

**SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES
(AUTONOMOUS)
COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)**

LECTURE NOTES

**Subject Name: AUTOMATA THEORY AND
COMPILER DESIGN**

Year / Branch: III B.TECH – V SEMESTER

Regulation: R23

Prepared By: Mr. A SRINIVASAN

Associate Professor



**23CSE351T III B.Tech. - V Semester
AUTOMATA THEORY AND COMPILER DESIGN**

**L T P C
3 - - 3**

PRE-REQUISITES: A course on Software Engineering

COURSE OBJECTIVES:

1. Able to understand the concept of abstract machines, construct FA, Regular Expressions for the regular languages and equivalent FSMs.
2. Able to construct pushdown automata equivalent to Context free Grammars, construct Turing Machines and understand undecidability.
3. Emphasize the concepts learnt in phases of compiler, lexical analyser and Top-down parser.
4. Able to understand the concepts of Bottom-up parser, Intermediate Code Generation.
5. Able to understand the concepts of Code optimizer and Code Generation.

UNIT 1:INTRODUCTION TO AUTOMATA AND REGULAR EXPRESSIONS (9)

Introduction, Alphabets, Strings and Languages, Chomsky Hierarchy, Automata and Grammars, Regular Grammar and Language, Finite Automata, Deterministic finite Automata (DFA), Nondeterministic finite Automata (NFA), Equivalence of NFA and DFA, Minimization of Finite Automata, Regular Expressions, Converting Regular Grammar and Expression into Finite Automata, Pumping lemma for regular sets, Closure properties of regular sets (Without proof).

UNIT -2: CONTEXT FREE GRAMMARS AND PUSHDOWN AUTOMATA (9)

Context Free Language, Context Free Grammar, Derivation and Parse tree, Ambiguity, Simplification of CFG's, Chomsky Normal Form, Greibach Normal Form, Push Down Automata (PDA), Design of PDA, Equivalence of PDA and CFL/CFG

UNIT -3: TURING MACHINES AND INTRODUCTION TO COMPILERS (9)

Turing Machine, TM Model, Language acceptance, Design of Turing Machine, Compilers, Phases of Compiler, The role of Lexical Analyzer, Input Buffering

UNIT -4: THE PARSERS AND INTERMEDIATE CODE GENERATION (9)

Parser, Top-Down parsers: Recursive Descent Parsers, Predictive Parsers Bottom-up Parsers: Shift-Reduce Parsing, LR parsers, Intermediate Code Generation: Three address codes.

UNIT -5: THE CODE OPTIMIZATION AND CODE GENERATION (9)

Code Optimization: Peephole optimization, Basic blocks and flow graphs, DAG, Principles of Source Code Optimization, Code Generation: Issues in Design of Code Generation, Simple Code Generator

Total Hours: 45



COURSE OUTCOMES:

On successful completion of the course, students will be able to		POs
CO1	Identify the various bugs and correcting them after knowing the consequences of the bug.	PO1, PO2
CO2	Perform functional testing using control flow and transaction flow graphs.	PO1, PO2, PO3
CO3	Design the path expression and reduce them very well when needed.	PO1, PO2, PO3
CO4	Test logic or an application and identifying the nice and ugly domains.	PO1, PO2, PO3, PO4
CO5	Use appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.	PO1, PO2, PO3, PO5

TEXT BOOKS:

1. **Michael Sipser**, *Introduction to the Theory of Computation*, 3rd Edition, Cengage Learning, Reprint 2021.
2. **K. L. P. Mishra & N. Chandrasekaran**, *Automata Theory and Formal Languages*, Revised Edition 2022, PHI Learning
3. **John C. Martin**, *Theory of Computation*, 2021 Edition, McGraw-Hill.
4. **Aho, Lam, Sethi & Ullman**, *Compilers: Principles, Techniques and Tools*, 2nd Edition, Pearson, Reprint 2020–2023.
5. **K. V. N. Sunitha & N. Kalyani**, *Compiler Design*, Latest Edition 2021/2022, McGraw-Hill.

REFERENCE WEBSITE:

1. GeeksforGeeks – Automata Theory <https://www.geeksforgeeks.org/automata-theory>
2. TutorialsPoint – Theory of Computation
https://www.tutorialspoint.com/automata_theory
3. NPTEL – Theory of Computation (IIT Courses)
<https://nptel.ac.in/courses/106/104/106104028>
4. GeeksforGeeks – Compiler Design <https://www.geeksforgeeks.org/compiler-design-tutorials>
5. TutorialsPoint – Compiler Design https://www.tutorialspoint.com/compiler_design

CO-PO MAPPING

CO\PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO.1	3	3	-	-	-	-	-	-	-	-	-	-
CO.2	2	3	3	-	-	-	-	-	-	-	-	-
CO.3	2	3	3	-	-	-	-	-	-	-	-	-
CO.4	2	3	3	3	-	-	-	-	-	-	-	-
CO.5	2	3	3	-	3	-	-	-	-	-	-	-
CO*	2.2	3	3	3	3	-	-	-	-	-	-	-

Introduction, Alphabets, Strings and Languages

Presented by
Mr A Srinivasan
Associate Professor
SITAMS, Chittoor

INTRODUCTION TO AUTOMATA AND REGULAR EXPRESSIONS

- Introduction, Alphabets, Strings and Languages, Chomsky Hierarchy, Automata and Grammars, Regular Grammar and Language, Finite Automata, Deterministic finite Automata (DFA), Nondeterministic finite Automata (NFA), Equivalence of NFA and DFA, Minimization of Finite Automata, Regular Expressions, Converting Regular Grammar and Expression into Finite Automata, Pumping lemma for regular sets, Closure properties of regular sets (Without proof).
- Text Book:
 - Michael Sipser, Introduction to the Theory of Computation, 3rd Edition, Cengage Learning, Reprint 2021.
 - K. L. P. Mishra & N. Chandrasekaran, Automata Theory and Formal Languages, Revised Edition 2022, PHI Learning

Alphabet

- Alphabet : An alphabet is a finite and nonempty set of symbols.
- It is denoted by Σ .

Example :

- a) $\Sigma = \{0,1\}$, the set of binary alphabet.
- b) $\Sigma = \{a,b,\dots,z\}$, the set of all lowercase letters.
- c) $\Sigma = \{+, \&, \dots\}$, the set of all special characters.

String or Word

- ✓ String or Word : A string is a finite set sequence of symbols chosen from some alphabets.
- It is denoted by w, x, y , or z

Example :

a) $w = "0111010"$ is a string over alphabet $\Sigma = \{0, 1\}$

b) $w = "aabbaacab"$ is a string over alphabet $\Sigma = \{a, b, c\}$

Empty String and Length of String

- Empty String : The empty string is the string with zero occurrences of symbol (no symbol).
- It is denoted by ϵ (epsilon).
- Length of String : The length of a string is number of symbols in the string.
- It denoted by $|w|$.

Example :

$w = 010110101$ from binary alphabet $\Sigma = \{0,1\}$

Length of a string $|w| = 9$

Power of an Alphabet

- Power of an Alphabet: If Σ is an alphabet, we can express the set of all strings of certain length from that alphabet by using an exponential notation.
- It is denoted by Σ^k is the set of strings of length k , each of whose symbols is in Σ .

Example 1:

$\Sigma = \{0,1\}$ has 2 symbols

i) $\Sigma^1 = \{0,1\}$ ($2^1 = 2$)

ii) $\Sigma^2 = \{00, 01, 10, 11\}$ ($2^2 = 4$)

iii) $\Sigma^3 = \{000,001,010,011,100,101,110,111\}$ ($2^3 = 8$)

Power of an Alphabet

- The set of strings over an alphabet Σ is usually denoted by Σ^* .

Example:

$$\Sigma^* = \{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11\} \quad (\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \dots) - \text{with } \epsilon \text{ symbol.}$$

- The set of strings over an alphabet Σ excluding ϵ is usually denoted by Σ^+ .

Example:

$$\Sigma^+ = \{0,1\}^+ = \{0, 1, 00, 01, 10, 11\} \quad (\Sigma^+ = \Sigma^* - \{\epsilon\} \text{ or } \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \dots) - \text{without } \epsilon \text{ symbol.}$$

Concatenation

- Concatenation of String Join the two or more strings.
- Let x and y be two strings. Concatenation of strings x and y is appending symbols of y to right end of x.

Example1:

$x = a_1a_2a_3\ldots a_n$

$y = b_1b_2b_3\ldots b_n$

Concatenation of Strings x and y = $a_1a_2a_3\ldots a_nb_1b_2b_3\ldots b_n$

Example 2:

S = welcome

T = SITAMS

Concatenation of S and T = welcomeSITAMS

Languages

- Languages: If Σ is an alphabet, and $L \subseteq \Sigma^*$ then L is a language.

Examples:

- The set of legal English words
- The set of legal C programs
- The set of strings consisting of n 0's followed by n 1's
$$L = \{ 0^n 1^n ; n \geq 0 \}$$

Operations on Languages

- **Complementation:** Let L be a language over an alphabet Σ . The complementation of L , denoted by $\bar{L}$, is $\Sigma^* - L$.
- **Union:** Let L_1 and L_2 be languages over an alphabet Σ . The union of L_1 and L_2 , denoted by $L_1 \cup L_2$, is $\{x \mid x \text{ is in } L_1 \text{ or } L_2\}$.
- **Intersection:** Let L_1 and L_2 be languages over an alphabet Σ . The intersection of L_1 and L_2 , denoted by $L_1 \cap L_2$, is $\{x \mid x \text{ is in } L_1 \text{ and } L_2\}$.
- **Concatenation:** Let L_1 and L_2 be languages over an alphabet Σ . The concatenation of L_1 and L_2 , denoted by $L_1.L_2$, is $\{w_1.w_2 \mid w_1 \text{ is in } L_1 \text{ and } w_2 \text{ is in } L_2\}$.

Operations on Languages

- **Reversal:** Let L be a language over an alphabet Σ . The reversal of L , denoted by L^r , is $\{w^r \mid w \text{ is in } L\}$.
- **Kleene's closure:** Let L be a language over an alphabet Σ . The Kleene's closure of L , denoted by L^*

Example: $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \dots$

where $L^0 = \{\epsilon\}$

$L^1 = \{L\}$

$L^2 = \{LL\}$

$L^n =$ Concatenating L itself for n times

Operations on Languages

- Positive Closure: Let L be a language over an alphabet Σ . The positive closure of L , denoted by L^+

Example:

$$L^+ = L^1 \cup L^2 \cup L^3 \dots = L^* - L^0$$

Finite Automata

- Automaton is an abstract computing device.
- It is a mathematical model of a system, with discrete inputs, outputs, states and set of transitions from state to state that occurs on input symbols from alphabet Σ
- **Finite Automata representations in three ways,**
 - o Graphical form (Transition Diagram)
 - o Tabular form (Transition Table)
 - o Mathematical (Transition Function or Mapping Function)

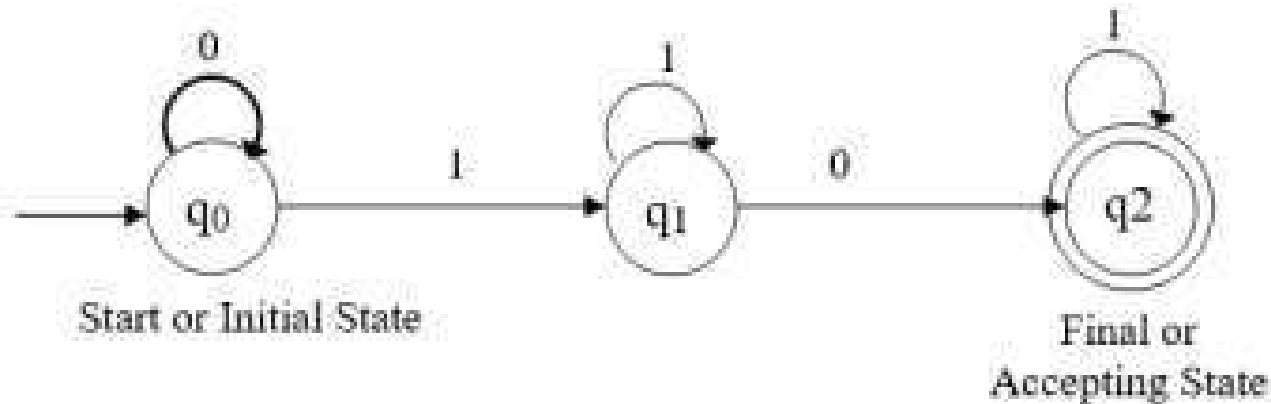
Mathematical Definition of Finite Automata

- A finite automata is consists of 5-tuple denoted by M and is defined as $M=(Q, \Sigma, \delta, q_0, F)$ Where
 - Q is a finite set of states.
 - Σ is a finite set of input alphabets.
 - $\delta : Q \times \Sigma \rightarrow Q$ is a finite set of transition functions.
 - $q_0 \in Q$ is the start state or initial state.
 - $F \in Q$ is the set of accept states or final states.

Transition Diagram (Transition graph)

- It is a directed graph associated with the vertices of the graph corresponds to the states of the finite automata.
- It is a 5-tuple graph used state and edges represent the transitions from one state to other state.

Example:



Transition Table

- It is the tabular representation of the DFA. For a transition table the transition function is used.

Example:

States	Input	
	0	1
$\rightarrow\{q_0\}$	$\{q_1\}$	$\{q_0\}$
$\{q_1\}$	-	$\{q_2\}$
$\ast\{q_2\}$	-	-

Transition Function

- The mapping function or transition function denoted by δ . Two parameters are passed to this transition function:

(i) current state

(ii) input symbol.

- It returns a state which can be called as next state.

$$\delta (\text{current_state}, \text{current_input_symbol}) = \text{next_state}$$

Example:

$$\delta (q_0, a) = q_1$$

Computation of a Finite Automata

$$\delta (q_0, a) = q_1$$

- The automaton receives the input symbols one by one from left to right.
- As per above example machine reads a symbol (a), then it changes from current state (q_0) to next state (q_1).
- When Machine reads the last symbol of the input it may produces accept or reject.
- The Machine may be in an accept state or not in an accept state.

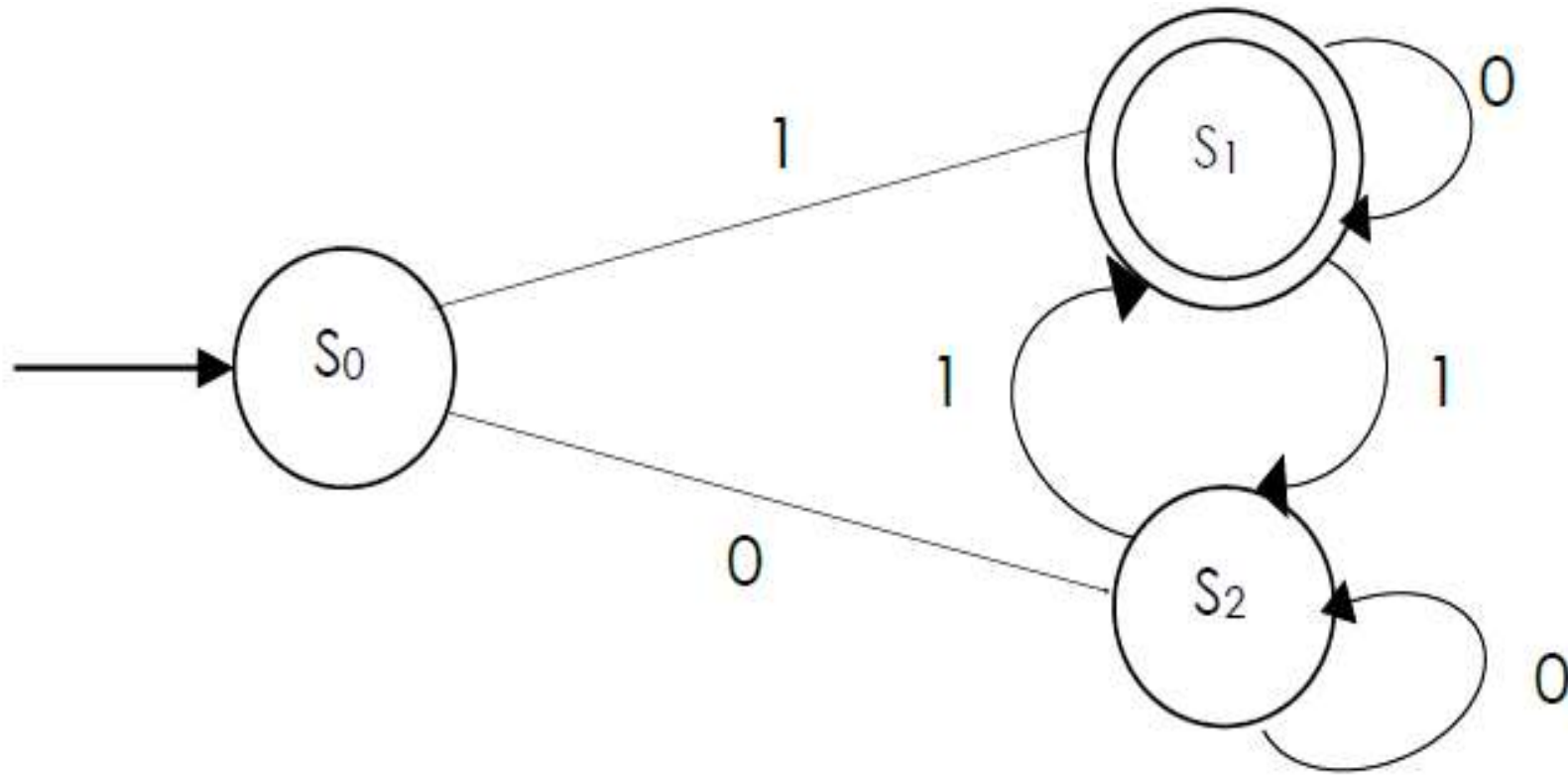
Types of Finite Automata

- Deterministic Finite Automata (DFA)
- Non-Deterministic Finite Automata (NFA or NDFA)
- Non-Deterministic Finite Automata with ϵ move (ϵ -NFA or ϵ -NDFA)

Deterministic Finite Automata (DFA)

- Deterministic Finite Automaton is a Finite Automata in which there is **only one path for a specific input from current state to next state**.
- There is a unique transition on each input symbol.
- Formal Definition of Deterministic Finite Automata
- A finite automaton is a 5-tuple; they are $M = (Q, \Sigma, \delta, q_0, F)$ Where
 - Q is a finite set of states.
 - Σ is a finite set of alphabets.
 - $\delta : Q \times \Sigma \rightarrow Q$ is a finite set of transition functions.
 - $q_0 \in Q$ is the start state or initial state.
 - $F \subseteq Q$ is the set of accept states or final states.

Deterministic Finite Automata (DFA)



Non-Deterministic Finite Automata

- Non-Deterministic Finite Automata is a Finite Automata in which there are **many paths for a specific input from current state to next state.**
- There are more than one transitions on each input symbol.
- Formal Definition of Non-Deterministic Finite Automata
- A finite automaton is a 5-tuple; they are $M = (Q, \Sigma, \delta, q_0, F)$

Where

Q is a finite set of states.

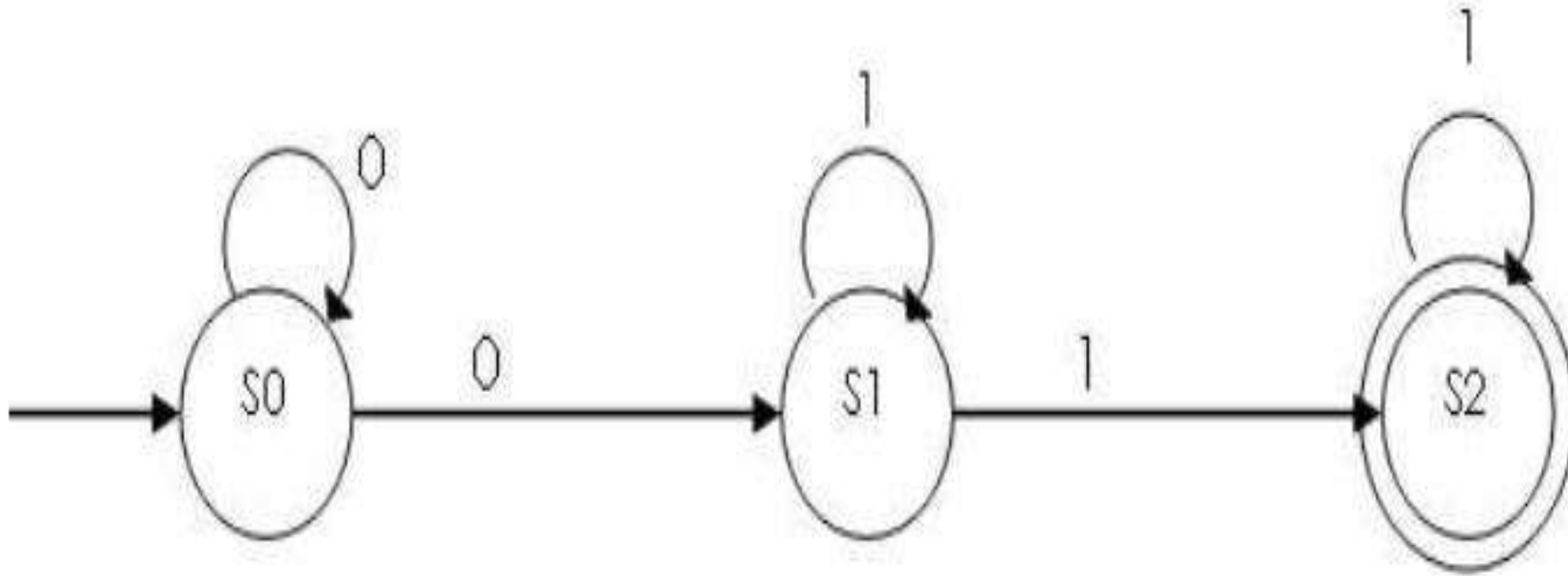
Σ is a finite set of alphabets.

$\delta : Q \times \Sigma \rightarrow 2^Q$ is a finite set of transition functions.

$q_0 \in Q$ is the start state or initial state.

$F \subseteq Q$ is the set of accept states or final states.

Non-Deterministic Finite Automata



Finite Automaton with ϵ - moves

- Non-Deterministic Finite Automata with moves is a Finite Automata in which **many paths for a specific input or ϵ from current state to next state.**
- The ϵ is a **character** used to indicate null string.
- Formal Definition of Non-Deterministic Finite Automata
- A finite automaton is a 5-tuples; they are $M = (Q, \Sigma, \delta, q_0, F)$

Where

Q is a finite set of states.

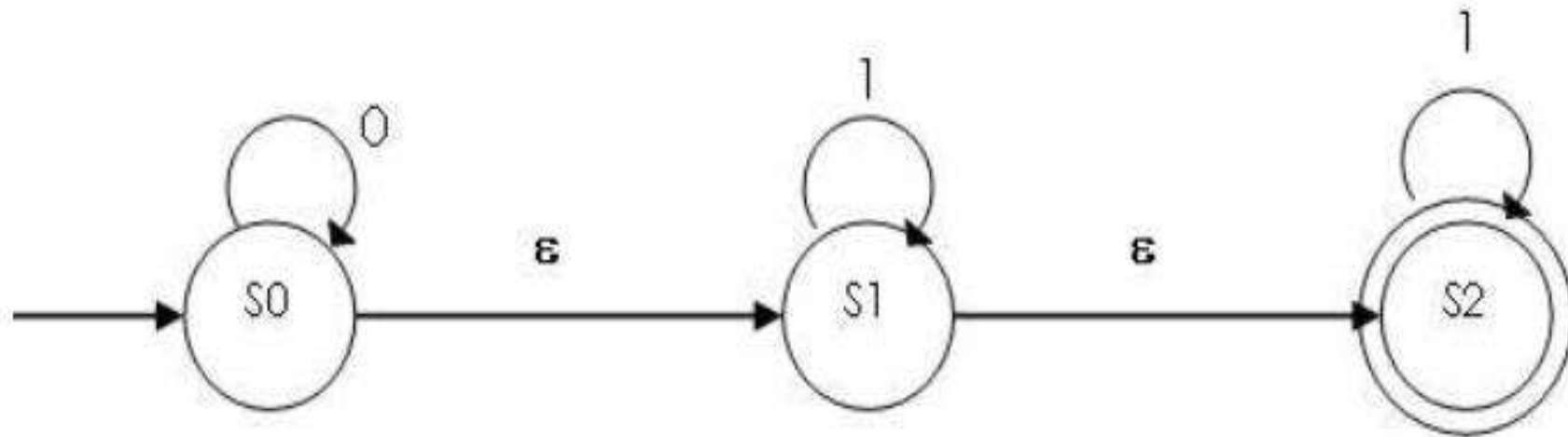
Σ is a finite set of alphabets.

$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is a finite set of transition functions.

$q_0 \in Q$ is the start state or initial state.

$F \subseteq Q$ is the set of accept states or final states

Finite Automaton with ϵ - moves

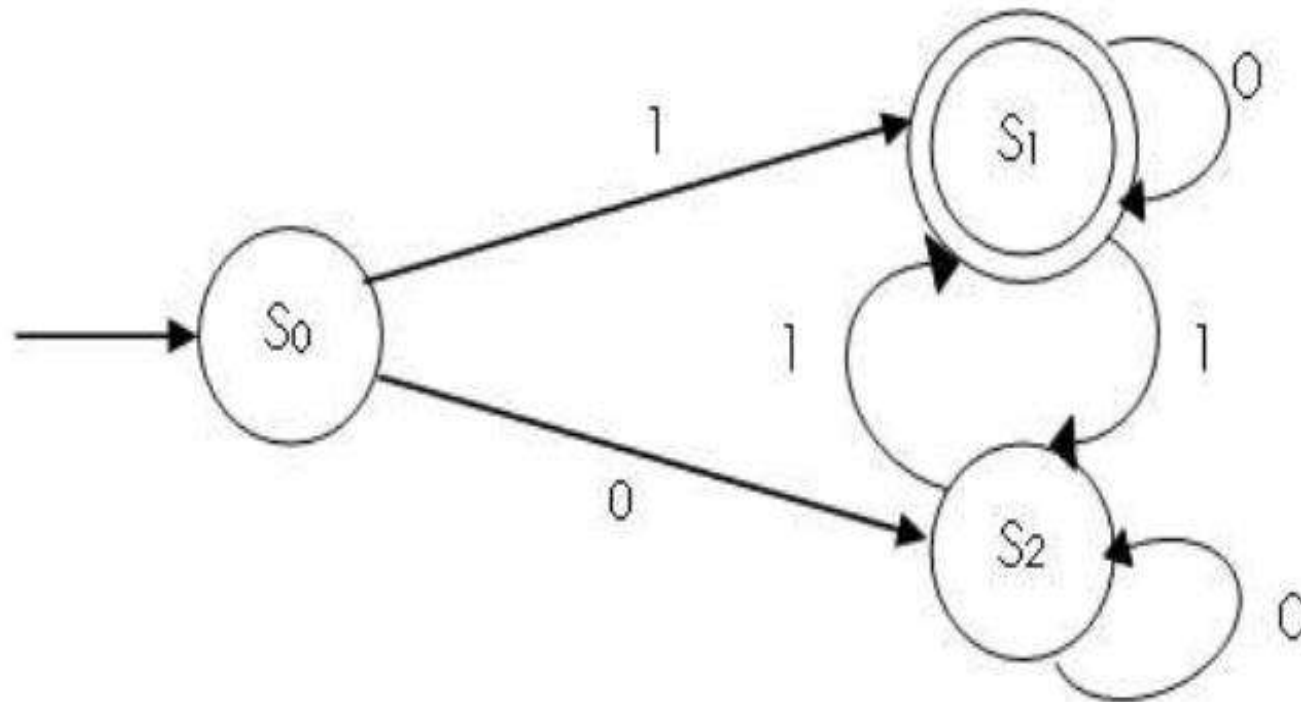


Differentiate DFA and NFA

Sl. No	DFA	NFA
1.	DFA is Deterministic Finite Automata	NFA is Non-Deterministic Finite Automata
2.	For given state, on a given input we reach to deterministic and unique state.	For given state, on a given input we reach to more than one state.
3.	DFA is a subset of NFA	Need to convert NFA to DFA in the design of compiler.
4.	$\delta : Q \times \Sigma \rightarrow Q$ Example: $\delta(q_0, a) = \{q_1\}$	$\delta : Q \times \Sigma \rightarrow 2^Q$ Example : $\delta(q_0, a) = \{q_1, q_2\}$

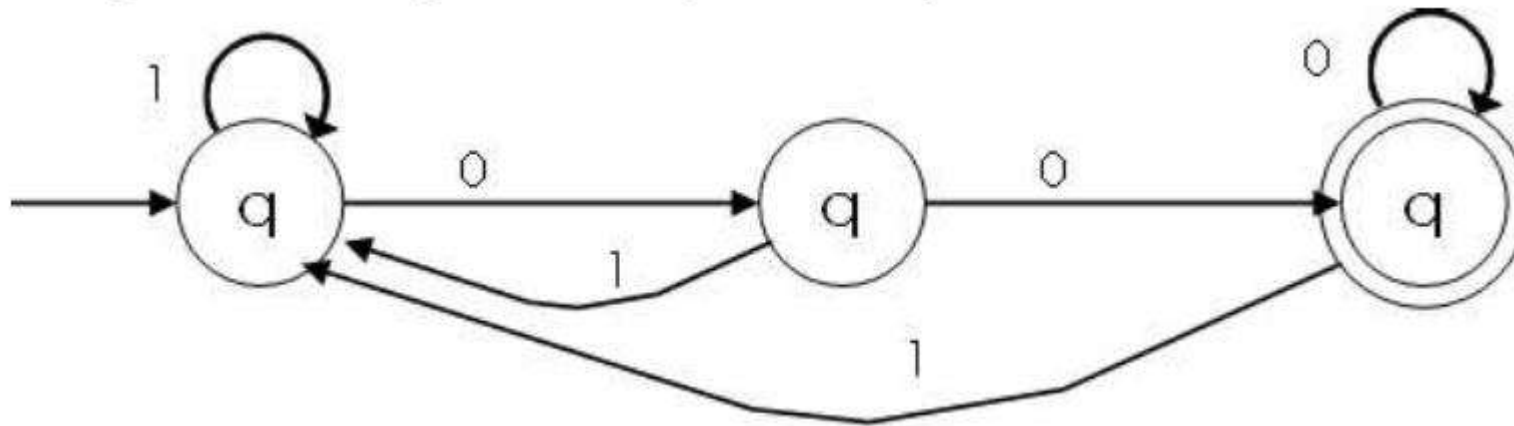
Problems for Finite Automata

1. Design a Deterministic Finite Automata which accepts odd number of 1's and any number of 0's.



Problems for Finite Automata

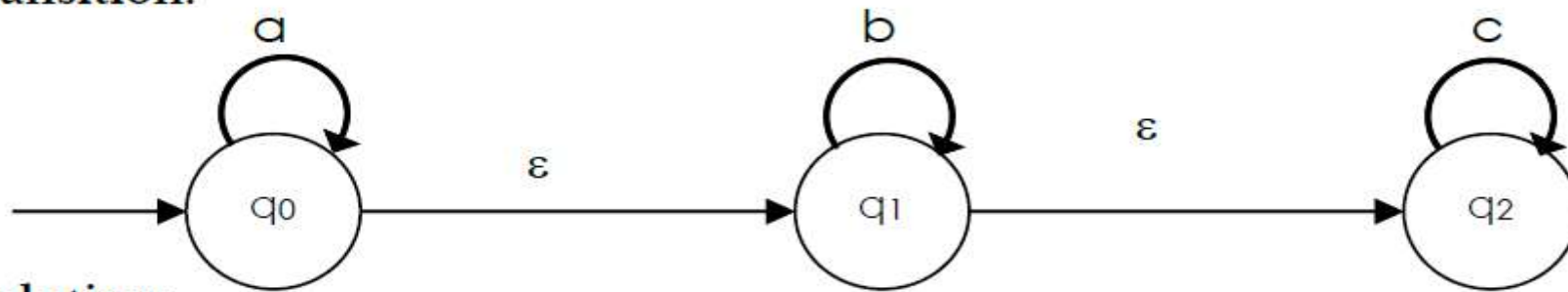
2. Design a Deterministic Finite Automata which accepts odd number of 0's and any number of 1's.
3. Design FA to accept the string that always ends with 00.



Problems for Finite Automata

4. Obtain the ϵ closure of states q_0 and q_1 in the following NFA with ϵ transition.

transition.



Solution:

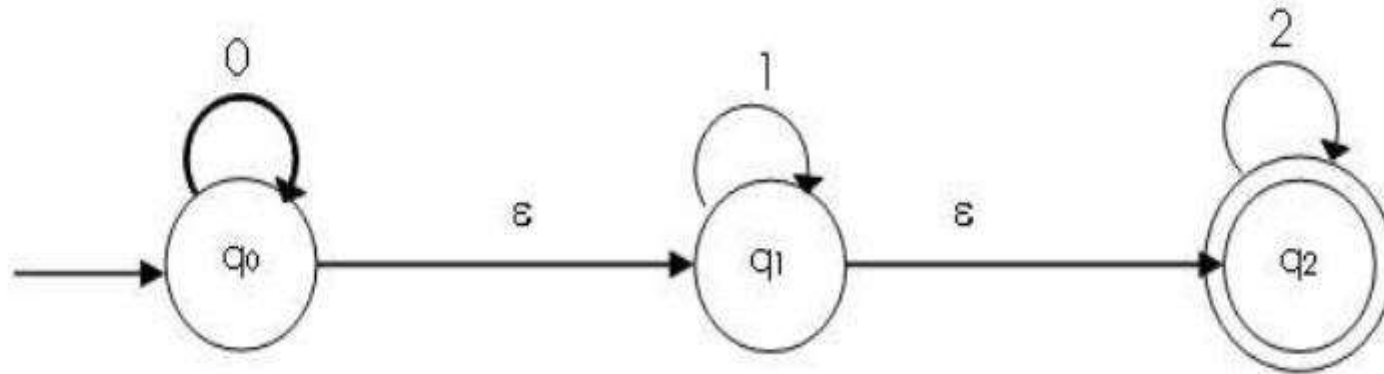
ϵ - CLOSURE $\{q_0\} = \{q_0, q_1, q_2\}$

ϵ - CLOSURE $\{q_1\} = \{q_1, q_2\}$

ϵ - CLOSURE $\{q_2\} = \{q_2\}$

Problems for Finite Automata

5. Obtain ϵ closure of each state in the following NFA with ϵ move.



Solution:

ϵ - CLOSURE $\{q_0\} = \{q_0, q_1, q_2\}$

ϵ - CLOSURE $\{q_1\} = \{q_1, q_2\}$

ϵ - CLOSURE $\{q_2\} = \{q_2\}$

Automata Theory And Compiler Design:-

Automata:-

It is a "abstract machine (or) a model."

Automata Theory:-

It is a study of "abstract machine" (or) a "computing device."

History of Automata Theory:-

- In 1930's, "Allen Turing" studied Abstract machine.
- He studied this theory, to know "what a computing device could do and what a computing device could not do."
- In 1940's and 1950's, "Finite Automata" was studied by some scientists.
- We use this study in "software and hardware", "computing device".
- In late 1950's, "Chomsky" studied, Grammer, these Grammers are used the "development of software".
- In 1960's, "S. Cook", he extended study of "Allen Turing" and he was able to separate Problems that can be solved by computer and that

can't be solved by computer.

What was the reason to study Automata Theory?

→ It was used to ^{design and} develop the some softwares.

→ We use this in lexical ^{analyser} software, digital circuits software

to verify some systems, ^{↳ Breaks the Programs into logical units}

to web scan systems ^{↳ Key words}

We use the automata Theory. ^{Identifiers Punctuations}

Introduction To Finite Automata Theory:-

Finite Automata:-

The Automata that exists in "finite no. of states."

→ The Purpose of the "state" is to "remember certain portion of system history".

Advantage:-

→ We can develop the device/system in fixed no. of states/resources.

How to represent Finite Automata:-

→ To represent finite automata, we use

(circle) $\bigcirc$ → To represent states

(start) → $\Rightarrow$ To represent start state

It is label to the start state. It will be there in arrow.

(double circle) $\bigcirc$ → To represent the final states.
↓ Accepting states.

(arc) $\curvearrowright$ → To represent transition [movement].

It is labelled with the some input.

How To model a software:-

→ The finite automaton for the part of lexical analyser that recognises the string "Then".

→ It requires 5 states. It refers to the prefixes of the "then" [t, th, the, then, $\bigcirc$ (one empty state)].



Structural Representation:-

Grammars

→ Used in the development of the processing ^{the} data with a recursive structure.

Regular Expressions

→ Used for the structure or organise the data.

Eq:- UNIX style

$[A-Z][a-z]^*[]$

$[A-Z], [A-Z]$

$[A-Z]$ → any capital letter

$[a-z]$ → any small letter

$*$ → any no. of occurrences

$[]$ → a blank space

Eq:- Delhi IN

The given regular expression represent a capitalized word

followed by blank space followed by 2 capital letters.

It is a single name followed by two letters.

For multiword city names

$[A-Z][a-z]^*[] [A-Z]$

West Bengal IN $[a-z]^*$

$[A-Z][A-Z]$

Automata & Complexity:-

It discusses limitations of computations.

→ What a computer can do at all?

This study is called decidability and problems that can be solved with sometimes are called decidable.

→ What a computer can do efficiently?

This study is called intractability and the problems that can be solved with ^{no} more time called "tractable" or "undecidable".

Central Concepts of Automata Theory:-

Alphabet: It is a finite non empty set of symbols.

It is denoted by " Σ ".

Eg:- $\Sigma = \{0,1\}$ -- Binary Alphabet.

$\Sigma = \{a, \dots, z\}$ -- Small letter Alphabet.

ASCII: character [A-Z - ASCII - values]

String: It is a finite sequence of input symbols chosen from an alphabet Σ .

Eg:- 0111 is a string chosen from a binary alphabet. $\Sigma = \{0,1\}$

It is represented by w .

chosen from an alphabet Σ .

1) Empty string:- It is a string that contains zero occurrences of input symbols.

→ It is represented as ϵ (Epsilon).

→ It can be a string that can be any alphabet taken from.

2) Length of the string:-

→ It can be no. of positions of symbols in the string. → It is represented as " $|w|$ ".

Ex:- $w = 1010111$

The length of the string " $|w|$ " = 7

3) Powers of the string:-

→ If Σ is an alphabet, then Σ^k is a set of strings of length k .

Ex:- $\Sigma = \{a, b\}$

$\Sigma^0 = \phi$, $\Sigma^1 = \{a, b\}$, $\Sigma^2 = \{aa, ab, ba, bb\}$

$\Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$

→ The set of all strings over some alphabet Σ is denoted by " Σ^* ".

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \dots$

→ The set of all non empty strings is denoted by " Σ^+ ".

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \dots$$

The relation b/w the both strings is " $\Sigma^* = \Sigma^1 \cup \epsilon$ ".

4) Concatenation of strings:-

If x and y are two strings then the concatenation of x, y is represented by a " xy " where the string x followed by y .

Ex:- $\left. \begin{array}{l} x = 10001 \\ y = 00110 \end{array} \right\} \Sigma = \{0, 1\}$

$$xy = 1000100110$$

$$\rightarrow \epsilon x = x \epsilon = x$$

$\hookrightarrow$ empty string [identity of concatenation].

Language:- It is a set of strings chosen from the Σ^* .

Where Σ is some alphabet.

Ex:- $\Sigma = \{0, 1\}$

L = set of strings with equal no. of 0s and equal no. of 1s.

$$L = \{01, 10, 0011, 1100, 1001, 0110, 0101,$$

$$1010, \dots\}$$

$\rightarrow$ It is represented by " L ".

$\rightarrow$ If Σ is an alphabet and L is subset of Σ^* ,

then L is a language.

→ If L is a language over Σ then it is also a language over superset of Σ .

Eq:- Set of binary integers whose value is prime

$L = \{10, 11, 101, 111, \dots\}$

→ Σ^* is also a language over the alphabet Σ .

→ ϕ is empty language $\phi \neq \{\epsilon\}$

$\hookrightarrow$ empty language

$\hookrightarrow$ empty language

$\hookrightarrow$ language with empty string ϵ

Problems:-

It is a question of deciding whether a string belongs to some particular language or not.

Eq:- $L = \{0, 00, 10, 000, 100, 010, 110, \dots\}$

determine 000 is part of given string or not (Yes)

→ It determine the membership of the string in a language.

→ If Σ is an alphabet and L is a language the question determining whether the string 'w' is there in language L or not is a problem.

→ There are two types of finite automata:

1) Deterministic Finite Automata:-

→ Deterministic refers that for each input symbol there is only one state to which finite automata can transition from its current state.

→ DFA is represented by "5-tuple" notation. It

consists of $A = \{Q, \Sigma, \delta, q_0, F\}$

→ It consists of i) A set of states. It is represented by " Q ".

ii) A set of input symbols, represented by " Σ ".

iii) A transition function $\delta(q_0, i)$ which takes two arguments "state and input symbol" and give one output.

iv) starting state, represented as " q_0 ".

v) set of final states ~~are~~ (or) Accepting states, represented as " F ".

Notations of DFA:-

→ There are two types of Notations:-

1) Transition Diagram:-

It is a Graph that contains

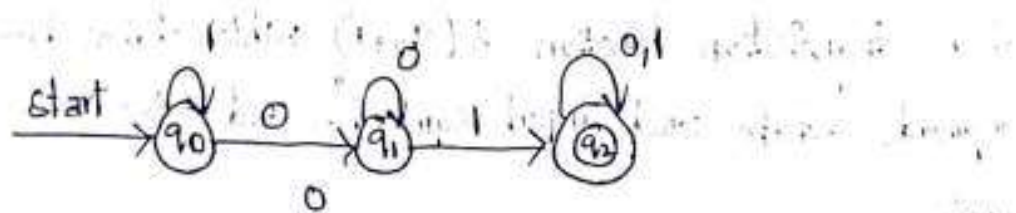
i) For each state, there is a node

- ii) For each input symbol and for each state transition function δ is represented by an arc labelled with an input symbol.
- iii) There is an arrow into the starting state labelled as start.
- iv) The final state or states must be in double circle $\odot$ and other states must be in single circle $\circ$.

Question:- Construct DFA which accepts all strings of "0's and 1's" that have the sequence "01".

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

$$L = \{01, 001, 010, 011, 101, \dots\}$$



2) Transition Table:-

→ Transition Table is a tabular representation of function δ .

→ Rows corresponds to states.

→ columns corresponds to input symbols.

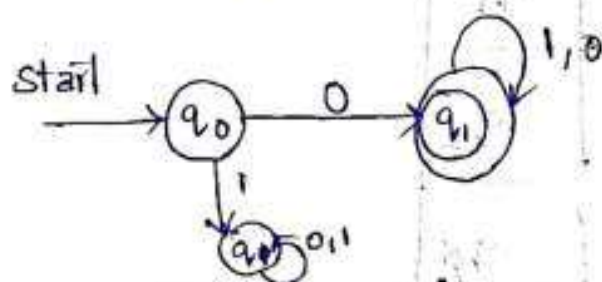
→ Entry corresponds to state, which results from $\delta(q, a)$.

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$* q_2$	q_2	q_2

→ Construct DFA which accepts all strings over alphabet $\Sigma = \{0, 1\}$ starts with "0".

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

$$L = \{0, 01, 000, 001, 010, 011, \dots\}$$



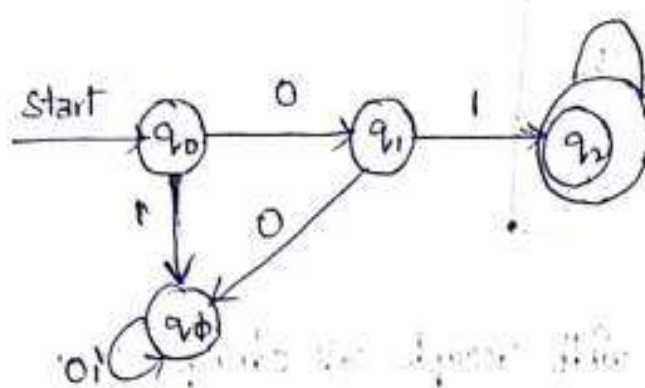
	0	1
$\rightarrow q_0$	q_1	q_2
$* q_1$	q_1	q_1
$* q_2$	q_1	q_1

	0	1
$\rightarrow q_0$	q_1	q_ϕ
q_ϕ	q_ϕ	q_ϕ
$* q_1$	q_1	q_1

→ Construct DFA which accepts all string over alphabet $\Sigma = \{0, 1\}$ starts with "01".

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

$$L = \{01, 10, 010, 011, 0100, 0101, \dots\}$$

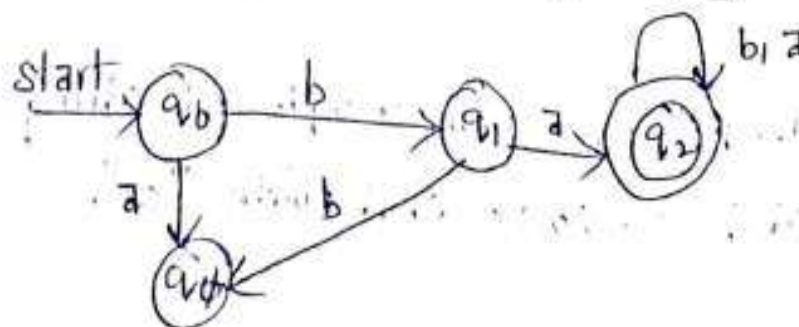


	0	1
$\rightarrow q_0$	q_1	q_ϕ
q_1	q_ϕ	q_2
q_2	q_2	q_2
q_ϕ	q_ϕ	q_ϕ

→ Construct DFA with accepts all string over alphabet $\Sigma = \{a, b\}$ starts with "ba".

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

$$L = \{ba, bab, babb, baba, \dots\}$$

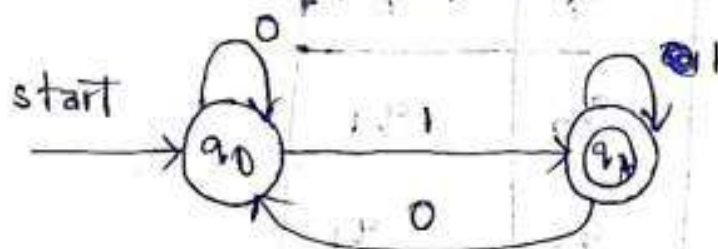


	a	b
→ q ₀	q ₀	q ₁
q ₁	q ₂	q ₀
q ₂	q ₂	q ₂
q ₀	q ₀	q ₀

→ construct DFA for all strings over alphabet $\Sigma = \{0, 1\}$ which ends with "1"

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

$$L = \{1, 01, 11, 001, 011, 101, 111, \dots\}$$

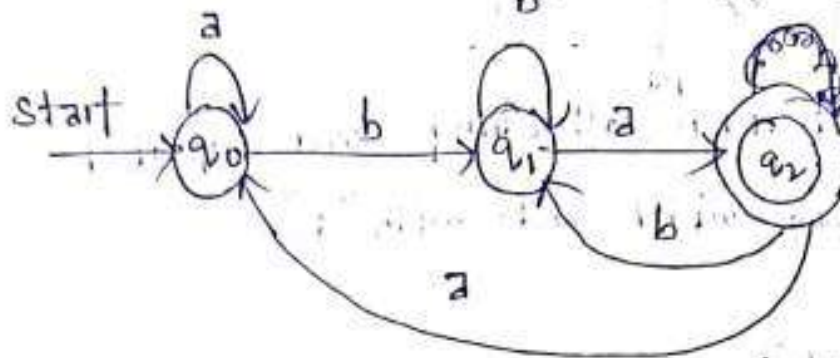


	0	1
→ q ₀	q ₀	q ₁
q ₁	q ₀	q ₁

→ Construct DFA for all strings over the alphabet $\Sigma = \{a, b\}$ which ends with "ba"

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

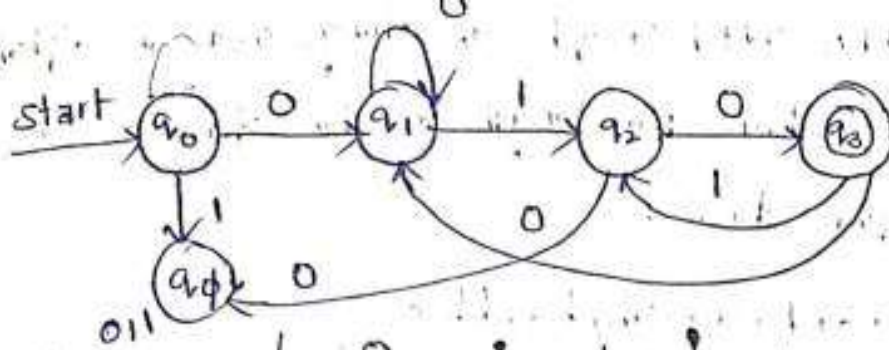
$$L = \{ba, aba, bba, aaba, abba, baba, bbbba, \dots\}$$



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

→ Construct DFA for all strings over the alphabet $\Sigma = \{0, 1\}$ which ends with 010

$$L = \{010, 0010, 1010, \dots\}$$

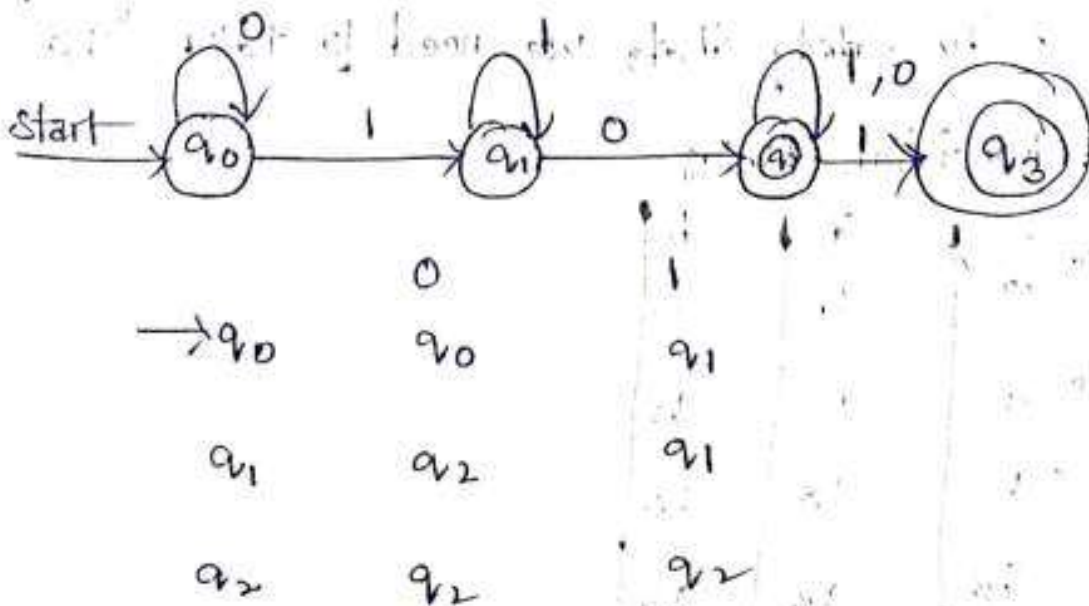


	0	1
→ q ₀	q ₁	q _φ
⇒ q ₁	q ₁	q ₂
q ₂	q ₃	q _φ
* q ₃	q ₁	q ₂
q _φ	q _φ	q _φ

→ construct DFA for all strings over the alphabet "011" which contains 10φ

010, 101, 1101

$L = \{ 010, 101, 1101, 1010, 1011, 0010, \dots \}$



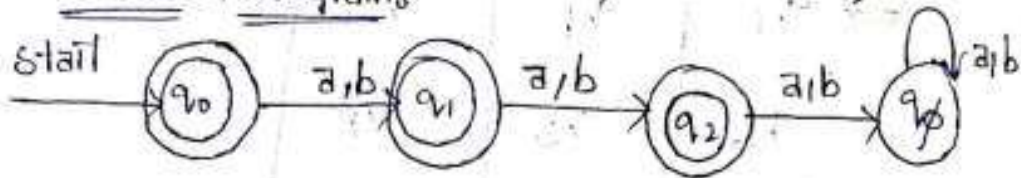
	0	1
→ q ₀	q ₀	q ₁
q ₁	q ₁	q ₂
q ₂	q ₂	q ₃

→ Construct DFA which accepts all strings over an alphabet $\Sigma = \{a, b\}$ where the length of the string is less than or equals to "2".

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

→ It depends on the length of strings not min "LOS".
 → The length of the string = 2 $\Rightarrow$ 3 states

Transition Diagrams:



→ This DFA satisfies all the strings of the length of the string is " ≤ 2 ".

→ There is no transition of q_2 if we take self transition, then we get more than 2 states.

→ But there is no numbers matching with the more than 2 in the language.

→ To satisfy that, we need to take " q_ϕ ".

Transition Table:

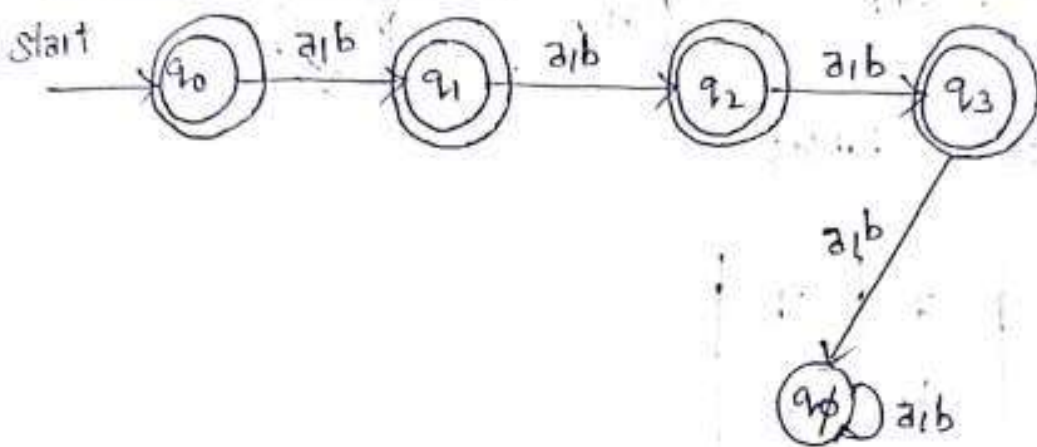
	a	b
$\rightarrow q_0$	q_1	q_1
q_1	q_2	q_2
q_2	q_ϕ	q_ϕ
q_ϕ	q_ϕ	q_ϕ

→ construct DFA which accepts all strings over an alphabet $\Sigma = \{a, b\}$ where the length of the string is less than or equals to "3".

$\Sigma = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}$

→ The length of the string is 3 $\Rightarrow$ 4 states,

Transition Diagram:-



Transition Table:-

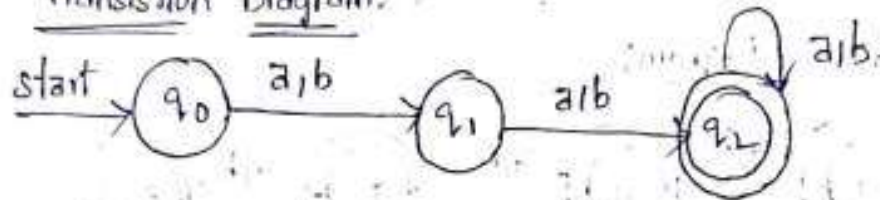
	a	b
* q_0	q_1	q_1
* q_1	q_2	q_2
* q_2	q_3	q_3
* q_3	q_ϕ	q_ϕ
q_ϕ	q_ϕ	q_ϕ

$\Rightarrow$ Construct DFA which accepts all the strings over the alphabet $\Sigma = \{a, b\}$ where the length of the string is greater than or equal to 3

$L = \{aaa, aab, aba, abb, \dots\}$

$\Rightarrow$ The length of the string is 3 states

Transition Diagram:-



Transition Table:-

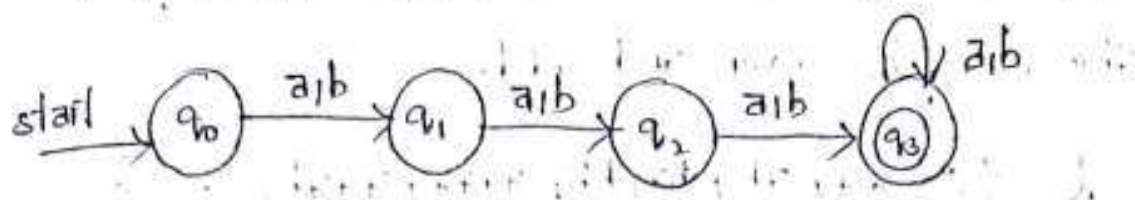
	a	b
$\rightarrow q_0$	q_1	q_1
q_1	q_2	q_2
$*q_2$	q_2	q_2

$\Rightarrow$ Construct DFA which accepts all the strings over the alphabet $\Sigma = \{a, b\}$ where the length of the string is "greater than or equal to 3"

$L = \{aaa, aab, aba, abb, aaaa, aaab, \dots\}$

The length of the string is 3 = 4 states:

Transition Diagram:-



Transition Table:-

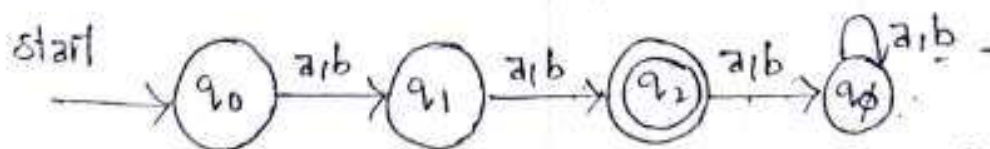
	a	b
→ q ₀	q ₁	q ₂
q ₁	q ₂	q ₂
q ₂	q ₃	q ₃
q ₃	q ₃	q ₃

→ construct DFA which accepts all the strings over the alphabet $\Sigma = \{a, b\}$ where the "length of the string" is "2".

$$\Sigma = \{aa, ab, ba, bb\}$$

→ The length of the string is 2 = 3 states

Transition Diagram:-



Transition Table:-

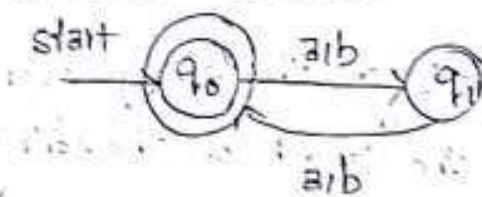
	a	b
→ q ₀	q ₁	q ₁
q ₁	q ₂	q ₂
q ₂	q ₃	q ₃
q ₃	q ₃	q ₃

→ construct the DFA which accepts all the strings over an alphabet $\Sigma = \{a, b\}$ where the length of the string is even and odd.

$L = \{ \epsilon, aa, ab, ba, bb, aaaa, aaab, \dots \}$

→ length of the string is = even and odd $\Rightarrow$ Takes only 2 states.

Transition Diagram:- For even



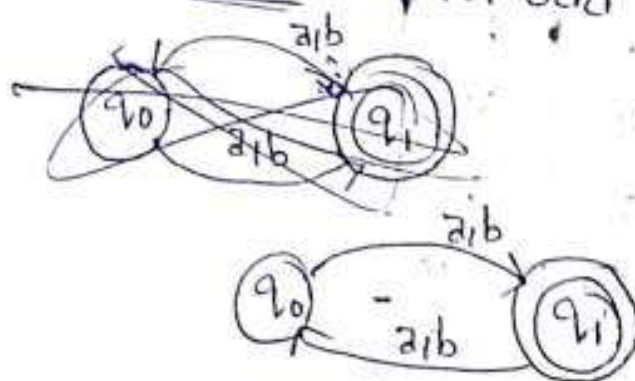
Even numbers [q_0 is final state]

Odd numbers [q_1 is final state]

Transition Table:-

	a	b
$\rightarrow q_0$	q_1	q_1
q_1	q_0	q_0

Transition Diagrams, For odd



Transition Table:-

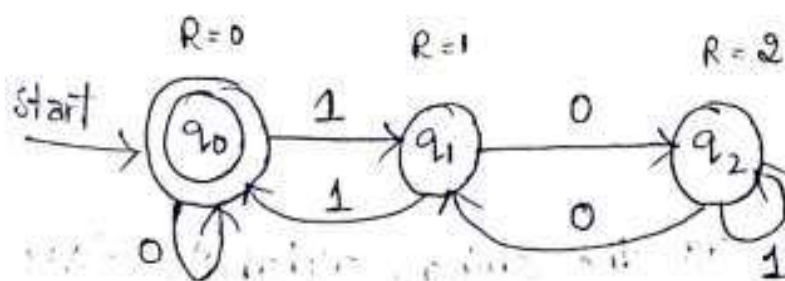
	a	b
$\rightarrow q_0$	q_1	q_1
$* q_1$	q_0	q_0

→ Construct DFA which accepts all the strings over an alphabet $\Sigma = \{0,1\}$ where the binary integers are divisible by 3. $\hookrightarrow$ The lengths of the

$L = \{0, 11, 110, 1001, 1100, 1111, \dots\}$

→ The length of string is depends on the "remainder of the number".

→ The length of the string is "3".



Transition Table:-

	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_2

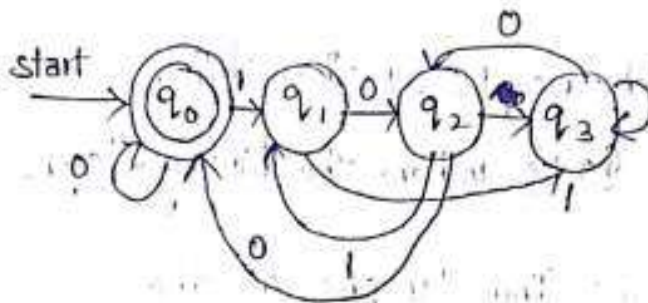
Decimal number	Binary equivalent	Remainder
0	0	0
1	1	1
2	10	2
3	11	0
4	100	1
5	101	0

→ construct DFA which accepts all the strings over alphabet $\Sigma = \{0,1\}$ where binary integers are divisible by "4".

$$d = \{0, 100, 1000, 1100, 1110, \dots\}$$

remainder $R = 0, 1, 2, 3$

- Decimal binary remainder



0	0	0
1	1	1
2	10	2
3	11	3
4	100	0
5	101	1
6	110	2
7	111	3

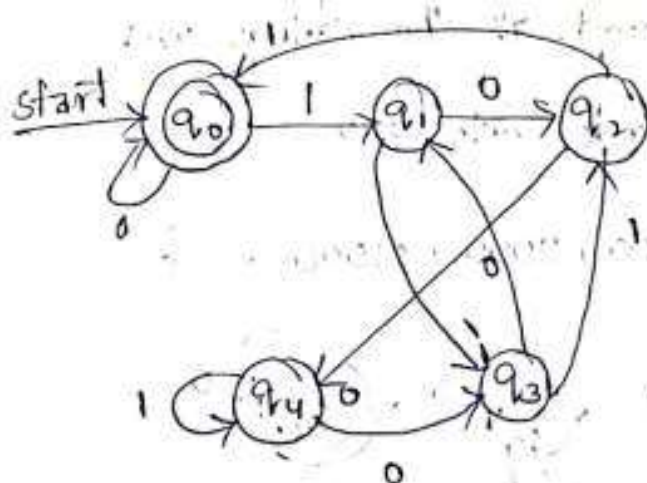
Transition Table

	0	1
→ q ₀	q ₀	q ₁
q ₁	q ₂	q ₃
q ₂	q ₀	q ₁
q ₃	q ₂	q ₃

→ construct DFA all the strings alphabet $\Sigma = \{0,1\}$ where are divisible by "5".

$$d = \{0, 101, 1010, 1111, \dots\}$$

$R = 0, 1, 2, 3, 4$



Decimal binary remainder

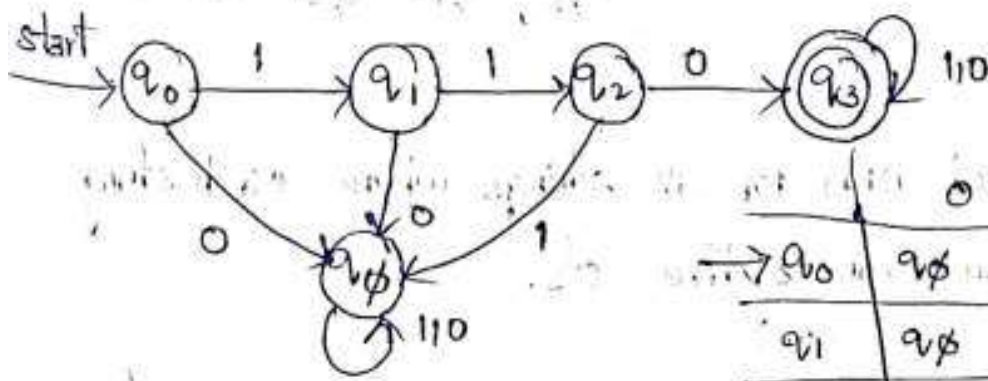
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	0
6	110	1
7	111	2
8	1000	3
9	1001	4

Transition Table:-

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

→ construct DFA which accepts all the strings over an alphabet $\Sigma = \{0, 1\}$ starts with 110.

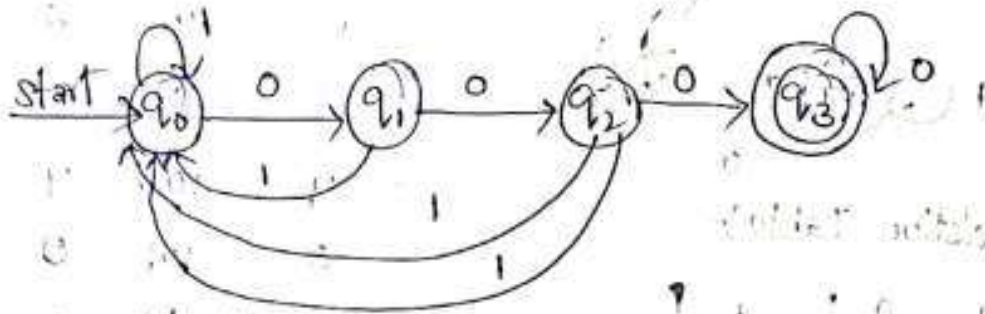
$$L = \{110, 1101, 1100, 11000, 11001, 11010, \dots\}$$



	0	1
$\rightarrow q_0$	q_ϕ	q_1
q_1	q_ϕ	q_2
q_2	q_3	q_ϕ
q_3	q_3	q_3
q_ϕ	q_ϕ	q_ϕ

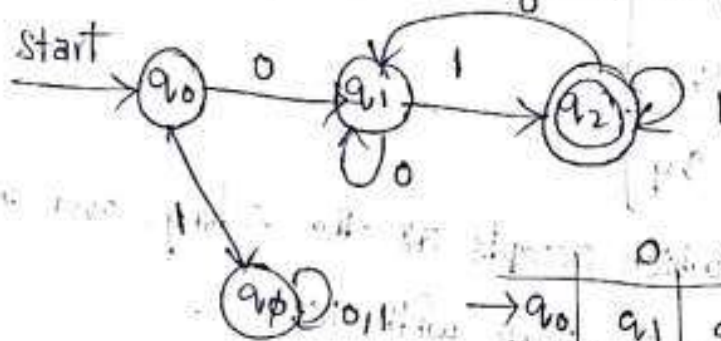
→ construct DFA which accepts all the strings over an alphabet $\Sigma = \{0,1\}$ ends with '000'

$L = \{000, 1000, 0000, 11000, 01000, \dots\}$



→ construct DFA $\Sigma = \{0,1\}$ starts with 0 and ends with 1.

$L = \{01, 001, 011, 0001, 0101, 0111, 0001, \dots\}$



	0	1
q_0	q_1	q_ϕ
q_1	q_1	q_2
q_2	q_1	q_2

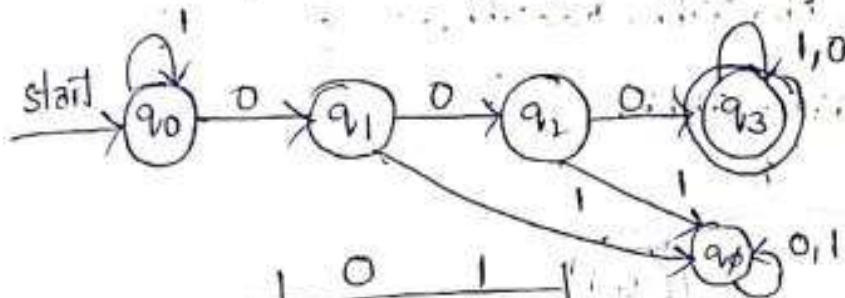
→ construct DFA for all strings where each string has 'three consecutive 0's'.

(or)

Construct DFA for all strings which have/contains '000'.

$\lambda = \{ 000, 1000, 0001, 0010, 00010, 10001, \dots \}$

length of the string = 3 then $n+1$ states = 4

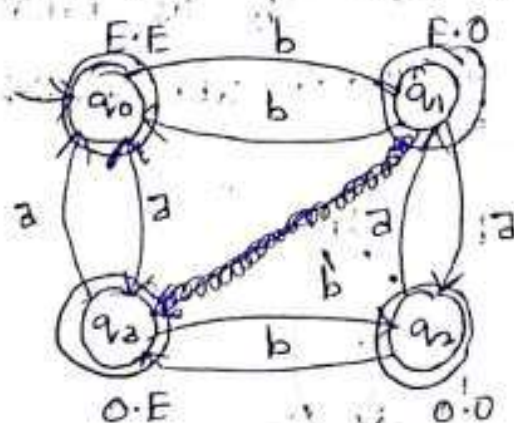


	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_4
q_2	q_3	q_4
$*q_3$	q_3	q_3

→ construct DFA's for:

- 1) Even number of a's and even number of b's.
- 2) Even number of b's and odd number of a's.
- 3) Even number of a's and even number of b's.
- 4) odd number of a's and odd number of b's.

1) Even number of a's and odd number of b's.

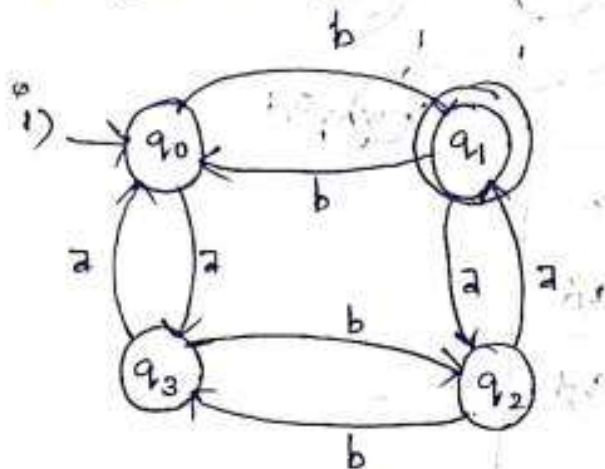


i) $L = \{b, bbb, aab, aabbb, \dots\}$

ii) $L = \{a, aaa, abb, aaabb, \dots\}$

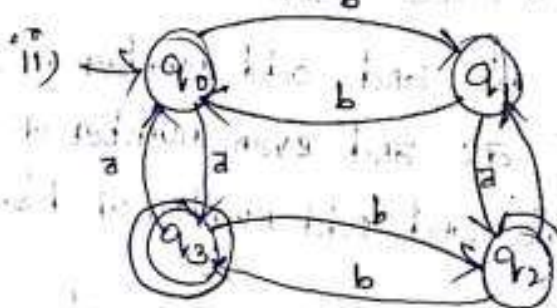
iii) $L = \{e, aa, bb, aabb, aaaaabb, \dots\}$

iv) $L = \{ab, aab, abbb, \dots\}$



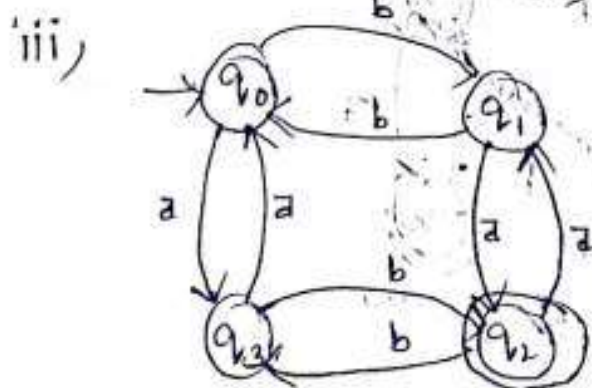
here we take " q_1 " as final state.

$L = \{b, bbb, aab, aabbb, \dots\}$



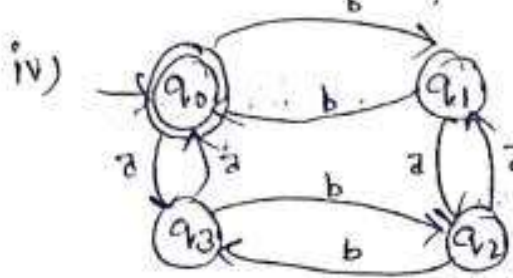
here we take " q_3 " as final state

$L = \{a, aaa, abb, aaabb, \dots\}$



here we take " q_1 " as final state

$L = \{ \epsilon, aa, bb, aabb, aaaabb, \dots \}$



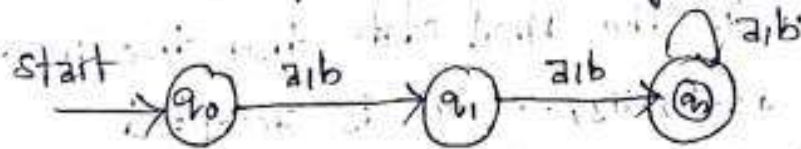
Here we take " q_0 " as final state

$L = \{ ab, aaab, abbb, \dots \}$

→ Design a DFA that $L(M) = \{ x/x \text{ is a string of 0's and 1's and } |x| \geq 2 \}$

$L = \{ aa, ab, ba, bb, aaa, aab, aba, abb, \dots \}$

→ The length of the string is ≥ 2 states



	a	b
→ q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_2	q_2

How DFA process a string:

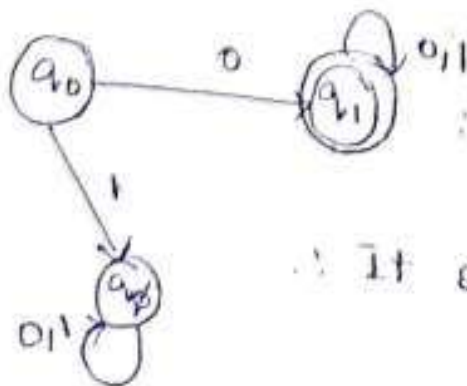
Let $a_1, a_2, a_3, \dots, a_n$ is a string (or)

Some input sequences.

- start DFA from initial state (q_0)
- consult transition from $\delta(q_0, a_1)$ to find state to which transition takes place from q_0 on input symbol a_1 .
- for the next input symbol evaluate $\delta(q_1, a_2)$ to find next state (q_2).
- continue this process to find $q_3, q_4, \dots, q_n$.
- If q_n is the final state then the input sequence $a_1, a_2, a_3, \dots, a_n$ is accepted otherwise it is rejected.

Example:-

0010010



It satisfies the data.

Extended Transition Function:-

An extended transition function describes, what happens when we start from any state and follow some input sequence.

→ Extended Transition function is denoted by $\hat{\delta}$.

→ Extended Transition function we give state and input sequence.

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) = \dots = p$$

Where $w = xa$.

Language of DFA:-

The language of DFA is denoted by $L(A)$.

→ It is defined as

$$L(A) = \{ w \mid \delta(q, w) \text{ is in } F \}$$

$$L(A) = \{ w \mid \delta(q, w) \text{ is in } F \}$$

→ If L is a language of DFA $(L(A))$ for some DFA "A" then L is called a "regular language".

Non-Deterministic Finite Automata:- (NFA)

→ In which the finite automata can exist in any number of states on input from a state.

→ NFA is also a "5-tuple" notation. It consists of set of $(Q, \Sigma, \delta, q_0, F)$.

→ Q : Set of states

Σ : Set of input symbols

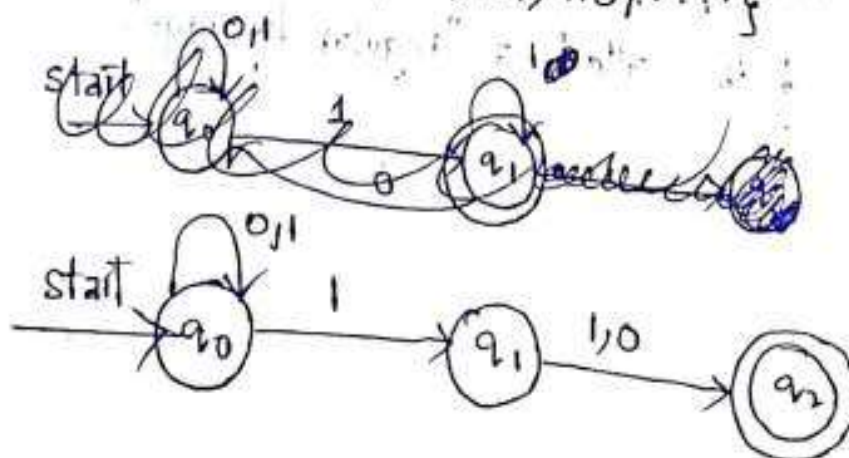
δ : A transition function, $\delta(q, a)$, which takes two arguments "state and input symbol" and give one output.

q_0 : starting state.

F : set of final states (or) Accepting states.

→ Construct NFA which accepts all strings whose second last symbol is "1".

$L = \{10, 11, 010, 011, 111, 110, \dots\}$



	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
q_1	q_2	q_2
$*q_2$	ϕ	ϕ

Extended Transition Function:-

The Language of NFA:-

It is also denoted by $L(A)$ and it is defined as $L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

Extended Transition Function:-

$$\hat{\delta}(q_0, w) = \{r_1, r_2, \dots, r_n\}$$

Where, $r_1, r_2, \dots, r_n$ are final states.

Finite Automata with epsilon Transitions:-

$\rightarrow$ It is represented as "NFA- ϵ ". (or) " ϵ -NFA".

$\rightarrow$ It is a "5 tuple" notation, where

$$A = (Q, \Sigma, \delta, q_0, F) \text{ - transition.}$$

Q = set of states

$$\Sigma = \Sigma \cup \{\epsilon\}$$

✓ $\delta =$ A transition function. $\delta(q, a)$, which takes two arguments "state and input symbol" and give one output.

$q_0 =$ starting state

$F =$ set of Final states/Accepting states.

Uses of Epsilon:-

i) Each epsilon along the path is "Invisible", i.e., it contributes nothing to string along the path.

→ Construct epsilon NFA which accepts decimal numbers consisting of i) an optional "+" or "-"

ii) A string of digits

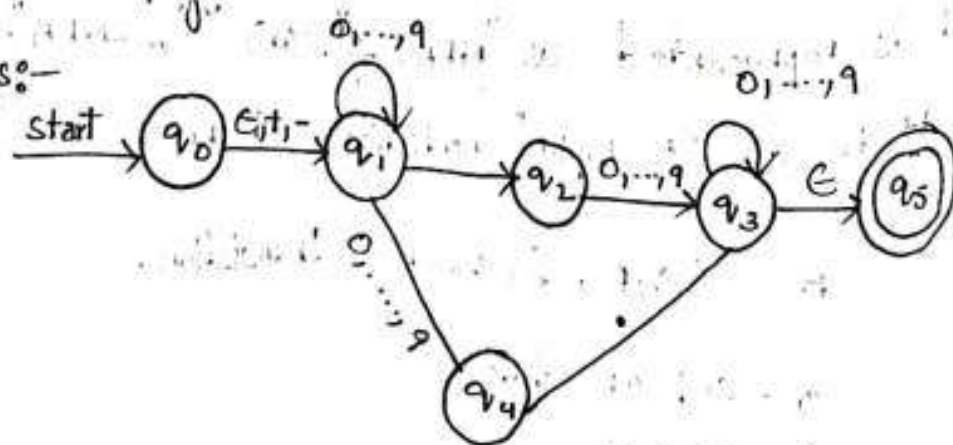
iii) A decimal point

iv) Another string of digits.

v) Either these string of digits can be empty but

vi) Atleast one of string of digits must be non empty.

Ans:-



Extended Transition Function:-

$$\hat{\delta}(q_0, w) = \bigcup_{j=1}^m (\epsilon\text{-closure}(r_j))$$

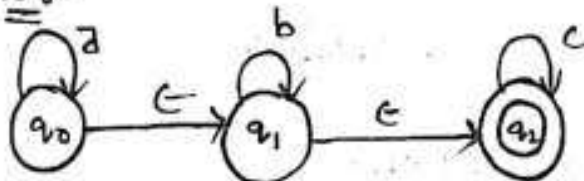
$$= \epsilon\text{-closure}(r_1) \cup \epsilon\text{-closure}(r_2) \cup \dots$$

$$\epsilon\text{-closure}(r_m)$$

Epsilon Closure:-

ϵ -closure of a state Q consists of every state that can be reached from Q along a path with " ϵ ".

Example:-



ϵ -closure(q_0):- There is an invisible transition to the itself. " $\{q_0, q_1, q_2\}$ "

ϵ -closure(q_1): " $\{q_1, q_2\}$ "

ϵ -closure(q_2): " $\{q_2\}$ "

Eliminating ϵ -Transitions:-

Conversion of " $\text{NFA} - \epsilon$ " (or) " $\epsilon\text{-NFA}$ " NFA without Transitions of " ϵ ":-

→ Find ϵ -closures for all states of the given

ϵ -NFA.

→ Find Transitions for each state/input symbol.

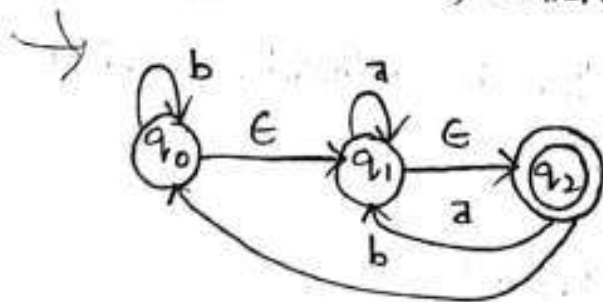
→ ~~construct~~ $\hat{\delta}(q, a) = \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q), a))$

→ construct NFA without epsilon Transitions based on the computations of Transition function.

→ Decide Final state as (F')

i) $F' = F \cup \{q_0\}$ if $\epsilon\text{-closure}(q_0)$ is having F as a member.

ii) $F' = F$, otherwise



Step 1:- Find ϵ -closures for all states

$$\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$$

$$\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$$

$$\epsilon\text{-closure}(q_2) = \{q_2\}$$

Step 2:- find the Transitions for each state on input symbol.

$$\begin{aligned}\Rightarrow \hat{\delta}(q_1, a) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_0), a))) \\ &= \epsilon(\text{closure}(\delta((q_0, q_1, q_2), a))) \\ &= \epsilon(\text{closure}(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))) \\ &= \epsilon(\text{closure}(\phi \cup q_1 \cup q_2)) \\ &= \epsilon(\text{closure}(q_1)) \\ &= \{q_1, q_2\}\end{aligned}$$

Step 3:-

$$\begin{aligned}\Rightarrow \hat{\delta}(q_1, b) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_0), b))) \\ &= \epsilon(\text{closure}(\delta((q_0, q_1, q_2), b))) \\ &= \epsilon(\text{closure}(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))) \\ &= \epsilon(\text{closure}(q_0 \cup \phi \cup q_0)) \\ &= \epsilon(\text{closure}(q_0)) \\ &= \{q_0, q_1, q_2\}\end{aligned}$$

$$\begin{aligned}\Rightarrow \hat{\delta}(q_1, a) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1), a))) \\ &= \epsilon(\text{closure}(\delta((q_1, q_2), a))) \\ &= \epsilon(\text{closure}(\delta(q_1, a) \cup \delta(q_2, a)))\end{aligned}$$

$$= \epsilon \cdot (\text{closure}(q_1 \cup q_1))$$

$$= \epsilon \cdot (\text{closure}(q_1))$$

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

$$\Rightarrow \hat{\delta}(q_1, b) = \epsilon \cdot (\text{closure}(\delta(\epsilon\text{-closure}(q_1), b)))$$

$$= \epsilon \cdot (\text{closure}(\delta(q_1, q_2), b))$$

$$= \epsilon \cdot (\text{closure}(\delta(q_1, b) \cup \delta(q_2, b)))$$

$$= \epsilon \cdot (\text{closure}(q_0 \cup q_0))$$

$$= \epsilon \cdot (\text{closure}(q_0))$$

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

$$\Rightarrow \hat{\delta}(q_2, a) = \epsilon \cdot (\text{closure}(\delta(\epsilon\text{-closure}(q_2), a)))$$

$$= \epsilon \cdot (\text{closure}(\delta(q_2), a))$$

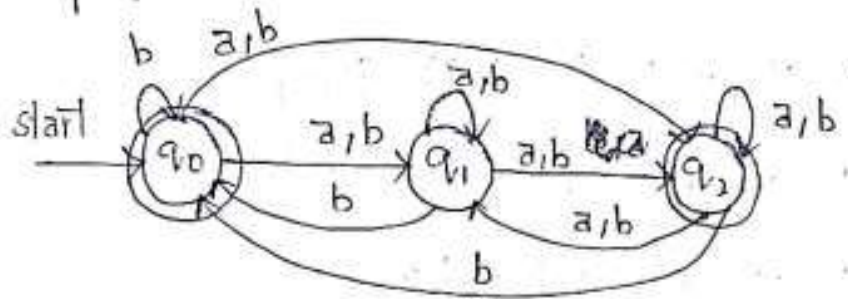
$$= \epsilon \cdot (\text{closure}(\delta(q_2, a)))$$

$$= \epsilon \cdot (\text{closure}(q_1))$$

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

$$\begin{aligned}
 \Rightarrow \hat{\delta}(q_2, b) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2), b))) \\
 &= \epsilon(\text{closure}(\delta(q_2), b)) \\
 &= \epsilon(\text{closure}(\delta(q_2, b))) \\
 &= \epsilon(\text{closure}(q_0)) \\
 &= \{q_0, q_1, q_2\}
 \end{aligned}$$

Step 3:-



Step 4:-

$F' = F \cup \{q_0\}$ if $\epsilon\text{-closure}(q_0)$ is
 having F as a member.
 Otherwise $F' = F$.

$$F = \{q_2\}$$

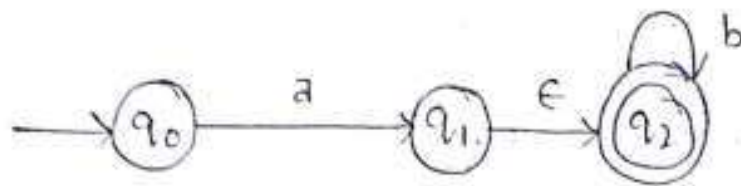
$$\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$$

$$F' = F \cup \{q_0\}$$

$$= \{q_2\} \cup \{q_0\}$$

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

→ convert following ϵ -NFA to the NFA without ϵ .



Ans:-

Step 1:- Find ϵ closures for all states

$$\epsilon\text{-closure}(q_0) = \{q_0\}$$

$$\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$$

$$\epsilon\text{-closure}(q_2) = \{q_2\}$$

Step 2:- Find the Transitions for each state on input symbol.

$$\begin{aligned}
 \Rightarrow \delta(q_0, a) &= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0), a)) \\
 &= \epsilon\text{-closure}(\delta(\emptyset, a)) \\
 &= \epsilon\text{-closure}(q_1) \\
 &= \{q_1, q_2\}
 \end{aligned}$$

$$\begin{aligned}
 \Rightarrow \delta(q_0, b) &= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0), b)) \\
 &= \epsilon\text{-closure}(\delta(q_0, b)) \\
 &= \epsilon\text{-closure}(\emptyset) \\
 &= \emptyset
 \end{aligned}$$

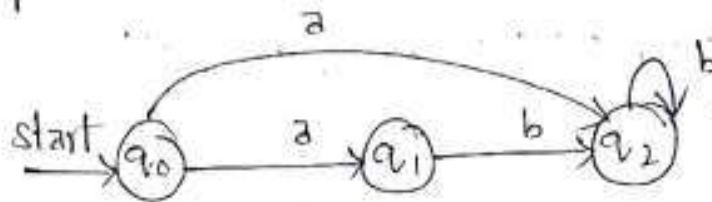
$$\begin{aligned}
 \Rightarrow \hat{\delta}(q_1, a) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1), a))) \\
 &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1, a)))) \\
 &= \phi
 \end{aligned}$$

$$\begin{aligned}
 \Rightarrow \hat{\delta}(q_1, a) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1), b))) \\
 &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_1, b)))) \\
 &= \epsilon(\text{closure}(q_2)) \\
 &= q_2 \notin q_2
 \end{aligned}$$

$$\begin{aligned}
 \Rightarrow \hat{\delta}(q_2, a) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2), a))) \\
 &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2, a)))) \\
 &= \phi \notin q_2
 \end{aligned}$$

$$\begin{aligned}
 \Rightarrow \hat{\delta}(q_2, b) &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2), b))) \\
 &= \epsilon(\text{closure}(\delta(\epsilon\text{-closure}(q_2, b)))) \\
 &= \epsilon(\text{closure}(q_2)) \\
 &= q_2 \notin q_2
 \end{aligned}$$

Step 3:-



Step 4:-

$F' = F \cup \{q_0\}$ if ϵ -closure(q_0) is
having F as a member
Otherwise $F' = F$

$F = \{q_2\}$ ϵ -closure(q_0) = q_0

Since F is not a member of
 ϵ -closure(q_0) so, $F' = F$

$F' = \{q_2\}$

Conversion of NFA to DFA:-

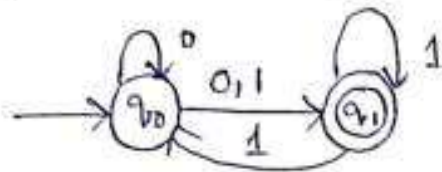
Step 1: start with initial state and find its
transitions.

Step 2: Star if there are any new states find
transitions for each of them.

Step 3:- Continue step 2 until there are no new transitions.

Step 4:- Mark all the states which has the final state of NFA as final state of DFA.

1) convert the following NFA into DFA / Find the equivalent DFA for the given NFA.



Step 1:- Find the transitions which starts with initial state.

	0	1
$\rightarrow q_0$	$[q_0, q_1]$	q_1
$[q_0, q_1]$	$[q_0, q_1]$	q_1
q_1	ϕ	$[q_0, q_1]$

Step 2:- Find the transitions for the new states occurred

$$\delta(q_0, 0) = \{q_0, q_1\}$$

$$\delta(q_0, 1) = \{q_1\}$$

$$\delta([q_0, q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$$

$$= [q_0, q_1] \cup \phi$$

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

$$\delta([q_0, q_1], 1) = \delta(q_0, 1) \cup \delta(q_1, 1)$$

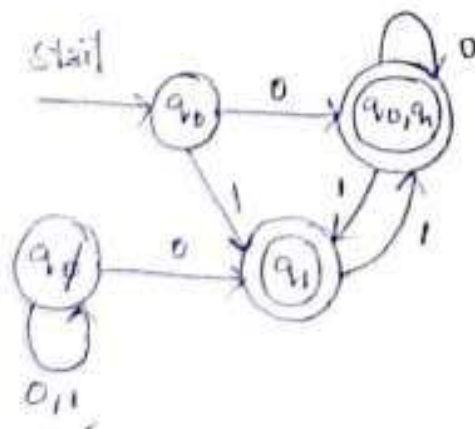
$$= q_1 \cup q_1$$

$$= q_1$$

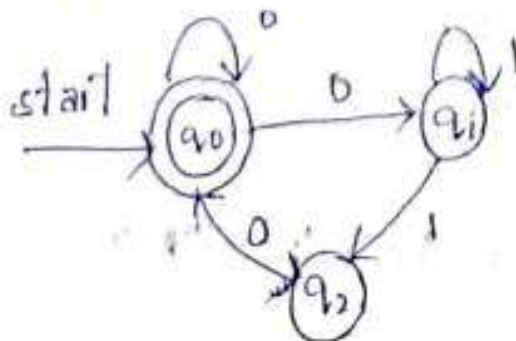
$$\delta(q_1, 0) = \phi$$

$$\delta(q_1, 1) = \{q_0, q_1\}$$

Step 3:- Continue the previous step if there are no new states, after that we need draw the transition diagram from the table.



2) Determine the equivalent DFA to the given NFA.



Step 1:-

$\rightarrow q_0$ ~~$[q_0, q_1]$~~

	0	1
$\rightarrow q_0$	$[q_0, q_1]$	ϕ
$[q_0, q_1]$	$[q_0, q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	q_0	$[q_1, q_2]$
$q\phi$	$q\phi$	$q\phi$

Step 2:-

$$\delta(q_0, 0) = \{q_0, q_1\}$$

$$\delta(q_0, 1) = \phi$$

$$\begin{aligned} \delta([q_0, q_1], 0) &= \delta(q_0, 0) \cup \delta(q_1, 0) \\ &= q_0 \cup \phi \cup q_1 \cup [q_0, q_1] \cup \phi \\ &= q_0 \cup [q_0, q_1] \end{aligned}$$

$$\begin{aligned} \delta([q_0, q_1], 1) &= \delta(q_0, 1) \cup \delta(q_1, 1) \\ &= \phi \cup [q_1, q_2] \\ &= [q_1, q_2] \end{aligned}$$

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

$$\delta(q_1, 0) = \emptyset$$

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

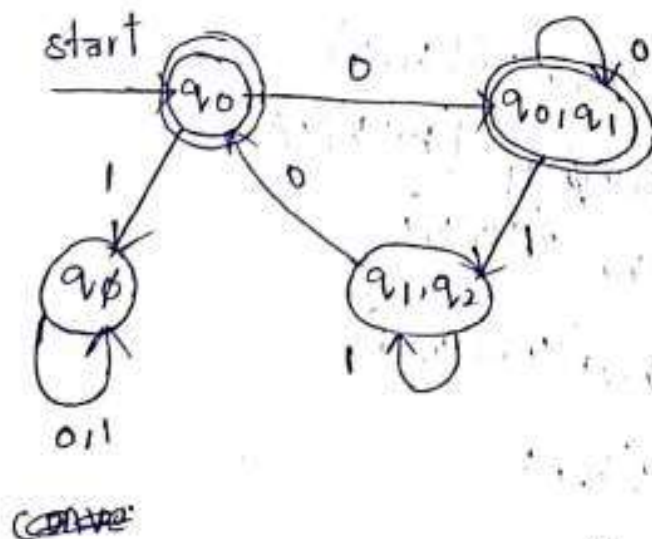
$$\delta(q_2, 0) = q_0$$

$$\delta(q_2, 1) = \emptyset$$

$$\begin{aligned} \delta((q_1, q_2), 0) &= \delta(q_1, 0) \cup \delta(q_2, 0) \\ &= (\emptyset) \cup q_0 \\ &= q_0 \end{aligned}$$

$$\begin{aligned} \delta((q_1, q_2), 1) &= \delta(q_1, 1) \cup \delta(q_2, 1) \\ &= (q_1, q_2) \cup \emptyset \\ &= q_1, q_2 \end{aligned}$$

step 3:-



conversion of E-NFA to DFA:-

Step 1: Take epsilon-closure for the starting state of NFA as the starting state of DFA

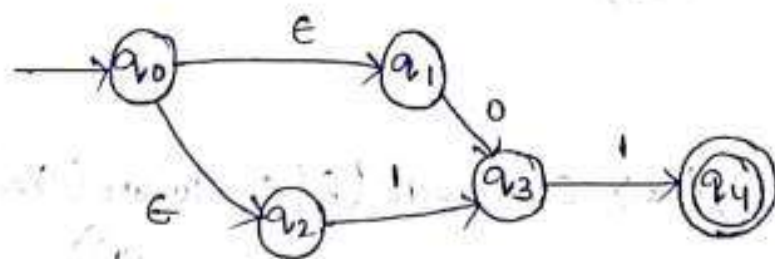
Step 2: Find the states for each input from the current state.

Step 3: If we find a new state, take it as a current state repeat step 2.

Step 4: & repeat step 2 and step 3 until there is no new state.

Step 5: Mark the states of DFA as final states which contains the final state of NFA.

1) convert the given e-NFA into DFA



Step 1: finding the ϵ -closures for the starting state of NFA as the starting state of DFA

$$\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$$

	0	1
$\rightarrow [q_0, q_1, q_2]$	$\{q_3\}$	$\{q_3\}$
q_3	ϕ	q_4
q_4	ϕ	ϕ
q_5	ϕ	ϕ

$$\hat{\delta}([q_0, q_1, q_2], 0) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1, q_2), 0))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1, q_2), 0))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$$

$$\Rightarrow \epsilon\text{-closure}(\phi \cup q_3 \cup \phi)$$

$$\Rightarrow \epsilon\text{-closure}(q_3)$$

$$\Rightarrow q_3$$

$$\hat{\delta}([q_0, q_1, q_2], 1) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1, q_2), 1))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_0, q_1, q_2), 1)$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$$

$$\Rightarrow \epsilon\text{-closure}(\phi \cup \phi \cup q_3)$$

$$\Rightarrow \epsilon\text{-closure}(q_3)$$

$$\Rightarrow q_3$$

$$\hat{\delta}([q_3], 0) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_3), 0))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_3), 0)$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_3, 0))$$

$$\Rightarrow \phi$$

$$\hat{\delta}([q_3], 1) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_3), 1))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_3), 1)$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_3, 1))$$

$$\Rightarrow q_4$$

$$\hat{\delta}([q_4], 0) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_4), 0))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_4), 0)$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_4, 0))$$

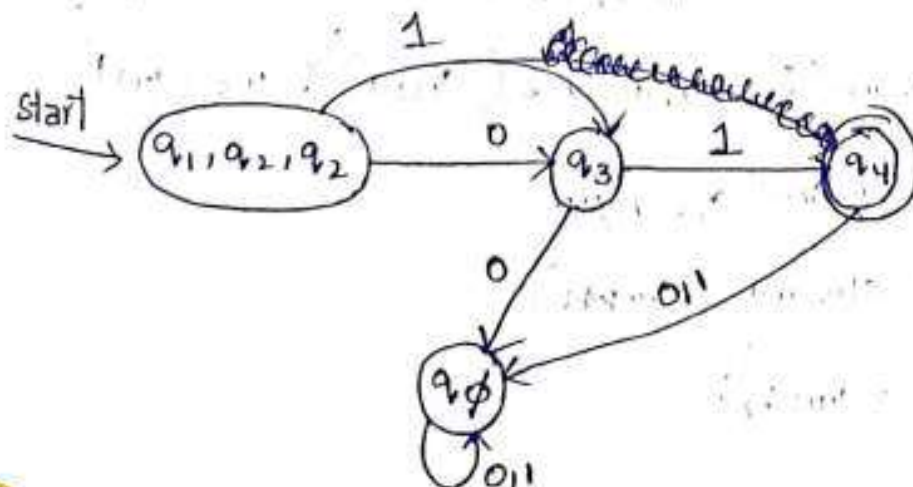
$$\Rightarrow \phi$$

$$\hat{\delta}[q_4, 1] \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_4), 1))$$

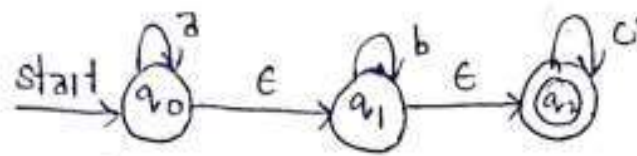
$$\Rightarrow \epsilon\text{-closure}(\delta(q_4), 1)$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_4, 1))$$

$$\Rightarrow \phi$$



→ Determine the equivalent DFA for the given ϵ -NFA.



i) ϵ -closure(q_0) = $\{q_0, q_1\}$

ii) ϵ -closure(q_1) = $\{q_1, q_2\}$

iii) ϵ -closure(q_2) = $\{q_2\}$

	a	b	c
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_1, q_2\}$	$\{q_2\}$
$\{q_1, q_2\}$	ϕ	$\{q_1, q_2\}$	

$$\hat{\delta}([q_0, q_1], a) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1), a))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, a))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$$

$$\Rightarrow \epsilon\text{-closure}(q_0 \cup \phi)$$

$$\Rightarrow \epsilon\text{-closure}(q_0)$$

$$\Rightarrow \{q_0, q_1\}$$

$$\hat{\delta}([q_0, q_1], b) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1), b))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b)))$$

$$\Rightarrow \epsilon\text{-closure}(\emptyset \cup q_1 \cup \emptyset)$$

$$\Rightarrow \epsilon\text{-closure}(q_1)$$

$$\Rightarrow \{q_1, q_2\}$$

$$\hat{\delta}([q_0, q_1, q_2], c) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, q_1, q_2), c))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0, c) \cup \delta(q_1, c) \cup \delta(q_2, c)))$$

$$\Rightarrow \epsilon\text{-closure}(\emptyset \cup \emptyset \cup q_2)$$

$$\Rightarrow \{q_2\}$$

$$\hat{\delta}([q_1, q_2], a) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, q_2), a))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, a) \cup \delta(q_2, a)))$$

$$\Rightarrow \epsilon\text{-closure}(\emptyset \cup \emptyset)$$

$$\Rightarrow \emptyset$$

$$\hat{\delta}([q_1, q_2], b) \Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, q_2), b))$$

$$\Rightarrow \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1, b) \cup \delta(q_2, b)))$$

$$\Rightarrow \epsilon\text{-closure}(q_1 \cup \emptyset)$$

$$\Rightarrow \{q_1, q_2\}$$

NFA - Applications: Text Search.

→ It is used for web search (www) and "finding information from text".

Finding strings in Text:-

1) Inverted indices

2) Finite Automata

↓

It is NFA

maintain "some set" of words & occurrences

• It requires more memory to travel to search indices

1) Take a transition on an initial state upon each input symbol.

→ That's why it is used.

2) To accept the string $a_1, a_2, \dots, a_k$ Take

transition from q_0 to q_1 on a_1

→ This is not suitable for where documents require rapid change.

1) Take Transition from

q_1 to q_2 on a_2 and

so on ... Then q_k

becomes the final state

Regular Expression: It is a simple expression to describe a language that the DFA accept.

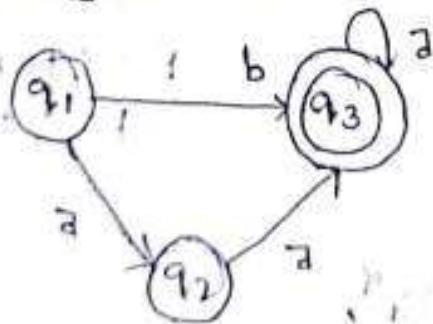
Finite Automata:-

Step 1: Take equation for each state based on incoming edges.

Step 2: Add ϵ to eqn of initial state.

Step 3: Simplify the eqns. using Arden's method and find regular expression for the final state.

1) Convert given DFA into regular expression.



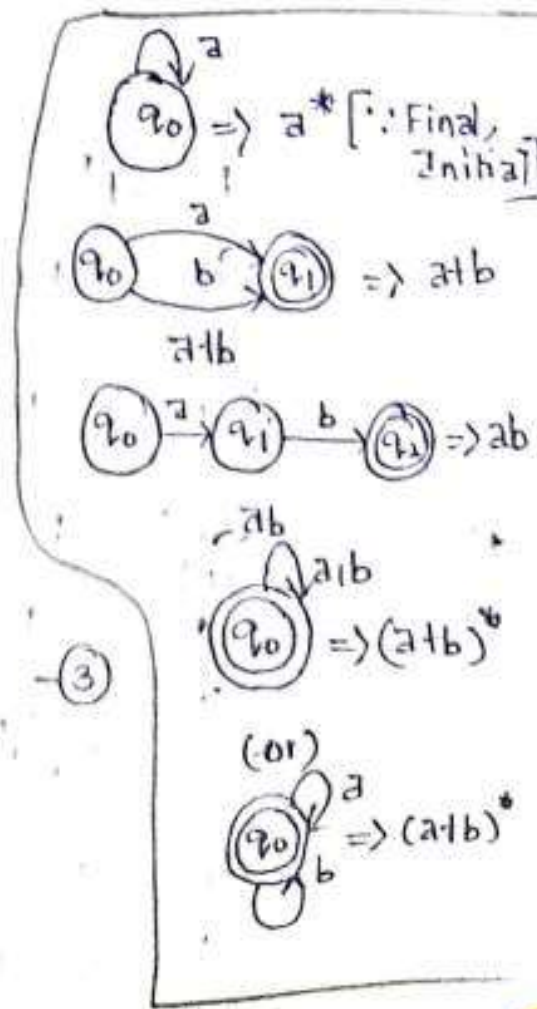
Step 1: Incoming edges

1) $q_1 = \emptyset$ - (1)

2) $q_2 = q_1 a$ - (2)

3) $q_3 = q_1 b + q_2 a + q_3 a$ - (3)

Step 2: $q_1 = \epsilon$ - (1)



Step 3:- $R = Q + RP \Rightarrow R = QP^*$

Substitute eq ① in eq ②

$$q_2 = q_1 a$$

$$= \epsilon a$$

$$q_2 = a \text{ --- (4)}$$

Substitute eq ④ & ③ in eq ③

$$q_3 = \epsilon b + q_2 a + q_3 a$$

$$= \epsilon b + a a + q_3 a$$

$$\frac{q_3}{R} = \frac{b + aa}{\epsilon} + \frac{q_3}{R} \cdot \frac{a}{R}$$

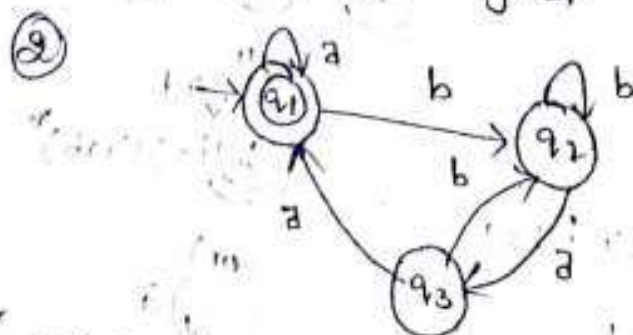
[This is in the form of

$$R = Q + RP \\ R = \frac{Q}{1 - P}]$$

$$R = QP^*$$

$$q_3 = (b + aa)a^*$$

This is regular expression.



$$1) q_1 = q_1 a + q_3 a \text{ --- (1)}$$

$$2) q_2 = q_2 b + q_1 b + q_3 b \text{ --- (2)}$$

$$3) q_3 = q_2 a \text{ --- (3)}$$

step 2: $q_1 = q_1 \epsilon + \epsilon$

step 3: $R = \epsilon + RP$

substitute eq (3) in eq (2)

$$q_2 = q_2 b + q_1 b + q_2 b$$

$$q_2 = q_2 b + q_1 b + q_2 ab$$

$$q_2 = q_1 b + q_2 b + q_2 ab$$

$$q_2 = q_1 b + q_2 (b + ab)$$

$$[\because R = \epsilon + RP]$$

$$q_2 = (q_1 b) (b + ab)^*$$

$\downarrow$

$$R = \epsilon P^*$$

Substitute (4) in (3)

$$q_3 = q_2 a$$

$$q_3 = (q_1 b) (b + ab)^* a \quad \text{--- (5)}$$

Substitute eq (5) in eq (1)

$$q_1 = q_1 a + q_3 a + \epsilon$$

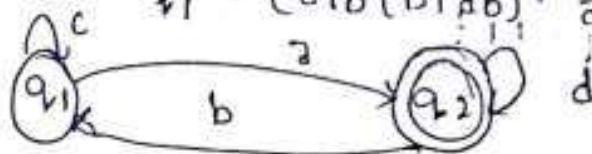
$$q_1 = q_1 a + (q_1 b) (b + ab)^* a + \epsilon$$

$$q_1 = \epsilon + q_1 (a + b(b + ab)^* a)$$

$$q_1 = \epsilon (a + b(b + ab)^* a)^*$$

$$q_1 = (a + b(b + ab)^* a)^*$$

(6)



Sol: Step 1: $q_1 = q_2 b + a_1 c \rightarrow (1)$

$$q_2 = q_1 a + q_2 d \rightarrow (2)$$

$$\text{Step 2: } q_1 = q_2 b + a_1 c + \epsilon \rightarrow (3)$$

$$\text{Step 3: } q_1 = (q_2 b + \epsilon) + a_1 c$$

$$q_1 = (q_2 b + \epsilon) c^* \Rightarrow RP = R = \emptyset P^*$$

substitute eq (3) in eq (2)

$$q_2 = q_2 d + q_1 a$$

$$q_2 = q_2 d + (q_2 b + \epsilon) c^* a$$

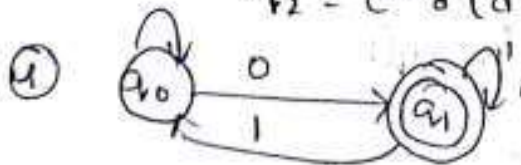
$$q_2 = q_2 d + q_2 b c^* a + \epsilon^* a$$

$$q_2 = q_2 (d + b c^* a) + \epsilon^* a$$

$$\therefore [R = RP + \epsilon]$$

$\therefore$ Now write this in $R = \emptyset P^*$ form

$$q_2 = \epsilon^* a (d + b c^* a)^*$$



$$\text{Step 1: } q_0 = q_0 1 + q_1 1 \rightarrow (1)$$

$$q_1 = q_1 1 + q_0 0 \rightarrow (2)$$

$$\text{Step 2: } q_0 = q_0 0 + q_1 1 + \epsilon$$

$$R = RP + \epsilon$$

$$q_0 = (q_1 + 1 \epsilon) \cdot 0^* = (3)$$

Substitute eq (3) in eq (2)

$$q_1 = q_1 + (q_1 + 1 \epsilon) 0^*$$

$$q_1 = q_1 + q_1 0^* + 0^*$$

$$q_1 = q_1 (1 + 0^*) + 0^*$$

$$= q_1 = (0^* 0) (1 + 0^* 0)^*$$

DFA to regular expression by state elimination method:-

Step 1:- Initial state should not have incoming edges, if exists create a new state make it as initial state.

Step 2:- Finite automata should have only one final state, if there are multiple final states then create a new state and make it as final.

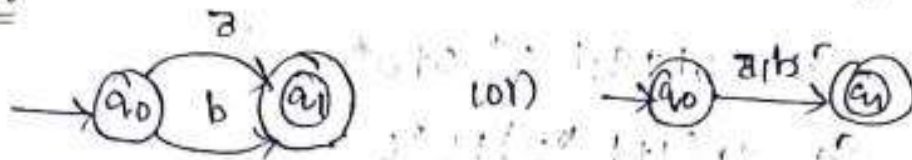
Step 3:- Final state should not have any outgoing edges if exists create a new state and make it as final state.

Step 4:- Eliminate every state except initial and final states.

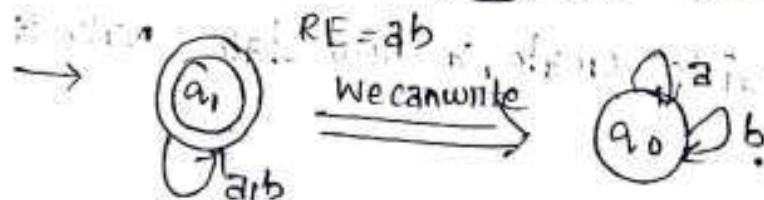
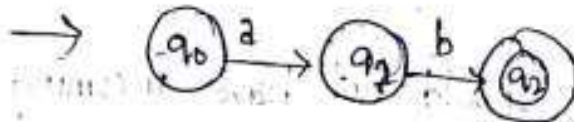
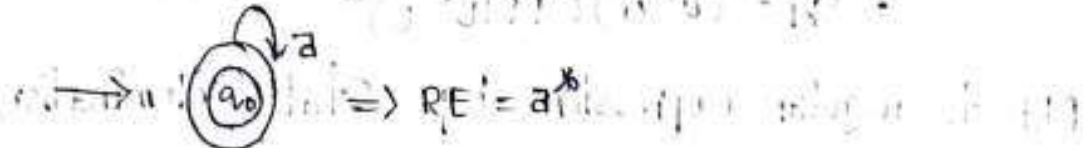
Step 5: Finally FA should have only initial state and final states.

Step 6: The label from initial state to final state becoming the regular expression.

Q:-



$\hookrightarrow RE = a|b$



$$\epsilon \cdot (c^* (b|ab)) = c^* b + c^* ab$$



Since, there is an incoming edge for q_1 , take a new state as initial state.

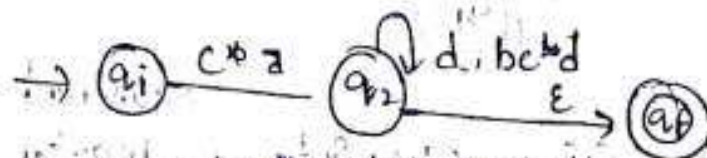
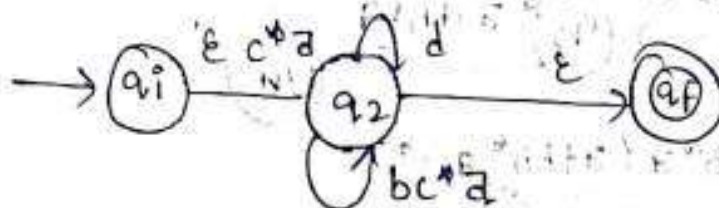


step 3:-

since there are outgoing edges from



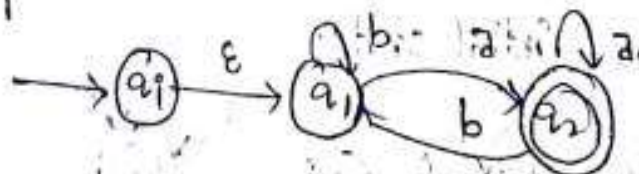
eliminate q_1 , then DFA



=> Write the regular expression:



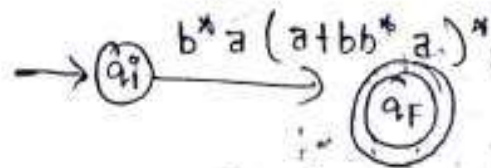
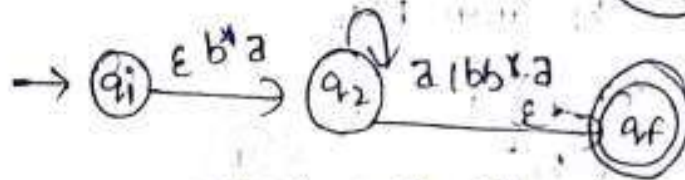
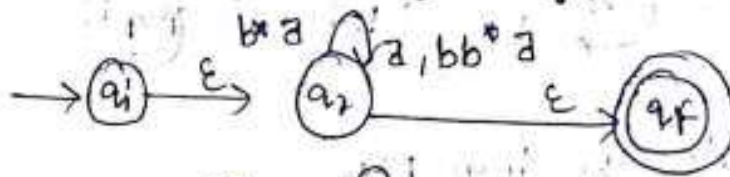
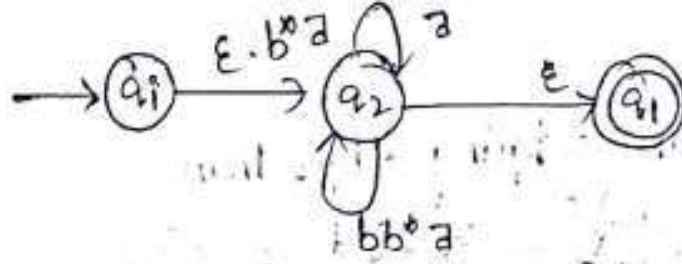
Step 1:



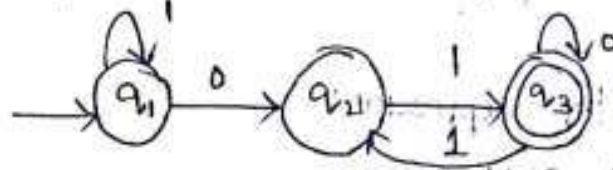
Step 2:



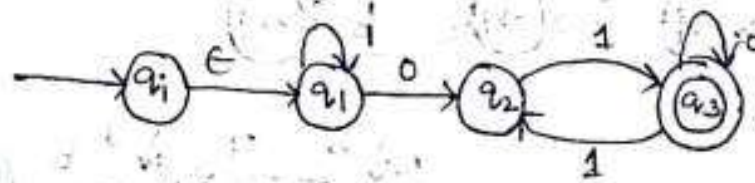
Step 3: Eliminate the q_1 state



⇒ Convert the Given DFA into a regular expression.



Step 1: Since there is an incoming edge to initial, create a new state (q_i) and make it as initial state.

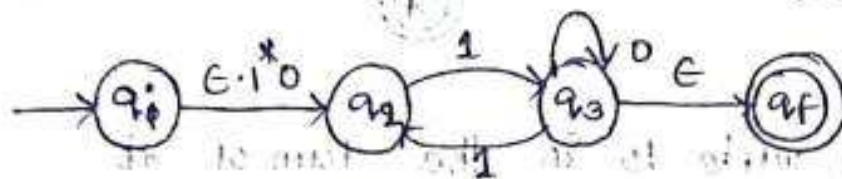


Step 2: There is only one final state.

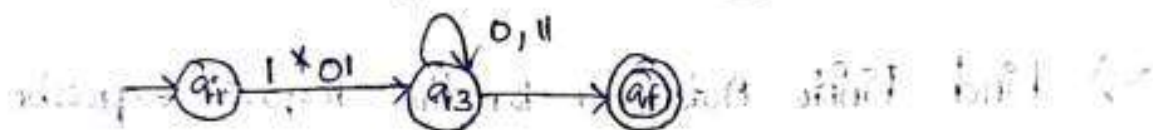
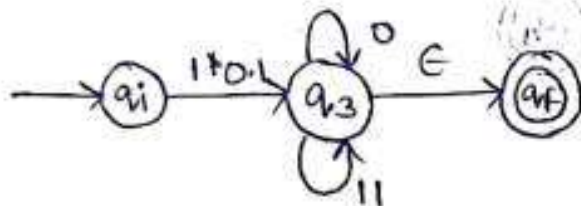
Step 3: Since there are outgoing edges to final state, create a new state and make it as final state.



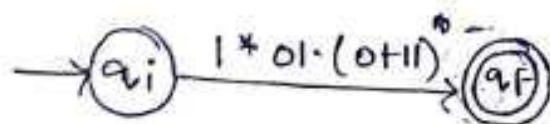
Step 4: Eliminate q_1 then the resultant DFA becomes



Eliminate q_2 then the resultant DFA becomes

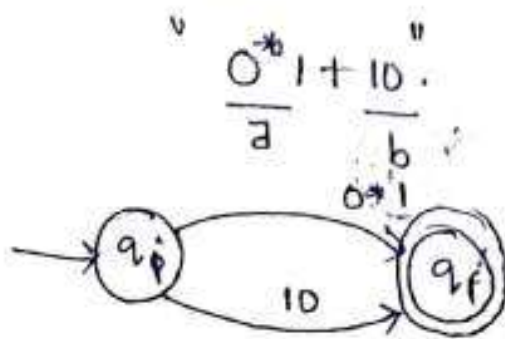


Eliminate q_3 then the resultant DFA becomes

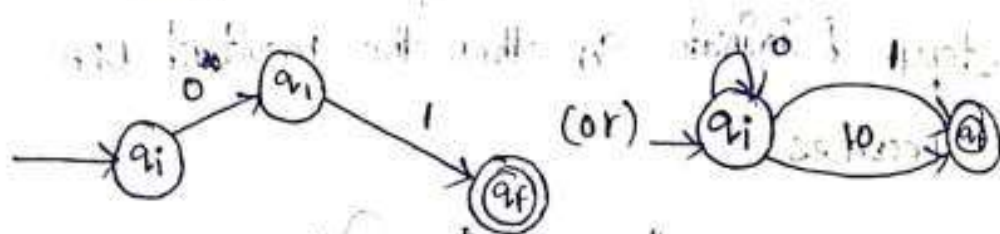


Regular Expression (RE) = $1^*01(0+1)^*$

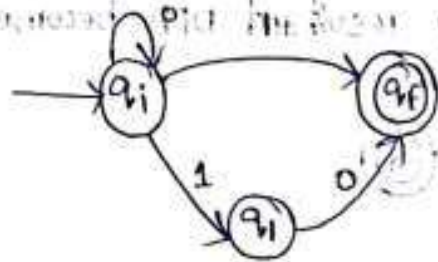
⇒ convert given regular expression into its equivalent finite Automata.



a is $\frac{0^*1}{a \quad b}$ and it is in the form of ab

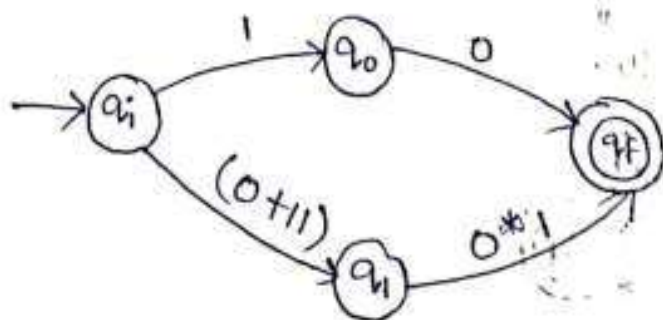
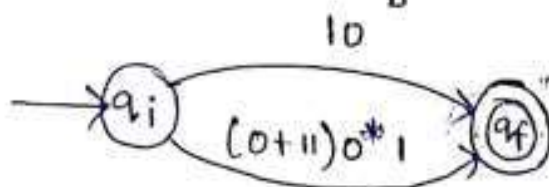


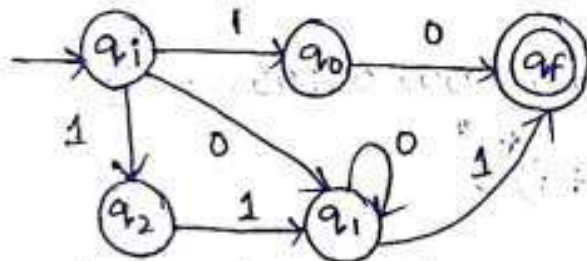
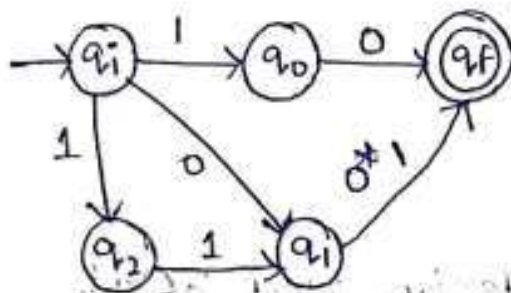
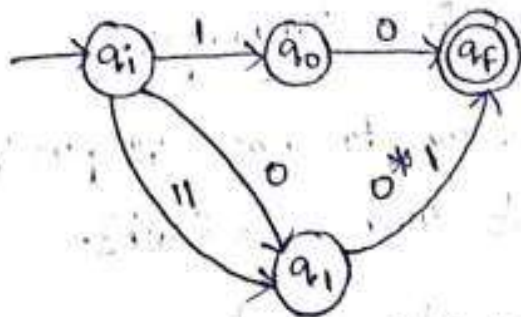
We can write 10 in the form of ab



$\Rightarrow$ Find Finite Automata for the regular expression

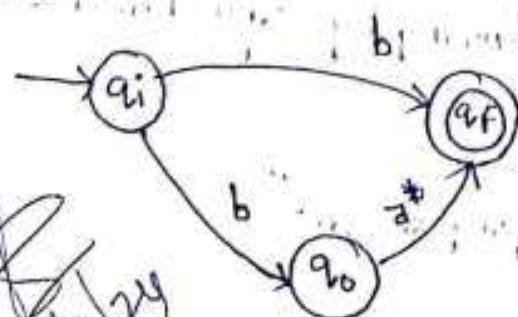
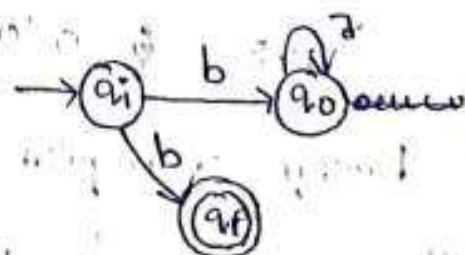
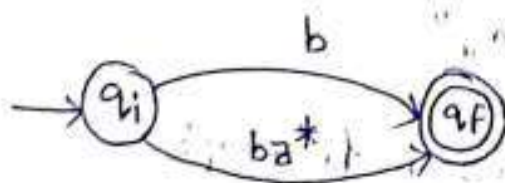
$$\frac{10 + (0+11)0^*1}{a \quad b}$$





⇒ Find finite Automato for the regular expression

$$\frac{b}{a} + \frac{ba^*}{b}$$



6/3/24

Regular Expression for the Given language:-

i) Design the regular expression, to access all possible combinations of a's and b's over $\Sigma = \{a, b\}$.

Sol:- $L = \{\epsilon, a, b, ab, aab, baa, aba, \dots\}$

" R.E = $(a|b)^*$ "

Transition diagram:



ii, Design a regular expression which accepts all strings which ends with "00".

Sol:- $L = \{00, 100, 1100, 0100, 01100, \dots\}$

" R.E = $(0+1)^*00$ "

iii, Design a regular expression for to accepts all strings which starts with "0" and ends with "1".

Sol:- $L = \{0101, 01, 011, 01011, \dots\}$

" R.E = $0(0+1)^*1$ "

iv) Design a regular expression to accepts any number of a's following by any no. of b's and any no. of c's.

Sol:- " R.E = $(a^*b^*c^*)$ "

$\Rightarrow$ v) atleast one b's, one a's, one c's

$$R.E = (aa^*bb^*cc^*)[0r](a^+b^+c^+)$$

vi) begins with "00 (or) 11" and ends with "00 (or) 11"

$$\text{sol: } R.E = (00+11)(0+1)^*(00+11)$$

vii) third character from the right end is "a"

$$\text{sol: } R.E = (a+b)^*a(a+b)(a+b)^*$$

viii) design R.E for atleast 2 b's.

$$\text{sol: } R.E = (a+b)^*b(a+b)^*b(a+b)^*$$

ix) exactly two b's.

$$R.E = a^*ba^*ba^*$$

x) Even length strings over $\Sigma = \{0\}$

$$R.E = (00)^*$$

$$L = \{ \epsilon, 00, 0000, 000000, \dots \}$$

xi) odd length strings over $\Sigma = \{1\}$

$$R.E = 1(11)^*$$

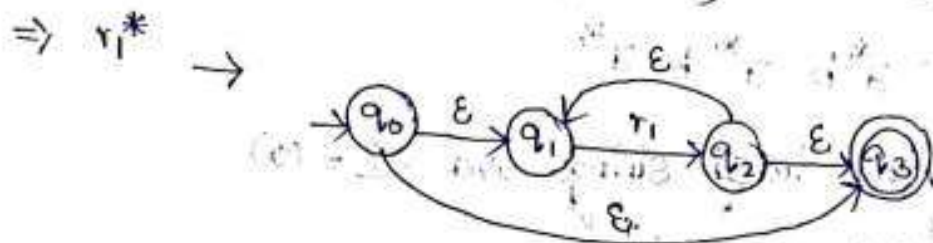
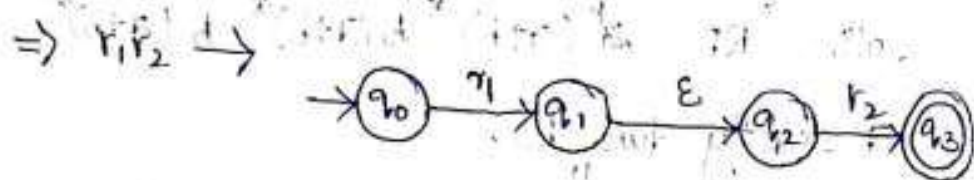
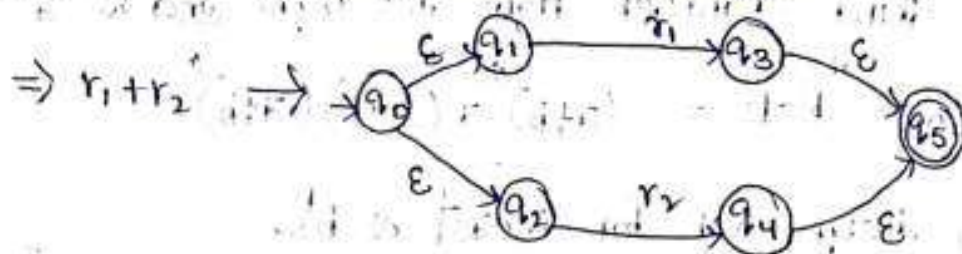
$$L = \{ \epsilon, 111, 1, 1111, 11111, \dots \}$$

Regular Expression to Finite Automata Using Subset

method:

Step 1: Convert the given expression into NFA with ϵ

Step 2: Convert NFA with ϵ into DFA.

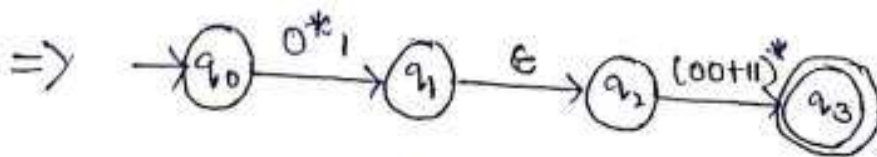


Q:- convert the given expression into the finite Automata using "subset" method.

$$\frac{0^*1}{r_1} (00+11)^*_{r_2}$$

A:- This is in the form of $r_1 r_2$

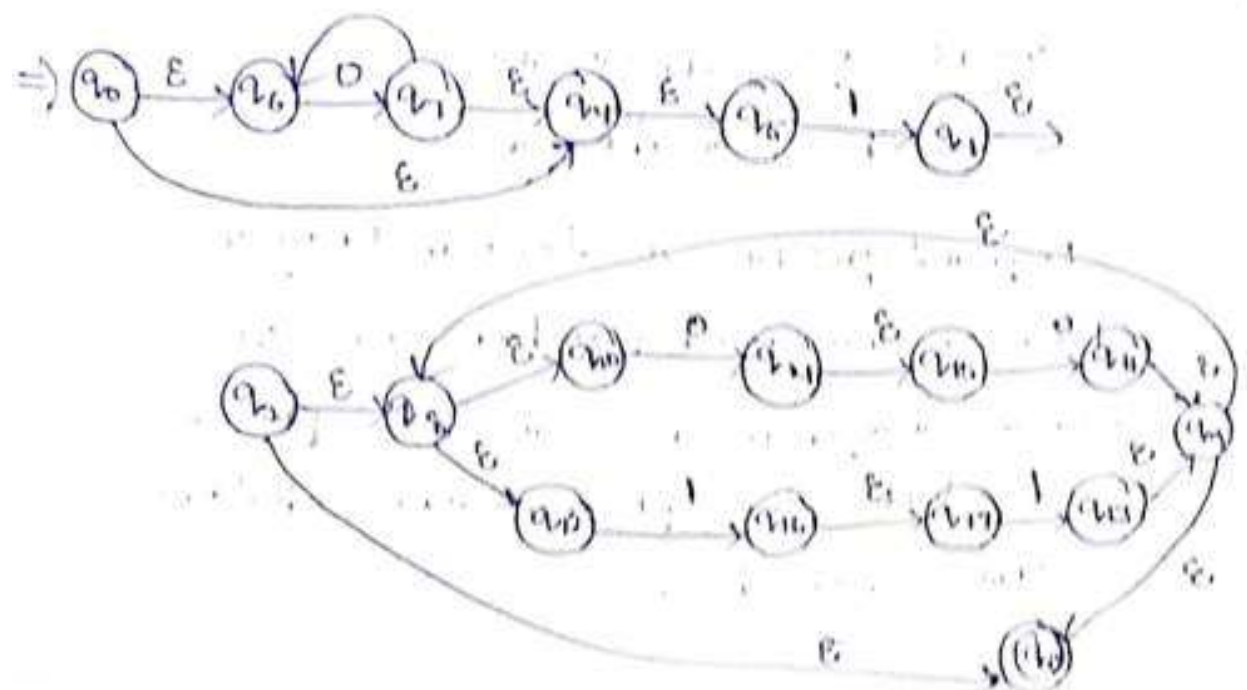
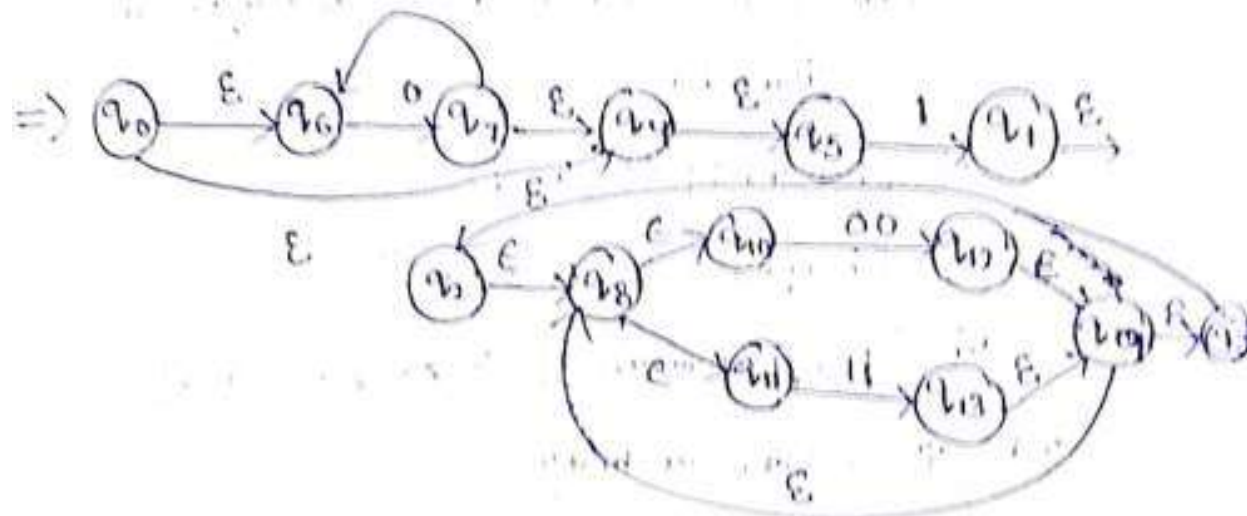
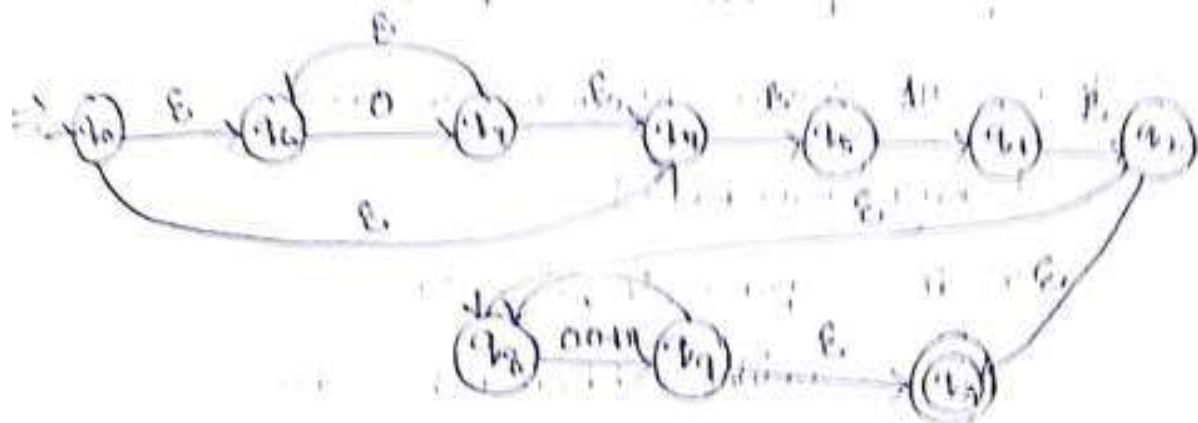
where $r_1 = 0^*1$, $r_2 = (00+11)^*$



$r_1 = 0^*1$ Again this is in the form of $r_1 r_2 \Rightarrow r_1 = 0^*$, $r_2 = 1$



$q_1 = q_1^*$ then q_0, q_1 the form of q^*



Applications of Regular Expressions:-

=> There are three applications of R.E

1) Regular Expressions in ^{unix} ~~Unix~~:-

→ There are various notations in unix

i) Any character $[.]$

ii) The sequence $[1, 2, \dots, 10]$

=> addition = $1 + 2 + 3 + 4 + \dots + 10$

iii) $[A-z], [a-z], [0-9] \rightarrow$ A range of character

iv) $[:\text{digit}:] \rightarrow [0-9]$

v) $[:\text{alpha}:] \rightarrow [A-z, a-z]$

vi) $[:\text{alphanumeric}:] \rightarrow [A-z, a-z, 0-9]$

vii) $*$ → zero or many

viii) $+$ → one or many

ix) $?$ → zero or one

x) $|$ → concatenation

2) Regular Expressions in Lexical Analyser:-

→ One of the most important phase of lexical analyser is to scan the program and divide the program into individual ^{unix} ~~unix~~ called tokens.

→ semantic analyser provides some tasks to the tokens.
→ First phase of the compiler it provides the program into tokens.

→ In tokens, there are identifier and keyword.

→ In unix, there are lex and flex they decide ^{what} ~~how~~ the lexical analyser do in unix operating system.

g) Regular Expressions in finding patterns in the string.

→ To find the patterns in the string we use RE.

123A Main st.

$[0-9]^+[A-Z]?[A-Z][a-z]^*([A-Z](a-z)^*)^*$

$(street | st | Road | Rd)$

Algebraic ^{laws} of Regular Expression:

Associative laws for regular Expression RegEx

$$A + (B + C) = (A + B) + C \quad \text{and} \quad A(B \cdot C) = (A \cdot B) \cdot C$$

Commutative laws for Regular Expression RegEx

$$A + B = B + A \quad \text{however} \quad A \cdot B \neq B \cdot A$$

Identity for Regular Expression: RegEx

$$\phi + A = A + \phi = A$$

$$\epsilon \cdot A = A \cdot \epsilon = A$$

Distributivity for Regular Expression

$$A(B+C) = AB+AC \quad [\text{left}]$$

$$(B+C)A = BA+CA \quad [\text{right}]$$

$\rightarrow \phi$ is an annihilator

Idempotent $A+A=A$

closure laws for Regular Expression: RegEx

$$(A^*)^* = A^*, \phi^* = \epsilon, \epsilon^* = \epsilon, A1 = AA^+, \text{ and } A^+ = A^*A$$

$$A^* = A + \epsilon$$

DeMorgan's Law:

$$(L+B)^* = (L^*B^*)^*$$

Test for the Regular Expression whether $E=F$

1) convert the E and F into concrete R.E C & D

2) Test $L(C) = L(D)$ th. if language of C is equals to the language of D then we can say $E=F$ is TRUE otherwise FALSE.

pumping lemma:- $\rightarrow$ To prove the language is regular expression or not.

$\rightarrow$ repetition [Pumping] lemma[word] $\rightarrow$ It is entry point, using that we can derive other words.

$\rightarrow$ Repetition of that word, by that word can derive the other words.

$\rightarrow$ There are two types of pumping lemma:

1) Regular Expression for pumping lemma

2) context free languages

Theorem:-

let " L " be the Regular language, Then there exists a constant " c " such that for each and every string " w " in " L ".

$$|w| \geq c$$

We can break the w in three strings; " x, y, z "

- $|y| > 0$
- $|xy| \leq c$
- For all $k \geq 0$, the string xy^kz is also in L .

procedure to prove a language is not regular using pumping lemmas

1) consider L is a regular language

2) select constant 'c' such that $|w| \geq c$, for every string (w) in L .

3) Take a string (w) from L and divide w into x, y and z .

such that

$$|y| > 0$$

$$|xy| \leq c$$

for $k \geq 0$, xy^kz is in L .

4) If not, say L is a irregular.

problem:-

prove $L = \{a^m b^n \mid \gcd(m, n) = 1\}$ is not a regular

language.

Ans: $L = \{aabb, aaabbbb, aaaabbbbb, \dots\}$

1. Let L is a regular language

2. Constant $c = 3$, where $|w| \geq c$.

3. Let $w = \underline{aa} \underline{ab} \underline{bbb}$
 $x \quad y \quad z$

divide w into x, y, z such that $|y| > 0$ & $|xy| \leq c$

then $x = aa$, $y = ab$, $z = bbb$

for every $k \geq 0$, compute xy^kz

let $k=0$,

$xy^0z \Rightarrow x \cdot y^0 \cdot z = xz = aabbb$, this is in d

let $k=1$,

$\Rightarrow xy^1z \Rightarrow x \cdot y^1 \cdot z = xyz = aaabbbb$ is in d

let $k=2$,

$\Rightarrow xy^2z \Rightarrow x \cdot y^2 \cdot z = xyyz = aaababbbb$ is not in d

$\therefore$ The given language d is irregular: not reg.

C: prove $d = \{0^n \mid n \text{ is a perfect square}\}$ is not a regular language.

Ans: $d = \{0, 0000, 0000000000, 00000000000000000000\}$

1) let d is a regular language.

2) constant $c \in \mathbb{N}$, where $|w| \geq c$...

3) select $w = 0000$,

divide w in xyz such that

$|y| > 0$ & $|xy| \leq c$ then

$x = \epsilon, y = 0, z = 000$

for every $k \geq 0$, compute xy^kz

let $k=0$,

$\Rightarrow xy^0z \Rightarrow xz = \epsilon \cdot 000 \Rightarrow 000$ is not in d

let $k=1$,
 $\Rightarrow xy^kz \Rightarrow xyz \Rightarrow \epsilon \cdot 0 \cdot 000 \Rightarrow 0000$ is in L .

let $k=2$,
 $\Rightarrow xy^kz = xyyz \Rightarrow \epsilon \cdot 0 \cdot 0 \cdot 0000 \Rightarrow 000000$ is not in L .

$\therefore$ The given language L is not regular.

Context-free Grammars:- (CFG) set of rules.

$\rightarrow$ Grammar is a ^{tuple} Quaternary Notation's

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

$V \rightarrow$ set of non-terminals (or) variables, (uppercase letters)

$T \rightarrow$ set of terminals (lowercase letters)

$P \rightarrow$ set of productions ($\alpha \rightarrow \beta$) - these are constructed
to terminals & Non-Terminals ($\alpha = aA, \beta = bBa$)

$$\Rightarrow \alpha, \beta \in (V \cup T)^*$$

$S \rightarrow$ start symbol

Types of Grammars:- 4 types of Grammars

1) Type 0:- It is an unrestricted Grammar.

$\rightarrow$ The language generated by this kind of Grammar
is "Recursively Enumerable Language (REL)".

$\rightarrow$ To represent this we use "Turing Machine (TM)".

Produ
 $\Rightarrow \alpha \rightarrow \beta$

where α has atleast one non-terminal

Context free Grammar (CFG):-

A context free grammar (G) is a 4 tuple notation where $G = (V, \Sigma, P, S)$

where $V \rightarrow$ set of non-terminals or variables

$\Sigma \rightarrow$ set of terminals

$\rightarrow$ represented in uppercase letters

$s \rightarrow$ represented in lowercase letters

$S \rightarrow$ start symbol

$P \rightarrow$ set of productions of the form $(\alpha \rightarrow \beta)$

where $\alpha \in V$, $|\alpha| = 1$ & (β can be a terminal or terminal followed by non-terminal or non-terminal followed by a terminal)

Derivations:-

$\rightarrow$ productions of the grammar are used to infer the strings of the language

$\rightarrow$ There are two approaches

1) Recursive Inference

2) Derivations

1) Recursive Inference Approach:-

- This is a bottom up approach (body to head).
- We start with known symbols (terminals).

Grammar is:

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow b$
6. $I \rightarrow a$
7. $I \rightarrow I_0$
8. $I \rightarrow I_1$
9. $I \rightarrow I_a$
10. $I \rightarrow I_b$

→ The string to be derived is $a^*(a+b00)$

$a^*(a+b00)$ $E^*(E+I00)$ $E * E$
 $I^*(a+b00)$ $E^*(E+I_0)$ E
 $E^*(a+b00)$ $E^*(E+I)$ E
 $E^*(I^*b00)$ $E^*(E+E)$
 $E^*(E+00)$ $E^*(E)$

S.No	String Inferred	for language of	Productions used	Steps used
1)	a	I	6	-
2)	a	E	1	1)
3)	b	I	5	-
4)	00	I	7	3)
5)	000	I	7	4)

6) boo E 1, 5)

7) a+boo E 2, 6)

8) (a+boo) E 4, 7)

9) a*(a+boo) E 3, 8)*

2) Derivation Approach:-

→ This is a topdown approach.

→ let "G" be a "CFG",

$$\alpha A \beta \in (V \cup T)^*$$

where, A is a non-terminal and $A \rightarrow \gamma$ then

$$\alpha A \beta \xRightarrow[G]{*} \alpha \gamma \beta$$

⇒ $a * (a + boo)$

Grammar is:

1. $E \rightarrow I$

6. $I \rightarrow a$

2. $E \rightarrow E + E$

7. $I \rightarrow I_0$

3. $E \rightarrow E * E$

8. $I \rightarrow I_1$

4. $E \rightarrow (E)$

9. $I \rightarrow I_2$

5. $I \rightarrow b$

10. $I \rightarrow I_b$

$$E \rightarrow E * E \rightarrow I * E \rightarrow a * E \rightarrow a * (E) \rightarrow a * (E + E)$$

$$\rightarrow a * (I + E) \rightarrow a * (a + E) \rightarrow a * (a + I) \rightarrow a * (a + I_0)$$

$$\rightarrow a * (a + I_0) \rightarrow a * (a + I_{00}) \rightarrow a * (a + boo)$$

Left Most Derivation And Right Most Derivation:

1) Left Most Derivation:

In which At each step of derivation

If we replace left most variable with its production body.

→ We represent leftmost derivation by $\xRightarrow{*}_{lm}$

2) Rightmost Derivation:

In which, at each step of derivation, we replace rightmost variable with its production body.

→ we represent leftmost derivation by $\xRightarrow{*}_{rm}$

Examples:- $\rightarrow E$

$\rightarrow E * (I + boo)$

$\rightarrow E * E$

$\rightarrow E * (a + boo)$

$\rightarrow E * (E)$

$\rightarrow I * (a + boo)$

$\rightarrow E * (E + E)$

$\rightarrow a * (a + boo)$

$\rightarrow E * (E + I)$

$\rightarrow E * (E + I + I)$

$\rightarrow E * (E + I + I)$

$\rightarrow E * (E + boo)$

→ Derive: abababa from the Given Grammar and leftmost derivation approach.

$$S \rightarrow sbs \mid a$$

case i) $\rightarrow S$ where $S \rightarrow sbs$ and $S \rightarrow a$

$$\rightarrow sbs \quad \{S \rightarrow sbs\}$$

$$\rightarrow sbsbs \quad \{S \rightarrow sbs\}$$

$$\rightarrow sbsbsbs \quad \{S \rightarrow sbs\}$$

$$\rightarrow absbsbs \quad \{S \rightarrow a\}$$

$$\rightarrow ababsbs \quad \{S \rightarrow a\}$$

$$\rightarrow abababs \quad \{S \rightarrow a\}$$

$$\rightarrow abababa \quad \{S \rightarrow a\}$$

case ii) $\rightarrow S$

$$\rightarrow sbs \quad \{S \rightarrow sbs\}$$

$$\rightarrow abs \quad \{S \rightarrow a\}$$

$$\rightarrow absbs \quad \{S \rightarrow sbs\}$$

$$\rightarrow absbsbs \quad \{S \rightarrow sbs\}$$

$$\rightarrow ababsbs \quad \{S \rightarrow a\}$$

$$\rightarrow abababs \quad \{S \rightarrow a\}$$

$$\rightarrow abababa \quad \{S \rightarrow a\}$$

Language of Context free Grammars

Let " G " be a context free Grammar, then its language is denoted by $L(G)$ and it is defined as " $L(G) = \{w \text{ in } T^* / s \xRightarrow[G]{*} w\}$ ".

→ The language of CFG is known as "context free language".

Sentential Form:-

Let " G " be a "CFG", $\alpha \in (V \cup T)^*$, s is a start symbol. If $s \xRightarrow[G]{*} \alpha$ then α is in "sentential form".

→ There are two types:

1) left^{most} sentential form:

If $s \xRightarrow[lm]{*} \alpha$ then your α is leftmost sentential form.

2) Rightmost sentential form:

If $s \xRightarrow[rm]{*} \alpha$ then your α is rightmost sentential form.

Parse tree / derivation tree / syntax tree:-

A parse tree is a tree which is constructed with the following conditions:

1) root node must be a start symbol.

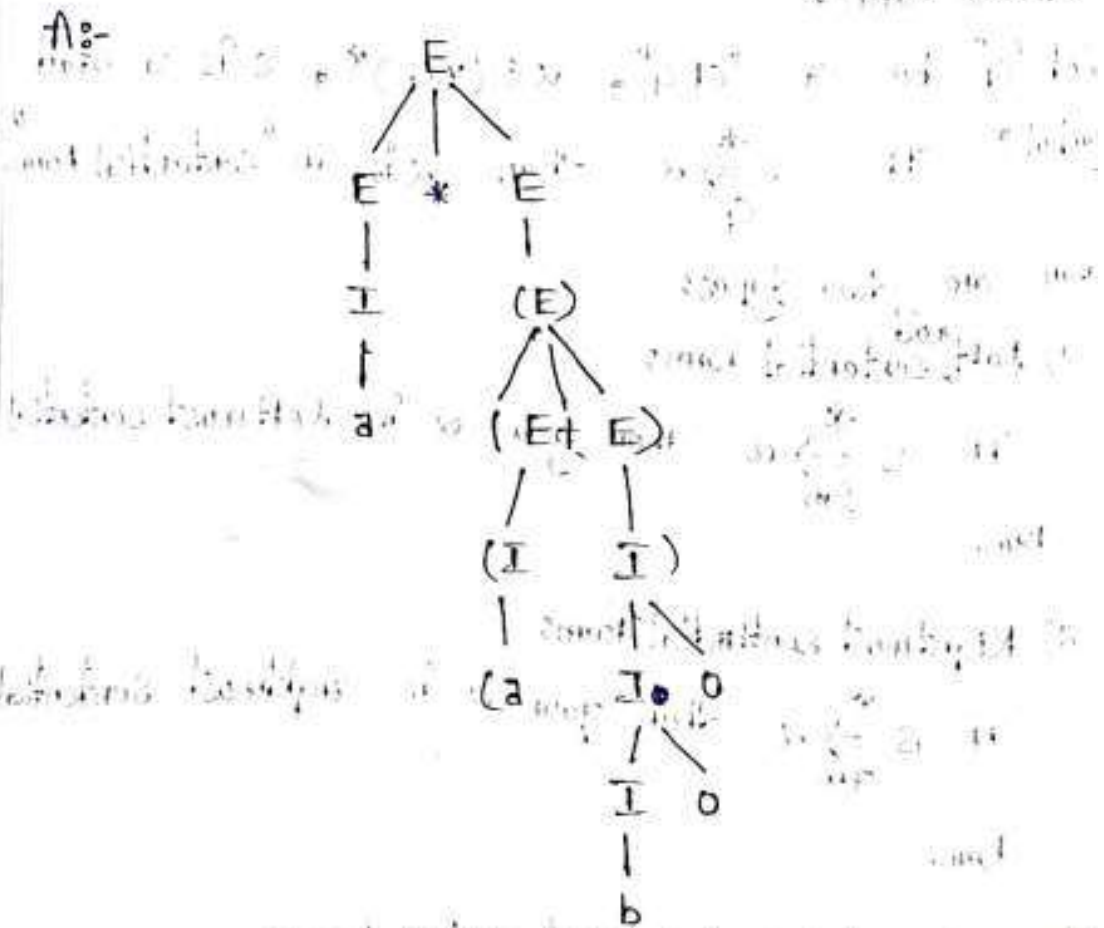
2) Intermediate / Internal node, must be a variable (including starting symbol).

3) leaf node must be a terminal (or) ϵ

→ construct parse tree for the string

$a*(a+b)$ using the CFG $E \rightarrow E+E/E*$

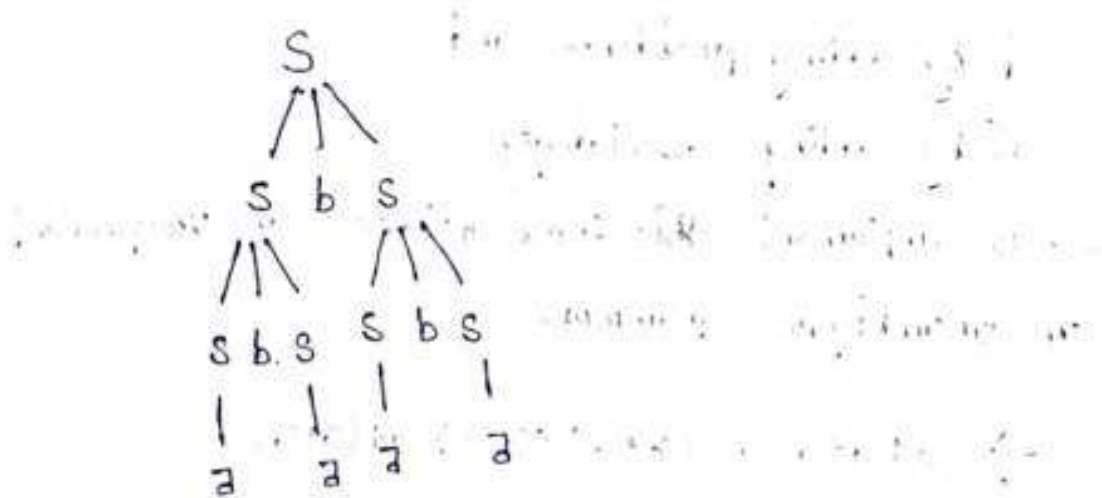
$(E)/I, I \rightarrow I_1/I_2/I_3/I_4/b/a$



→ construct parse tree for the string

abababa using the Grammar

$S \rightarrow sb/a$



Ambiguous Grammar:

A CFG "G" is said to be ambiguous for atleast one string in its language if there exist two distinct parse trees.

Example: Grammar is $E \rightarrow E + E, E \rightarrow E * E, E \rightarrow a$
construct "a + a * a" parse tree.



$\therefore$ The Given Grammar is ambiguous Grammar.

~~Ex~~ Removing Ambiguity:

Removing ambiguity is done

i) by adding precedence and

ii) by adding associativity.

→ To implement ~~this~~ these, take an unequivalent
an unambiguous grammar.

$$\Rightarrow a + a + a \Rightarrow (a + a) + a \Rightarrow a + (a + a)$$

$$\Rightarrow E \rightarrow E + E$$

$$\rightarrow E \rightarrow E * E \rightarrow F \rightarrow a$$

$$\rightarrow E \rightarrow E + T$$

$$\rightarrow T \rightarrow T * F$$

$$\rightarrow E \rightarrow T$$

$$\rightarrow F \rightarrow T \rightarrow F$$

$a + a + a$



Leftmost derivations in ambiguity:

→ There are more derivations in leftmost and
Rightmost ambiguity.

→ There are only one derivation in the
an ambiguity grammar.

Inherent Ambiguous:-

A language is said to be ambiguity if all its
Grammer is ambiguous.

push down Automata (PDA):-

→ If we add stack to the finite Automata then it will become Pushdown Automata.

→ If we want to store the infinite information, we cannot store in finite automata then we use the push down Automata.

→ PDA = FA + stack.

→ This is used to represent the "context free Grammar".

→ In PDA, we use three components

1) input tape / input string

2) finite Control (Push, Pop)

3) stack

Formal Definition of PDA:-

→ It is represented by "P", it is a 7 tuple

notation

$$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

Where, $P = \text{PDA}$

$Q = \text{set of states}$

$\Sigma \rightarrow$ set of input symbols

$T \rightarrow$ set of stack elements

$\delta \rightarrow$ It is a transition Function

$$\delta(q_0, a, x) = (q_n, \alpha)$$

$q \rightarrow$ current state

$a \rightarrow$ input to be processed

$x \rightarrow$ top of the stack

$q_n \rightarrow$ new state

$\alpha \rightarrow$ Symbol to be replaced in place of x

$q_0 \rightarrow$ Initial state

$z_0 \rightarrow$ top of stack

$F \rightarrow$ set of accepting states/Final states

Q:- Construct PDA for the given language

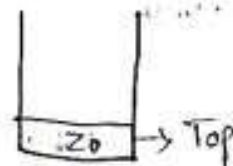
$$L = \{a^n b^n / n \geq 1\}$$

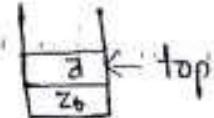
Sol:- $L = \{a^n b^n / n \geq 1\}$

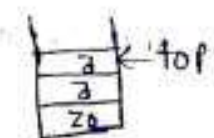
$$L = \{ab, aabb, aaabbb, \dots\}$$

Let us assume "aaabbb" is our input tape/string,
 q_0 is the initial state.

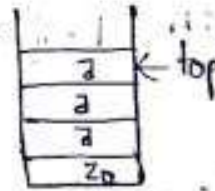
Let z_0 be at the top of the stack.



$$\delta(q_0, a, z_0) = (q_0, az_0)$$


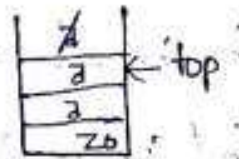
$$\delta(q_0, a, a) = (q_0, aa)$$


$$\delta(q_0, a, a) = (q_0, aaa)$$

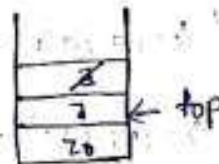


$$\delta(q_0, b, a) = (q_1, \epsilon)$$

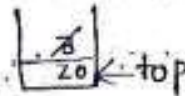
$$\delta(q_1, b, a) = (q_1, \epsilon)$$



$$\delta(q_1, b, a) = (q_1, \epsilon)$$



$$\delta(q_1, \epsilon, z_0) = (q_2, \epsilon)$$



Stack empty.

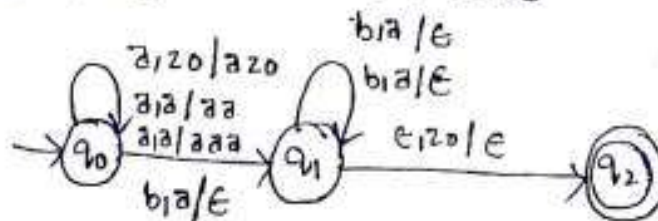
$$PDA = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

$$Q \rightarrow \{q_0, q_1, q_2\} \quad \Gamma \rightarrow \{a, z_0\} \quad F \rightarrow \{q_2\}$$

$$\Sigma \rightarrow \{a, b\} \quad \delta$$

$$q_0 \rightarrow q_0$$

$$z_0 \rightarrow \{z_0\}$$



Instantaneous Description of PDA:

→ To check the acceptance of string

Find the acceptance of a string $aabb$ by the PDA $P = \{q_0, q_1, q_2\}$

$$P = [\{q_0, q_1, q_2\}, \{a, b\}, \{z_0\}, \delta, q_0, z_0, \{q_2\}]$$

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

$$\delta(q_0, a, az_0) = (q_0, aaz_0)$$

$$\delta(q_0, a, aaz_0) = (q_0, aaaz_0)$$

$$\delta(q_0, b, aaaz_0) = (q_1, \epsilon)$$

$$\delta(q_0, b, aaz_0) = (q_1, \epsilon)$$

$$\delta(q_1, b, az_0) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) = (q_2, z_0)$$

$$\delta(q_0, aabb, z_0) \vdash (q_0, abb, az_0)$$

$$\vdash (q_0, bb, aa)$$

$$\vdash (q_1, b, a)$$

$$\vdash (q_1, \epsilon, z_0)$$

$$\vdash (q_2, z_0)$$

∴ the given string $aabb$ is accepted

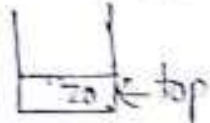
by $\vdash (q_2, z_0)$ by the PDA

Construct PDA to accept the language $L = \{a^n b^{2n} \mid n \geq 1\}$
and find acceptance of string "aaabbbb".

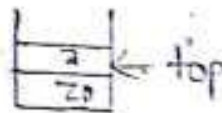
A:- push for "a" and pop for every two "b's".

$L = \{abb, aabbbb, aaabbbbb, \dots\}$

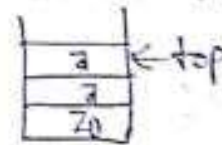
Let $aaabbbb \in$ is the input tape, q_0 is the initial state and z_0 is at the top of stack



$$\delta(q_0, a, z_0) = (q_0, az_0)$$



$$\delta(q_0, a, a) = (q_0, aaa)$$



$$\delta(q_0, b, a) = (q_1, az_0)$$

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, b, a) = (q_1, az_0)$$

$$\delta(q_1, b, a) = (q_2, \epsilon)$$

$$\delta(q_2, \epsilon, z_0) = (q_3, z_0)$$

$$P = \left[\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \{q_3\} \right]$$



$$\delta(q_0, aaabbbbb, z_0) \vdash (q_0, aabbbbb, az_0)$$

$$\vdash \delta(q_0, abbbbb, aa) \vdash (q_0, bbbbb, aaa)$$

$$\vdash (q_1, bbbbb, aaa) \vdash (q_2, bbbb, aa) \vdash (q_1, bbba)$$

$$\vdash (q_2, bb, a) \vdash (q_1, b, a) \vdash (q_2, \epsilon, z_0) \vdash (q_3, z_0)$$

$\therefore$ The given string aaabbbbb is accepted

$\vdash (q_3, z_0)$ by the given PDA.

→ Construct PDA for the given language $L = \{a^n b^m c^n \mid m, n > 0\}$

A:- push for 'a's', Pop for 'c's' and 'b's' for no change. $m=2, n=3, m=1, n=2$

$$L = \{abc, abbc, aaabbbccc, aabcc, \dots\}$$

Languages of PDA:-

1) Acceptance by Final state

$$L = \{w \mid \delta(q_0, w, z_0) \xRightarrow[p]{*} (q_f, \epsilon)\}$$

continuation of $d = \{a^m b^n c^n \mid m, n \geq 0\}$

let $aabcc \in$ as input tape, q_0 is the ^{initial} ~~final~~ state and z_0 at the top of the stack.

$$\rightarrow \delta(q_0, a, z_0) = (q_0, az_0)$$

$$\rightarrow \delta(q_0, a, a) = (q_0, aa)$$

$$\rightarrow \delta(q_0, b, a) = (q_1, aa)$$

$$\rightarrow \delta(q_1, c, a) = (q_2, a)$$

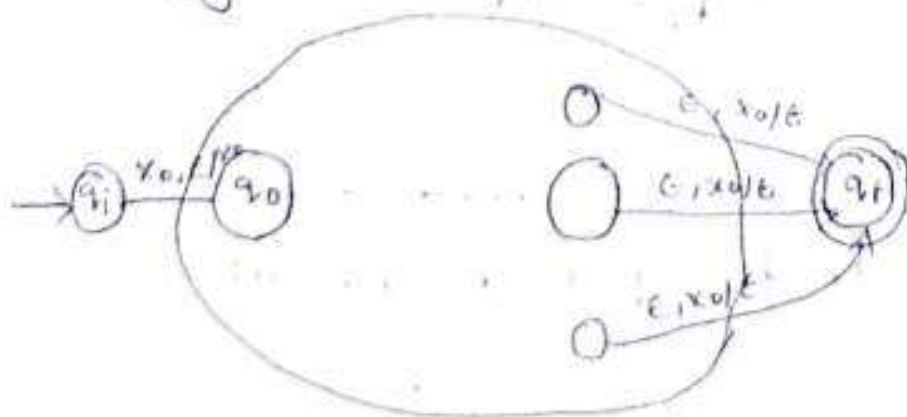
$$\rightarrow \delta(q_2, c, a) = (q_2, z_0)$$

$$\rightarrow \delta(q_2, \epsilon, z_0) = (q_3, z_0)$$

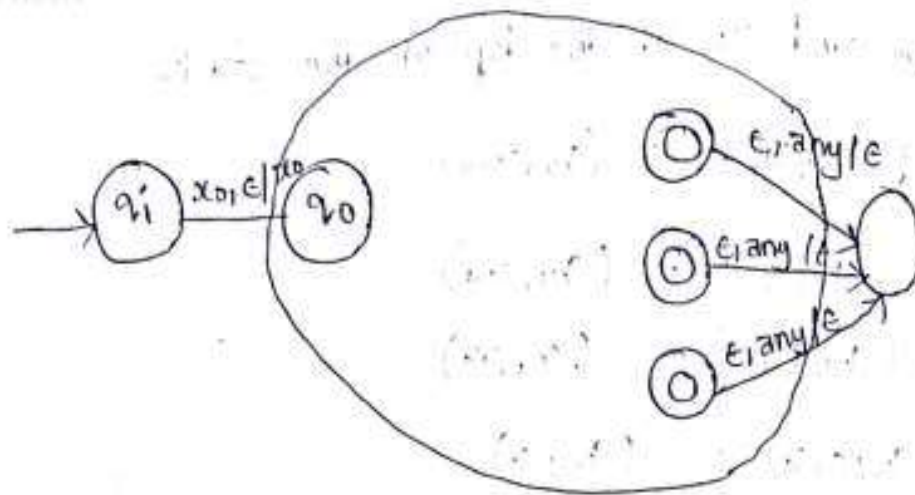
3) Acceptance by Empty stack:

$$N(P) = \{w \mid \delta(q_0, w, z_0) \xrightarrow{*}_P (q, \epsilon, \epsilon)\}$$

2) From empty stack to final state:



4) From final state to empty stack



→ Equivalence of PDA and CFG's

Conversion of CFG to PDA:-

A For non terminal: $A \rightarrow \alpha$

$$\delta(q_1, \epsilon, A) = (q_1, \alpha)$$

For terminal:

$$\delta(q_1, a, a) = (q_1, \epsilon)$$

Example:-

convert the given CFG into PDA

$$S \rightarrow AB$$

$$A \rightarrow 0s/0$$

$$B \rightarrow 1s/1$$

$$\begin{array}{lll} S \rightarrow AB & B \rightarrow 1s & A \rightarrow 0 \\ A \rightarrow 0s & B \rightarrow 1 & \end{array}$$

consider $S \rightarrow AB$

$$\delta(q, \epsilon, S) = (q, AB)$$

$A \rightarrow 0S$

$$\delta(q, \epsilon, A) = (q, 0S)$$

$A \rightarrow 0$

$$\delta(q, \epsilon, A) = (q, 0)$$

$B \rightarrow 1S$

$$\delta(q, \epsilon, B) = (q, 1S)$$

$B \rightarrow 1$

$$\delta(q, \epsilon, B) = (q, 1)$$

$$\Rightarrow \delta(q, 0, 0) = (q, \epsilon)$$

$$\delta(q, 1, 1) = (q, \epsilon)$$

$$\delta(q, \epsilon, S) = \{(q, AB)\}$$

$$\delta(q, \epsilon, 0) = \{(q, 0S), (q, 0)\}$$

$$\delta(q, \epsilon, 1) = \{(q, 1S), (q, 1)\}$$

$$\delta(0, 0, 0) = \{(q, \epsilon)\}$$

$$\delta(\overset{1,1,1}{1,1,1}) = \{(q, \epsilon)\}$$

Construct PDA

convert the given CFG to PDA:-

$$E \rightarrow E + E / E * E / I / (E)$$

$$I \rightarrow I_0 / I_1 / I_a / I_b / a / b$$

$$\rightarrow E \rightarrow E + E$$

For Non-Terminal $A \rightarrow \alpha$

$$\delta(q, \epsilon, A) = (q, \alpha)$$

$$\rightarrow E \rightarrow E + E$$

$$\delta(q, \epsilon, E) = (q, E + E)$$

$$\rightarrow E \rightarrow E * E$$

$$\delta(q, \epsilon, E) = (q, E * E)$$

$$\rightarrow E \rightarrow I$$

$$\delta(q, \epsilon, E) = (q, I)$$

$$\rightarrow E \rightarrow (E)$$

$$\delta(q, \epsilon, E) = (q, (E))$$

$$\rightarrow I \rightarrow I_0$$

$$\delta(q, \epsilon, I) = (q, I_0)$$

$$\rightarrow I \rightarrow I_1$$

$$\delta(q, \epsilon, I) = (q, I_1)$$

$$\rightarrow I \rightarrow I_a$$

$$\delta(q, \epsilon, I) = (q, I_a)$$

$$\rightarrow I \rightarrow I_b$$

$$\delta(q, \epsilon, I) = (q, Ib)$$

$$\rightarrow I \rightarrow a$$

$$\delta(q, \epsilon, I) = (q, a)$$

$$\rightarrow I \rightarrow b$$

$$\delta(q, \epsilon, I) = (q, b)$$

For Terminal

$$T = \{a, b, +, *, 0, 1\}$$

$$\delta(q, a, a) = (q, \epsilon)$$

$$\rightarrow \delta(q, a, a) = (q, \epsilon)$$

$$\rightarrow \delta(q, b, b) = (q, \epsilon)$$

$$\rightarrow \delta(q, 0, 0) = (q, \epsilon)$$

$$\rightarrow \delta(q, 1, 1) = (q, \epsilon)$$

$$\rightarrow \delta(q, +, +) = (q, \epsilon)$$

$$\rightarrow \delta(q, *, *) = (q, \epsilon)$$

$$\Rightarrow \delta(q, \epsilon, \epsilon) = \{(q, \epsilon \epsilon), (q, \epsilon * \epsilon), (q, \epsilon I), (q, \epsilon 0)\}$$

$$\Rightarrow \delta(q, \epsilon, I) = \{(q, Ib), (q, Ia), (q, Ib), (q, Ia), (q, Ib)\}$$

$$\therefore PDA = [\{q\}, \{a, b, 0, 1, +, *\}, \Gamma, \delta, q, z_0, \{q\}]$$

Conversion of PDA to CFG:- based on Transition functions

convert the given PDA to CFG if there are no stack symbols - 1

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

$$\delta(q_0, \epsilon, a) = (q_1, \epsilon)$$

$$1 \rightarrow 2$$

$$2 \rightarrow 4$$

$$PDA = [\{q_0, q_1\}, \{a, b\}, \{z_0\}, \delta, q_0, z_0, \{q_1\}]$$

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

$$\therefore Q \Gamma Q$$

*:- Let us assume, s be the start symbol

Find variables

$$V = \{S, [q_0 a q_0], [q_0 a q_1], [q_0 z_0 q_0], [q_0 z_0 q_1], [q_1 a q_0], [q_1 a q_1], [q_1 z_0 q_0], [q_1 z_0 q_1]\}$$

$$T = \{a, b\}$$

Find productions:

$$\rightarrow \delta(q_0, a, z_0) = (q_0, az_0)$$

$$[q_0 a q_0] \rightarrow a [q_0 a q_0] [q_0 z_0 q_0]$$

$$[q_0 z_0 q_0] \rightarrow a [q_0 a q_1] [q_1 z_0 q_1]$$

$$[q_0 z_0 q_1] \rightarrow a [q_0 a q_0] [q_0 z_0 q_1]$$

$$[q_0 z_0 q_1] \rightarrow a [q_0 a q_1] [q_1 z_0 q_1]$$

$$\rightarrow \delta(q_0, a, a) = (q_0, a)$$

$$[q_0 a q_0] \rightarrow a [q_0 a q_0] [q_0 a q_0]$$

$$[q_0 a q_0] \rightarrow a [q_0 a q_1] [q_1 a q_1]$$

$$[q_0 a q_1] \rightarrow a [q_0 a q_0] [q_0 a q_1]$$

$$[q_0 a q_1] \rightarrow a [q_0 a q_1] [q_0 a q_0]$$

$$\rightarrow \delta(q_0, b, a) = (q_1, \epsilon)$$

$$[q_0 a q_0] \rightarrow b [q_1 a q_0]$$

$$[q_0 a q_1] \rightarrow b [q_1 a q_1]$$

$$\rightarrow \delta(q_1, b, a) = (q_1, \epsilon)$$

$$[q_1 a q_1] \rightarrow b$$

$$S \rightarrow [q_0 z_0 q_0]$$

$$S \rightarrow [q_0 z_0 q_1]$$

$$\rightarrow \delta(q_1, \epsilon, z_0) = (q_1, \epsilon)$$

$$[q_1 z_0 q_1] \rightarrow \epsilon$$

Construct ^{CFG} PDA for the given PDA

$$\delta(q_1, 1, z_0) = (q_1, xz_0)$$

$$\delta(p, 0, z_0) = (q_1, z_0)$$

$$\delta(q_1, 1, x) = (q_1, xx)$$

$$\delta(q_1, 0, x) = (p, x)$$

$$P = \{q, p, q\}, \{0, 1\}, \{x, z_0\}, \{ \epsilon, q_1, z_0, \epsilon, p \}$$

$$\delta(q_1, \epsilon, x) = (q_1, \epsilon)$$

$$\delta(p, 1, x) = (p, \epsilon)$$

Sol:- let S be the start symbol

let $CFG \Rightarrow G = (V, T, P, S)$

Find Variables

$$V = \{S, [P \bar{x} P], [P x q], [q x \bar{p}], [q x q], \\ [P z o P], [P z o q], [q z o P], [q z o q]\}$$

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

$P \rightarrow$ Find productions: $S \rightarrow q z o q$ $S \rightarrow q z o P$

$$\rightarrow \delta(q, 1, z_0) = (q, x z_0)$$

$$[q z o q] \rightarrow 1 [q x q] [q z o q]$$

$$[q z o q] \rightarrow 1 [q x P] [P z o q]$$

$$[q z o P] \rightarrow 1 [q x q] [q z o P]$$

$$[q z o P] \rightarrow 1 [q x P] [P z o P]$$

$$\rightarrow \delta(q, 1, x) = (q, x x)$$

$$[q x q] \rightarrow 1 [q x q] [q x q]$$

$$[q x q] \rightarrow 1 [q x P] [P x q]$$

$$[q x P] \rightarrow 1 [q x q] [q x P]$$

$$[q x P] \rightarrow 1 [q x P] [P x P]$$

$$\rightarrow \delta(q, 0, x) = (p, x)$$

$$[a \times a] \rightarrow 0 [p \times a]$$

$$[a \times p] \rightarrow 0 [p \times a]$$

$$\rightarrow \delta(q, \epsilon, x) = (q, \epsilon)$$

$$[a \times a] \rightarrow \epsilon$$

$$\rightarrow \delta(p, 1, x) = (p, \epsilon)$$

$$[p \times p] = \epsilon$$

$$\rightarrow \delta(p, 0, z_0) = (q, z_0)$$

$$[p \ z_0 \ q] = z_0 [q \ z_0 \ q]$$

$$[p \ z_0 \ p] \rightarrow 0 [q \ z_0 \ p]$$

Turing Machine:

Turing machine consists of a finite control, which can be any finite set of states.

$\rightarrow$ There is a tape divided into cells, each cell can hold finite number of symbols.

Finite
control

...	B	B	x_1	x_2	...	x_n	B	B	...
-----	---	---	-------	-------	-----	-------	---	---	-----

$x_1, x_2, \dots, x_n \rightarrow$ Input Symbols

B $\rightarrow$ Blank symbol

→ choose input string from the input alphabet and place in tape.

→ Extend the length of the tape to the left and right and place blank symbol over the extended cells.

→ The Blank is a tape but not input symbol.

→ There is a read write head (or) tape head. Initially the tape head is at leftmost cell that holds the input.

→ A move of the Turing machine is a function of the state of the finite control and tape symbol scanned.

→ In one move a Turing machine will

1) change state: The next state optionally may be same as the current state.

2) Write a tape symbol to the cell scanned, this tape symbol replaces the symbol in that cell.

3) move the tape head to left or right but don't allow the head to remain stationary.

Defination:-

Turing machine is a "7 tuple" notation. And the symbol is $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$.

$\Gamma \rightarrow$ input tape / Tape symbols

$B \rightarrow$ Blank symbol

$Q \rightarrow$ set of states

$\Sigma \rightarrow$ set of input symbols

$q_0 \rightarrow$ Initial/Starting state

$F \rightarrow$ Final state set

$\delta \rightarrow$ Transition Function

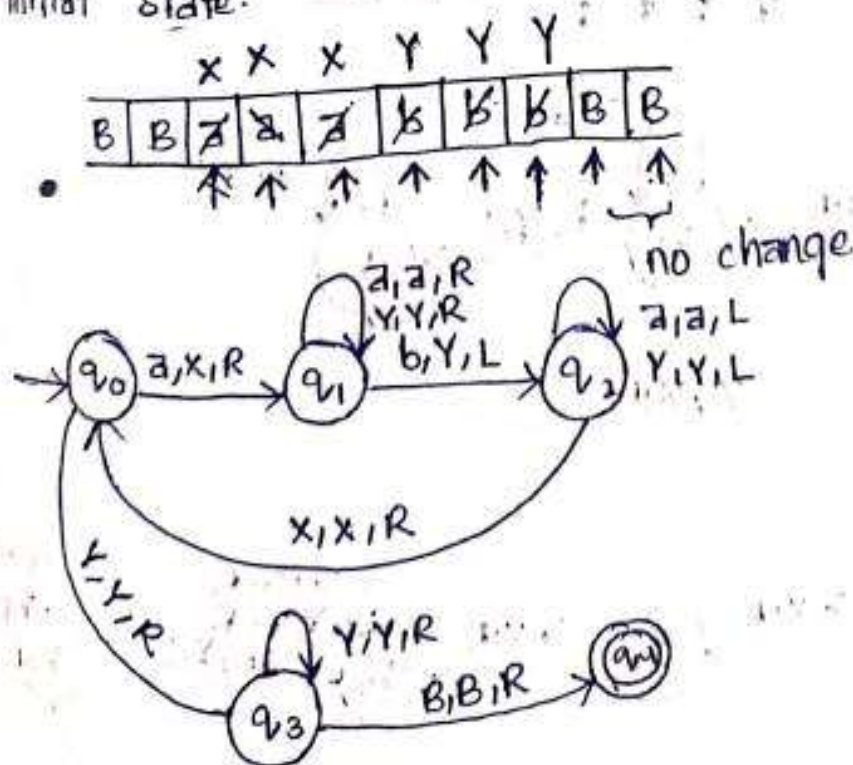
where, the transition function maps to
 $Q \times \Sigma \Rightarrow Q \times \Gamma \times \{L, R\}$

Transition diagram in Turing machine:

Q Design Turing machine for the language $L = \{a^n b^n \mid n \geq 0\}$.

A:- $L = \{ab, aabb, aaabbb, \dots\}$

let "aaabbb" be the input tape, q_0 be the initial state.



$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

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

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

$$\Gamma = \{a, b, x, y, B\}$$

$$F = \{q_4\}$$

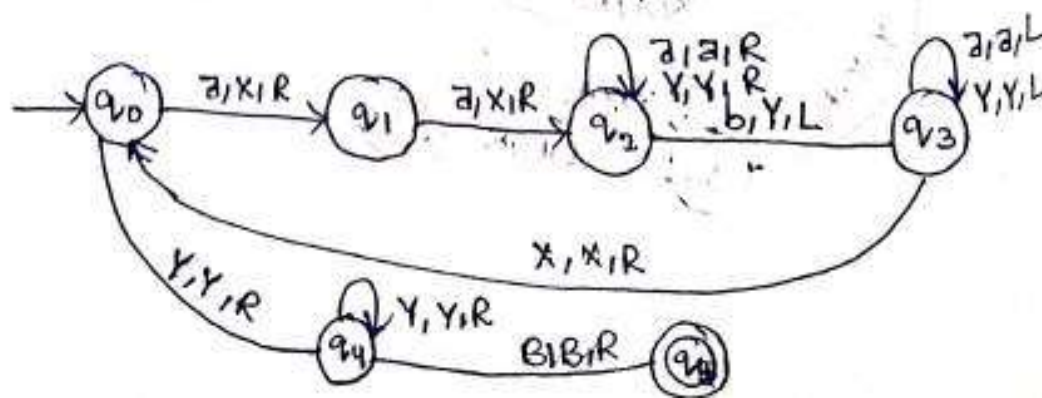
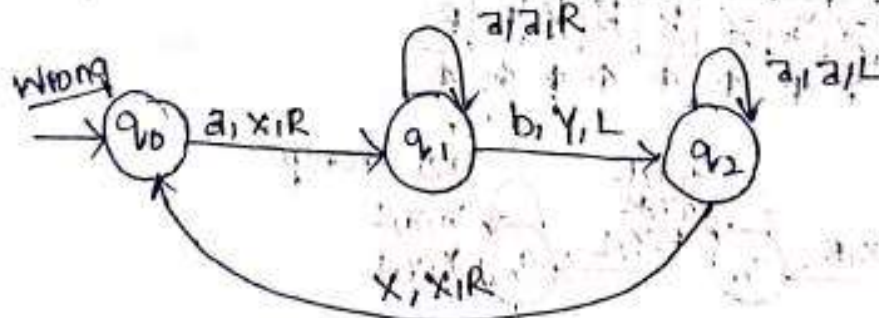
② Design Turing machine for the language

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

A:- $L = \{aab, aaaabb, aaaaaabbb, \dots\}$

Let "aaaabb" be the input tape and q_0 be the initial state. [∵ For 2 a's we need to change to x.]

		x	x	x	x	y	y		
B	B	a	a	a	a	b	b	B	B
		↑	↑	↑	↑				



$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

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

$$\Gamma = \{a, b, X, Y, B\}$$

$$F = \{q_5\}$$

Transition Table:-

	a	b	X	Y	B
q_0	q_1, X, R	-	-	q_4, Y, R	-
q_1	q_2, X, R	-	-	-	-
q_2	q_2, a, R	q_3, b, L	-	q_2, Y, R	-
q_3	q_3, a, L	-	q_0, X, R	q_3, Y, L	-
q_4	-	-	-	q_4, Y, R	q_5, B, R
q_5	-	-	-	-	-

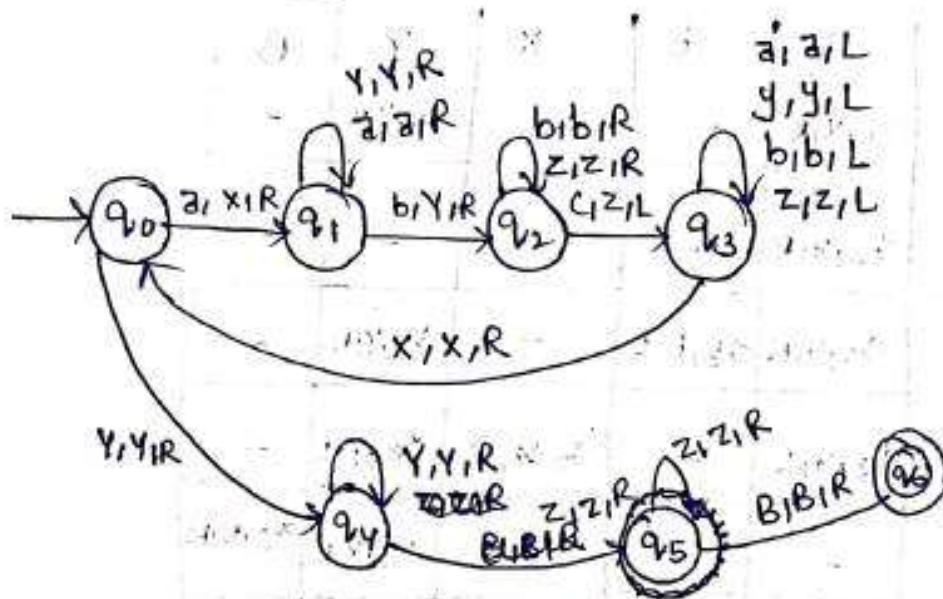
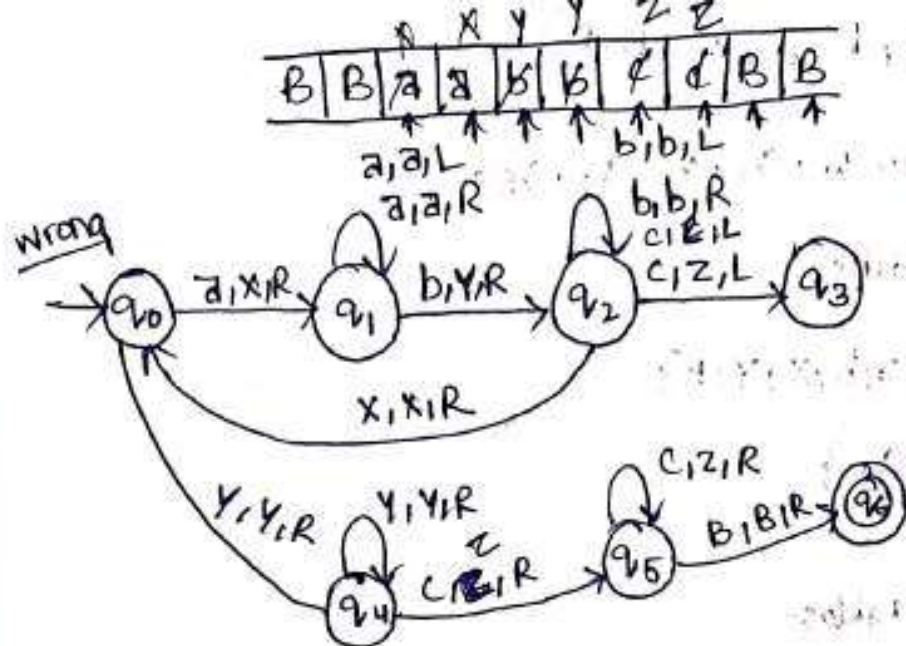
③ Design Turing machine for the given language

$$L = \{a^n b^n c^n \mid n > 0\}$$

Ans:- $L = \{abc, aabbcc, aaabbbccc, \dots\}$

let "aabbcc" be the input tape and q_0

be the initial state.



$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

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

$$\Gamma = \{a, b, c, x, y, B\}$$

$$F = \{q_6\}$$

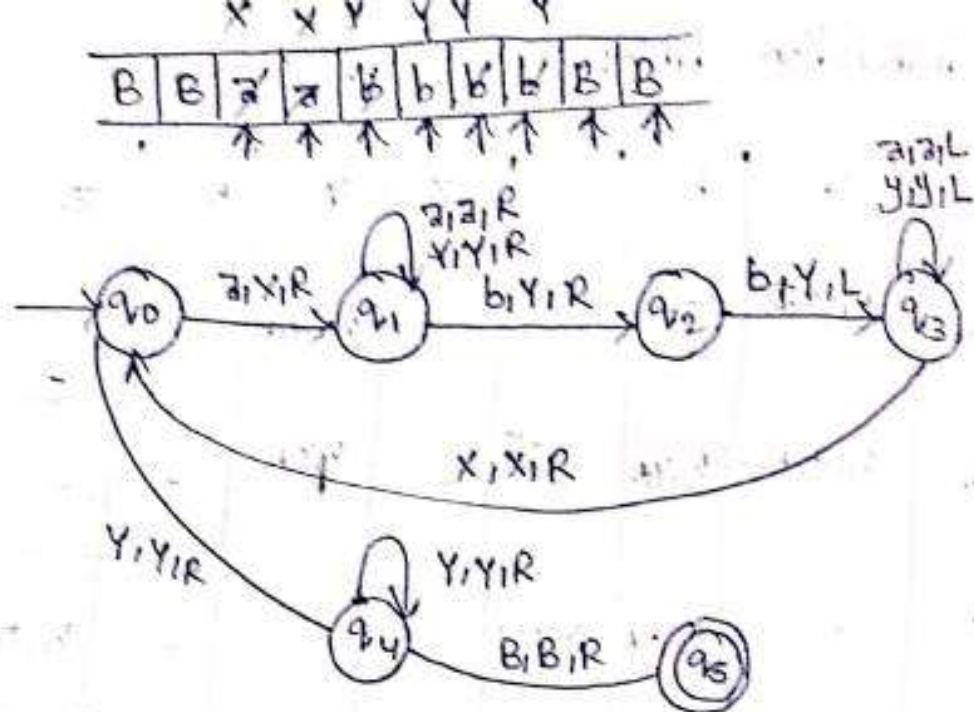
Transition Table:

	a	b	c	x	y	B	z
q_0	q_1, x, R	-	-	-	q_4, y, R	-	-
q_1	q_{11}, a, R	q_2, y, R	-	-	q_4, y, R	-	-
q_2	-	q_{21}, b, R	q_{21}, z, L	-	-	-	q_{21}, z, R
q_3	q_{31}, a, L	q_{31}, b, L	-	q_{31}, x, R	q_{31}, y, L	-	q_{31}, z, L
q_4	-	-	-	-	q_{41}, y, R	-	q_{51}, z, R
q_5	-	-	-	-	-	q_{61}, B, R	q_{51}, z, R
q_6	-	-	-	-	-	-	-

④ Design Turing machine for the language $L = \{a^n b^{2n} / n \geq 1\}$

A:- $L = \{abb, aabbbb, aaabbbbbbb, \dots\}$

let "aabbbb" be the input tape and q_0 be the initial state [$\because$ for $a^1 b^2$'s we need to change to y]



$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$\Gamma = \{a, b, x, y, B\}$$

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

$$F = \{q_5\}$$

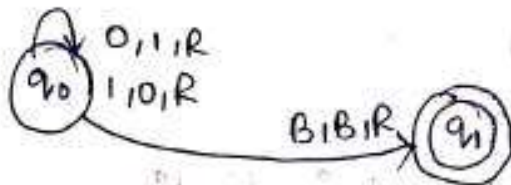
Transition Table:

	a	b	x	y	B
q_0	q_0, a, R	-	-	q_4, y, R	-
q_1	q_1, a, R	q_2, b, R	-	q_1, y, R	-
q_2	-	q_2, b, L	-	-	-
q_3	q_3, a, L	-	q_0, x, R	q_3, y, L	-
q_4	-	-	-	q_4, y, R	q_5, B, R
q_5	-	-	-	-	-

- 5) construct Turing machine for
 i) Ones complement ii) Two's complement.

Ans- Ones complement

B | 1 | 0 | 1 | 0 | B



⇒ Two's complement

1 0 1 0 1

0 1 0 1 0

0 1 0 1 1

- ii) Two's complement

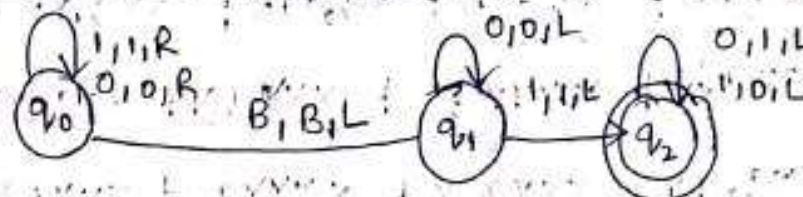
1 0 1 0

0 1 0 1

0 1 1 0

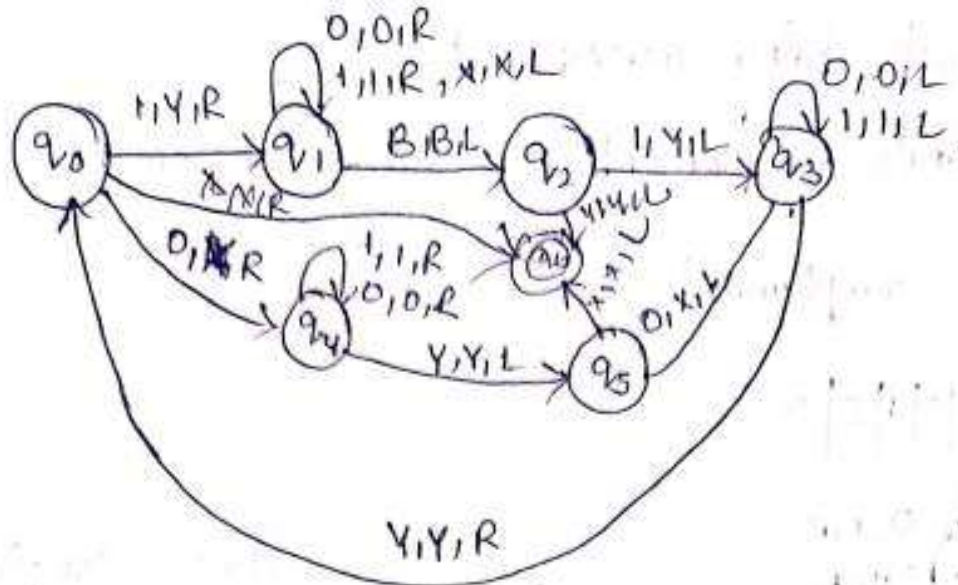
B | 0 | 1 | 1 | 0 | B

B | 0 | 1 | 0 | 1 | 1 | B



- 6) construct Turing machine for the palindrome of the 0s and 1s.

0 1 0, 0 0 0, 0 1 0 1, 1 1



Instantaneous Description For Turing Machine:-

Let "aaabbb" be the input string/tape.

Acceptance For the Turing machine:

$w = aabb$

$q_0 a a b b \vdash x q_1 a b b \vdash x a q_1 b b \vdash x a y q_2 b \vdash$

$x a q_2 y b \vdash x q_2 a y b \vdash x a_0 a y b \vdash x x q_1 y b \vdash$

$x x y q_1 b \vdash x x y y q_2 \vdash x x y q_2 y \vdash x x q_3 y y \vdash$

$x x q_0 y y \vdash x x y q_3 y \vdash x x y y q_3 \vdash x x y y q_4$

$\therefore$ The Acceptance of Turing machine is verified.

Language of Turing machine:

Language of Turing machine M is denoted by $L(M)$ and it is defined as

$$L(M) = \{w \text{ in } \Sigma^* / q_0 w \vdash^* \alpha p \beta\}$$

$\Rightarrow \alpha, \beta$ are any strings of tape symbols

$\Rightarrow P$ can be any state.

Undecidability :-

Recursive language: A language, if there exists a Turing machine, that accepts all strings in L and rejects all strings not in L .

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

Let "abab" be the input tape symbols, if we find the "Acceptance" for this, this is not present in the language then it is a "recursive language".

Recursively Enumerable language: If L is a language, if there exists a Turing machine that accepts all strings in L that may or may not reject all strings in L .

$\rightarrow$ These are said to be partially decidable.

⇒ The

Decidability:-

The recursive languages are said to be decidable.

Undecidability:-

A language which is not decidable is said to be undecidable.

→ There is no Turing machine for the undecidable languages.

→ The languages which are not recursively enumerable.

Language that is not recursively enumerable:-

1) In order to prove that not a recursively enumerable we need to

i) Enumerating binary strings:- In order to prove this we need to write "binary strings".

$\{ \epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots \}$

Here " ϵ " is the "0th" string

"0" is the "1st" string and so on.

Constructing codes for Turing machine:

→ Assign positive integers to the symbols of transition in Turing machine.

→ For input symbols, Assign the / Take the $x_1(0)$,

$x_2(1)$ and $x_3(8)$.

→ For directions, Take the $D_1(L)$, $D_2(R)$.

Diagonalized language: It is represented by " L_d ".

It is a set of strings " w_i " and " w_i " is not in " $L(M_i)$ " for some M_i .

Table of Acceptance: We have write "0 and 1" in the "acceptance table". Rows we take Turing machine and for columns we take strings.

The strings must be accepted by the Turing machine.

→ If Turing machine accepts take "1", if not accepts we take "0".

→ $L(M_i)$ will have the diagonal of w_i and M_i .

Proof of the language that is not recursively enumerable (Diagonalized language):

Let L_d be $L(M_i)$ for some M_i .

1) If w_i is in L_d , then w_i is in $L(M_i)$.

But, by the definition L_d , w_i is not in the ~~same~~ L_d .

2) If w_i is not there in L_d , then it is not there in $L(M_i)$.

But by the definition of L_d , w_i is in the L_d .

∴ There is no Turing machine for the L_d .

So that we can have Turing machine for L_d , which means L_d is not recursively enumerable language. This proof is called as "proof by contradiction". This means: Firstly we assume that there will be the L_d later that we assume there will not have L_d .

Undecidable problem that is not recursively
enumerable:

Complements of recursive and recursively enumerable
languages:

Theorem 1: If L is recursive then, $\bar{L}$ is also recursive.

proof:- let " L " be a recursive language.

so, " L " halts always for Turing machine " M ".

→ Let M be the Turing machine of $\bar{A}$.

~~To some changes~~ To get M' , do some changes:

→ Make all accepting states as non-accepting states.

→ Take a new state as a accepting state and write transitions from each non-accepting state of M on each tape symbol to the accepting state.

The Turing machine M' accepts strings which are not accepted by M and rejects all strings which are accepted by M .

That means, the Turing machine M' always halts.

$\therefore \bar{A}$ is recursive.

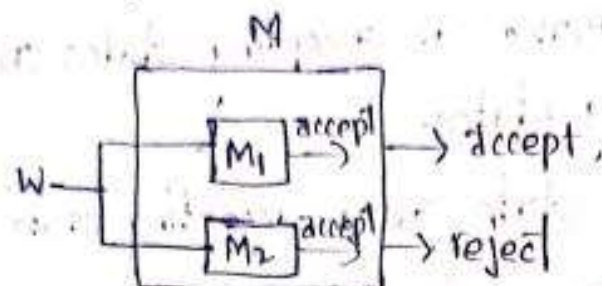
Theorem:- If A and $\bar{A}$ are recursively enumerable then A is recursive.

Proof:- Take a Turing Machine M .

(Let A be recursive)

Let M_1 be the Turing machine for A and M_2 be the Turing machine for $\bar{A}$.

we can have a Turing machine 'M' by taking M_1 & M_2 as tapes.



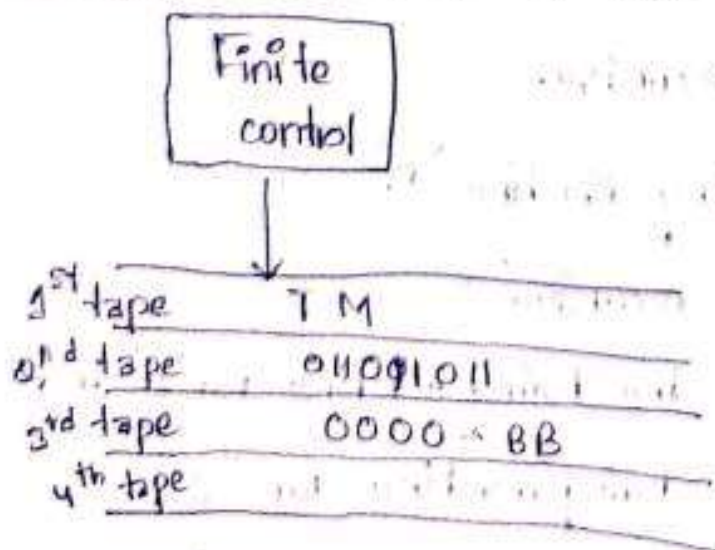
"M" acts same as "M"

Since "M" always halts

"L" is recursive

Universal language:- set of

It is a language of strings representing Turing machine and an input that is accepted by the Turing machine.



→ 1st tape holds TM, w

→ 2nd tape holds simulated code of TM

- 3rd tape holds the state q_i in form.
- 4th tape is used to scratch.
- It is represented by the Δ .

① Examine input to ensure that the code of Turing machine is legitimate.

② place input in 2nd tape in encoded form i.e., place 10 for each '0', 100 for each 1 and 1000 for each B.

③ place start state (q_0) in tape 3, and also place tape head at 1st symbol of simulated in 2nd tape.

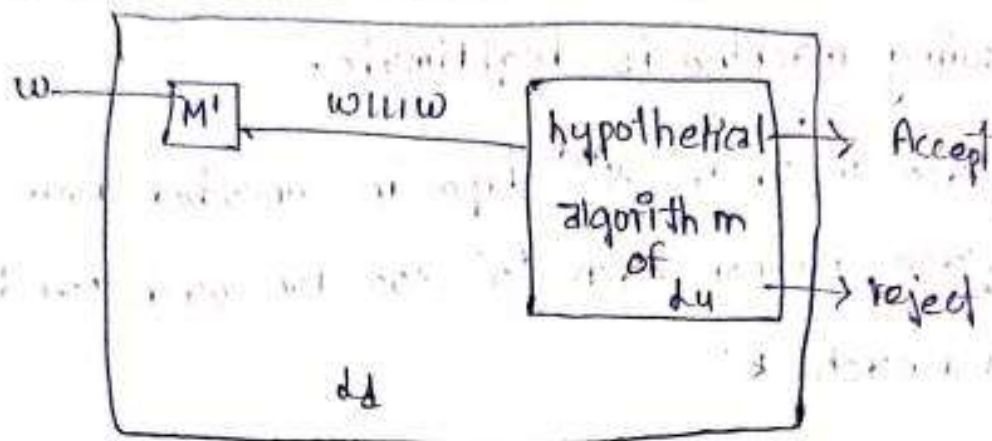
④ for each move of TM of the form

$$s(q_i, x_j) = (q_k, x_l, D_m)$$

- a) replace the contents x_j with x_l
- b) change the content of tape 3 by places q_k and make all previous q_i 's 10's.

Theorems If L_u is RE, then L_u is not recursive

proof: let L_u is Recursively Enumerable, then there exists a Turing Machine M .



If there exist TM for L_u then TM can also possible for L_d .

As per the definition of L_d , there exist no TM for L_d .

So that there exist no TM for L_u i.e.

L_u is not recursively enumerable.

So that L_u is not recursive.

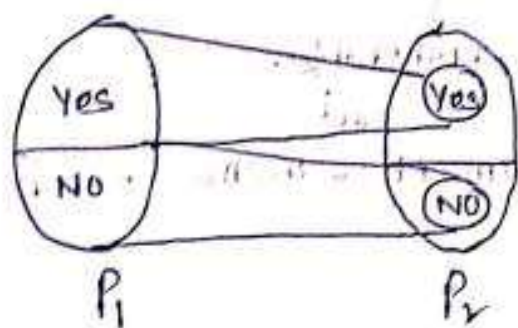
Undecidability about Turing machine:

Turing Reducability: It is a technique to convert one language into another language, when L_1 is recursive in L_2 .

→ In general an instance of, if there is an algorithm to convert P_1 to P_2 then we can say that an instance of P_1 is reduced to an instance of P_2 with the same answer.

→ If P_1 is (recursively enumerable) is not recursive then P_2 is not recursive.

→ If P_2 is not recursive then P_1 can't be recursive.



Reduction turns the positive.

Reduction from P_1 to P_2 is TM which take an instance of P_1 and accepts instance of P_2 .

In General, The reduction from P_1 to P_2 is program which takes instance of P_1 as Input.

Theorems- If there is reduction from P_1 to P_2

a) If P_1 is undecidable, then P_2 is also undecidable.

Proof:- Let P_1 is undecidable and then exists an algorithm to solve P_2 .

So, P_2 is decidable, but P_1 is reduced to P_2 .

Since, there is reduction from P_1 to P_2 and P_1 is undecidable.

So that P_2 is also undecidable.

b) If P_1 is non-recursively enumerable then P_2 is also non-recursively enumerable.

Proof:- Let P_1 be form RE ^{and} there exists an algorithm for P_2 .

Since there is a reduction from P_1 to P_2 and P_1 is not RE P_2 is non RE.

Turing Machine to accept empty language:

$L_e \rightarrow$ empty language

$L_{ne} \rightarrow$ non-empty language

$$L_e = \{M / L(M) = \emptyset\}$$

$$L_{ne} = \{M / L(M) \neq \emptyset\}$$

$\rightarrow L_e$ and L_{ne} are complement to each other.

→ L_1 is recursively enumerable, and L_2 is not recursive.
→ L_2 is not recursively enumerable.

Rice Theorem and Property of RE languages :-

Property of RE languages:- It is a set of RE languages. There are two properties

① Trivial Property of RE language: A property said to be Trivial, when it is has no strings or when it is empty or a set of all RE languages.

② Non-Trivial Property of RE languages: A property said to be non-trivial, when it is a non-empty not a set of all RE languages.

Rice Theorem:-

For Every non-Trivial property of a recursively enumerable language is undecidable.

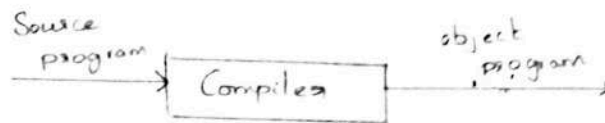
Implementation of Turing machine specifications:-

Applications of (Proof of) Rice Theorem:-

① The Rice theorem is used to say the given problems are undecidable or not.

Compiler:

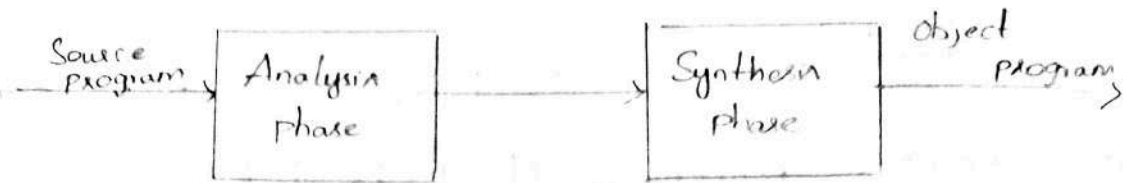
Compiler is a program which converts the source program (high level language) into the object program (low level language)



Basic functions of the Compiler:

1. It converts high level language into low level language
2. It detects the errors in the program and reports to the user
3. Reporting of error is in high level language (inspite of machine language)
4. Compiler creates a data structure called symbol table where the identifiers and constants are stored.

Phases of the Compiler:



Analysis model of Computer.

Compilation process is divided into

1. Analysis phase

2. Synthesis phase

Analysis phase:

It divides the source program into constituent pieces and checks the syntax & semantics

It consist of 1. lexical analyser

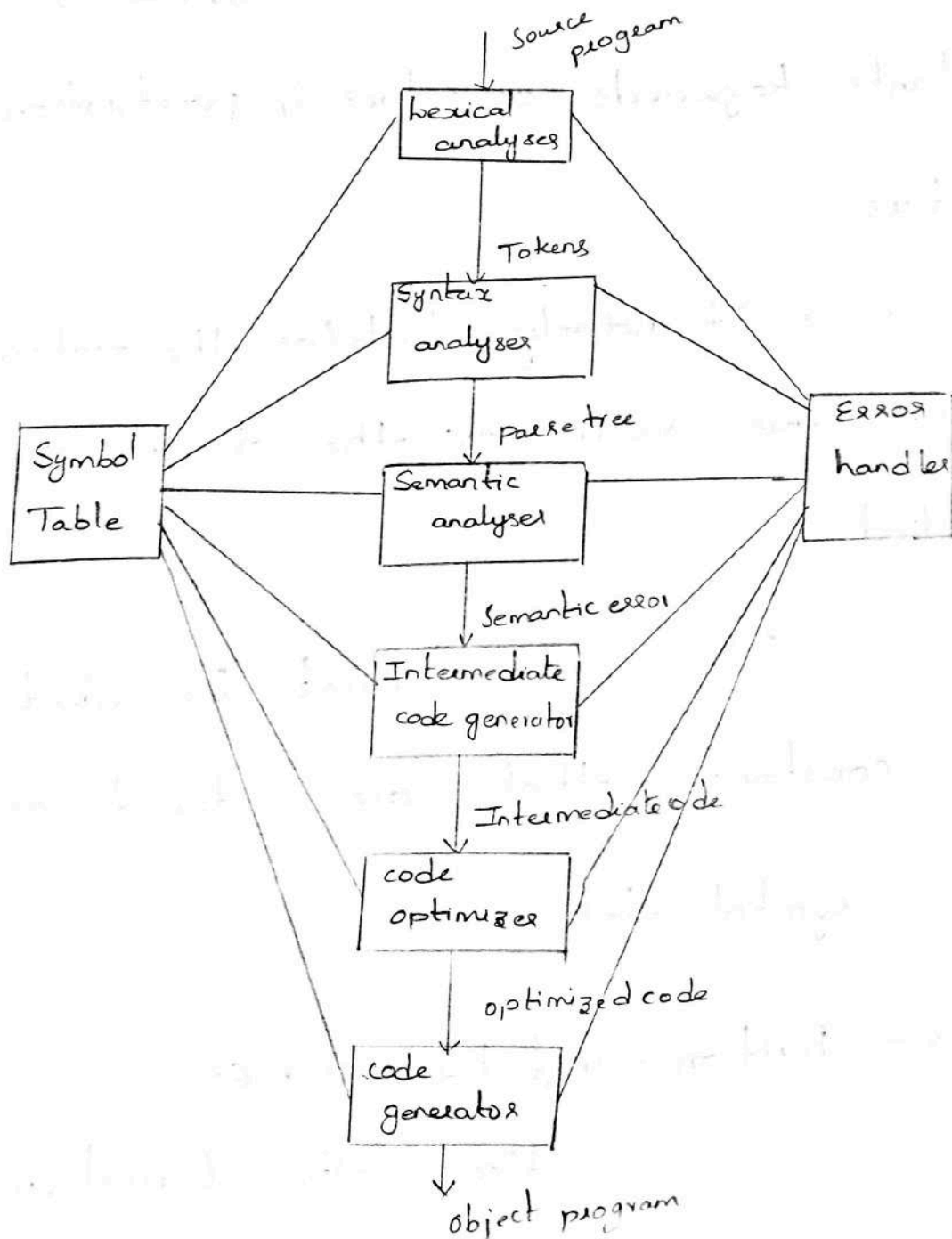
2. Syntax analyser

3. Semantic analyser

Synthesis phase:

It converts the source program into intermediate form and optimize it to get improved object program.

It consist of intermediate code generator, code optimizer & code generator.
Phases of Compiler:



Lexical Analyser (Scanner):-

It reads the source program & group them into stream of tokens

Tokens represents the sequence of characters that are identifiers, constants, keywords, operators & punctuation characters.

It not only identifies the token but also the value of the token are identified

The value exist for identifiers & constants that are stored in the symbol table

Ex:- $\text{Position} = \text{initial} + \text{rate} * 60$

Here the lexical analyser recognises the position, initial & rate

as identifiers & 60 as constant &
stores it in the symbol table.

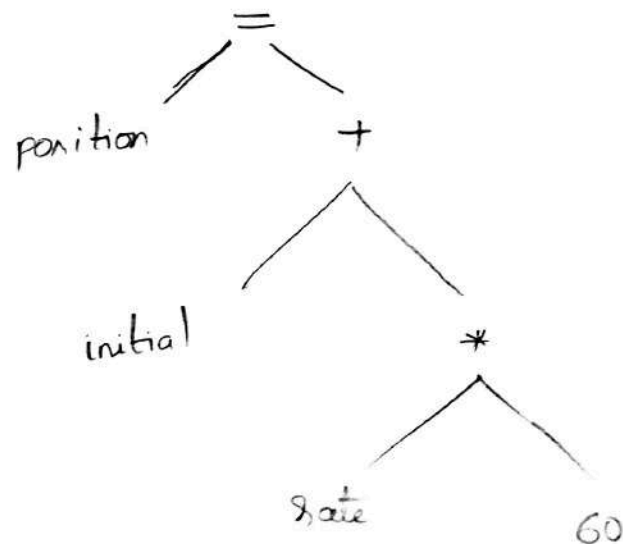
Syntax Analyses (Parser):

There are two functions of
the syntax analyses.

1. It checks for the tokens given
by the lexical analyses.

2. It creates a tree structure
using the tokens by identifying which
part of the token should be grouped
together is known as parse tree

Ex:



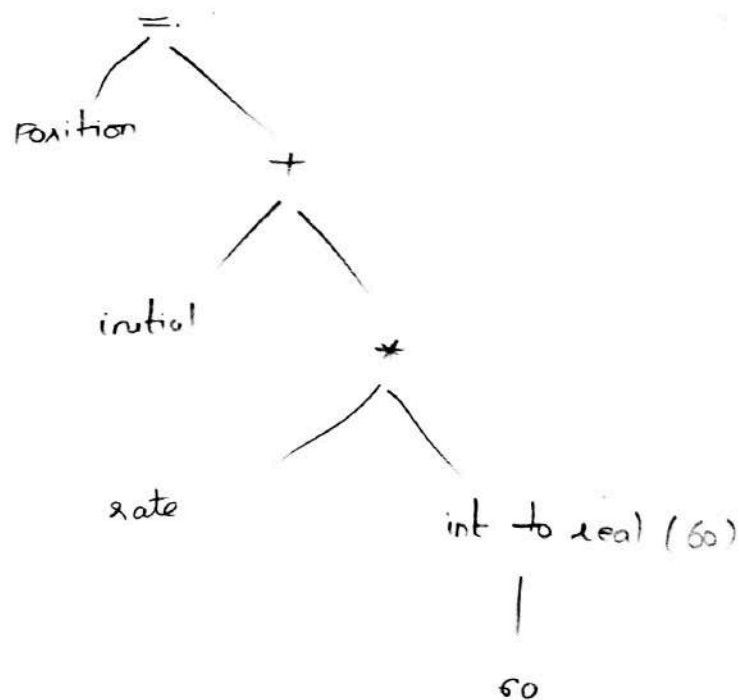
Semantic Analyses (Type checker):

It performs type checking of operands & operators.

It checks each operator has necessary operands to perform the operation.

Automatic type conversion is also done in this phase.

Ex: 1. If we apply multiplication operator on state which is a real type of data on integer 60, the 60 will be implicitly converted into real type.



Ex: 2. Modular division ($\%$) operator is operated only on integers and it will be checked by semantic analyser.

Intermediate Code generator:-

It has two properties

1. It should be easy to convert the source program into intermediate form
2. It should be easy to convert the intermediate form into object code

It transforms the parse tree into intermediate ~~language~~ language may be of three forms. They are

1. Postfix notation

2. Syntax tree

3. Three address code

Most of the compilers uses three address code as an intermediate language which is like a assembly code where each and every memory location acts like a registers.

The intermediate code (three address code) for the given example is

$temp_1 = int_to_real(60)$

$temp_2 = rate * temp_1$

$temp_3 = initial + temp_2$

$position = temp_3$

Code optimizer

It is an optional phase.

It is an attempt to improve the intermediate code so that faster running object code is ultimately produced

The compiler which makes use of code optimization phase is called as optimizing compilers.

It improves the running time of a compiler and only a fraction of time is spent during this phase

The optimized code is

$$\text{temp1} = \text{rate} * 60.0$$

$$\text{position} = \text{initial} + \text{temp1}$$

Code generator:

It generates the object code which is normally an assembly code

The memory locations are selected for each variables which will be assigned to a registers.

The assembly code generated by the code generator is

MOV #60.0 R1

MULF rate R1

ADDF initial R1

Symbol Table:

It is a data structure containing the records for each identifier along with the attribute informations such as type, value and storage allocated for the identifiers

Use of Symbol table:

1. It is used to find the record for each identifier quickly
2. It is easy to retrieve and store the identifiers in the symbol table
3. As soon as the source program enters the lexical analyzer the identifiers and constants are stored in the symbol table

Error handler:

It detects the error in each phase and report immediately

It should not slow down the processing of correct program

It should detect and report the error and continue further processing without stopping the rest of the program

Types of errors:

1. During lexical analyzer the token formation errors have been detected

2. During syntax analysis phase the violation of structural rules, errors will be encountered

Ex: Unbalanced parenthesis

3. During semantic analysis phase type checking errors will be encountered

Ex: Invalid operand applied to an operator

4. logical errors

Ex: infinite recursive calls

* Difference between Compiler & Interpreter

Compiler

1. Compiler is a language translator which converts the source program into object program

2. It scans the whole program at a time

3. Compilation process takes less time for execution

4. Compiler requires more memory for execution

Interpreter

1. Interpreter is a language translator which translates one statement at a time into an intermediate form

2. It converts one instruction at a time

3. Interpretation process takes more time for execution

4. Interpreter requires less memory for execution

5. Compiler process the source program according to the physical flow of input sequence.

6. Compiler is not suitable for command level languages

7. The cost of decoding must be paid after the entire program is executed

8. Ex:- C, C++, PASCAL, COBOL

5. Interpreter process the source program according to logical flow

6. Interpreter is suitable for command level language

7. The cost of decoding must be paid after each & every instruction is executed.

8. Ex:- LISP, APL, Snow ball

Pass and phases of the compiling :-

Phase :-

It converts the source program from one representation into another representation.

Pass :-

It is a physical scan of source program which consist of one or more phases.

Phases are collected into

1. Front end

2. Back end

Frontend consist of those phases that are dependent on source program and independent of object program

Frontend consist of lexical analyser, syntax, semantic analyser & some part of intermediate code generator in addition to symbol table & error handler.

Backend consist of those phases that are dependent on object program and independent of source program.

Backend consist of the remaining part of intermediate code generator, code optimizer & code generator in addition to symbol table & error handler.

Difference between phase & pass:

Phase

1. It converts the source program from one representation into another
2. Splitting into more number of phases reduces the complexity
3. Reduction in number of phases increases the complexity

Pass

1. It is a physical scan of source program which consists of one or more phases
2. Splitting into more number of passes increases the memory space
3. Reduction in number of passes decreases the memory space

4. During the conversion there is no need of an intermediate file

5. The operation of phase is to transfer the control from one phase to the another phase

4. During the conversion there is a need of intermediate file

5. The code It gets stored in an intermediate file

Keywords of lexical analyzer:-

Token:-

It is a sequence of characters that have collective meaning

Ex:- Operators, keywords, identifiers, constants, punctuation characters.

Pattern:-

It is a rule that describes the token

Ex:- identifier has the pattern which

starts with a letter followed by
a letter or digit repeatedly.

Lexeme :-

It is a sequence of
character matched by a pattern for
a token

Ex:- Token	- Lexeme	Pattern
Identifier	pi	starts with a letter followed by a letter or digit repeatedly
Constant	70.0	Numerical value

Show the output of all the phases of
the compiler for the following line of code

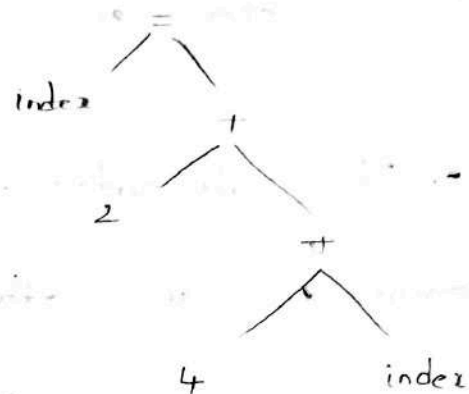
$index = 2 + 4 + index$

$$\text{index} = 2 + 4 + \text{index}$$

↓
Lexical analyzer

The tokens generated by lexical analyzer phase are identifiers is index & constants are 2 & 4 & operators are '=' & '+'

↓
Syntax analyzer



↓
Semantic analyzer

↓
Intermediate code generator

$$\text{temp1} = \text{index}$$

$$\text{temp2} = 4 + \text{temp1}$$

$$\text{temp3} = 2 + \text{temp2}$$

$$\text{index} = \text{temp3}$$

↓
Code optimizer

$$\text{temp1} = 2 + 4$$

$$\text{index} = \text{temp1} + \text{index}$$

↓
Code generator

MOV #2 R1

ADD 4 R1

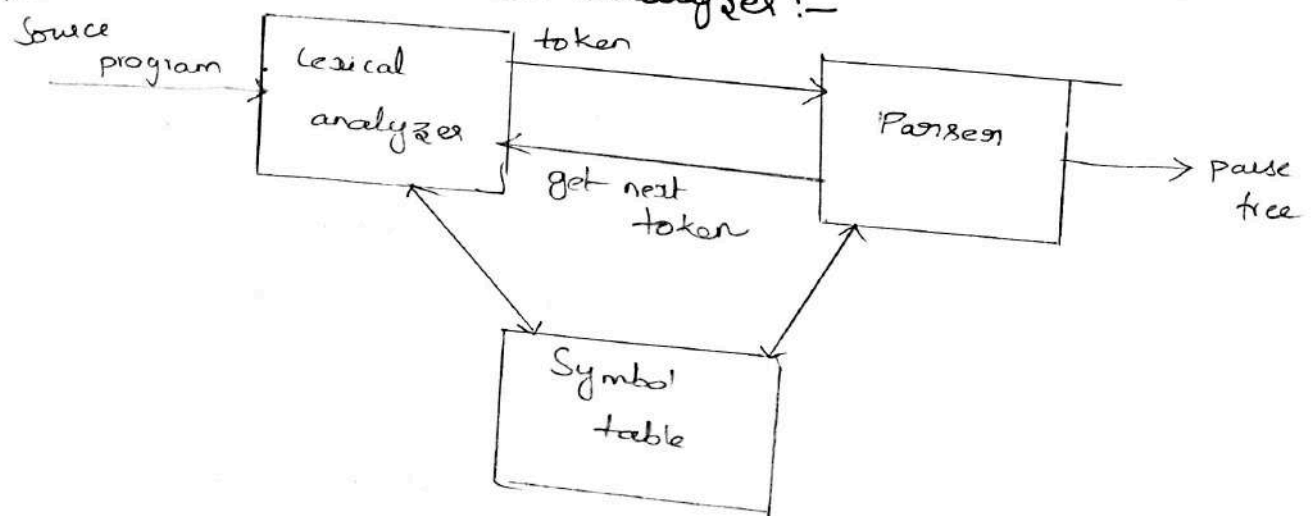
ADD index R1

MOV R1 index

Basic functions of Lexical analyzer:

1. It reads the source program & divides them into tokens such as identifiers, keywords, constants, operators & punctuation characters
2. It eliminates the white spaces & ⁺ comments in the source program
3. It stores the identifiers & constants in the symbol table along with the attribute information.

Role of lexical analyzer:-



The main purpose of lexical analyzer is to read the characters in the source program & divide them into tokens.

The lexical analyzer and parser works interactively with each other where lexical analyzer acts as a subroutine to the parser, which is called by the parser whenever it needs next token information.

Lexical analyzer returns a representation of a token that consist of a pair of information such as an integer code that specifies the category of token such as identifiers, keywords, operators etc & pointers that points to the symbol table.

The pointers is used for the token to get the attribute informations from the symbol table

Reasons ^{Issues} for separating lexical analyzer from parser:-

1. Simple Design
1. The separation of lexical analyzer from the parser results in simpler design of the compiler.

Ex:- To avoid the complexity the elimination of white spaces & comments are done in lexical analysis phase instead of doing it in syntax analysis phase

2. Compiler efficiency is improved.

Ex:- Large amount of time is spent in reading the source program & partitioning them into tokens. So a specialized buffering technique is used for reading the input characters & processing them into tokens to speed up the performance of the compiler.

3. Compiler portability is enhanced.

Ex:- The representation of special & nonstandard symbols can be isolated in the lexical analyzer phase i.e. *p is identified as pointer variable instead of multiplication operators.

Typical Lexical Errors :-

If a misspelled keyword occurs in the source program, the lexical analyzer neither tells that it is a misspelt word nor as an undeclared one but it returns as a valid identifier & continue the scanning process.

Ex: `fi (a==1) - - -`

Error recovery strategies:-

1. Panic mode recovery:

Deleting successive characters from the input until the lexical analyzer finds a well formed token

Error production

2. Deleting extraneous characters

Phrase level Recovery

3. Inserting the missing character

Global Correction

4. Replacing two adjacent characters

5. Replacing incorrect character by a correct character

Input Buffering:

During lexical analysis phase, the scanner scans from left to right one character at a time to identify the token.

It uses two pointers for doing this analysis:

1) Begin pointer (BP)

2) Forward pointer (FP)

Begin pointer:

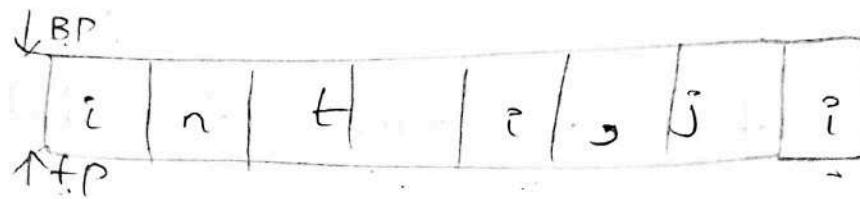
It is used to keep track of the first character of the token.

Forward pointer:

It is used to keep track of next character.

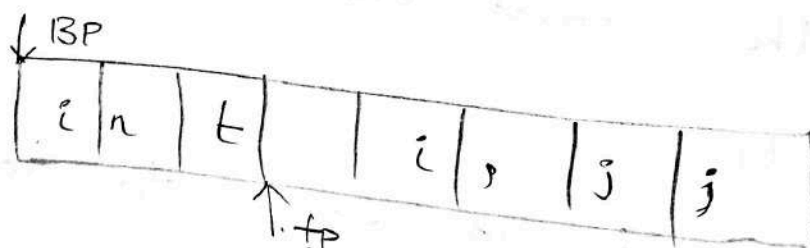
Steps in Scanning the input:

1. Initially both begin & forward pointer points to the first character of the source program

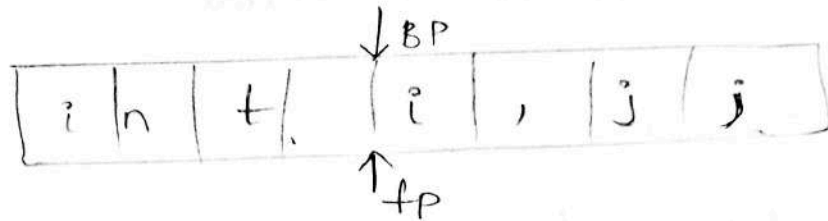


2. The forward pointer scans the source program until there is a match with described token is formed

3. Once the token is formed the forward pointer will be pointing to the right end of the token



4. After processing the token both pointers will be set to point the character immediately after the token



5. The procedure is repeated for entire source program.

6. Usually the source program will reside in the secondary memory and the scanner has to access the secondary memory each & every time to identify the token.

7. As it is time consuming process the input strings are stored in the buffer.

8. The input buffering can be of two types. They are:

1. One buffer Scheme

2. Two buffer Scheme

One buffer Scheme:

In this scheme we use only one buffer of size 'n' characters.

The first 'n' characters of the source program are read into the buffer, the pointers are set to point the first character of the buffer. When the FP reaches the end of the buffer it will be filled with next 'n' characters of source program.

The problem with this implementation is that if the size of next token is greater than 'n' characters, the scheme fails to produce the token.

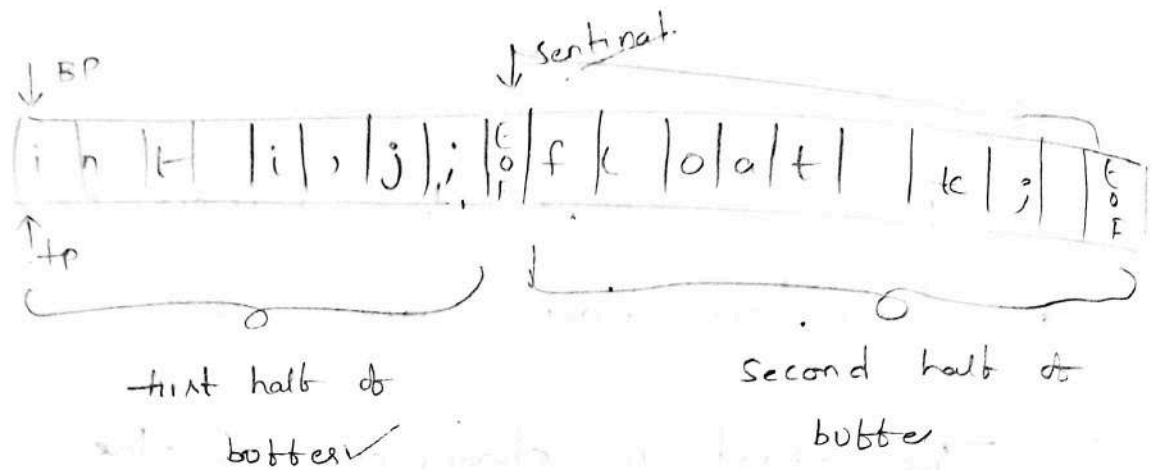
Two buffer Scheme:

In this scheme two buffers of size n characters each are used

The first n characters of the source program are read into the first half of the buffer & it is filled with a special character called Sentinel character which is usually an EOF character to indicate the end of the buffer.

When the pointer reaches the end of the first half then second half will be loaded with next 'n' characters of the source program

when the pointer reaches the end of the second half then the first half will be loaded with next 'n' characters of source program



Algorithm: to advance the forward pointer:-

If forward pointer is at the end of the first half then begin

reload the second half of the buffer

Increment the forward pointer by 1 character
end

Else if forward pointer is at the end of second half then begin

reload the first half of the buffer
set forward pointer pointer to the
first character of the first half of
the buffer

end

else increment the forward pointer by
1 character

*** Bootstrapping:

Compiling a compiler in its own
language is called as Bootstrapping.

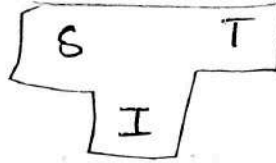
Bootstrapping process is
characterized by 3 languages. They are

1. Source language (S) that it compiles
2. Implementation language (I) that
it was written in
3. Target language (T) that it
generates the code for

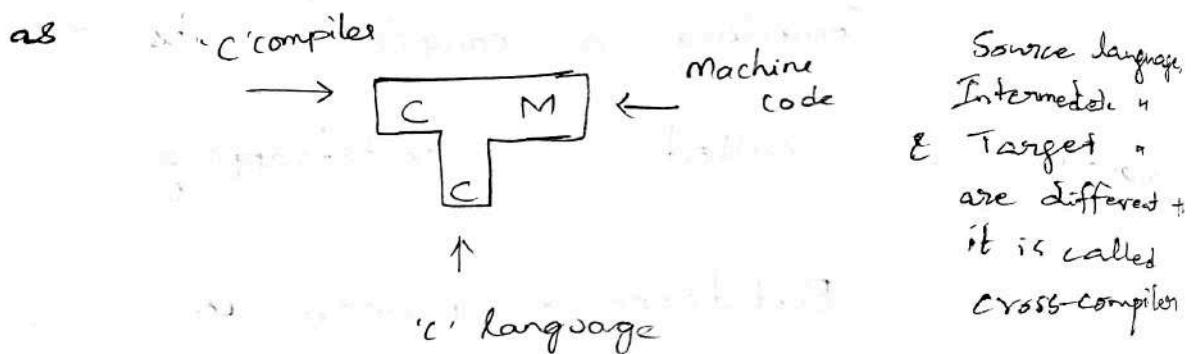
Within ^{test} It is represented as

S_I^T

ϵ in a diagrammatic form as



Example a compiler was written in language itself and it can be given



To create a bootstrapping compiler for the new language 'L' we use the following steps

Step-1:

Create a compiler for the language S which is the subset of

language 'L' written in the assembly language 'A' to generate the code for machine 'A' and it is represented as

$$\begin{matrix} & S & A \\ & \swarrow & \searrow \\ L & & A \end{matrix}$$

Step-2:

Create a compiler for the language 'L' using the language 'S' as an implementation language to produce the code for machine 'A' & it is represented as

$$\begin{matrix} & L & A \\ & \swarrow & \searrow \\ & S & \end{matrix}$$

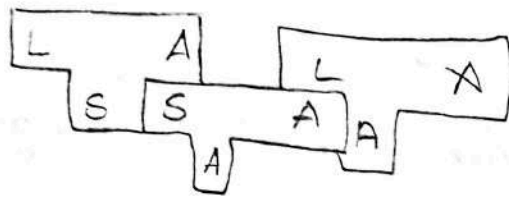
Step-3:

Run the compiler $\begin{matrix} L & A \\ & S \end{matrix}$ in the compiler $\begin{matrix} S & A \\ & A \end{matrix}$ to produce a bootstrapping compiler $\begin{matrix} L & A \\ & A \end{matrix}$

It is represented by an equation

$$\begin{array}{c} L & S & A \\ S & & \end{array} + \begin{array}{c} S & A \\ A & \end{array} = \begin{array}{c} L & A \\ A & \end{array}$$

and in a diagrammatic form as



Cross Compiler:

Some compilers run on one machine and produce the code for another machine is called as cross compiler.

LEX - lexical Analyzer generator.

Lexical analyzer can be implemented by 3 general approaches.

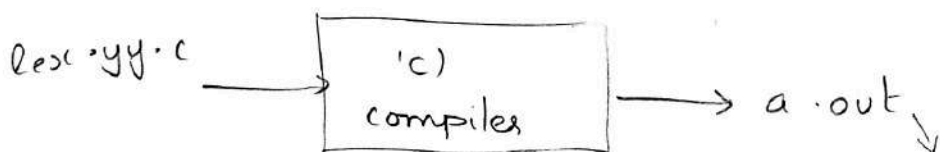
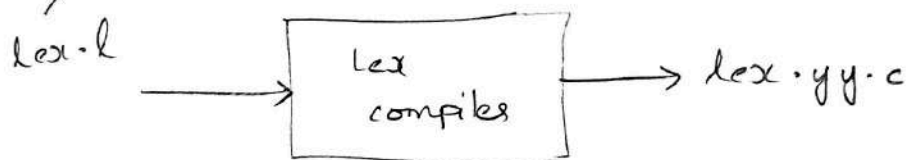
1. Lex compiler

2. Conventional system programming language

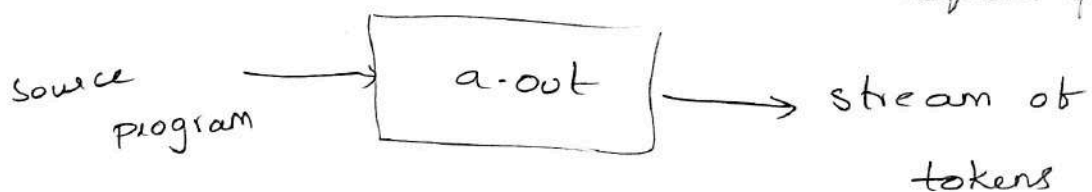
3. Assembly language

Creating lexical analyzer using lex compiler:

A specification of a lexical analyzer is prepared by creating a prog lex.l



lexical analyzer transforms input into sequence of tokens



lex.yy.c program, consists of a tabular representation of transition diagram constructed from the regular expressions together with a standard routine that uses the table to recognize lexemes.

lex is a particular tool that has been widely used to specify the lexical analyzer for variety of programming languages.

We refer this tool as a lex compiler to its input specification as a lex language.

A specification of lexical analyser is created by the program `lex.l` using lex language, then it is run through the lex compiler to produce a 'C' program `lex.yy.c` as an output.

The program `lex.yy.c` consists of tabular representation of transition diagram for the given regular expression

finally lex.yy.c is run through the C compiler to produce an object program a.out which is a lexical analyser that takes the source program as an input and generates tokens as an output.

Lex specification:-

Lex program consists of 3 parts. They are

1. Declaration part

2. Translation rules

3. Auxiliary procedures

Declaration part:-

It consists of the declaration of variables, manifest constant and regular definition.

Manifest constant is the identities used to declare the constant values

Regular definition is the component of regular expression appearing in the translation rules

The declaration part should be declared within the

%. %

%. %

Translation rules:

Translation rules of the lex program should be as follows

$P_1 \{action_1\}$

$P_2 \{action_2\}$

$\vdots$

$P_i \{action_i\}$

%. %

where P_i is the regular expression
and action i is the program
fragment that identifies what
action the lexical analyser has
to do when a pattern matches
the lexeme.

Auxiliary procedures:

The auxiliary procedures are
needed by the action part of the
translation rules and these ^{cedures} ~~productions~~
compiled separately and loaded in the
lex program.

In the lex program the action
part is written in 'C' language.

UNIT-II

Context free grammar:-

①

It gives the syntactic structure of the programming language

It is also called as Backus Normal Form (BNF).

It consist of

1. Terminals

2. Non-Terminals

3. Start Symbol

4. Production rules

Two basic requirements for a grammar are 1. To generate a valid string
2. "recognize" " " "

Terminal:

It is the basic function from which the set of strings are formed

Generally tokens are called as terminals & it is represented in lower case letters.

Terminals are identifiers, keywords, operators, constants & punctuation characters.

Non-Terminal:

It is the syntactic variable that derives the set of strings and it also gives the hierarchical structure of the programming language.

It is represented in uppercase letters and lower case italic letters.

The uppercase letters X, Y, Z and lower case greek letters α, β, γ represents the grammar symbol which can be either terminal or non-terminal.

Start Symbol:-

In the grammar one non-terminal is distinguished as the start symbol & the set of strings it denotes is the language.

Ex:
S
a
b
c
d

generated by the grammars.

The left hand side of the first production is the start symbol.

Production rules:

It specifies the manner in which the terminals & non-terminals can be combined together to form a string

Each production consist of the non-terminal followed by a $\rightarrow$ & string of terminals & non-terminals

If $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots$

$A \rightarrow \alpha_n$ are the productions of A then we can write the grammar as

$A \rightarrow \alpha_1 / \alpha_2 / \alpha_3 \dots / \alpha_n$

Ex:- $E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Terminals:- $+, *, (,), id$

NonTerminals:- E, T, F

start symbol:- E

Ex:

$expr \rightarrow expr \text{ op } expr$ $op \rightarrow -$

$expr \rightarrow (expr)$ $op \rightarrow \%$

$expr \rightarrow - expr$ $op \rightarrow +$

$expr \rightarrow id$ $op \rightarrow \uparrow$

$op \rightarrow +$

Terminals:- $id, +, -, *, /, \uparrow, \%$

non-terminals:- $expr \& op$

start symbol:- $expr$

*** Left recursive grammar :-

The grammar is left recursive if it has the non-terminal

A such that there is a derivation

$A \rightarrow AX$ for some input string where

X is the grammar symbol & A

is the non-terminal which is recursively

called to the left.

Elimination of left recursion :-

$$A \rightarrow AX/Y$$

The above grammar is left recursive as A is recursively called to the left in which X & Y are the grammar symbols & A is the non-terminal.

The left recursion can be eliminated as follows

$$A \rightarrow YA'$$

$$A' \rightarrow XA' / \epsilon$$

$$A \rightarrow \underline{BA'} \quad \text{--- second}$$

$$A' \rightarrow \cancel{XA'} / \epsilon$$

/ remain symbols in the first part

Ex:- 1) $E \rightarrow \underline{E} + \underline{T} / \underline{T}$

$\bar{A} \quad \bar{A} \quad \bar{X} \quad \bar{Y}$

After eliminating left recursion the

above grammar is converted into

$$E \rightarrow TE'$$

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

2) $T \rightarrow T * F / F$

After eliminating left recursion

$$T \rightarrow FT'$$

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

Computation of first and follow:-

first Computation:-

Rule 1:- If x is a terminal symbol then

$\text{first}(x)$ is the same x itself

Ex:- $\text{first}(+) = \{+\}$

Rule 2:-

If X is a non-terminal symbol
then $\text{first}(X)$ is the set of terminals
derived from that non-terminal

$$\text{Ex:- } E' \rightarrow +TE' / \epsilon$$

$$\text{first}(E') = \{+, \epsilon\}$$

Rule 3:-

If there is a production
of the form $A \rightarrow B$ then $\text{first}(A) =$
 $\text{first}(B)$

$$\text{Ex:- } A \rightarrow B$$

$$B \rightarrow a$$

$$\text{first}(A) = \text{first}(B) = a$$

Rule 4:-

$$\text{If } A \rightarrow \epsilon \text{ then } \text{first}(A) = \{\epsilon\}$$

i.e. $\text{first}(A)$ will have ϵ as its

first.

Follow Computation:-

$$\text{Ex:- } E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) / id$$

$$\text{first}(E) = \text{first}(T) =$$

$$\text{first}(T) = \text{first}(F)$$

$$\text{first}(F) = \{ (, id \}$$

$$\text{first}(E') = \{ +, \epsilon \}$$

$$\text{first}(T') = \{ *, \epsilon \}$$

Follow Computation:-

Rule 1:- If S is the starting symbol then $\text{follow}(S)$ should have $\$$ symbol in it.

computation

Rule 2:-

If $A \rightarrow XBY$ then $\text{follow}(B) = \text{first}(Y)$ except ϵ symbol where $X \& Y$ are the grammar symbols & $A \& B$ are non-Terminals.

Rule 3:-

If $A \rightarrow XB$ or $A \rightarrow XBY$ if $\text{first}(Y)$ has ϵ symbol then $\text{follow}(A)$ will be in $\text{follow}(B)$

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

Note:-

If $A \rightarrow XA$ then write the grammar as $A \rightarrow \epsilon XA$ if X is a non-terminal symbol or $A \rightarrow XA\epsilon$ if X is a terminal symbol and proceed from the second rule.

Ex:- $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

$first(E) = first(T) = first(F) = \{ (, id \}$

$first(E') = \{ +, \epsilon \}$

$first(T') = \{ *, \epsilon \}$

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

$follow(E') = follow(E) = \{ \$,) \}$
 Rule 2 on $F \rightarrow (E) / id$ & Rule 3

$follow(T) = first(E') = \{ +, \$,) \}$
 except ϵ - Rule 2 on $E \rightarrow TE'$

$follow(T') = follow(T) = \{ +, \$,) \}$ - Rule 3

$follow(F) = first(T') = \{ *, +, \$,) \}$
 except ϵ - Rule 3

$follow(E) = first() = \{), \$ \}$

$follow(E') =$

Predictive parser

$$\begin{aligned} 2. S &\rightarrow (L)/a \\ L &\rightarrow L, S/S \rightarrow L \rightarrow \cancel{S} L' \\ L' &\rightarrow , S L' / \epsilon \end{aligned}$$

$$\text{first}(S) = \{ (, a \}$$

$$\text{first}(L) = \text{first}(S) = \{ (, a \}$$

$$\text{first}(L') = \{ , , \epsilon \}$$

$$\text{follow}(S) = \{ \$,), \}$$

$$\text{follow}(L) = \{) \}$$

$$\text{follow}(L') = \{) \}$$

$$3. E \rightarrow E + T / T$$

$$T \rightarrow T F / F$$

$$F \rightarrow F * | a | b$$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' / \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow F T' / \epsilon$$

$$F \rightarrow a F' / b F'$$

$$F' \rightarrow * F' / \epsilon$$

$$\text{first}(E) = \text{first}(T) = \text{first}(F) = \{a, b\}$$

$$\text{first}(E') = \{+, \epsilon\}$$

$$\text{first}(T') = \{\epsilon\} = \{a, b, \epsilon\}$$

$$\text{first}(F') = \{*, \epsilon\}$$

$$\text{follow}(E) = \{\$ \}$$

$$T \rightarrow \epsilon F F'$$

$$\text{follow}(E') = \{\$ \}$$

$$\text{follow}(T) = \{+, \$ \}$$

$$\text{follow}(T') = \{+, \$ \}$$

$$F' \rightarrow * F' \epsilon / \epsilon$$

$$\text{follow}(F') = \{\epsilon, a, b, +, \$ \}$$

$$\text{follow}(F) = \{a, b, +, \$ \}$$

$$S \rightarrow a | \uparrow | (\uparrow)$$

$$T \rightarrow T, S | S$$

$$T \rightarrow ST |$$

$$T' \rightarrow , ST' | \epsilon$$

$$\text{first}(S) = \{a, \uparrow, (\uparrow)\}$$

$$\text{first}(T) = \text{first}(S) = \{a, \uparrow, (\uparrow)\}$$

$$\text{first}(T') = \{, , \epsilon\}$$

$$\text{follow}(S) = \{ \$, , ,) \}$$

$$\text{follow}(T) = \{) \}$$

$$\text{follow}(T') = \{) \}$$

to go

Left factoring^{ing}:-

Left factoring is the another useful transformation to make certain grammar for parsing

If the non-terminal has two choices for expansion which are mostly same then there will be a confusion for selecting the choice for a certain input string.

Left factoring is the transformation for factoring out the common prefixes

$$\text{Ex} \rightarrow A \rightarrow XB/XC$$

After left factoring

$$A \rightarrow XA'$$

$$A' \rightarrow B/C$$

1. Eliminate the ambiguity for the following grammars

$$S \rightarrow iEtSeS / iEtS/a$$

$$E \rightarrow b|c|d$$

$$S \rightarrow iEtSS'/a$$

$$S' \rightarrow eS/e$$

$$E \rightarrow b|c|d$$

2. Eliminate the ambiguity for the following grammars

$$bexpr \rightarrow bexpr \text{ or } bterm / bterm$$

$$bterm \rightarrow bterm \text{ and } bfactor / bfactor$$

$$bfactor \rightarrow \text{not } bfactor / (bexpr) / \text{true} / \text{false}$$

where or, and, not, (,), true, false are

the terminal symbols in above grammars

$$bexpr \rightarrow bterm bexpr'$$

$$bexpr' \rightarrow \text{or } bterm bexpr' / \epsilon$$

$bteem \rightarrow bfactor\ bteem'$

$bteem' \rightarrow \text{and}\ bfactor\ bteem' \mid \epsilon$

$bfactor \rightarrow \text{not}\ bfactor \mid (bexpr) \mid \text{true} \mid \text{false}$

3. Write the context-free grammar for following language

(i) $L_1 = \{a^* b^*\}$

(ii) $L_2 = \{a^n b^n\}$ where $n > 0$

i) $S \rightarrow aS \mid \overset{bA}{\cancel{sb}} \mid \epsilon \mid a/b$

$A \rightarrow bA \mid b$

ii) $S \rightarrow asb$

$S \rightarrow ab \mid \epsilon$

Derivation:-

A non-terminal can be expanded to derive certain tokens called as derivation.

Consider the following grammar

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow id$$

The derivation of the grammar to the input string $id + id * id$ is given follows

$$E \rightarrow E + E \text{ by } E \rightarrow E + E$$

$$E \rightarrow E + E * E \text{ by } E \rightarrow E * E$$

$$E \rightarrow E + E * id \text{ by } E \rightarrow id$$

$$E \rightarrow E + id * id \text{ by } E \rightarrow id$$

$$E \rightarrow id + id * id \text{ by } E \rightarrow id$$

Reduction:-

The terminals does not derive any string but it can be reduced to any non-terminal called as reduction

The input string $id + id * id$ can be reduced to the starting non-terminal as follows

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

$$E \rightarrow id + id * E \text{ by } E \rightarrow id$$

$$E \rightarrow id + E * E \text{ by } E \rightarrow id$$

$$E \rightarrow E + E * E \text{ by } E \rightarrow id$$

$$E \rightarrow E + E$$

Parse tree:-

It is the graphical representation for the derivation that filters out the choice regarding the replacement order.

It pictorially shows how the starting symbol of the grammar derives the string in the language

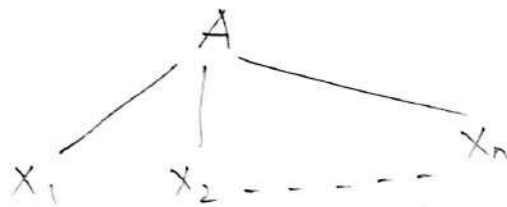
If the non-terminal A has the production $A \rightarrow XYZ$, then A is labelled as interior node with 3 children X, Y & Z .



Given the grammar, parse tree is the tree with the following properties

1. Root nodes are labelled by the starting symbol
2. The leaf nodes are labelled by tokens or empty string
3. Each interior node is labelled by the non-terminal

If A is the non-terminal labelled a
 interior node, $X_1, X_2, \dots, X_n$ are labels of
 children of that node from left to right
 then $A \rightarrow X_1/X_2/\dots/X_n$ is the production
 where $X_1, X_2, \dots, X_n$ are the Grammars
 symbols

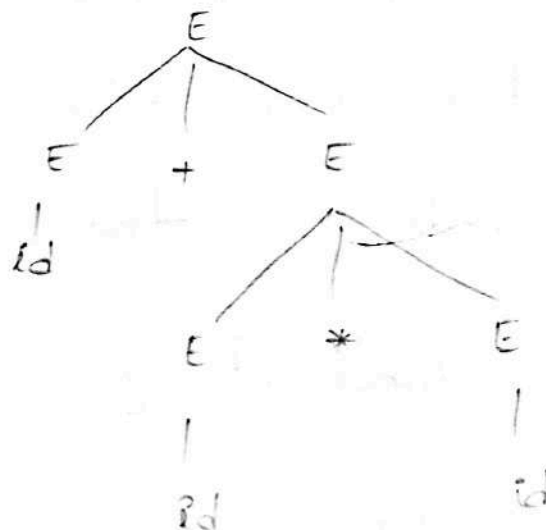


The parse tree for the below grammar
 with input string $id + id * id$ is as

follows $E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$



Parsing :-

The process of constructing the parse tree for the given string tokens is called as parsing.

Parser is the program which receives the valid tokens from the lexical analyzer & generates the parse tree

Two broad categories of parsing are

1. Top down parsing

2. Bottom-up parsing

Top down parsing :-

④

The process of constructing the parse tree from the root node to the leaf node in the pre order to the given input string is called as top down parsing

Here the starting non-terminal is expanded to derive the given input string

Top down parsing is classified as

1. Brute force method of parsing
(Backtracking Parser)
2. Recursive descent parsing
3. Non-recursive descent parsing
(predictive parser)

Bottom up parsing:-

The process of constructing the parse tree from the leaf node to the root node for the given input string is called as bottom up parsing

Here the given input string is reduced to the starting non-terminal

Bottom up parsing is classified

as

1. LR parsing
2. Shift reduce parsing
3. Operator precedence parsing

Brute force method of parsing :-

It is a kind of top parsing with full back up

This method operates as follows

1. Given the starting non-terminal i.e. to be expanded with the first production for that non-terminal
2. With the newly expanded string the next non-terminal is selected for expansion & its first production is applied.

3. The process of applying the production is repeated for all the subsequent non-terminals that are selected until the process cannot be continued.

4. The termination may be due to two cases

case 1:- If no more non-terminal may be present in which the string is successfully parsed

case 2:- It may result in an incorrect expansion, It should be indicated by the productions of terminals which does not match with the input string.

5. Instead of using the particular expansion that causes error, the next term of production is used for next expansion & the process continues.

6. The error causing production is replaced by the non-terminal & the process is back up again by undoing next most recently applied production.

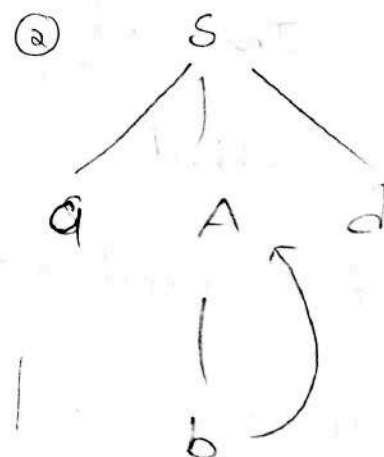
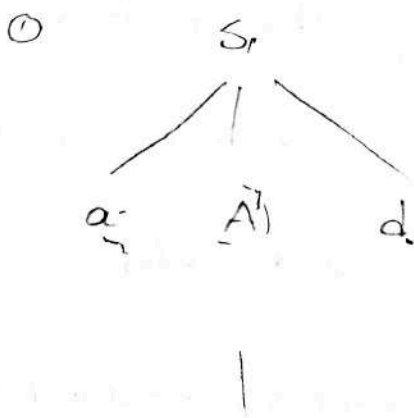
7. Backing up continues until we are able to resume the normal application to the selected non-terminal or until we back up upto the start symbol.

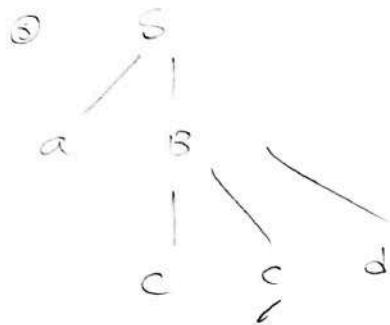
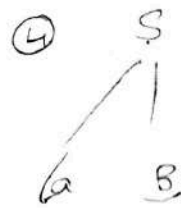
Ex:- $S \rightarrow aAd / aB$

$A \rightarrow b/c$

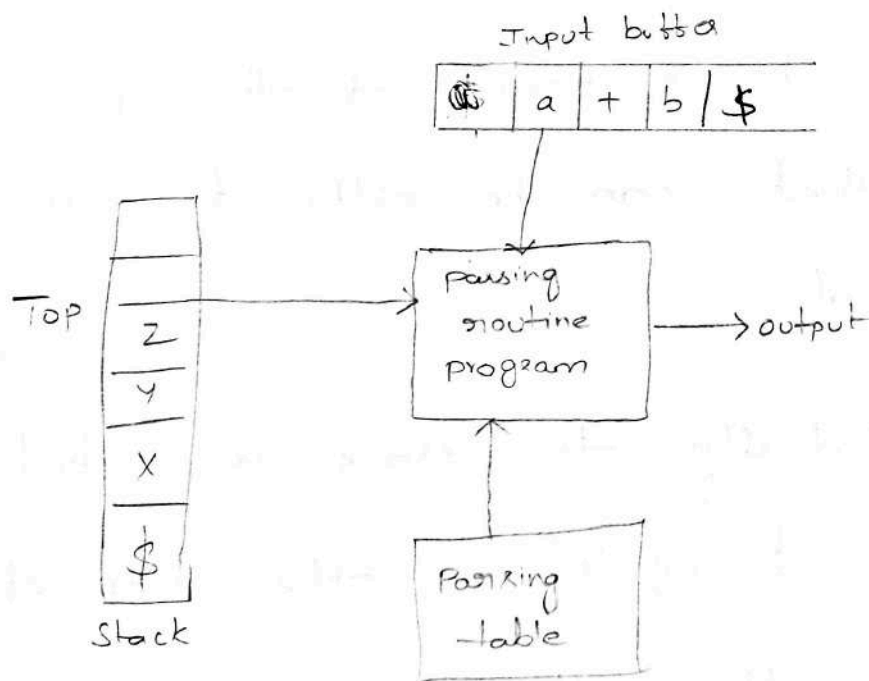
$B \rightarrow ccd/ddc$

input string: accd





Non-recursive descent parser:—



It is also called as predictive parsing.

The following components are needed to check the given input string is successfully parsed or not. They are

1. Stack
2. Input buffer
3. Parsing table
4. Parsing routine program

Stack:

It keeps track of the grammar symbols that can be either terminal or non-terminal

Initially the stack is pushed with the \$ symbol on the top of the stack then the parsing progresses the grammar symbols are pushed on to the top of the stack.

→ The \$ symbol is used to announce the completion of parsing

Input buffer:-

It keeps track of the input string i.e. to be parsed

→ The input string is followed by the \$ symbol & it is used to indicate the input string termination

→ The \$ symbol is used as right end marker

Parsing table:-

It is a two dimensional array.
→ the entry in the table is referred as $T[A, a]$ where T is the tablename
 A is the non-terminal & a is the terminal

Passing routine program:-

It works in conjunction with the passing table that reads the input string from the input buffer & considers the grammar symbol on the top of the stack to announce the output as successfully parsed or not.

*** Construct the predictive parsing table for the following grammar.

(2)

$$E \rightarrow E+T/T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / id$$

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) / id$$

$$\text{first}(E) = \text{first}(T) = \text{first}(F) = \{ (, id \}$$

$$\text{first}(E') = \{ +, \epsilon \}$$

$$\text{first}(T') = \{ *, \epsilon \}$$

$$\text{follow}(E) = \{ \$,) \}$$

$$\text{follow}(E') = \{ \$,) \}$$

$$\text{follow}(T) = \{ +, \$,) \}$$

$$\text{follow}(F) = \{ *, +, \$,) \}$$

$$\text{follow}(T') = \{ +, \$,) \}$$

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

Stack implementation for the input string $id + id * id$

Stack (S)	Input buffer (a)	Action
\$E	\$ $id + id * id$ \$ ↑ ↑	$E \rightarrow TE'$
\$E'T	\$ $id + id * id$ \$ ↑ ↑	$T \leftarrow FT$
\$E'T'F	\$ $id + id * id$ \$ ↑	$F \leftarrow id$
\$E'T'id	\$ $id + id * id$ \$ ↑	$T' \rightarrow E$ pop the context
\$E'	\$ $+ id * id$	$E' \rightarrow +TE'$
\$E'T+	\$ $+ id * id$	$T \rightarrow FT$
\$E'T'F	\$ $id * id$	$F \rightarrow id$
\$E'T'id	\$ $id * id$	$T' \rightarrow *F'$
\$E'T'F*	\$ $* id$	$F \rightarrow id$
\$E'T'id	\$ id	$T' \rightarrow E$
\$	\$	$E' \rightarrow E$

Successfully
parsed

2. $S \rightarrow (L)/a$ input string (a,a)

$L \rightarrow L, S/s$

$S \rightarrow (L)/a$

$L \rightarrow SL'$

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

$\text{first}(S) = \{ (, a \}$

$\text{first}(L) = \text{first}(S) = \{ (, a \}$

$\text{first}(L') = \{ \epsilon, \}$

$\text{follow}(S) = \{ \$, \epsilon \}$

$\text{follow}(L) = \{ \epsilon \}$

$\text{follow}(L') = \{ \epsilon \}$

	(	)	,	a	\$
S	$S \rightarrow (L)$			$S \rightarrow a$	
L	$L \xrightarrow{SL'} L, S/s$			$L \xrightarrow{SL'} L, S/s$	
L'	$L' \rightarrow \epsilon$	$L' \rightarrow \epsilon$	$L' \rightarrow , SL$		

Stack	Input buffer	Action
\$ S	\$ (a, a)	$S \rightarrow (L)$ ←
\$) L (	\$ (a, a)	$L \rightarrow SL'$
\$) L' S	\$ a, a)	$S \rightarrow a$
\$) L' a	\$ a, a)	$L' \rightarrow) SL'$
\$) L' S)	\$ a)	$S \rightarrow a$
\$) L' a	\$ a)	$L' \rightarrow \epsilon$
\$)	\$)	
\$	\$ successfully parsed	

2. $S \rightarrow aAB/bA/\epsilon$

$A \rightarrow aAb/\epsilon$

$B \rightarrow bB/c$

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

$\text{first}(A) = \{a, \epsilon\}$

$\text{first}(B) = \{b, c\}$

$$\text{follow}(S) = \{\$ \}$$

$$\text{follow}(A) = \{b, c, \$ \}$$

$$B \rightarrow bB \epsilon / c$$

$$\text{follow}(B) = \{\$ \} \Rightarrow S \rightarrow \underbrace{aA}_{x} \underbrace{B}_{y} \quad | \quad y$$

	a	b	c	\$
S	$S \rightarrow aAB$	$S \rightarrow bA$		$S \rightarrow \epsilon$
A	$A \rightarrow aAb$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
B		$B \rightarrow bB$	$B \rightarrow c$	

1. Consider the following grammar

5

$$S \rightarrow 0X / 1B / 0 / 1$$

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

$$B \rightarrow 0X / 1S$$

construct the left most derivation & parse tree for the following input string

i) 0101

ii) 1100101

Algorithm for constructing productive

①. parsing table:-

Input: Grammar G

Output: Parsing table T

Method:

step-1: for each production $A \rightarrow \alpha$ of the grammar do step-2 & step-3

step-2: for each terminal a in $\text{first}(\alpha)$
add $A \rightarrow \alpha$ to $T[A, a]$

step-3:- If ϵ is in $\text{first}(\alpha)$ add $A \rightarrow \alpha$ to $T[A, \epsilon]$ for each terminal b in $\text{follow}(A)$

ex:- If ϵ is in $\text{first}(\alpha)$ & $\$$ is in $\text{follow}(A)$ then add $A \rightarrow \alpha$ to $T[A, \$]$

step-4:- Mark each undefined entries of the parsing table T as error.

LL(1) grammar :-

The grammar whose parsing table has no multiple defined entries is said to be LL(1) grammar.

first L stands for left to right scanning of input, second L stands for left most derivation & 1 stands for number of input symbols that is to be parsed.

Properties:-

1. No ambiguous grammar can be an LL(1) grammar.
2. No left recursive grammar can be an LL grammar.
3. No left factored grammar can be an LL(1) grammar.
4. All the entries in the parsing table should be unique.

5. The grammars can be LL(1) grammars whenever $A \rightarrow \alpha/\beta$ are the two productions then the following conditions should hold

i) $\text{first}(\alpha) = \text{first}(\beta) \neq \{a\}$ where a is the terminal symbol of the grammar $\alpha \neq \beta$ should not begin with the same terminal symbol.

ii) $\text{first}(\alpha) = \text{first}(\beta) \neq \{\epsilon\}$ i.e. at most one of the α & β can derive the empty string.

iii) If $\text{first}(\alpha) = \{\epsilon\}$ then $\text{first}(\beta)$ should not derive any string beginning with the terminal as in follow (A)

Similar situation holds if

$$\text{first}(\beta) = \{\epsilon\}$$

Consider

$$i) S \rightarrow AS|b$$

$$A \rightarrow SA|a$$

Construct LL(1) parsing table

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

	a	b	\$
S	$S \rightarrow AS$ $S \rightarrow b$	$S \rightarrow b$	
A	$A \rightarrow a$	$A \rightarrow SA$ $A \rightarrow a$	

The given grammar is not an LL(1) grammar as it contains multiple productions in row S & A

$$2) E \rightarrow E+T|T$$

$$T \rightarrow TF|F$$

$$F \rightarrow F*|a|b$$

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

$$T' \rightarrow FT'| \epsilon$$

$$F \rightarrow aF'|bF'$$

$$F' \rightarrow *F'| \epsilon$$

$$\text{first}(E) = \text{first}(T) - \text{first}(F) = \{a, b\}$$

$$\text{first}(E') = \{+, \epsilon\}$$

$$\text{first}(T') = \{\epsilon, a, b\}$$

$$\text{first}(F') = \{*, \epsilon\}$$

$$\text{follow}(E) = \{\$ \}$$

$$\text{follow}(E') = \{\$, +\}$$

$$\text{follow}(T) = \{+\}$$

$$\text{follow}(T') = \{+\}$$

$$T' \rightarrow \epsilon FT' | \epsilon$$

$$\text{follow}(F) = \{+\}$$

$$\text{follow}(F') = \{+\} \quad F' \rightarrow * F' \epsilon | \epsilon$$

	+	*	a	b	ϵ
E			$E \rightarrow TE'$	$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$				$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$	$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$		$T' \rightarrow FT'$ $T' \rightarrow FT'$	$T' \rightarrow FT'$ $T' \rightarrow FT'$	
F			$F \rightarrow aE'$	$F \rightarrow bE'$	
F'	$F' \rightarrow \epsilon$	$F' \rightarrow *F'$			

The given grammar is not an LL(1) grammar as it contains only multiple production

Recursive descend parsing:-

It is a kind of top down parsing which is divided into two types

They are

1. Recursive descend parsing with backtracking

2. Recursive descend parsing without backtracking

Recursive descend parsing which does not involve backtracking is called as predictive parsing

The general form of recursive descendant parsing that involves backtracking makes use of repeated scan of the input string.

Generally backtracking parsers are not used frequently.

Recursive descent parser with backtracking works as follows

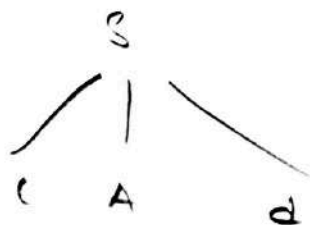
Consider the input string cad with the following grammar

$$S \rightarrow cAd$$

$$A \rightarrow ab|a$$

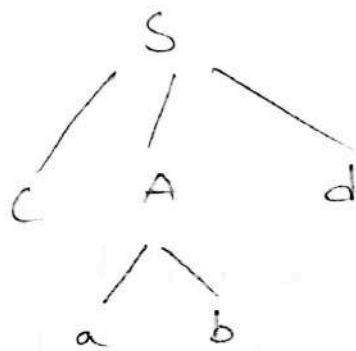
Initially we create a parser, consisting of a single node labelled

An input pointer points to the first input string c then we use the first production for S to expand the tree as



The left most leaf labelled c matches the first input string so, advance the input pointer to a .

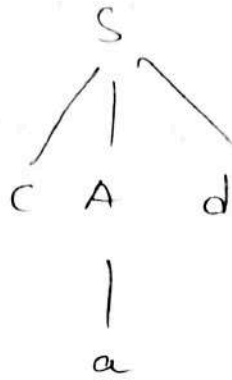
Consider the next leaf labelled A to expand the first production for A to obtain the tree as



We now match for the second input string, so advance the input pointer to d . Compare d against the next leaf labelled b

Since b does not match we report as failure & go back to A to see whether there is second alternative for A that we have not expanded

: In going back to A, we try to
reset the input pointer and
the second alternative for A
obtain the tree as



The leaf a matches the second
input string and the leaf d matches
the third input symbol

Now we have obtained the
parse tree which is the successful
completion of a parsing

write the procedure for the non-terminals of the following grammar to make recursive descent parsing.

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / id$$

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) / id$$

Procedure $E()$

{

$T()$;

$Eprime$;

}

Procedure $Eprime()$

{

 if input string = '+'

 { advance();

$T()$;

$Eprime$;

 return true

```
else  
{  
    return false  
}
```

```
}
```

Procedure T()

```
{  
    F();  
    Tprime();  
}
```

Procedure Tprime()

```
{  
    if input string = '*'  
    {  
        advance();  
        F();  
        Tprime;  
        return true;  
    }  
    else  
    {  
        return false  
    }  
}
```

procedure F()

{

if inputstring = '('

{

advance();

E();

if inputstring = ')'

{

advance();

return true;

}

else

{

return false;

}

return true;

}

else if inputstring = 'id'

{

advance();

return true;

}

else

{

return false;

}

$$2. E \rightarrow E + T / T$$

$$(1) T \rightarrow TF / F$$

$$F \rightarrow F * a / b$$

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow aF' / bF'$$

$$F' \rightarrow *F' / \epsilon$$

Procedure E()

{

T();

Eprime;

}

procedure Eprime()

{

if input string = '+'

{

advance();

T();

Eprime();

return true;

}

else

{

return false;

}

}

procedure T()

{

F();

Tprime();

}

procedure Tprime()

{

F();

Tprime();

}

procedure F()

{

if inputstring = 'a'

{

advance();

Fprime();

return true;

}

else if inputstring = 'b'

{

advance();

Fprime();

return true;

}

else

{

return false;

}

procedure Fprime ()

{

if input string = '*'

{

advance ();

Fprime ();

return true;

}

else

{

return false;

}

}

Consider the following grammar

$$S \rightarrow (L)a$$

$$L \rightarrow L, S \mid S$$

Construct the leftmost derivation

Parse tree for the following i) $(a, (a, a))$

ii) $(a, ((a, a), (a, a)))$

1

$$S \rightarrow (L)$$

$$S \rightarrow (L, S)$$

$$S \rightarrow (S, S)$$

$$S \rightarrow (a, S)$$

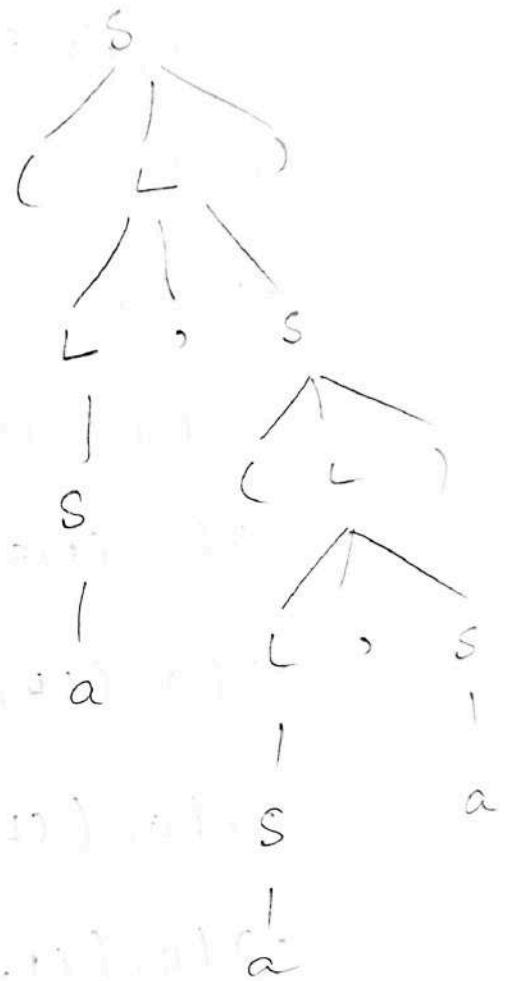
$$S \rightarrow (a, (L))$$

$$S \rightarrow (a, (L, S))$$

$$S \rightarrow (a, (S, S))$$

$$S \rightarrow (a, (a, S))$$

$$S \rightarrow (a, (a, a))$$



$$(a, ((a, a), (a, a))) \quad S \rightarrow (L) / a$$

$$L \rightarrow L, S / S$$

$$S \rightarrow (L)$$

$$\Rightarrow (L, S)$$

$$\Rightarrow (S, S)$$

$$\Rightarrow \cancel{(L, S)}$$

$$\Rightarrow /$$

$$\Rightarrow (a, S)$$

$$\Rightarrow (a, (L))$$

$$\Rightarrow (a, (L, S))$$

$$\Rightarrow (a, (S, S))$$

$$\Rightarrow (a, ((L), S))$$

$$\Rightarrow (a, ((L, S), S))$$

$$\Rightarrow (a, ((S, S), S))$$

$$\Rightarrow (a, ((a, S), S))$$

$$\Rightarrow (a, ((a, a), (L)))$$

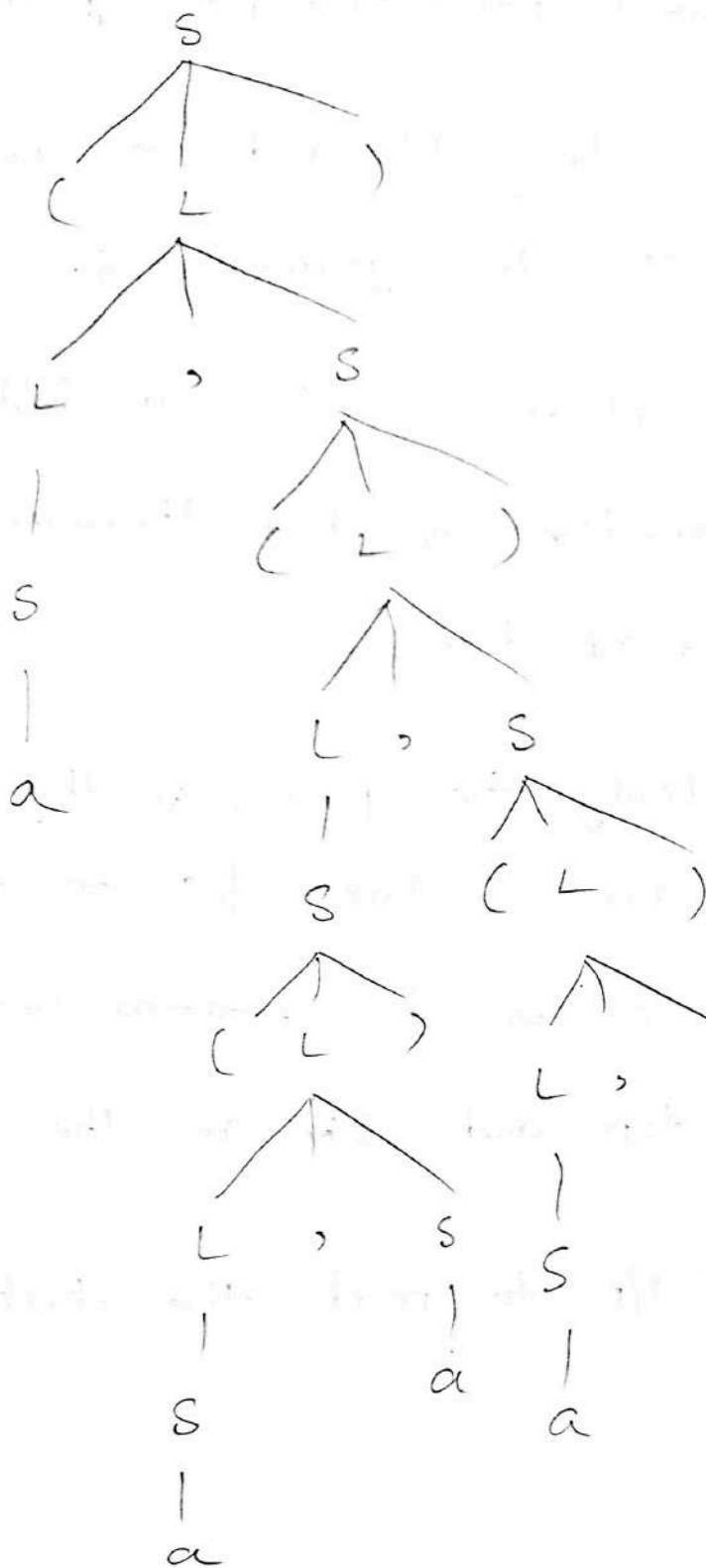
$$\Rightarrow (a, ((a, a), (L, S)))$$

$$\Rightarrow (a, ((a, a), (S, S)))$$

$$\Rightarrow (a, ((a, a), (a, a)))$$

$$\Rightarrow (a, ((a, a), (a, a)))$$

Pause here



Non-Recursive predictive parsing algorithm:-

Input:- A string W and a parsing table T for grammar G .

Output:- If W is in $L(G)$, a left most derivation of W , otherwise an error indication.

Method:- Initially the parser is the configuration in which it has $\$S$ on the stack with S as the starting symbol of G on top and $\$W$ in the i/p buffer. Set i/p to point the first symbol of $\$W$.

repeat

let X be the top of stack and ' a ' be the symbol pointer by i/p

if X is a terminal or $\$$ then

if $X = 'a'$ then

Pop X from the stack and
advance the i/p ~~else~~

else error

else

if $T[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ then

Pop X from the stack

Push $Y_k Y_{k-1} \dots Y_1$ on to stack

with Y_1 on top

output the production $X \rightarrow Y_1 Y_2 \dots Y_k$

else error

until $X = \$$

Bottomup parsing:-

The process of constructing the parse tree from leaf node to the root node is called as bottomup parsing.

Here the terminals are reduced to the respective non-terminal to obtain the starting symbol.

If we reduce the input string to the starting symbol then the parsing is successful otherwise it is unsuccessful.

Types of bottomup parsing:-

1. Shift reduce parser
2. LR parser
3. Operator precedence parser

Handle:-

The handle of the string is a substring that matches the right hand side of the production and whose reduction to the NT on left side helps in construction of parse tree.

Ex:- $S \rightarrow aAb$

$A \rightarrow a$

Here a is the handle because it reduces to A & helps in construction of the parse tree.

Handle pruning:-

The process of obtaining the starting non-terminal by constructing bottom up parsing in which the handle is reduced to respective non-terminal is called as handle pruning.

1. Shift reduce parsing^{er} :-

It is a general form of bottom up parsing which attempts to construct the parse tree from leaf node to the root node.

Various components required in shift reduce parser are

1. Input buffer :-

It keeps track of the input symbol & \$ symbol is used as right end marker

2. Stack :-

It is used to hold the grammar symbols & initially \$ symbol is pushed on to the stack

3. The primary actions performed by the shift reduce parser are

a. Shift :-

It shifts the next input symbol on to the stack when there is no

handle for reduction.

b. Reduce :-

It reduces the string to the non-terminal ϵ that non-terminal is pushed on the stack by reducing the handle.

c. Accept :-

If the input string is valid ϵ the parsing is successfully done then the parsing action accept will be performed.

d. Error :-

This is the syntax error ϵ the parser calls for the error recovery strategy.

Write the stack implementation of the shift reduce parser for the following grammar with input string $icdd$

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

stack	input buffer	Action
\$	cdcd\$	shift
\$c	dcd\$	shift
\$cd	cd\$	reduce $C \rightarrow d$
\$cC	cd\$	reduce $C \rightarrow cC$
\$C	cd\$	shift
\$Cc	d\$	shift
\$Ccd	\$	reduce $C \rightarrow d$
\$CcC	\$	reduce $C \rightarrow cC$
\$CC	\$	reduce $S \rightarrow CC$
\$S	\$	Accept

stack	input buffer	Action
\$	ccdd\$	shift
\$c	cdd\$	shift
\$cc	dd\$	shift
\$ccd	d\$	reduce $C \rightarrow d$
\$ccc	d\$	reduce $C \rightarrow cC$
\$cC	d\$	reduce $C \rightarrow cC$
\$C	d\$	reduce shift
\$Cd	\$	reduce $C \rightarrow d$
\$CC	\$	reduce $S \rightarrow CC$
\$S	\$	Accept

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

input string: $(id + id * id)$

stack	input buffer	Action
\$	$(id + id * id) \$$	shift
$\$ ($	$id + id * id) \$$	shift
$\$ (id$	$+ id * id) \$$	reduce $E \rightarrow id$
$\$ (E$	$+ id * id) \$$	shift
$\$ (E +$	$id * id) \$$	shift
$\$ (E + id$	$* id) \$$	shift reduce $E \rightarrow id$
$\$ (E + E$	$* id) \$$	reduce $E \rightarrow E + E$
$\$ (E$	$* id) \$$	shift
$\$ (E *$	$id) \$$	shift
$\$ (E * id$	$) \$$	reduce $E \rightarrow id$
$\$ (E * E$	$) \$$	reduce $E \rightarrow E * E$
$\$ (E$	$) \$$	shift
$\$ (E)$	$\$$	reduce $E \rightarrow (E)$
$\$ E$	$\$$	Accept

Conflicts during shift reduce parser:-

When shift reduce parser is applied to the context free grammar it leads to two conflicts. They are

1. Shift reduce conflict
2. Reduce reduce conflict

Shift reduce conflict:-

The parser after knowing moving the entire stack content & next input symbol, it cannot be able to decide whether to do shift or reduce action. This is called as shift reduce conflict.

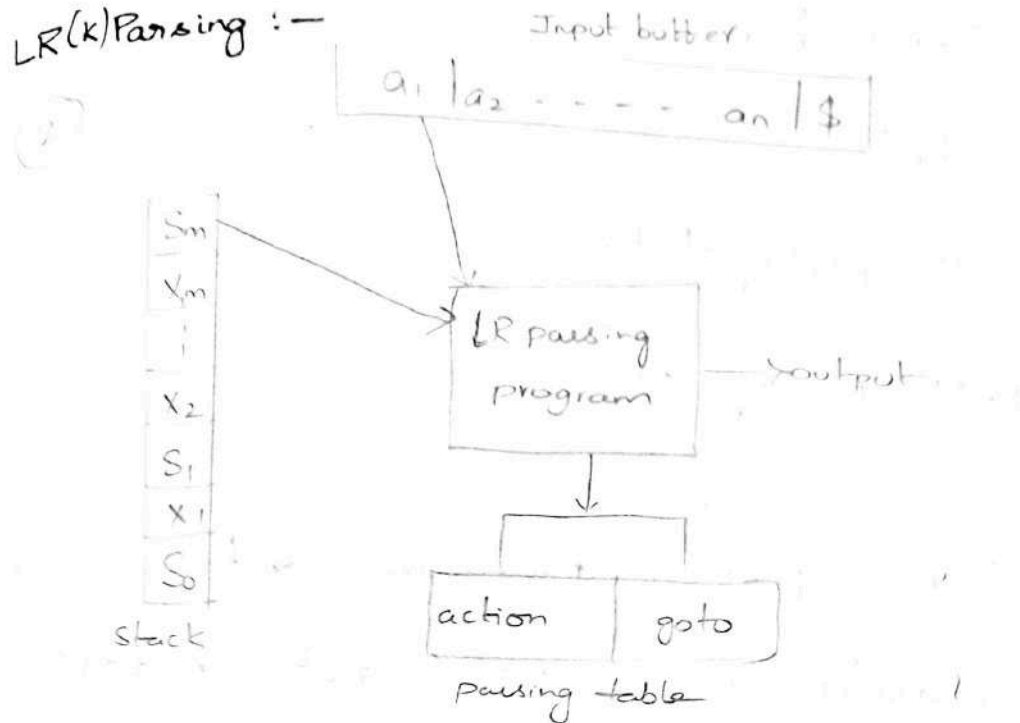
reduce reduce conflict:-

The parser after knowing moving entire stack content & the next

A → a
B → a

input symbol, it cannot be able to decide which production to apply is called as reduce reduce conflict

LR(k) Parsing :-



It is a kind of bottom up parsing which is used to parse large class of context free grammar

LR(k) is the parsing technique where L stands for left to right scanning of input, R stands for right most derivation & k stands for number of input symbols.

If k is not specified
it is assumed to be 1

LR parsing method consist of

1. Input buffer
2. Stack
3. LR parsing table
4. LR parsing program

The parsing program action reads the character from the input buffer one at a time

The program uses the stack to store the string of the form $S_0 X_1 S_1 X_2 \dots X_m S_m$ where S_m is on the top. Each X_i is a grammar symbol and each S_i is a symbol called as state.

The combination of state symbol on the top of the stack & the current input symbol are used to index the parsing table & determine the shift reduce parsing action.

The parsing table consist of two parts, a parsing action function called action and goto function called goto

Types:-

The three types of LR parsing are

1. Simple LR parsing
2. Canonical LR parsing
3. Look ahead LR parsing

Simple LR:-

The steps to be followed for constructing simple LR parsing table are

1. find the augmented grammars
2. find the LR(0) items
3. Apply the closure operation
4. Apply the goto function.

1. Augmented grammar:-

If G is the grammar with starting symbol S then the augmented grammar will be of the form $S' \rightarrow S$

2. LR(0) items:-

It consist of the production of the form $S' \rightarrow \cdot S$ & let be at the state I

3. closure operation:-

If I is the set of items for the grammar G then $\text{closure}(I)$ will be constructed by the following rules

Rule 1:- Initially, every items in the state I are added to the $\text{closure}(I)$

Rule 2:- If $A \rightarrow X \cdot B Y$ is the production in $\text{closure}(I)$ & $B \rightarrow Z$ is the production given in the Grammar G then add the item $B \rightarrow \cdot Z$ to $\text{closure}(I)$ if it is not already exist.

Rule 3:- Repeat the rule 2 until no more new items can be added to $\text{closure}(I)$.

4. Goto function :-

It is denoted by $\text{goto}(I, X)$

where I is the set of items

& X is the grammar symbol.

$\text{Goto}(I, B)$ is defined to be all the set of items $A \rightarrow X B \cdot Y$ such that $A \rightarrow X \cdot B Y$ is in I

Construct SLR parsing table for the following grammar

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / id$$

Step-1: Augmented grammar.

$$E' \rightarrow E \rightarrow 0$$

$$E \rightarrow E + T \rightarrow 1$$

$$E \rightarrow T \rightarrow 2$$

$$T \rightarrow T * F \rightarrow 3$$

$$T \rightarrow F \rightarrow 4$$

$$F \rightarrow (E) \rightarrow 5$$

$$F \rightarrow id \rightarrow 6$$

Step-2 :- LR(0) item

$$LR(0) = \{ E' \rightarrow \cdot E \} \rightarrow \textcircled{I}$$

step-3 :- closure operation

$$\text{closure}(I) = \{ E' \rightarrow \cdot E \text{ — accept} \}$$

$$E \rightarrow \cdot E + T \text{ — } \gamma_1$$

$$E \rightarrow \cdot T \text{ — } \gamma_2$$

$$E \rightarrow \cdot T * F \text{ — } \gamma_3$$

$$T \rightarrow \cdot F \text{ — } \gamma_4$$

$$F \rightarrow \cdot (E) \text{ — } \gamma_5$$

$$F \rightarrow \cdot id \text{ — } \gamma_6$$

$$\gamma \rightarrow I_0$$

step-4 :- Goto function

$$\text{goto}(I_0, E) = \{ \underline{E' \rightarrow E \cdot} \}$$

$$E \rightarrow E \cdot + T$$

$$\gamma \rightarrow I_1$$

$$\text{goto}(I_0, T) = \{ \underline{E \rightarrow T \cdot} \}$$

$$T \rightarrow T \cdot * F$$

$$\gamma \rightarrow I_2$$

$$\text{goto}(I_0, F) = \{ \underline{T \rightarrow F \cdot} \} \rightarrow I_3$$

$$\text{goto}(I_0, () = \{ F \rightarrow (\cdot E) \} \rightarrow \text{next page}$$

$$E' \rightarrow \cdot E$$

$$\gamma \quad E \rightarrow \cdot E + T \rightarrow I_4$$

$$\text{goto}(I_0, \text{id}) = \{ (F \rightarrow \text{id}) \} \rightarrow I_5$$

$$\text{goto}(I_0, () = \{ F \rightarrow (.E), E \rightarrow .E + T,$$

$$E \rightarrow .T, T \rightarrow T * F, T \rightarrow .F,$$

$$F \rightarrow .(E), F \rightarrow .\text{id} \} \rightarrow I_4$$

$$\text{goto}(I_1, +) = \{ E \rightarrow E + .T, T \rightarrow .T * F,$$

$$T \rightarrow .F, F \rightarrow .(E), F \rightarrow .\text{id} \}$$

$$\rightarrow I_6$$

$$\text{goto}(I_2, *) = \{ T \rightarrow T * .F, F \rightarrow .(E),$$

$$F \rightarrow .\text{id} \} \rightarrow I_2$$

$$\text{goto}(I_4, E) = \{ F \rightarrow (E \cdot), E \rightarrow E \cdot + T \} \rightarrow I_8$$

$$\text{goto}(I_4, T) = \{ E \rightarrow T \cdot, T \rightarrow T \cdot * F \} \rightarrow I_2$$

$$\text{goto}(I_4, F) = \{ T \rightarrow F \cdot \} \rightarrow I_3$$

$$\text{goto}(I_4, () = \{ F \rightarrow (\cdot E), E \rightarrow \cdot E + T, \quad 17$$

$$E \rightarrow \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F,$$

$$F \rightarrow \cdot (E), F \rightarrow \cdot id \} \rightarrow \textcircled{I_4}$$

$$\text{goto}(I_4, id) = \{ F \rightarrow id \cdot \} \rightarrow \textcircled{I_5}$$

$$\text{goto}(I_6, T) = \{ E \rightarrow E + T \cdot, T \rightarrow T \cdot * F \} \rightarrow \textcircled{I_9}$$

$$\text{goto}(I_6, F) = \{ T \rightarrow F \cdot \} \rightarrow \textcircled{I_3}$$

$$\text{goto}(I_6, () = \{ F \rightarrow (\cdot E), E \rightarrow \cdot E + T, E \rightarrow \cdot T,$$

$$T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E),$$

$$F \rightarrow \cdot id \} \rightarrow \textcircled{I_4}$$

$$\text{goto}(I_6, id) = \{ F \rightarrow id \cdot \} \rightarrow \textcircled{I_5}$$

$$\text{goto}(I_7, F) = \{ T \rightarrow T * F \cdot \} \rightarrow \textcircled{I_{10}}$$

$$\text{goto}(I_7, () = \{ F \rightarrow (\cdot E), E \rightarrow \cdot E + T, E \rightarrow \cdot T,$$

$$T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E),$$

$$F \rightarrow \cdot id \} \rightarrow \textcircled{I_4}$$

$$\text{goto}(I_7, \text{id}) = \{ F \rightarrow \text{id} \cdot \} \rightarrow I_5$$

$$\text{goto}(I_8, ()) = \{ F \rightarrow (E) \cdot \} \rightarrow I_{11}$$

$$\begin{aligned} \text{goto}(I_8, +) = \{ & E \rightarrow E + \cdot T, T \rightarrow \cdot T * F, \\ & T \rightarrow \cdot F, F \rightarrow \cdot (E), \\ & F \rightarrow \cdot \text{id} \} \rightarrow I_6 \end{aligned}$$

$$\begin{aligned} \text{goto}(I_9, *) = \{ & T \rightarrow T * \cdot F, F \rightarrow \cdot (E), \\ & F \rightarrow \cdot \text{id} \} \rightarrow I_7 \end{aligned}$$

first & follow for $E \rightarrow E + T / t$

$$T \rightarrow T * F / f$$

$$F \rightarrow (E) / \text{id}$$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' / \epsilon$$

$$T \rightarrow F T'$$

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

$$F \rightarrow (E) / \text{id}$$

$$\text{first}(E) = \text{first}(T) = \text{first}(F) = \{ (, id \}$$

19

$$\text{first}(E') = \{ +, \epsilon \}$$

$$\text{first}(T') = \{ *, \epsilon \}$$

$$\text{follow}(E) = \{ \$,) \}$$

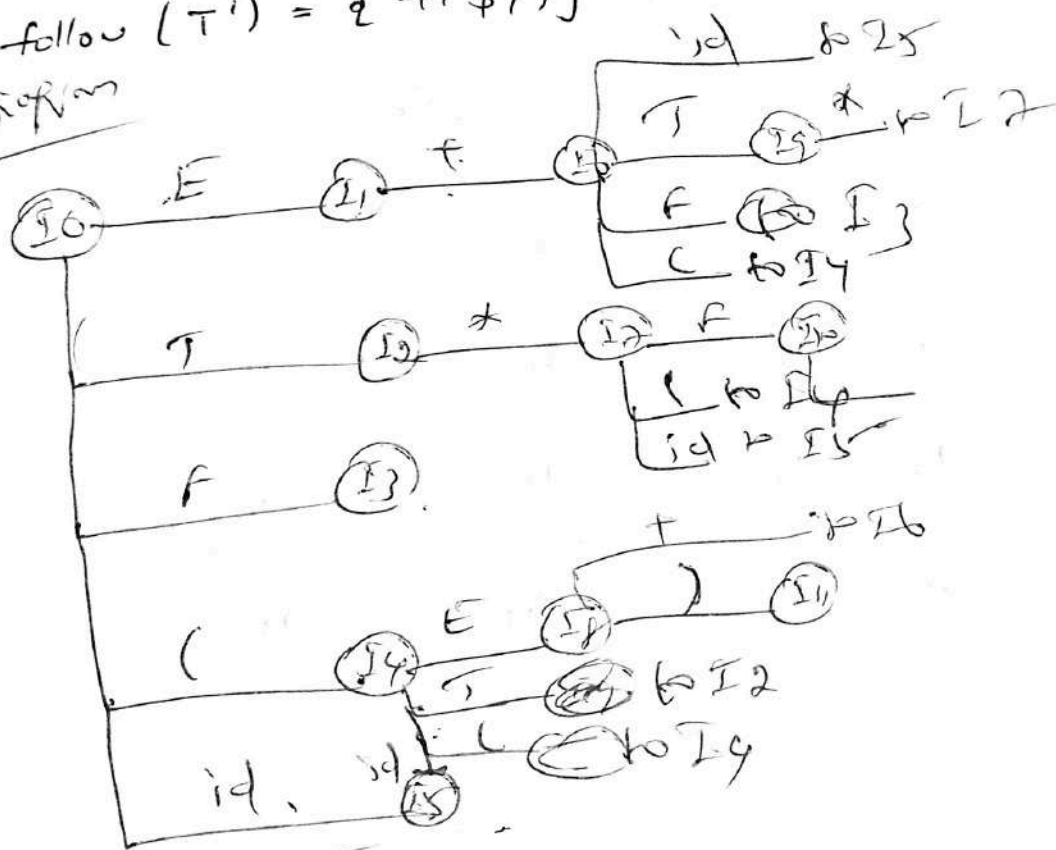
$$\text{follow}(E') = \{ \$,) \}$$

$$\text{follow}(T) = \{ +, \$,) \}$$

$$\text{follow}(F) = \{ *, +, \$,) \}$$

$$\text{follow}(T') = \{ +, \$,) \}$$

LR diagram



states	Action						Goto		
	+	*	(	)	id	\$	E	T	F
I_0			S_4		S_5		1	2	3
I_1	S_6					acc			
I_2	r_2	S_7		r_2		r_2			
I_3	r_4	r_4		r_4		r_4			
I_4			S_4		S_5		8	2	3
I_5	r_6	r_6		r_6		r_6			
I_6			S_4		S_5			9	3
I_7			S_4		S_5				10
I_8	S_6			S_{11}					
I_9	r_1	S_7		r_1		r_1			
I_{10}	r_3	r_3		r_3		r_3			
I_{11}	r_5	r_5		r_5		r_5			

✓

$$E \rightarrow E+T/T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / id$$

$$id + id * id$$

Stack implementation

Stack	I/P buffer	Output
\$0	id + id * id \$	shift
\$oid5	+ id * id \$	reduce $F \rightarrow id$
\$OF3	+ id * id \$	reduce $T \rightarrow F$
\$OT2	+ id * id \$	reduce $E \rightarrow T$
\$OE1	+ id * id \$	shift
\$OE1+6	id * id \$	shift
\$OE1+6id5	* id \$	reduce $F \rightarrow id$
\$OE1+6F3	+ id \$	reduce $T \rightarrow F$
\$OE1+6T9	* id \$	shift
\$OE1+6T9*7	id \$	shift
\$OE1+6T9*7id5	\$	reduce $F \rightarrow id$
\$OE1+6T9*7F10	\$	reduce $T \rightarrow T * F$

\$ OE1+6T9
\$ OE1

\$
\$

reduce $E \rightarrow E+T$ 27
accept

id * id + id

stack

I/p bobber

output

\$0

id * id + id \$

shift

\$0id5

* id + id \$

reduce $F \rightarrow id$

\$OE3

* id + id \$

reduce $T \rightarrow F$

\$OT2

* id + id \$

shift

\$OT2 * 7

id + id \$

shift

\$OT2 * 7 id 5

+ id \$

reduce $F \rightarrow id$

\$OT2 * 7 F 10

+ id \$

reduce $T \rightarrow T * F$

\$OT2

+ id \$

reduce $E \rightarrow T$

\$OE1

+ id \$

shift

\$OE1 + 6

id \$

shift

\$OE1 + 6 id 5

\$

reduce $F \rightarrow id$

\$OE1 + 6 F 3

\$

reduce $T \rightarrow F$

\$OE1 + 6 T 9

\$

reduce $E \rightarrow E+T$

\$OE1

\$

accepted

Procedure to fill the SLR parsing table:-

Shift:-

For each state get the items reachable from goto function on each symbol, the state to which it is reaching is the shift number.

Reduce:-

1. Compute first and follow for the given grammar.
2. Select the item in the goto function where $\cdot$ is present at the end of each production except for the first production in the augmented grammar.

29

3. Give the numbers for each production in the augmented grammar starting from 0

4. If the production is of the form $A \rightarrow \alpha$. then consider $\text{follow}(A)$ for reduce if α is terminal symbol
& $\text{follow}(\alpha)$ if α is non-terminal symbol.

Ex:- The state I_2 has the $\cdot$ at the end i.e. $E \rightarrow T \cdot$. The number for the production is 2 & $\text{follow}(T) = \{ +,), \$ \}$ then apply in the state I_2 for all the symbols in $\text{follow}(T)$ then reduce r_2

Accept:-

find the item which contains $\cdot$ symbol at the end for the starting production in the augmented

grammar then apply the accept action Acc on the corresponding state in $\$$ column.

Error: -

Blank spaces represent the error messages.

LR Parsing algorithm: -

Input: -

An input string $w \in a$

LR parsing table with the function action

$\in$ goto for the grammar G .

Output: -

If w belongs to the language

generated by the grammar G , a

bottom up parses will be obtained

otherwise an error will be indicated

Method: -

31

Initially the parser has S_0 on its stack where S_0 is the initial state & $w\$$ in the input buffer. The parser then executes the algorithm until an accept or error action is encountered.

Step-1: - If action $[S, a] = \text{shift } S'$ then push a and S' on the top of the stack. Advance the input pointer to the next input symbol.

Step-2: -

If action $[S, a] = \text{reduce } A \rightarrow B$ then pop $2 \times B$ symbols of the stack. Let S' be the state now on the top of the stack, push A the left

hand side of the production and
then push goto (S', A) on the top of
the stack

Step-3:- If action $[S, a] = \text{accept}$
then return else error.

Canonical LR Parsing :-

33

The steps for constructing CLR parsing table are

1. Find the augmented grammar
2. find the LR(1) item
3. Apply the closure operation
4. Apply the goto function

Augmented Grammar :-

If G is the grammar with starting symbol S then the augmented grammar for G is $S' \rightarrow S$

LR(1) item :-

(IV)

It consist of the production of the form $S' \rightarrow \cdot S$ or $\$$

Closure operation :-

If I is the set of items then the closure (I) is the set of items constructed by the following rules

Rule-1:- Initially every items in I is added to $\text{closure}(I)$

Rule-2:- If $A \rightarrow X \cdot BY$, a is in $\text{closure}(I)$,

$\& B \rightarrow Z$ is one of the given production in the grammar G then add $B \rightarrow \cdot Z, \text{first}(Ya)$

Note:-

1. $\text{first}(Ya) = \text{first}(a)$ if $\underline{Y = \epsilon}$
2. $\text{first}(Ya) = \text{first}(Y)$ when Y is Non-Terminal
 $\& a$ is \$ symbol
3. $\text{first}(Ya) = \text{first}(Y)$ and $\text{first}(a)$ when $Y \& a$ are terminal symbols

Goto function :-

It is denoted by $\text{goto}(I, x)$

where I is the set of items &

x is the grammar symbol

$\text{goto}(I, x)$ is defined to be the closure of all of the set of items

$A \rightarrow XB \cdot Y, \$$ such that $A \rightarrow XBY, \$$

is in I

Construct CLR parsing table for

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

Step-1 :- Augmented grammar

$$S' \rightarrow S \quad -0$$

$$S \rightarrow CC \quad -1$$

$$C \rightarrow cC \quad -2$$

$$C \rightarrow d \quad -3$$

Step-2 :- LR(1) item

$$LR(1) = \{ S' \rightarrow \cdot S, \$ \} \rightarrow I$$

lookahead

lookahead are useful in determining where we are going to place reduce moves (or final item) in step-3 :- closure(I)

$$\text{closure}(I) = \{ S' \rightarrow \cdot S, \$$$

$$S \rightarrow \cdot CC, \$$$

$$C \rightarrow \cdot cC, c/d$$

$$C \rightarrow \cdot d, c/d$$

$$\} \rightarrow I_0$$

Step-4 :- goto function

$$\text{goto}(I_0, S) = \{ S' \rightarrow S \cdot, \$ \} \rightarrow I_1$$

$$\text{goto}(I_0, C) = \{ S \rightarrow C \cdot C, \$, C \rightarrow \cdot cC, \$,$$

$$C \rightarrow \cdot d, \$ \} \rightarrow I_2$$

$$\text{goto}(I_0, c) = \{ C \rightarrow c \cdot C, c/d, C \rightarrow \cdot cC, c/d,$$

$$C \rightarrow \cdot d, c/d \} \rightarrow I_3$$

$$\text{goto}(I_0, d) = \{ C \rightarrow d \cdot, c/d \} \rightarrow I_4$$

$$\text{goto}(I_0, c) = \{ S \rightarrow (C \cdot, \$) \} \rightarrow I_5$$

$$\text{goto}(I_2, c) = \{ C \rightarrow c \cdot C, \$, C \rightarrow \cdot c C, \$, \\$$

$$C \rightarrow \cdot d, \$ \} \rightarrow I_6$$

$$\text{goto}(I_2, d) = \{ C \rightarrow d \cdot, \$ \} \rightarrow I_7$$

$$\text{goto}(I_3, c) = \{ C \rightarrow c C \cdot, c/d \} \rightarrow I_8$$

$$\text{goto}(I_3, c) = \{ C \rightarrow c \cdot C, c/d, C \rightarrow \cdot c C, c/d, \\$$

$$C \rightarrow \cdot d, c/d \} \rightarrow \textcircled{I_3}$$

$$\text{goto}(I_3, d) = \{ C \rightarrow d \cdot, c/d \} \rightarrow \textcircled{I_4}$$

$$\text{goto}(I_6, c) = \{ C \rightarrow c C \cdot, \$ \} \rightarrow I_9$$

$$\text{goto}(I_6, c) = \{ C \rightarrow c \cdot C, \$, C \rightarrow \cdot c C, \$, \\$$

$$C \rightarrow \cdot d, \$ \} \rightarrow \textcircled{I_6}$$

$$\text{goto}(I_6, d) = \{ C \rightarrow d \cdot, \$ \} \rightarrow \textcircled{I_7}$$

CLR Parsing table

states	Action			Goto	
	c	d	\$	S	C
I_0	S_3	S_4		1	2
I_1			acc		
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		

$SLR(0)$ - Entire row

$SLR(1)$ - Follow of left hand side.

$CLR(1)$ - only in lookahead entries

When reduce moves are decreased, the conflicts are decreased, and when white spaces are increased, error detecting capability is increased.

stack implementation for input string codd

37

stack	Input bubble	Output
\$0	cdd \$	shift
\$0c3	cd \$	shift
\$0c3c3	dd \$	shift
\$0c3c3d4	d \$	reduce $C \rightarrow d$
\$0c3c3C8	d \$	reduce $C \rightarrow cC$
\$0c3C8	d \$	reduce $C \rightarrow cC$
\$0C2	d \$	shift
\$0C2d7	\$	reduce $C \rightarrow d$
\$0C2C5	\$	reduce $S \rightarrow CC$
\$0S1	\$	accept

stack implementation for input string cdd

stack	I/p buffer	Output
\$0	cdd\$	shift
\$0c3	dcd\$	shift
\$0c3d4	cd\$	reduce $c \rightarrow$
\$0c3c8	cd\$	reduce $C \rightarrow cC$
\$0C2	cd\$	shift
\$0C2c6	d\$	shift
\$0C2c6d7	\$	reduce $C \rightarrow d$
\$0C2c6C9	\$	reduce $C \rightarrow cC$
\$0C2C5	\$	reduce
\$0S1	\$	$S \rightarrow cC$ accept

$$S \rightarrow L = R \mid R$$

$$L \rightarrow * \mid id$$

$$R \rightarrow L$$

Augmented Grammar

$$S' \rightarrow S \quad 0$$

$$S \rightarrow L = R \quad 1$$

$$S \rightarrow R \quad 2$$

$$L \rightarrow * \quad 3$$

$$L \rightarrow id \quad 4$$

$$R \rightarrow L \quad 5$$

LR(1) item

$$LR(1) = \{ S' \rightarrow \cdot S, \$ \} \rightarrow I$$

closure(I)

$$\text{closure}(I) = \{ S' \rightarrow \cdot S, \$ \}$$

$$S \rightarrow \cdot L = R, \$,$$

$$S \rightarrow \cdot R, \$$$

$$L \rightarrow \cdot *, \$ \mid =$$

$$L \rightarrow \cdot id, = \mid \$$$

$$R \rightarrow \cdot L, \$$$

$$\} \rightarrow I_0$$

Goto function:-

$$\text{goto}(I_0, S) = \{ S' \rightarrow S \cdot, \$ \} \rightarrow I_1$$

$$\text{goto}(I_0, L) = \{ S \rightarrow L \cdot = R, \$, R \rightarrow L \cdot, \$ \} \rightarrow I_2$$

$$\text{goto}(I_0, R) = \{ S \rightarrow R \cdot, \$ \} \rightarrow I_3$$

$$\text{goto}(I_0, *) = \{ L \rightarrow * \cdot, =, \$ \} \rightarrow I_4$$

$$\text{goto}(I_0, id) = \{ L \rightarrow id \cdot, =, \$ \} \rightarrow I_5$$

$$\text{goto}(I_2, =) = \{ S \rightarrow L = \cdot R, \$,$$

$$R \rightarrow \cdot L, \$ \} \rightarrow I_6$$

$$L \rightarrow \cdot id, L \rightarrow \cdot *, \$ \}$$

$$\text{goto}(I_6, R) = \{ S \rightarrow L = R \cdot, \$ \} \rightarrow I_7$$

$$\text{goto}(I_6, L) = \{ R \rightarrow L \cdot, \$ \} \rightarrow I_8$$

$$\text{goto}(I_6, id) = \{ L \rightarrow * \cdot, \$ \} \rightarrow I_9$$

$$\text{goto}(I_6, id) = \{ L \rightarrow id \cdot, \$ \} \rightarrow I_{10}$$

states	Action				Goto		
	=	*	id	\$	S	L	R
I_0		S_4	S_5		1	2	3
I_1				acc			
I_2	S_6			γ_5			
I_3				γ_2			
I_4	γ_3			γ_3			
I_5	γ_4			γ_4			
I_6		S_9	S_{10}		8	7	
I_7				γ_1			4
I_8				γ_5			
I_9				γ_3			
I_{10}				γ_4			

LALR Parsing:-

465

$$S \rightarrow CC$$

$$C \rightarrow cC$$

$$C \rightarrow d$$

first four steps same as CLR parsing

step-5:- find the core items & their unions

core items

$$I_3, I_6$$

$$I_4, I_7$$

$$I_8, I_9$$

Union

$$\underline{I_{36}} = \{ C \rightarrow c \cdot C, \underline{c|d|\$}, C \rightarrow \cdot cC, \underline{c|d|\$}, \\ C \rightarrow \cdot d, \underline{c|d|\$} \}$$

$$I_{47} = \{ C \rightarrow d \cdot, c|d|\$ \}$$

$$I_{89} = \{ C \rightarrow cC \cdot, c|d|\$ \}$$

✓

or 49

States	Action			Goto	
	c	d	\$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			acc		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

Construct LALR parsing table for

$A' \rightarrow A$ - 0

$A \rightarrow (A)$ - 1

$A \rightarrow a$ - 2

Step-2

LR(1) item:-

$LR(1) = \{ A' \rightarrow \cdot A, \$ \} - I$

Step-3

$closure(I) = \{ A' \rightarrow \cdot A, \$$

$A \rightarrow \cdot (A), \$$

$A \rightarrow \cdot a, \$ \} \rightarrow I_0$

Step-4:-

Goto function

$$\text{goto}(I_0, A) = \{ A' \rightarrow A \cdot, \$ \} \rightarrow I_1$$

$$\text{goto}(I_0, () = \{ A \rightarrow (\cdot A), \$ \}$$

$$A \rightarrow \cdot (A), \rangle / \$$$

$$A \rightarrow \cdot a, \rangle / \$ \} \rightarrow I_2$$

$$\text{goto}(I_0, a) = \{ A \rightarrow a \cdot, \$ \} \rightarrow I_3$$

$$\text{goto}(I_2, A) = \{ A \rightarrow (A \cdot), \$ \} \rightarrow I_4$$

$$\text{goto}(I_2, () = \{ A \rightarrow (\cdot A), \rangle / \$$$

$$A \rightarrow \cdot (A), \rangle / \$$$

$$A \rightarrow \cdot a, \rangle / \$ \} \rightarrow I_5$$

$$\text{goto}(I_2, a) = \{ A \rightarrow a \cdot, \rangle / \$ \} \rightarrow I_6$$

$$\text{goto}(I_4, \rangle) = \{ A \rightarrow (A) \cdot, \$ \} \rightarrow I_7$$

$$\text{goto}(I_5, A) = \{ A \rightarrow (A \cdot), \rangle / \$ \} \rightarrow I_8$$

$$\text{goto}(I_5, () = \{ A \rightarrow (\cdot A), \rangle / \$, A \rightarrow \cdot (A), \rangle / \$$$

$$A \rightarrow \cdot a, \rangle / \$ \} \rightarrow I_5$$

$$\text{goto}(I_5, a) = \{ A \rightarrow a \cdot, \rangle / \$ \} \rightarrow \textcircled{I_2}$$

$$\text{goto}(I_8, ()) = \{ A \rightarrow (A) \cdot, \rangle / \$ \} \rightarrow \textcircled{I_9}$$

Step 5: - find the core items & their unions

Core items:

$$I_2, I_5$$

$$I_3, I_6$$

$$I_7, I_9$$

$$I_4, I_8$$

Unions:-

$$I_{25} = \{ A \rightarrow (\cdot A), \$ / \rangle, A \rightarrow - (A), \rangle / \$, \\ A \rightarrow \cdot a, \rangle / \$ \}$$

$$I_{36} = \{ A \rightarrow a \cdot, \rangle / \$ \}$$

$$I_{79} = \{ A \rightarrow (A) \cdot, \rangle / \$ \}$$

$$I_{48} = \{ A \rightarrow (A \cdot), \rangle / \$ \}$$

states	Action				Goto
	(	)	a	\$	
I ₀	S ₂₅		S ₃₆		1
I ₁				acc	
I ₅	S ₂₅		S ₃₆		48
I ₃₆		Y ₂		Y ₂	
I ₆		S ₇₉			
I ₇₉		Y ₁		Y ₁	

Operator precedence parsing :-

It is a kind of bottom up parsing.

If the grammar satisfies the following condition then it is an operator grammar.

1. No production rules should contain ϵ symbol on the right hand side of the production.

2. No production rules should have two adjacent non terminals on the right hand side of the production.

Eg:- $E \rightarrow EAE \mid (E) \mid id$

$A \rightarrow + \mid - \mid * \mid \mid / \mid \uparrow$

The above grammar is not an operator grammar because the right hand side of the first production contains 2 consecutive non-terminals.

It can be converted into an operator grammar by substituting the alternatives of A and we obtain the grammar as

$$A \rightarrow E + E / E - E / E * E / E \div E / E \uparrow E / (E) / id$$

Relations in operator precedence parsing:-

In operator precedence parsing we define 3 disjoint relations between the terminals.

They are

$a < b$ where a yields precedence to b

$a > b$ where a takes precedence over b

$a \equiv b$ where a & b have the same precedence

Operator precedence parsing from associativity:-

1. Power operator $\wedge$ is the highest precedence operator and it is right associative

2. Multiplication and division operators are the next highest precedence operators and they are left associative.

3. Addition and subtraction operators are the lowest precedence operators and they are left associative.

Rules:-

Rule 1:- If the operator θ_1 has the higher precedence than the operator θ_2 then make

$$\theta_1 \rightarrow \theta_2 \quad \& \quad \theta_2 < \theta_1$$

Ex:-

$$\begin{array}{llll} \uparrow \rightarrow + & + < \uparrow & * \rightarrow + & + < * \\ \uparrow \rightarrow - & - < \uparrow & * \rightarrow - & - < * \\ \uparrow \rightarrow * & * < \uparrow & / \rightarrow + & + < / \\ \uparrow \rightarrow / & / < \uparrow & / \rightarrow - & - < / \end{array}$$

Rule 2:- If the operators θ_1 & θ_2 are of same precedence and it is left associative

then make $\theta_1 \rightarrow \theta_2$ & $\theta_2 \rightarrow \theta_1$

Ex:-

$$\begin{array}{ll} + \rightarrow - & + + id \\ + \rightarrow + & \\ - \rightarrow - & 1 + 1 + 12 \\ * \rightarrow * & \\ * \rightarrow / & \\ / \rightarrow / & \\ - \rightarrow + & \\ / \rightarrow + & \end{array}$$

Rule 3:- If the operators θ_1 & θ_2 are of same precedence and if it is right associative then make $\theta_1 < \theta_2$ &

$\theta_2 < \theta_1$

Ex:- $\uparrow < \cdot \uparrow$

Rule 4:- Make

$\theta < \cdot id$	$\theta > \$$	$\theta < ($	$\theta >)$
$id > \theta$	$\$ < \theta$	$(> \theta$	$) > \theta$
		$(< \theta$	

where θ represents the operators

Ex:-

$+ < \cdot id$	$id > +$	$+ > \$$	$\$ < \cdot +$
$- < \cdot id$	$id > -$	$- > \$$	$\$ < \cdot -$
$* < \cdot id$	$id > *$	$* > \$$	$\$ < \cdot *$
$/ < \cdot id$	$id > /$	$/ > \$$	$\$ < \cdot /$
$\uparrow < \cdot id$	$id > \uparrow$	$\uparrow > \$$	$\$ < \cdot \uparrow$

$+ < ($	$(< +$	$+ >)$	$) > +$
$- < ($	$(< -$	$- >)$	$) > -$
$* < ($	$(< *$	$* >)$	$) > *$
$/ < ($	$(< /$	$/ >)$	$) > /$
$\uparrow < ($	$(< \uparrow$	$\uparrow >)$	$) > \uparrow$

Rule 5:- make

$\$ (id + id * id) \$$

$id - id$	$\$ < id$	$(< id$	$) - id$
$id - ($	$\$ < ($	$(< ($	$) - ($
$id >)$	$\$ -)$	$(\div)$	$) >)$
$id > \$$	$\$ - \$$	$(- \$$	$) > \$$

	+	-	*	/	↑	(	)	id	\$
+	>	>	<	<	<	<	>	<	>
-	>	>	<	<	<	<	>	<	>
*	>	>	>	>	<	<	>	<	>
/	>	>	>	>	<	<	>	<	>
↑	>	>	>	>	<	<	>	<	>
(	<	<	<	<	<	<	$\div$	<	-
)	>	>	>	>	>	-	>	-	>
id	>	>	>	>	>	-	>	-	>
\$	<	<	<	<	<	<	-	<	-

Operator precedence parsing from precedence function.

(6) Precedence table can be encoded by two precedence functions f and g that maps the terminal symbols to an integer.

We attempt to select f and g as a function of the terminals and their relations are given below

$$f(a) < g(b)$$

$$f(a) > g(b)$$

$$f(a) = g(b)$$

The precedence relation between a and b can be determined by the numerical comparisons between $f(a)$ and $g(b)$

Consider an expression $id + id * id$, the operator - precedence relation table from associativity for the given expression is given below.

	id	+	*	\$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	-

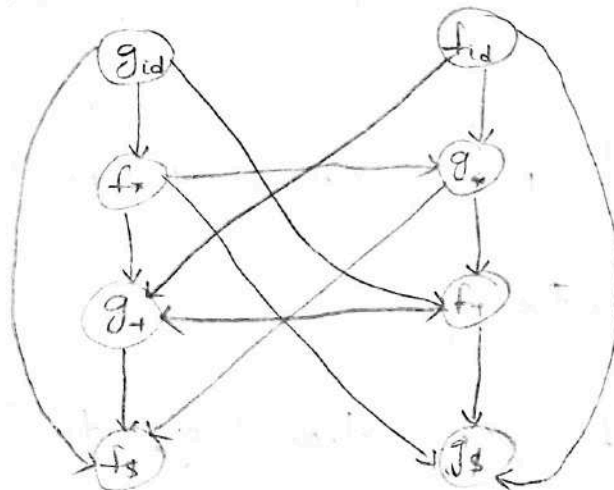
Steps for constructing the directed graph:-

1. All the terminal symbols are considered as a function such as

$f_{id}, f_+, f_*, f_\$, g_{id}, g_+, g_*, g_\$$

2. If $a < b$ then there will be an edge from g_b to f_a

3. If $a > b$ then there will be an edge from f_a to g_b



Steps for constructing precedence table:-

1. t_s & g_s has no outgoing edges. Therefore

$$t_s = g_s = 0$$

2. The longest path from t_{id} to t_s is

$$4 \text{ i.e. } t_{id} \rightarrow g_* \rightarrow t_+ \rightarrow g_+ \rightarrow t_s$$

3. The longest path from t_+ to t_s is

$$2 \text{ i.e. } t_+ \rightarrow g_+ \rightarrow t_s$$

4. The longest path from t_* to t_s is

$$t_* \rightarrow g_* \rightarrow t_+ \rightarrow g_+ \rightarrow t_s$$

5. The longest path from g_{id} to g_s is

$$5 \text{ i.e. } g_{id} \rightarrow t_* \rightarrow g_* \rightarrow t_+ \rightarrow g_+ \rightarrow t_s$$

6. The longest path from g_+ to t_s is

$$1 \text{ i.e. } g_+ \rightarrow t_s$$

7. The longest path from g_* to t_s is

$$3 \text{ i.e. } g_* \rightarrow t_+ \rightarrow g_+ \rightarrow t_s$$

	id	+	*	\$
f	4	2	4	0
g	5	1	3	0

*** Operator precedence parsing algorithm:-

① Input:- An input string and the table of precedence relation.

Method:- Initially, stack contains \$ and input buffer contains the input string w\$

set the i/p pointer point to the first symbol of i/p buffer

If \$ is on the top of the stack and i/p pointer points to \$ symbol then accept

else if i/p pointer points to terminal symbol 'b' and top of the stack has terminal symbol 'a' then

if $a < b$ or $a = b$

then push 'b' on to the stack

Advance the i/p pointer

else if $a > b$ then

pop the top of stack until the

terminal symbol is related with $<$ to

the terminal most recently popped

else error.

Ex:- Stack implementation

Stack	i/p buffer	Operation
\$	id + id * id \$	$\$ < id$ push
\$ id	+ id * id \$	$id > +$ pop
\$	+ id * id \$	$\$ < +$ push
\$ +	id * id \$	$+ < id$ push
\$ + id	* id \$	$id > *$ pop
\$ +	* id \$	$+ < *$ push
\$ + *	id \$	$* < id$ push
\$ + * id	\$	$id > \$$ pop
\$ + *	\$	$* > \$$ pop
\$ +	\$	$+ > \$$ pop
\$	\$	accept

stack

\$

\$id

\$

\$*

\$*id

\$*

\$

\$+

\$+id

\$+

\$

if/btree

id * id + id \$

* id + id \$

* id + id \$

id + id \$

+ id \$

+ id \$

+ id \$

id \$

\$

\$

\$

operation

\$ < id push

id > * pop

\$ < * push

* < id push

id > + pop

* > + pop

\$ < + push

+ < id push

id > \$ pop

+ > \$ pop

accept

Error recovery in op parsing

There are 2 points in parsing process at which operator precedence parser can discover syntactic errors

- ① If no precedence relation holds b/w the terminal on top of the stack and current i/p
- ② If a handle has been found but there is no production with the handle as a right side.

Ambiguous Grammars:-

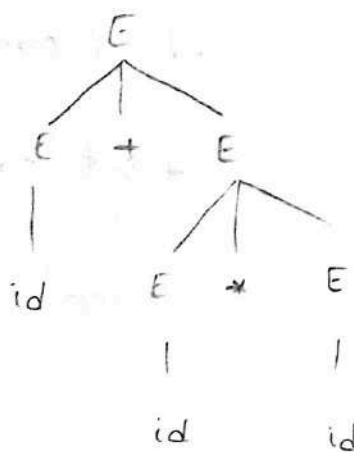
Ambiguous grammar is the grammar that produces more than one parse tree for the same input string.

Ex:- $E \rightarrow E + E$

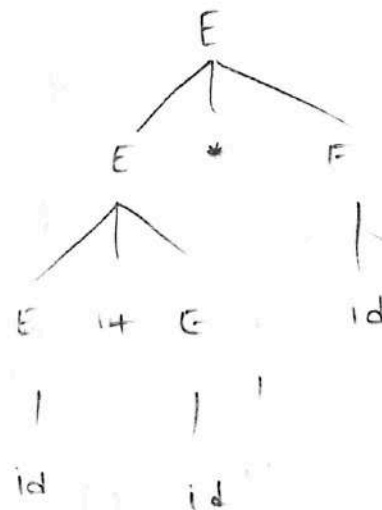
$E \rightarrow E * E$

$E \rightarrow id$

Input string :- $id + id * id$



(parse tree 1)



parse tree 2

The input string $id + id * id$ cannot be parsed by certain parsers because of ambiguous in nature.

So before parsing the ambiguity should be eliminated to remove invalid parse tree.

Generally the computations are done from bottom of the tree so the valid parse tree is the parse tree, because multiplication operator has higher precedence than addition operator.

Eliminate the ambiguity for the following grammars.

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow id$$

$$E \rightarrow \underbrace{E + E}_{+} \mid \underbrace{E * E}_{*}$$

$$A \rightarrow xB \mid xC$$

$$A \rightarrow xA'$$

$$A' \rightarrow B \mid C$$

$$E \rightarrow EE'$$

$$E' \rightarrow +E \mid *E$$

$$E \rightarrow id$$

$$S \rightarrow Aa \mid bAc \mid dc \mid bda$$

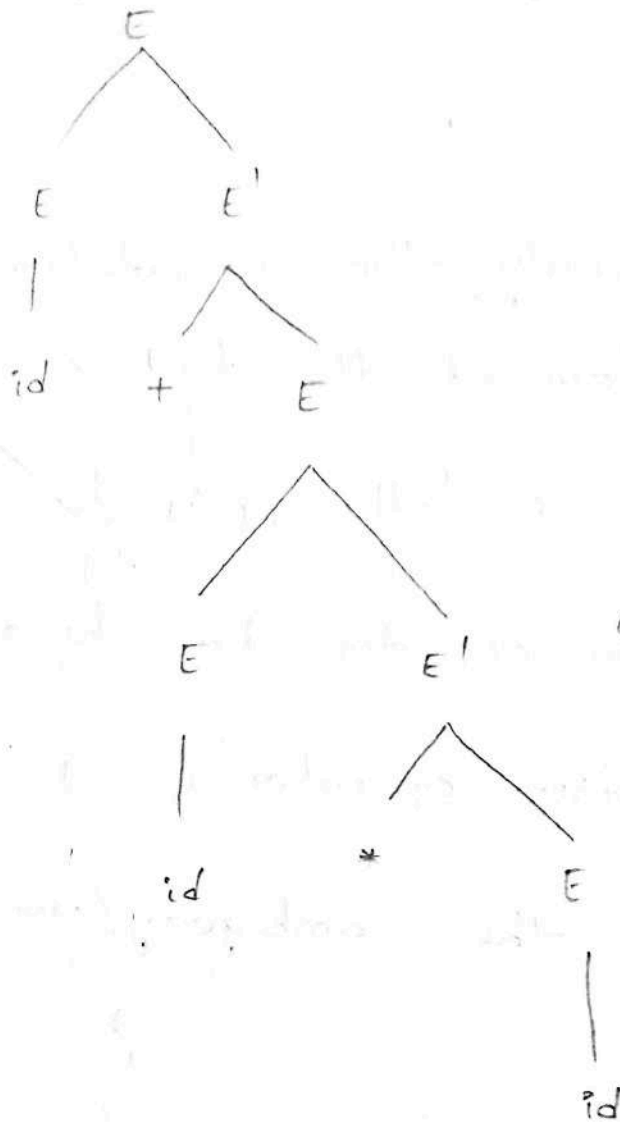
$$A \rightarrow d$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow TF \mid F$$

$$F \rightarrow F * \mid a \mid b$$

input string id + id * id



Elimination of ambiguity for the dangling else grammars.

Consider the following dangling else grammars

$\text{stmt} \rightarrow \text{if expr then stmt}$

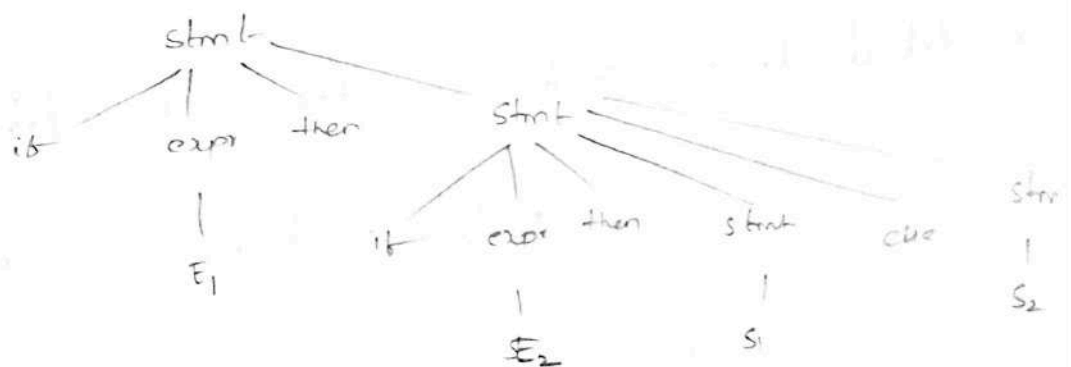
$\text{stmt} \rightarrow \text{if expr then stmt else stmt}$

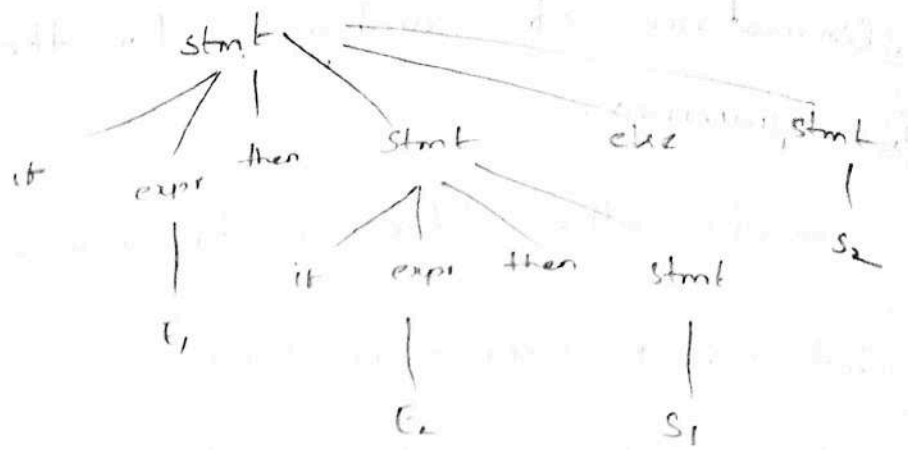
$\text{stmt} \rightarrow S_1 / S_2 / S_3 \dots$

$\text{expr} \rightarrow E_1 / E_2 / E_3 \dots$

The dangling else grammar is an ambiguous grammar for the following input string which has two parse trees as follows

if E_1 then if E_2 then S_1 else $S_2 \rightarrow \textcircled{2}$





Ambiguous grammars can be rewritten to eliminate the ambiguity. As an example we can rewrite the dangling else grammars as follows to eliminate the ambiguity.

$stmt \rightarrow \text{matched stmt} / \text{Unmatched stmt}$

$\text{matched stmt} \rightarrow \text{if expr then matched stmt else}$

$\text{matched stmt} / S_1 / S_2 / S_3 \dots$

$\text{Unmatched stmt} \rightarrow \text{if expr then stmt / if expr then}$

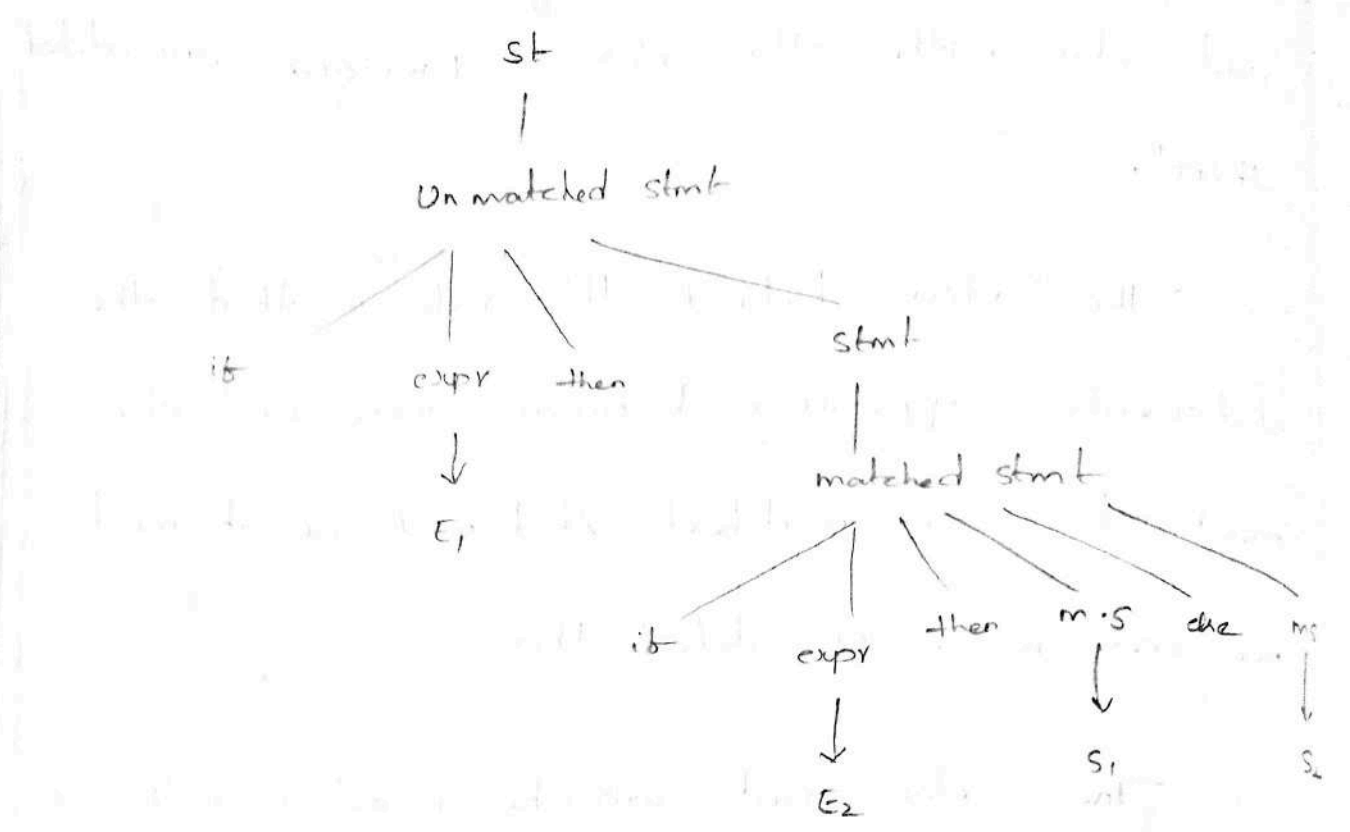
$\text{matched stmt else unmatched stmt}$

$\text{expr} \rightarrow E_1 / E_2 / E_3 \dots$

The general rule says that "match each else with the closest previous unmatched then".

The idea behind this rule is that the statements appearing between then and else must be a matched statement i.e., it must not end with unmatched then.

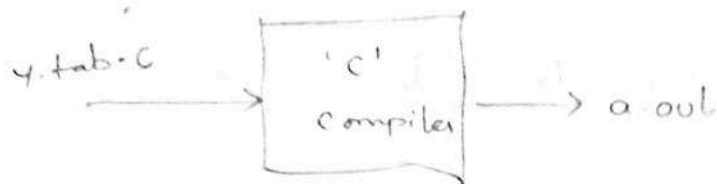
The else part can be matched with the unmatched statement. The rewritten grammar generates same set of input ~~three~~ strings as in eq① but it allows only one parse tree for the input string in the eq② nond. that associates each else with the closest previous unmatched then.



YACC - automatic parser generator

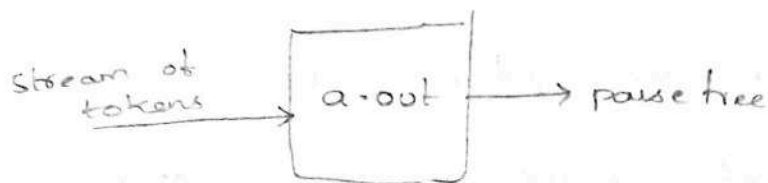


YACC - Yet another
compilers
compiler



\$YACC translate y

\$CC y.tab.c -lj



YACC stands for yet another compilers compiler, reflecting the popularity of the parser generator.

YACC is available as a command in the "UNIX operating system" and it has been used for constructing hundreds of compilers.

The file `translate.y` containing YACC specification of the translator is prepared.

The UNIX system command `yacc` (translate.y) transforms the file `translate.y` into a C program.

`y.tab.c`

The program `y.tab.c` is the representation of LALR parser written in C language.

By compiling `y.tab.c` along with the lex library that contains LR parsing program using the command

`$ cc y.tab.c -llex`, we obtain the

desired object program `a.out` which constructs the parse tree from the stream of tokens.

YACC specification:-

The YACC program consists of 3 parts. They are

1. Declaration part
2. Translation rules
3. Supporting 'C' routines.

Declaration part:-

There are two sections in the declaration part of the YACC program.

1. An ordinary temporary variable declaration delimited by `%. {` --- `%. }` which is used in the translation rules.

2. The second section is the declaration of grammar tokens.

Ex:- `%. token digit` where `digit` is the token declared.

Translation rules:-

The translation rules of the YACC program is followed by a

pair $\% \%$.

Each rule consist of grammar productions and associated semantic actions.

The YACC semantic action is the sequence of statements in which $\$ \$$ indicates the attribute ^{value} information associated with the non-terminals on the left. & $\$ i$ indicates attribute ^{value} information associated with the non-terminal on the right.

Ex:- $E \rightarrow E+T/T$

The translation rules for the above grammar is

$\% \%. E : E+T \{ \$ \$: \$_1 + \$_2 \}$

$| T \{ \$ \$: \$_1 \}$

Supporting 'c' routines:-

The third part of the YACC program consist of supporting 'c' program used in the execution of the passes.

45 min - 698

1/2 - Open

15 min - Flat

10 min - 1070

the set of procedures we have been with
<kind> -> call 10 1070

we write in yacc as
<kind> : call 10 { ... }
: call 20 { ... }
: call 100 { ... }

Syntax tree:-

It is the condensed form of parse tree used for representing the language constructs.

In syntax tree the operators & keywords appears as an interior node and operands appears as an leaf node.

Construction of syntax tree:-

1. Each node in the syntax tree is a record structure with several fields for operators & operands.

E2:- In the operator node 1-field identifies the operator & the remaining fields contains the pointers to the children's of that node.

2. functions used to create the nodes of the syntax tree are

1. `mknode (op, left, right) :-`

It creates an operator node with the label `op` & the two fields `left` & `right` contain pointers to the left & right child of the operator node.

2. `mkleaf (id, entry) :-`

It creates ~~the~~ an identifier leaf node with the label `id` & field `entry` contains pointer to the symbol table.

3. `mkleaf(num, value)`

It creates a number leaf node with the label `num` & the field `val` contains the value of the number.

Construct the syntax tree for the following expression

i) $a * 4 + 5$

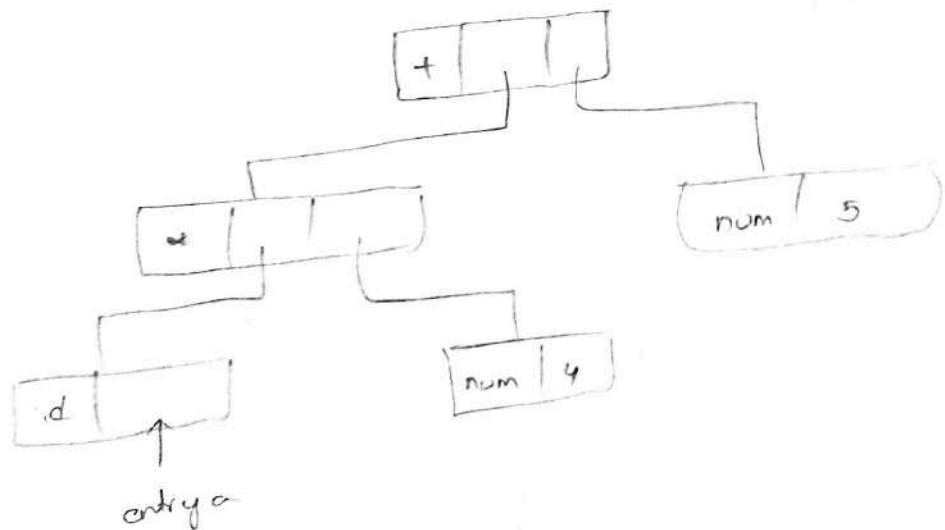
$P_1 = \text{mkleaf}(\text{id}, \text{entry } a)$

$P_2 = \text{mkleaf}(\text{num}, 4)$

$P_3 = \text{mknode}(*, P_1, P_2)$

$P_4 = \text{mkleaf}(\text{num}, 5)$

$P_5 = \text{mknode}(+, P_3, P_4)$



Syntax tree

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

$P_1 = \text{mkleab}(\text{id}, \text{entry } a)$

$P_2 = \text{mkleab}(\text{id}, \text{entry } b)$

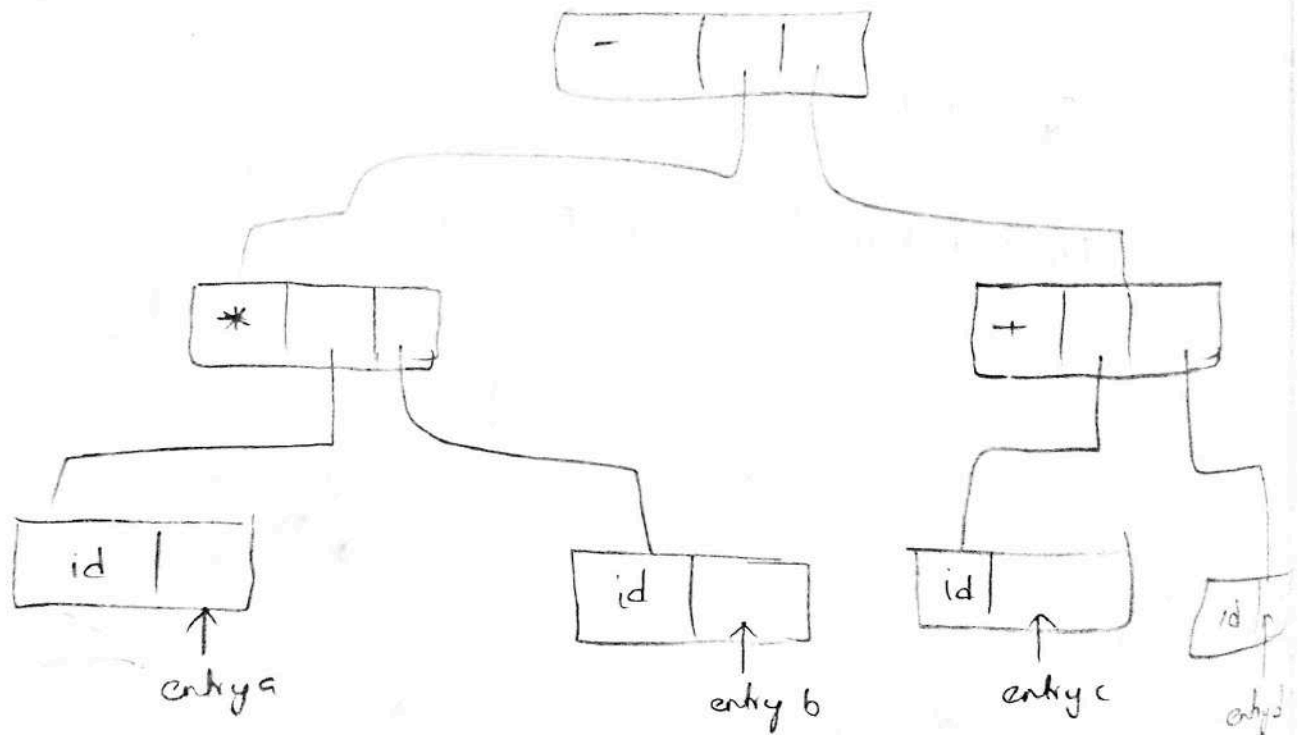
$P_3 = \text{mknode}(*, P_1, P_2)$

$P_4 = \text{mkleab}(\text{id}, \text{entry } c)$

$P_5 = \text{mkleab}(\text{id}, \text{entry } d)$

$P_6 = \text{mknode}(+, P_4, P_5)$

$P_7 = \text{mknode}(-, P_3, P_6)$



Syntax tree

Syntax directed definition :-

It is the generalisation of context free grammar in which each grammar symbol is associated with the set of attributes.

Attributes consist of 2 subsets

1. Synthesized attribute
2. Inherited attribute.

Synthesized attribute :-

The value of the synthesized attribute at a node is computed from the values of the attributes of the children of that node in the parse tree.

Inherited attribute :-

The value of inherited attribute is computed from the values of the attributes from the siblings & parent of that node in the parse tree. $(\rightarrow ABC)$

Annotated parse tree :-

The parse tree showing the value of the attribute at each node is called as annotated parse tree.

The value of the attribute at each node is computed using the semantic rule associated with the production.

$\rightarrow$ Ex. $E \rightarrow E + T$
 $Eval \Rightarrow Eval + Eval$
 $\quad \quad Eval$
 $\quad \quad \quad Eval + Eval$

20	9	18
7	9	11
		14
		16
		29
		36
		38
		42
		48
		49
		51
		52

Syntax directed definition for desk calculator:-

Grammar

Semantic rule

$$L \rightarrow E_n$$

$$\text{print}(E.\text{val})$$

$$E \rightarrow E + T$$

$$E.\text{val} = E.\text{val} + T.\text{val}$$

$$E \rightarrow T$$

$$E.\text{val} = T.\text{val}$$

$$T \rightarrow T * F$$

$$T.\text{val} = T.\text{val} * F.\text{val}$$

$$T \rightarrow F$$

$$T.\text{val} = F.\text{val}$$

$$F \rightarrow (E)$$

$$F.\text{val} = E.\text{val}$$

$$F \rightarrow \text{digit}$$

$$F.\text{val} = \text{digit}.\text{lexval}$$

— This definition associates an integer value synthesized attribute called val with the non terminals E, T & F .

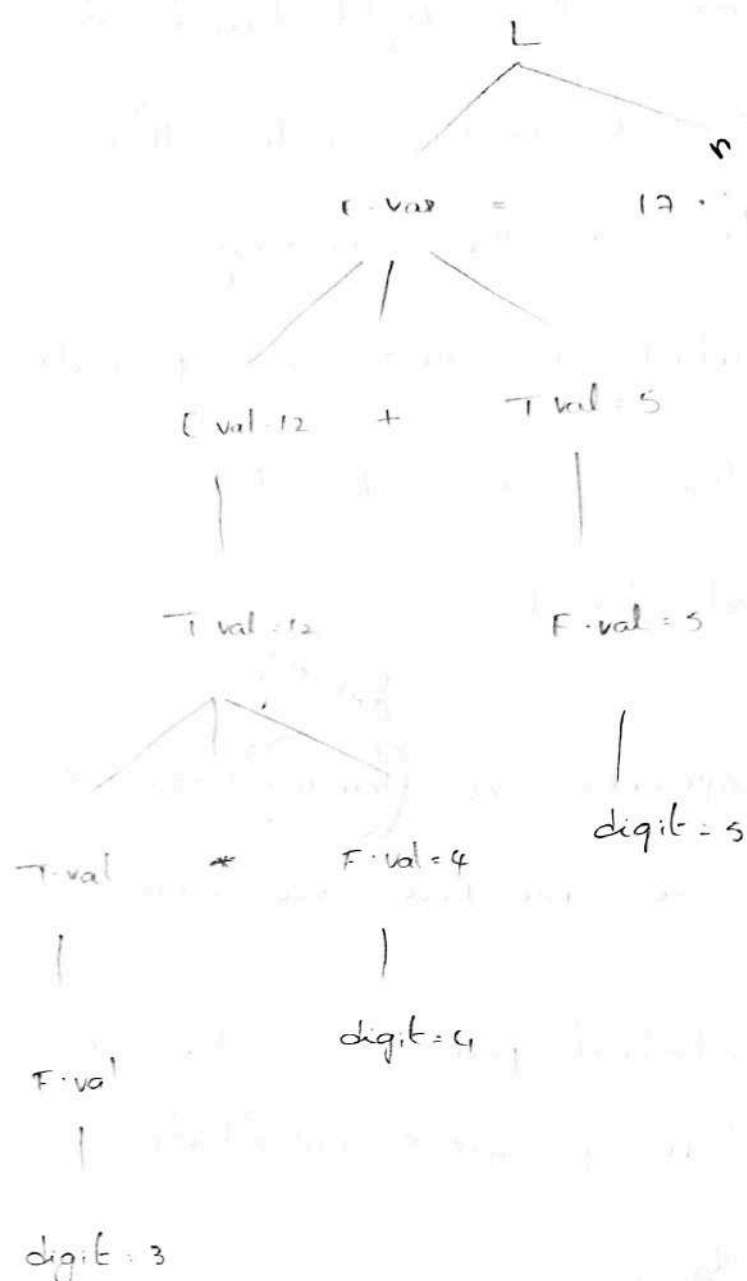
— The semantic rule computes the value of the attribute val for the non terminal on the left hand side from the value of the attribute val for the

non terminals on the right hand side

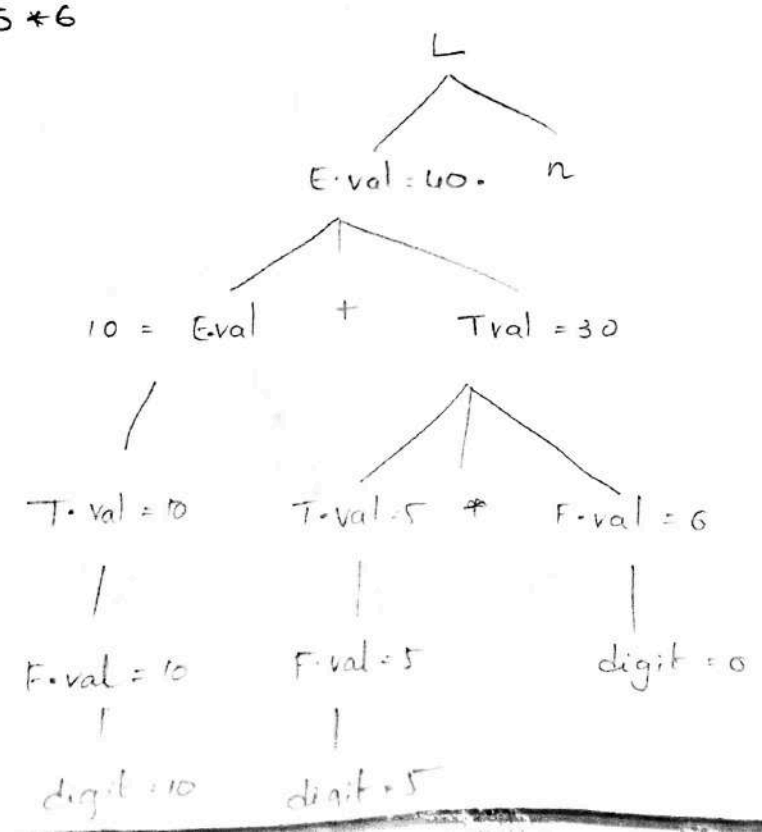
The rule associates with the production $L \rightarrow E_n$ for the starting nonterminal L which is just a procedure that prints the value of the expression generated by E .

Here L specifies a ^{dummy} dummy variable and n specifies a ~~new~~ new line character.

The annotated parse tree for the expression $3 * 4 + 5$ using desk calculator is given as follows.



$12 + 5 * 6$



Suppose the declarations are generated by the following grammar

$$D \rightarrow TL$$

$$T \rightarrow \text{int}$$

$$T \rightarrow \text{real}$$

$$L \rightarrow L, \text{id}$$

$$L \rightarrow \text{id}$$

a. Write the syntax directed definition to add the type of each identifier into the symbol table

b. Draw the annotated parse tree for the declaration `real id1, id2, id3.`

a. Grammar

$$D \rightarrow TL$$

$$T \rightarrow \text{int}$$

$$T \rightarrow \text{real}$$

$$L \rightarrow L, \text{id}$$

$$L \rightarrow \text{id}$$

Semantic rule

$$L.in = T.type$$

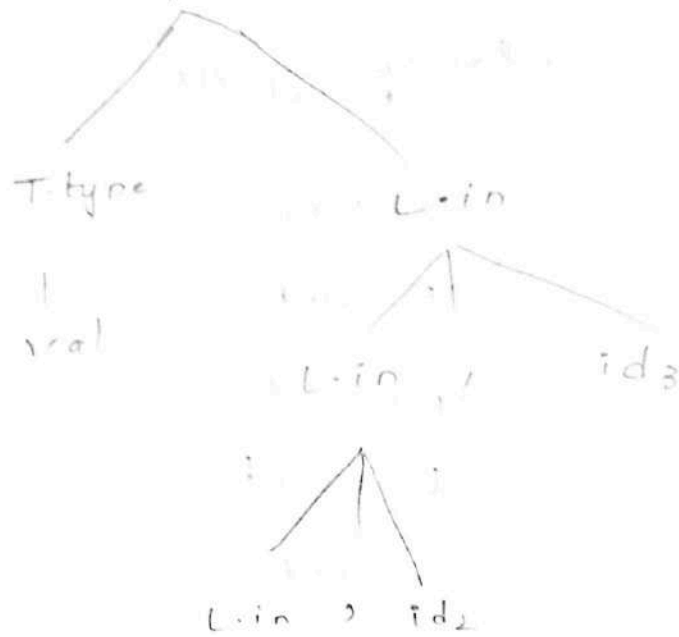
$$T.type = \text{integer}$$

$$T.type = \text{real}$$

$$L.in = L.in, \text{addtype}(. \\ \text{id.entry}, T.type)$$

$$L.in = \text{addtype}(\text{id.entry}, \\ T.type)$$

b)



S-attributed definition:-

The syntax directed definition that uses synthesized attribute exclusively is said to be S-attributed definition.

A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rule for the attributes at each node by bottomup parsing from leaves to the root.

L-attributed definition:-

A syntax directed definition is L-attributed definition if each inherited attribute of X_j where $1 < j < n$ on the right hand side of the production $A \rightarrow X_1 X_2 X_j \dots X_n$ depends on

1. Attributes of the symbols $X_1 X_2 \dots X_{j-1}$ to the left of X_j in the production
2. Inherited attributes of A

Dependency graph:-

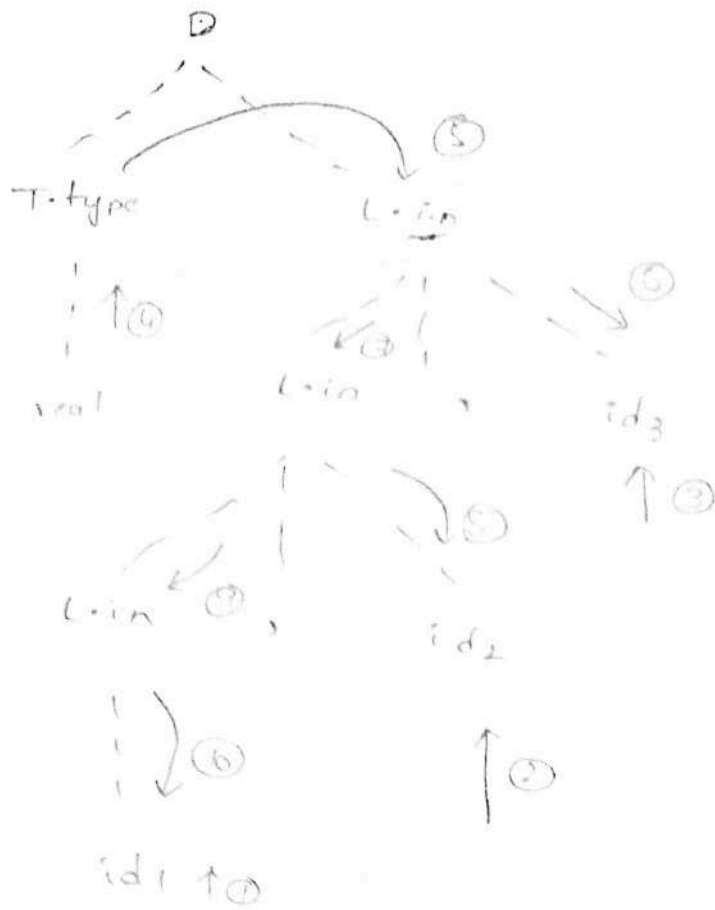
The dependency graph is a graph that contains both synthesized attribute and inherited attribute.

The interdependency among inherited attribute and synthesized attribute can be depicted by a directed graph called as dependency graph.

The dependency graph for the grammar

$$D \rightarrow TL$$
$$T \rightarrow \text{int}$$
$$T \rightarrow \text{real}$$
$$L \rightarrow Lid$$
$$L \rightarrow id$$

With the input string $\text{real } id_1, id_2, id_3$



Intermediate Code generators:-

There are several ways to represent the intermediate code generators.

They are

1. Three address code
2. Syntax tree
3. Post-fix notation

The choice of intermediate code depends on the two important criteria. They are

1. It should be easy to convert the source program into intermediate code.

2. It should be easy for the subsequent processing to obtain the object program

Three address code:-

It is the sequence of statements of the general form $A = B \text{ op } C$ where A , B and C are the identifiers, constants or compiler generated temporaries

and op represents the operator.

Ex:- $T = x + y + z$

The three address code for the above expression is

$$t_1 = x + y$$

$$t_2 = t_1 + z$$

$$t = t_2$$

Types of three address code:-

1. Assignment statement of the form $A = B \text{ op } C$ where op is the binary or logical operators.

2. Assignment statement of the form $A = \text{op } B$ where op is the unary operators like unary minus, logical shift or conversion operators (int to real).

3. Copy statement of the form $A = B$ which copies the value of B to A .

4. Indexed assignment to handle the arrays like $A = B[i]$ where A, B, i are the data objects.

5. Address and pointer assignment of the form a) $A = \&B$ which assigns the address of B to A

b) $A = *B$ which assigns the value of B to A

6. Unconditional jump statement such as goto L₁ in which the three address statement with the label L₁ is the next statement to be executed.

7. Conditional jump statement

if A relop B, then goto L₁

where relop can be $<, >, \leq, \geq, =, !=$

If the condition is satisfied the control jumps to the three address statement with the label L₁ otherwise it starts the execution in the sequence of three address statement.

8. Param x_1 the above statement are used to

param x_2 implement the procedure calls.

It is generated as a part of

call to

param x_n the procedure $p(x_1, x_2, \dots, x_n)$

call $P_i n$ where n is an integer indicates the number of actual parameters in the procedure call.

Implementation of three address code :-

The three address statement are implemented as a record structure which contains several fields for the operator & operands.

There are 3 different types of representations. They are

1. Quadruple

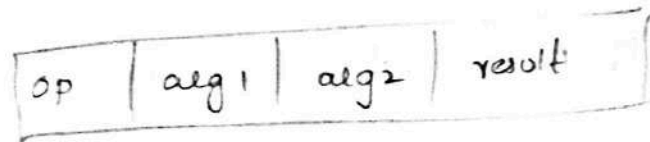
2. Triple

3. Indirect triple.

Quadruple:-

It is a record structure with four fields such as op, arg1, arg2 & result.

The general form of representing quadruple structure is



The op field contains an internal code to perform the operation specified by the operator.

The three address statement of the form $X = Y + Z$ is represented by placing Y in arg1, Z in arg2, and X in result and + in op field.

The statement with unary operators $X = -Y$ or $X = Y$ do not use the arg_2 field.

The content of the fields arg_1 , arg_2 and $result$ are normally a pointer to the symbol table.

The temporary names must be entered into the symbol table as they are created.

Ex:- $t = x + y + z$

The three address code for the above expression is

$$t_1 = x + y$$

$$t_2 = t_1 + z$$

$$t = t_2$$

The quadruple structure for the above three address code is

Op	arg1	arg2	result
+	x	y	t ₁
+	t ₁	z	t ₂
=	t ₂	-	t

Ex:- $a = b * -c + b * -c$

$$t_1 = -c$$

$$t_2 = b * t_1$$

$$t_3 = -c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$t = t_5$$

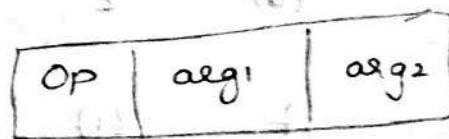
op	arg1	arg2	result
0 minus 3	c	-	t ₁
*	b	t ₁	t ₂
0 minus	c	-	t ₃
*	b	t ₃	t ₄
+	t ₂	t ₄	t ₅
=	t ₅	-	t

Triple:-

To avoid entering the temporary names into the symbol table we can store the value of temporary value by the position of the statement that computes it.

The three address statement for the triple structure can be represented as a record structure with only 3 fields such as OP, arg1, arg2

The triple structure is represented as



The fields arg1 & arg2 are either pointer to the symbol table or pointer to the triple structure itself.

Generally the parenthesised number represents pointer to the triple structure and the names represents pointer to the symbol table.

$$E_2: - t = x + y + z$$

$$t_1 = x + y$$

$$t_2 = t_1 + z$$

$$t_3 = t_2$$

	OP	arg1	arg2
(0)	+	x	y
(1)	+	(0)	z
(2)	=	t	(1)

The indexed assignments like $x = y[i]$ & $x[i] = y$ requires two entries in the triple structure

$$X = Y[i]$$

	OP	arg1	arg2
(0)	= []	Y	i
(1)	=	X	(0)

$$X[i] = Y$$

	OP	arg1	arg2
(0)	[] =	X	i
(1)	=	(0)	Y

Indirect triple :-

In this three address representation instead of ~~are~~ directly using the triple structure we can list a pointer to the triple structure. This is called as indirect triple.

Ex:- $t = X + Y + Z$

$$t_1 = X + Y$$

$$t_2 = t_1 + Z$$

$$t = t_2$$

(0)	(100)
(1)	(101)
(2)	(102)

	OP	arg1	arg2
(100)	+	X	Y
(101)	+	(100)	Z
(102)	=	t	(101)

$$*tx: a = b * -c + b * -c$$

$$x = DJX$$

$$t_1 = -c$$

$$t_2 = b * t_1$$

$$t_3 = -c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

Triple :-

	OP	arg1	arg2
(0)	Uminus	c	-
(1)	*	b	(0)
(2)	Uminus	c	-
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Syntax tree:-

It is the condensed form of parse tree used for representing the language constructs.

In syntax tree the operators & keywords appears as an interior node and operands appears as an leaf node.

Construction of syntax tree:-

1. Each node in the syntax tree is a record structure with several fields for operators & operands.

Ex:- In the operator node 1-field identifies the operator & the remaining fields contains the pointers to the children's of that node.

2. functions used to create the nodes of the syntax tree are

1. `mknode (op, left, right) :-`

It creates an operator node with the label `op` & the two fields `left` & `right` contain pointers to the left & right child of the operator node.

2. `mkleaf (id, entry) :-`

It creates ~~the~~ an identical leaf node with the label `id` & field `entry` contains pointer to the symbol table.

3. `mkleab (num, value)`

It creates a number leaf node with the label `num` & the field `val` contains the value of the number.

Construct the syntax tree for the following expression

i) $a * 4 + 5$

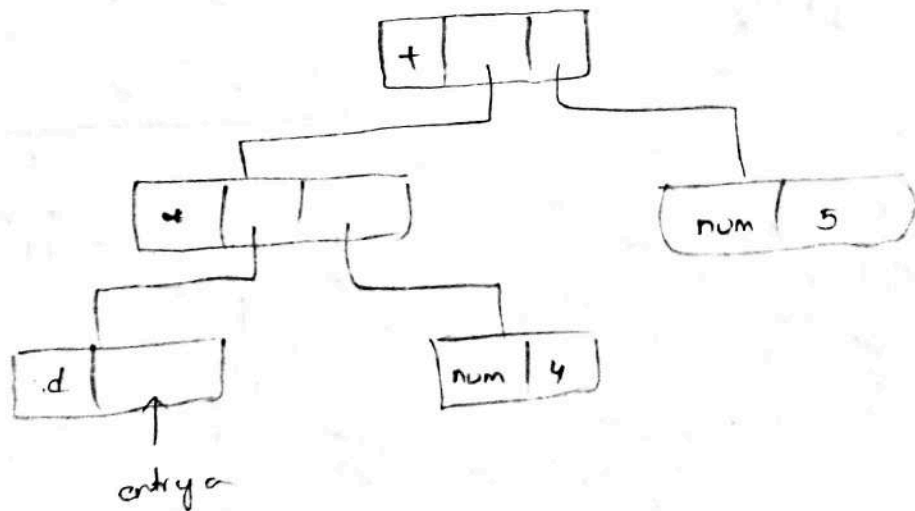
$P_1 = \text{mkleab}(\text{id}, \text{entry } a)$

$P_2 = \text{mkleab}(\text{num}, 4)$

$P_3 = \text{mknode}(*, P_1, P_2)$

$P_4 = \text{mkleab}(\text{num}, 5)$

$P_5 = \text{mknode}(+, P_3, P_4)$



Syntax tree

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

$P_1 = \text{mkleaf}(\text{id}, \text{entry a})$

$P_2 = \text{mkleaf}(\text{id}, \text{entry b})$

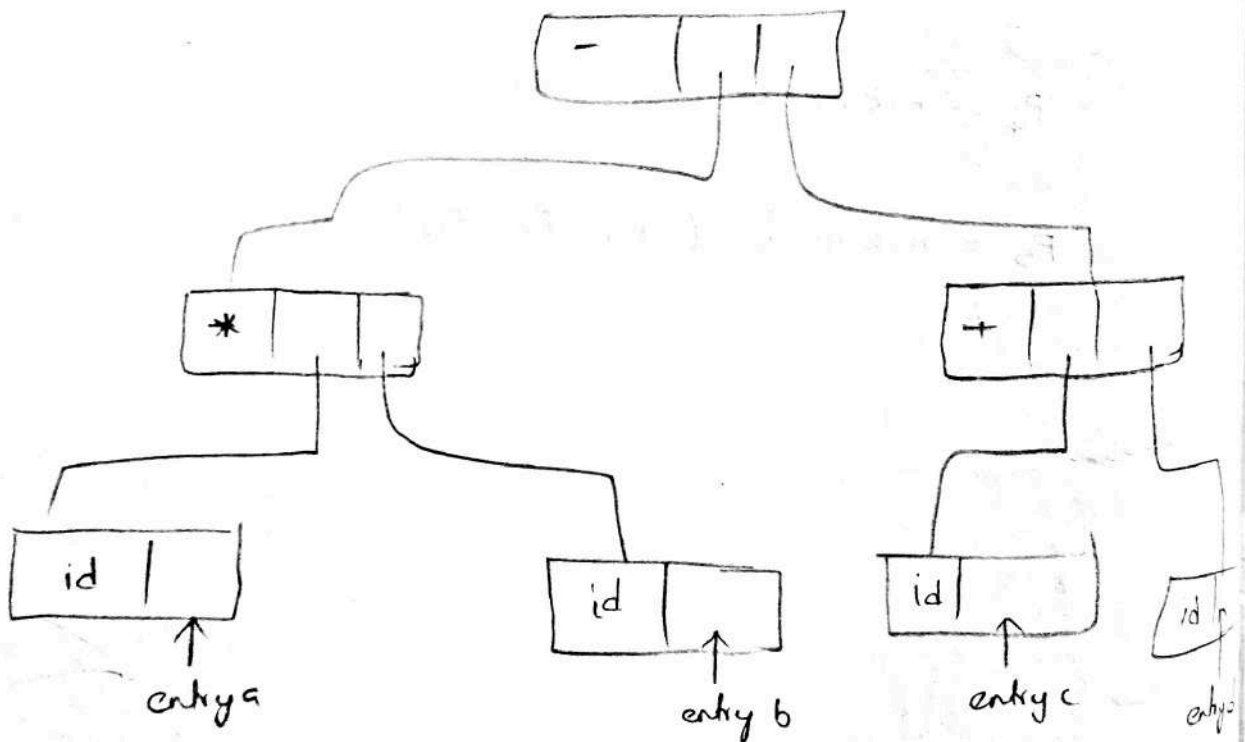
$P_3 = \text{mknode}(*, P_1, P_2)$

$P_4 = \text{mkleaf}(\text{id}, \text{entry c})$

$P_5 = \text{mkleaf}(\text{id}, \text{entry d})$

$P_6 = \text{mknode}(+, P_4, P_5)$

$P_7 = \text{mknode}(-, P_3, P_6)$



Syntax tree

Syntax directed definition :-

It is the generalisation of context free grammar in which each grammar symbol is associated with the set of attributes.

Attributes consist of a set

1. Synthesized attribute
2. Inherited attribute.

Synthesized attribute :-

The value of the synthesized attribute at a node is computed from the values of the attributes of the children of that node in the parse tree.

Inherited attribute :-

The value of inherited attribute is computed from the values of the attributes from the siblings & parent of that node in the parse tree. $\rightarrow ABC$

Annotated parse tree :-

The parse tree showing the value of the attribute at each node is called as annotated parse tree.

The value of the attribute at each node is computed using the semantic rule associated with the production.

$\rightarrow$ Ex. $E \rightarrow E + T$

$$E.val \Rightarrow E.val + T.val$$
$$\begin{array}{c} E.val \\ / \quad \backslash \\ T.val \end{array}$$

$$\begin{array}{r} 20 \mid 9 \mid 18 \\ \hline 7, 9, 11, 14, 16, 29, \\ 30, 38, 42, \\ 48, 49, 51, 52 \end{array}$$

Syntax directed definition for desk calculator

Grammar

Semantic rule

$$L \rightarrow E_n$$

print(E.val)

$$E \rightarrow E + T$$

$$E.val = E.val + T.val$$

$$E \rightarrow T$$

$$E.val = T.val$$

$$T \rightarrow T * F$$

$$T.val = T.val * F.val$$

$$T \rightarrow F$$

$$T.val = F.val$$

$$F \rightarrow (E)$$

$$F.val = E.val$$

$$F \rightarrow \text{digit}$$

$$F.val = \text{digit} \cdot \text{lexval}$$

— This definition associates an integer value synthesized attribute called val with the non terminals E, T & F .

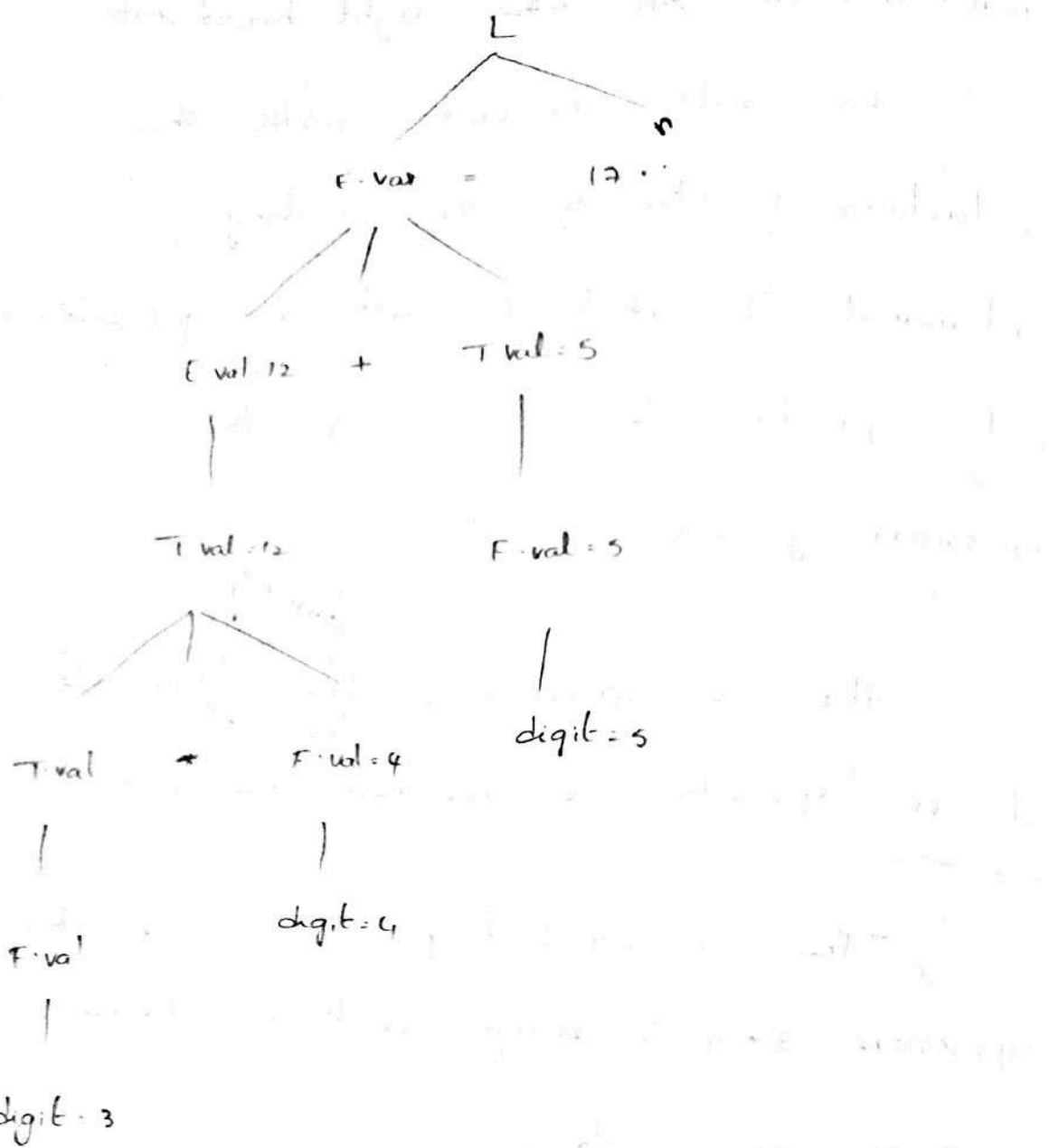
— The semantic rule computes the value of the attribute val for the non terminal on the left hand side from the value of the attribute val for the

non terminals on the right hand side

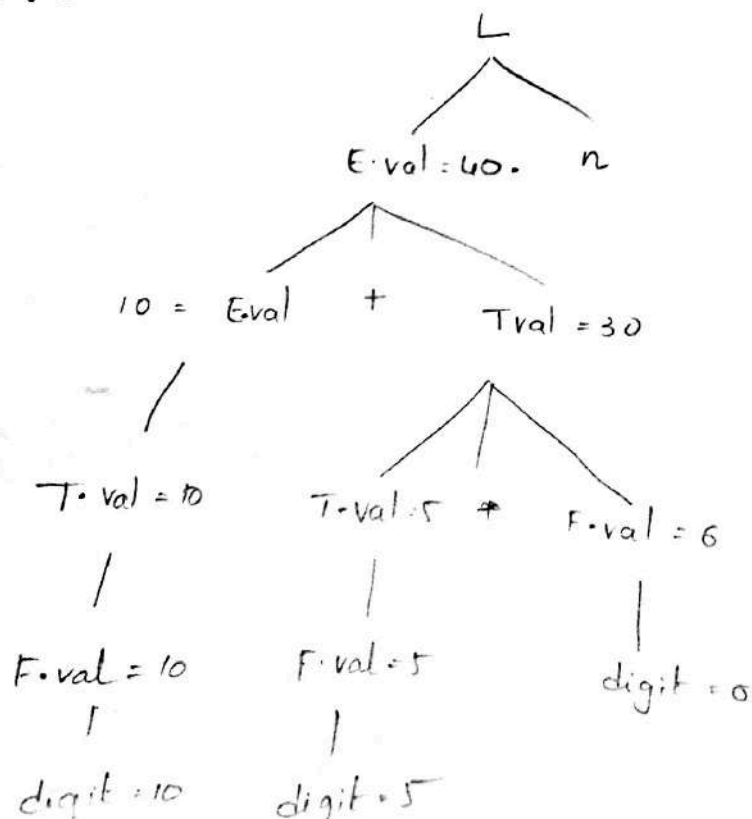
The rule associates with the production $L \rightarrow E_n$ for the starting nonterminal L which is just a procedure that prints the value of the expression generated by E .

Here L specifies a ^{dummy} dummy variable and n specifies a new line character.

The annotated parse tree for the expression $3 * 4 + 5$ using desk calculator is given as follows



$10 + 5 * 6$



S-attributed definition:-

The syntax directed definition that uses synthesized attribute exclusively is said to be S-attributed definition.

A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rule for the attributes at each node by bottomup parsing from leaves to the root.

L-attributed definition:-

A syntax directed definition is L-attributed definition if each inherited attribute of X_j where $1 < j < n$ on the right hand side of the production $A \rightarrow X_1 X_2 X_j \dots X_n$ depends on

1. Attributes of the symbols $X_1 X_2 \dots X_{j-1}$ to the left of X_j in the production
2. Inherited attributes of A

Dependency graph:-

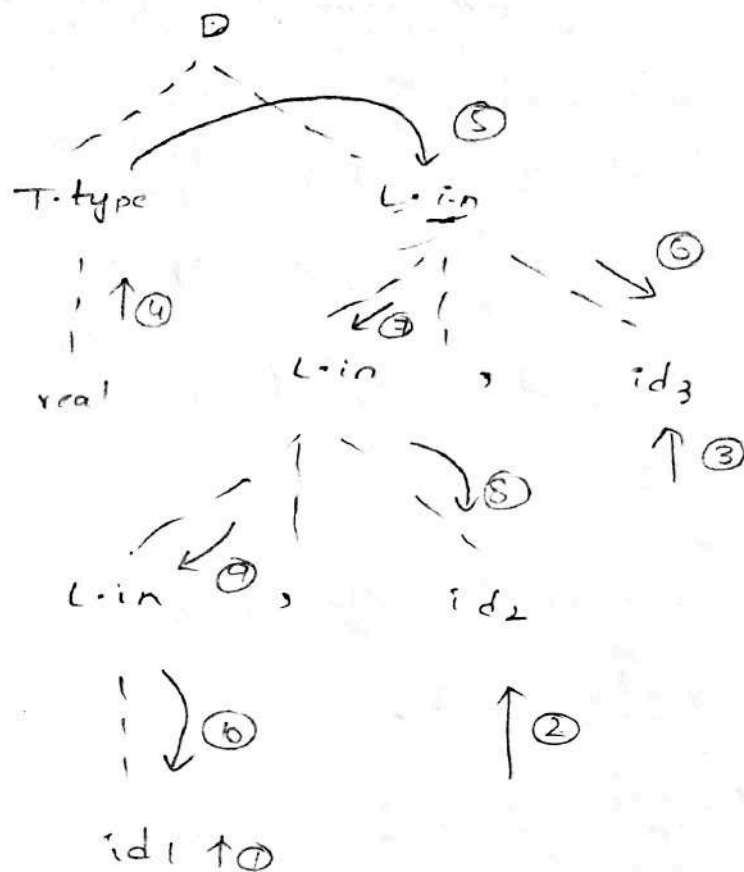
The dependency graph is a graph that contains both synthesized attribute and inherited attribute.

The interdependency among inherited attribute and synthesized attribute can be depicted by a directed graph called as dependency graph.

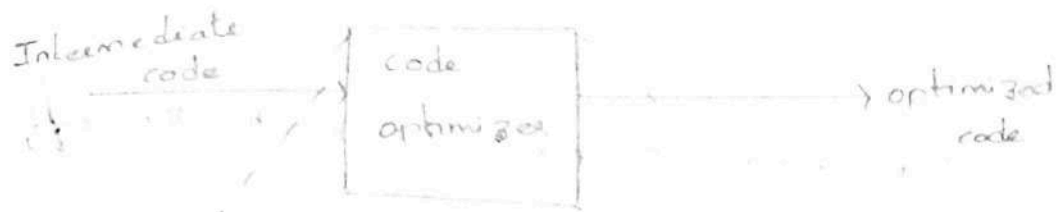
The dependency graph for the grammar

$$D \rightarrow TL$$
$$T \rightarrow \text{int}$$
$$T \rightarrow \text{real}$$
$$L \rightarrow L_1 id$$
$$L \rightarrow id$$

With the input string $\text{real id}_1 \text{id}_2 \text{id}_3$



Code optimizer :-



Code optimization is a technique which is used by most of the compilers to produce an "efficient and simple object code for the given source program."

Code optimization increases the processing speed of the compiler.

Properties of an optimizing compiler :-

1. The transformation must will not change the meaning of the original program.

2. Optimizing technique will increase the speed of the program in a considerable amount.

3. Transformation must be worth the effort.

Code optimization is based on two important criteria

1. Machine independent code optimization

2. Machine dependent code optimization

Basic blocks :-

It is the sequence of consecutive three address statement in which the control enters at the beginning and leaves at the end without any halt or branching statement except at the end.

Algorithm for dividing the program into basic blocks:-

Input: Series of three address statements

Output: List of basic blocks with each three address statement present in only one block.

Method:

Step-1:- Determine the set of leaders from the sequence of three address statement. We use the following rules to determine the leaders

Step-1.1:- The first statement of the program is a leader

Step-1.2:- Any statement i.e the target of conditional or unconditional goto statement is a leader

Step-1.3:- Any statement i.e immediately after the conditional or unconditional goto statement is a leader.

Step-2 :- For each leader its own block consist of the leader and all the statements upto the next leader or end of the program but not including the next leader.

Example :-

1. $l = n - 1$

2. $j = n / l$

3. $t_1 = 4 * i$

4. $V = a[t_1]$

5. $i = i + 1$

6. $t_2 = 4 * 2$

7. $t_3 = a[t_2]$

8. if $t_3 > V$ goto 3

9. $j = j - 1$

10. $t_4 = 4 + j$

11. $t_5 = a[t_4]$

if $t_5 < V$ goto 9

The leaders are

1. $l = m - 0$

3. $t_1 = 4 * i$

9. $j = j - 1$

The basic blocks are

basic block 1 :-

1. $l = m - 0$

2. $j = n / l$

basic block 2 :-

3. $t_1 = 4 * i$

4. $v = a[t_1]$

5. $i = i + 1$

6. $t_2 = 4 * 2$

7. $t_3 = a[t_2]$

8. if $t_3 > v$ goto 3

basic block 3 :-

9. $j = j - 1$

10. $t_4 = 4 + j$

11. $t_5 = a[t_4]$

12. if $t_5 < v$ goto 9

flow graph:-

It is a directed graph in which the basic blocks can be represented graphically with the flow of control information is called as flow graph

The nodes of the flow graph are the basic blocks, one node is distinguished as initial node and its leader is the first statement of the program

There will be a directed edge from the block B_1 to block B_2 due to the following reasons

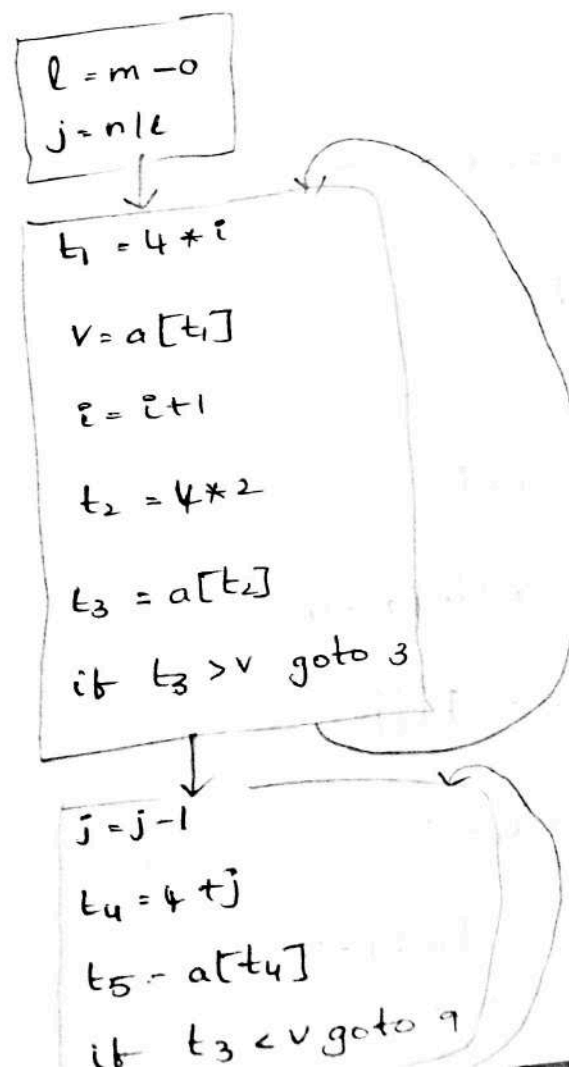
1. There is a conditional or unconditional jump statement in last statement of the block B_1 ^{if} The target of the statement is the first statement

of block B_2 then there will be an edge from block B_1 to B_2 .

a. In the order of the program the block B_2 immediately follows the block B_1 and if there is no unconditional jump statement in the last statement of the block B_1 then there is an edge from block B_1 to B_2 .

The block B_1 is called as predecessor of block B_2 & block B_2 is called as successor of block B_1 .

Ex:-



Construct the flow graph for the following program statements.

begin

prod = 0

i = 1

do

begin

prod = prod + a[i] * b[i]

i = i + 1

end

while i <= 20

end

Three address code :-

1. prod = 0

2. i = 1

3. t₁ = 4 + i

4. t₂ = addr(a) - 4

5. t₃ = t₂[t₁]

6. t₄ = 4 * i

7. t₅ = addr(b) - 4

8. t₆ = t₅[t₄]

$$9. t_2 = t_3 + t_4$$

$$10. \text{prod} = \text{prod} + t_2$$

$$11. i = i + 1$$

$$12. \text{if } i \leq 20 \text{ goto } 3$$

The leaders are

$$1^{\text{st}} \text{ leader } 1. \text{prod} = 0$$

$$2^{\text{nd}} \text{ leader } 3. t_1 = 4 + i$$

The basic blocks are

$$\text{basic block 1} :- 1. \text{prod} = 0$$

$$2. i = 1$$

$$\text{basic block 2} :- 3. t_1 = 4 + i$$

$$4. t_2 = \text{addr}(a) - 4$$

$$5. t_3 = t_2[t_1]$$

$$6. t_4 = 4 + i$$

$$7. t_5 = \text{addr}(b) - 4$$

$$8. t_6 = t_5[t_4]$$

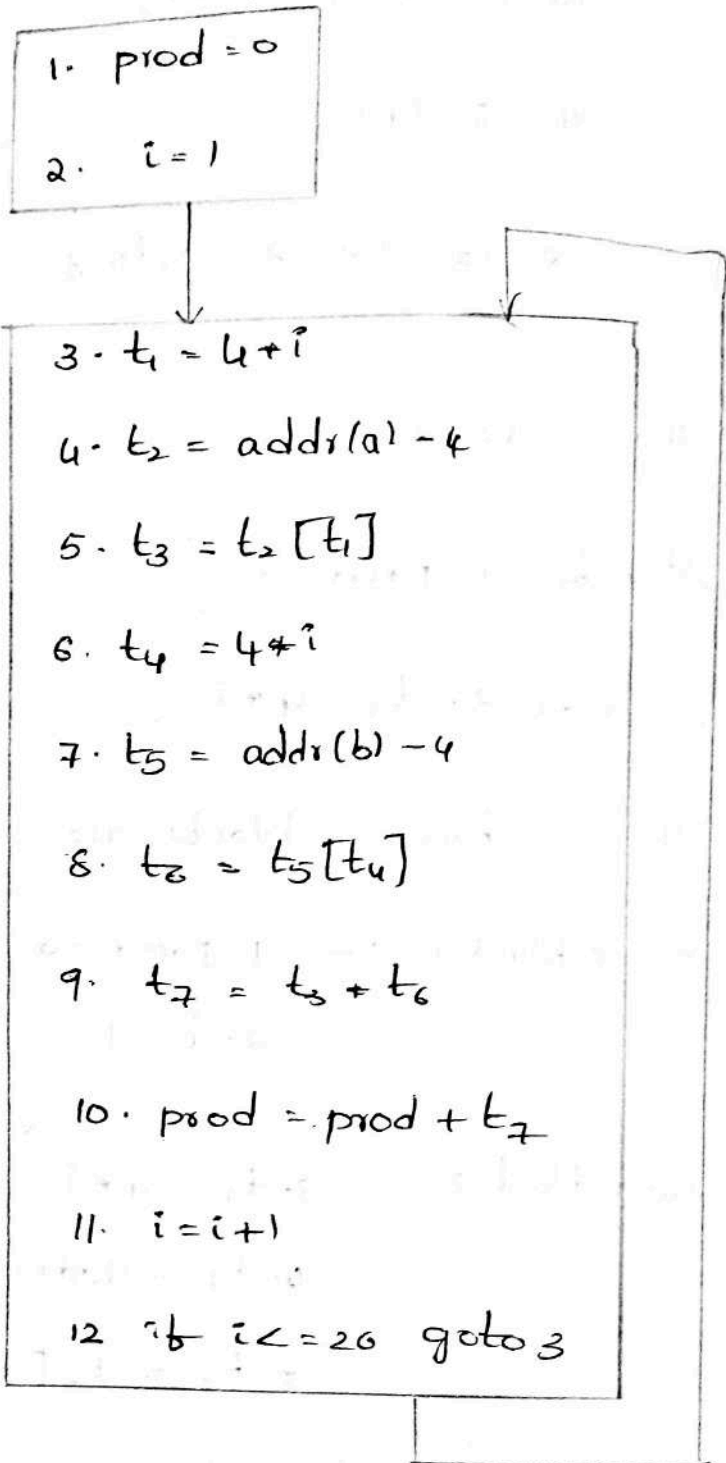
$$9. t_2 = t_3 + t_6$$

$$10. \text{prod} = \text{prod} + t_2$$

$$11. i = i + 1$$

$$12. \text{if } i \leq 20 \text{ goto } 3$$

The flow graph is



Machine independent code optimization:- or
Principle sources of
Machine independent code optimization
has been classified into

1. functional preserving transformation
structural preserving transformation
local optimization
2. loop optimization
3. Algebraic Simplification.

functional preserving transformation:-

It is classified into

1. Common sub expression elimination
2. Copy propagation
3. Dead code elimination
4. Renaming temporary variables.

Common subexpression elimination

$$a = b + c \rightarrow 1$$

$$b = a + d \rightarrow 2$$

$$c = b + c \rightarrow 3$$

$$d = a + d \rightarrow 4$$

Here the expression $b+c$ appears in the same expression $(b+c)+d$. We can eliminate the eq③ because the eq① & eq② are producing the same value & they are not changed throughout the calculation.

We cannot eliminate eq③ even though eq① & eq② has the same expression because the value of b is changed meanwhile.

The above statements are optimized as follows

$$a = b + c$$

$$b = a + d$$

$$c = b + c$$

$$d = b$$

Copy propagation:-

The technique of replacing one variable with the other assigned variable is called as copy propagation.

Ex:-

$$V_1 = V_2 \rightarrow 1$$

$$t_4 = a + b \rightarrow 2$$

$$t_3 = t_4 \rightarrow 3$$

$$t_1 = V_1 + 4 \rightarrow 4$$

$$Z = V_2 + 4 - 2 \rightarrow 5$$

In the above example in the statement 4 the variable V_2 can be used in the place of V_1 by copy propagation.

Therefore the optimised code will

be

$$t_2 = a + b$$

$$t_3 = t_4$$

$$t_1 = v_2 + 4$$

$$z = v_2 + 4 - 2$$

The optimized transformation further leads to the common sub expression elimination for $v_2 + 4$.

Therefore the optimized code is

$$t_2 = a + b$$

$$t_3 = t_4$$

$$t_1 = v_2 + 4$$

$$z = t_1 - 2$$

Dead Code elimination :-

A variable is said to be dead if it is not used subsequently in the program.

Suppose a variable x is never used subsequently in the program after the statement $x = a + b$ then the statement $x = a + b$ can be removed without any change in execution. This process is called as dead code elimination.

It may happen accidentally in the program.

Renaming temporary Variables:-

Consider the statements

$$x = a + b$$

$$y = a + b$$

The value of the basic block does not change if all the instance of y is changed to x as $x = a + b$ which is same as the y computation.

Loop Optimization :-

It is very important since it gives more scope for optimizing techniques in the loops.

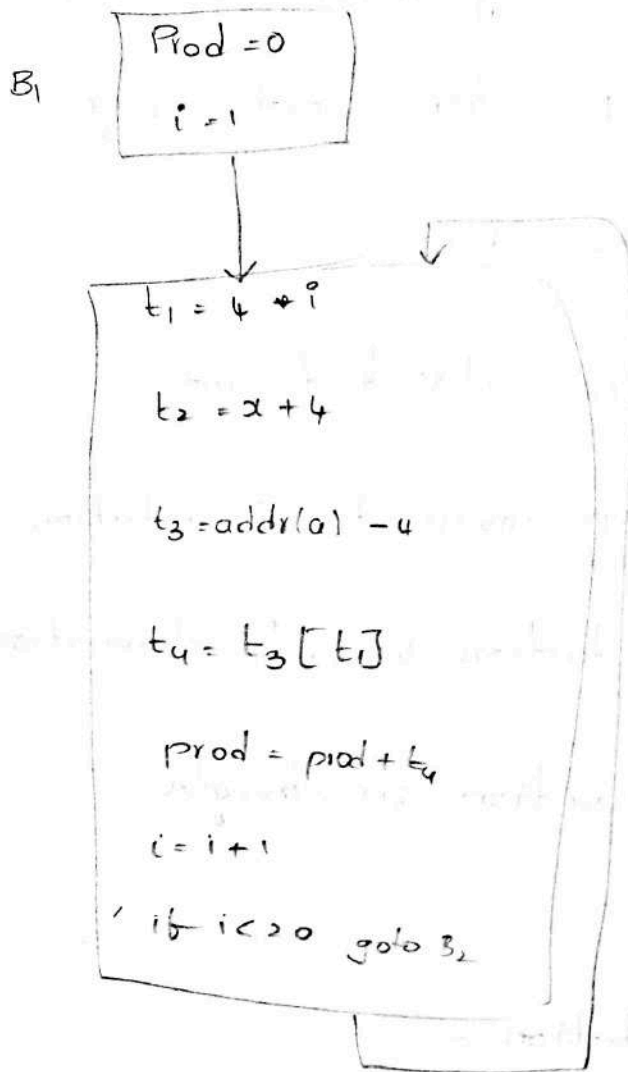
It is classified into

1. Loop invariant Computation
2. Induction variable elimination
3. Reduction in strength

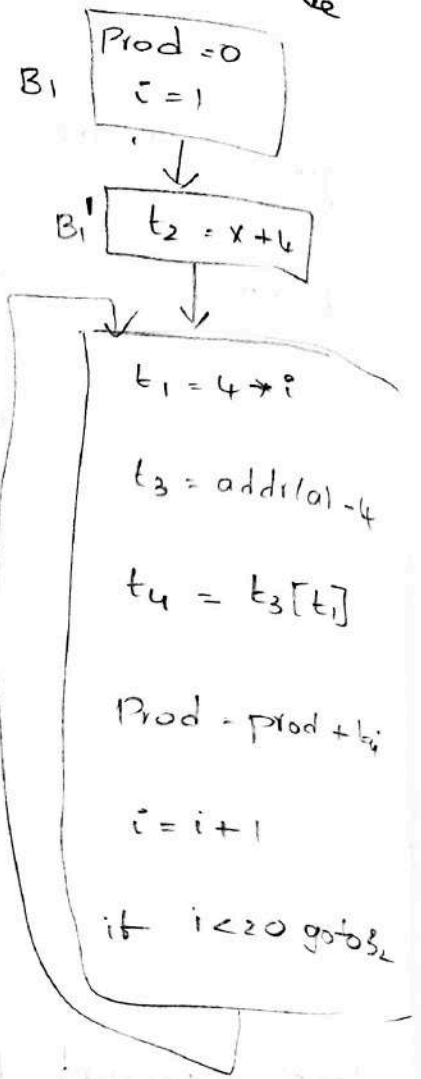
Loop invariant Computation :-

The computations which are independent of the loops can be brought before the loop. This is called as loop invariant computation.

Eg:- Unoptimized Code



Optimized Code



In the above example the loop invariant computations have been brought before the loop & made the block as B_1'

The loop invariant computations is also called as code motion which are not dependent on the computations of the loop

Induction Variable elimination.

Any two variables are said to be induction variable if there is a change in one variable then there will be a corresponding change in another variable

Induction variable elimination is only subjected to the following kinds of ~~statements~~ assignments in the loop

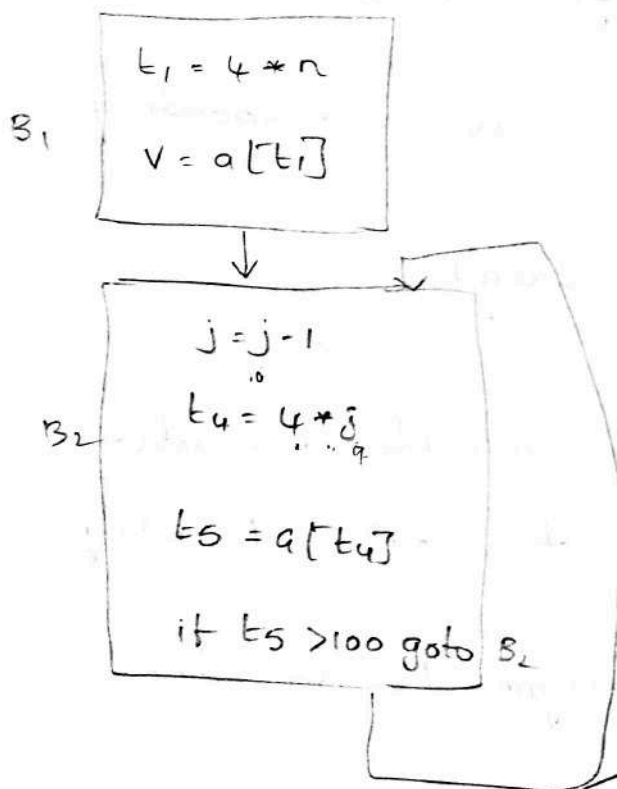
Variable = Variable ^{Operator} Constant

Ex:-1

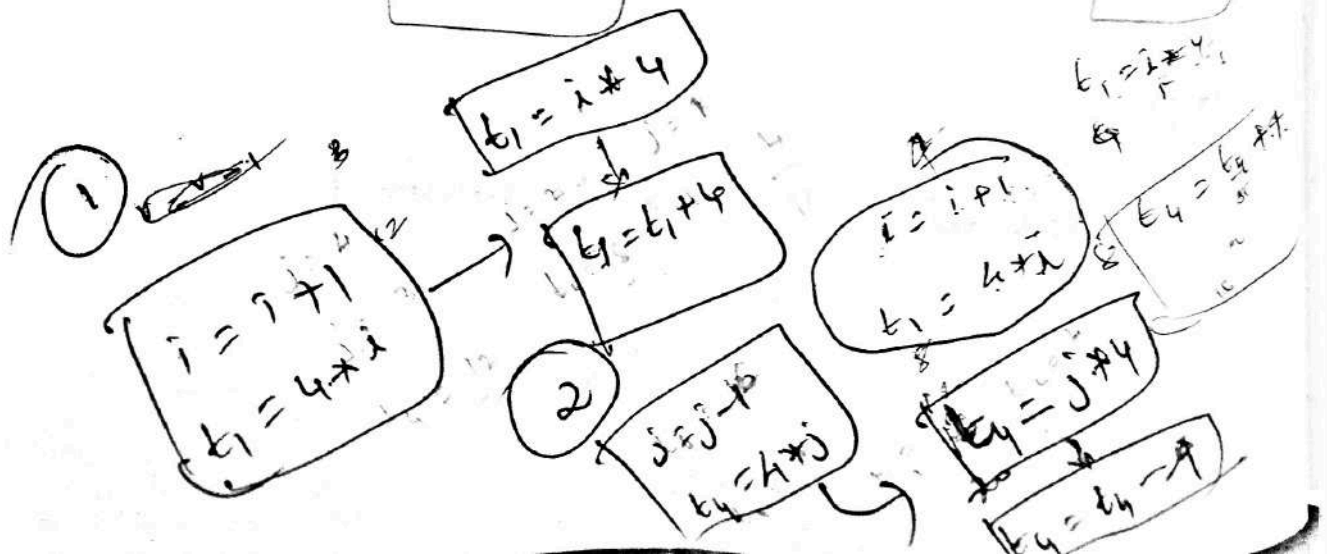
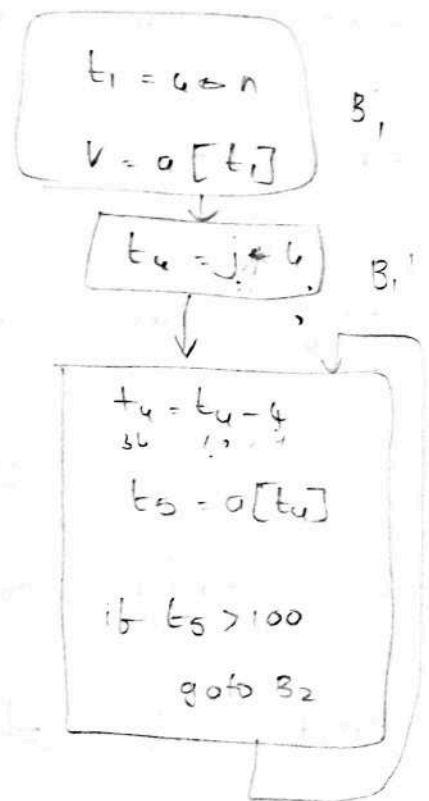
The variable x & y are read to be induction variable if x is incremented then y will also be affected by the variable x .

Ex:-2

Unoptimized code



Optimized code



Induction variable elimination
gives scope for the optimization as
it allows only one variable in the
loop instead of using both

In the above example the
variable J & t_4 are the induction
variables as j is decremented by 1
the t_4 is correspondingly affected
by j

The induction variable is eliminated
by the statement $t_4 = t_4 - 4$ by elimination
 j in the loop & initializing $t_4 = j * 4$
in a separate block B_1'

Reduction in strength:—

It replaces the expensive operation by a equivalent cheaper operation

Ex:-1 x^2 is invariably cheaper to implement as $x * x$ because x^2 has to call an exponential routine which may be more expensive

Ex:-2 $x * 2$ is implemented as $x + x$ because addition is cheaper than multiplication operation.

Ex:-3 floating point division by a constant can be implemented as multiplication by a constant.

Algebraic Simplification:—

$$X = X + 0$$

$$X = X - 0$$

$$X = X * 1$$

$$X = X / 1$$

$$X = X * 1 / 1$$

These kinds of statements has no effect in the program so they can be easily removed from the program.

Machine dependent code Optimization:—

The quality of the target code can be improved by applying the optimization techniques to the target code.

Machine dependent code optimization is classified into

1. Peephole Optimization
2. Constant folding
3. Constant Propagation

Peephole Optimization:-

The performance of the target code is improved by finding the set of instructions & replacing them by a shorter & faster sequence of instruction.

It is classified into

1. Redundant instruction elimination
2. Unreachable code
3. Flow of control optimization
4. Algebraic transformation
5. Use of machine idioms.

Redundant instruction elimination:-

Ex:-1 $\text{MOV } R_0, A \rightarrow \textcircled{1}$

$\text{MOV } A, R_0 \rightarrow \textcircled{2}$

In the above example we can eliminate the instruction $\textcircled{2}$ because whenever the instruction $\textcircled{1}$ is executed it will ensure that the value of A is already in the register R_0

Ex:-2 $\text{MOV } R_0, A \rightarrow \textcircled{1}$

$L_1: \text{MOV } A, R_0 \rightarrow \textcircled{2}$

Here we cannot eliminate any instruction because of the

reference L_1

Unreachable code -

```
// define debug 0
```

```
if (debug == 1) then goto L
```

```
L : print debug information
```

In the above example the label L statement will not be executed at any time. So the Unreachable code with label L will be eliminated in the target code flow of Control Optimization:-

Intermediate code frequently produces 3 kinds of statements. They are

1. Jump to Jump statement
2. Jump to Conditional Jump
3. Conditional Jump to Jump

The unnecessary jumps can be eliminated by the flow of control optimization technique.

Jump to Jump statement :-

Unoptimized Code

goto L1

L1: goto L2

L2:

Optimized Code

goto L2

L2: ---

In the above example there is no possibility to jump to L1 so the statements with label L1 has been eliminated.

Jump to Conditional Jump:-

Unoptimized code

goto L₁

- - -
- - -

L₁: if i < 20 goto L₂

- - -

L₂: - - -

Optimized code

if i < 20 goto L₂

- - -

- - -

L₂: - - -

In the above example there is no possibility to jump to L₁ so the label L₁ is eliminated.

Conditional Jump to Jump:-

Unoptimized code

if i < 20 goto L₁

- - -

L₁: goto L₂

L₂: - - -

Optimized code

if i < 20 goto L₂

- - -

L₂:

In the above example there is no possibility to jump to L_1 so the label L_1 is eliminated.

Algebraic transformation:—

$$X = X + 0$$

$$X = X - 0$$

$$X = X * 1$$

$$X = X / 1$$

$$X = X * 1 / 1$$

These kinds of statements has no effect in the program so they can be easily removed from program.

Use of machine idioms:—

The target machine may have the facility to implement certain specific operation more efficiently.

Ex:— Some machine may support 'auto increment or auto decrement operations.

Then instead of using $i = i + 1$ or

$i = i - 1$ we can use the corresponding instruction respectively.

Constant folding :-

Constant folding is the replacement of runtime computation by a compile time computation.

It is generally done only for the constants

Ex:- $t_1 = (22/7) + x * y$

In the above example $22/7$ can be calculated during compile time rather than at run time.

Constant propagation:-

Propagation means propagating an entity from one statement to the another statement

It is done only for the constants during compile time.

Constant propagation is the replacement of variable by a constant that appears on the right hand side of an assignment.

- Ex:-

Unoptimized Code

$$t_1 = 4$$

$$t_2 = t_1 + 4 - 2$$

Optimized Code

$$t_2 = 4 + 4 - 2$$

Directed acyclic graph:— (DAG)

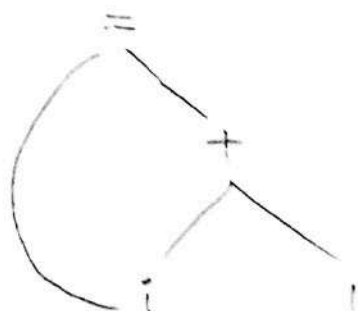
Directed acyclic graph for an expression identifies the common subexpression in the given expression

The duplicate nodes are eliminated in the DAG

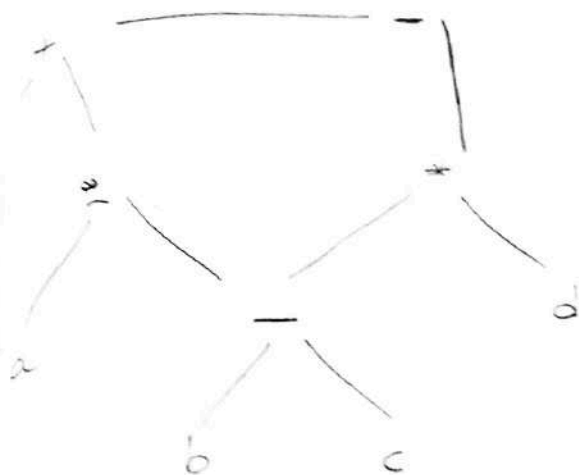
DAG has the node as every subexpression of a given expression where an "interior node" represents the operators & leaf nodes represent operands.

Nodes are optionally labelled by the list of identifiers.

Ex:- $i = i + 1$



Ex:-2 $a + a + (b - c) - (b - c) + d$



Three address code

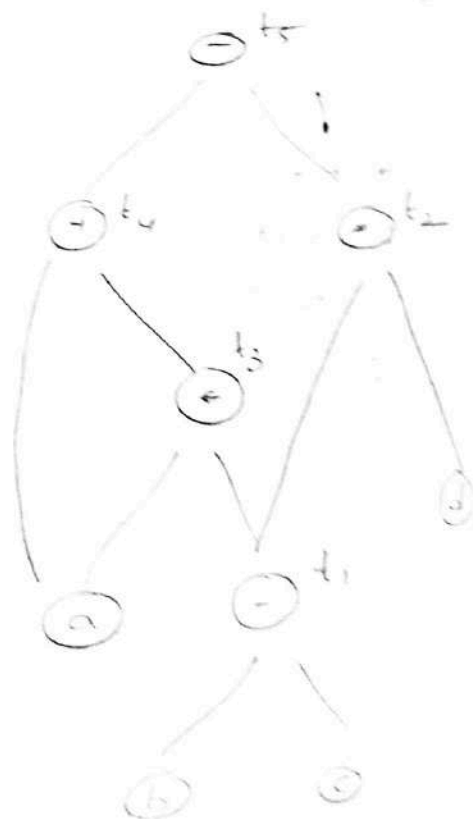
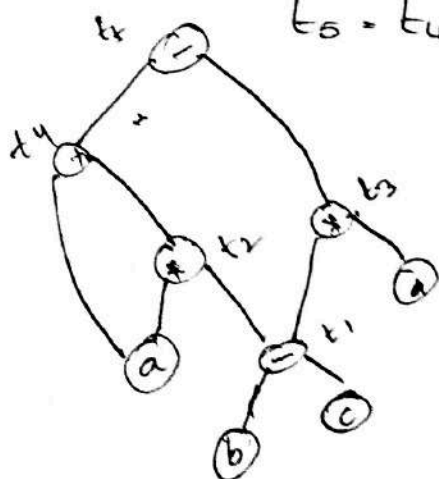
$$t_1 = b - c$$

$$t_2 = t_1 + d$$

$$t_3 = a + t_1$$

$$t_4 = a + t_3$$

$$t_5 = t_4 - t_3$$



$$a + b + (a + b) + (c + d)$$

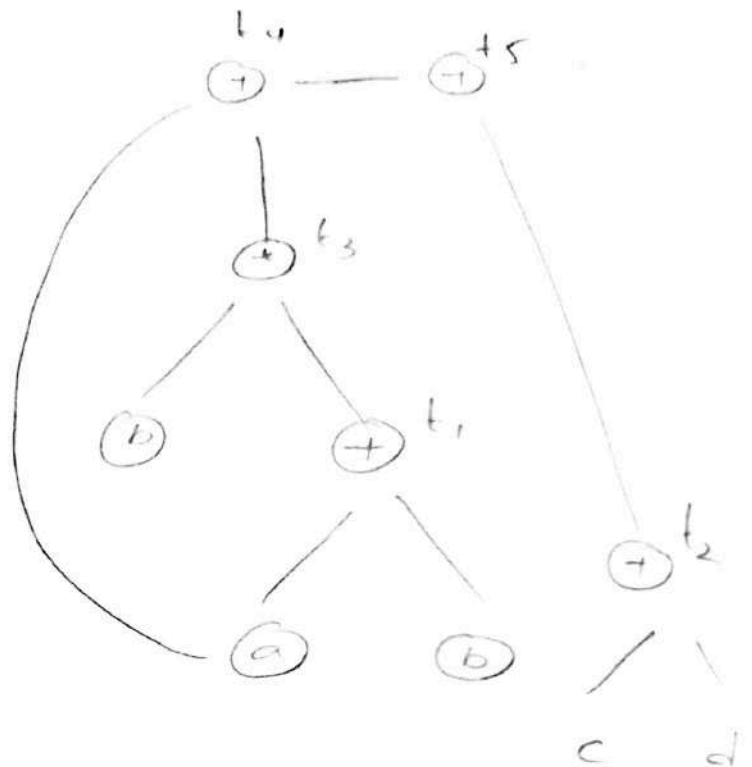
$$t_1 = a + b$$

$$t_2 = c + d$$

$$t_3 = b + t_1$$

$$t_4 = a + t_3$$

$$t_5 = t_4 + t_2$$

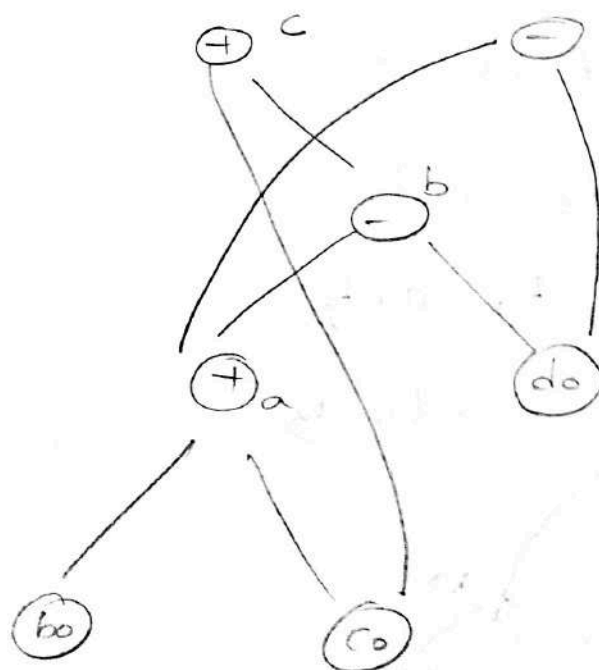


$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = a - a$$

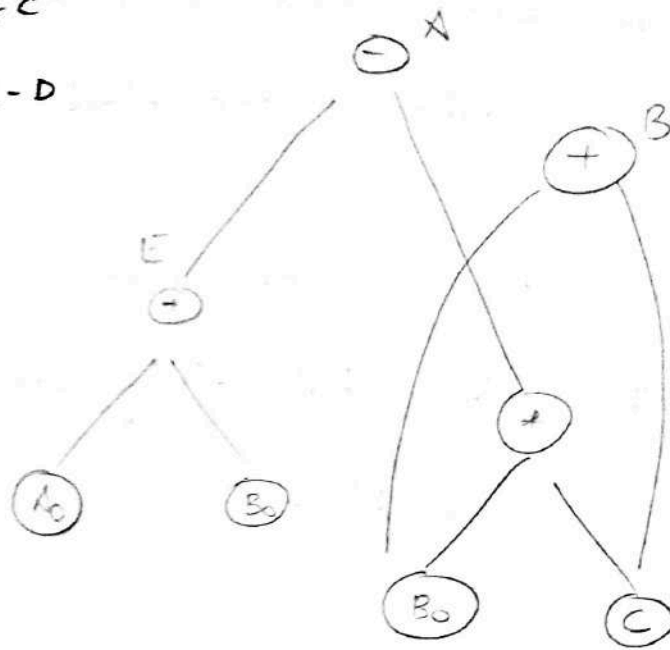


$$D = B + C$$

$$E = A + B$$

$$B = B + C$$

$$A = E - D$$

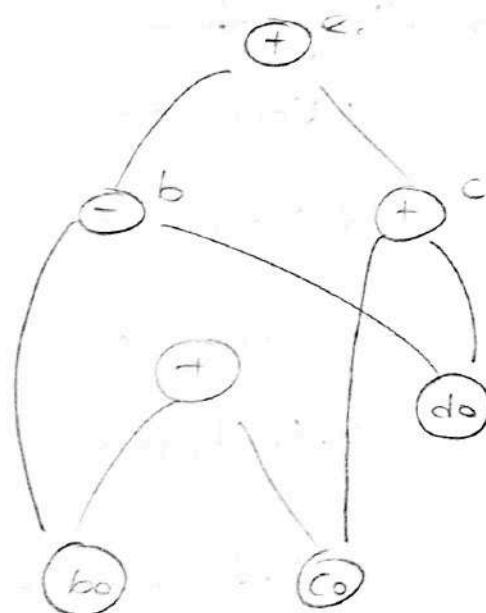


$$a = b + c$$

$$b = b - d$$

$$c = c + d$$

$$e = b + c$$



Application of DAG :-

1. We can detect the common sub expressions in the basic block.
2. We can determine the names computed outside the block, but they are used inside the basic block.
3. We can determine the statement's whose values computed inside the block but they are used outside the block.

Loops in Flowgraph :-

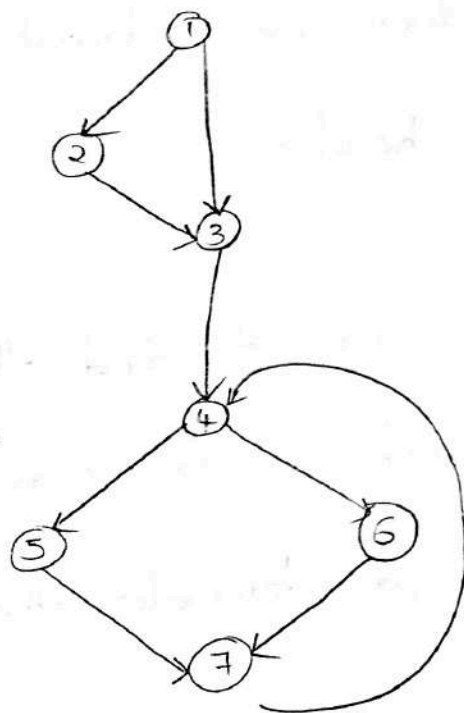
1. Dominator
2. Natural loop
3. Inner loop
4. Pre header
5. Reducible flow graph
6. Non-reducible flow graph

1. Dominator:-

The node 'd' in a flow graph dominates the node 'n' which can be written as $d \text{ dom } n$, if every path from the initial node of the flowgraph to the node n goes through the node d.

Every node in the flowgraph dominates itself and the entry of the loop dominates all the nodes in the loop.

Consider the following flowgraph with the initial node as node 1



The initial node of the flowgraph dominates all the nodes in the flowgraph and the node 4 dominates all the nodes except 1, 2 & 3.

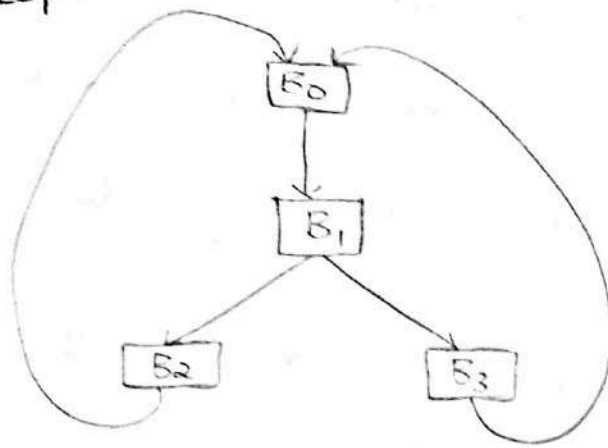
2. Natural loop:-

The two essential properties of the natural loop are

1. A loop must have single entry point called as header. This entry point dominates all the nodes in the loop.
2. There must be atleast one way to iterate the loop i.e. atleast one path back to the header.

A good way to find the loops in the flow graph is to search for the edges whose head dominates their tail.

3 Inner loop:-



Inner loops are the loops that contains no other loops.

When the two loops have the same header, it is difficult to identify which is the inner loop.

If there is a test condition at the end of the block B_1 as if $i == 10$ then goto B_2 .

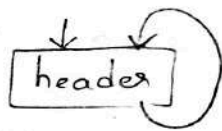
So, probably the loop B_0, B_1, B_3 could be the inner loop. However, we could not be sure without a detailed examination of the code.

If i is almost always 10, it will go around the loop B_0, B_1, B_2 as many times before branching to B_3 .

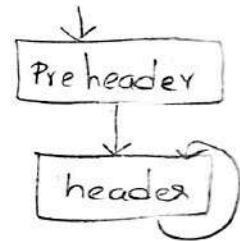
Thus, we shall assume that when two natural loops have the same header then they can be considered and treated as a single loop.

4. Preheader:-

Before transformation



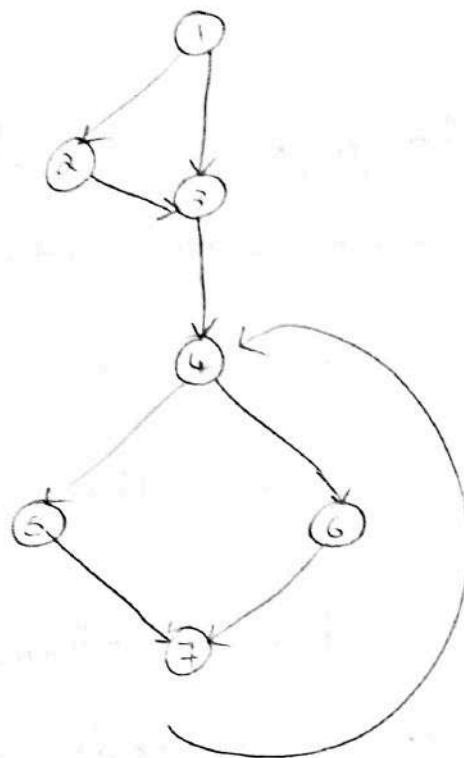
After transformation



Several transformations requires us to move the statements before the loop so, we create a separate block called as the preheader.

The preheader has only the header as successor and all the edges which formally enters through the header instead of entering through preheader.

5. Reducible Flowgraph:-



Flowgraphs are frequently ~~div~~ into the classes of reducible & non reducible flowgraphs.

Flow of control statements such as if-then else, while, for, do-while, goto, break, continue and so on produces the programs whose flowgraphs are always reducible.

The important property of reducible flowgraph is that there should not be any jump into middle of the loop body and the entry of

the loop must be only through the header.

A flowgraph is reducible if we partition the edges into two disjoint groups often called as forward edge and backedge with the following properties.

1. The forward edge forms an acyclic graph in which every node can be reached from the initial node.

2. The back edge consists of the edges whose head dominates their tail.

The above flowgraph is reducible by finding and removing all the back edges in the flowgraph.

If the graph is reducible, then it has to be checked whether the forward edges forms an acyclic graph.

If the back edge $7 \rightarrow 6$ is removed, the remaining graph is acyclic.

Non reducible flowgraph:-



Consider the above flowgraph, whose initial node is node 1.

In this flowgraph, there is no backedge since no head dominates the tail. It could only be reducible if the entire graph were acyclic, but since it is not, the flowgraph is non reducible.

The reason that this flowgraph is non reducible is that the cycle 2 to 3 can be entered at two different nodes, node 2 & node 3 so, it is not reducible.

Object code forms:-

The target program is the output of the code generator which can take the variety of forms such as

1. Absolute machine code
2. Relocatable machine code
3. Assembly code.

The input to the code generator is an optimized intermediate code which may be 3 address code, syntax tree or postfix notation and output is the object code.

If the object code produced is an absolute machine code, then it can be placed in the memory and immediately executed.

The execution of this form of object code is faster than the other two forms.

If the object code produced is an relocatable machine code, then it will allow the "subprograms to be compiled" separately. By using linker & loader, the set of relocatable object code can be linked and loaded for execution.

If the object code produced is an assembly code, then the symbolic instructions and use of macro facilities of assembler helps to "generate the code easily".

Generally, the code generator produces the assembly code as an output as it is easy to produce.

Code generation algorithm

It takes the input as the sequence of 3 address statements that constitutes the basic block.

Code generation algorithm makes use of the descriptors to keep track of the register content and the location of memory. They are

1. Register descriptor
2. Address descriptor

Register descriptor :-

It keeps track of the contents of each register at any instant of time.

Initially it will be empty, as the code generation algorithm progresses it will be set to '0'.

7

Address descriptor :-

It keeps track of the location where the current value of the name can be found.

The location can be register, stack location or any memory address which can be retrieved from the symbol table.

For each 3 address statement of the form $A = B \text{ op } C$ the code generation algorithm performs the following steps:

Step-1: Invoke $\text{GETREG}()$ to determine the location R_0 , where the result of $B \text{ op } C$ should be stored. R_0 can be either register or memory location.

(B)

step-2: Contact the address descriptor for B to determine the current location of B. Take the value of B in register if it is available in both registers and memory location. Move the copy of B into R₀ by using the instruction `MOV R0, B`.

step-3: Generate the instruction op R₀, C where C is the current location that can be fetched from the register if it is available in both registers & memory location. Update the address descriptor of A to indicate A is in register R₀.

step-4: - If the current value of B or C does not have any next use & are not alive after exiting from the block. But they are in registers, change "the registers descriptor to remove the content". (C)

Eg. - $A = B + C$

Statements	Code generator	Register description	Address description
$A = B + C$	mov R0, B	R0 has value of B	
	add R0, C	R0 has value of $A (= B + C)$	

Generate the target code for the following expression

$$W = (A - B) + (A - C) + (A - C) + (A - C)$$

3 address code

$$t_1 = A - B$$

$$t_2 = A - C$$

$$t_3 = t_1 + t_2$$

$$t_4 = t_3 + t_2$$

$$W = t_4 + t_3$$

$$t_1 = A - B$$

$$t_2 = t_1 + A$$

$$t_3 = t_2 - C$$

$$t_4 = t_3 - C$$

$$t_5 = t_4 - C$$

$$t_6 = t_5 - C$$

$$W = t_6$$

statements

Code generator

Register descriptor

Address
descriptor

$$t_1 = A - B$$

mov R0, A

R0 has value of A -

sub R0, B

R0 has value of t_1 , t_1 is in
i.e. (A-B) R0

$$t_2 = A - C$$

mov R1, A

R1 has value of A -

sub R1, C

R1 has value of t_2 is in
 t_2 i.e. (A-C) R1

$$t_3 = t_1 + t_2$$

mov R2, t1

R2 has value of t_1 -

add R2, t2

R2 has value of t_3 t_3 is in
i.e. ($t_1 + t_2$) R2

$$t_4 = t_3 + t_2$$

mov R3, t3

R3 has value of t_3 -

add R3, t2

R3 has value of t_4 t_4 is in
i.e. ($t_3 + t_2$) R3

$$D = t_4 + t_3$$

mov R4, t4

R4 has value of t_4 -

add R4, t3

R4 has value of D D is in
i.e. ($t_4 + t_3$) R4

(E)

* Register allocation and assignment

Instructions involving the registers
operands are faster when compared
to those involving memory operands.

Allocation of registers involves
two states

1. Register allocation

2. Register assignment.

A variable is allocated to a
register when there is a need to
store and use it in the program
execution.

A variable is assigned to a
specific register when the specific
register is chosen for that variable.

The main approach for register allocation & assignment is to assign a specific values into a certain specific registers. So the decision can be made to assign base addresses to one group of registers, top of non time stack to another group of registers, arithmetic computations to another group of registers & so on.

Allocation of global registers:-

The code generation algorithm uses the registers to hold the values for the duration of single basic block but the live variables will be stored at the end of the basic block, because of this number of stores & loads are increased.



It is better to assign the registers to frequently used variables & keep them as global so that this value can be accessed from any block and it will be consistent through out all the basic block.

Usage Count:-

A simple method for determining the savings to be obtained by keeping the variable in a register for the duration of a basic block.

We can say one unit of cost for each reference to the variable if it is in a register

Register assignment for outer loops:-

If an outer loop L_1 contains an inner loop L_2 , the names allocated to a register in L_2 need not be allocated to the register in L_1 .

If the name x is allocated to the register in L_1 but not in L_2 , we must store x before entering the loop L_2 and load x ~~back~~ if we leave the loop L_2 & enter the loop L_1 .

Similar case if we choose to allocate x in loop L_2 but not in L_1 .

Register allocation by graph coloring:-

when there is a need of register for computation but if there is no available register then the contents of used register should be spilled, i.e.

stored in memory allocation in order to free up the register.

The technique called graph coloring is used to spill out the registers R_i allocate.

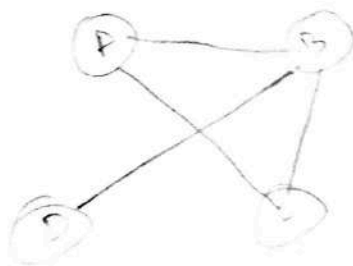
There are two passes in this technique.

1. In the first pass the names used in the intermediate code will become as the names of the registers R_i the intermediate code statements are converted into machine instructions.

2. In the second pass it is used to find an assignment that minimizes the cost of the spills.

by constructing register interference graph where the nodes are the symbolic registers & an edge connects two nodes if one is live at a point where the other is defined. Therefore the nodes have fewer edges have been removed & allocated.

$$A = B + C$$
$$B = C + D$$



*** Issues in the design of code generator:-

The design of the code generator is dependent on the target language and the operating system. There are some issues which should be considered while designing the code generator. They are

1. Input to code generator
2. Target program
3. memory management
4. Instruction selection
5. Register allocation
6. Evaluation order
7. Code generation

Input to the code generator:—

The input to the code generator consist of the intermediate representation of the source program along with the information of symbol table.

There are many choices for intermediate code such as

1. Postfix notation
2. Syntax tree
3. Three address code.

Target program:—

It is the object code format

in the form

Memory management :-

The names in the source program are mapped into the address of the data object in the runtime memory.

The names created as declaration in the program, the type in the declaration determines the amount of storage required for that declared name.

The symbol table information gives the relative address for the names in the program.

Instruction selection :-

The difficulty in instruction selection is dependent on the nature of the instruction set of the target machine.

Two important factors that should be considered while selecting the instruction set are

1. Uniformity of the instruction
2. The completeness of the instruction set.

Target machine must support each data types in an uniform manner.

Instruction speed and use of machine idioms are also the important factor that makes difficult to select the instruction set.

Register allocation:—

Register allocation

&

assignment.

Last topic

Evaluation order:-

The order in which the computations are to be performed can affect the efficiency of the target code.

Some computations order requires fewer registers to hold the intermediate result than the other order.

The problem can be solved if the target program generates the code as they appear in the program.

Code generation:-

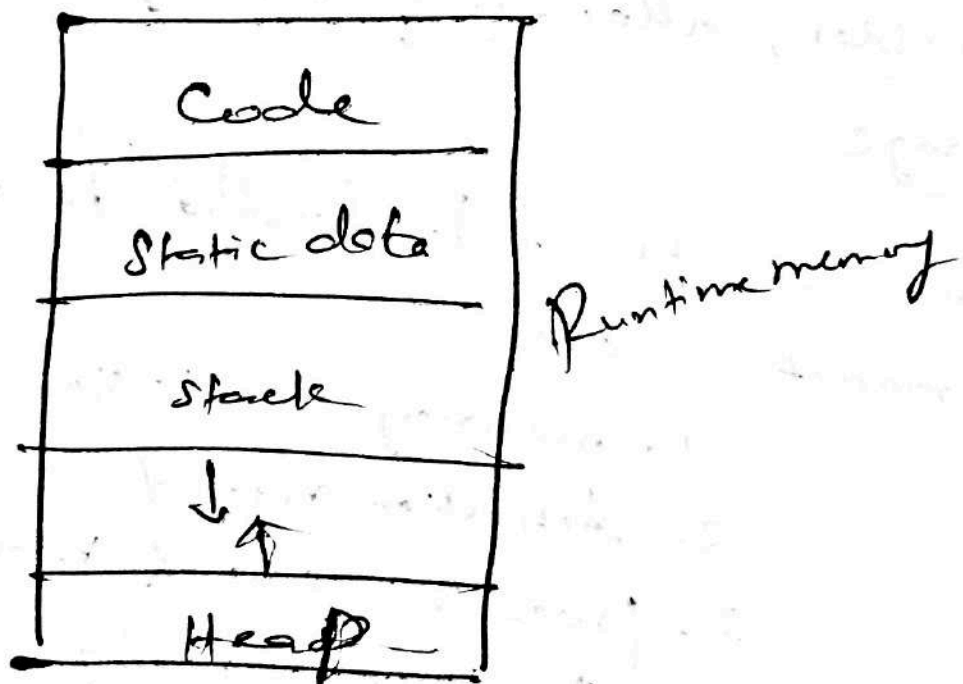
The most important design issue is that the code generator must generate the correct code.

The main design goal is to design the code generator which is easy to implement, test & maintain.

Storage Organization:

Runtime storage memory can be subdivided as follows.

1. Generated target code
2. Data objects
3. Counterpart of Control stack
to keep track of procedure activation.



→ The size of generated target code is fixed at compile time, so compiler can place it in a statistically determined area.

→ Similarly the size of some of data objects may also be known at compile time and these too can be placed in statistically determined area as shown in above fig.

→ When a call occurs, execution of current activation is interrupted and information about status of the machine such as program counter and machine registers is saved on the stack. Before the control is passed to the calling program/procedure, when the control returns from the call, this activation can be restarted after restoring the values of relevant registers and setting the PC to point immediately after the call.

→ The data objects whose lifetime are contained in that of activation can be allocated on the stack along with other information with the activation.

→ A separate area of runtime memory called heap, holds all other information.

→ Languages in which the lifetime of activation cannot be represented, we use heap to store the activation information.

→ The way in which data is allocated and deallocated on stack makes it cheaper to place data on stack than on the heap.

→ The size of stack and heap can change as the program executes.

Activation of Procedures:

- Each execution of a procedure is referred to as an activation of the procedure.
- If the procedure is recursive there will be several activation for the procedures which may be alive at the same time.
- The lifetime of an activation is the sequence of steps present in the execution of procedure.

Activation tree:

- If the program control flows in a sequential manner and a procedure is called, its control is transferred to called procedure.
- When the called procedure is executed, it returns the control to the caller.

This type of control flow makes it easier to represent a series of activations in the form of a tree known as activation tree.

→ The activation tree is built such that

* Each node represents an activation of a procedure

* The root represents the activation of main program.

* The node for 'a' is the parent of node for 'b' if and only if

Control flows from activation 'a' to 'b'

* The node for 'a' is left to the node of 'b' if and only if the lifetime of 'a' occurs before the life time of 'b'

```
Ex: int main()
{
```

```
    A();
```

```
}
```

```

A()
{
    int b = B();
    int c = C();
    int a = b + c;
    return a;
}

```

```

B()
{
    // ...
    return r1;
}

```

```

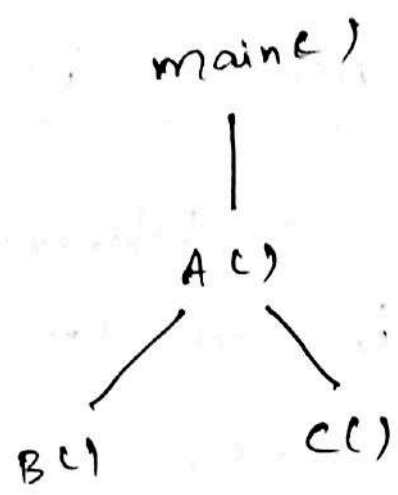
C()
{
    // ...
    return r2;
}

```

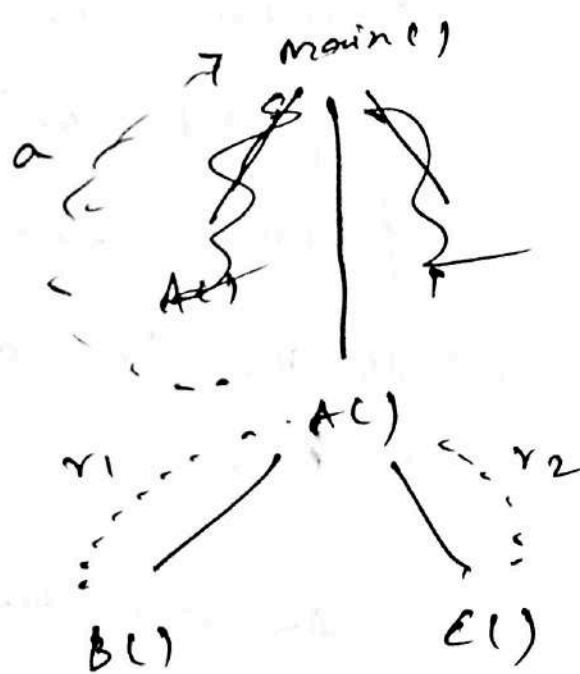
Control Stack:

→ A stack representing procedure calls, returns and flow of control is

called a Control Stack (or) return stack.



Activation tree



Activation tree with return.

→ Control stack is used to keep track of line procedure activation.

Activation Record:

→ Whenever a procedure is executed its activation record is stored on the stack.

→ When a procedure calls another procedure, the execution of called is suspended until the called procedure finishes execution.

→ At this time activation record of called procedure is stored on the stack.

→ An activation record contains collection of fields as shown below

Return value
Actual Parameter
optimal Control link
optimal access link
Saved machine status
Local data
Temporaries

General Activation record

Temporaries:

→ Stores temporary and intermediate values of an expression during its evaluation.

Local data:

Holds data that is local to the execution of the procedure.

Machine status:

→ Holds information about status of machine just before the function call.
 → Stores value of pc, m/c register etc before the procedure is called.

Access link :

Refers to non local data held in other activation record.

Control link :

Points to the activation record of called ^{procedure}. That is it stores the address of the activation record of the called procedure.

Actual parameter :

Stores the actual parameter that is parameter that are used to send i/p to called procedure.

Return value : Stores the return value

used by called procedure to return a value to calling procedure.

Stack Storage allocation :

→ In stack allocation, storage is organized as a stack and activation records are pushed and popped as activation begins and ends respectively.