25 Jan 24

ॐ

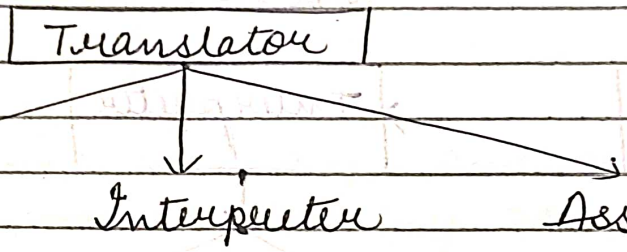
Date: / / Page no:

Unit - I

Compiler Design...

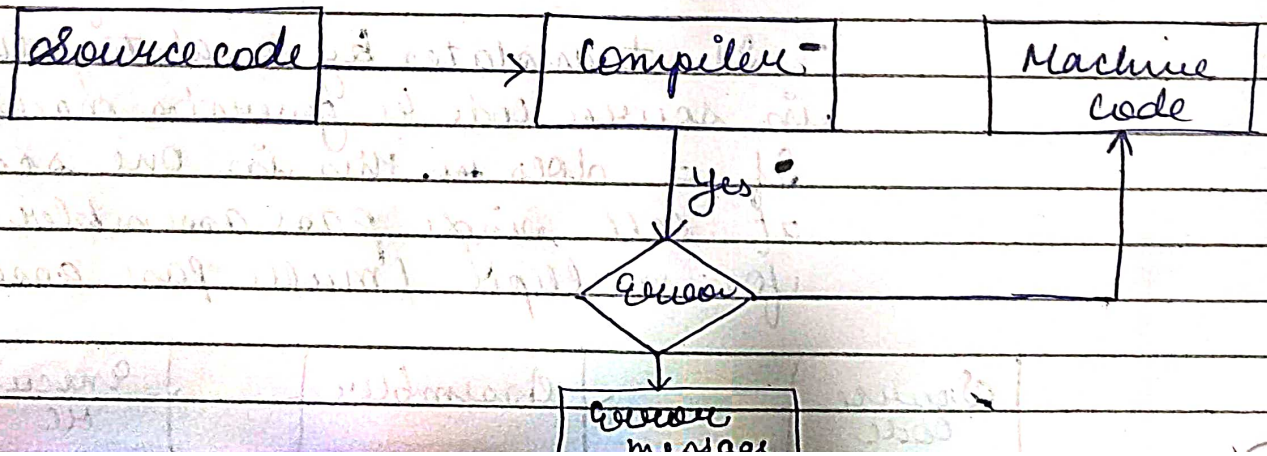
Translator A translator in compiler design translates this high-level lang. into machine code.

It takes input as source code (HLL) and gives output as machine code (LLL)



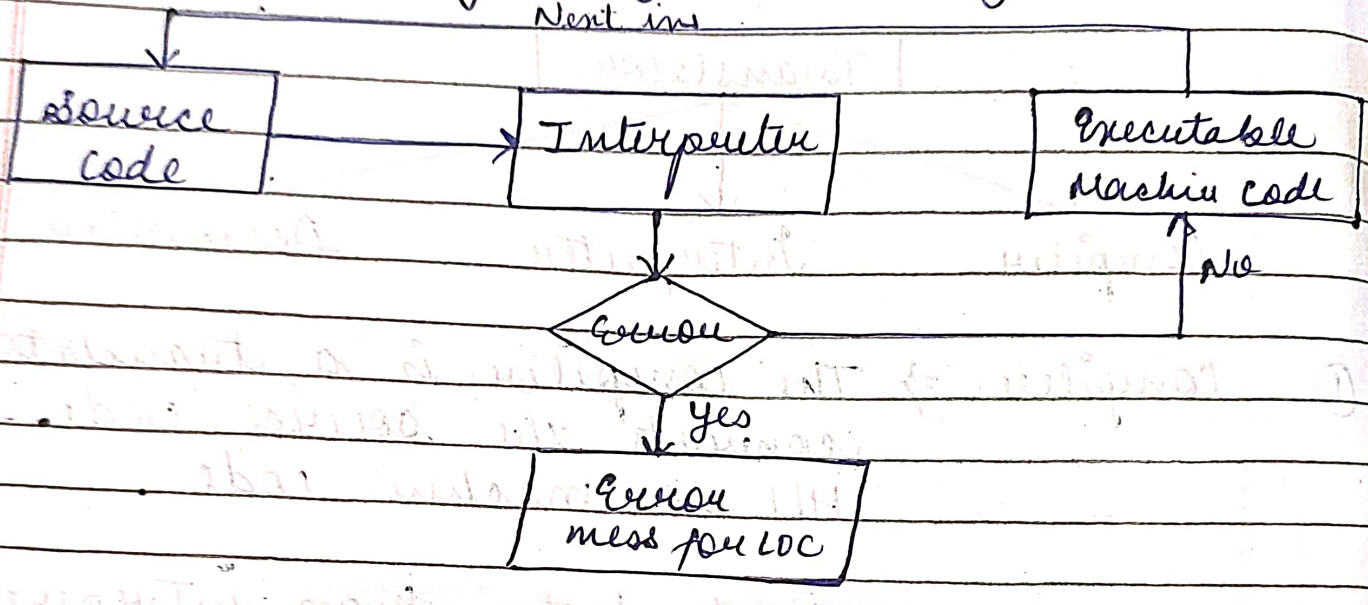
① **Compiler** → The compiler is a translator that converts the source code from HLL to machine code.

- Much faster than interpreter
- Error debugging is little diff process
- Eg → C++, C, Java.



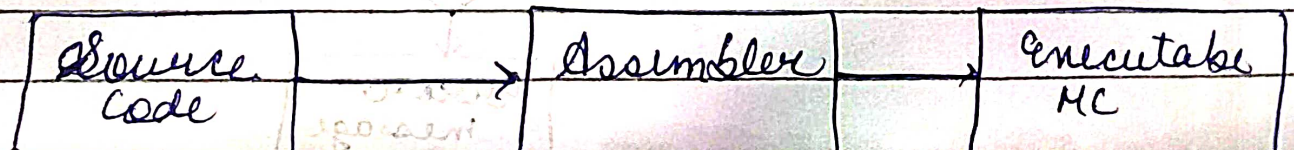
(2) Interpreter → It changes the source code into machine code but the process is a bit changed.

- Accepts a source code & executes directly without producing any object
- Solving error is easy but ↑ execution
- eg. → Python, Ruby, BASIC



(3) Assembler → Fastest translator among three, as the assembler translates LLL to machine code.

- It translates by evaluating the symbol in source code & generates machine code. If it does so this in one scan, it call single pass assembler & for multiple (multi pass assembler)



① Compiler

- It helps to validate the code. so that there is no system error.
- User need not to run the code on same machine
- execution of code is faster with the help of the compilers as they optimize the code to take less time and space complexity.

② Interpreter

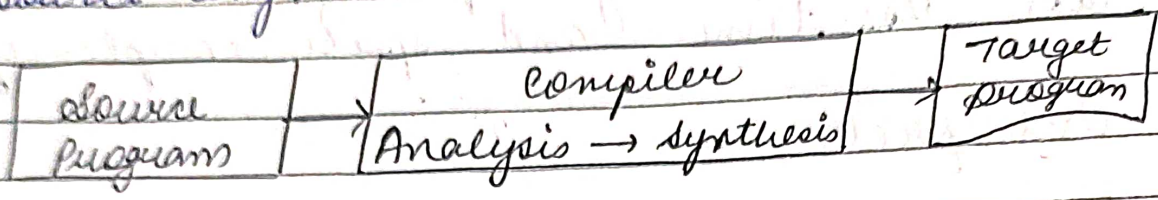
- Error detection is easy and simultaneously when the user is writing the code
- User can work on small code & join or merge them, which can be done with more accuracy

③ Assembler

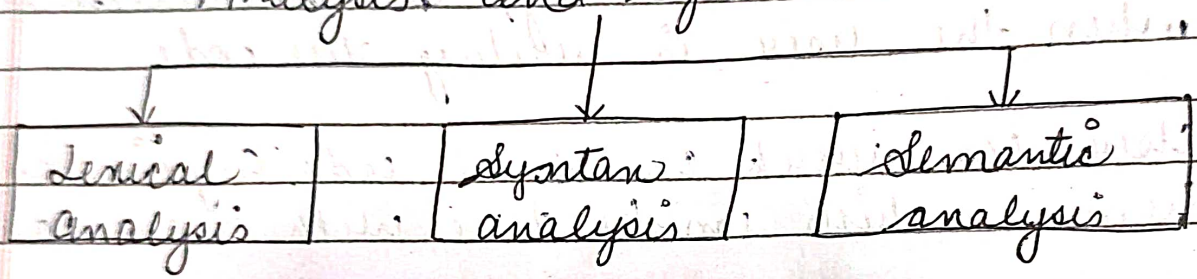
- Efficiency is code execution
- Ease of code maintainance and error fixing
- less time in code execution. as the code is in assembly lang.

→ Compiler : Analysis Synthesis Model

Compilation can be done in 2 parts – analysis and synthesis.
In Analysis part the source program is read and broken down in contituent pieces.
In Synthesis part this intermediate form of the source lang. is taken an converted into target



Analysis and Synthesis model

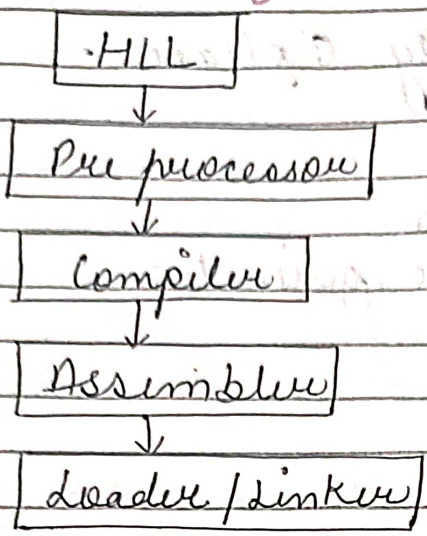


(1) Lexical analysis → In this step the source program is read and then. It is into streams of strings. Such strings called Tokens.
Tokens are not charts having same meaning.

(2) Syntax Analysis → In this step the tokens are arranged in hierarchical structure that helps in finding the syntax of source string.

(3) Symentic Analysis → This step the meaning of the source string is determined.

Language processing systems



{ source code }
{ # include, # define }
{ HLL → LLL }
{ LLL → machine code }
{ load / link to prog }

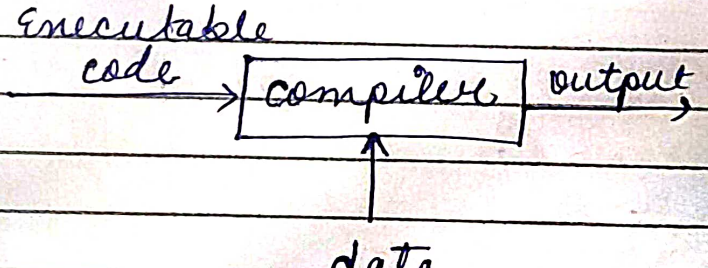
1) C program {HLL}

2) C. Compiler translates progm into LLL (Assembly lang)

3) Assembler translates LL into machine lang / code (0101--)

4) Linker used to link all parts of ~~pro~~ program together for execution.

5) Loader loads all them into memory & the program is executed.



4) Error checking
5) Productivity

1) Portability
2) Optimization
3) Maintainability

Phases of Compilers —

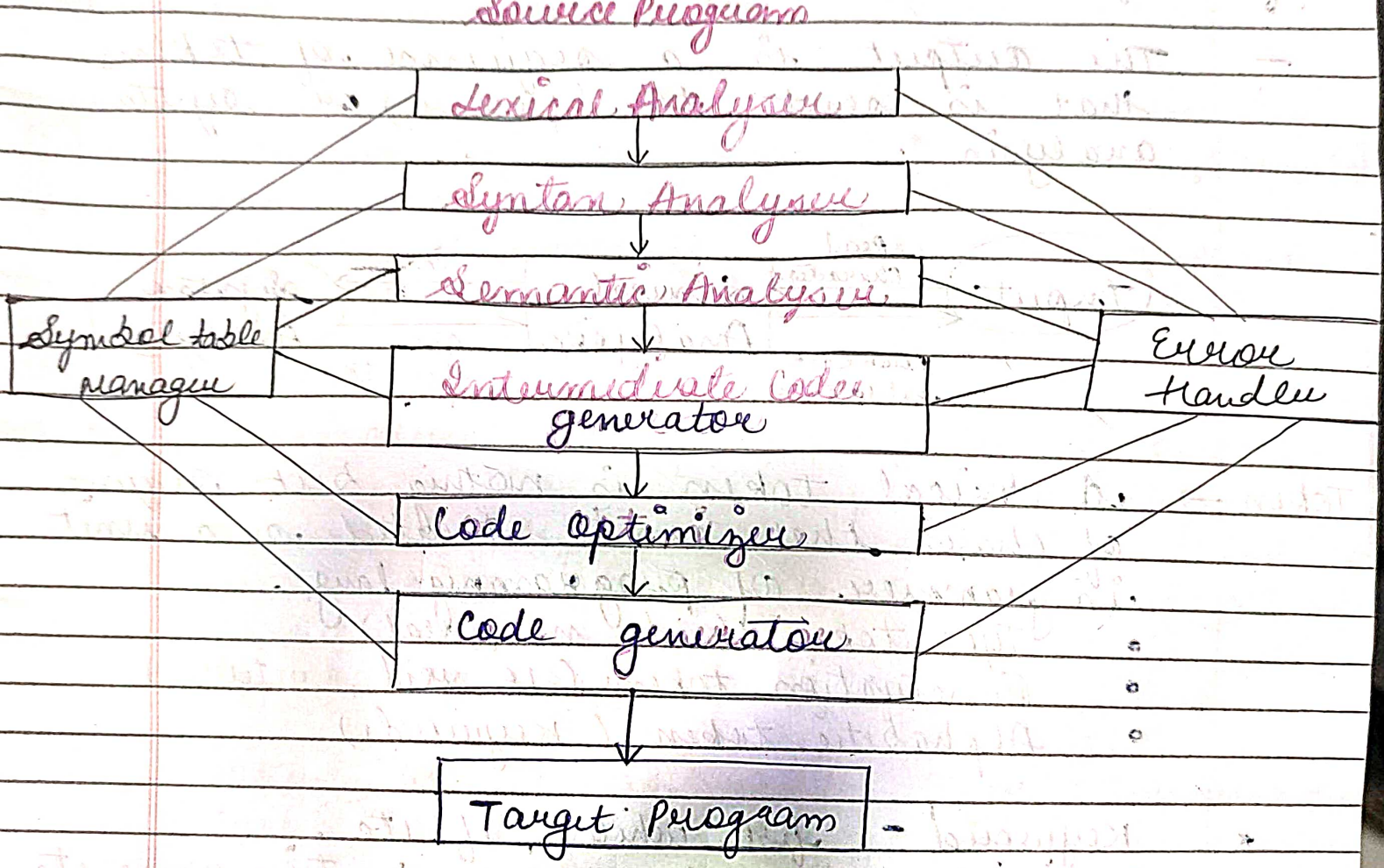
There are typically 6 phases —

1) Lexical Analysis
2) Syntax Analysis
3) Symentic Analysis
4) Intermidiate code generation
5) Optimization
6) Code generation.

Phase → Phases are logically interrelated oper^{n} that takes source pgm in I/p and produce o/p in another I/p^{n}.

a) Analysis phase
- Machine independent
- Language dependent

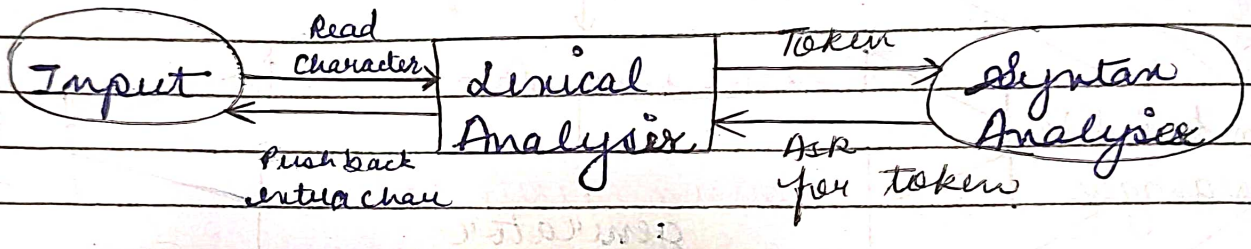
b) Synthesis phase
- Machine dependent
- Lang independent



27 Jan 24

1'st Phase (Lexical Analyzer)

- Lexical Analysis is the first phase of compiler Also Known as scanner
- It convert HL program into a sequence of Tokens.
- It can be implemented with Deterministic finite Automata.
- The output is a sequence of tokens that is sent to (parser) or syntax analysis.



Token → A lexical token is nothing but sequence of char that can be treated as a unit in grammer of programming lang.

- Type token (id, no, real)
- Punctuation tokens (IF, void, return)
- Alphabetic tokens (Keywords)

* Keyword - for, while, if etc.
* Identifiers - Variable name, function name etc
* Operators - +, ++, - etc.
* Seperators - , ; etc

Non Tokens → Comments, preprocessor directive, macro, blanks, tabs, newline etc..

Lexeme → The sequence of char matched by a pattern to form the corresponding token or a sequence of input char that comprises a single token is called lexemes. eg - "float", "abs_zero_Kelvin", "=", "-", "273", ";".

How the phase works —

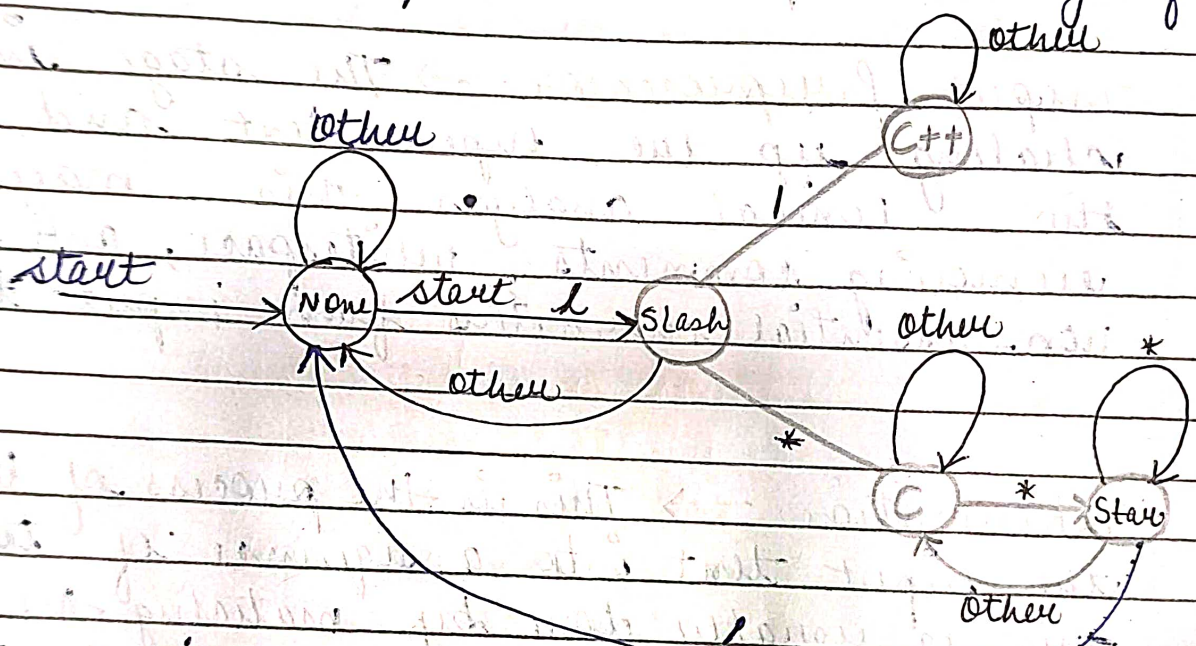
1) Input Preprocessor → The stage involves cleaning up the input text and preparing the lexical analysis this may removing comments, whitespace, and other non-essential characters from input text.

2) Tokenization → This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions.

3) Token Classification → The lexer determines the type of each token. For eg - in programming lang the lexer might classify keywords, identifiers, operators or punctuation symbols

4) Token Validation → The lexer checks that each token is valid according to the rules of programming language.
For eg → it might checks that a variable name is a valid identifier, or that an operator has correct syntax.

5) Output generation → The lexer generates the output of lexical analysis process, which is typically a list of tokens, this list of tokens can then be passed to the next stage of compilation.



The lexical analyser identifies the error with the help of automation machine and the grammar of given lang, on which it is based like C, C++,

Suppose we pass a stmt through lexical analyser. — a = b + c ;

It will generate token sequence like id = id + id ;

where, { id = each variables } in symbol table.

Program —

```
int main()
{
    // 2 variables
    int a, b;
    a = 10;
    return 0;
}
```

Ans → Tokens are : 18

printf (" Geeks Quiz ") ;
1 2 3 4 5

There are 5 valid tokens

#
```
int main()
{
    int a = 10, b = 20;
    printf("sum is : %d", a+b);
    return 0;
}
```

Ans → Total no of tokens are : 27

	Lexeme	Tokens
1)	while	WHILE
2)	(	LAPREN
3)	a	IDENTIFIER
4)	>=	COMPARISION
5)	b	IDENTIFIER
6)	)	RAPAREN

Lexemes	Tokens
a	IDENTIFIER
=	ASSIGNMENT
a	IDENTIFIER
-	ARITHMETIC
2	INTEGER
;	SEMICOLON

Advantages—

1) Efficiency → Improves efficiency, but the input Dre into chunks, to focus on structure of code.

2) Flexibility → Allows for the use of keywords. This makes it easier to create new programming.

3) Error Detection → It can detect errors such as misspelled word, missing semicolon(;) and undefined variables. Saves a lot of time in debugging process

4) Code Optimization → Optimize code by identifying comman pattern and replacing them with more efficient code. Improve the program.

Disadvantages -

1) Complexity → Complex require lot of computational power. This make is diff to implement.

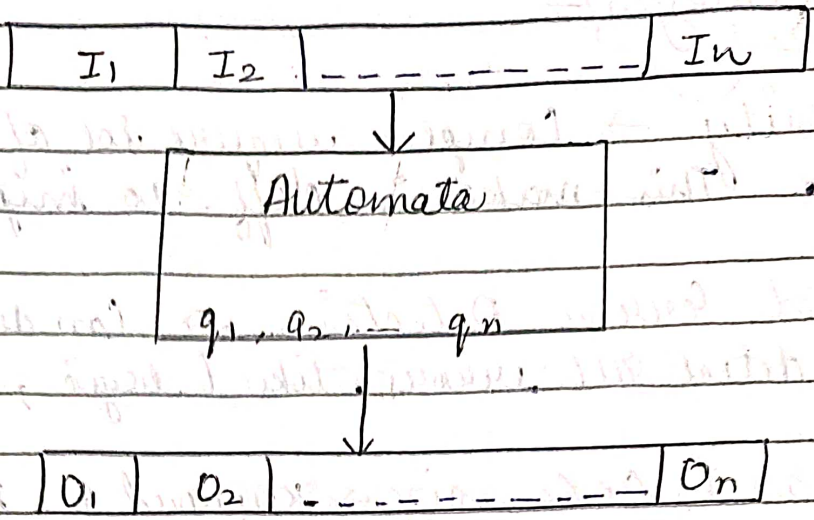
2) Limited Error Detection → Can detect error, can't detect all errors, like (logic, type) errors.

3) Increased Code size → Keyword & reserved words can ↑ the size of code, make diff to read/write

4) Reduced Flexibility → Use of Keyword reduce flexibility.

Finite Automata

- FA is a simplest machine to recognize patterns. It is used to characterize a Regular lang. for eg. /&aa+1/.
- Also used to analyze and recognize natural lang. expression.
- The FA & Finite State Machine (FSM).
- Graphical (Transition Diagram / Table)
- Tabular (Transition table)
- Mathematical (Transition function / Mapping function)



1) Input , Output
2) State of Automata
3) State of relation
4) Output relation

A finite automata consist of following Tuples :-

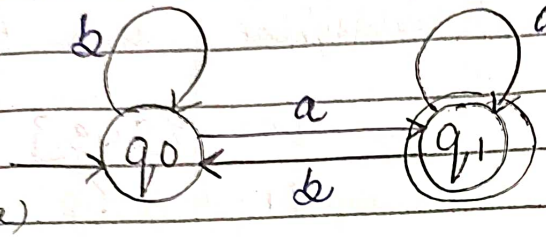
Q → set of all states (finite)
Σ ? → set of Input symbols (finite)
q → Initial state
F → set of final states
δ ? → Transition function. ($Q * \Sigma$)

1) DFA (Deterministic finite Automata) → The machine goes to one state only.
- A transition fun is defined on every state for every input symbol.
- DFA can't change state without any input character.

Eg → Construct a DFA which accept a language of all strings ending with 'a'.

Given : $\Sigma = \{a, b\}$ $q = \{q_0\}$ $F = \{q_1\}$ $Q = \{q_0, q_1\}$

Consider a language set of all the possible acceptable strings in order to construct an accurate state transition diagram



Strings that are accepted

L = { a, aa, aaa, aaaa, aaaaa, ba, bba, bbbaa, aba, abba, aaba, ababa }

Strings that are not accepted

L = ab, bb, aab, abbb etc.

- There can be many DFA pattern.
- A DFA with a min no of state is generally preferred.

State / Symbol	a	b
q_0	q_1	q_0
q_1	q_1	q_0

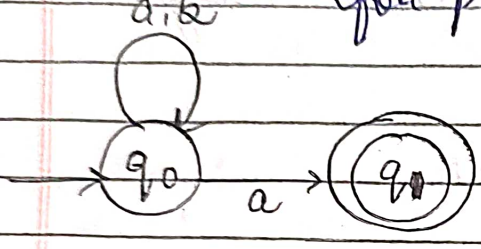
There can be many possible DFA's for pattern.

Every NFA is not DFA but
each NFA can translated into DFA ..

2) NFA (Non Deterministic finite Automata) -
Just similar to DFA but -

1) NULL (or ?) move is allowed i.e. it can move forward without reading symbols.

2) Ability to transmit to any no. of states for particular input.



?State / symbol	a	b
q_0	$\{q_0, q_1\}$	q_0
q_1	?	?

3) In terms of power both are equivalent.

4) In NFA, if any path for an input string lead to final state, then the input string is accepted.

NFA states (same as DFA) but diff transition function -

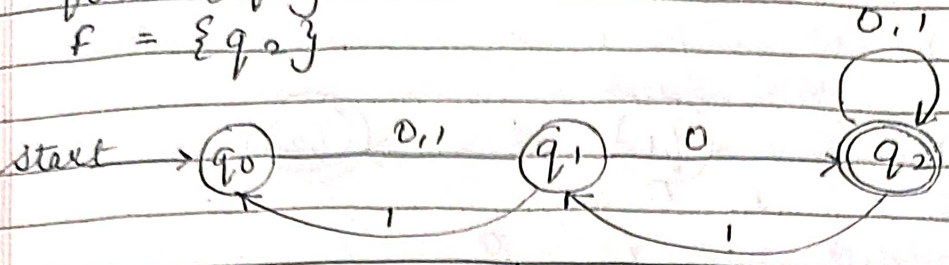
δ : $Q \times \Sigma \rightarrow 2^Q$

Q : finite set of states
Σ : finite set of input symbols
q_0 : initial state
F : final state
δ : Transition function.

strings ending is in (ab) (abab), (abab)
ending is in (ab) (aab) (abab)

Date: / / Page no:

Eg → $Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
$q_0 = \{q_0\}$
$F = \{q_2\}$



State	0	1
q_0	q_0, q_1	q_1
q_1	q_2	q_0
q_2	q_2	q_1, q_2

DFA :- **Type 1 (Problem)** 1 Feb 2024

→ Construction of DFA for a lang consisting of strings ending with a particular substring

- Determine the min no. of states req. in DFA
- Decide the strings for which DFA will be constructed.

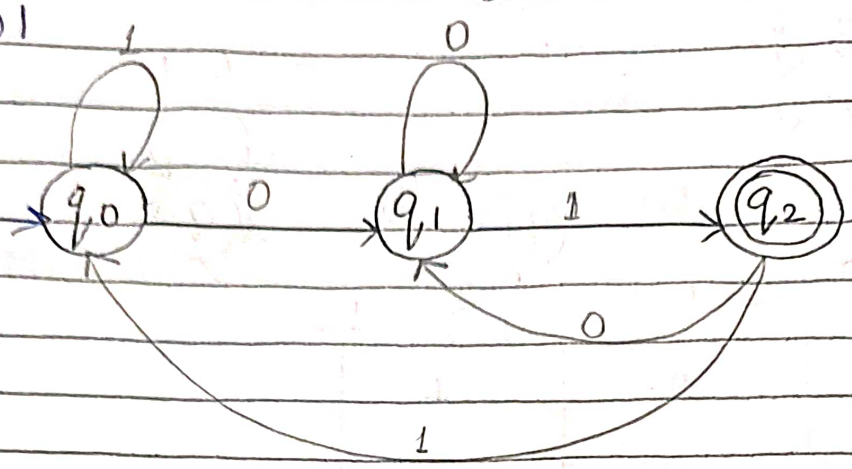
Problem 1) Draw a DFA for a lang accepting strings ending with '01' over input alphabet $\Sigma = \{0, 1\}$

→ Regular expression for given language = $(0+1)^*01$

- All strings of the language ends with substring '01'
- So length is = 2+1 ⇒ 3 (min no of states req)

01
01 01
01 01

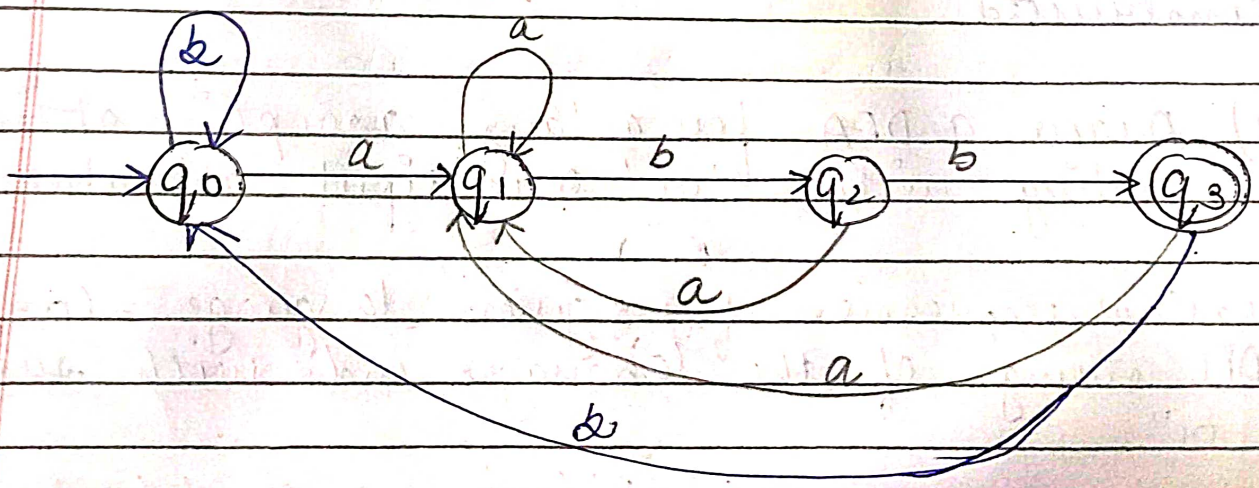
DFA



Q.2 Q Derive a DFA for lang ending with 'abb'
$\Sigma = \{a, b\}$

→ All strings of language ends with substring "abb".
→ So, string length is = 3
→ The min no of states req DFA = 3+1 = 4

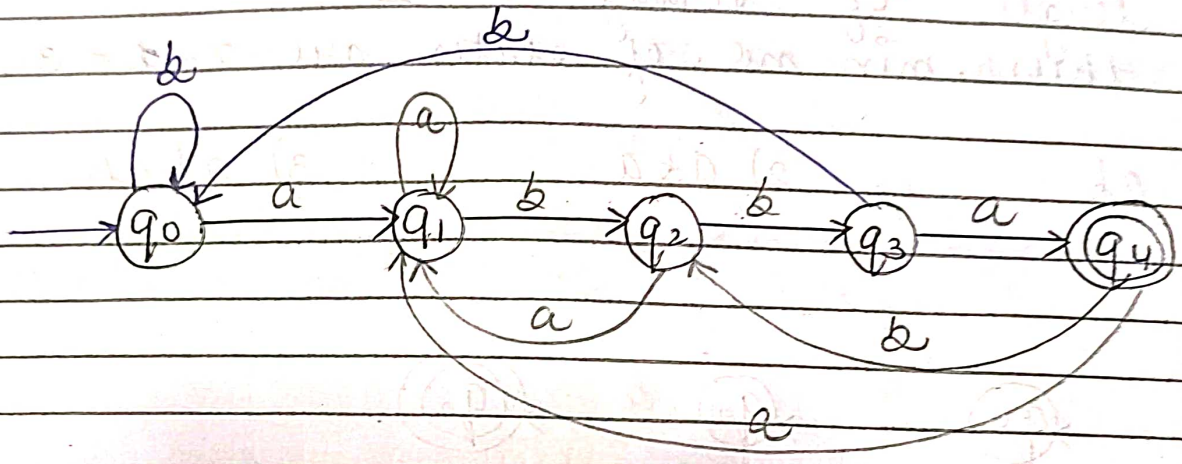
- abb
- aabb
- ababb
- abbabb



Q3 Q Draw a DFA for string ending with 'abba'

→ All the strings of lang ends with substring 'abba'
So, length of substring = 4
That minimized DFA are ⇒ 4+1 = 5

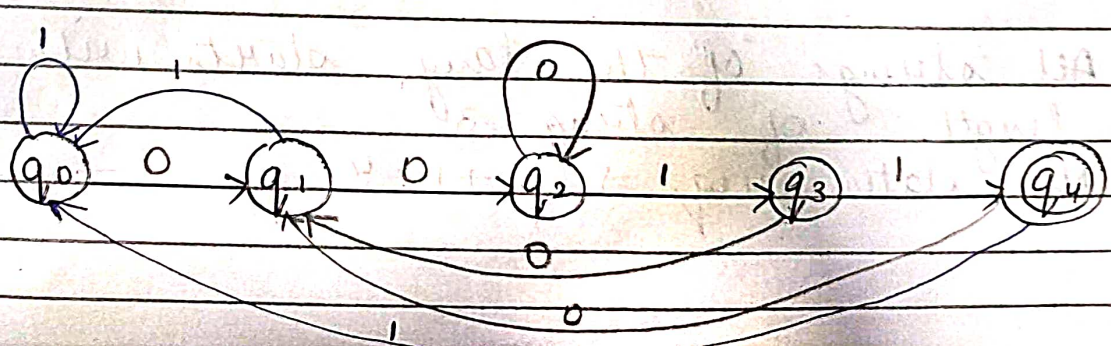
1) abba
2) aabba
3) ababba
4) abbabba
5) abbaabba



Ans4 Q Draw a DFA for string ending with '0011'

→ All strings of lang ends with substring '0011'
length is 4 so,
the minimized DFA are 4+1 = 5.

0011
00011
000011
0010011
00110011



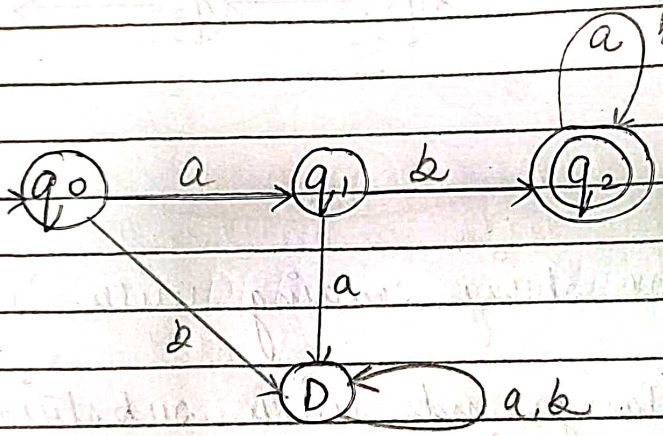
☆ Type 2 :- Problems

The construction of DFA for languages consisting of string 'starting' with a particular substring.

Q1) Draw a DFA for lang starting with 'ab' over input alphabet $\Sigma = \{a, b\}$

→ All strings of the lang starts with 'ab'
length of strings = 2
Thus, min no. of states are 2+1 = 3.

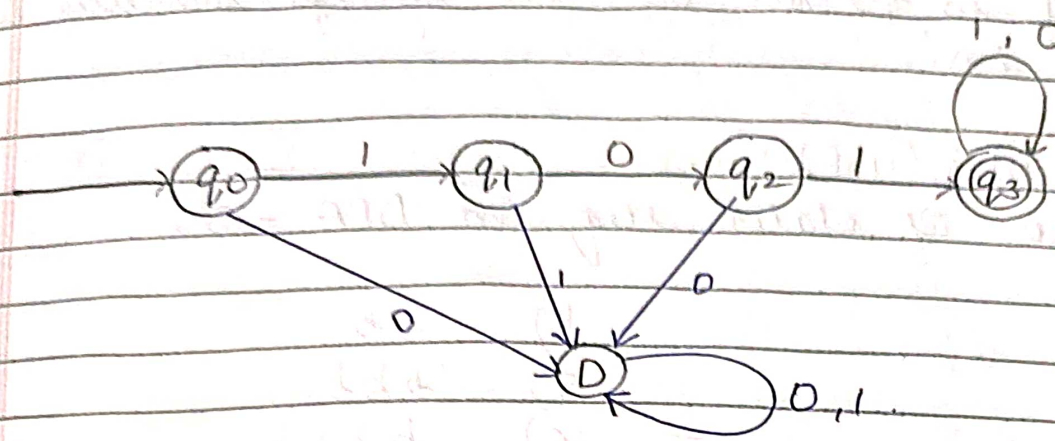
1) ab ✓ 2) aba 3) abab



Q2) Draw a DFA for lang stating with substring '101' over input alpha $\Sigma \{0, 1\}$

→ All strings of the lang starts with '101'
length of string = 3
Min states req ⇒ 3+1 = 4.

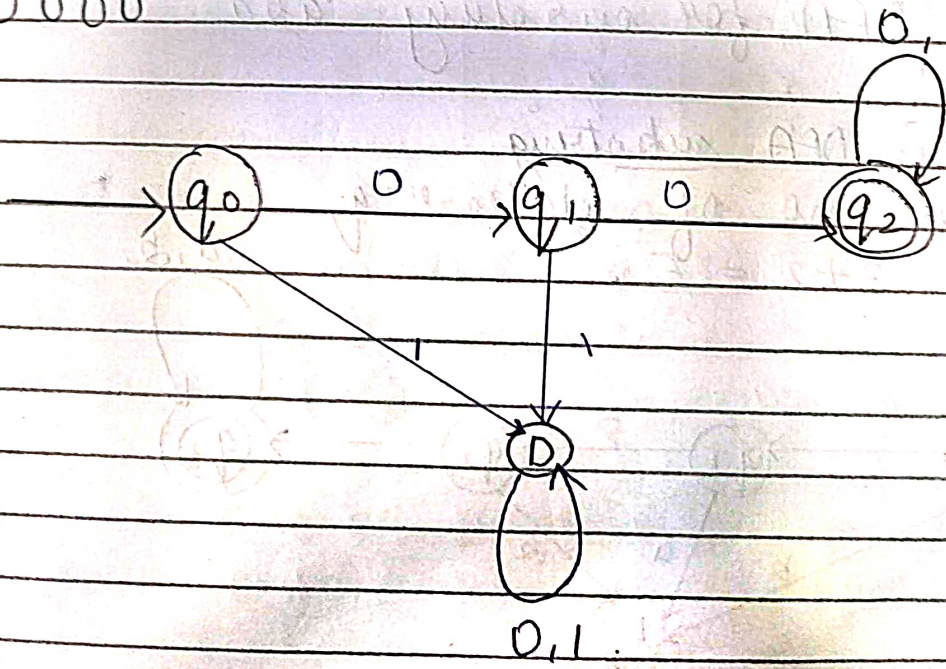
1) 101
2) 1011
3) 10110
4) 101101



Ques 4) Draw the DFA accepts lang L over alphabet $\Sigma = \{0, 1\}$ substring = "00"

→ All the strings of lang start with '00'
2 + 1 = 3 min 3 states of DFA

1) 00
2) 000
3) 0000

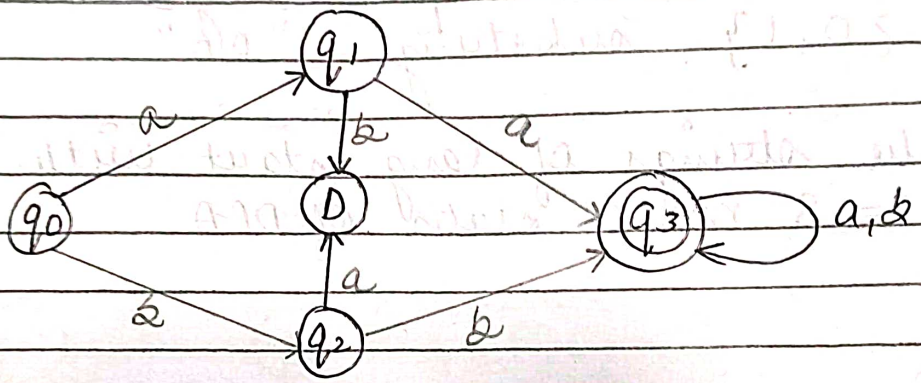


Ques5) Construct DFA that accept a lang. L over input alphabet $\Sigma = \{a, b\}$. L is the set of all strings starting with aa or 'bb'

→ $(aa + bb)(a+b)^*$

Min no. of states req in DFA = 5

1) aa
2) aaa
3) aaaa
4) bb
5) bbb
6) bbbb

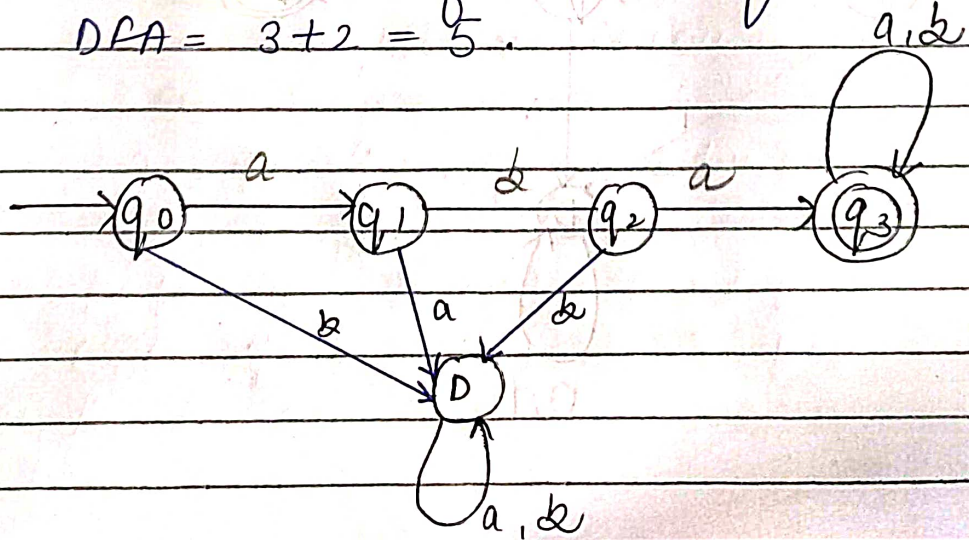


Ques6) Construct DFA for substring 'aba'

→ 3+1 = 4 min DFA substring

Thus, min no. of states req DFA = 3+2 = 5.

aba
abaa
abaab
baaba



2 Feb 24

→ Minimization of DFA.

The process of reducing a given DFA to its minimal form is called minimization of DFA.

- It consists of min state in DFA.
- The DFA is its minimal from is called as Minimal DFA..

The popular methods of minimizing of DFA.

- Equivalance Theorem
- Myhill Nerode Theorem (Table filling Method)

Step 1) Eliminate all the dead states and inaccessible states from the given DFA.

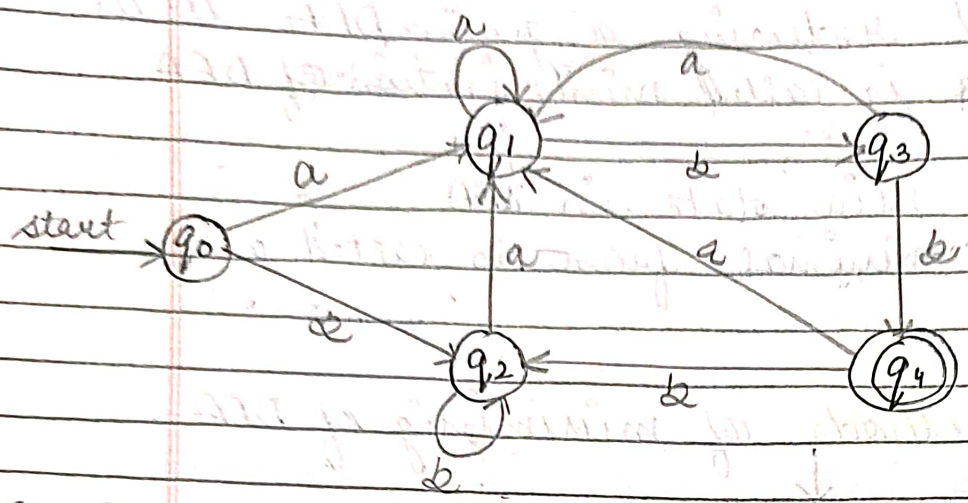
Dead state - All those non-final states which transit to itself for all inputs symbols Σ

Inaccessible state - All those states which can never be reached from initial state.

Step 2) Draw a state transition table for given DFA. Show, all transition states on all inputs Σ.

Step 3) Take a counter variable K and initialize it with 0. Divide Q into 2 sets called Partition P_0
Increment K by 1.

Ques1 Minimize the given DFA -



Step 1) The given DFA contain no dead states & it is accible

Step 2) Drawn a transition table

States	a	b
→ q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	* q_4
* q_4	q_1	q_2

Step 3) Now, use equivalance Theorem,

Non-final final

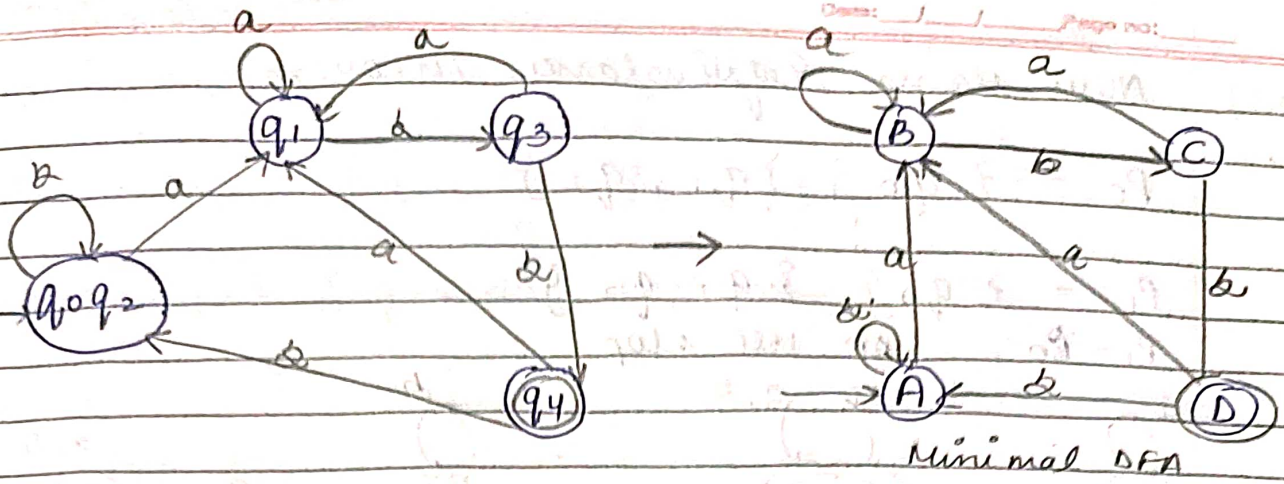
$P_0 = \{q_0, q_1, q_2, q_3\}\ \{q_4\}$

$P_1 = \{q_0, q_1, q_2\}\ \{q_3\}\ \{q_4\}$

$P_2 = \{q_0, q_2\}\ \{q_1\}\ \{q_3\}\ \{q_4\}$

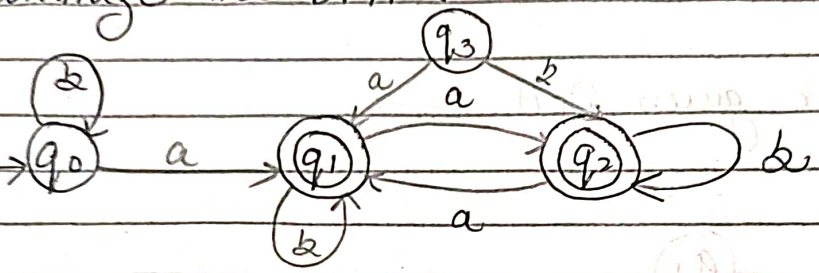
$P_3 = \{q_0, q_2\}\ \{q_1\}\ \{q_3\}\ \{q_4\}$

Since $P_2 = P_3$ so we stop.

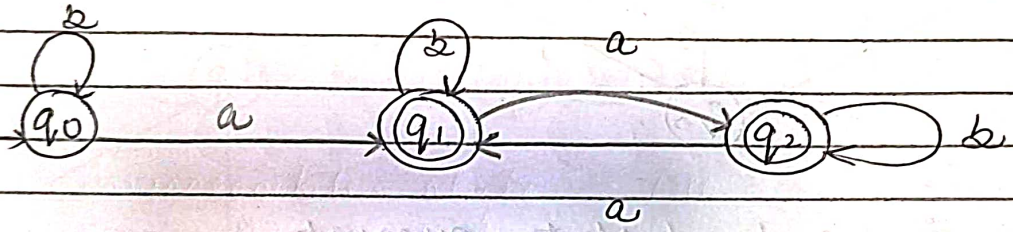


Minimal DFA

Ques 2) Minimize the DFA.



Step 1) Since state q_3 is inaccessible from initial state so, we eliminate it and its associated edges from



Step 2) Draw a state transition table

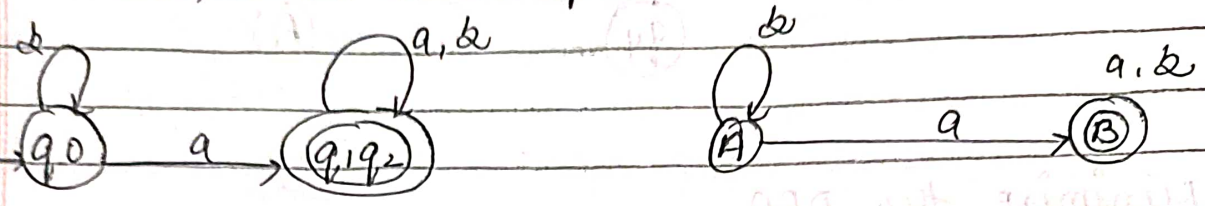
State	a	b
$\rightarrow q_0$	$*q_1$	q_0
$*q_1$	$*q_2$	$*q_1$
$*q_2$	$*q_1$	$*q_2$

Step 3) Now using Equivalance Theorem,

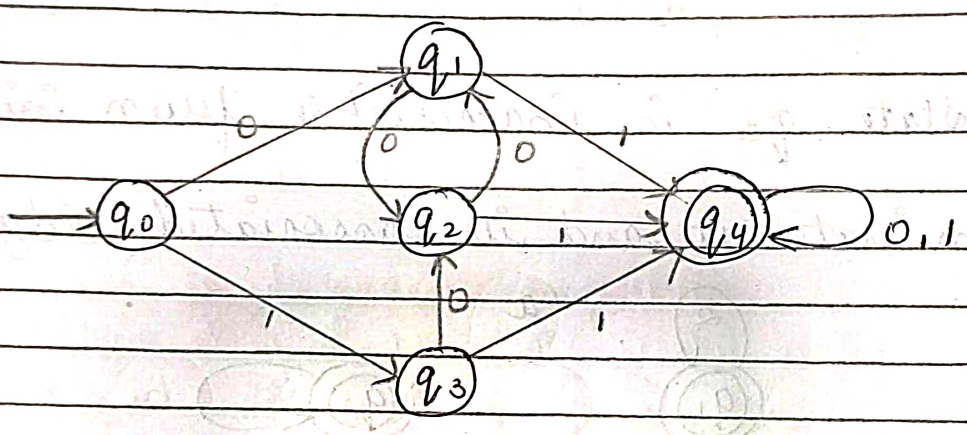
$P_0 = \{q_0\}\ \{q_1, q_2\}$

$P_1 = \{q_0\}\ \{q_1, q_2\}$

$P_1 = P_0$, so we stop.



Ques 3) Minimize the given DFA



Step 1) There is no dead state present

Step 2)

states	0	1
$\rightarrow q_0$	q_1	q_3
q_1	q_2	$*q_4$
q_2	q_1	$*q_4$
q_3	q_2	$*q_4$
$*q_4$	$*q_4$	$*q_4$

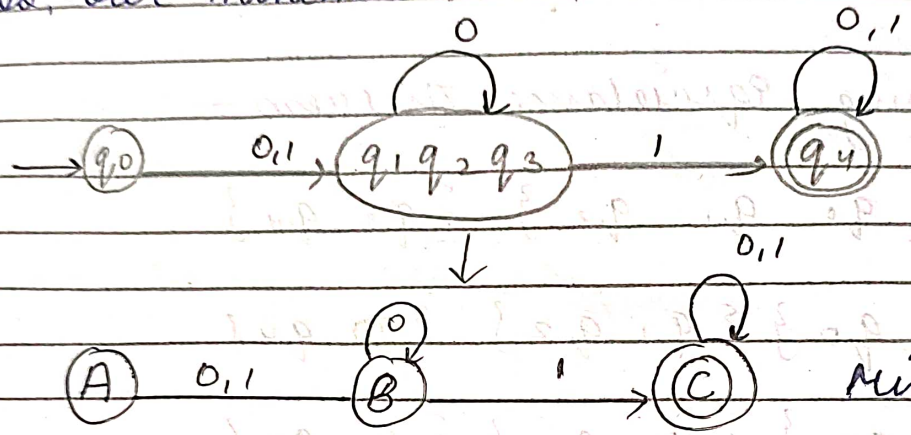
$P_0 = \{q_0\ q_1\ q_2\ q_3\}\ \{q_4\}$

$P_1 = \{q_0\}\ \{q_1\ q_2\ q_3\}\ \{q_4\}$

$P_2 = \{q_0\}\ \{q_1\ q_2\ q_3\}\ \{q_4\}$

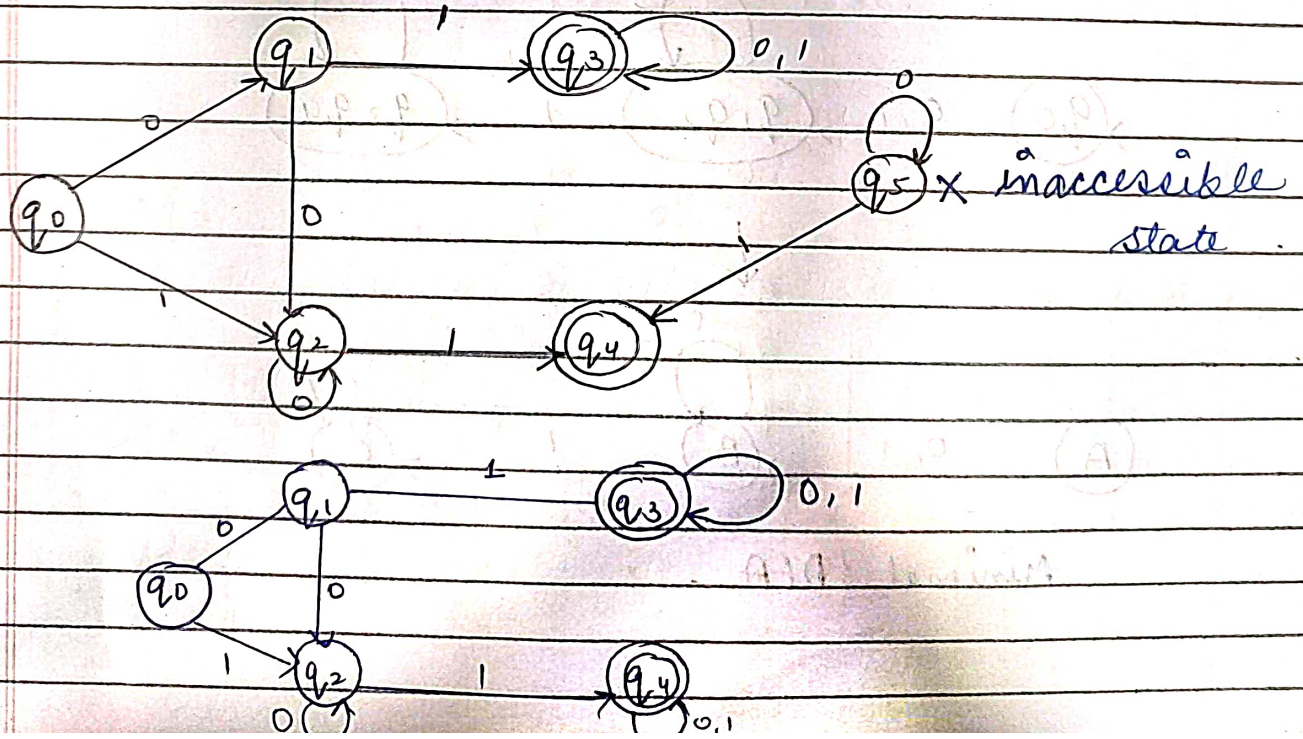
$P_2 = P_1$, we stop

So, our minimal DFA is :-



Minimal DFA.

Ques 4) Minimize the given DFA -



Step2)

State	0	1
→ q_0	q_1	q_2
q_1	q_2	$*q_3$
q_2	q_2	$*q_4$
$*q_3$	$*q_3$	$*q_3$
$*q_4$	$*q_4$	$*q_4$

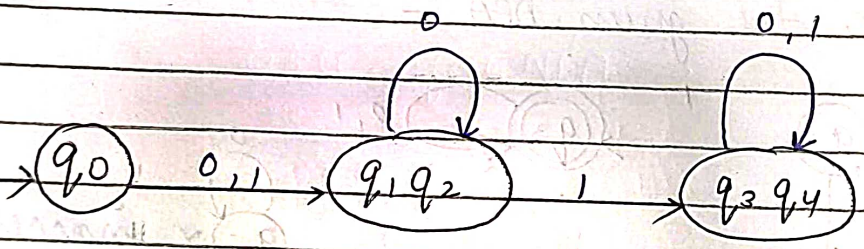
Step3) Now using Equivalance Theorem —

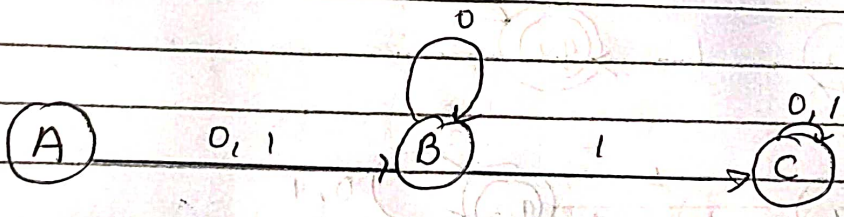
$P_0 = \{q_0\ q_1\ q_2\}\ \{q_3\ q_4\}$

$P_1 = \{q_0\}\ \{q_1\ q_2\}\ \{q_3\ q_4\}$

$P_2 = \{q_0\}\ \{q_1\ q_2\}\ \{q_3\ q_4\}$

$P_2 = P_1$, so we stop.





Minimal DFA.

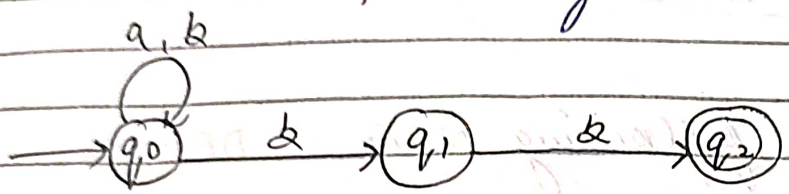
3 feb 2024)

↳ Non Deterministic finite Automata (NFA)

Converting NFA to DFA :

for some current state & input symbol, there exist more than one next output state.

Convert the following NFA to DFA.



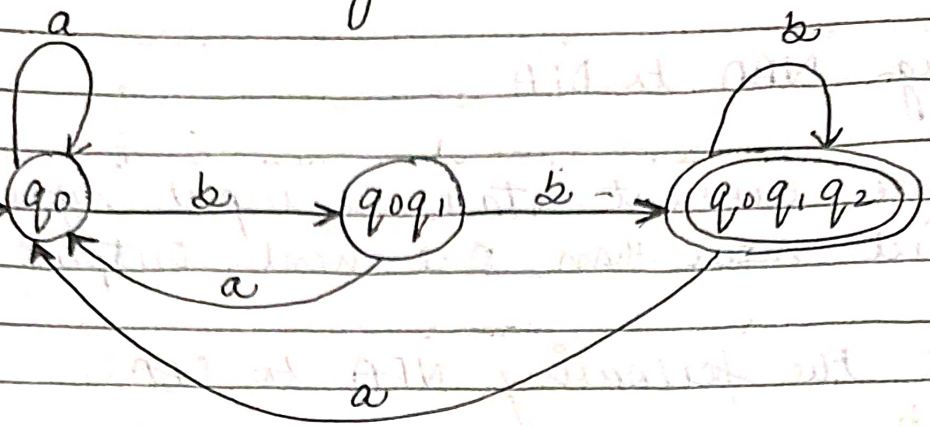
Sol": Transition table.

state	a	b	
$\rightarrow q_0$	q_0	q_1, q_0	} New state.
q_1	–	$*q_2$	
$*q_2$	–	–	

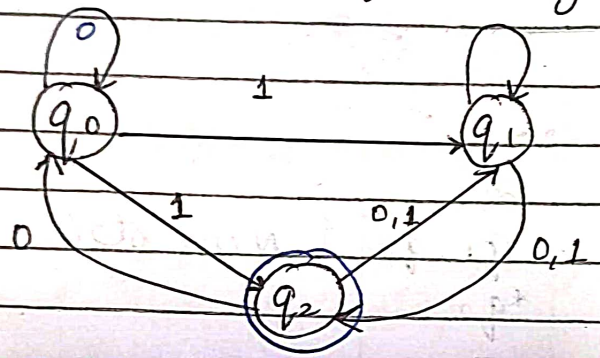
state	a	b	
$\rightarrow q_0$	q_0	$\{q_0 q_1\}$	
$\{q_0 q_1\}$	q_0	$\{q_0, q_1, q_2\}$	New state.

state	a	b	
$\rightarrow q_0$	q_0	$\{q_0 q_1\}$	
$\{q_0 q_1\}$	q_0	$*\{q_0 q_1 q_2\}$	
$*\{q_0 q_1 q_2\}$	q_0	$*\{q_0 q_1 q_2\}$	

New DFA may be drawn —



Ques2) Convert the following NFA → DFA



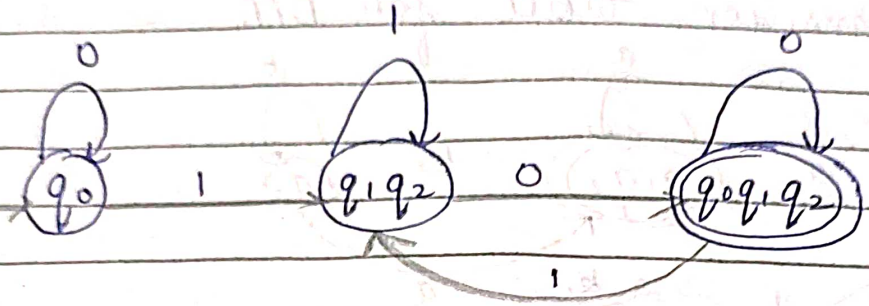
Transition table for NFA →

State	0	1
→ q_0	q_0	$q_1 * q_2$
q_1	$q_1, * q_2$	$* q_2$
* q_2	$q_0 q_1$	q_1

Let q_0 → be the new transition state.

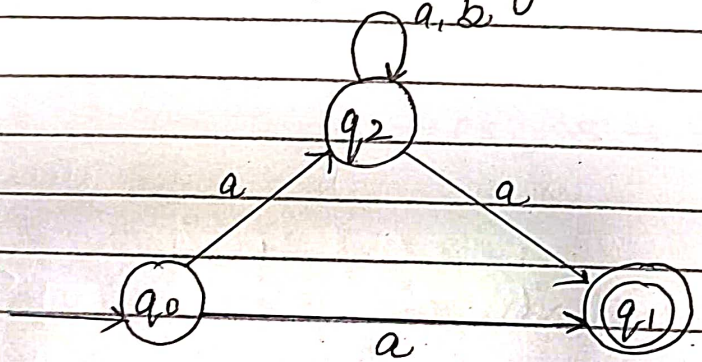
finally transition table for DFA →

State	0	1
→ q_0	q_0	$*\{q_1 q_2\}$
$*\{q_0 q_2\}$	$*\{q_0 q_1 q_2\}$	$*\{q_1 q_2\}$
$*\{q_0 q_1 q_2\}$	$*\{q_0 q_1 q_2\}$	$*\{q_1 q_2\}$



The final DFA

Ques) Convert the following NFA → DFA.

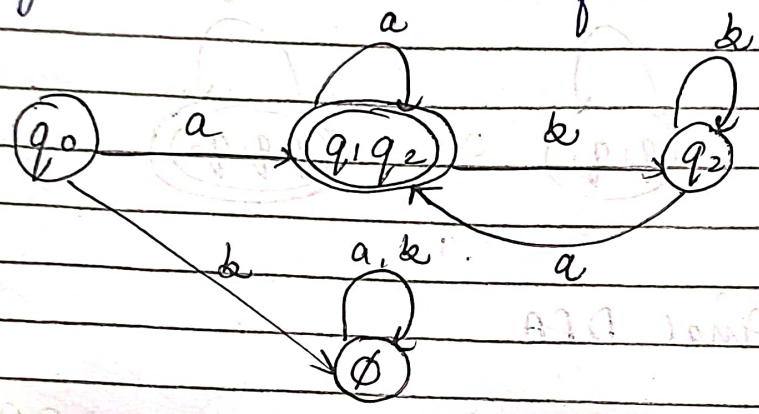


Transition table for NFA

States	a	b
→ q_0	$*q_1 q_2$	–
$*q_1$	–	–
q_2	$*q_1 q_2$	q_2

states	a	b
$\rightarrow q_0$	$*\{q_1 q_2\}$	ϕ
$*\{q_1 q_2\}$	$*\{q_1 q_2\}$	q_2
q_2	$*\{q_1 q_2\}$	q_2
ϕ	ϕ	ϕ

Finally transition table for DFA



8 Feb 24

Context free grammar

A context free grammar (CFG) is a 4 tuple set
$G = (V, T, P, S)$
where –

V – Finite non-empty set of variables / non-terminal
T – Finite set of terminal symbol
P – Finite non-empty set of production rules of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$
S – Start symbol

Ex 1) $V = \{S\}$
$T = \{a, b\}$
$P = \{S \rightarrow aSbS, S \rightarrow bSaS, S \rightarrow \epsilon\}$
$S = \{S\}$

Ex 2) $V = \{S\}$
$T = \{(,)\}$
$P = \{S \rightarrow SS, S \rightarrow (S), S \rightarrow \epsilon\}$
$S = \{S\}$

Why is used / Application –

- For defining Programming lang.
- For Parsing the program by constructing syntax tree.
- For translation of programming languages.
- For describing Arithmetic ins.
- For construction of compiler.

↳ Content free grammar →

* The language generated using content free grammar is called content free grammar.

* Properties –

- The content free language are closed under union.

- The content free language are closed under concatenation

- The context free language are closed under Kleen closure.

- The content free lang are not closed under intersection.

Remember –

* $L_1 \cup L_2$
* $L_1 . L_2$
* L_1^* and L_2^*

} L_1 & L_2 are content free lan

* $L_1 \cap L_2$
* L_1' and L_2'

} It is not a content free language

↳ Ambiguity in content free grammar –
A grammar is said to be ambiguous if for a given string generated by the grammar exist.

- more than one left most & Right most derivation
- or more than one parse tree.

↳ Parse Tree

- The process of deriving a string known as derivation
- The geomatrical representation of a derivation is called parse tree or derivation tree.

Types of Derivation

- Leftmost
- Rightmost

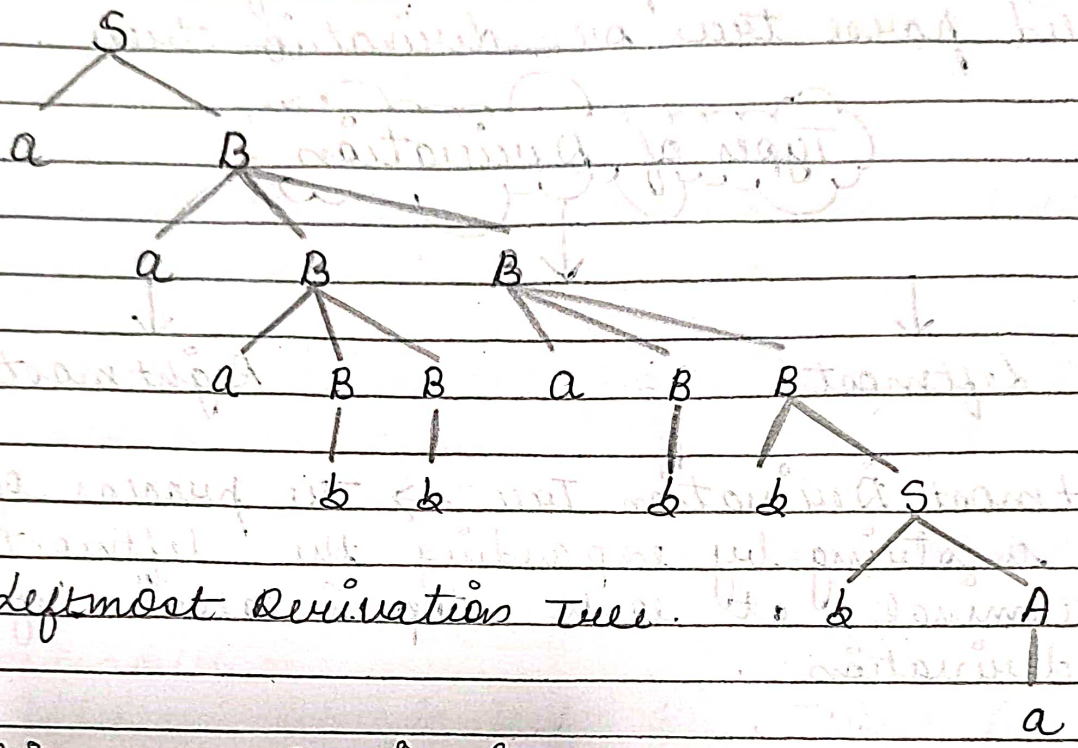
1) Leftmost Derivation Tree → The process of deriving a string by expanding the leftmost non-terminal at each step is called leftmost derivation.

Eg →
$$\left.\begin{aligned} S &\to aB \mid bA \\ A &\to aS \mid bAA \mid a \\ B &\to bS \mid aBB \mid b \end{aligned}\right\} \text{Unambiguous grammar.}$$

Let us consider a string → w ⇒ aaabbabbba.
Now let us derive the string w.

$S \to aB$ (B → aBB)
$aaBB$ (B → aBB)
$aaaBBB$ (B → b)
$aaabBB$ (B → b)
$aaabbB$

aaabbaBB (B → aBB)
aaabbabB (B → b)
aaabbabbS (B → bS)
aaabbabbbA (S → bA)
aaabbabbbba (A → a)



Leftmost Derivation Tree.

2) Rightmost Derivation :-

- The process of deriving a string by expanding the rightmost non-terminal at each step

- The geometrical representation of RM derivation

Eg → S → aB / bA
B → aS / bAA / a
B → bS / aBB / b

$W = aaabbabbba$

$S \rightarrow aB$

$aaBB$ ($B \rightarrow aBB$)

$aaBaBB$ ($B \rightarrow aBB$)

$aaBaBbS$ ($B \rightarrow bS$)

$aaBaBbbA$ ($S \rightarrow bA$)

$aaBaBbba$ ($A \rightarrow a$)

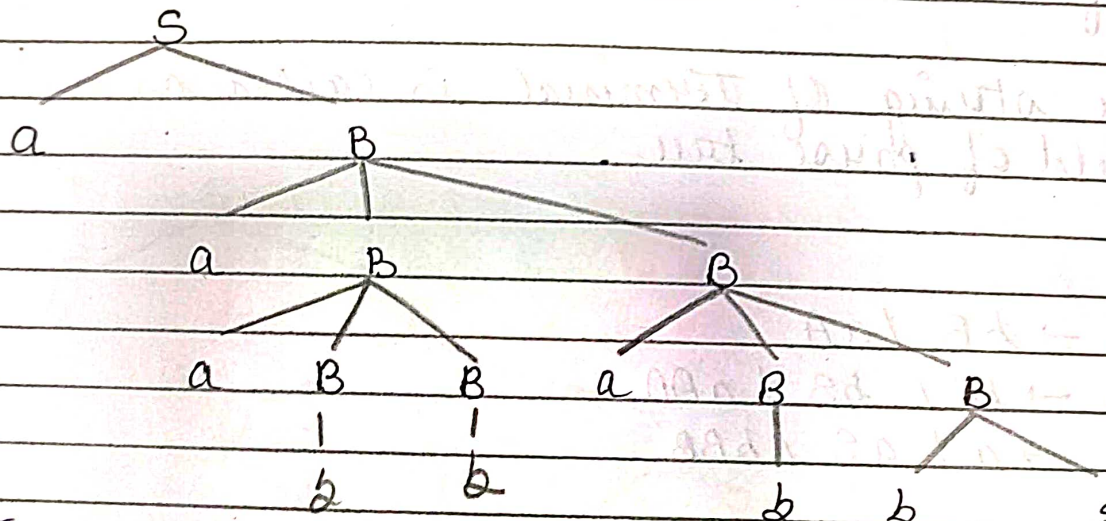
$aaBabbba$ ($B \rightarrow b$)

$aaaBBabbba$ ($B \rightarrow aBB$)

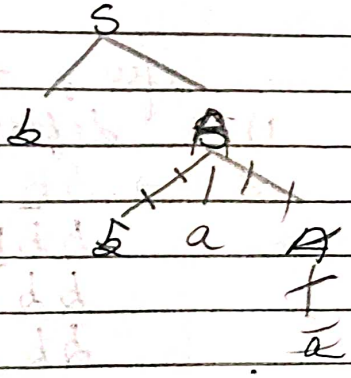
$aaababbba$ ($B \rightarrow b$)

$aaabbabbba$ ($B \rightarrow b$)

Rightmost Derivation tree.



* For Unambiguous grammar
LD & RD represent same parse tree.



* For Ambiguous grammar
LD & RD represent diff parse Tree.

LD ≡ RD

↳ Properties of Parse Tree :-

* Root node of a parse tree is the start symbol of grammar.
* Each leaf node represent terminal symbol
* Each interior node " non-terminal

↳ Yield of Parse tree :-

* Concatenating the leaves of parse tree from the left produces a string of terminal.
* The string of terminal is called as yield of parse tree.

Ques 2) $S \rightarrow bB \mid aA$
$A \rightarrow b \mid bS \mid aAA$
$B \rightarrow a \mid aS \mid bBB$

for, the string $w = bbaababa$

1) Leftmost Derivation

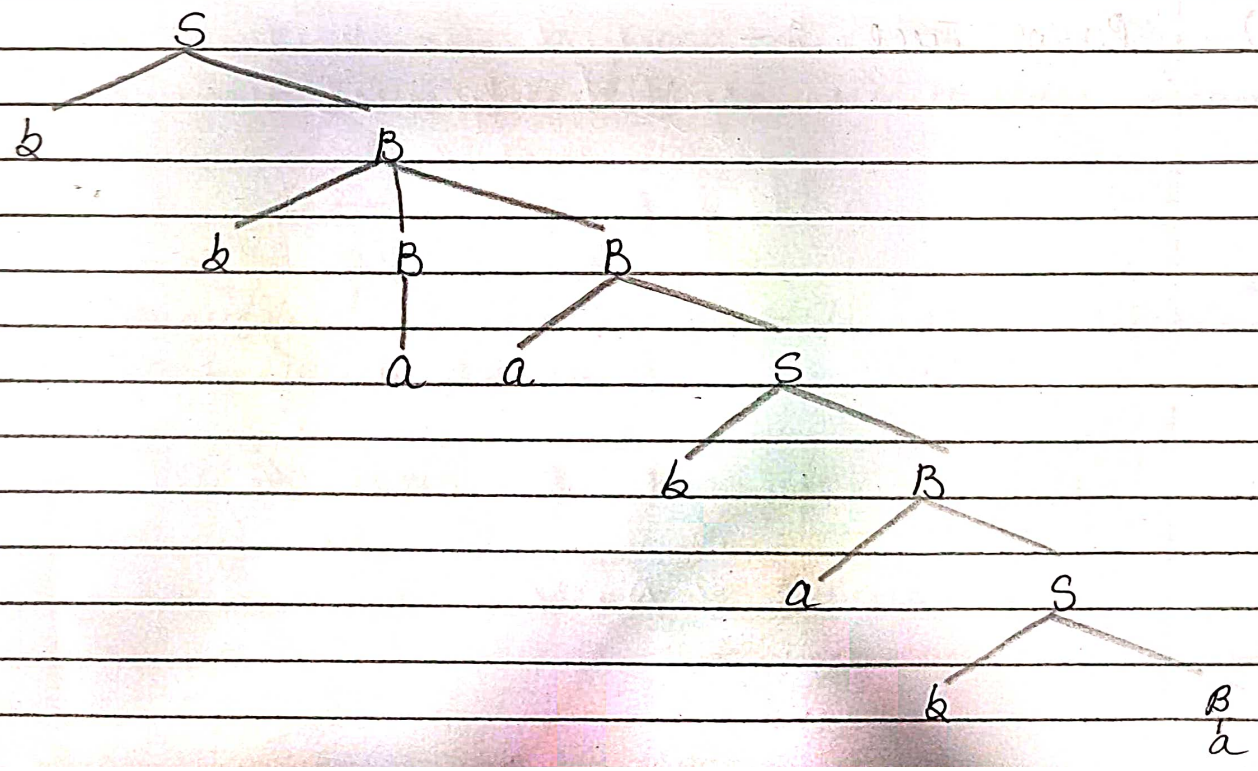
$S \rightarrow \underline{bB}$
$bb\underline{BB}$ $(B \rightarrow bBB)$
$bba\underline{B}$ $(B \rightarrow a)$
$bbaa\underline{S}$ $(B \rightarrow aS)$

bbaabB (S → bB)
bbaabaS (B → aS)
bbaabaBB (S → bB)
bbaababa (B → a)

2) Rightmost Derivation

S → bB
bbBB (B → bBB)
bbBaS (B → aS)
bbBabB (S → bB)
bbBabaS (B → aS)
bbBababB (S → bB)
bbBababa (B → a)
bbaababa (B → a)

3) Parse Tree -



Ques3) Consider the grammar —

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

$w = 00101$

1) Leftmost Derivation

$S \rightarrow \underline{A}1B$ $(A \rightarrow 0A)$
$0\underline{A}1B$ $(A \rightarrow 0A)$
$00\underline{A}1B$ $(A \rightarrow 0A)$
$001\underline{B}$ $(A \rightarrow \epsilon)$
$0010\underline{B}$ $(B \rightarrow 0B)$
$00101\underline{B}$ $(B \rightarrow 1B)$
00101 $(B \rightarrow \epsilon)$

2) Rightmost Derivation

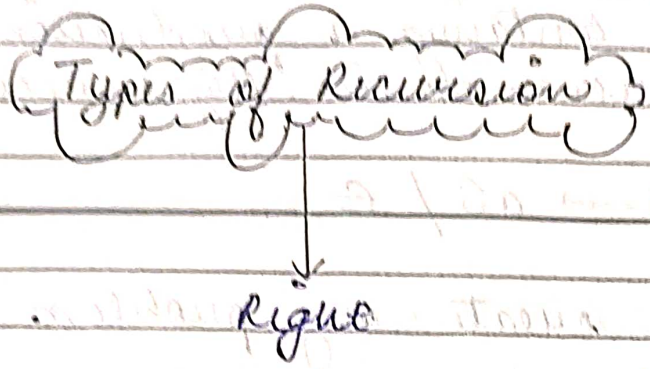
$S \rightarrow A1\underline{B}$ $(B \rightarrow 0B)$
$A1\underline{0B}$ $(B \rightarrow 0B)$
$A\underline{10B}$ $(B \rightarrow 1B)$
$\underline{A}101$ $(B \rightarrow \epsilon)$
$0\underline{A}101$ $(A \rightarrow 0A)$
$00A101$ $(A \rightarrow 0A)$
00101 $(A \rightarrow \epsilon)$

3) Parse Tree :-

Page 24

Recursion

Recursion can be classified into following three types —



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.

A grammar containing a production having left recursion is called Left recursive grammar.

Eg → $S \rightarrow Sa \mid \epsilon$

It is considered to be a problematic situation for Top - Down parsers.

LRG → $A \rightarrow A\alpha \mid \beta$ | $A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$ | Right recursive grammar.

2) Right Recursion —

- If the rightmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having right recursion is Right recursive grammar.

eg → S → aS / ∈

- It does not create any problem for top-down parsers.
- ∴ No need of eliminating RR from grammar.

3) General Recursion —

The Recursion which is neither left nor right is called General recursion.

eg → aSb / ∈

Problems —

Ques1) Consider the fell grammar & eliminate LR.

A → ABd / Aa / a
B → Be / b

$A \rightarrow A\alpha \mid B$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

→ The grammar after eliminating left recursion

$A \rightarrow aA'$

$A' \rightarrow BdA' \mid aA' \mid \epsilon$

$\beta = a$

$\alpha = Bd, a$

$B \rightarrow \ell B'$

$B' \rightarrow eB' \mid \epsilon$

Ques 2) $E \rightarrow E+E \mid EXE \mid a$ (α = +E, XE; β = a)

→ $E \rightarrow aE'$

$E' \rightarrow +EE' \mid XEE' \mid \epsilon$

Ques 3) $E \rightarrow E+T \mid T$

$T \rightarrow TXF \mid F$

$F \rightarrow id$

→ $E \rightarrow TE'$

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

α = +T, XF

β = T, F

$T \rightarrow FT'$

$T' \rightarrow XFT' \mid \epsilon$

$F \rightarrow id$

Ques 4) $S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$ (α = ,S; β = S)

→ $S \rightarrow (L) \mid a$

$L \rightarrow SL'$

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

Date: / / Page no:

Ques 5) $S \rightarrow \underset{\alpha}{S0S1S} \mid \underset{\beta}{01}$

$\alpha = 0S1S$
$\beta = 01$

→ $S \rightarrow 01B'$
$S' \rightarrow 0S1SAS' \mid \epsilon$

Ques 6) $S \rightarrow A$
$A \rightarrow \underset{\alpha}{Ad} \mid \underset{\alpha}{Ae} \mid \underset{\beta}{aB} \mid \underset{\beta}{ac}$
$B \rightarrow bBc \mid f$

→ $S \rightarrow A$
$A \rightarrow aBA' \mid acA'$
$A' \rightarrow dA' \mid eA' \mid \epsilon$
$B \rightarrow bBc \mid f$

$\alpha = d, e$
$\beta = aB, ac$

Ques 7) $A \rightarrow \underset{\alpha}{AA\alpha} \mid \underset{\beta}{\beta}$

→ $A \rightarrow \beta A'$
$A' \rightarrow A\alpha A' \mid \epsilon$
$A'' \rightarrow \alpha A'' \alpha \mid \epsilon$

$\beta = \beta$
$\alpha = A\alpha$

Ques 8) $A \rightarrow \underset{\beta}{Ba} \mid \underset{\alpha}{Aa} \mid \underset{\beta}{c}$
$B \rightarrow Bb \mid Ab \mid d$

→ This is case of indirect left recursion

Ques 8) First let us eliminate LR from $A \rightarrow Ba \mid Aa \mid c$
We get,
$A \rightarrow BaA' \mid cA'$
$A' \rightarrow aA' \mid \epsilon$
Now, given grammar becomes —

$A \rightarrow BaA' \mid CA'$ — (1)

$A' \rightarrow aA' \mid \epsilon$

$B \rightarrow Bb \mid Ab \mid d$ — (3)

step 2) Substituting production of A in $B \rightarrow Ab$, we get

$A \rightarrow BaA' \mid CA'$

$A' \rightarrow aA' \mid \epsilon$

$B \rightarrow Bb \mid BaA'b \mid CA'b \mid d$

(Bb: α; $BaA'b$: α; $CA'b$: β; d: β)

step 3) Now eliminating LR from production of B, we get

$A \rightarrow BaA' \mid CA'$

$A' \rightarrow aA' \mid \epsilon$

$B' \rightarrow CA'bB' \mid dB'$

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

Ques 9) Eliminate LR from grammar —

$X \rightarrow XSb \mid Sa \mid b$

$S \rightarrow Sb \mid Xa \mid a$

(Xb: α; Sa: β; b: β; Sb: α; Xa: β; a: β)

→ This is a case of indirect left recursion.

step 1) Let us eliminate left recursion from $X \rightarrow XSb \mid Sa$. Eliminate left recursion from here —

$X \rightarrow SaX' \mid bX'$

$X' \rightarrow SbX' \mid \epsilon$

Now, given grammar becomes —

$X \rightarrow SaX' \mid bX'$ — ①
$X' \rightarrow SbX' \mid \epsilon$
$S \rightarrow Sb \mid Xa \mid a$ — ③

Step2) Substituting ① in ③ we get —

$X \rightarrow SaX' \mid bX'$
$X' \rightarrow SbX' \mid \epsilon$
$S \rightarrow Sb \mid SaX'a \mid bX'a \mid a$

Step3) Now eliminate left recursion from production S, we get.

$X \rightarrow SaX' \mid bX'$
$X' \rightarrow SbX' \mid \epsilon$
$S \rightarrow bX'aS' \mid aS'$
$S' \rightarrow bS' \mid aX'aS' \mid \epsilon$.

Ques10) $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$
① → ②

→ $S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$.

10 Feb 24

↳ Left factoring → Grammar with comman Prefixes

If RHS of more than one productions starts with the same symbol, such grammar is called.

Grammar with comman Prefixes -

Eg → $A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$

- This kind of grammar creates a problematic situation for Top Down parser.
- Top Down parser can not decide which production must be chosen to parse the string in hand.

* Left factoring → Left factoring is a process by which grammar with comman prefixes is transformed, to make it useful for top down parser.

How? - We make one production for each comman prefixes.
- The comman prefix may be a terminal or a non-terminal or a combination.
- Rest the derivation is added by new production.
- The grammar obtained after the process of LF

Example →

$A \to a\alpha_1 / a\alpha_2 / a\alpha_3$ ⟹ LF ⟹ $A \to aA'$, $A' \to \alpha_1 / \alpha_2 / \alpha_3$

Grammar with common prefix — Left factored

Ques 1) $S \to \underset{\alpha}{iEtS} / \underset{\alpha}{iEtSeS} / \underset{\beta}{a}$

$E \to b$

→ The left factored grammar is —

$S \to iEtSS' / a$ $\quad \alpha = iEtS$

$S' \to eS / \epsilon$

$E \to b.$

Ques 2) $A \to aAB / aBC / aAC$

→ $A \to aA'$ $\quad \{\alpha = a\}$

$A' \to AB / Bc / Ac$

Again, this grammar with common prefixes.

$A \to aA'$

$A' \to AD / Bc$

$D \to B / c$

This is left factored grammar.

Ques 3) $S \to bSSaaS / bSSaSb / bSb / a$

→ $S \to bSS' \mid a$ $\{\alpha = bS\}$

$S' \to SaaS \mid SaSb \mid b$

↳ $S \to bSS' \mid a$

$S' \to SaAS'' \mid b$ $\{\alpha = Sa\}$

$A \to aS \mid Sb$ $(S'' = A)$

Ques 4) $S \to aSSbS \mid aSaSb \mid abb \mid b$

→ $S \to aS' \mid b$

$S' \to SSbS \mid SaSb \mid bb$

$S \to aS' \mid b$

$S' \to SA \mid bb$

$A \to SbS \mid aSb$

Ques 5) $S \to a \mid ab \mid abc \mid abcd$

→ $S \to aS'$

$S' \to b \mid bc \mid bcd \mid \epsilon$

$S \to aS'$

$S' \to bA \mid \epsilon$

$A \to c \mid cd \mid \epsilon$

$S \to aS'$

$S' \to bA \mid \epsilon$

$A \to cB \mid \epsilon$

$B \to d \mid \epsilon$

Ques 6) $S \rightarrow aAd \mid aB$
$A \rightarrow a \mid ab$
$B \rightarrow ccd \mid ddc$

→ $S \rightarrow aS'$
$S' \rightarrow Ad \mid B$
$A \rightarrow aA'$
$A' \rightarrow b \mid \epsilon$
$B \rightarrow ccd \mid ddc$

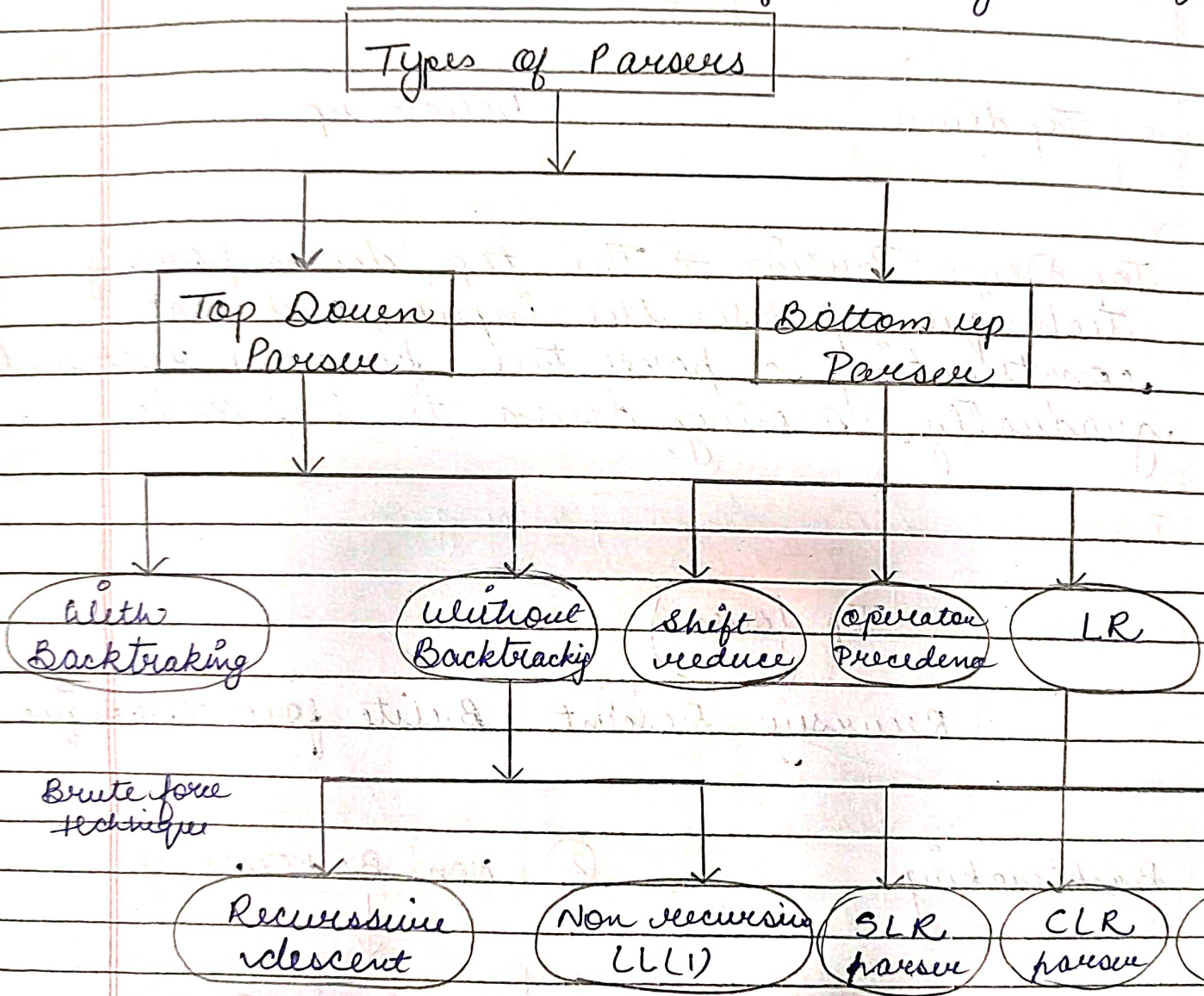
- Causes such as left recursion, common pref

By adding Precedence rule

By fixing the grammar

Parsing Technique in compiler design..

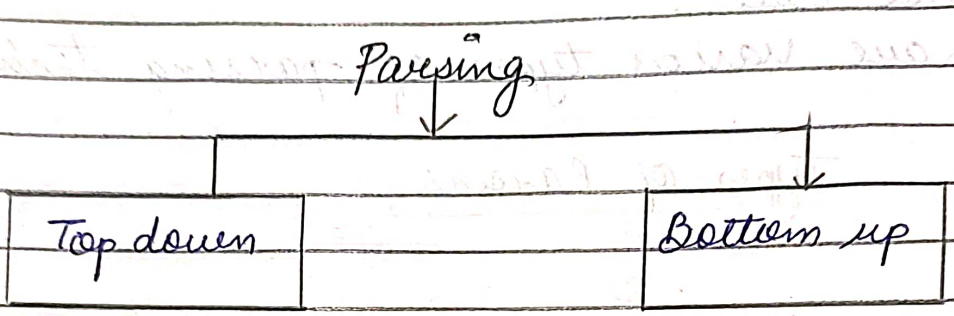
- Parsing is known as syntax analysis.
- It contains arranging the tokens as source code into → grammatical phases are defined by parse tree.
- There are various types of parsing techniques.



SLR Simple LR parser
CLR Canonical LR parser
LALR Look ahead LR parser

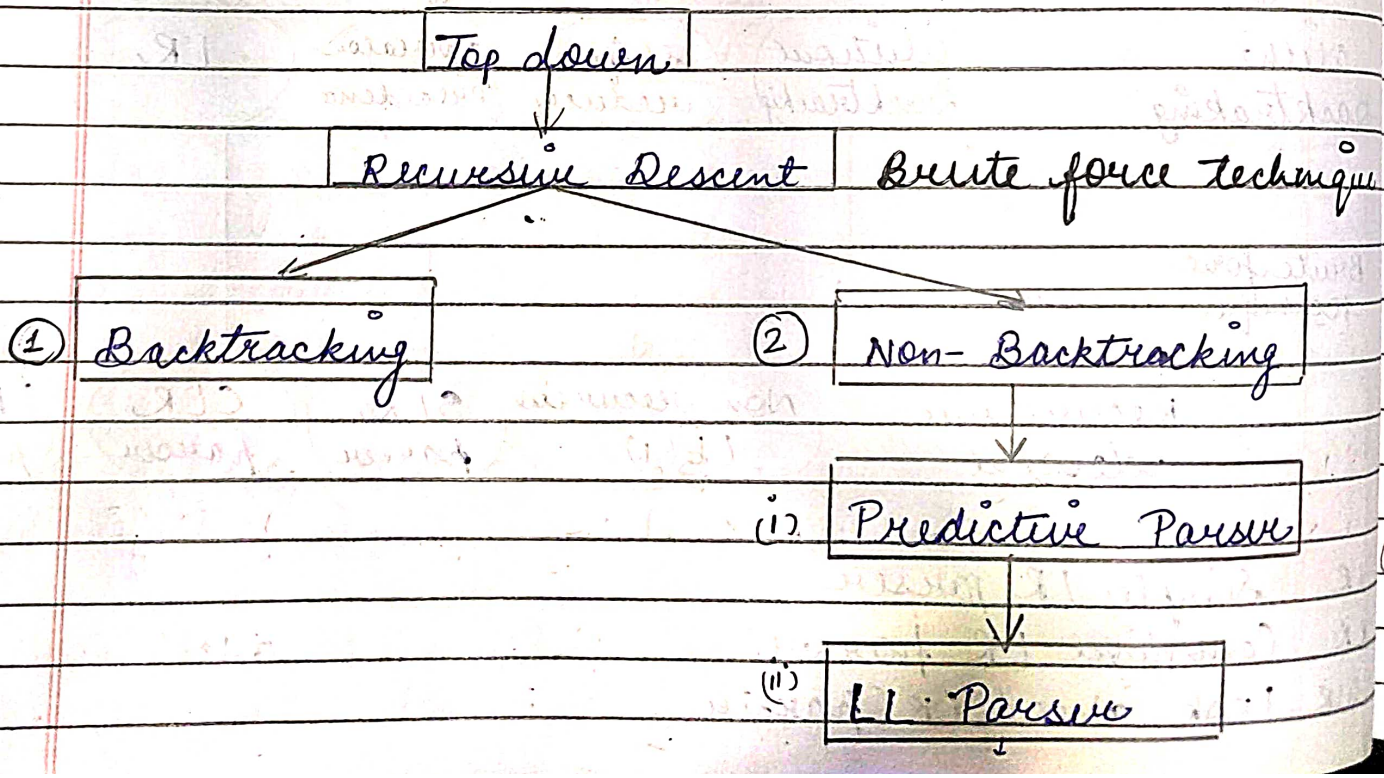
Parser

Parsing can be defined as top-down or bottom up based on how the parse tree is constructed.



1) Top Down Parsing → The top down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to leaf node.

Types of top-down parsing =



(i) Recursive descent Parsing

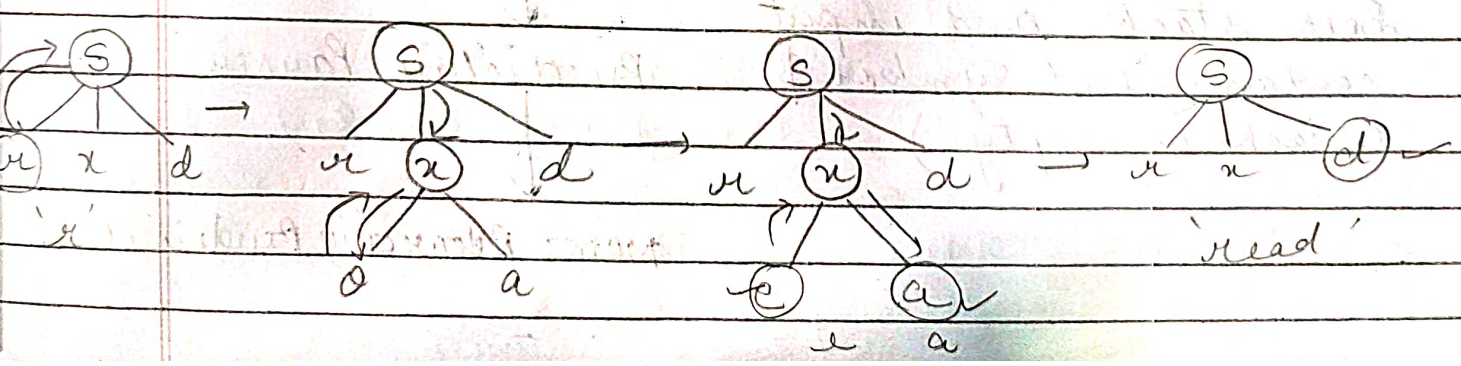
- Recursive descent is a top-down parsing technique that constructs the parse tree from top and I/P read from Left → Right
- Uses procedures for every terminal & non-terminals
- It recursively parses the I/P to make parse tree, which may or may not require Backtracking.
- A form of recursive descent parsing that does not require back-tracking is known as predictive parsing.

(ii) Backtracking → Top-Down parsers starts from root node (start symbol) and match the input strings against the production rules, to replace them (if matched)

Take the eg of CFG:-

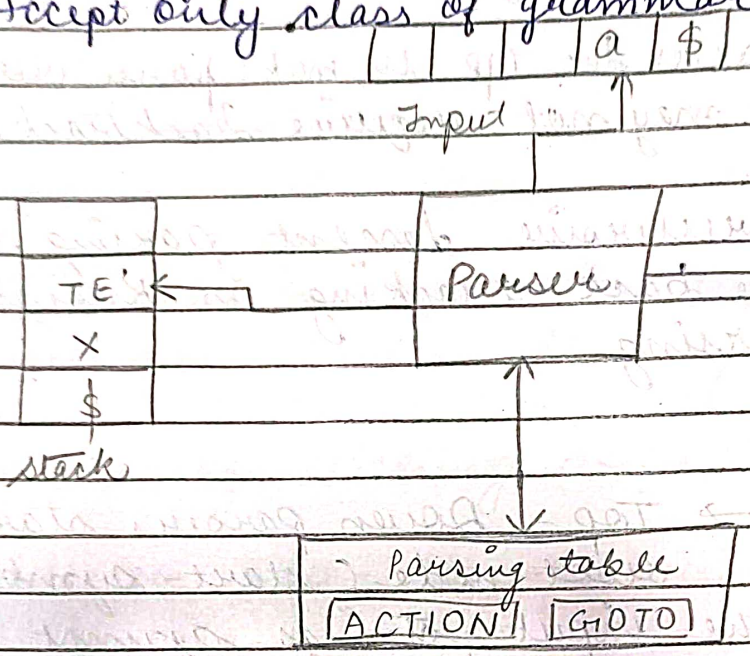
S → rXd | rzd
X → oa | ea
z → ai

{ Input string → read }



(2) Non-backtracking / Predictive Parser -

- Predictive parser is recursive descent parser which has capabilities to predict.
- It does not suffer from backtracking.
- It uses a look ahead pointer (points next input symbol.
- Accept only class of grammar called LL(K) gramm



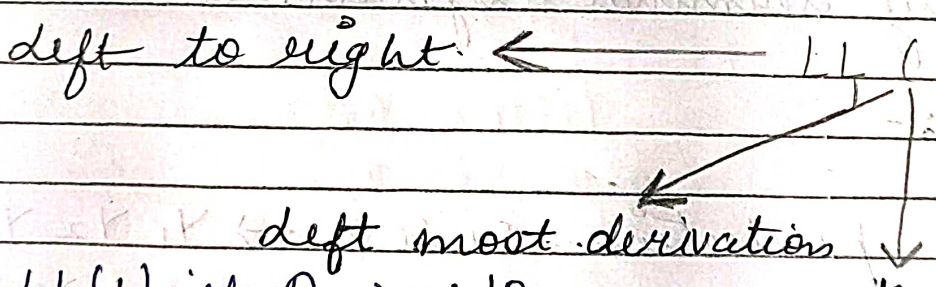
- Uses a stack and a parsing table to parser the input and generate parse tree
- Both stack and input contains end symbol $ (stack is empty)

Top-Bottom Parser
↓ Remove left R. / Left factored gram
Recursive Descent
↓ Remove Backtrack
Predictive Parser
↓ Use table / Remove recursi
Non-Recursive Predictive Pars

- The parser may have more than one production to choose from for a single instance of input,
- Each step has at most one production to choo

(i) LL Parser

- An LL parser accepts LL grammar.
- It is a subset of context free grammar but with some restrictions.
- It can be implemented by means of both algorithms namely, recursive descent & table driven.
- LL parser is denoted as LL(K), parsing the input from L → R (first L) and second L is for left most derivation and K itself represent (No of look ahe
- Generally (K = 1) so, LL(K) = LL(1)



G is LL(1) if A → α | β

- for no terminal both α & β derive string begin with a
- At most α & β can derive empty string
- If β → t then α doesn't derive any string beginning with a terminal in follow(A).

★ First and follow :-

First and follow set are needed so that the parser can properly apply the needed production rule at the correct position.

(a) First function :- First (α) is a set of terminal symbols begin in string derived from α.

Rules :- For calculating first function

01) For a production rule $X \to \epsilon$

$$First(X) = \{\epsilon\}$$

02) Rule :-

For any terminal symbol 'a'

03) Rule :-

For a production rule $X \to Y_1 Y_2 Y_3$

Calculating First (X)

- If $\epsilon \notin$ First (Y_1), then First (X) = First (Y_1)
- If $\epsilon \in$ First (Y_1), then First (X) = { First (Y_1) $- \epsilon$ } $\cup$ First ($Y_2 Y_3$)

Can make expansion for any production $X \to Y_1 Y_2 Y_3 \ldots Y_n$

Calculating first ($Y_2 Y_3$)

- If $\epsilon \notin$ first (Y_2), then first ($Y_2 Y_3$) = first (Y_2)
- If $\epsilon \in$ first (Y_2), then first ($Y_2 Y_3$) = { first (Y_2) $- \epsilon$ } $\cup$ First (Y_3).

∴ We can make for any production $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n$

Rules for calculating follow function.

01) For the start symbol S, place $ in follow(S)

02) For any production rule $A \rightarrow \alpha B$,
Follow (B) = Follow (A)

03) For any production rule $A \rightarrow \alpha B \beta$

- If $\epsilon \notin$ First (β), then Follow (B) = First (β)
- If $\epsilon \in$ First (β), then Follow (B) = { First (β) $- \epsilon$ } $\cup$ Follow A.

NOTES →
- ϵ may appear in first function of non-terminal
- ϵ will never appear in the follow function of a non terminal.
- Before cal F & F eliminates LR from the grammar
- We cal. the follow function of non-terminal by looking where it is present on RHS.

Practice problems on First and Follow

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

LHS = RHS
Follow First

→ First (S) = {a}
First (B) = {c}
First (C) = {b, ϵ}
First (D) = {First (E) − ϵ} ∪ {First F}
= {g, f, ϵ}
First (E) = {g, ϵ}
First (F) = {f, ϵ}

→ Follow (S) = {$}
S → aBDh Follow (B) = {First (D) − ϵ} ∪ First (h)
= {g, f, h}

B → cC Follow (C) = Follow (B)
= {g, f, h}

S → aBDh Follow (D) = First (h) = {h}

D → EF Follow (E) = {First (F) − ϵ} ∪ {Follow D}
= {f, h}

Follow (F) = Follow {D}
{h}

$\{ A \rightarrow A\alpha + \beta . \}$

2) $S \rightarrow A$
$A \rightarrow aB^{\beta} / A\alpha$
$B \rightarrow b$
$C \rightarrow g$

→ We have,
- The given grammar is left recursive.
- So, we remove first left Recursion.

$S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' / \epsilon$
$B \rightarrow b$
$C \rightarrow g$

First function →

First (S) = First (A) = {a}
First (A) = {a}
First (A') = {d, ∈}
First (B) = {b}
First (C) = {g}

Follow Follow (S) = {$}
Follow (A) = Follow (S) = {$}
Follow (A') = Follow (A) = {$}

Follow (B) = {First (A') − ∈} − Follow (A)
= {d, $}
Follow (C) = NA

3) $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow , SL' \mid \epsilon$

→ First(S) = { (, a }

First(L) = First(S) = { (, a }

First(L') = { , , ϵ }

Follow :-

→ Follow(S) = { $ } ∪ { First(L') − ϵ } ∪ Follow(L) • ∪ Follow(L') = { $, , ,) }

Follow(L) = {) }

Follow(L') = Follow(L) = {) }

4) $S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

First(S) = { First(A) − ϵ } ∪ First(a) ∪ { First(B) − ϵ } ∪ First(b) = { a, b }
First(A) = { ϵ }
First(B) = { ϵ }

Follow (S) → {$}

Follow (A) = First (a) ∪ First (b) = {a, b}

Follow (B) = First (b) ∪ First (a) = {a, b}.

5) E → E+T / T
T → T X F / F
F → (E) / id.

→ Eliminating left recursion.

E → TE'
E' → +TE' / ∈
T → FT'
T' → X F T' / ∈
F → (E) / id.

First (E) = {First (T) - ∈} ∪ First (F) = {(, id}
First (E') = {+, ∈}
First (T) = First (F) = {(, id}
First (T') = {X, ∈}
First (F) = {(, id}

Follow (E) = {$,)}
Follow (E') = {Follow E} = {$,)}
Follow (T) = {First (E') - ∈} ∪ Follow (E)
First (E') = {+, $,)}
Follow (T') = Follow (T) = {+, $,)}
Follow (F) = {First (T') - ∈} ∪ Follow (T) ∪ Follow (T') = {X, +, $,)}

6) $S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

$\{d, g, h, \epsilon, b, a\}$

First (S) = { First (A) - ϵ } ∪ { First (C) - ϵ } ∪ First (B) ∪ First (b) ∪ { First (ϵ) }

First (A) = First (d) ∪ { First (B) - ϵ } ∪ First (C) = { d, g, h, ϵ }

First (B) = { g, ϵ }

First (C) = { h, ϵ }

Follow (S) = { $ }

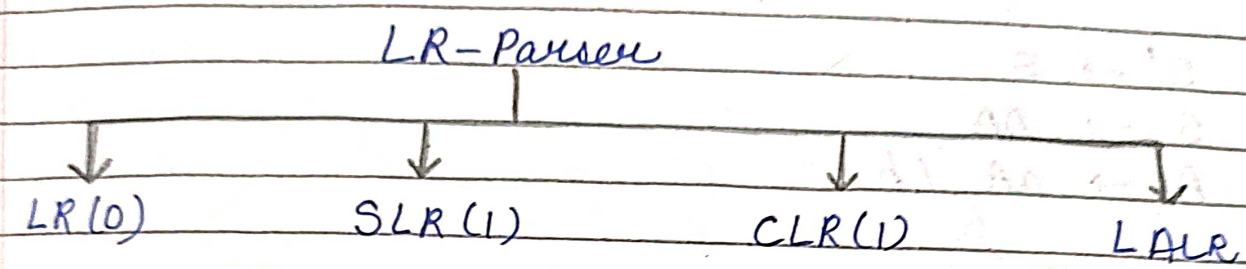
Follow (A) = { First (C) - ϵ } ∪ { First (B) - ϵ } ∪ Follow (S) = { h, g, $ }

Follow (B) = Follow (S) ∪ First (a) ∪ { First (C) - ϵ } ∪ Follow (A) = { $, a, h, g }

Follow (C) = { First (B) - ϵ } ∪ Follow (S) ∪ First (b) ∪ Follow (A) = { g, $, b, h }.

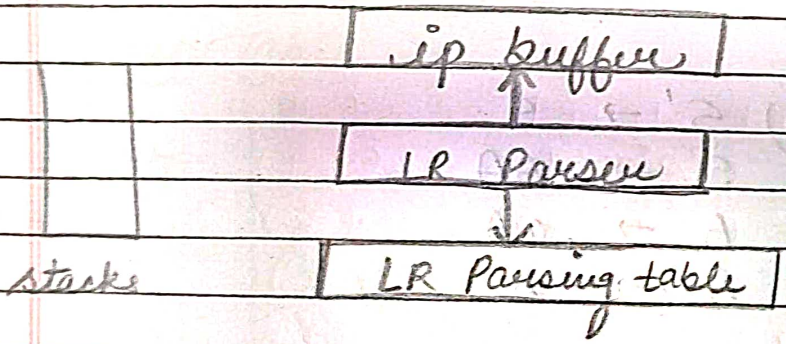
LR Parser

- One type of bottom up parsing.
- Used to parse large class of grammar.
- 'L' → Left to Right scanning
  'R' → Right most derivation in reverse.
  'K' → Lookahead.



Algo :-

- Requires stack, input, output and parsing table



↳ LR(1) Parsing

- Write a context free grammar.
- Check the ambiguity of grammar.
- Add Augment Production.
- Create canonical coll of LR(0)
- Draw DFA.
- Construct LR(1) parsing table.

Date: / / Page no:

Augmented Grammar:- G' will be generated if we add one more production in the given grammar G.

Eg → $S \to AA$
$A \to aA \mid b$

The Augmented grammar G' is rep by

$S' \to S$
$S \to AA$
$A \to aA \mid b$

Canonical coll of LR(0) items.

Eg → $S \to AA$
$A \to aA \mid b$

→ $S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

I0) $S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$

I1) $S' \to S.$
$S \to A.A$
A

I2) $S \to A.A$
$A \to .aA$
$A \to .b$

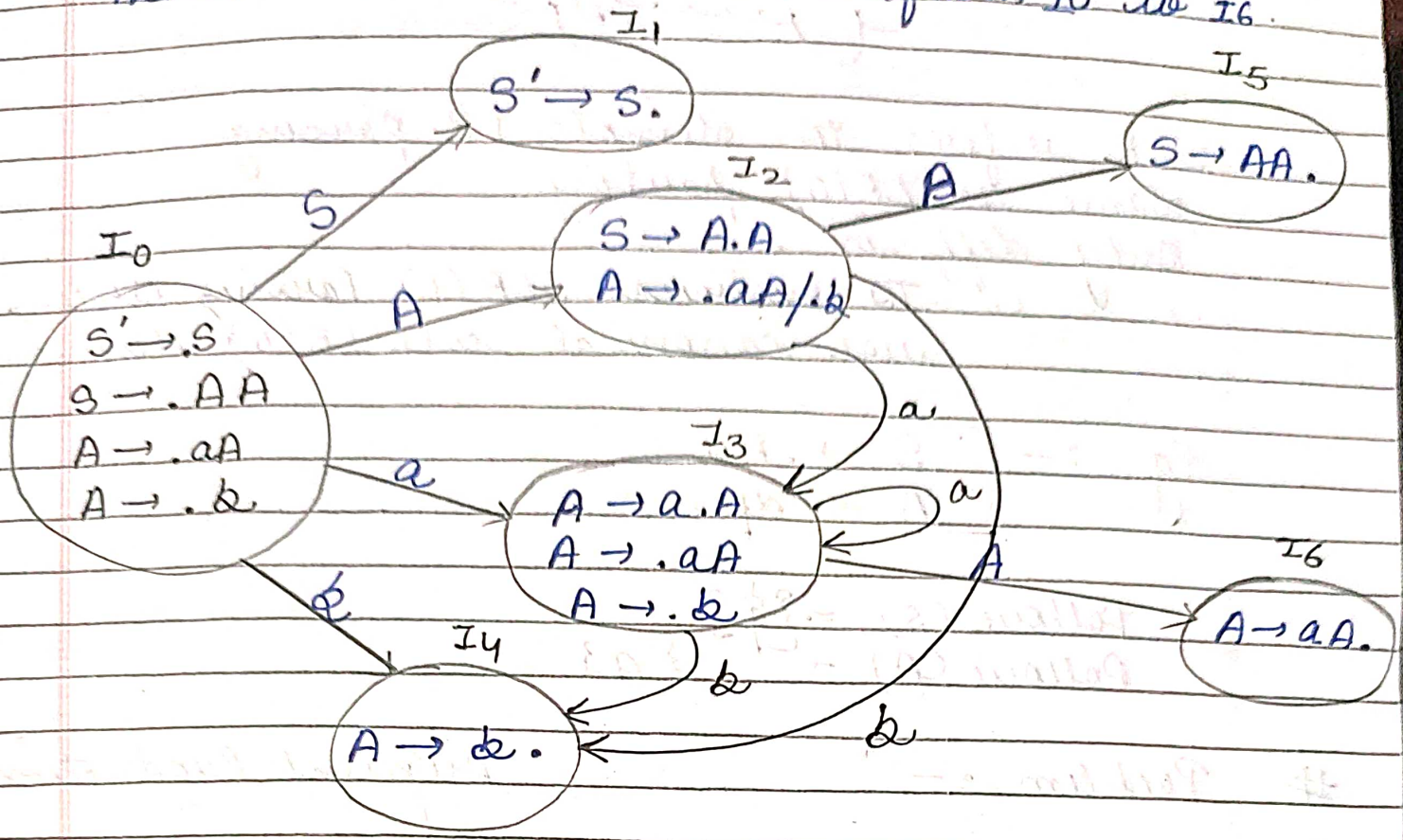
I3) $A \to a.A$
$A \to .aA$
$A \to .b$

I4) $A \to b.$

I5) $SA \to A.$

I6) $A \to aA.$

The DFA contains 7 states from I0 to I6.



Parse Table :-

States	Action			Goto	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	R_3	R_3	R_3		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		

15th March 24)

SLR Parsing

- SLR refers to simple LR parsing
- Same as LR(0) parsing
- Only diff is :
  TO construct SLR(1) Parsing table, use canonical coll LR(0) item.

Eg :- $S \rightarrow .Aa$
$A \rightarrow \alpha\beta.$

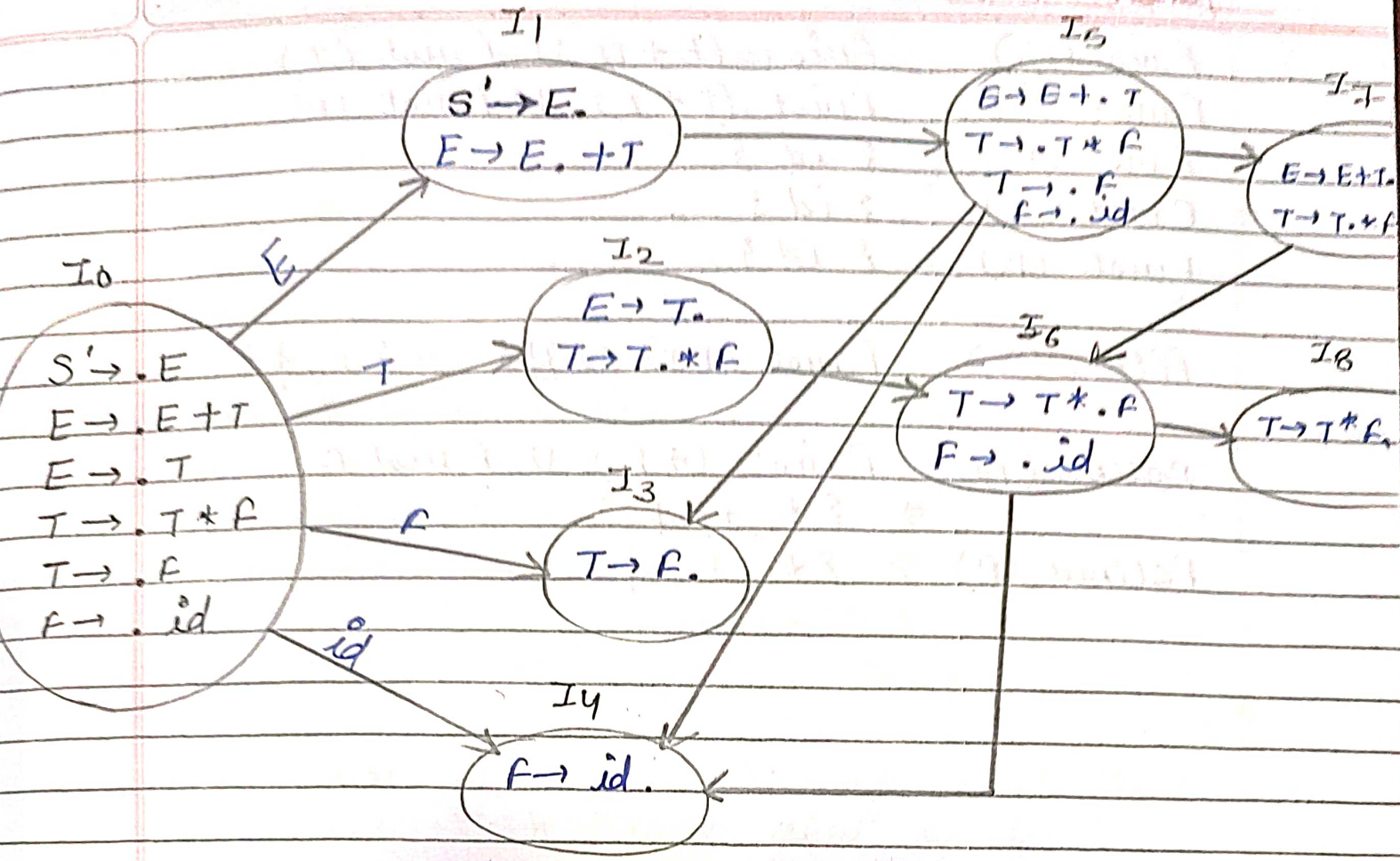
Follow (S) = {$}
Follow (A) = {a}

Problem :-

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

Augment Production

$S \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .id$



states	Action				Goto		
	id	+	*	$	E	T	F
I0	S4			Accept	1	2	3
I1		S5		R2			
I2		R2	S6	R2			
I3		R4	R4	R4			
I4		R5	R5	R5			
I5	S4					7	3
I6	S4						8
I7		R1	S6	R1			
I8		R3	R3	R3			

Date: / / Page no:

First(E) = First(E+T) ∪ First(T)
First(T) = First(T * F) ∪ First(F)
First(F) = { id }
First(T) = { id }
First(E) = { id }

Follow(E) = First(+T) ∪ {$} = {+, $}

Follow(T) = First(*F) ∪ First F
⇒ {*, +, $}
Follow(F) ⇒ {*, +, $}

15 March 24

CLR (1) Parsing

- Refers to canonical lookahead parsing.
- Use the canonical collection of LR (1) → CLR(1)
- Produces the more no. of states as compared to SLR(1) parsing.

Steps → For a given input string, write context free Gr
Check the ambiguity of the grammar.
Add Augment production.
Create canonical coll" of LR(0).
Draw a data flow D (DFA)
Construct a CLR(1) parsing table.

LR(1) item is a collection of LR(0) items and a look ahed symbol.

LR(1) item = LR (0) item + lookAhd

- LA is used to determine that where we place the final item.
- LA always add $ symbol for augment productio

Eg :-

1) S → AA
A → aA
A → b

Add Augment ⇒
S' → .S, $
S → .AA, $
A → .aA, a/b
A → .b, a/b

I_0) $S' \rightarrow .S, \$$
$S \rightarrow .AA, \$$

I_1) $S' \rightarrow S., \$$

I_2) $S \rightarrow A.A, \$$

I_3) $S \rightarrow A.A, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .b, \$$

I_4) $A \rightarrow b., a/b$

I_5) $S \rightarrow AA., \$$

I_6) $A \rightarrow a.A, \$$

I_7) $A \rightarrow b., \$$

I_8) $A \rightarrow aA., a/b$

I_9) $A \rightarrow aA., \$$

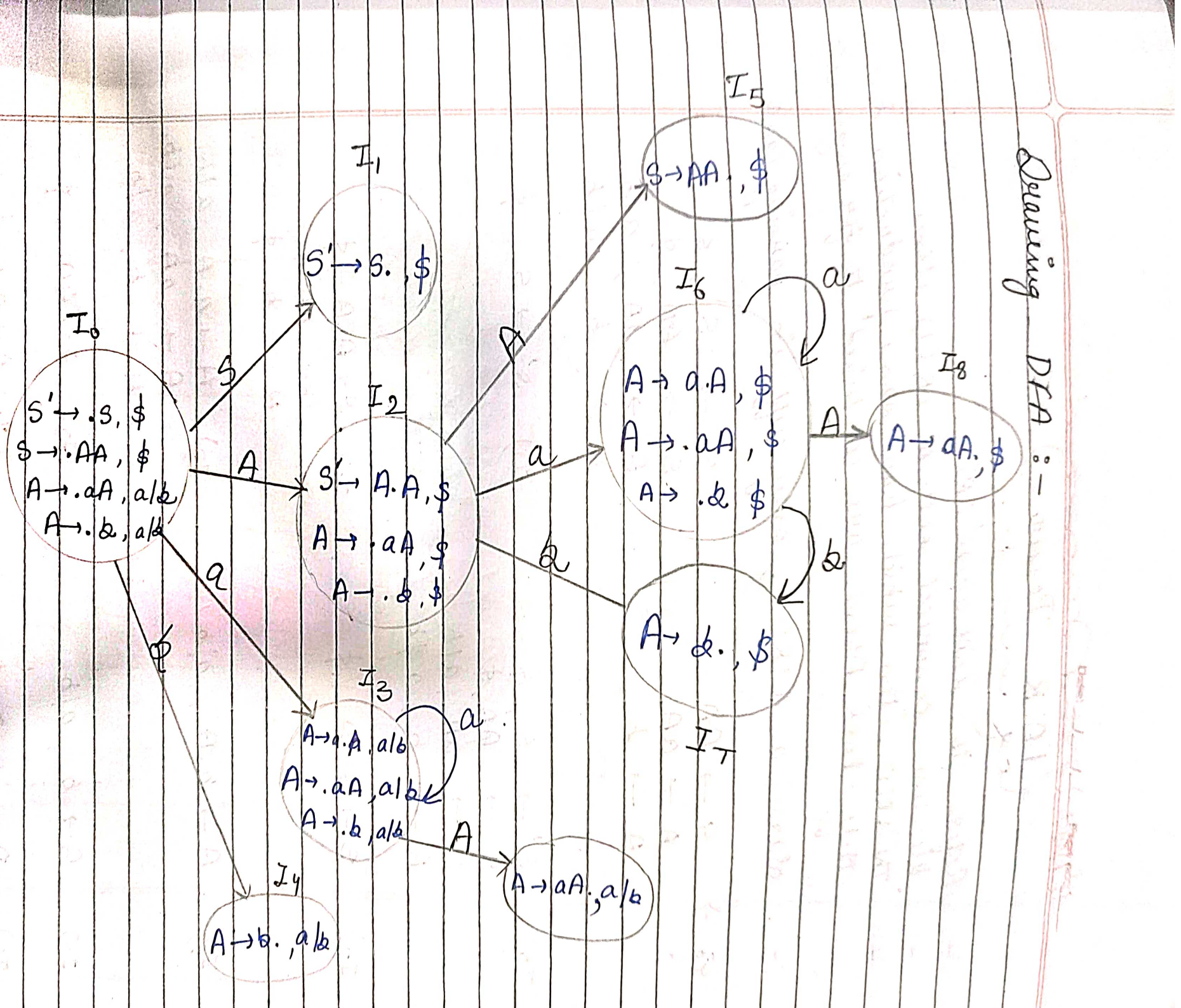
I_0) $S' \rightarrow .S, \$$
$S \rightarrow .AA, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .b, \$$

I_3) $A \rightarrow a.A, a/b$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$

I_6) $A \rightarrow a.A, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .b, \$$

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Drawing DFA :-


I0
S' → .S, $
S → .AA, $
A → .aA, a/b
A → .b, a/b
S
I1
S' → S., $
A
I2
S' → A.A, $
A → .aA, $
A → .b, $
a
I3
A → a.A, a/b
A → .aA, a/b
A → .b, a/b
a
b
I4
A → b., a/b
b
A
I5
S → AA., $
a
I6
A → a.A, $
A → .aA, $
A → .b $
a
b
A
I8
A → aA., $
b
I7
A → b., $
A
A → aA., a/b

15th March

LALR Parser

- Refers to Lookahead LR
- We use canonical coll of LR(1) items
- The LR(1) item which have same production but diff look Ahead are combined to form single set
- Same as CLR(1), only diff in Parsing Table.

Eg →

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

$S' \rightarrow .S, \$$
$S \rightarrow .AA, \$$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$

I0)
$S' \rightarrow .S, \$$
$S \rightarrow .AA, \$$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$

I1) $S' \rightarrow S., \$$

I2) $S \rightarrow A.A, \$$

I2)
$S \rightarrow A.A, \$$
$A \rightarrow .aA, \$$
$A \rightarrow .b, \$$

I3) $A \rightarrow a.A, a/b$

I3)
$A \rightarrow a.A, a/b$
$A \rightarrow .aA, a/b$
$A \rightarrow .b, a/b$

I4) $A \rightarrow b., a/b$

I5) $S \rightarrow AA., \$$

I6) $A \rightarrow a.A, \$$

I6)
$A \rightarrow a.A, \$$
$.aA, \$$
$.b, \$$

I7) A → b. $

I9) A → aA. , $

I6) { A → a.A , $
.aA , $
.b , $ }

I47) { A → b. , a/b, $ }

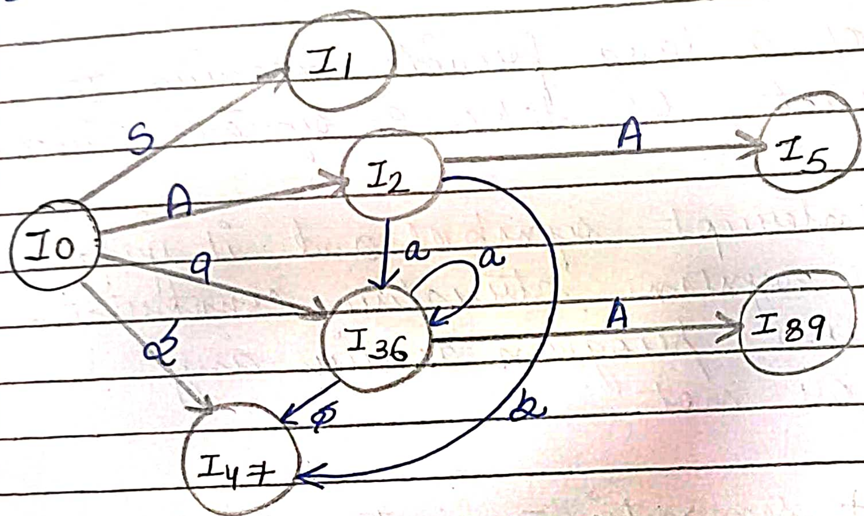
I8) A → aA. , a/b

I3) { A → a.A , a/b
.aA , a/b
.b , a/b }

I36) { A → a.A , a/b, $
A → .aA , a/b, $
A → .b , a/b, $ }

I89) { A → aA. , a/b, $ }

DFA :-



Parsing Table.

States	a	b	$	S	A
I0	S36	S47		12	
I1		accept			
I2	S36	S47			5
I36	S36 S47				89
I47	R3R3	R3			
I5			R1		
I89	R2	R2	R2		

16th March 24)

Semantic Analysis

$E \rightarrow E + T$

- The above CFG production has no semantic rule associated with it.
- The production of CFG which makes the rules of the language.

Semantics –

- Semantics of a lang provide meaning to its constructs, like tokens & syntax structure
- It help interpret symbol and it judges whether the syntax structure constructed in the source program denotes any meaning or not

CFG + semantic rules = Syntax directed definition

Eg → int a = "value"

The following task should be done in semantic analysis –

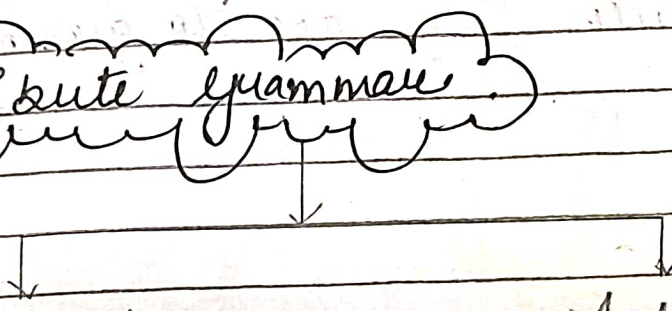
1) Scope resolution.
2) Type checking
3) Array bound checking

→ Semantic Errors –

We have mentioned some of semantic errors that the semantic analyzer expected

- Type mismatch
- Undeclared variables
- Reserved Identifier misuse
- Multiple declaration of variable in scope
- Accessing an out of scope variable
- Actual & formal parameters mismatch.

↳ Attribute grammar



- It is a special form of CFG, where some additional information are appended to one and more of its non-terminal in order to provide context-sensetive information.
- It has each attribute has well defined domain of value such as integer, float, char, string and expression.
- Can pass value among the nodes of a tree

Eg $E \rightarrow E+T$ { E value = E.value + T.value }

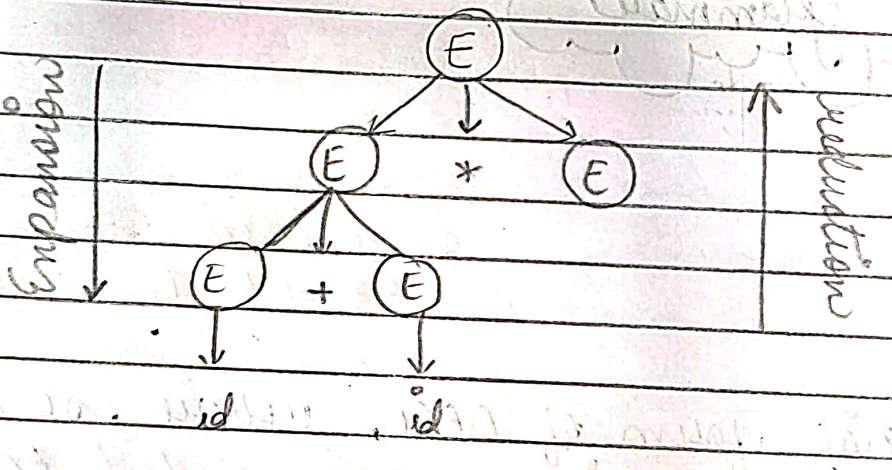
Inherited Attribute → can take values from parent and/or siblings.

S → ABC.

A can get values from S, B and C. (B can take values from S, A & C, C can get from S, A & B etc.

* Expansion :- Non-terminal expanded to terminals as per grammatical rules

Reduction - Terminal is reduced to its non-terminal acc to gramma.



Eg → < Attributes > are two tupled value

int value = 5;

< type ; " integer " >

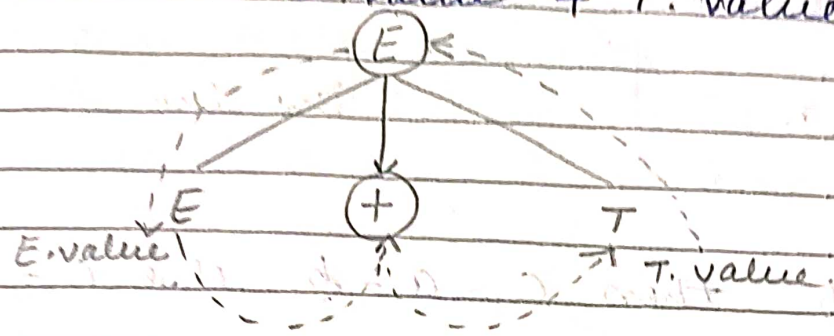
< present value ; " 5 " >.

{ AST - Abstract Syntax tree
{ SDT - Syntax directed translation

↳ **S - attributed SDT**

In an SDT uses only synthesized attributes. These attributes are evaluated using S-attributed SDT that have semantic actions written after production.

E.value = E.value + T.value.



↳ **L - attributed SDT**

The form of SDT uses both synthesized and inherited attribute with restriction of not taking values from right siblings.

S → ABC.
S can take values from A, B, C,
A can take values from S only.
B can take values from S and A.
C can take values from S, A & B.

L - attributed SDT

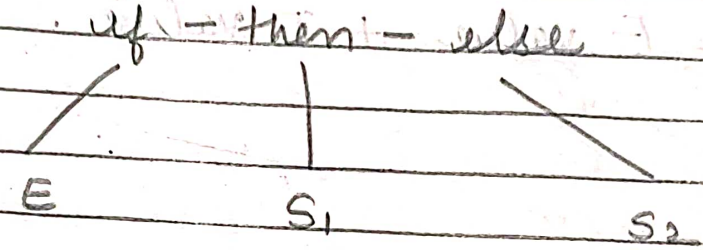
S - attributed SDT

Date: / / Page no:

Numerical Problem on Syntax tree (See Annexure...)

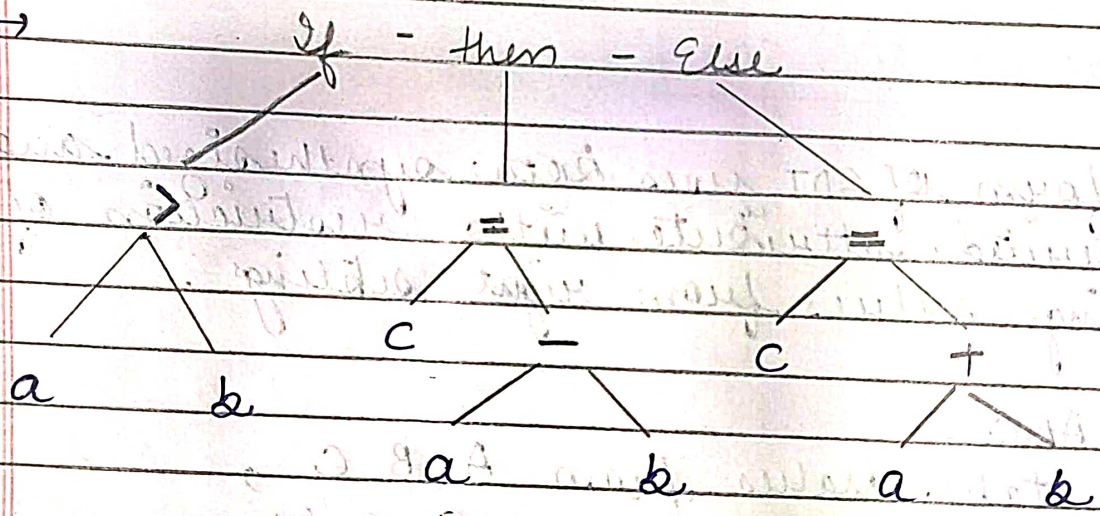
(1) . Syntax tree:

$S \rightarrow$ if E then S_1 else S_2

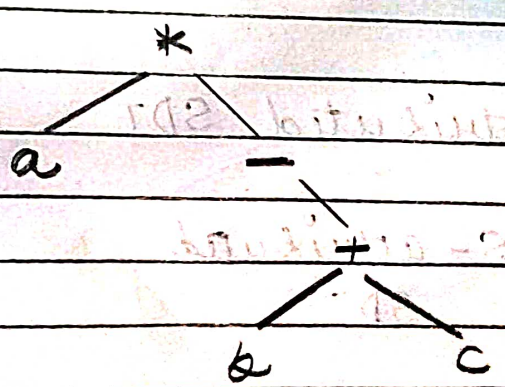


Prob If $a > b$ then $c = a - b$ else $c = a + b$.

Soln →



Syntax tree $\{a * -(b + c)\}$

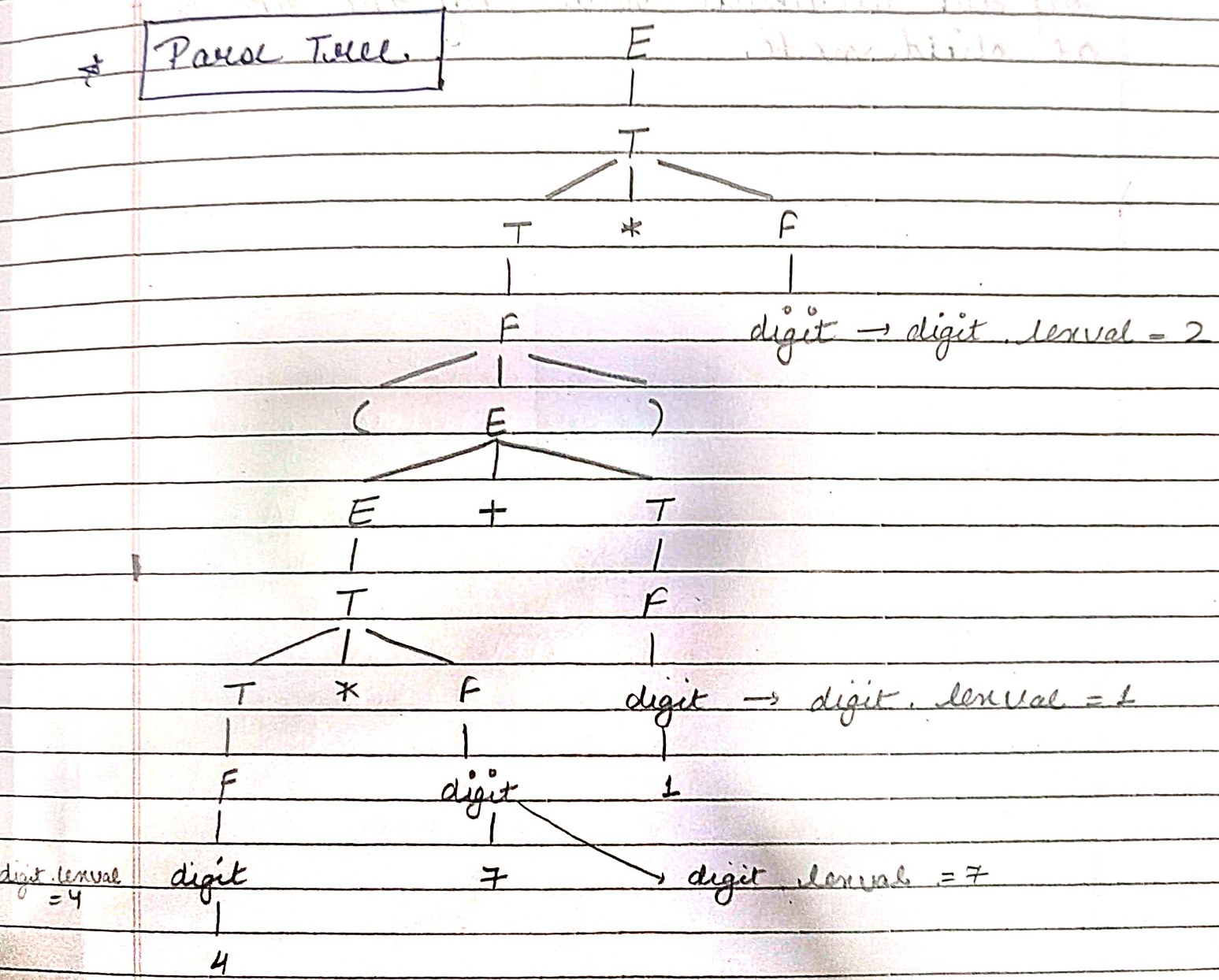


Expression → (4 * 7 + 1) * 2.

Design the Annotated parse tree.

$E \rightarrow E + T$	E.value = E.value + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow E$	F.val = E.val
$F \rightarrow$ digit 0/1/2/3../9	F.val = digit.lexval

* **Parse Tree**



22/3/24

Date: / / Page no:

Difference between synthesized & inherited attributes.

	Synthesized Attributes	Inherited Attribute
Def	whose parse tree node value is determined by the attribute value at child node	If its parse tree node value is determined by the attribute