of control (short-circuit

eg a or b and not c

t1 = not c

t2 = b and t1

t3 = a or t2

eg a < b

if a < b then 1 else 0

100 : if a < b go to 103

101 : t = 0

102 : go to 104

103 : t = 1

SDT using numerical representation for Boolean expression

E → E1 or E2 {E.place = newtemp;
emit (E.place = E1.place or
E2.place)}

E → E1 and E2 {E.place = newtemp;
emit(E.place = E1.place and
E2.place)}

→ not E1 {E.place = newtemp;
emit(E.place = not E1.place)}

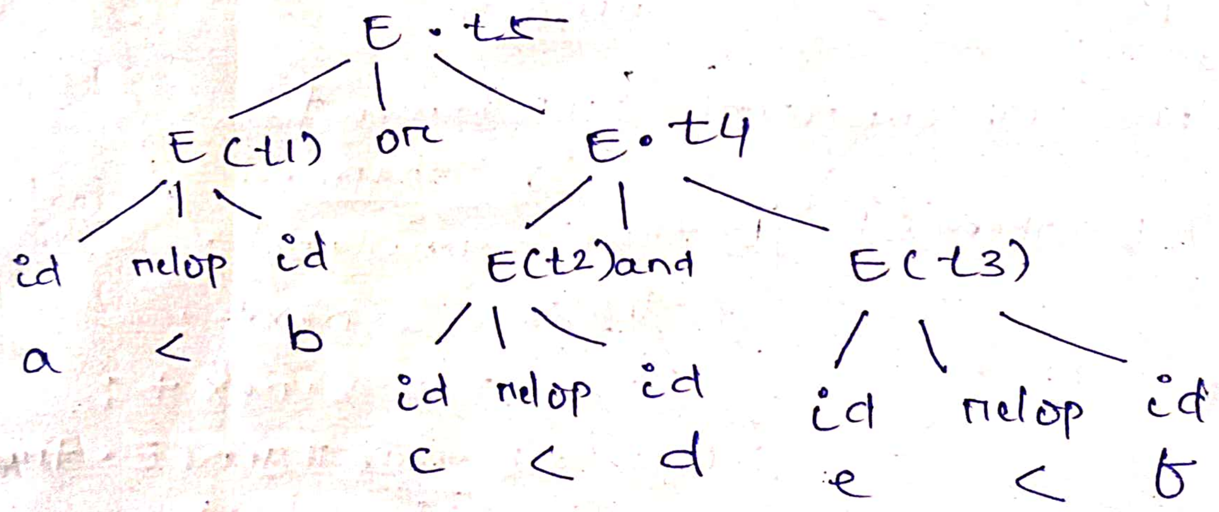
E → (E1)   E.place = E1.place

E → id1 relop id2   E.place = new temp ;
emit(if id 1.place relop
id 2.place goto (nextstate + 3);
emit(E.place = 0) ;
emit('go to' nextstate + 2);
emit (E.place = 1);

E → true   {E.place = new temp ;
emit(E.place = 1)}

E → false   {E.place = new temp;
emit(E.place = 0)}

a<b or c<d and e<f



100 : if a < b go to 103

101 : t1 = 0

102 : go to 104

103 : t1 = 1

104 : if c < d go to 107

105 : t2 = 0

106 : go to 108

107 : t2 = 1

108 : if e < f go to 111

109 : t3 = 0

110 : go to 112

111 : t3 = 1

112 : ~~t12 and t~~ 3

112 : t4 = t2 and t3

113 : t5 = t1 or t4

Flow of control statement

if - then

S → if E then S1 {E.true = newlabel ;
E.false = S.next ;
S1.next = S.next ;
S.code = E.code || gen(E..
|| S1.code }

S → if E then S1
else S2

E.true = newlabel ;
E.false = newlabel ;
S1.next = S.next

Backpatching

Leaving Labels as empty & filling them later is called backpatching

A) If - else 3 Address code

Q if (a<b) then t 1 else t=0

1) if a<b go to 4
2) t=0
3) go to 5
4) t = 1
5)

Q if (a<b) && (c<d) then t= 1 else t=0

3 address code

1) if a<b go to 4
2) t=0
3) go to 7
4) if c<d go to 6
5) go to 2 (if c<d is false)
6) t= 1
7)

Q
```
for(i=1; i≤n; i++)
{
    x = a+b*c;
}
```

3 address code

```
1) i = 1
2) if (i≤n) go to 4
3) go to 9
4) t1 = b*c
5) t2 = a+t1
6) x = t2
7) i = i+1
8) go to 2
9)
```

False

```
1) i = 1
2) if (i>n) go to 8
3) t1 = b*c
4) t2 = a+t1
5) x = t2
6) i = i+1
7) go to 2
8)
```

c) switch - case

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

```
1)  if (i == 1)  go to  7
2)  if (i == 2)  go to  11
3)  t1 = b3 * c3 ;        default
4)  t2 = a3 + t1
5)  x3 = t2
6)
7)  t1 = b1 * c1          case - 1
8)  t2 = a1 ~~*~~ + t1
9)  x1 = t2
10) go to  6
11) t1 = b2 * c2          case - 2
12) t2 = a2 ~~*~~ + t1
13) x2 = t2
14) go to  6
```

D) 1D Array

```
int A[10], B[10]
int x=0, i;
for (i=0; i<10; i++)
{
     x = x + A[i] * B[i].
}
```

	100	102	104	106	108
A	4	5	9	3	2
	0	1	2	3	4.

3 address code

$A[3] = BA + (i - lb) * c^{size}$

$= 100 + (3-0) * 2$

$= 106$

```
1) x = 0
2) i = 0
3) if (i ≮ 10) go to 15  false
4) t1 = Base address of A
5) t2 = i * 2
              ↳ size
6) t3 = t1[t2]
7) t4 = base address of B
8) t5 = i * 2
9) t6 = t4[t5]
10) t7 = t3 * t6
11) t8 = x + t7
12) x = t8
13) i = i + 1
14) go to 3
15)
```

calculation : in 2D
system :

E) 2D array

$x = A[i][j]$

$$A[4][4] = \begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ 0 & 00 & 01 & 02 & 03 \\ 1 & 10 & 11 & 12 & 13 \\ \rightarrow 2 & 20 & 21 & 22 & 23 \\ 3 & 30 & 31 & 32 & 33 \end{array}$$

RMO (Row Measure order

00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

i	j		element skip	
2	3	$2 \times 4 + 3 = 11$	element skip	offset
3	2	$3 \times 4 + 2 = 14$		

Address   offset value

$LOC(23) = BA + \underbrace{(2 \times 4 + 3)}_{} \times \underset{\text{if int (size)}}{2}$

$$\boxed{BA + (i \times N_c + j) \times w}$$ elements

$x = A[i][j]$ $A : \overset{R}{10} \times \overset{C}{15}$

3 address code

$t1 = i \times 15$

$t2 = t1 + j$

$t3 = t2 * 2$

$t4 =$ Base address of A

$t5 = t4[t3]$ $x = t5$

E) 3D address

For a C program accessing x[i][j][k] the following intermediate code generated by a compiler. Assume that size of an integer is 32 bits and the size of character is 8 bits

t0 = i × 1024

t1 = j * 32

t2 = k * 4

t3 = t1 + t0

t4 = t3 + t2

t5 = x[t4]

a) x is declared as int x[32][32][8]

b) " " int x[4][1024][32]

c) " " char x[4][32][8]

d) " " char x[32][16][2]

$2 \times 4 + 1 = 9$

$= (x N_c + j) w$

$= 2 \times 4 + 1 \times 2$

int x[32][32][8]

RMO

$[i][j][k] = x[i \times 32 \times 8 + \overset{j}{\cancel{32}} \times 8 + k] \times 4$

$= x[\cancel{8} i \times 32 \times 8 * 4 + j \times 8 \times 4 + k \times 4$

$= x[i \times 2^5 \times 2^3 \times 2^2 + j \times 2^3 \times 2^2 + k \times 2$

i][k

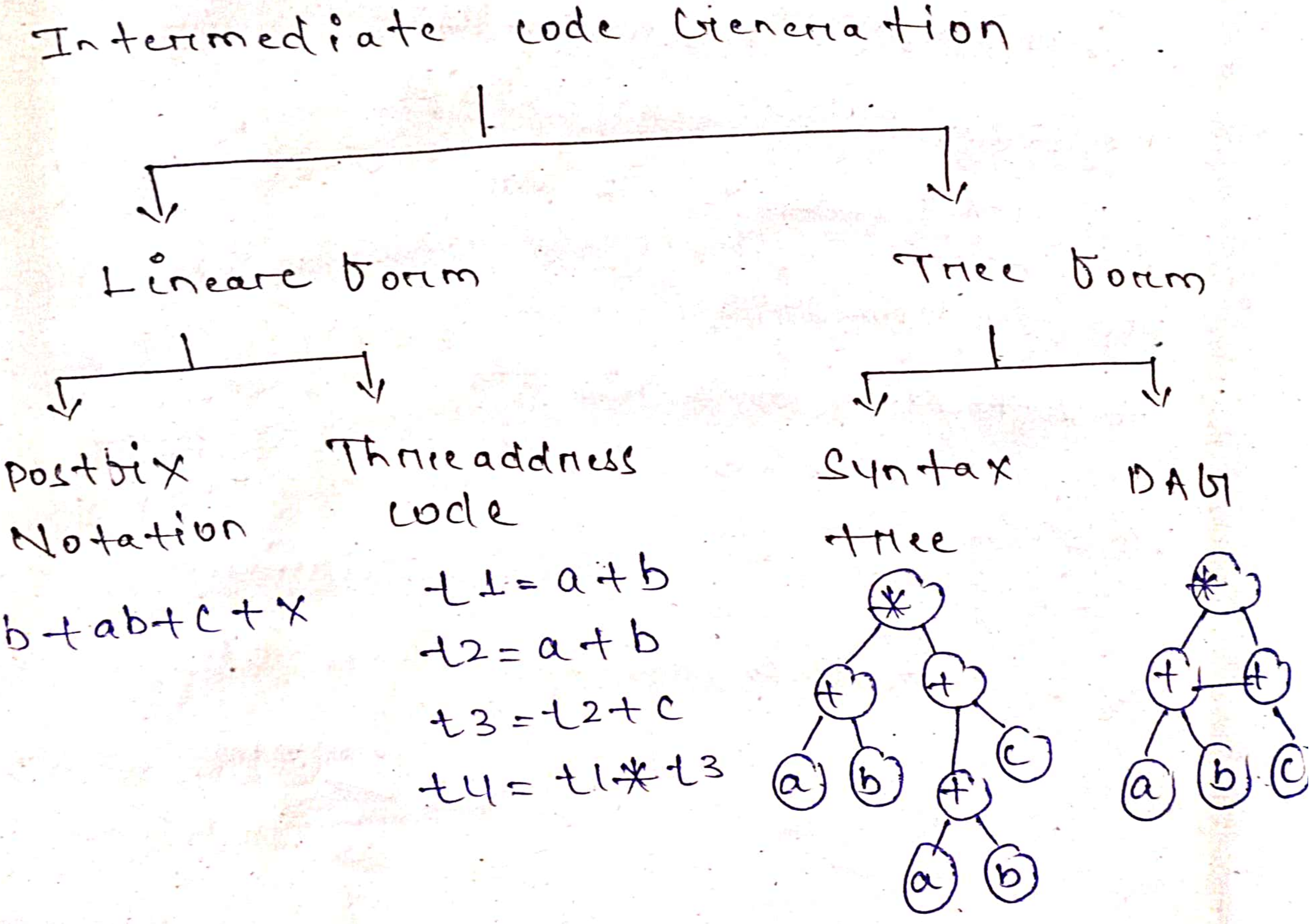
ted

n

$$= x[\underbrace{i \times 1024}_{t_0} + \underbrace{j \times 32}_{t_1} + \underbrace{k \times 4}_{t_2}]$$

$$= \underbrace{\underbrace{t_0 \quad t_1}_{t_3} \quad t_2}_{t_4}$$

$t5 = x[t4]$


Intermediate code Generation

Lineare Form

Tree Form

Postfix Notation

ab+ab+c+×

Three address code

t1 = a+b
t2 = a+b
t3 = t2+c
t4 = t1*t3

~~Postfix Notation~~

Operator Precedence Table

operator	Precedence	Associativity
↑	1	R to L
*, /	2	L to R
+, −	3	L to R

infix

<operand> <operator> <operand>

prefix (polish Notation)

<operator> <operand> <operand>

Postfix (Reverse Polish Notation)

<operand> <operand> <operator>

infix (LR)

$2 + 3 \times 4 = 2 + 12 = 14$

Postfix (LR)

234*+ 212+ = 14

prefix (RL)

+2*34 +2 12 = 14

$2 \uparrow 3 \uparrow 2$ (LR)

$2^{3^{2}} = 2^3 = 8^2 = 64$

RL

$3^2 = 9$ $2^9 = 512$ ✓

Postfix

567*8*+

5 42 8 * +

5 336 + = 341

E → E+T {print(+)}

E → T

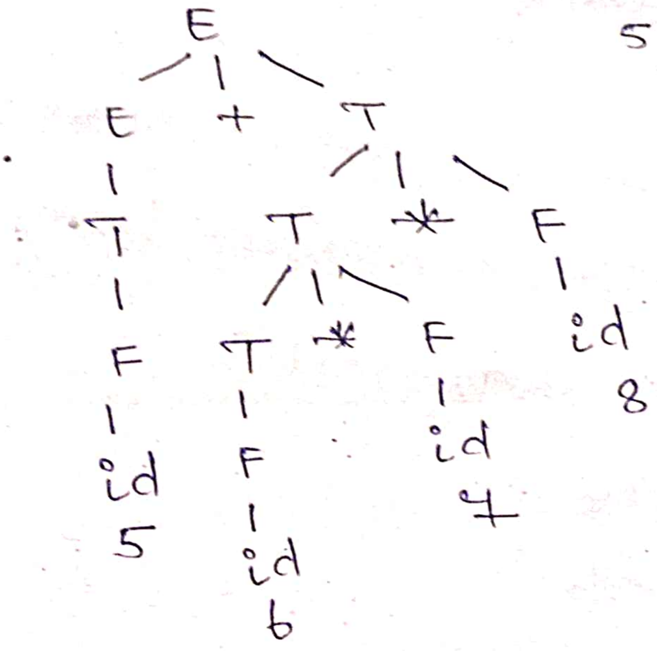
T → T*F {print(*)}

T → F

F → id {print(id.Lexval)}

Q 5+6*7*8

567*8*+



Intermediate code for procedure

procedures/function → return a value

eg r = f(a[i]);

a = array of integere

f = fun^n from integers to integers

1) t1 = i*4

2) t2 = a[t2]

3) Param t2

4) t3 = call f, 1

5) r = t3

Module - III

code Generation

→ It is the final phase of compiler

→ It takes input from code optimiza phase & produce the target code as result.

→ The objective of this phase is to allocate storage & produce target code

eg a = b + 70 (AL - ML)

```
MOV   R1, 10
MOV   R2, b
ADD   R1, R2
MOV   a, R1
```

Register Allocation & Assignment

instructions with register operands are faster than memory operands. efficient utilization of registers is important in generating good code.

→ various startegies for registers allocation & assignment

1. Assign specific values in target program to certain register

base address
arithmetic computation
top of the stack

1. Global Registere Allocation

→ Keep frequently used value in a fixed register.

→ Assign some fixed no. of registere to hold most active values in each innere loop.

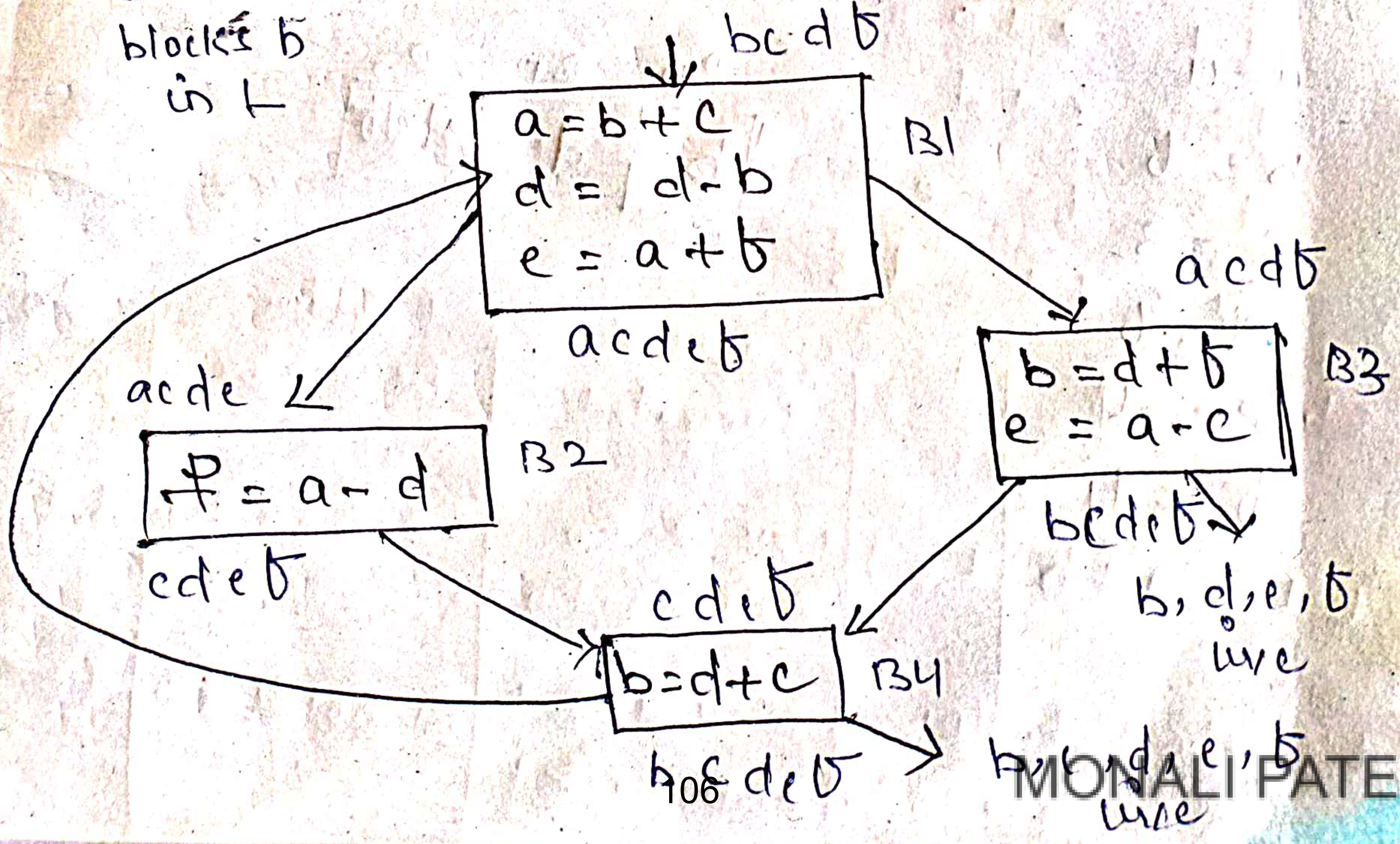
2. Usage counts

count a savings of one for each used of x in Loop L

→ If x is allocated a register then count a saving of two for each block in L

How many times variable used

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B)$$



Σ use(x,B) + 2*live(x,B))

↓ no. of times x used & not proceded by an assignment to x in same block

↳ 1 if x is live one exit & x is assigned value in B
→ 0

use(a,B1) + 2*live(a,B1) = 0 + 2* 1 = 2

use(a,B2) + 2*live(a,B2) = 1 + 2*0 = 1

use(a,B3) + 2*live(a,B3) = 1 + 2*0 = 1

use(a,B4) + 2*live(a,B4) = 0 + 2*0 =

use(b,B1) + 2*live(b,B~~1~~) = 2 + 2*0 = 2

use(b,B2) + 2×live(b,B2) = 0 + 2*0 = 0

use(b,B3) + 2*live(b,B3) = 0 + 2*1 = 2

use(b,B4) + 2*live(b,B4) = 0+ 2*1 = 2

a = 4
b = 6
c = 3
d = 6
e = 4
f = 4

R0	R1	R2
b	d	a/e/f

A simple code Generator

Generate target code for a sequence of 3 address statements.

→ For each operator in a statement, there is a corresponding target lang. operator.

Register & Address Descriptors

1. Register Descriptors

→ Keeps track of what is currently in each register.

→ Initially all registers are empty

2. Address Descriptors

→ Keeps track of the location where the current values of the name can be found

→ Location may be register, a stack location or memory address

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

3 address code sequence

$t1 =$	$a-b$		
$t2 =$	$a-c$		
$t3 =$	$t1+t2$		
$d =$	$t3+t2$		

Statement	code Generated	Registers Descriptore	Address Descri
$t1 = a-b$	MOV a, R0 SUB b, R0	Registers are empty R0 contains t1	t1 i
$t2 = a-c$	mov a, R1 sub c, R1	R0 contains t1 R1 contains t2	t1 t
$t3 = t1+t2$	Add R0, R1	R0 contains t3 R1 contains t2	t2 t
$d = t3+t2$	Add R0, R1 mov R0, d	R0 contains d	d in R d in R0, memory

Q $x = (a+b) - ((c+d) - e))$

$t1 = a+b$

$t2 = c+d$

$t3 = t2 - e$

$x = t1 - t3$

getreg()

Statement	L	code Generated	Registers Descriptore	Address descripto
$t1 = a+b$	R0	Mov a, R0 ADD b, R0	R0 holds t1	t1 in R0
$2 = c+d$	R1	mov c, R1 ADD d, R1	R1 holds t2	t2 in R1

t3=t2-e	R1	sub e, R1	R1 holds t3	t3 is in R1
x=t1-t3	R0	sub R1, R0 MOV R0, x	R0 holds x	x is in R0 and memory

Basic Blocks & flow Graph

Basic block is a set of statements that always executes in a sequence one after the other.

characteristics

→ They do not contain any kind of jump statements in them

→ There is no possibility of branching or getting halt in the middle.

→ All the statements executes in the same order they appear.

→ eg $a = b + c + d$

(1) $t1 = b + c$
(2) $t2 = t1 + d$
(3) $a = t2$

Basic block

Partitioning intermediate code into Basic blocks

P

Rule - 01 Determining Leaders

Following statements of the codes are called leadere

→ First statement of the code

→ Statement i.e a target of the condi or unconditional go to statement.

→ Statement that appears immediately at a go to statement.

Rule - 02 Determining Basic Blocks

→ All the statements that follow the leadere (including leader), till the next leadere appears from one basic blocks

→ The first statement of the code is called ~~basic~~ as the first leadere.

→ The block containing the first lead is called as initial block

Flow Graph

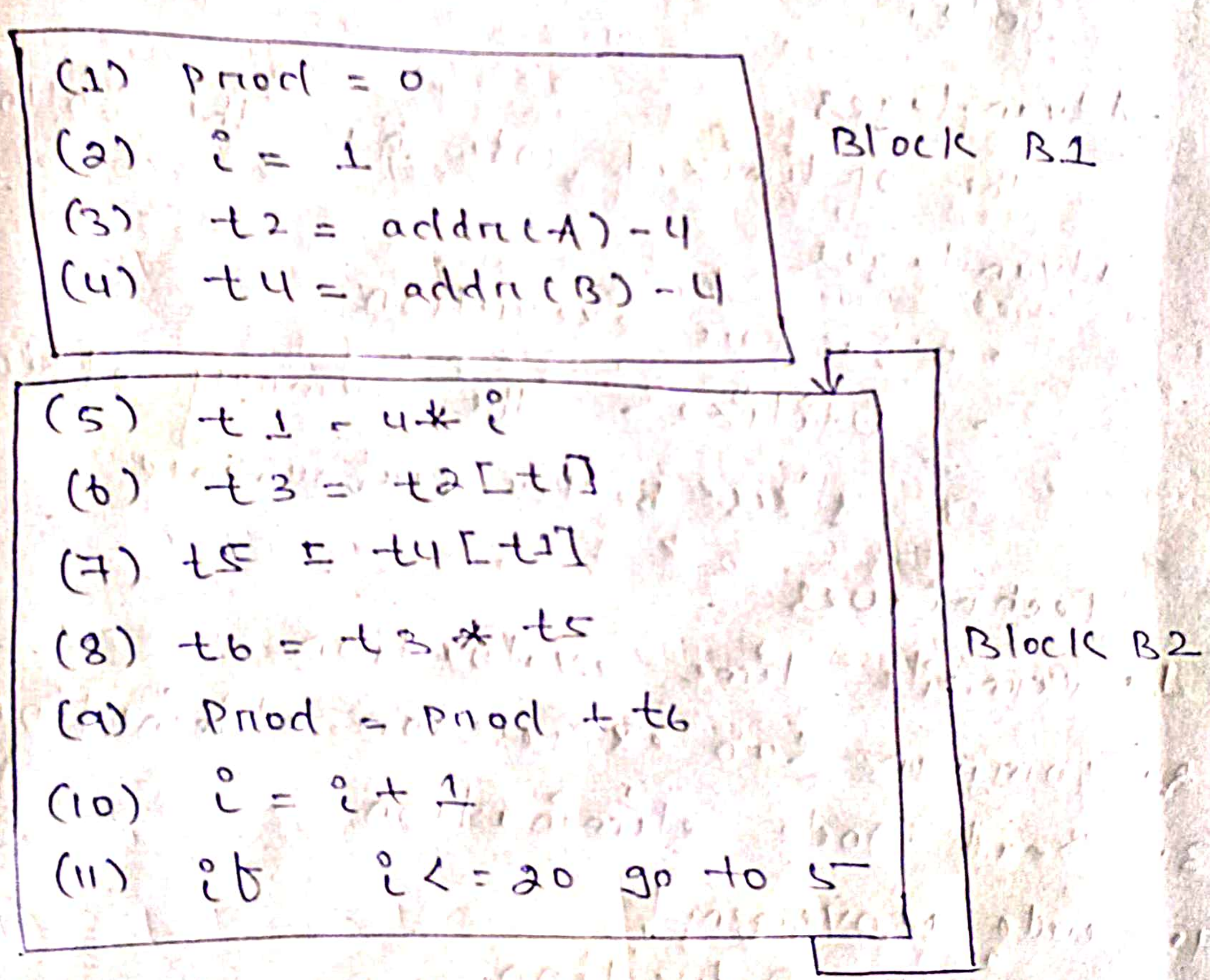
The basic blocks serve as nodes of the flow Graph.

→ There is a directed edge from block B1 to Block B2 if B2 appears immedia after B1

1. Prod = 0

2. i = 1

```
(3) t2 = addr(A) - 4
(4) t4 = addr(B) - 4
(5) t1 = 4 * i        → conditional go to
(6) t3 = t2[t1]
(7) t5 = t4[t1]
(8) t6 = t3 * t5
(9) prod = prod + t6
(10) i = i + 1
(11) if i <= 20 go to 5
```



```
(1) prod = 0
(2) i = 1
(3) t2 = addr(A) - 4
(4) t4 = addr(B) - 4
```

```
(5) t1 = 4 * i
(6) t3 = t2[t1]
(7) t5 = t4[t1]
(8) t6 = t3 * t5
(9) prod = prod + t6
(10) i = i + 1
(11) if i <= 20 go to 5
```

Flow Graph

→ Directed Graph

→ Nodes (Basic block)

→ edges (flow of control)

code optimization

code optimization is an approac to enhance the performance of the cod.

→ It involves

- Eliminating the unwanted code lin
- Rearranging the statement of t code.

Advantages

The optimized code has the follow advantages

- Faster execution
- Utilizes the memory efficie
- gives better performance

Techniques

1. compile time evaluation
2. common sub expression elimination
3. Dead code elimination
4. code Movement
5. strength Reduction

1. compile time Evaluation

a) constant folding

→ It involves folding the constants

→ The expressions that contain the operands having constant values compile time are evaluated

→ Those expressions are then replaced with theire respective results

eg circumference of circle $= \frac{22}{7} * \text{diameter}$

$$\Downarrow$$

$$3.14 * \text{diameter}$$

b. constant Propagation

If some variable has been assigned some constant value, then it replaced that variable with its constant values in the further program.

→ eg $pi = 3.14$
$radius = 10$

Area of circle $= pi \times radius \times radius$
$= 3.14 \times 10 \times 10$

2. common subexpression elimination

The expression that has been already computed before & appears again in the code for computation is called common sub-expression.

→ it involves eliminating the common sub expression

→ Redundant expressions are eliminated

→ already computed result

Before optimization	After optimiz
S1 = 4*i	S1 = 4*i
S2 = a[S1]	S2 = a[S1]
S3 = 4*j	S3 = 4*j
S4 = 4*i	S5 = n
	S6 = b[S1] + S
S5 = n	
S6 = b[S4] + S5	

3. Code Movement

→ it involves movement of the code

→ code present inside the loop is moved out if it does not mattere whether it is present inside or out

B4

Eg

```
for (int j=0; j<n; j++)
{
    x = y * z;
    a[j] = 6 * j ;
}
```

Aftere

```
x = y * z
for( .. )
{
    a[j]= 6*j;
}
```

4. Dead code Elimination

Eliminating the dead code.

→ The statement of the code which either never executes or are unreachable ore theire o/p is nevere used are eliminated

Beforre

```
i = 0;
if (i = = 1)
  {
      a = x + 5;
  }
```

Aftere

```
i = 0
```

5. Strength Reduction

Reducing the strength of expression Replaces the expensive & costly op. operators with simple & cheapere one

Beforre

```
B = A * 2
```

Aftere

```
B = A + A
```

What is Peephole Optimization in Compiler Design?

Code optimization that is applied to a small section of the code is known as peephole optimization in compiler design. It is called local optimization because it works by evaluating a small section of the generated code, generally a few instructions, and optimizing them based on some predefined rules. Peephole or window refers to the brief sequence of instructions or brief section of code on which peephole optimization in compiler design is carried out.

Objectives of Peephole Optimization in Compiler Design

The following are the objectives of peephole optimization in compiler design:

- **Increasing code speed:** Peephole optimization in compiler design seeks to improve the execution speed of generated code by removing redundant instructions or unnecessary instructions.
- **Reduced code size:** Peephole optimization in compiler design seeks to reduce generated code size by replacing the long sequence of instructions with shorter ones.
- **Getting rid of dead code:** Peephole optimization in compiler design seeks to get rid of dead code, such as unreachable code, redundant assignments, or constant expressions that have no effect on the output of the program.
- **Simplifying code:** Peephole optimization in compiler design also seeks to make generated code more understandable and manageable by removing unnecessary complexities.

Working of Peephole Optimization in Compiler design

The working of Peephole optimization in compiler design can be summarized in the following steps:

Step 1 – Identify the peephole: In the first step, the compiler finds the small sections of the generated code that needs optimization.

Step 2 – Apply the optimization rule: After identification, in the second step, the compiler applies a predefined set of optimization rules to the instructions in the peephole.

Step 3 – Evaluate the result: After applying optimization rules, the compiler evaluates the optimized code to check whether the changes make the code better than the original in terms of speed, size, or memory usage.

Step 4 – Repeat: The process is repeated by finding new peepholes and applying the optimization rules until no more opportunities to optimize exists.

Peephole Optimization Techniques

Here are some of the commonly used peephole optimization techniques:

Constant Folding

Constant folding is one of the peephole optimization techniques that involves evaluating constant expressions at compile-time instead of run-time. This optimization technique can significantly improve the performance of a program by reducing the number of computations performed at run-time.

Here is an example of Constant folding:

Initial Code:

```
int x = 10 + 5;
int y = x * 2;
```

Optimized Code:

```
int x = 15;
int y = x * 2;
```

Explanation: In this code, the expression 10 + 5 is a constant expression, which means that its value can be computed at compile-time. Instead of computing the value of the expression at run-time, the compiler can replace the expression with its computed value, which is 15.

Strength Reduction

Strength Reduction

Strength reduction is one of the peephole optimization techniques that aims to replace computationally expensive operations with cheaper ones, thereby improving the performance of a program.

Here is an example of strength reduction:

Initial Code:

```
int x = y / 4;
```

Optimized Code:

```
int x = y >> 2;
```

Explanation: In this code, the expression y / 4 involves a division operation, which is computationally expensive. So, we can replace this with a shift right operation, as bit-wise operations are generally faster.

Redundant Load and Store Elimination

Redundant load and store elimination is also one of the peephole optimization techniques that seeks to reduce redundant memory accesses in a program. This optimization works by finding code that performs the same memory access many times and removes the redundant accesses.

Here is an example of this:

Initial Code:

```
int x = 5;
int y = x + 10;
int z = x + 20;
```

Optimized Code:

```
int x = 5;
int y = x + 10;
int z = y + 10;  // optimized line
```

Explanation: In this code, the variable x is loaded from memory twice: *once in the second line and once in the third line. However, since the value of x does not change between the two accesses, the second access is redundant. In the optimized code, the redundant load of x is eliminated by replacing the second access with the value of y,* which is computed using the value of x in the second line.

Null Sequences Elimination

Null sequences Elimination is a peephole optimization technique used in compiler design to remove unnecessary instructions from a program. The optimization involves identifying and removing sequences of instructions that have no effect on the final output of a program.

Here is an example of null sequences elimination:

Initial Code:

```
int x = 5;
int y = 10;
int z = x + y;
x = 5;  // redundant instruction
```

Optimized Code:

```
int x = 5;
int y = 10;
int z = x + y;
```

Explanation: In this code, the value of x is assigned twice: once in the first line and once in the fourth line. However, since the second assignment has no effect on the final output of the program, it is a null sequence and can be eliminated.

Code Optimization Technique is an approach to enhance the performance of the code by either eliminating or rearranging the code lines. Code Optimization techniques are as follows:

1. *Compile-time evaluation*
2. *Common Sub-expression elimination*
3. *Dead code elimination*
4. *Code movement*
5. *Strength reduction*

Common Sub-expression Elimination:

The expression or sub-expression that has been appeared and computed before and appears again during the computation of the code is the common sub-expression. Elimination of that sub-expression is known as Common sub-expression elimination.

The advantage of this elimination method is to make the *computation faster and better* by avoiding the re-computation of the expression. In addition, it utilizes memory efficiently.

Types of common sub-expression elimination

The two types of elimination methods in common sub-expression elimination are:

1. Local Common Sub-expression elimination– It is used within a single basic block. Where a basic block is a simple code sequence that has no branches.

2. Global Common Sub-expression elimination– It is used for an entire procedure of common sub-expression elimination.

Example 1:

Before elimination –

```
a = 10;

b = a + 1 * 2;

c = a + 1 * 2;

//'c' has common expression as 'b'
d = c + a;
```

After elimination –

```
a = 10;

b = a + 1 * 2;

d = b + a;
```

Let's understand Example 1 with a diagram:

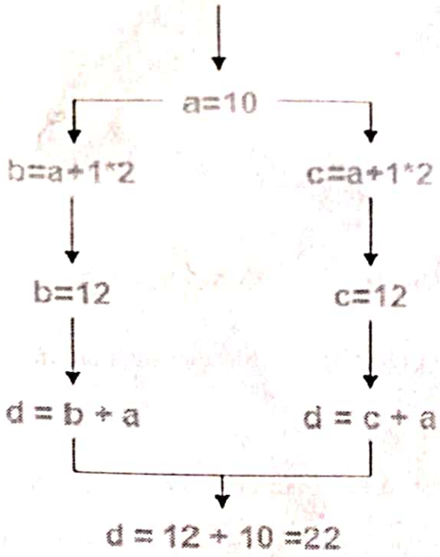


fig.: Example 1

As shown in the figure (*fig.: Example 1*), the result of 'd' would be similar with both expressions. So, we will eliminate one of the common subexpressions, as it helps in faster execution and efficient memory utilization.

Example 2:

Before elimination –

x = 11;

y = 11 * 24;

z = x * 24;

//'z' has common expression as 'y' as 'x' can be evaluated directly as done in 'y'.

After elimination –

y = 11 * 24;

In compiler design, redundant code elimination removes unnecessary computations, while unreachable code elimination removes code that will never be executed, improving efficiency and performance.

Redundant Code Elimination:

- Definition: Redundant code involves computations performed multiple times when the result is already known or can be derived from a previous calculation.
- Example:
  C
  ```
  int a = 5;
  int b = a * 2;
  int c = a * 2; // Redundant, as 'c' can be derived from 'b'
  ```

In this case, calculating a * 2 for c is redundant because the value is already stored in b.

- Compiler Optimization: A compiler can optimize this by replacing the redundant calculation with c = b;.
  Unreachable Code Elimination:
- Definition: Unreachable code consists of instructions that are never executed during program execution.

- Example:

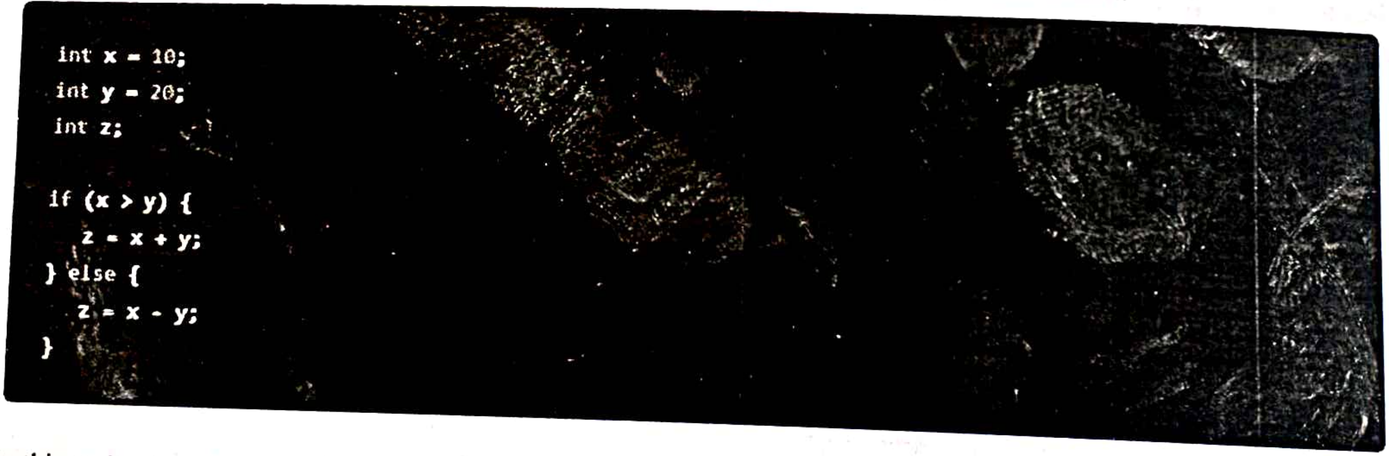
```
C
   int x = 10;
   if (x > 100) {
      // This code will never be executed
      printf("This will not be printed");
   }
   printf("This will be printed");
```

The code inside the if statement is unreachable because x is initialized to 10, and the condition x > 100 will always be false.

- Compiler Optimization: A compiler can identify and remove such unreachable code, resulting in a smaller and faster executable.

Control flow optimization is a technique in compiler design that improves the efficiency of control flow structures in a program. It includes optimizations such as branch prediction, loop optimization, dead code elimination, simplification of control flow graphs, and tail recursion elimination.

The main objective is to minimize the impact of conditional branches and loops on program performance. By predicting branch outcomes, optimizing loops, removing dead code, simplifying control flow graphs, and transforming tail recursion, the compiler enhances execution speed and resource utilization.



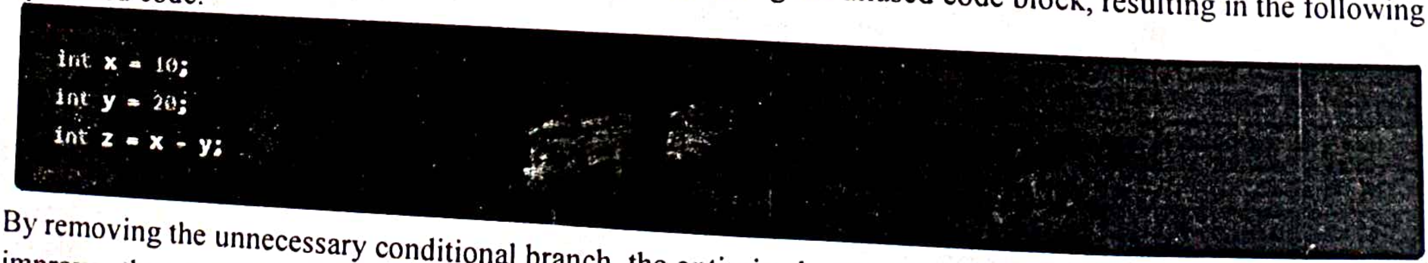
```
int x = 10;
int y = 20;
int z;

if (x > y) {
  z = x + y;
} else {
  z = x - y;
}
```

In this code, there is a conditional statement that checks if x is greater than y. Based on the condition, either the addition or subtraction operation is performed, and the result is stored in variable z.

Through control flow optimization, the compiler can perform branch prediction and determine that the condition x > y is always false. In this case, it knows that the code inside the if block will never be executed.

As a result, the compiler can optimize the code by eliminating the unused code block, resulting in the following optimized code:



```
int x = 10;
int y = 20;
int z = x - y;
```

By removing the unnecessary conditional branch, the optimized code becomes simpler and more efficient. This improves the program's execution speed and reduces any overhead associated with evaluating the condition.

Module - IV

Storage organization

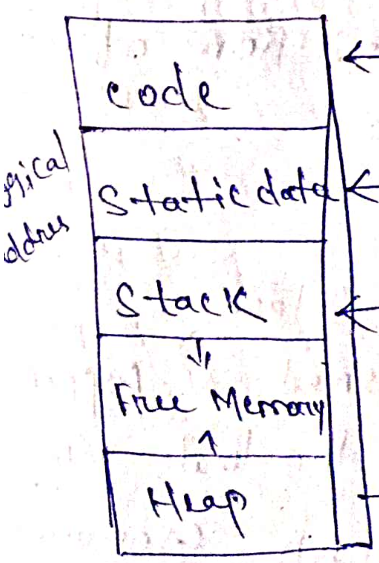
The executing target program runs in its own logical address space in which each program value has a location.

→ The management & org^n of this logical address space is shared bet^n compiler, OS & target machine.

Logical Address	physical Address
→ address is generated by cpu while a program is running.	→ physical location of required data in a memory.

→ The OS maps the logical address into physical address, which are usually spread throughout memory.

Sub division of Runtime Memory



← memory location for code are determine at compile time.

← location of Static data can also be determi

← Data objects Allocated at run time (Activation Records)

→ other Dynamically allocated data object at run-time

Storage Allocation

The different ways to allocate memory are

1. Static Storage Allocation
2. Stack Storage allocation
3. Heap Storage Allocation

1. Static Storage Allocation

→ In static allocation names & bounds to storage location

→ If memory is created at compile time the the memory will be created in static area & only once.

→ Static allocation support the dynamic DS that means memory is created only at compile time & deallocated after program completion

→ The drawback with static storage allocation is that the size & position of data objects should be known at compile time. 2

→ Another drawback is restriction of the recursion procedure.

2. Stack Storage Allocation

→ Storage is organized as stack

→ Activation record are pushed & popped

→ Activation record contains the locals so that they are bound to fresh storage in each activation record.

The value of locals is deleted when the activation ends.

→ It works on the basis of LIFO & its allocation supports the recursion process

3. Heap Storage Allocation / Dynamic

→ It is most flexible allocation scheme.

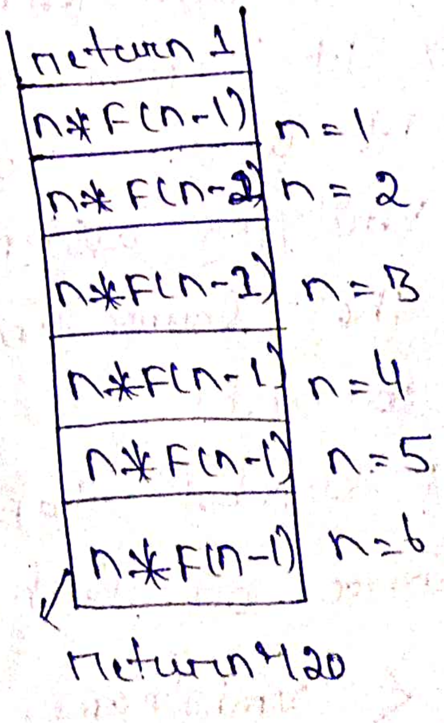
→ Allocation & deallocation of memory can be done at any time & at any place depending upon the users requirement.

→ Heap allocation is used to allocate memory to the variable dynamically & when the variables are no more used then claim it back.

→ Heap storage allocation supports the recursion process.

Eg

```
fact(int n)
{ if (n<=1)
    return 1;
  else
    return n*fact(n-1)
  }
fact(6)
```



Symbol Table

→ Symbol tables are DS that are used by compiler to hold info. about source program constructs.

→ It is used to store info. about the occurance of various entities such as,
objects, classes, variable names, fun^{n} etc

→ It is used by both analysis phase & synthesis phases.

Purpose

→ It is used to store the name of all entities in a structured form at one place.

→ It is used to verify if a variable has been declared.

2 → It is used to determine scope of name

→ It is used to implement type checking by verifying assignment & expressions in the source codes are semantically correct.

→ A Symbol table can either be a linear or Hash table

eg < Symbolname, type, attribute>

<Static, int, Salary>

Activation Record

→ control stack is a runtime stack which is used to keep track of, the live procedure activations, i.e it is oled to find out the procedurec whose execution have not been completed.

→ when the activation begins then the procedure name will push on to the stack & when ~~returns~~ (activation ends) then it will popped.

→ Activation record is used to manage the info. needed by a single execution of a purpose.

→ An activation record is pushed into the stack when a procedure is called & is popped when the control returns to the caller fun^{n}.

content of Activation Record

Return value: It is used by called procedure to return a value to calling procedure

Actual Parameter:- It is used by calling procedure to supply parameters to called procedure.

Return value
Actual parameter
control link
Access link
saved machine status
local Data
Temporaries

control link :- It points to activation record of the caller.

Access Link It is used to refer to non local data held in other activation records.

saved machine Status :-

It holds the info. about Status of machine before the procedure is called.

Local Data

It holds the data i.e local to the execution of the procedure.

Temporaries

It stores the value that arises in the evaluation of an expression

2