Total No. of Pages 1
VI-SEMESTER
MID SEMESTER EXAMINATION
Roll No.....
B.Tech (CO)
March- 2023

CO-302 Compiler Design

Time: 1:30 Hours
Max. Marks: 20
Note: Attempt all questions, assume suitable missing data if any

---

Total No. of Pages 1
Roll No.....
VI-SEMESTER
MID SEMESTER EXAMINATION
B.Tech.(CO)
March- 2020

CO-302 Compiler Design

Time: 1:30 Hours
Max. Marks: 30
Note: Attempt all questions

Total No. of Pages 1
Roll No.....
VI-SEMESTER
MID SEMESTER EXAMINATION
B.Tech.(CO)
March- 2019

CO-302 Compiler Design

Time: 1:30 Hours
Max. Marks: 30
Note:Attempt all questions

Total No. of Pages 01
Roll No. .....
VI- SEMESTER
MID SEMESTER EXAMINATION
B.Tech [CO]
MARCH-2018

CO-302: Compiler Design

Time: 1:30 Hours
Max. Marks: 30
Note : 1) Attempt all questions.
2) All parts of a question must be attempted together.
3) Assume suitable missing data, if any.

Total No. of Pages 02
Roll No.....
VI-SEMESTER
B.Tech.(CO)
END SEMESTER EXAMINATION
May- 2023

CO302 Compiler Design

Time: 3:00 Hours
Max. Marks: 40
Note: Attempt any five questions

Total Number of Pages 1
Roll No. .....

SEVENTH SEMESTER

B.E.(COE)

MID SEMESTER EXAMINATION

SEPTEMBER-2010

COE- 401 COMPLIER & TRANSLATION DESIGN

Time: 1 Hour 30 Minutes

Max. Marks : 20

Note : Answer ALL questions.
Assume suitable missing data, if any.

Total No. of Pages 2
Roll No.....
VI-SEMESTER
END SEMESTER EXAMINATION
B.Tech.(CO)
May- 2019

CO-302 Compiler Design

Time: 3:00 Hours
Max. Marks: 40
Note: Attempt any five questions

Total No. of Pages 2
Roll No.....
VI-SEMESTER
END SEMESTER EXAMINATION
B.Tech.(CO)
May- 2018

CO-302 Compiler Design

Time: 3:00 Hours
Max. Marks: 40
Note: Attempt any five questions

Total No. of Pages 2
Roll No.....
VI-SEMESTER
END SEMESTER EXAM(Supplementary)
B.Tech.(COE)
Aug- 2018

CO-302 Compiler Design

Time: 3:00 Hours
Max. Marks: 40
Note:Attempt any Five questions

Total No. of Pages 2
Roll No.....
VI-SEMESTER
END SEMESTER EXAMINATION (Supl)
B.Tech.(CO)
Sept- 2019

CO-302 Compiler Design

Time: 3:00 Hours
Max. Marks: 40
Note: Attempt any five questions

Total No. of Pages 01
Roll No. .....
SIXTH SEMESTER
END SEMESTER EXAMINATION
B.Tech[CO/SE]
MAY-2018

CO/SE-312: Compiler Design

Time: 3:00 Hours
Max. Marks: 70
Note : Attempt any five questions
All parts of a question must be attempted together.

Total No. of Pages 02
Roll No. 2K 17/CO/045
SIXTH SEMESTER
END SEMESTER EXAMINATION
B.Tech[CO/SE]
MAY-2017

CO/SE-312: Compiler Design

Time: 3:00 Hours
Max. Marks: 70
Note :1) Q1 is compulsory.
2) Attempt any four questions from Q2 to Q7.

UNIT- 1 : INTRODUCTION

B. How is boot strapping of a compiler is done to a second machine? And also explain the function of each phase of compiler with suitable example. 2023 E / [4] [CO#1]

Page No.
Date

B. Explain different phases of compiler with suitable example. 2019M [3]

Explain different phases of compiler for the source code Position =initial+ rate *45 2020M [4]

The compilation process is divided into two parts:

Q2. a) Describe analysis and synthesis phases of Complier? 2018E [7]

① ANALYSIS PHASE (front end)

It is a machine independent and language dependent phase which breaks up the source program into constituent pieces and imposes grammatical structure to them.

② SYNTHESIS PHASE (back end)

It is a machine dependant and language independant phase which constructs the desired target program from the intermediate representation and information in symbol table

Q4.a) Describe machine dependent and independent phases of Compiler? 2017E [7]

Ⓐ M/C dependant

LEXICAL ANALYZER

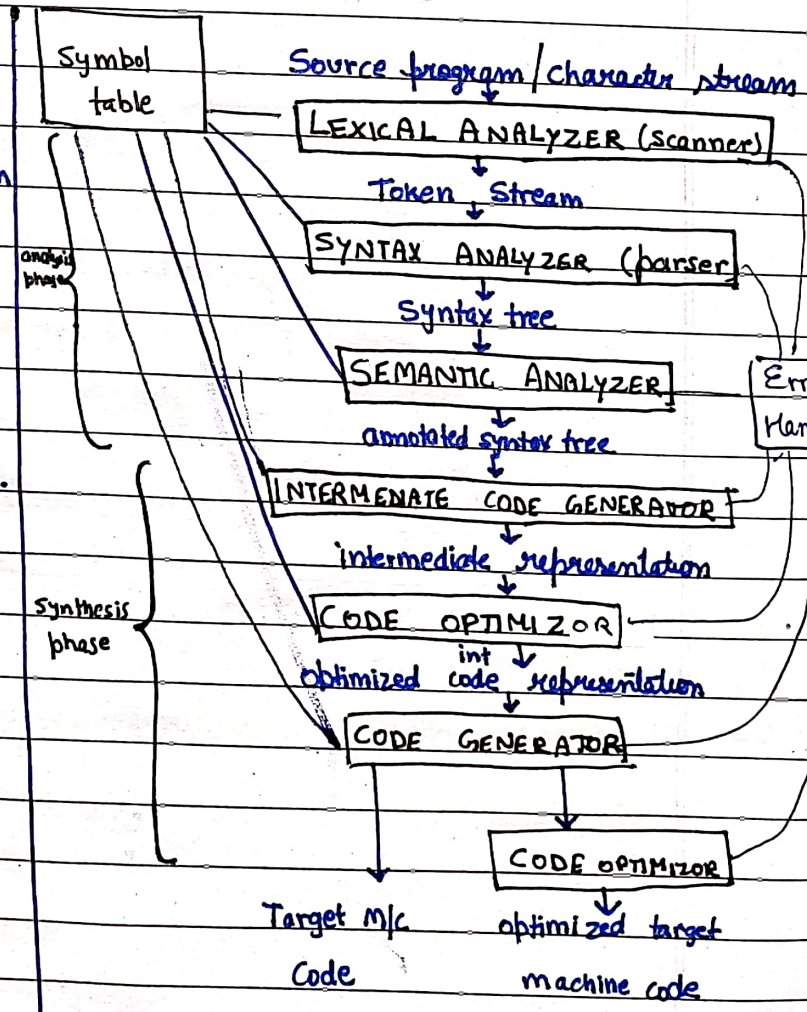
It is scanner which reads the prog. and converts it into lexemes, a stream of lexemes into a stream of tokens. Tokens are defined by regular exp which are understood by lexical analyzer

It also removes white space, comments.

SYNTAX ANALYZER

It is parser which constructs parse tree. It uses productions of CFG to construct parse tree.

It reads all tokens one by one.



Page No.
Date

SEMANTIC ANALYZER

It verifies whether the parse tree is meaningful or not.
It produces a verified / annotated parse tree.
(B) M/c Independant

INTERMEDIATE - CODE GENERATOR

It generates intermediate code, that is a form which can be readily understood by machine.
Eg.) Three address code etc.

CODE OPTIMIZER

It transforms the code so that it consumes fewer resources and produces more speed. (The code is transformed, not altered)

It can be : i.) M/c dependant
ii.) M/c independent

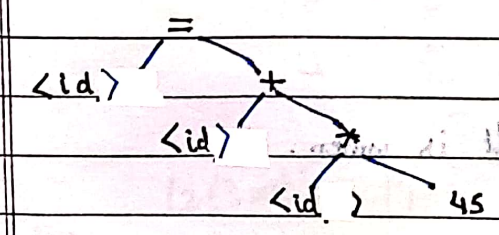
TARGET CODE GENERATOR

It writes a code that the m/c can understand and also register allocation, instruction selection etc.
The o/p is dependant on the type of assembler.

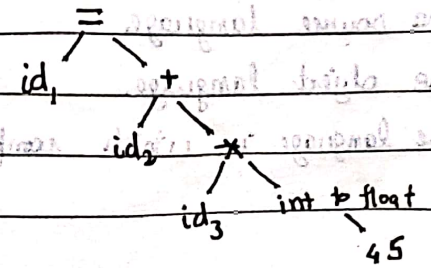
Eg.) Position = initial + rate * 45

i.) Lexical Analyzer
<id, 1> <assign op> <id, 2> <add op> <id, 3> <mult op> <int const 45>

ii.) Syntax Analyzer



iii.) Semantic Analyzer



Page No.
Date

iv.) Intermediate code generator

t_1 = int to float (45)
$t_2 = id3 * t_1$
$t_3 = id_3 * t_2$
$id_1 = t_3$

v.) Code Optimizer

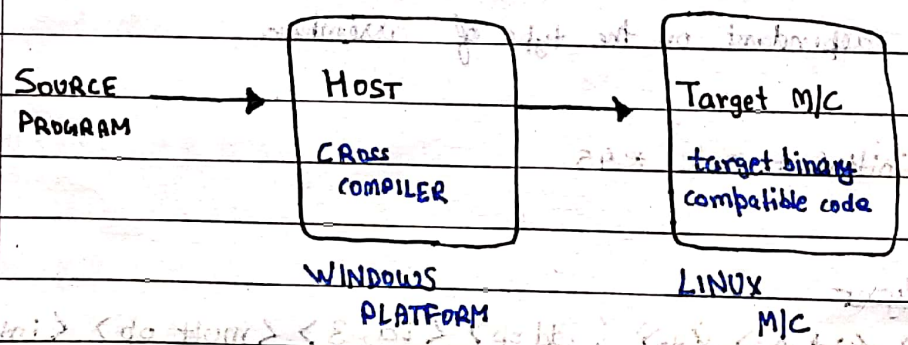
$t_1 = id_3 * 45.0$
$id_1 = id2 + t_1$

vi.) Code Generator

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

B. What is cross compiler? 2020M

A cross compiler is a compiler that runs on one machine and produces object code for another machine.

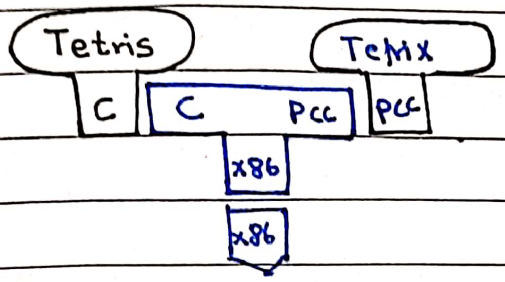


The cross-compiler is used to implement the compiler, which is categorized by three languages:

1. The source language
2. The object language
3. The language in which compiler itself is written.

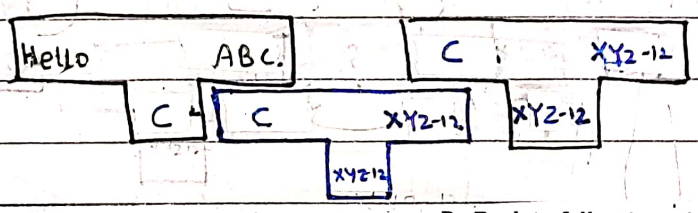
Page No.
Date

Eg)



Q.No. 1

What is cross compiler? Create a cross compiler for ABC using C compiler, written in XYZ-12, producing code in XYZ-12 and a ABC language producing code for text formatter, HELLO written in C. 2023M [4][CO#1]



[b] Boot strapping. 2010M

Bootstrapping is an arrangement which refers to writing a compiler in the same language it intends to compile. It is the process of using a compiler written in a particular lang to compile a new version of the compiler written in same lang.

$^{L}C^{A}_{S} \rightarrow {}^{S}C^{A}_{A} \rightarrow {}^{L}C^{A}_{A}$

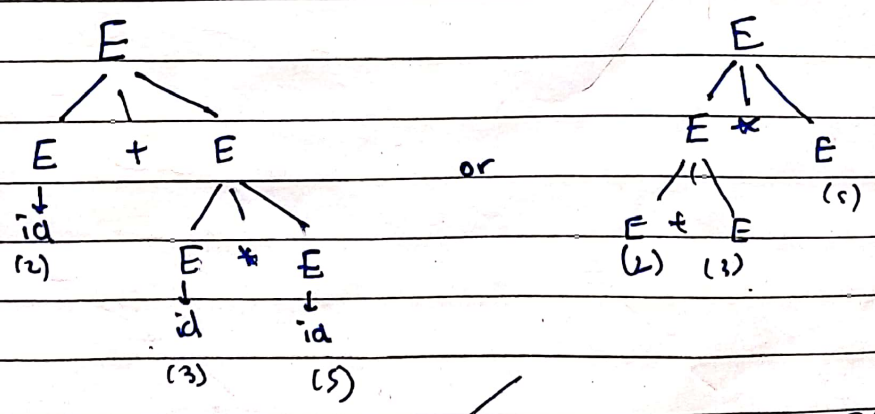
B. Explain following with suitable examples 2019E
a. Ambiguous grammar and cross compiler

B. Explain following with suitable examples
a. Ambiguous grammar and cross compiler 2019S

AMBIGUOUS GRAMMAR

An ambiguous grammar is a context free grammar for which there exists a string that can have more than one leftmost derivation or parse tree. or more than one rightmost derivation or parse tree.

Eg.) w = id + id * id



(2+(3*5)) = 17 ✓

(2+3)*5 = 25 ✗

B. Design a context free grammar(CFG) for the language L= $\{x^c y^c z^d u^d$ for c>=1,d>=1} 2019M [5]

Page No.
Date

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

$N = \{S, A, B\}$
$T = \{xyzu\}$
$S = \{S\}$

P:

$$\left\{\begin{array}{l} S \rightarrow AB \\ A \rightarrow xAy \mid xy \\ B \rightarrow zBu \mid zu \end{array}\right\}$$

refer. TOC

[c] Regular Expression

B. Design a DFA with input alphabet {a,b} for the language [2x2=4]

2018E

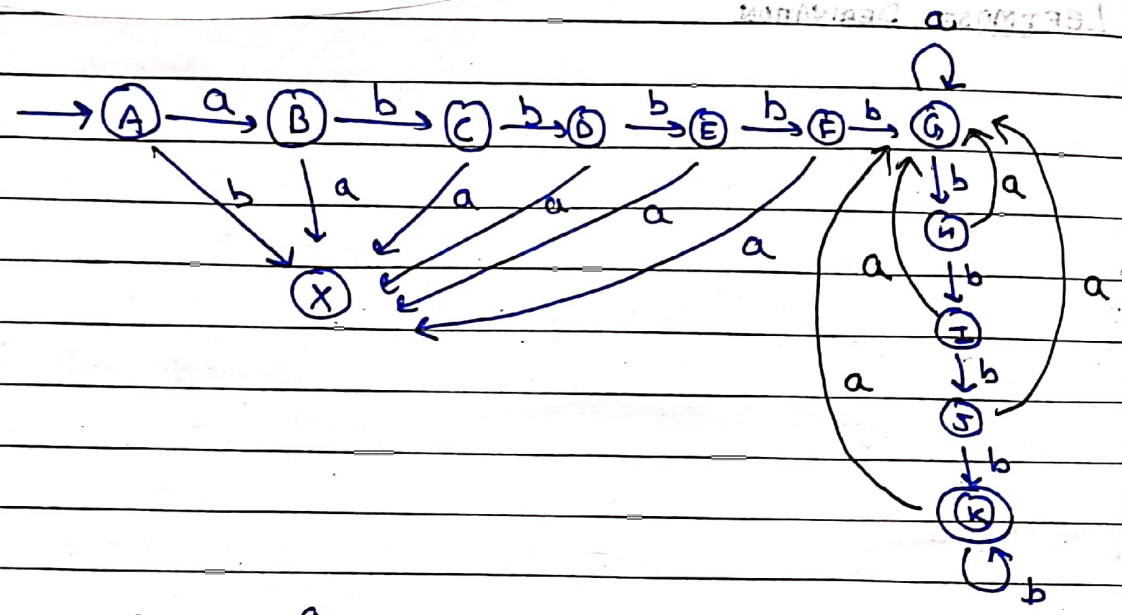
i) $L = \{w \in (a,b)^* : n_b(w) \bmod 3 > 1\}$

Where $n_b(w)$ is number of b's in w

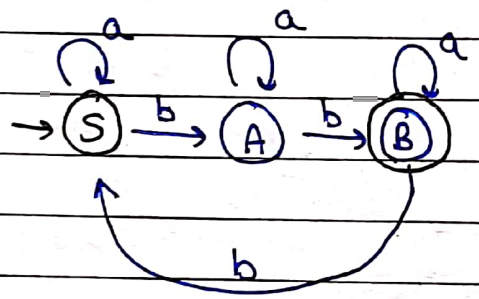
ii) $L = \{ab^5 w b^4 : w \in (a,b)^*\}$

Page No.
Date

ii)

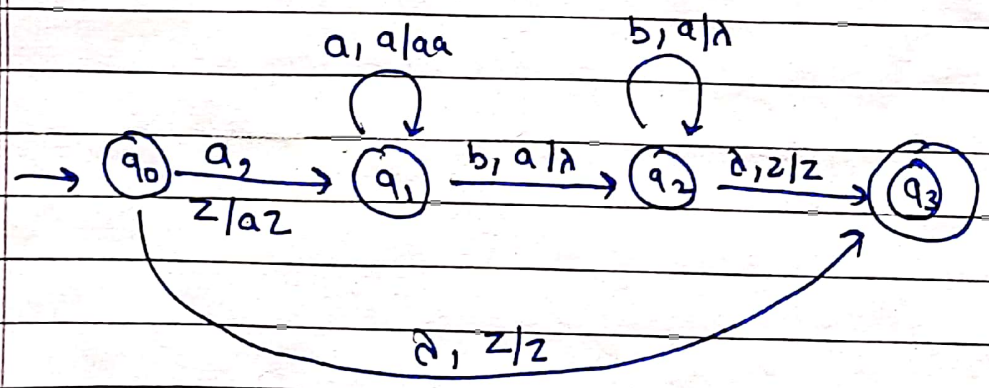


i)



b. Design a PDA accepting language $L = \{a^n b^n \mid n \geq 0\}$ 2018M (4)

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



Page No.
Date

Q.No. 1 2018S

A. Construct DFA accepting strings of binary digits which are divisible by 3. [3]

let DFA be $M = \{Q, \Sigma, F, S, \delta\}$

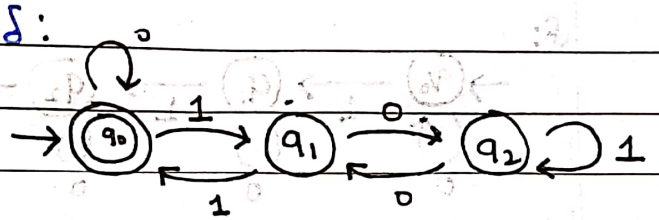
$Q : \{q_0, q_1, q_2\}$

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

$F : \{q_0\}$

$S : \{q_0\}$

δ :



b) Design a DFA (with input alphabet {0,1}) that accepts [7]
i) The set of strings where number of 0's is multiple of 3.
ii) $L = \{(0\ 1)^i 1^{2j} / i>=1, j>=1\}$ 2017E

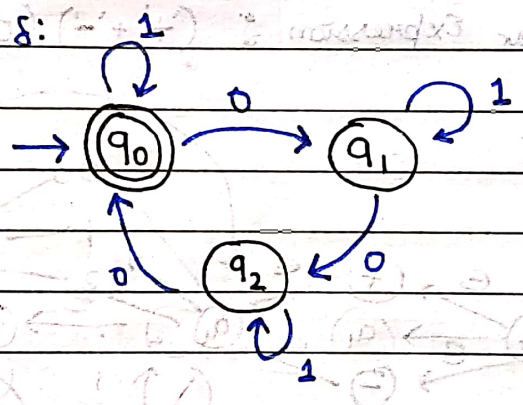
i.) $Q : \{q_0, q_1, q_2\}$

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

$F : \{q_0\}$

$S : \{q_0\}$

δ :



ii.) $L = \{ (01)^i 1^{2j} \mid i>=1, j>=1 \} = \{ 0111, 010111, 01011111 \ldots \}$

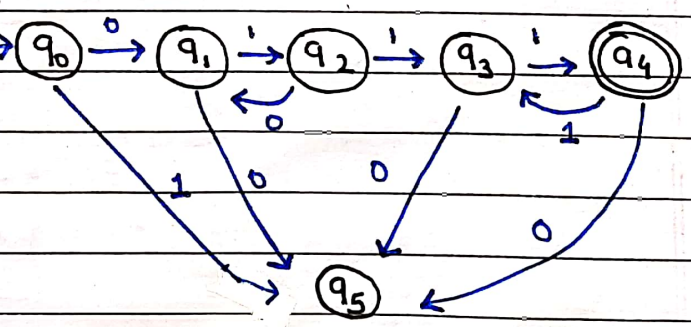
$Q : \{q_0, q_1, q_2, q_3, q_4, q_5\}$

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

$F : \{q_4\}$

$S : \{q_0\}$

δ :



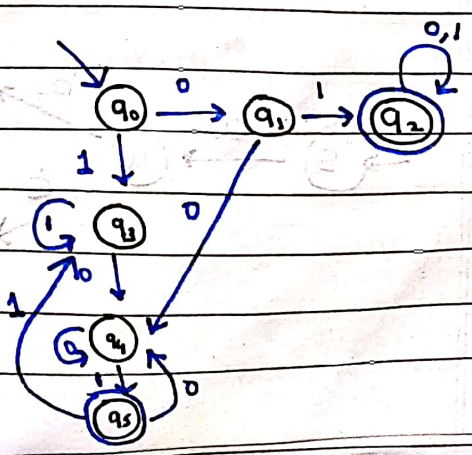
b) Design a DFA (with input alphabet {0,1}) that accepts 2018E [7]
i) The set of strings which either starts with 01 or end with 01.
ii) The set of strings containing four 1's in every string.

i.) $Q : \{q_0, q_1, q_2, q_3, q_4, q_5\}$

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

$F : \{q_2, q_5\}$

$S : \{q_0\}$



Page No.
Date

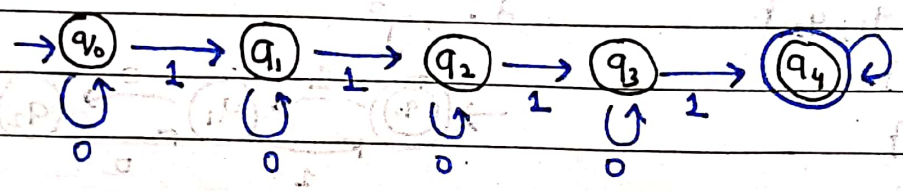
(ii) $Q: \{q_0, q_1, q_2, q_3, q_4\}$

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

$q_0: q_0$

$q_f: \{q_4\}$

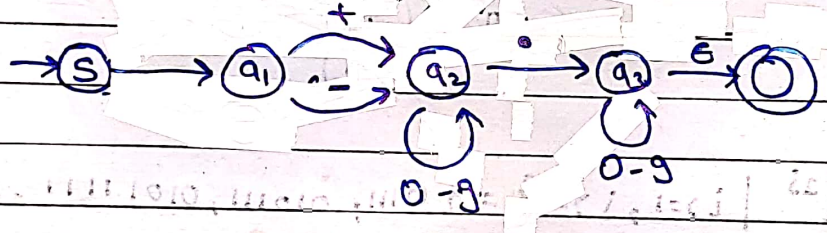
δ:



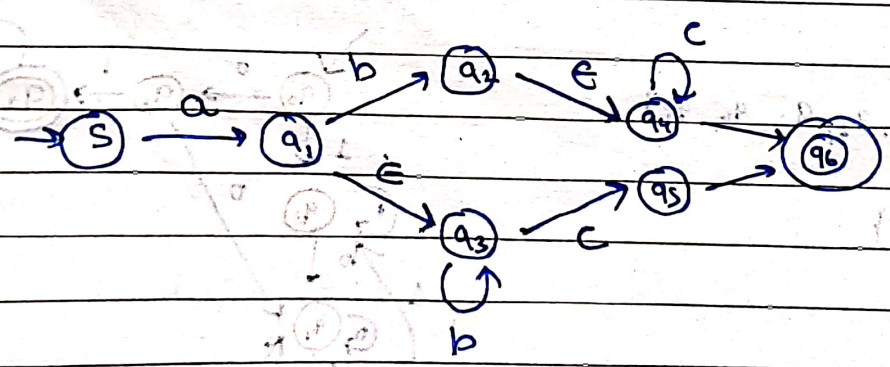
b. Write a regular expression for real number and draw FA for that regular expression? 2018M (3)

Regular Expression : $(+ + -)(0+1+2+3+\dots+9)^*(.)(0+1+2\dots+9)^*$

FA :



1 Convert the following regular expression into minimal state equivalent DFA.
$(ab^*c)|(abc^*)$ 5



Unit 2: Syntax Analysis

Page No.
Date

c. How a compiler differentiate between keywords and identifiers? 2018M (3)

In the lexical analysis phase, the compiler distinguishes keywords and identifiers by comparing scanned tokens with a list of reserved keywords for matches stored in a keyword table. While identifiers are managed using symbol table, checking for existing entries and adding new ones as needed.

Q.No. 2

a. Rectify the problems with the following grammers to make them suitable for LL(1) parsing: 2018M (4)

i) S → Sa/Sb/AB/C/DEF

ii) A→da/acB ; B→abB/daA/Af

i) we have to remove left recursion.

∴ $S \rightarrow ABS' / CS' / DEFS'$

$S' \rightarrow aS' / bS' / \epsilon$

ii) Given

$A \rightarrow da / acB$

$B \rightarrow abB / daA / Af$

it can be written as.

$A \rightarrow da / acB$

$B \rightarrow abB / dada / daacB / daf / acBf$

now B exhibits left factoring.

$A \rightarrow da / acB$

$B \rightarrow abB / Cda / CacB / Cf / acBf$

$C \rightarrow da$

?

Page No.
Date

2 A production of the form $A \rightarrow A\alpha$ is said to be left recursive. Similarly a production of the form $B \rightarrow \beta B$ is said to be right recursive. Show that any grammer that contains both left & right recursive productions with the same left hand side symbol must be ambiguous. 2012M 3

Let us consider the grammar

$S \rightarrow A|B$

$A \rightarrow Aa / B / \epsilon$

$B \rightarrow bB / A / \epsilon$

$\therefore$ we can derive "bbaa" as

1.) $A \rightarrow B \rightarrow bB \rightarrow bbB \rightarrow bbA \rightarrow bbaA \rightarrow bbaa$

2.) $A \rightarrow Aa \rightarrow Aaa \rightarrow Baa \rightarrow Bbaa \rightarrow bbaa$

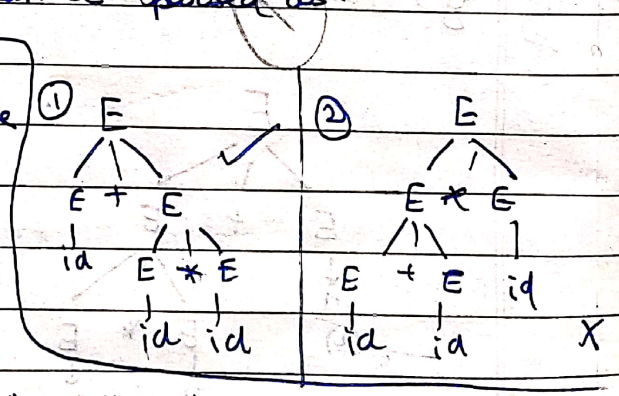
Hence, the grammar is ambiguous as more than one way to generate a given string exists.

Q.No. 1

a. Give an example of grammer to show that determinism cannot eliminate ambiguity? 2018M (2)

Ex.) $E \rightarrow E+E / E*E / id$

It is deterministic but also ambiguous at the same time as $\omega = id + (id*id)$ can be parsed as

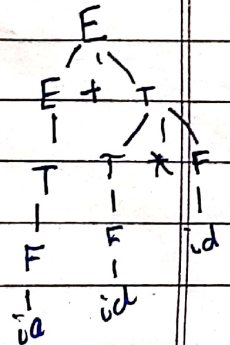


A non ambiguous grammar may look like

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

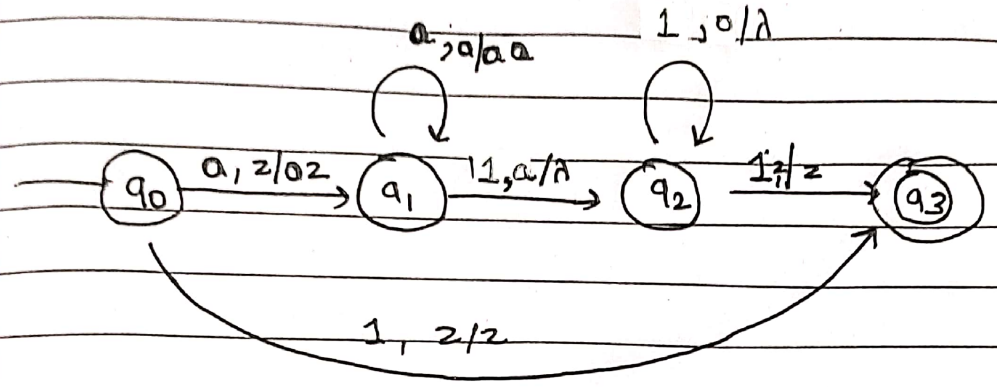
$F \rightarrow id$



This grammar takes into consideration the procedure as well as associativity.

Page No.
Date

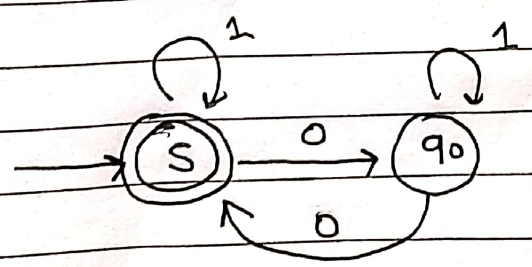
B. Explain importance of PDA in compiler design and Design a PDA for Language L={ $0^n 1^n 1$ for n>=0}. 20185 [5]



Q.No, 5

A. What is the use of FA in lexical analysis? Design a DFA for strings over {0, 1} having an even number of 0's and any no. of 1's. 20185 [3]

Finite automata in lexical analysis represents the language's structure using regular expressions, facilitating efficient tokenization of input strings into tokens by recognizing patterns and transitioning between states based on i/p symbols



DFA

3 Consider the following CFG:

$S \rightarrow a|\wedge|(T)$

$T \rightarrow T,s|s$

2010 M

[a] Compute the operator precedence relations for this grammer. 3

[b] Eliminate left recursion from the grammer. 1

[c] Show the steps of a Top-down passer without back tracking i.e. predictive passer for the string (((a,a), ∧, (a)), a) 4

Page No.

Date

Page No.
Date

ii) Expain common prefix problem with example?

It is also called left factoring, it is a problem as it is not clear which production to choose for NT because multiple productions begin with the same T/NT/lookahead.

It gives problem ~~in~~ backtracking

$$\left.\begin{array}{l} A \rightarrow \alpha \beta_1 \\ A \rightarrow \alpha \beta_2 \end{array}\right\}$$

it is removed by rewriting prod^n rules as

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2$

Eg.) $S \rightarrow ict s$

$S \rightarrow ict S e s$

$S \rightarrow a$

$C \rightarrow d$

after Elemination

$S \rightarrow ict S S'$

$S' \rightarrow eS / \epsilon$

$S \rightarrow a$

$C \rightarrow d$

Q1. 2017E [14]

a) (0* / 1)* & (0 /1*)* are equivalent or not?
b) G= {N, { (,) } , { S-> SS, S-> (S), S-> ε },S} is a CFG to produce_____?
c) Recursive descent parser is a predictive or non-predictive parser?
d) All CFL's are regular languages or not?
e) YACC is a tool for _____?
f) _____ is a solution for common prefix problem?
g) Number of states in SLR and CLR are same or not?
h) _____ is used to identify common sub expressions?
i) Type conversion automatically done by complier is called _____?
j) Folding is the technique for _____?
k) Syntax tree is the form of _____?
l) A group of statements with single entry and single exit is called _____? (with respect to code generation)
m) Two adjacent transposed characters is an example of _____ error?
n) _____ is a hashing technique in which there is no limit on number of entries that can be made to the symbol table.

a.) yes
b.) 1
c.) predictive parser
d.) No
e.) generating parser
f.) Trie
g.) No
h.) global optimization tech.
i.) implicit TC
j.) optimizing exp
k.) parse tree
l.) basic block
m.) typographical
n.) Linear Probing

B What is the role of context free grammar (CFG) in compiler design? Construct CFG for Language $L=\{0^a 1^b \mid a \neq b\}$ and Eliminate Left recursion from following grammar 2023E [4] [CO#1]

S→A
A→ Ad|Ae|aB|aC
B→bBC| f
C→g

[2]

B. Eliminate Left recursion from following grammar 2019E
S→A
A→ Ad|Ae|aB|aC
B→bBC| f
C→g

[5] 2020M
recursion from following grammar
S→A
A→Ad | Ae| aB| aC
B→bBC | f
C→g

Its left recursive as it contains

$A \to Ad \mid Ae \mid aB \mid aC$

we replace these with

$A \to aBD \mid aCD$

$D \to dD \mid eD \mid \epsilon$

∴ new resultant grammar.

$S \to A$

$A \to aBD \mid aCD$

$D \to dD \mid eD \mid \epsilon$

$B \to bBC \mid f$

$C \to g$

Construct a CFG for language $L=\{0^a 1^b \mid a \neq b\}$ and eliminate Left

$P =$

$S \to AT \mid TB$

$T \to 0T1 \mid \epsilon$

$A \to 0A \mid 0$

$B \to 1B \mid 1$

$G = \{N, T, P, S\}$

$S = \{S\}$

$N = \{S, T, A, B\}$

$T = \{0, 1\}$

remove left recursion from following CFG : 2023M [4][CO#2]
S → Bb | a
B →Be |Sd |e

It is left left recursive as it contains indirect and direct LR

① remove indirect calls.

$S \to Bb \mid a$

$B \to Be \mid Bbd \mid ad \mid e$

② remove $B \to Be$ and $B \to Be$

$$\boxed{\begin{aligned} S &\to Bb \mid a \\ B &\to eB' \mid adB' \\ B' &\to eB' \mid bdB' \mid \epsilon \end{aligned}}$$

B. Eliminate Left recursion from following grammar [2]

2018E

S→aBDh
B→Bb | c
D→EF
E→g | ϵ
F→f | ϵ

Page No.
Date

B. Eliminate Left recursion from following grammar [3]

2019S

S→aBDh
B→ Bb|c
D→ EF; E→ g | ϵ ; F→f | ϵ

The grammar is left-recursive because it contains a pair of productions B→ Bb | c. To eleminate this, replace this pair with

B → cC
C → bC | ϵ

∴ New grammar :

S → aBDh
B → cC
C → bC | ϵ
D → EF
E → g | ϵ
F → f | ϵ

B. What is the role of Push down automata (PDA) in syntax analysis?
Eliminate Left recursion from following grammar 2019M [5]

S→aBDh
B→Bb | c
D→EF
E→g | ϵ
F→f | ϵ

B. What is the role of Context free grammar (CFG) in Compiler Design?

PDAs with stacks are used to implement parsing algorithms for conduct free grammars enabling recognition CFLs and generation of parse trees,

So, CFGs define the syntax of programming languages and guide various stages of compilation, particularly syntax analysis, by specifying how valid programs are structured and facilitating the construction of parsers to analyse program syntax

Q.No. 3

a. For the string id+id*id+id and the grammer (4)

E → E+T/T T → T*F/F F → id 2018M

Find i) leftmost derivation ii) rightmost derivation iii) parse tree

iv) Is the grammer ambiguous ?

Page No.
Date

i) LEFT MOST DERIVATION

E → E+T

E → E+T+T

E → T+T+T

E → F+T+T

E → id+T+T

E → id+T*F+T

E → id+F*F+T

E → id+id*F+T

E → id+id*id+T

E → id+id*id+F

E → id+id*id+id.

ii) Right most derivation | ω = id+id*id+id.

E → E+T

E → E+F

E → E+id

E → E+T+id

E → E+T*F+id

E → E+T*id+id

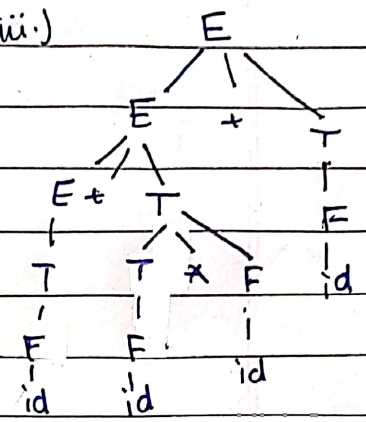
E → E+F*id+id

E → E+id*id+id

E → F+id*id+id

E → id+id*id+id

iii)



iv) Non ambiguous as only one derivation

Page No.
Date

Q.No. 2
Explain functions of various phases of compiler design and Compute FIRST & FOLLOW for the following grammar: 2023M [4][CO#1]

S'→ S —①
S→ fAh | gBh | fBk | gAk —②
A→ d —③
B→ d —④ (S' is the start symbol of grammar)

FIRST

① S' → S

FIRST (S') = First (S') if Fir(S) is not having ϵ
= {f, g} — Ⓐ using Ⓑ

② S → fAh | gBh | fBk | gAk

FIRST (S) = {f, g} — Ⓑ as X → aα and First (X) = a

③ A → d

FIRST (A) = d

④ B → d

First (B) = d

FOLLOW

① for S'

Follow (S') = {$}

② for S

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

③ for A

follow (A) = {h, k}

④ for B

Follow (B) = {h, k}

	FIRST	FOLLOW
S → S'	{f, g}	{$}
S → fAh/gBh/fBk/gAk	{f, g}	{$}
A → d	{d}	{h, k}
B → d	{d}	{h, k}

B. Compute FIRST and FOLLOW sets and Construct a predictive parsing table for the following grammar, where S is the start symbol. [5]

S→aBDh
B→cC | ϵ
C→bC | ϵ
D→EF
E→g | ϵ
F→ f | ϵ

2018 S

(where 'ϵ' denotes epsilon)

Q.No. 1

A. Compute FIRST & FOLLOW and Construct a predictive parsing table for the following grammar, where S is the start symbol. 2020M [6]

S→aBDh
B→Bb | c
D→EF
E→g | ϵ
F→f | ϵ

Q.No. 2

A. How top down parsing is different from bottom up parsing? Design predictive parsing table for the following grammar: [4] [CO# 2]

S → aBDh
B → Bb|c
D→ EF
F→ f/ϵ
E→ g /ϵ

2023E

(where 'S' is start symbol)

① Removal of left recursion in B→Bb|c

∴ Grammar : S → aBDh
B → cC
C → bC | ϵ
D → EF
E → g | ϵ
F → f | ϵ

② FIRST

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

FOLLOW

Follow (S) = {$}
Follow (B) = {FIRST (D) − ϵ} ∪ First(h)
= {g, f, h}
Follow (C) = Follow (B) = {g, f, h}
Follow (D) = follow (h) = {h}
Follow (E) = {FIRST (F) − ϵ} ∪ follow (D)
= {f, h}
Follow (F) = Follow (D) = {h}

③ PREDICTIVE PARSING TABLE

	a	b	c	g	f	h	$
S	S→aBDh	err	err	err	err	err	err
B	err	err	B→cC	err	err	err	err
C	err	C→bC	err	C→ϵ	C→ϵ	C→ϵ	err
D	err	err	err	D→EF	D→EF	D→EF	err
E	err	err	err	E→g	E→g	E→g	err
F	err	err	err	err	F→f	F→f	err

Q.No. 1

A. Compute FIRST and FOLLOW sets and Construct a predictive parsing table for the following grammar, where S is the start symbol. [7]

S→iEtS | a
S→ iEtSeS
E→ b

2019M

Page No.
Date

① Gramonar is unambiguous but has left factoring

$S \to iEtS \mid a$
$S \to iEtSeS$
$E \to b$

$\Rightarrow$

$S \to iEtSS' \mid a$
$S' \to eS \mid \epsilon$
$E \to b$

② FIRST

First (S) = {i, a}
First (S') = {e, ϵ}
First (E) = {b}

FOLLOW

Follow (S) = {First (S)} = {i, a}
Follow (S') = {i, a}
Follow (E) = {$}

Q.No. 1

A. Compute FIRST, FOLLOW sets, and Construct a predictive parsing table for the following grammar, where S is the start symbol. [5]

S→iEtSS₁ | a
S₁→eS | ϵ
E→ b

2019S

(where 'ϵ' denotes epsilon)

③ Predictive Parsing Table

	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ϵ			S'→ϵ
E		E→b				

Q.No. 2

A. Compute FIRST and FOLLOW sets for the following grammar

D→T L;
L→id M;
M→id M /ϵ
T→int / float

(where 'ϵ' denotes epsilon) [4]

2018E

FIRST

First (T) = {int, float}
First (D) = First (T) = {int, float}
First (L) = {id}
First (M) = {id, ϵ}

Follow

Follow (T) = First (L) = {id}
Follow (D) = {$}
Follow (L) = {Follow (D)} = {$}
Follow (M) = follow (L) = {$}

Page No.
Date

Q.No. 4
What is shift reduce parsing? Design predictive parsing table for the following grammar: 2023M [4][CO#2]

A→ dBcAD | f
D→ tA| e
B→ g (where 'A' is the start symbol.)

Shift reduce parsing is a bottom up parsing method which uses reverse of rightmost derivation to reduce the string into start symbol using

→ input buffer for input string
→ Stack for storage and accessing handle.

Its basic actions are:

- Shift
- Reduce
- Accept
- Error.

① Grammar: (unambiguous)

A → dBcAD | f
D → tA | ϵ
B → g.

② FIRST

First (A) = {d, f}
First (D) = {t, ϵ}
First (B) = {g}

Follow

Follow (A) = { First (D) - {} ∪ { Follow A }
= { t, $ }
Follow (D) = {Follow (A)} = { t, $ }
Follow (B) = First (c) = { C }

③ TABLE

	c	d	f	g	t	$
A		A→ dBcAD	A→ f			
D					D→ ϵ ? D→ t	D→ ϵ
B				B→ g		

Page No.
Date

Q.No. 2

A. Design predictive parsing table for the following grammar and differentiate between top down and bottom up parsing. [4

$S \to X$
$X \to aY|Xd$
$Y \to bYZ / f$
$Z \to g$ (where 'S' is start symbol)

2019 E

TOP-DOWN	BOTTOM-UP
Parse tree can be built from root to leaves	Parse tree can be built from leaves to root.
Simple to implement and less power	Complex to impl. and more powerful
Less efficient parsing techniques due to ambiguity and left recursion	More efficient parsing technique, efficiently handles ambiguity in parse tree
Applicable to small class of lang.	Applicable to broader class of lang
Eg.) Recursive Descent Parser, Predictive Parser	Eg.) Shift reduce parser, Operator precedence Parser, LR parser

(1) Grammar is ambiguous, $X \to Xd$ $\Rightarrow$ $S \to X$
$X \to aYX'$
$X' \to dX' | \epsilon$
$Y \to bYZ | f$
$Z \to g$

(2)

FIRST	Follow
First(S) = {a} = First(X)	Follow(S) = {$}
First(X') = {d, ε}	Follow(X) = {$}
First(Y) = {b, f}	Follow(X') = {$}
First(Z) = {g}	Follow(Y) = {g}
	Follow(Z) = {g}

(3) Table

	a	b	d	f	g	$
S	$S \to X$	–	–	–	–	–
X	$X \to aYX'$	–	–	–	–	–
X'	–	–	$X' \to dX'$	–	–	$X' \to \epsilon$
Y	–	$Y \to bYZ$	–	$Y \to f$	–	–
Z	–	–	–	–	$Z \to g$	–

Page No.
Date

B. Design predictive parsing table for the following grammar and differentiate between top down and bottom up parsing. [4]

S → aAcD|BCe
A → b| c
B → Cf | d
C → fe (where 'S' is start symbol)

20195

① Grammar is LL(1) ✓

② FIRST

First (S) = {a, f, d}
First (A) = {b, ε}
First (B) = {f, d}
First (C) = {f}

FOLLOW

Follow (S) = {$}
Follow (A) = {c}
Follow (B) = {f}
Follow (C) = {e, f}

③ Predictive parsing TABLE

	a	b	c	d	e	f	$
S	S→aACD			S→BCE		S→BCE	
A		A→b	A→ε				
B				B→d		B→Cf	
C						C→fe	

Page No.
Date

B. Construct parsing table for the following grammar, where S is the start symbol [5]

S→AaAb
S→BbBa
A→ ε
B→ ε

2018S

① Grammar is LL(1) ✓

② First

First (S) = First (Aa Ab) ∪ First (Bb Ba)

First (S) = {a} ∪ {b} = {a, b}

First (A) = {ε}

First (B) = {ε}

Follow

Follow (S) = {$}

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

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

③ Predictive parsing table

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

B. Consider following grammar [5]

S→aAb| bB
A→ Aa | ε
B→ Bb| ε

and test whether the grammar is LL(1) or not?

2018S

test

① for S → aAb | bB

First (aAb) ∩ First (bB) = ϕ ✓

② for A → Aa | ε

First (Aa) ∩ First (ε) = ϕ ✓

and First (Aa) ∩ Follow (A) = {a, ε} ∩ {a, b} = {a} ✗

Not LL(1)

Q.No. 2

A. Consider following grammar and test whether the grammar is LL(1) or not ? [4]

2019 S

S→1AB| ϵ
A→ 1AC|0C
B→ 0S
C→ 1

Page No.
Date

Q.No. 3

A. Consider following grammar and test whether the grammar is LL(1) or not ? 2019 M [5]

S→1AB| ϵ
A→ 1AC|0C
B→ 0S
C→ 1

$A \rightarrow \alpha | \beta$

for a grammar to be LL(1);
for every pair of productions:
{

FIRST (α) $\cap$ FIRST (β) = ϕ

and

if FIRST (β) contains ϵ, and FIRST (α) dont contain ϵ
then

FIRST (α) $\cap$ FOLLOW (A) = ϕ

for pair of productions $S \rightarrow 1AB | \epsilon$

FIRST (1AB) $\cap$ FIRST (ϵ) = $\{1\} \cap \{\epsilon\} = \phi$, and

FIRST (1AB) $\cap$ FOLLOW (S) = $\{1\} \cap \{\$\} = \phi$

for pair of prodn $A \rightarrow 1AC | 0C$

FIRST (1AC) $\cap$ FIRST (0C) = $\{1\} \cap \{0\} = \phi$

Hence, its LL(1) grammar.

[e] LL(1) grammer. 2010 M

A context free grammar $G = (V_T, N, P, S)$ whose parsing table has no multiple entries is said to be LL(1). here in LL(1),

→ first L stands for scanning the input from left to right.
→ second L stands for producing a leftmost derivation.
→ 1 stands for using 1 symbol of lookahead at each step to make parsing action decision.

Page No.
Date

8. Explain recursive descent parsing and shift reduce parsing with suitable examples. 2020M [5]

b) Explain recursive decent parser? 2018E

[d] Recursive descent passing. 2010M

A recursive descent parser is a top-down parser built from a set of mutually recursive procedures where each such procedure implements one of the non-terminals of the grammar.
Execution starts with procedure corresponding to start symbol.

Eg:) $A \rightarrow X_1 X_2 X_3 ---- X_n$

$\therefore X_i \rightarrow ab$

```
Void A()
{
    take A → X1 X2 X3 --- Xn
    for (i=0 to n)
    {
        if Xi is a non terminal
            then call procedure Xi();
        else if Xi is a terminal a
            then enhance ptr to next input.
        else
            error
    }
```

SHIFT REDUCE PARSING

It is a simple bottom up parsing technique which uses:
→ an input buffer for storing the input string
→ a stack for storing and accessing the prod^n rules.

It has basic operations:

1. Shift: Moves symbols from input buffer to stack
2. Reduce: finds handle and replace with NT
3. Accept: Pair ($S, $)
4. Error: Cant reduce nor shift

Page No.
Date

(eg.) $E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id_1 \mid id_2 \mid id_3$
string, $w = id_1 + id_2 * id_3$

$E \rightarrow E + E$
$\rightarrow E + E * E$
$\rightarrow E + E * id_3$
$\rightarrow E + id_2 * id_3$
$\rightarrow id_1 + id_2 * id_3$

STACK	INPUT	ACTION
$	$id_1 + id_2 * id_3$ $	Shift
$ id_1	$+ id_2 * id_3$ $	Reduce $E \rightarrow id_1$
$ E	$+ id_2 * id_3$ $	Shift
$ E +	$id_2 * id_3$ $	Shift
$ E + id_2	$* id_3$ $	Reduce $E \rightarrow id_2$
$ E + E	$* id_3$ $	Shift
$ E + E *	id_3 $	Shift
$ E + E * id_3	$	Reduce $E \rightarrow id_3$
$ E + E * E	$	Reduce $E \rightarrow E * E$
$ E + E	$	Reduce $E \rightarrow E + E$
$ E	$	Accept as $S \rightarrow E$, $

∴ successful parsing → HALT

4 Explain the following terms (any four):-
[a] Handle pruning 2010M

Handle pruning is the process of removing the children of the left-hand side non terminal from the parse tree, i.e. it is the replacement of handle by NT.

$S \rightarrow \alpha A \beta \longrightarrow \alpha \underbrace{\gamma}_{} \beta$

↳ handle → replaced by NT in left side of prodn $A \rightarrow \gamma$

Page No.
Date

b. For the grammer 2018M (10)

S→aBDh B→cC C→bC/ϵ D→EF
E→g/ϵ F→f/ϵ

Check whether the string "acbbgfh" is parsable by LL(1) parser or not
(ϵ is epsilon . Show all steps involved in parsing)

It is parsable

First (S) = {a}
First (B) = {c}
First (C) = {b, ϵ}
First (D) = {g, f, ϵ}
First (G) = {g, ϵ}
First (F) = {f, ϵ}

Follow (S) = {$}
Follow (B) = {
Follow (C) = {Follow(D)} = {g, f, h}
Follow (D) = {h}
Follow (E) = {f, h}
Follow (F) = {h}

TABLE

	a	b	c	g	f	h	$
S	S→aBDh						
B			B→cC				
C		C→bC		C→ϵ	C→ϵ	C→ϵ	
D				D→EF	D→EF	D→EF	
E				E→g	E→ϵ	E→ϵ	
F					F→f	F→ϵ	

STACK	INPUT	MOVE
$S	acbbgfh	S→aBDh
$hDBa	acbbgfh	POP
$hDB	cbbgfh	B→cC
$hDCc	cbbgfh	POP
$hDC	bbgfh	C→bC
$hDCCb	bbgfh	POP
$hDCb	bgfh	C→bC
$hDCC	gfh	C→ϵ
$hDC	gfh	C→ϵ
$hD	gfh	D→EF
$hFE	gfh	E→g
$hFg	gfh	POP
$hF	fh	F→f
$hf	fh	POP
$h	h	

Accept

Page No.
Date

Q.No. 3

A. What are the problems associated with Top down parsing? Is the following grammar LL(1)? 2020M [5]

Problems with top down parsing are:-

1.) Left Recursion: Occurs when a non-terminal directly or indirectly produces itself. It causes infinite loops in parsing.

Eg.) $A \rightarrow A\alpha / \beta$

can be reduced to

$A \rightarrow \beta A'$

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

2.) Left Factoring: Technique to remove common prefixes in grammar rules, reducing ambiguity and aiding predictive parsing.

Eg.) $A \rightarrow \alpha \beta_1$

$A \rightarrow \alpha \beta_2$

rewrite as

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2$

3.) Backtracking: Strategy used in parsing to explore multiple parsing paths. Can be inefficient, especially for ambiguous grammars.

Page No.
Date

Q.No. 5
A. What is an operator precedence parsing? Explain operator precedence parsing Algorithm. 2023E [4] [CO#2]

Operator precedence parser is an efficient shift reduce parser which can be constructed for both ambiguous and unambiguous grammar. It is done on operator precedence grammar)
(operator grammar is CFG where no production of the grammar has null or two adjacent non-terminal in RHS of grammar).
These are useful for scientific applications

Parsing Algo

```
Set pointer i/p to first symbol of w$
while {
    if TOS is $ and i/p points to $
        return
    else {
        let terminal a is on TOS and i/p points to terminal
        if a<b or a=b then{
            push b on TOS
            advance i/p to next symbol
        }
        if a< b {
            while (top terminal of stack is related by < to
                most recently popped terminal) {
                pop
            }
        }
    }
}
```

Eg) Table for $E \to E+E$
$E \to E*E$
$E \to id$

$\theta_1 \backslash \theta_2$	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	-	>
$	<	<	<	-

Page No.
Date

for string, ω = id+id*id

STACK		input buffer	Action
$	<	id+id*id$	Shift id
$ id	>	+id*id$	pop
$	<	+id*id$	Shift +
$ +	<	id*id$	Shift id
$ +id	>	*id$	Pop
$ +	<	*id$	Shift *
$ + *	<	id$	Shift id
$ + * id	>	$	Pop
$ + *	>	$	Pop
$ +	>	$	Pop
$		$	Accept

Successful parsing → HALT.

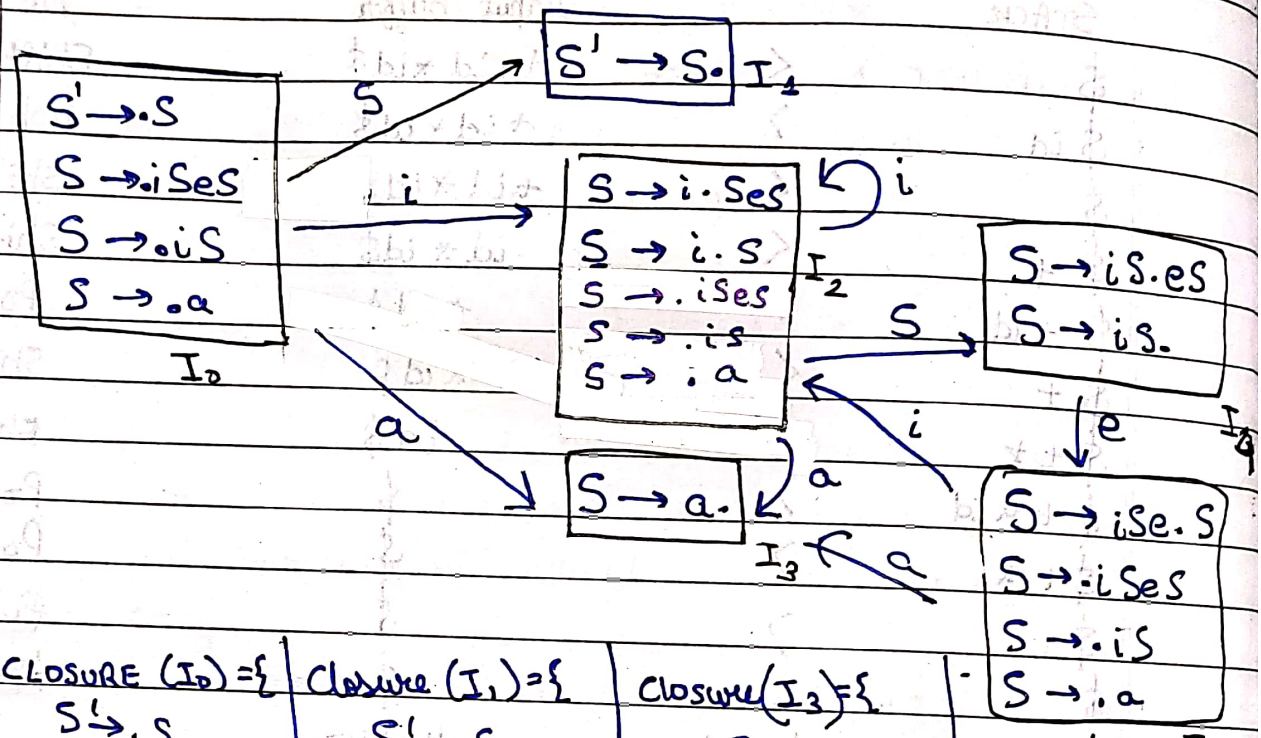
Page No.
Date

Q.No. 3

A. Define CLOSURE(I) and GOTO(I,X) functions and construct the sets of LR(0) items for the following grammar. [4][CO#3]

$S' \rightarrow S$ 2023E

$S \rightarrow iSeS \mid iS \mid a$ (where 'S' is start symbol)



CLOSURE (I_0) = {
$S' \rightarrow .S$
$S \rightarrow .iSeS$
$S \rightarrow .iS$
$S \rightarrow .a$
}

Closure (I_1) = {
$S' \rightarrow S.$
}

Closure (I_2) = {
$S \rightarrow i.Ses$
$S \rightarrow i.S$
$S \rightarrow .iSes$
$S \rightarrow .iS$
$S \rightarrow .a$
}

Closure (I_3) = {
$S \rightarrow a.$
}

Closure (I_4) = {
$S \rightarrow iS.eS$
$S \rightarrow iS.$
}

Closure (I_5) = {
$S \rightarrow iSe.S$
$S \rightarrow .iSeS$
$S \rightarrow .iS$
$S \rightarrow .a$
}

Closure (I_6) = {
$S \rightarrow iSeS.$
}

GoTo $(I_0, S) = I_1$
GoTo $(I_0, i) = I_2$
GoTo $(I_0, a) = I_3$
GoTo $(I_2, i) = I_2$
GoTo $(I_2, S) = I_4$
GoTo $(I_2, a) = I_3$
GoTo $(I_4, e) = I_5$
GoTo $(I_5, S) = I_6$
GoTo $(I_5, a) = I_3$

Page No.
Date

Q3.a i) Explain type of conflicts that occur in LR parsing with example? [7]

The two types of conflicts in LR parsing are SR and RR conflict:

1.) Shift Reduce Conflict:
It is caused when grammar allows a production rule to be reduced in a state and in the same state another production rule is shifted for same token.

2.) Reduce - Reduce Conflict:
It is when two or more productions apply to the same sequence of inputs. This grammar becomes ambiguous because a program can be interpreted in more than one way.

Eg: $\boxed{A \to B. \text{ — } ③} \xrightarrow{id} \bigcirc$ I_2

I_1

	id	+	*
I_1	S_2/r_3	r_3	r_3

$\boxed{\begin{array}{l} A \to B. \text{ — } ② \\ B \to id. \text{ — } ③ \end{array}}$

I_1

.	t_1	t_2	t_3
I_1	r_2/r_3	r_2/r_3	r_2/r_3

Q.No. 7
A. Explain and find canonical collection of sets of LR(1) items for following grammar [4]

2019 5

S → aAd| bBd |aBc |bAc
A → e
B → e

Page No.
Date

Q.No. 3
Explain algorithm for constructing SLR parsing table and

Algorithm for construction of SLR parsing table.

① find canonical collection of set of items / LR(0) items
$C = \{I_0, I_1, I_2 ----\}$ i.e. set of states of DFA.

② Follow following rules for construction of Action and Goto table.

for ACTION TABLE

a.) If $A \to \alpha . a\beta$ is in I_i and $Goto(I_i, a) = I_j$
then set action $[I_i, a] = S_j$ (shift J)

b.) If $A \to \alpha .$ is in I_i then for every b in follow(A)
set action $[I_i, b] = R_k$ or Reduce by $A \to \alpha$
↳ (production no.)

c.) If $S' \to S$, where S': initial start system of aug. grammar
S: Start state of given grammar.
Set Action $[I_i, \$] =$ accept

for GOTO TABLE

d.) If $Goto(I_0, A) = I_j$ then
set $Goto[I_i, A] = J$.

e.) All undefined entries in Action and Goto table are an error.

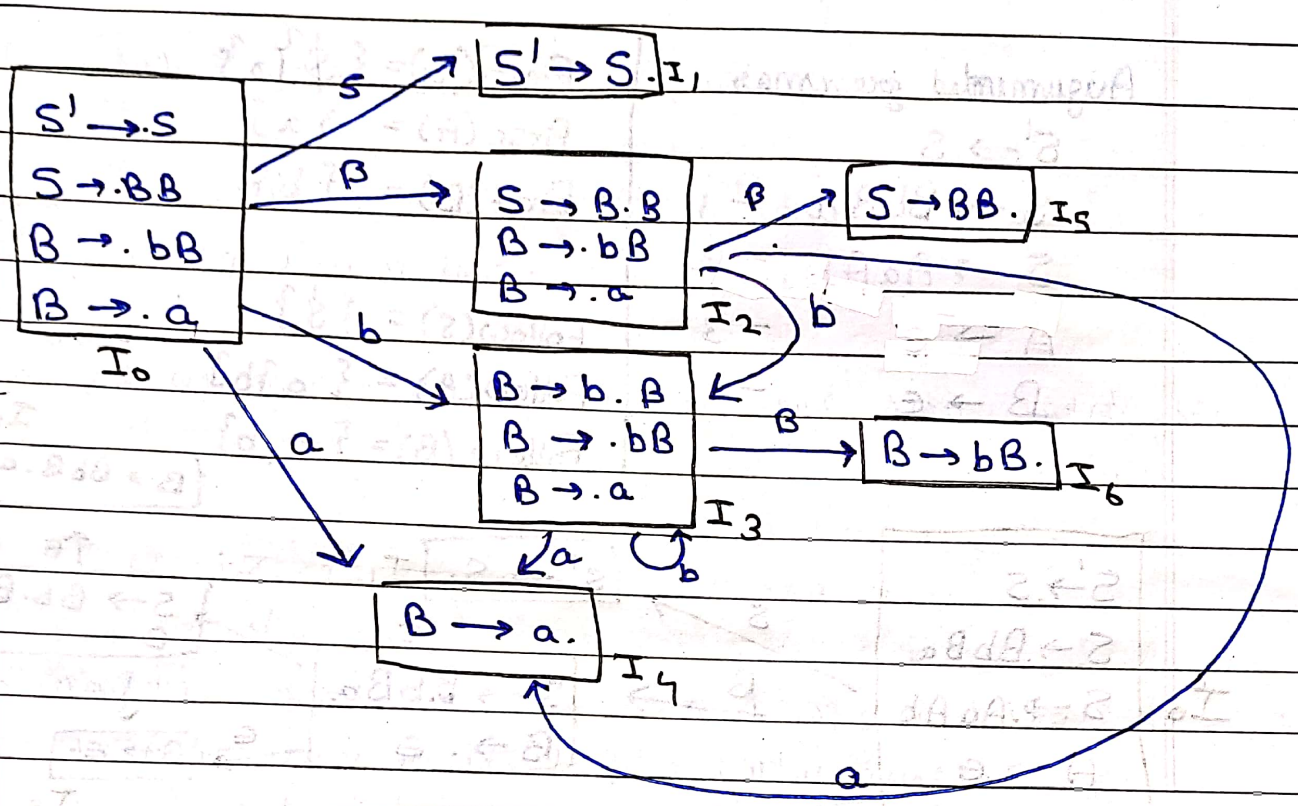
Q.No. 2
A. Construct SLR(1) parsing table for the following grammar, where S is the start symbol 2019M [5]
S→BB
B→bB | a

Augumented grammar
$S' \to S$
$S \to BB$ — R_1
$B \to bB$ — R_2
$B \to a$ — R_3

First (B) = {a, b}
First (S) = {a, b}

Follow (B) = {$}
Follow (S) = {$}

Page No.
Date



States of DFA	Action			GoTo	
	$	a	b	S	B
I_0	—	S_4	S_3	1	2
I_1	accept	—	—	—	—
I_2	—	S_4	S_3	—	5
I_3	—	S_4	S_3	—	6
I_4	R_3	—	—	—	—
I_5	R_1	—	—	—	—
I_6	R_2	—	—	—	—

SLR(1) Parsing table.

Q.No. 2

A. Construct SLR(1) parsing table for the following grammar, where S is the start symbol [5]

2020M

S→BbBa A→ ϵ

S→AaAb B→ ϵ

Page No.
Date

Augumented grammar

$S' \to S$

$S \to BbBa$ – 1

$S \to AaAb$ – 2

$A \to \epsilon$ – 3

$B \to \epsilon$ – 4

First (S) = {a, b}

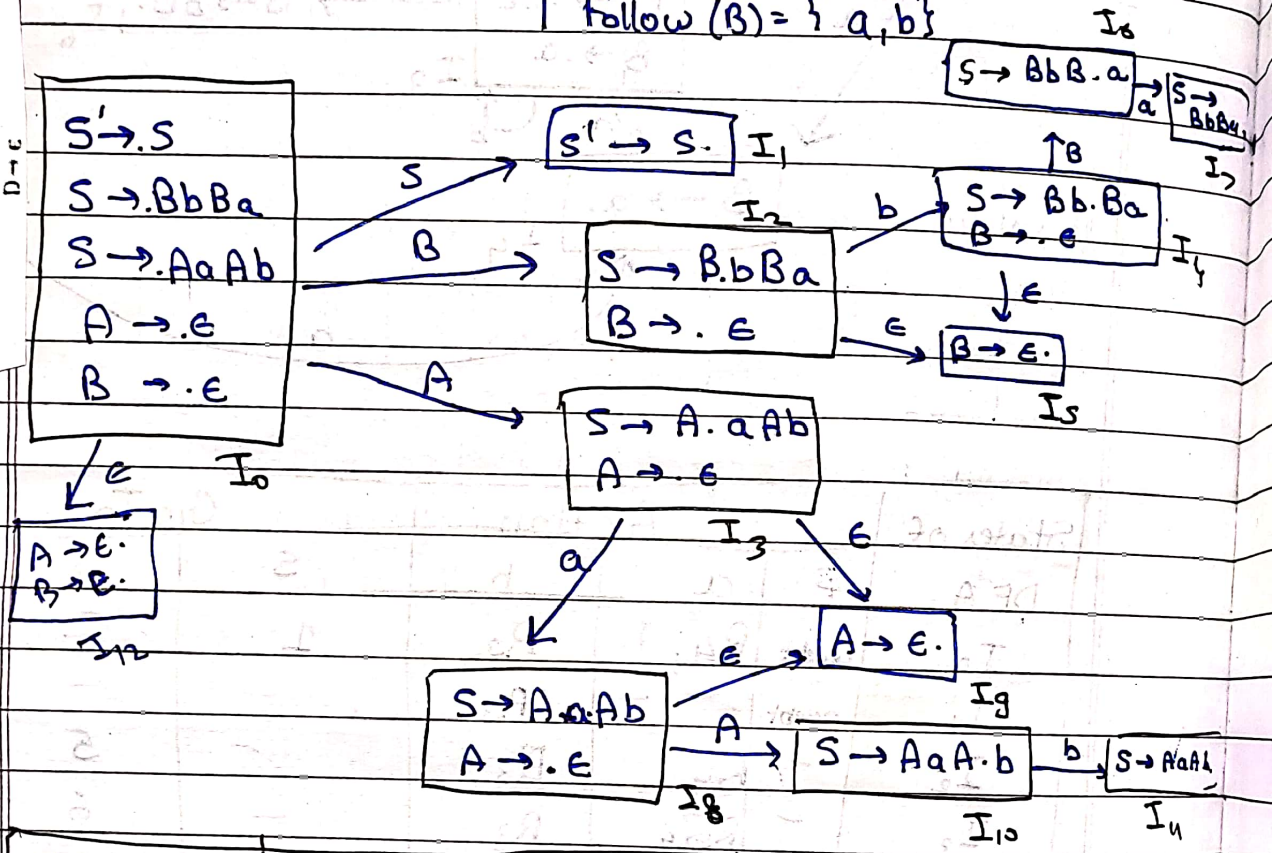
First (A) = {a}

First (B) = {b}

Follow(S) = {$}

Follow(A) = {a, b}

Follow (B) = {a, b}



States of DFA	Action				Goto		
	a	b	ϵ	$	S	A	B
I_0	–	–	S_{12}	–	1	3	2
I_1	–	–	–	accept	–	–	–
I_2	–	S_4	S_5	–	–	–	–
I_3	S_8	–	S_9	–	–	–	–
I_4	–	–	S_5	–	–	–	6
I_5	R_4	R_4	–	–	–	–	–
I_6	S_7	–	–	–	–	–	–
I_7	–	–	–	R1	–	–	–
I_8	–	–	S_9	–	–	10	–

Page No.
Date

I_9	R_3	R_3	—	—	—	—	—
I_{10}	—	S_{11}	—	—	—	—	—
I_{11}	—	—	—	R_2	—	—	—
I_{12}	R_3/R_4	R_3/R_4	—	—	—	—	—

Q.No. 1

A. Construct SLR (1) parsing table and compute FIRST & FOLLOW for the following grammar: [6]

S→xAy|xBy|xAz
A→ aS /q
B→ q

2019 E

Q.5 Construct SLR(1) parsing table for the following grammer : [14]

S→xAy /xBy / xAz
A→aS / q
B→ q

2017E

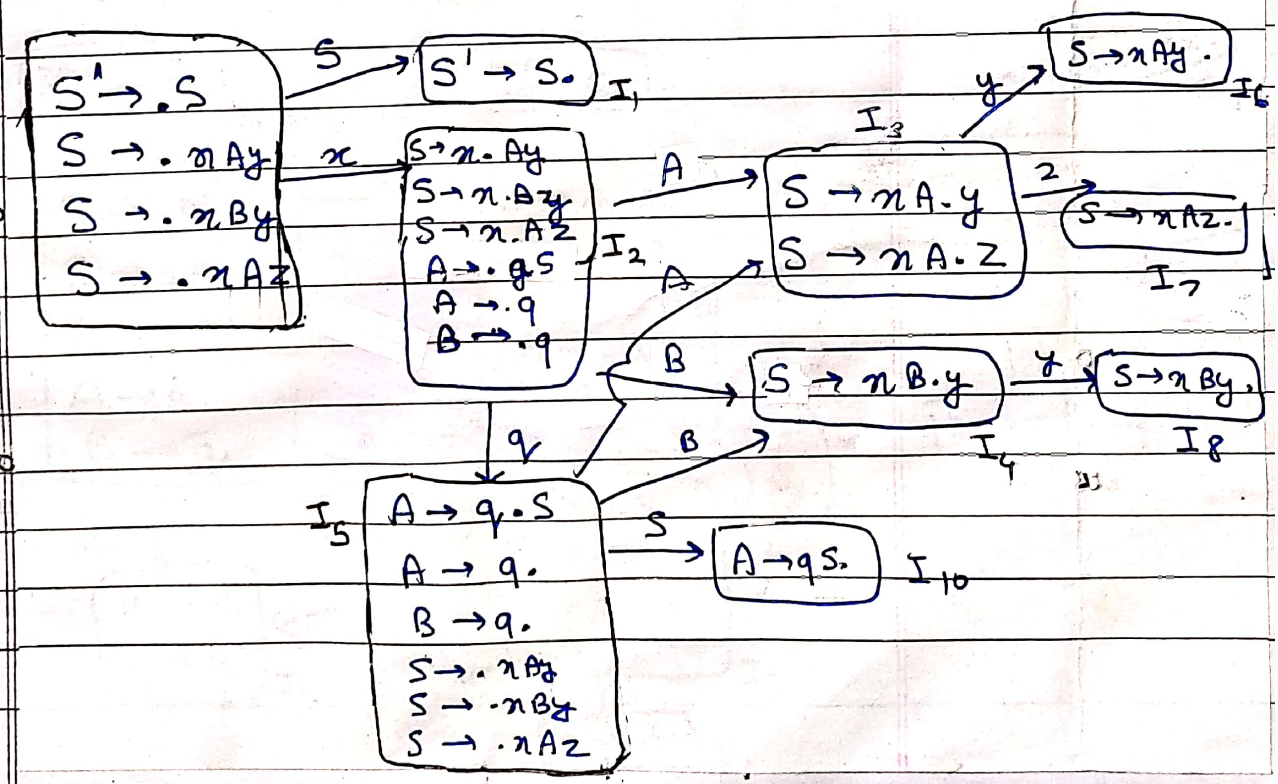
(Also show canonical collection diagram & follow sets)

Augumented Grammar

$S' \to S$
$S \to xAy$ — 1
$S \to xBy$ — 2
$S \to xAz$ — 3
$A \to aS$ — 4
$A \to q$ — 5
$B \to q$ — 6

First (S) = {x}
First (A) = {a, q}
First (B) = {q}

Follow (S) = {$, y, z}
Follow (A) = {y, z}
Follow (B) = {y}



Page No.
Date

States of DFA	ACTION					GOTO		
	x	y	z	q	$	S	A	B
I_0	S_2	R_3/R_4				1		
I_1					accept			
I_2				S_5			3	4
I_3		S_6	S_7					
I_4		S_8						
I_5	S_2	R_5/R_6	R_5			9		
I_6		R_1	R_1		R_1			
I_7		R_3	R_3		R_3			
I_8		R_2	R_2		R_2			
I_9		R_4	R_4					

Q.No. 4

A. What is the advantage of left recursive grammar over right recursive grammar in LR parsing. Explain with suitable example. 2018 S [3]

We should use left recursive grammar as we can parse a sequence of any number of elements with bounded stack space.

With right recursion, no reduction takes place until the entire list of elements has been read; with left recursion, a reduction takes place as each new list element is encountered. It saves space.

In LR(1), left recursive grammar elimination allows for easier construction of LR parsing tables and avoids the need for backtracking during parsing. Right recursive requires extra work for conversion

Eg.

$E \to E+T \mid T$	$E \to TE'$
$T \to T*F \mid F$	$E' \to +TE' \mid \epsilon$
$F \to (E) \mid id$	$T \to FT'$
	$T' \to *FT' \mid \epsilon$
	$F \to (E) \mid id$
left	right

Page No.
Date

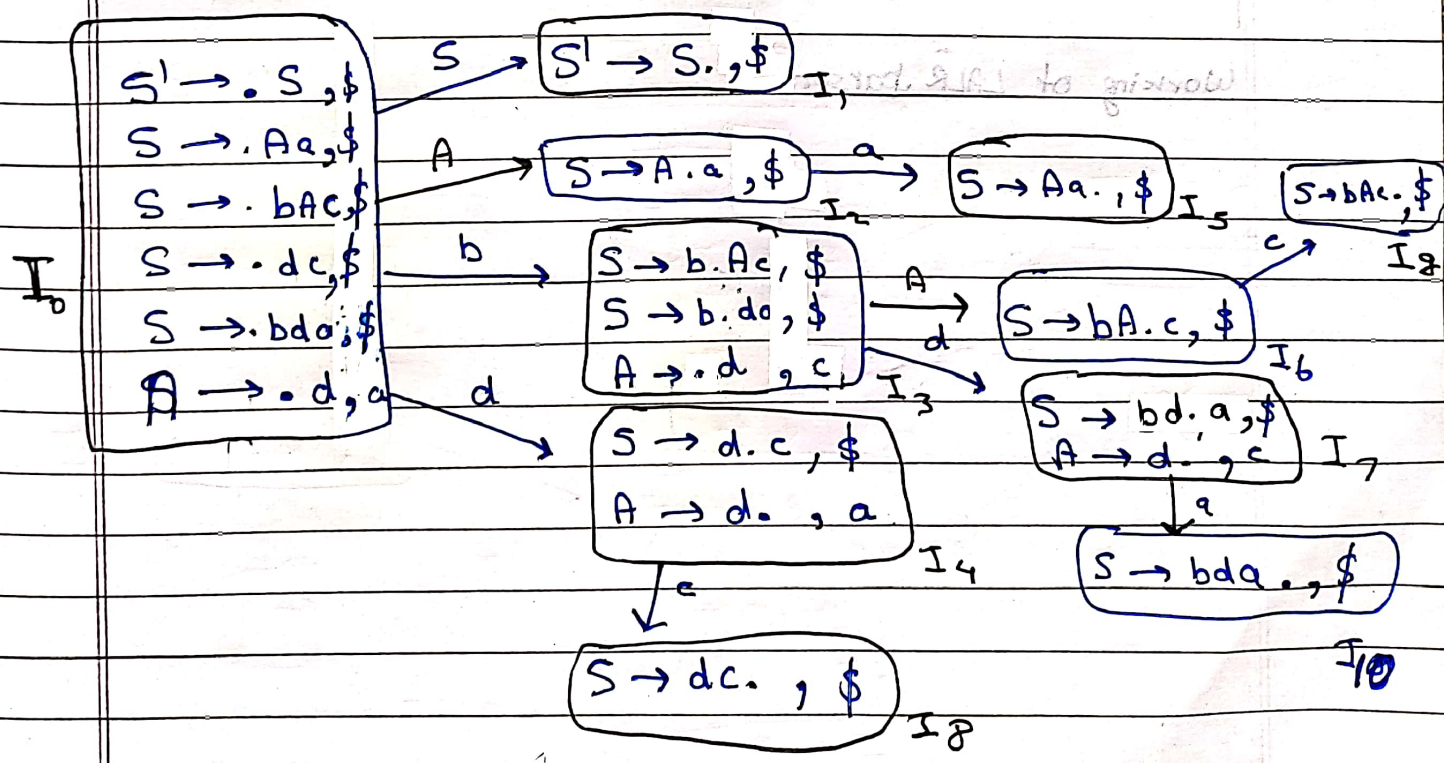
B. How LALR is different from CLR? explain and find canonical collection c
sets of LR(1) items for following grammar [4
S → Aa| bAc |dc |bda
A → d

2019 E

In LALR(1) parsing, the LR(1) items ~~are~~ with equal productions but different look ahead are grouped to form an individual set of items. It is frequently the similar as CLR(1) except for the one difference that is the parsing table.

LALR Parser	CLR Parser
It is cheap and easy to implement.	It is expensive and difficult to imp
Less states / ~~more TC/SC~~	More states
Error detection isn't immediate	
power is less	power is more

Grammar : $S' \rightarrow S$
$S \rightarrow Aa \mid bAc \mid dc \mid bda$
$A \rightarrow d$



A. Discuss algorithm for computation of the sets of LR(1) items. Shows that the following grammar is LR(1) but not LALR(1) [3]

S→Aa|bAc|Bc|bBa
A→d
B→ d

2018S

B. Explain the working of LALR parser and Construct canonical LR parsing table for following grammar [4] [CO#4]

S → Aa| bAc |dc |bda
A → d

2023E

Q.No. 3

Page No.
Date

Parsing Table (CLR(1))

It's not LALR(1) as it cant be reduced w/o conflict ($I_4 - I_5$)

States of DFA	ACTION a	d	c	d	$	GOTO S	A
I_0		S_3		S_4		1	2
I_1					accept		
I_2	S_5						
I_3				S_7		1	
I_4	R_5		S_8				
I_5					R_1		
I_6	S_{10}		S_9				
I_7			R_5				
I_8					R_3		
I_9					R_2		
I_{10}					R_4		

Working of LALR parser

Page No.
Date

Q.No. 5

How LALR is different from CLR? explain and find sets of LR (1) items for following grammar 2023M {4}{CO#4}

E → Gb| cGd |ed |ceb

G → e .(where 'E' is the start symbol)

augumented grammar

S → E

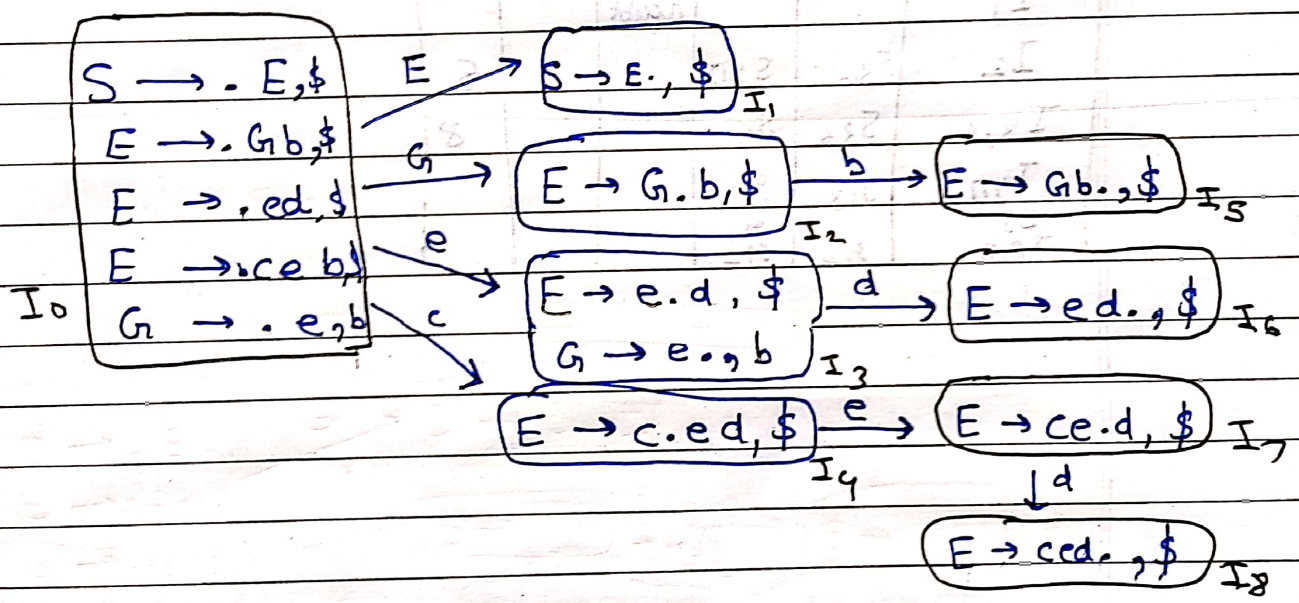
E → Gb

E → cGd

E → ed

E → ceb

G → e



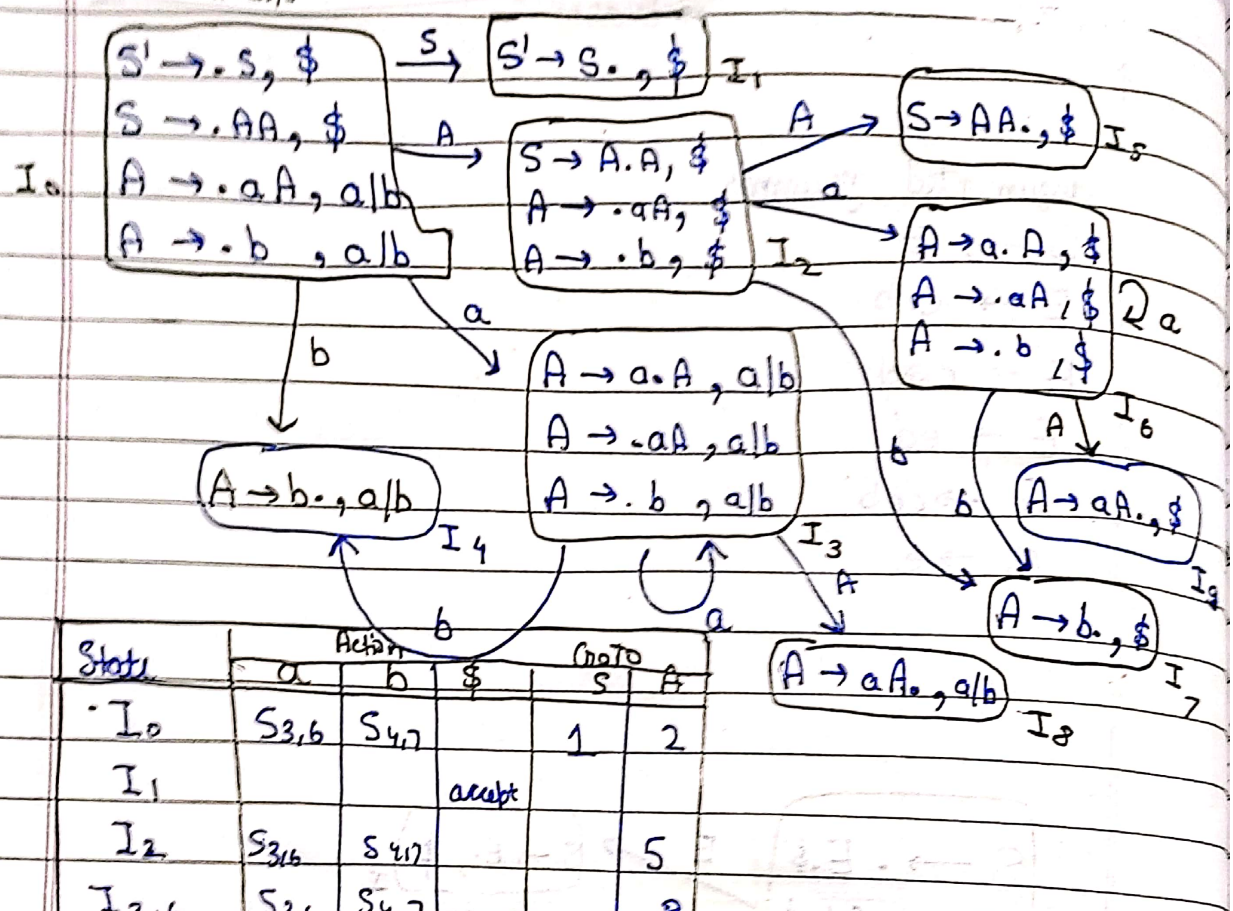
$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

Q.No. 1

A- How LALR is different from CLR? Construct LALR (1) parsing table for the following grammar: [6]

$S \to AA$
$A \to aA / b$

2018E



State	Action			GoTo	
	a	b	$	S	A
I_0	$S_{3,6}$	$S_{4,7}$		1	2
I_1			accept		
I_2	$S_{3,6}$	$S_{4,7}$			5
$I_{3,6}$	$S_{3,6}$	$S_{4,7}$			8
$I_{4,7}$	r_3	r_3			
I_5		r_1			
$I_{8,9}$	r_2	r_2			

Q.1 Construct CLR(1) parsing table for the following grammer : [14]

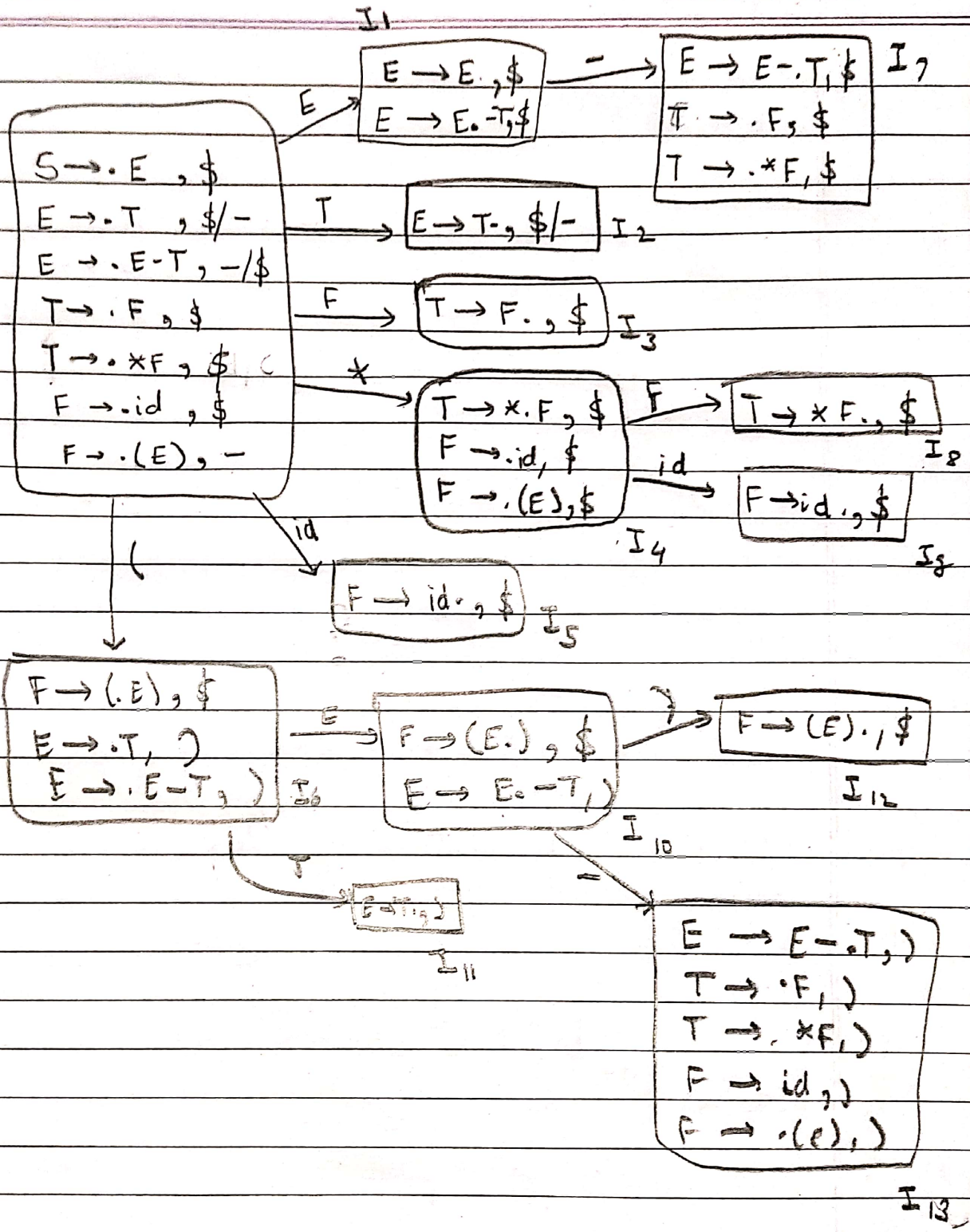
$E \to T / E-T$
$T \to F / *F$
$F \to id / (E)$

2018E

augumented grammer

$S \to E$
$E \to T / E-T$
$T \to F / *F$
$F \to id / (E)$

Page No.
Date


I0
S → . E , $
E → . T , $/-
E → . E-T , -/$
T → . F , $
T → . *F , $
F → . id , $
F → . (E) , -
E
I1
E → E . , $
E → E . -T, $
-
I7
E → E- . T, $
T → . F, $
T → . *F, $
T
I2
E → T- , $/-
F
I3
T → F . , $
*
I4
T → * . F , $
F → . id, $
F → . (E), $
F
I8
T → * F . , $
id
I9
F → id . , $
id
I5
F → id . , $
(
I6
F → (. E) , $
E → . T,)
E → . E-T,)
E
I10
F → (E .) , $
E → E . -T,)
)
I12
F → (E) . , $
T
I11
E → T . ,)
-
I13
E → E- . T,)
T → . F,)
T → . *F,)
F → . id ,)
F → . (e),)

Page No.
Date

B. Enlist the problems with the following grammar

E→ E +E , E→ E* E , E→ Id 2018 E

Also rectify them to make it suitable for LL (1) parsing? [4]

UNIT-3: SYNTAX DIRECTED TRANSLATION

Page No.
Date

B. Explain following with suitable examples [4] [CO#2]
a. Syntax directed translation Schemes

A SDT scheme is a CFG in which attributes are associated with the grammar symbols, and semantic actions enclosed within braces ({}), are inserted in the right side of the productions.
The semantic actions are subroutines that are called at appropriate times by the parser, enabling the translation.

We make sure an attribute value is available when action refers it by
1. Compute inherited attribute before using them in actions.
2. Place actions computing synthesized attributes at the end of the production's right side to ensure all needed attributes are available.

Eg:)
$D \rightarrow T \ \{L.type := T.type\} \ L;$
$T \rightarrow int \ \{T.type := int\}$
$T \rightarrow real \ \{T.type := real\}$
$L \rightarrow \{L1.type = L.type\} \ L, id \ \{enter(id.prt, L.type);\}$
$L \rightarrow id \ \{enter(id.prt, L.type);\}$

In predictive parsing, requirements dictate implementing actions within the production rules. Top-down parsers benefit from easily calling translation routines after recognizing symbols.

In contrast, bottom up parsers handle attributes after reducing symbols.

Page No.
Date

Q4. a) Differentiate SDT and SDD by giving suitable examples? 2018E [7]

Syntax-Directed Translation (SDT)	Syntax-Directed Defination (SDD)
A technique used to translate a source program into a target program or intermediate code representation by associating semantic rules with grammar productions.	A technique used to define attributes and their evaluation rules for programming language constructs to specify the static semantics of a language.
Used for translation tasks like code generation, type checking and ICG	Used for defining static semantics like type checking, scope rules, and other constraints
Ex.) Generating machine code or ICG for an assignment statement during code generation phase of a compiler	Defining the type of an identifier based on a type declaration during the sematic analysis phase of a compiler.

SDD are generalization of CFG in which:

1.) Grammar symbols are having a set of attributes

2.) production rules are associated with semantic rules for computation of attribute rules.

Ex.)

productions	Semantic rules
$L \rightarrow E$	$L.val = E.val$
$E \rightarrow E+T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T*F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = (E.val)$
$E \rightarrow digit$	$E.val = (digit.lexval)$

Page No.
Date

b) Write SDD for generating three address code for control flow statement. [7]
2018E

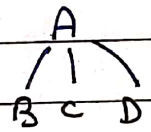
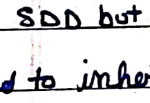
Q.No. 5
A. Write SDD for generating three address code for Boolean expressions with &&, || (OR) and ! Operators. 2019 E [4]

Operator	Production Rule	Semantic Actions
~~E1 && E1 OR E2~~ && (AND)	$E \rightarrow E_1$ AND E_2	{ E.place = newtemp(); emit (E.place := E_1.place 'and' E_2.place)
|| (OR)	$E \rightarrow E_1$ OR E_2	{ E.place = newtemp() emit (E.place := E_1.place 'or' E_2.place)
~~E → not E1~~ !	$E \rightarrow$ not E_1	{ E.place = newtemp() emit (E.place := 'not' E_1.place }

Page No.
Date

b) Write SDT for generating three address code for an assignment statement. 2017E [7]

Q2. a) Differentiate L-attributed and S-attributed definitions? 2017E [7]

S- Attributed	L-attributed
A SDD that uses only sythesized attribute is called S-Attributed SDD Ex.) $A \rightarrow BCD$ $A.i = B.i$ $A.i = C.i$ $A.i = D.i$ 	A SDD that uses both synthesized and inherited attribute is called L-attributed SDD but each inherited attribute is limited to inherit from parent or left sibling only. Ex.) $A \rightarrow BCD$  $\therefore C_i = A.i$ / $C.i = B.i$ / $C_i = D.i$ (X)
Semantic operations are always placed at the right of production (postfix SDD)	Semantic action are placed anywhere on the RHS of production.
Attributes are evaluated with BOP	Attributes are evaluated by traversing parse tree depth first, left to right.

Q.No. 4
A. Explain why every S-attributed definition is L-attributed? with suitable example 2019S [4]

Page No.
Date

B. Write S-attributed definition for constructing a syntax tree for an assignment statement and Write down the structure of a typical activation record. 2018E [4]

Consider the grammar

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow (E_1)$

$E \rightarrow id$

E.place :- a name that holds the value of E

E.code :- the sequence of 3-address statements evaluating E

Production	Semantic Rules
$S \rightarrow id := E$	S.code := E.code || gen(id.place ':=' E.place)
$E \rightarrow E_1 + E_2$	E.place := new temp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow E_1 * E_2$	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)
$E \rightarrow -E_1$	E.place = newtemp E.code = E_1.code || gen(E.place ':=' 'uminus' E_1.place)
$E \rightarrow (E_1)$	E.place := E_1.place E.code := E_1.code
$E \rightarrow id$	E.place := id.place; E.code := ''

Page No.
Date

ACTIVATION RECORD

The information needed by a single execution or a single activity of a procedure is managed using a contiguous block of storage called a "Activation Record".
Size of fields are decided at compile time.
Information in record is enabled for easy access at execution time.

RETURN VALUE
ACTUAL PARAMETERS
CONTROL LINK (optional)
ACCESS LINK (optional)
MACHINE STATUS
LOCAL DATA
TEMPORARIES

FORMAT OF ACTIVATION RECORD

1. **Temporaries**: are used to store temporary variables
2. **LOCAL DATA**: holds data that are local to an execution of a procedure.
3. **MACHINE STATUS**: This field is used to store information about the state of the machine before procedure is called.
4. **ACCESS LINK**: used to refer to non-local data held in other activation records
5. **CONTROL LINK**: used to point to the activation record of the calling class.
6. **ACTUAL PARAMS.**: used by calling procedure to supply parameters to the called procedure.
7. **RETURN VALUE**: This field is used by the called procedure to return a value to the calling procedure

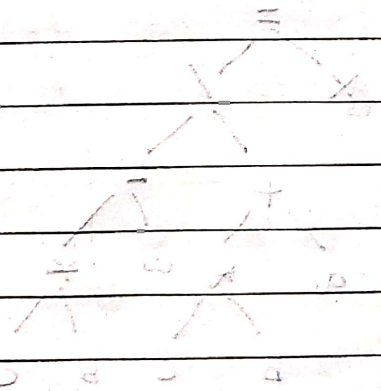
Page No.
Date

Q.No. 3

Write the S-attributed SDD for implementation of an assignment statement and show the stack implementation for x:= a* b+c with a=5, b=6 and c= 2. [8]

2018E

INPUT	STACK	VAL	PRODUCTION
x := a*b+c			
x			



Page No.
Date

Q7. a)Explain different representations of intermediate code? 2018E [7]

The different representations of intermediate code are:

① POSTFIX NOTATION

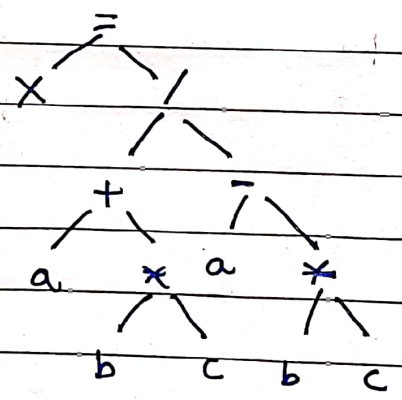
It is known as reverse polish / suffix notation
Post fix notation positions the operator at the right end.

Ex.) $(a+b) * (c+d) + (a+b)$
$\therefore$ postfix: $ab+ cd+ * ab++$

② SYNTAX TREE

A syntax tree serves as a condensed representation of a parse tree. The operator and keyword nodes present in the parse tree undergo a relocation process to be a part of their respective parent nodes in the syntax tree. the internal nodes are operators and child nodes are operands.

Ex.) $X = (a + (b*c)) / (a - (b*c))$



③ Three Address Code

It is a commonly used sequence of statements of the form
$X = y \text{ op } z$
A single statement of this form is called 3-add statement as it involves no more than 3 ref

Page No.
Date

B. What are the various three address code representations? [4] [CO#3]

2023E

Three address code representations are intermediate representations used during code generation phase of a compiler. 2

The various three address code representations include:

A := - B * (C/D)

1. Quadruples:

A quadruple is a four tuple of the form:
(operator, operand 1, operand 2, result)

They are straightforward and easy to understand.

(0)	-	B	-	T_1
(1)	/	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3	-	A

2. Triples:

A triple is a three-tuple of the form:
(operator, operand 1, operand 2)

They are similar to quadruples, but the destination operand is implicit, and the result is typically stored in a register or temporary variable.

	OP	ARG 1	ARG 2
(0)	-	B	-
(1)	/	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

3) Indirect Triples

An indirect triple is a three-tuple of the form:
(operator, operand 1, operand 2)

operand 2 is a pointer to a memory location that holds the second operand. This representation is useful when dealing with complex data structures or when operands are large and cant be directly stored in the instructions.

P.T.O

Page No.
Date

	STATEMENT		OP	ARG1	ARG 2
(0)	(21)	(21)	−	B	−
(1)	(22)	(22)	/	C	D
(2)	(23)	(23)	*	(21)	(22)
(3)	(24)	(24)	:=	A	(23)

B. Generate three address code for the following program fragment [4]

```
while(x<y and u<v) do
if x=1 then y=y+1
else
while x<=v do
x=x+3
```

2019E

1. if (x<y) goto 3
2. goto 16
3. if (u<v) goto 5
4. goto 16
5. if (x=1) goto 7
6. goto 10
7. $t_1 = y+1$
8. $y = t_1$
9. goto 1
10. if (x≤v) goto 12
11. goto 1
12. $t_2 = x+3$
13. $x = t_2$
14. goto 10
15. goto 1
16. —

Q.No. 4

A. Translate the following expression into three address statements

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

2023 E

Also give quadruple and triple representation of the same? [4] [CO#4]

Page No.
Date

Triple

$t_1 = b * c$
$t_2 = a + t_1$
$a = t_2$
$t_3 = d * c$
$t_4 = c + t_3$
$b = t_4$
$t_5 = c * e$
$t_6 = b + t_5$
$d = t_6$

	Operator	operand 1	operand 2
1	*	b	c
2	+	a	(1)
3	=	a	(2)
4	*	d	c
5	+	c	(4)
6	=	b	(5)
7	*	c	e
8	+	b	(7)
9	=	d	(8)

Quadruple

	Operator	Operand 1	Operand 2	Result
1	*	b	c	t_1
2	+	a	t_1	t_2
3	=	t_2	—	a
4	*	d	c	t_3
5	+	c	t_3	t_4
6	=	t_4	—	b
7	*	c	e	t_5
8	+	b	t_5	t_6
9	=	t_6	—	d

Q.No. 4
A. Translate the following expression into three address instructions
a:= - b * (c+d) 2018E
Also give quadruple and triple representation of the same? [4]

Page No.
Date

Q.No. 4
A. Translate the following expression into three address instructions
x:= - y * (a+b) 2019E
Also give quadruple and triple representation of the same? [4]

Three address instructions :

$t_1 = a+b$

$t_2 = -y$

$x = t_2 * t_1$

Quadruple

	Operator	Operand 1	Operand 2	Result
1	+	a	b	t_1
2	−	y	−	t_2
3	*	t_2	t_1	t_3
4	:=	t_3	−	x

Triple

	Operator	Operand 1	Operand 2
1	+	a	b
2	−	y	−
3	*	(2)	(1)
4	:=	x	(3)

B. Generate three address code for the following program fragment [4]

```
Switch (i+j)
{   case1: x=y+z
    default: p=q+r
    case2: u=v+w    }
```

2019S

1. $t_1 = i+j$
2. if (t_1=1) goto 5
3. if (t_1=2) goto 8
4. goto 10
5. $t_2 = y+z$
6. $x = t_2$
7. goto 12
8. $t_3 = v+w$
9. goto 12
10. $t_4 = q+r$
11. $p = t_4$
12. end_switch.

Page No.
Date

B. Consider the program fragment and Generate three address code for it [5]

2018 S

```
sum=0
for(i=1;i<=20; i++)
sum=sum+a[i]+b[i];
```

1. $t_1 = 0$
2. $sum = t_1$
3. $t_2 = 1$
4. $i = 1$
5. if $(i < 20)$ goto 7
6. goto 15
7. $t_3 = a[i]$
8. $t_4 = b[i]$
9. $t_5 = t_3 + t_4$
10. $t_6 = t_5 + sum$
11. $sum = t_6$
12. $t_7 = i + 1$
13. $i = t_7$
14. goto 5

UNIT-6 : CODE OPTIMIZATION

Page No.
Date

b) Explain any four optimizing transformations with examples? 2017E [7]

Q.No. 6

A. What is loop jamming?Explain code optimization by eliminating induction variables and code motion with suitable examples. 2018S [4]

Loop Jamming is a mechanism to optimize the loop by merging the body of the loop if there are same number of iterations in the loop and same indices.

It increases the execution speech by reducing the overhead associated with loops.

Ex.)

B1	B_1
$i = i+1$	$i = i+1$
$x = 3*i$	$x = x+4$
$y = a[x]$	$y = a[x]$
if $y<15$, goto B2	if $y<15$, goto B2
goto B_1	goto B_1
B2	B_2

Q.No. 3

What is loop unrolling? Explain and construct program flow graph for the following program fragment [8]

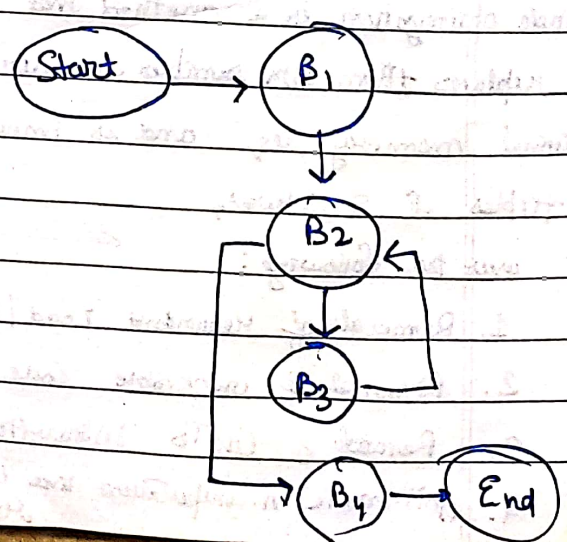
```
Fact(x) {
int f=1;

for(i=2; i<=x; i++)
f=f*i
return (f) }
```

Loop unrolling is a loop transformation technique that helps to optimize code. It is a mechanism to optimize a loop by replication of the body of the loop.

Ex) Three address code is :

1. f = 1
2. i = 2
3. if (i>x) goto 9
4. t_1 = f*i
5. f = t_1
6. t_2 = i+1
7. i = t_2
8. goto (3)
9. goto (next)

B_1 (B_1 = 1, 2); B_2 = 3; B_3 = 4, 5, 6, 7; B_4 = 8, 9



Page No.
Date

B. Explain loop unrolling and Loop jamming with example and construct program flow graph for the following program fragment. [4] [CO1]

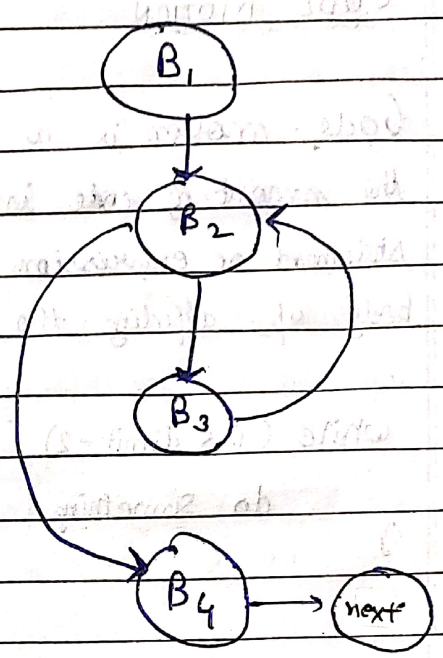
```
int main() {
extern int f(int);
int i;
int *a;
for (i=0; i<10; i++) {
a(i)=f(i); }}
```

2023E

Three Address Code

1. $t_1 = 0$
2. $i = t_1$
3. $t_2 = \&a$
4. if $(i > 10)$ goto next
5. $t_4 = f(i)$
6. $*t_2 = t_4$
7. $t_5 = i + 1$
8. $i = t_5$
9. goto 4
10. next

B_1, B_2, B_3, B_4



Q.No. 3

Eliminate loop invariant computations and construct program flow graph for the following program fragment 2019 E [8

```
Fact(x) {
Int f=1;
for(i=2; i<=x; i++)
f=f*i
return (f) }
```

To eliminate a loop invariant computation, we first identify the invariant computations and then move them outside loop if the move does not lead to a change in the program's meaning

No scope of this in given Q

Page No.
Date

b) Explain the following optimizingtransformations with examples: [7]
i) code motion ii) loop optimization 2018E

-END-

B. Differentiate between common sub-expression elimination and dead code elimination with example and also discuss code motion with suitable example 2019 E [4]

CODE MOTION

Code motion is a optimizing transformation used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop body w/o affecting the semantics of program.

Eg.)
```
while (i ≤ limit-2) {
        do something
}
```
→
```
a = limit - 2
while (i ≤ a) {
        do something
}
```

LOOP OPTIMIZATION

Loop optimization refers to the process of improving the efficiency of loops within a program. It includes :-

i.) Eliminating invariant computation : Removing set of statement which have same value in each iteration

Eg.)
```
int sum=0;
for (int i=0; i<1000; i++)
        sum += 10
```
→
```
int sum=0
invariant val = 10
for(i=0; i<1000; i++)
        sum += invarent val
```

ii.) Eliminating Induction Variable : removing AP/GP formed by ≥2 variables

Eg.)
```
for(i=A ; i≤C; i+=B)
        T = i*K
        print (T)
```
→
```
T=A*K ; T1 = B*K
for (i=A; i≤C; i+=B)
        print (T)
        T=T+T1
```

B. Differentiate between common sub-expression elimination and dead code elimination with example and also discuss elimination of local common subexpression 2018 E [4]

B. Explain common sub-expression elimination and dead code elimination code motion with suitable example 2019 S [4]

Page No.
Date

Elimination of Common Subexpression:

The expression or sub-expression that has appeared and been computed before again during the computation is eleminated in this process. This makes computation faster by avoiding recomputation of the expression, thus utilizing memory efficiently.

LOCAL-SUBEXPRESSION: It is used within a single basic block, where a basic block is a sumble code sequence that has no branches

GLOBAL-SUBEXPRESSION: It is used for an entire procedure of common subexpression elimination

Eg)

$T_1 = 4*i$	$\longrightarrow$	$S = 4*i$
$T_2 = 4*i$		$T_1 = S$
Label $T_3 = 4*i$		$T_2 = S$
		Label $T_3 = 4*i$ → different block

Dead Code Elimination:

It is an optimization that removes code which does not affect the program results. It shrinks program size, reduces resources usage and avoids executing irrelevant operations, optimizing running time.

Eg)

```
int global;
void func () {
    int i
    i = 1
    global = 1
    global = 2
    return
    global =>
}
```

$\longrightarrow$

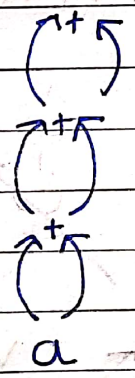
```
int global;
void func() {
    global = 2
    return
}
```

Page No.
Date

B. What is DAG? Construct DAG for the expression [4]
(((a+a)+(a+a))+((a+a)+(a+a))) 2019 E

A Directed Acyclic Graph (DAG) is a directed graph that contains nodes connected by edges, with the property that the graph contains no cycles.
It is used to represent the control flow and data dependencies of a program. This gives IR.

Given: (((a+a)+(a+a))+((a+a)+(a+a)))



B. What is DAG? What are its advantages in context of optimization? [4] [CO#6]
Construct DAG for the expression
Z=X-Y+X*Y*U-V/W+X+V 2023 E

ADVANTAGES

→ DAGs provide a compact and efficient representation of the program's control flow and data dependencies.

→ The acyclic nature of DAG allows for easy tracing of data and control dependencies.

→ The DAG structure helps identify independent computations that can be executed in //.

→ By analyzing the DAG, the compiler can easily identify and eliminate unreachable or unused code.

→ The DAG representations facilitates the identification of constant expressions for constant folding and propagation.

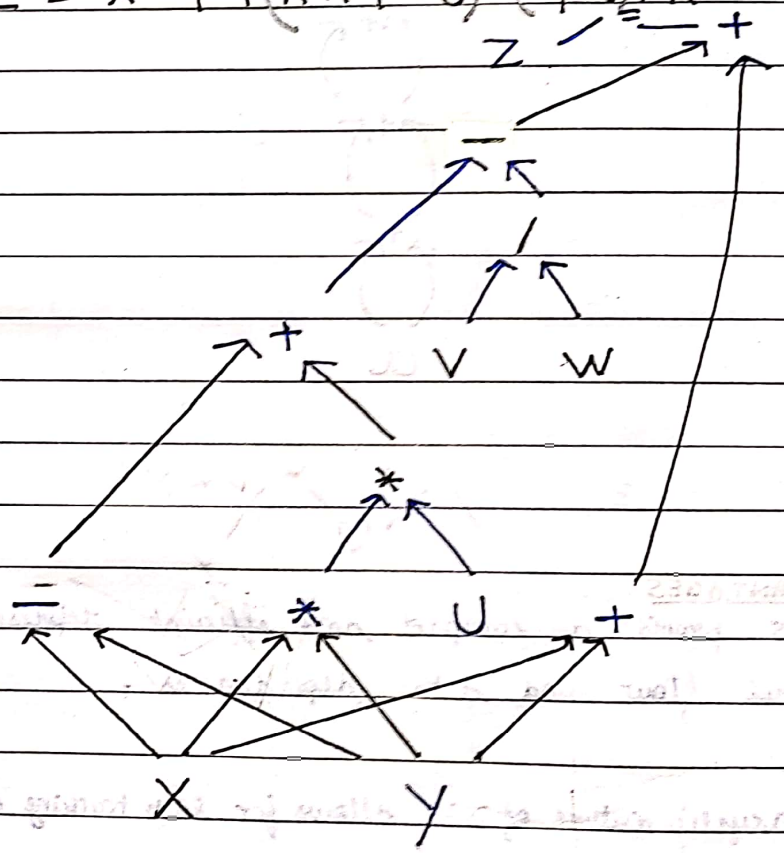
Page No.
Date

→ The DAG structure allows the compiler to easily identify and eliminate common subexpressions.

→ The DAG representation aids in the analysis of loop structures for loop level optimizations.

→ The DAG structure supports various transformation and simplification techniques for more efficient code.

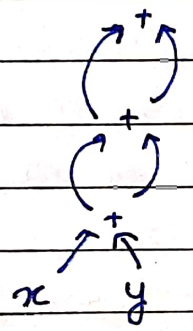
Given: $Z = X - Y + (X * Y * U) - (V / W) + X + V$



Page No.
Date

B. What is DAG? Construct DAG for the expression [4]
$(((x+y)+(x+y))+((x+y)+(x+y)))$ 2019S

Given : $(((x+y)+(x+y))+((x+y)+(x+y)))$



Q7. a) Define a basic block? Write the steps involved in dividing the code into basic blocks. Findout the basic blocks in following piece of intermediate code. [7]

2017E

```
1.  sum := 0
2.  i := 1
3.  If i > 10 goto 11
4.  t1 := 2*i
5.  t2 := a[t1]
6.  t3 := sum + t2
7.  sum := t3
8.  t4 := i + 1
9.  i := t4
10. goto 3
11. t5 := sum / i
12. Average = t5
```

B_1 B_2 B_3

A basic block is a sequence of instr with a single entry and exit point allowing no branching except at beginning and end. It is crucial for optimizations and analysis as it offers predictable control flow within a block.

ALGO:

First we determine those instructions in the IC that are leaders, that is the first instructions in each basic block. Instructions just past the end of the intermediate program is not included as a leader. The rules for finding leader are:

1. The first 3-address instructions in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions upto but not including the next leader or end of intermediate program.

Page No.
Date

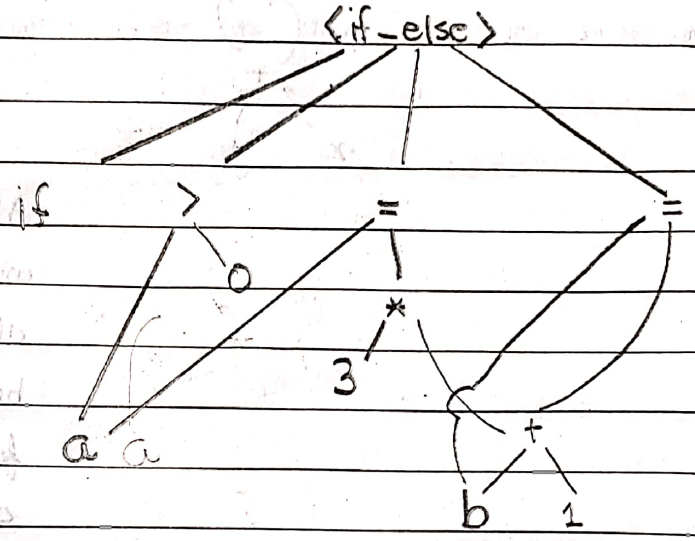
B. Give syntax tree, Directed acyclic graph (DAG) And three address code for expression If (a>0) then a= 3*(b+1) else b = b+1. 2018S [5]

if (a>0)
a = 3* (b+1)
else
b = b+1

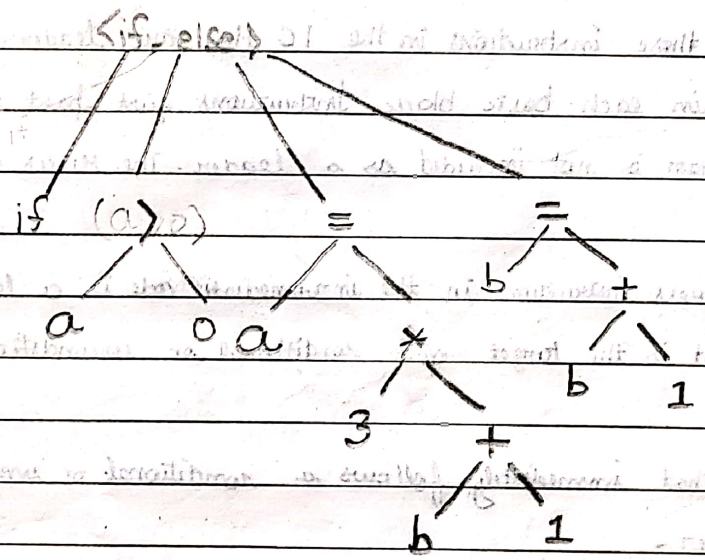
3 Address Code

1. if a>0 goto
2. goto 7
3. $t_1 = b+1$
4. $t_1 = 3 * t_1$
5. $a = t_1$
6. goto next
7. $t_1 = b+1$
8. $b = t_1$
9. goto next.

DAG



SYNTAX TREE

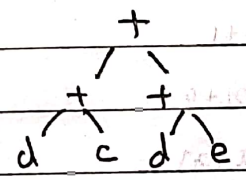


Page No.
Date

Q.No. 5
A. Explain how commutativity and associativity can be used to generate more efficient code from the DAG by taking suitable source code. [4]
2019S

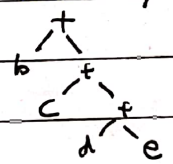
Commutativity refers to the property where the order of operands in an operation does not affect the result. (i.e ((a+b) = (b+a)))
It allows reordering of elements to optimize hardware usage, like aligning memory accessess or enabling SIMD instructions.

Eg.) a = b + c + d + e = (b+c) + (d+e)



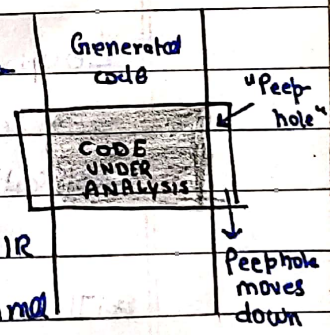
Associability describes the property where the grouping of operands does not alter the result. (i.e. (a+b)+c = a+(b+c))
The compiler can rearrange operations in the DAG to combine multiple additions or multiplications, minimize temp. variables, and improve instruction-level parallelism

Eg.) a = b + c + d + e



b. Peephole optimization 2019 S

Peephole optimization is done by one or more passes over the entire IR in a linear fashion. It inspects and manipulates a smaller portion of IR at a time.



Peephole optimization is a method that identifies specific patterns in the IR and replaces them with simpler or more efficient ones. It operates minimal additional memory req. and is known for its speed and compactness. Ideal for 3,4-tuples IR structures.

It uses the following:

1. Removal of redundant Load/Store instr.
2. Removal of unreachable code
3. Removal of GoTo instructions.
4. Arithmetic manipulations like constant folding, strength reduction and algebraic simplification
5. Process code specific instructions.

UNIT - 4

SYMBOL TABLES

Page No.
Date

Q5. a) Explain the different data structures for symbol table implementation and compare them? 2018 E [7]

Q.No. 7

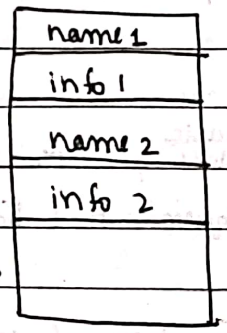
A. What are the various approaches for symbol table organization? Explain with examples. [3]

2018 S

Q.No. 6

A. Explain various data structures used for symbol table and also compare the midsquare method and folding method used for generating hash values. 2023E ✓ [4] [CO#3]

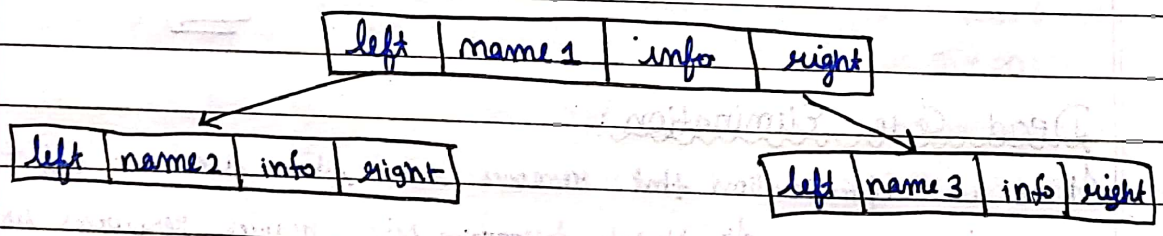
1. The Linear List



- Simplest implementation, adding new names as they arrive
- Searching is done sequentially, resulting in average of (n+1)/2 comparisons for successful search and n comparisons for unsuccessful search. (n is # records)

- Advantage: takes less space and additions are simple
- Disadvantage: Higher access time

2. Search Tree

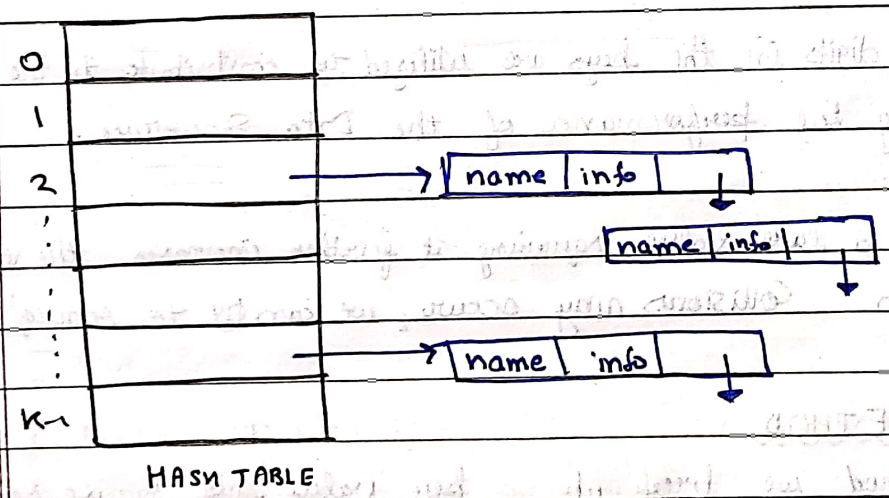


- A search tree is more efficient than linear lists
- Each record has left and right links pointing to other records in tree
- Provides alphabetical accessibility names before the current one are accessed via left links, names after successful search.
- Expected time for n entries and m queries is $(n+m)\log(n)$ so it is better for larger records.

Page No.
Date

3. Hash Tables

- They use a hash function to map names to indexes in a tables of K pointers.
- Each index points to a list of records in the symbol table
- To insert a name, hash it to find the idx, then add the record to the head of that list.
- To retrieve information, hash the name, then search the corr. list.
- It provides faster access than linear lists in comparison



Q3. a) What is the principal criterion of choosing a hash function? Explain any three hashing functions. 2017E [7]

The principal criterion for choosing a hash function ~~is~~ is collision resistance, which means it should be difficult for two different inputs to produce the same hash value.

Others include consistency and efficiency.

Page No.
Date

Q.No. 6
A. Explain the midsquare method and folding method used for generating hash values. [4]

2019 E

MID SQUARE METHOD

It is the process of computing the hash value. Given a {key : value} pair, the Hash Function would be calculated by :

1. Square the key → key * key
2. Choose some digits from the middle of the number to obtain the Hash value.

select mid 2 digits.

Suppose the size of the hash table is 10 and the KVP are
{ 10 : 'Sudha', 11 : 'Venkat', 12 : 'Jeevani' }

so $H(10) = 10 * 10 = 100 = 0$

$H(11) = 11 * 11 = 121 = 2$

$H(12) = 12 * 12 = 144 = 4$

Here, all the digits in the keys are utilized to contribute to the index, thus increasing the performance of the Data Structure.

If the key is a large value, squaring it further increases the value, which is a con. Collisions may occur, we can try to reduce/handle it.

FOLDING METHOD

In this method we break up a key value into precise segments that are added to form a hash value, and look at another technique is to apply a multiplicative hash function to each segment before addition. Its magnitude of distribution

- Folding method for has functions divide item into equal sized pieces.
- Sum of these peices give a hash value. $H(x) = (a+b+c) \% M$
- Remainder is hash key.

Page No.
Date

DIVISION METHOD

This is the most simble method to generate hash value. It divides the value k by M and use the remainder.

$h(k) = k \% M$

M is a prime number for more uniform distribution.

5) Explain the three storage allocation techniques? 2017E [7]

Storage allocation is the process of assigning memory addresses to different data structures such as variables, arrays and objects.

b) Explain the static storage allocation strategy? 2018G [7]

1.) **STATIC ALLOCATION**

In static allocation, all variables that are created are assigned to memory at compile time and the addresses of these variables will remain the same throughout the program's execution.
As the size of variables does not vary much so this allocation is easy to understand and efficient, but this allocation is not flexible or scalable.

2.) **DYNAMIC ALLOCATION**

Dynamic allocation is a type of storage strategy where the memory is allocated to the data structure at run-time, not at compile time.
The DS is allocated in memory depending on the program's need and can be allocated or deallocated at the program's execution.
It can be

→ Stack Based : Fast memory alloc and dealloc on a fix size stack

→ Heap Based : DMA in heap, with flexibility but with potential for slower speed and memory leaks.

Page No.
Date

3. HYBRID ALLOCATION

This is a type of allocation strategy which is used to achieve multiple memory allocation. A compiler can use static memory allocation for global variables and heap memory allocation for dynamic DS

∴ It combines the benefits of stack and heap allocation for improved performance but has complexity with difficulty for developers due to its combination nature.

UNIT-5: ERROR DETECTION AND RECOVERY

Page No.
Date

Q6. a) Discuss the desirable features of error reporter? 2017E [7]

Q6. a) Discuss the desirable features of error reporter? 2018E [7]

1. The message should be produced in terms of the original source program rather than in terms of some internal representation.
   Ex:) the message should be produced along with the line number of the source program.

2. The error message should be easy to understand by the user.

3. The error message should be specific and should localize the program
   Ex:) an error message should read, "n is not declared in the function", and not just "Missing declaration"

4. The message should not be redundant; i.e., the same message should not be produced again and again.

Q.No. 7
A. Explain error recovery strategies adopted by complier. [4] [CO#5]
b) Explain error recovery strategies adopted by complier? 2018G [7]

Page No.
Date

Q.No. 2
A. What are the different error recovery strategies adopted in complier ?explain 2018 S [3]

Q.No. 7
A. Explain error recovery strategies adopted by complier. 2019E [4]

Q.No. 7
A. Explain error recovery strategies adopted by complier: 2018 G [4]

b) Explain panic-mode error recovery strategy? 2017E [7]

1.) PANIC MODE

This strategy is used by most parsing methods. This is simple to implement.

In this method on discovering error, the parser discards input symbols one at a time. This process is continued till one of a designated set of synchronizing tokens is found.
Synchronization tokens are delimiters such as semicolon or end. These tokens indicate an end of input statement.

Thus, in panic mode recovery a considerable amount of input checking is done for additional errors. If there is less number of errors in the same statement then this strategy is best choice.

Ex) int a, ~~[illegible]~~, sum, ~~[illegible]~~;

2.) PHRASE LEVEL RECOVERY

In this method, on discovering an error, parser performs local correction on remaining input. It can replace a prefix of remaining input by some string. This actually helps parser to contine its job

The local correction can be replacing comma by semicolon, deletion of semicolons or inserting missing semicolons. This type of local correction is decided by compiler designer.

Page No.
Date

While doing the replacement a care should be taken for not going in an infinite loop.

3. ERROR PRODUCTION

If we have good knowledge of common error that might be encountered, then we can augument the grammar for the corr. language with error productions that generate erroneous constructs.

- If error production is used during parsing, we can generate appropriate error message to indicate the erroneous construct that has been recognized in the input. This method is extremely difficult to maintain, because if we change grammar then it becomes necessary to change the corresponding productions.

Ex.) Suppose the input string is abcd.

∴ Grammar :	Grammar :
$S \rightarrow A$	$E \rightarrow SB$
$A \rightarrow aA \mid bA \mid a \mid b$	$S \rightarrow A$
$B \rightarrow cd$	$A \rightarrow aA \mid bA \mid a \mid b$
	$B \rightarrow cd$.

4. GLOBAL CORRECTION

We often want such a compiler that makes very few changes in processing an incorrect input string.

Given an incorrect input string x and grammar G, the algorithm will find a parse tree for a related string y, such that no. of insertions, deletions and changes of token require to transform x into y as small as possible.

Such Methods increase time and space req at parsing time.

Its a theoritical concept

b. LEX and YACC

b. LEX and YACC 2019E

B. What is LEX? How it is different from YACC tool. 2018E [4]

LEX (a lexical Analyzer Generator) is one of the compiler writing tools, that is used to generate a lexical analyzer or scanner from the description of tokens of the programming language to be implemented. And this description is required ~~to~~ as regular expressions. Therefore, LEX is a scanner generator used to generate a scanner automatically from the description of tokens as regex.

YACC (yet another compiler compiler) is a parser generator, which is a program that takes as its input a specification of the syntax of the programming language, and produces as its output a parse procedure for that language whose name is yyparse(). The notation used for preparing this specification is a grammar (CFG)

FEATURE	LEX	YACC
PURPOSE	Generates a lexical analyzer or scanner.	Generates a parser.
INPUT	Regular expressions describing tokens.	CFG rules.
OUTPUT	C code for yylex() function.	C code for yyparse() function.
PROCESSING	Tokenizes input into a stream of tokens.	Parses the stream of tokens according to the grammar rules.
USE IN COMPILER	Lexical analysis phase	Syntax analysis phase
NOTATION	Regular Expressions	Context-free grammars (CFG)
Output file	lex.yy.c / lexyy.c	y.tab.c / ytab.c

Total No. of Pages: 01 *Roll No.....*

VI-SEMESTER *B.Tech.(CO)*

MID SEMESTER EXAMINATION March- 2024

CO302 Compiler Design

Time: 1:30 Hours Max. Marks: 20

Note: Attempt all questions

Q.No. 1 [4] CO1

a. Explain Handle and Handle pruning with suitable example.

b. Remove left recursion from following grammar if any

S→ Aa | b S→ Ac |Sd |e

c. Compute FOLLOW of C in the following grammar

S→ ABC A→ aCAb/€ B→ bBCc/€ C→ abc

d. Describe various phases of compiler design with example

Q.No. 2

What is cross compiler? Explain and test whether the following grammar is LL(1) or not? [4] CO2

S→ aBC | €
B→ aBD |bD
C→ bS
D→ a (where 'S' is start symbol)

Q.No. 3

Compute FIRST and FOLLOW and construct predictive parsing table for the following grammar: [4] CO2

S → aYXh
Y → Yb|c
X→ EF
E→ g |€
F→ f |€ (where 'S' is start symbol)

Q.No. 4

Compute canonical collection of sets of LR(1) items and Construct canonical LR (CLR) parsing table for following grammar. [4] CO4

S → Da |bDc |dc |bda
D→ d , (where 'S' is start symbol)

Q.No. 5

Consider the grammar { S→BS|0 , B→SB|1 }. Construct the SLR parse table for the grammar. Show the actions of the parser for the input string"1010".{S is start symbol} [4] CO4

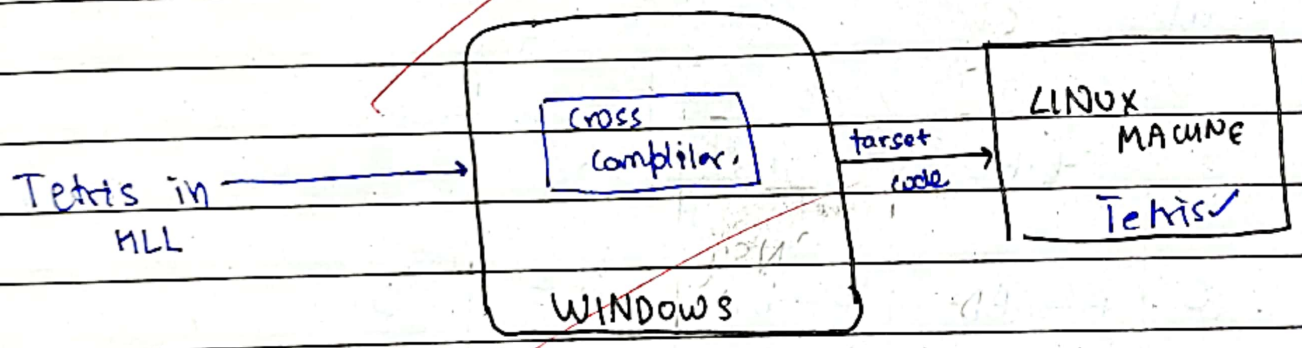
ANSWER 2

CROSS COMPILER

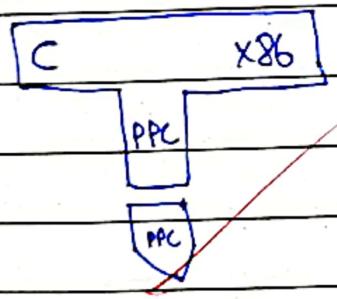
A cross compiler is a compiler program which is written for one machine but produces target machine code for another machine.

It is characterized by 3 languages:

→ Source language
→ Target Language
→ Implementation language



Eg.)



P.T.O.

Given Grammar,

$S \to aBC \mid \epsilon$

$B \to aBD \mid bD$

$C \to bS$

$D \to a$.

$S \to aBC$ — (1)

$S \to \epsilon$ — (2)

$B \to aBD$ — (3)

$B \to bD$ — (4)

$C \to bS$ — (5)

$D \to a$ — (6)

to test if it is LL(1), we make table for predictive parser.

First (S) = $\{a, \epsilon\}$

First (B) = $\{a, b\}$

First (C) = $\{b\}$

First (D) = $\{a\}$

Follow (S) = Follow (C) = $\{\$\}$

Follow (B) = First (D) $\cup$ First (C) = $\{a, b\}$

Follow (C) = Follow (S) = $\{\$\}$

Follow (D) = Follow (B) = $\{\$\}$

∴ Table.

	a	b	$
S	$S \to aBC$		$S \to \epsilon$
B	$B \to aBD$	$B \to bD$	
C		$C \to bS$	
D	$D \to a$		

4

There are no conflicts of SR / RR form in the parsing table.

Hence grammar is LL(1).

ANSWER-3

Given

$S \rightarrow aYXh$

$Y \rightarrow Yb \mid c$

$X \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

We need to remove left recursion in $Y \rightarrow Yb$

$\therefore$ we do this. $Y \rightarrow cY'$

$Y' \rightarrow bY' \mid \epsilon$

$\therefore$ New grammar,

$S \rightarrow aYXh$

$Y \rightarrow cY'$

$Y' \rightarrow bY' \mid \epsilon$

$X \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$.

① Finding FIRST of non terminals.

First (S) = {a}.

First (Y) = {c}

First (Y') = {b, ε}

First (X) = {g, f, ε}

First (E) = {g, ε}

First (F) = {f, ε}

② Finding Follow of NT

Follow (S) = {$}

Follow (Y) = {g, f, h}

Follow (Y') = {g, f, h}
= Follow(Y)

Follow (X) = {h}

Follow (E) = {f} ∪ Follow (X) = {f, h}

Follow (F) = {h}

	a	b	c	g	f	h	
S	$S \to aYXh$	–	–	–	–	–	–
Y	–	–	$Y \to cY'$	–	–	–	–
Y'	–	$Y' \to bY'$	–	$Y' \to \epsilon$	$Y' \to \epsilon$	$Y' \to \epsilon$	–
X	–	–	–	$X \to EF$	$X \to EF$	X	$E \to EF$
E	–	–	–	$E \to g$	$E \to \epsilon$	$E \to \epsilon$	–
F	–	–	–	–	$F \to f$	$F \to \epsilon$	–

PREDICTIVE PARSING TABLE

3

ANSWER-1

a.) Handle

Handle is a any substring which occurs on the right-hand side of a production rule in the given grammar.

Eg.) $w = abcd$, $A \to cd$ } then substring 'cd' is a handle.

Handle Pruning

It is the process of replacement of a handle from the top of stack with the appropriate corresponding Non Terminal which produces that handle.

1

Eg.)

STACK	ACTION
$ a b c	Reduce S → cba
$ S	accept

b.) The question is ambiguous so considering the grammar.

$S \rightarrow Aa|b$

$A \rightarrow Ac|Sd|e$

∴ we can see indirect left recursion, so the grammar can be written as.

$S \rightarrow Aa|b$

$A \rightarrow Ac|Aad|bd|e$

$\frac{1}{2}$

we need to remove Left Recursion by this rule.

$A \rightarrow A\alpha_1|A\alpha_2|\beta_1|\beta_2$

can be written as

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

$A' \rightarrow \alpha_1 A'|\alpha_2 A'|\epsilon$.

here, $\beta_1 = bd$ | $\beta_2 = e$

$\alpha_1 = c$ | $\alpha_2 = ad$.

∴ New grammar:

$S \rightarrow Aa|b$

$A \rightarrow bdA'|eA'$

$A' \rightarrow cA'|adA'|\epsilon$

⇒ Without Left Recursion

c.) Given grammar,

$S \rightarrow ABC$

$A \rightarrow aCAb$

$A \rightarrow \epsilon$

$B \rightarrow bBCc$

$B \rightarrow \epsilon$

$C \rightarrow abc$

① we calculate FIRST

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

First (A) = $\{a, \epsilon\}$

First (B) = $\{b, \epsilon\}$

First (C) = $\{a\}$

② we calculate Follow

Follow (S) = $\{\$\}$ ① S is start symbol.

Follow (A) = First (b) ∪ First (B) = $\{b\} \cup \{b \cup$ First (c)$\}$

= $\{b, a\}$

Follow (B) = $\{c\} \cup$ First (C) = $\{c, a\} \cup$ First(A) = $\{a, c\} \cup$ First (b)

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

Follow (C) = Follow (S) ∪ First (A) ∪ First (c)

= $\{\$, c, a\} \cup$ First (b) = $\{a, b, c, \$\}$

Thus,

Follow (S) = {$}

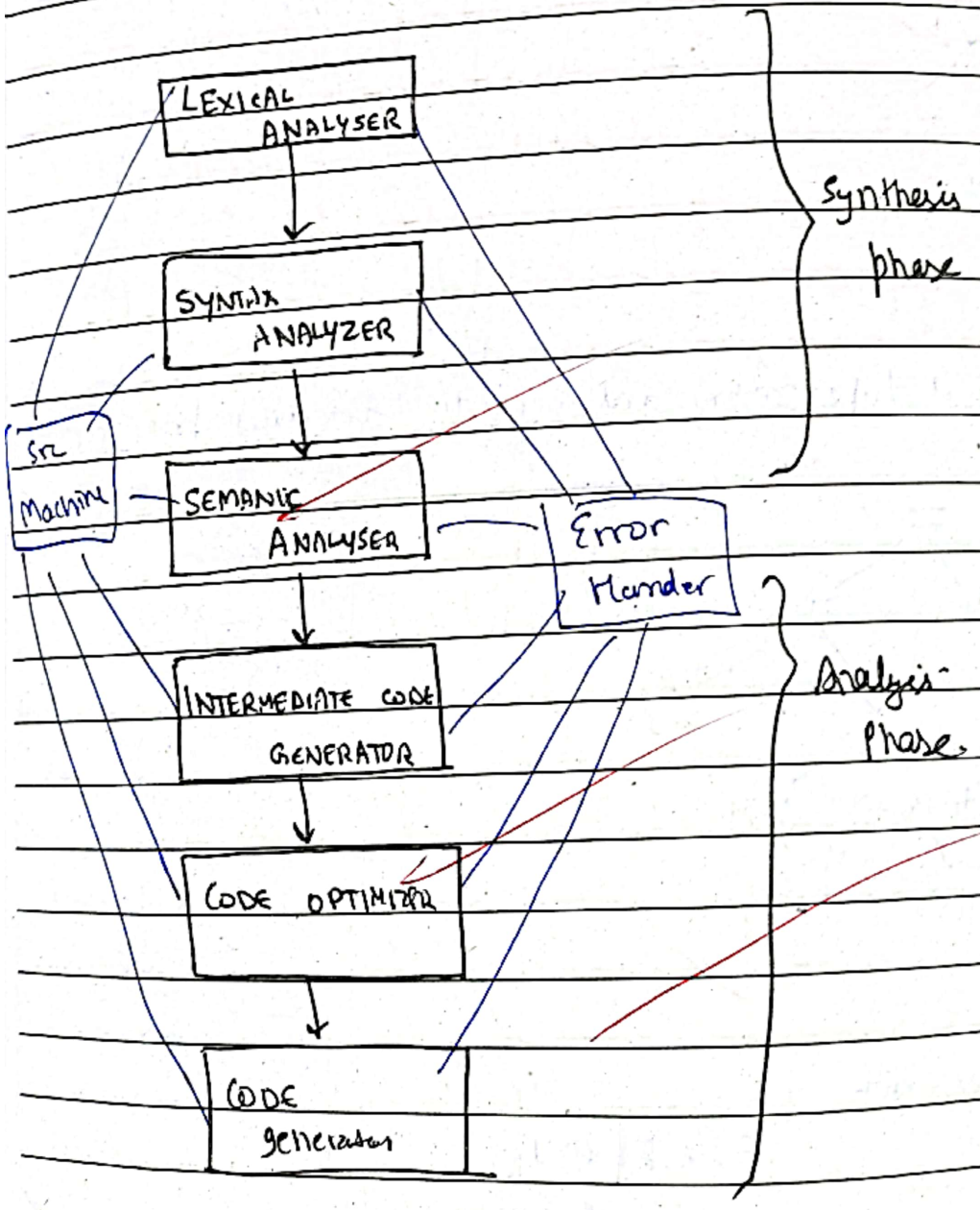
Follow (A) = {a, b}

Follow (B) = {a, b, c}

Follow (C) = {a, b, c, $} ← Follow of C.

1

d.) PHASES OF COMPILER



Eg.) $x = 10 * y + 5.5$

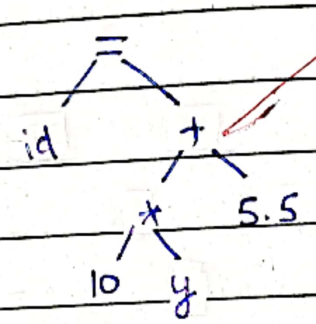
LEXICAL ANALYZER

This phase breaks down the statement into token lexemes.

E.) <id, identifier> <assign op> <int, 10> <id_2, y> <add op> <int, 5>

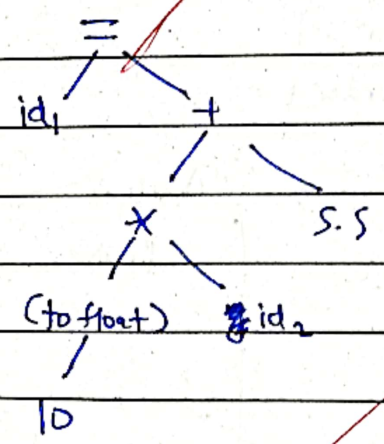
SYNTAX ANALYZER

It checks the grammar and makes syntax tree.



SEMANTIC ANALYSER

It takes care of type casting and generate annotated parse tree



INTERMEDIATE CODE GEN

It generates assignment.

$E_1 = E_2$

$E_2 = E_3 + 5.5$

$E_3 = E_4 * id_2$

$E_4 = \text{to float}(10)$

Code optimizer
It optimizes the assignments.

Code Generator
It gives assembly code in form of register assign

Eg MOL R_1 R_2
ADD R_1 S

ANSWER-4

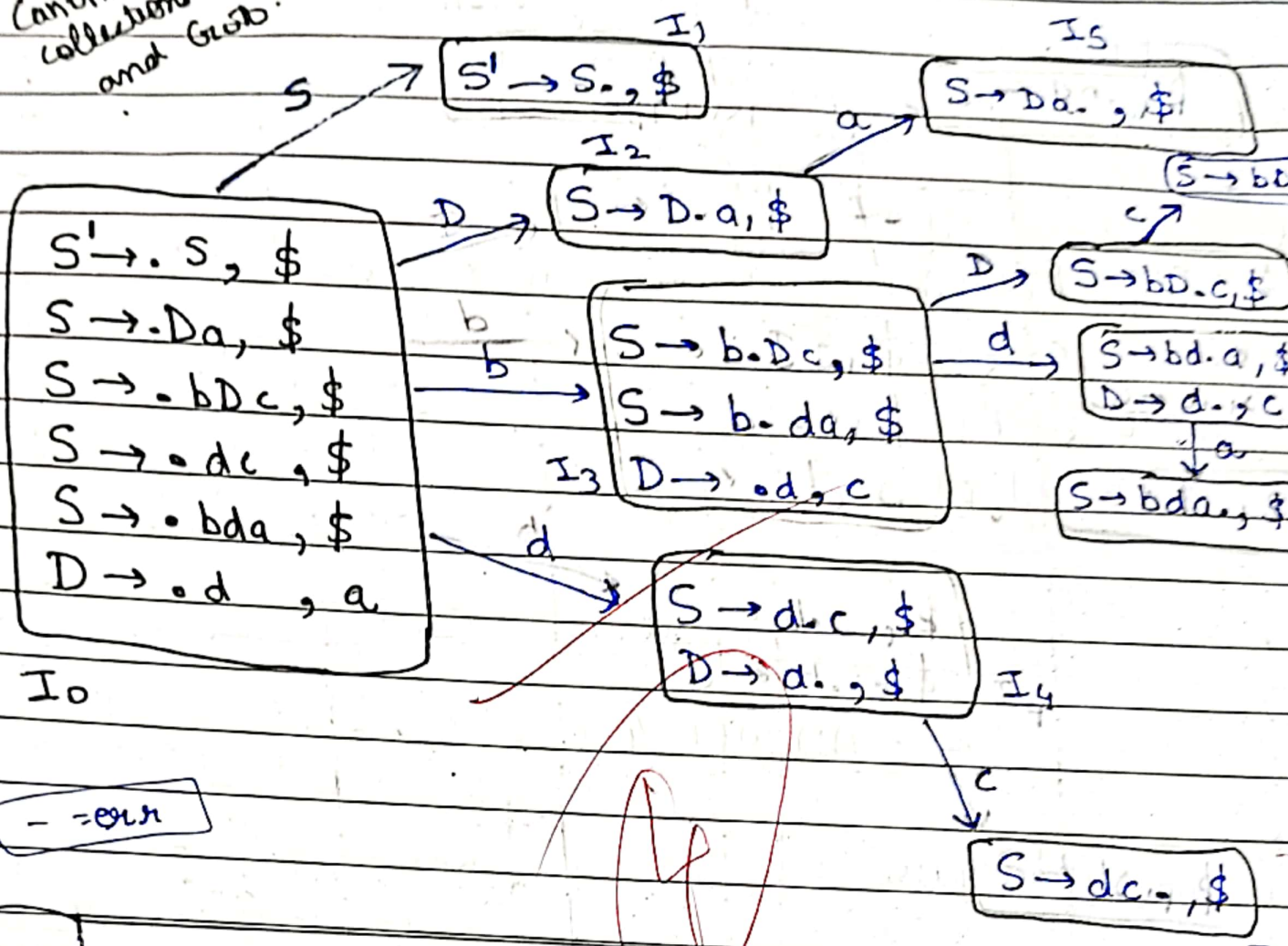
Augumented Grammar

$S' \rightarrow S$
$S \rightarrow Da \mid bDc \mid dc \mid bda$
$D \rightarrow d.$

numbering
$\therefore S' \rightarrow S$ (1)
$S \rightarrow Da$ (2)
$S \rightarrow bDc$ (3)
$S \rightarrow dc$ (4)
$S \rightarrow bda$ (5)
$D \rightarrow d$ (6)

P.T.O.

Canonical collections and GoTo.



– = error

LR(1) items	ACTION					GoTo		
	a	b	c	d	$	S	D	
I_0	–	S_3	–	S_4	–	1	2	
I_1	–	–	–	–	accept	–	–	
I_2	S_5	–	–	–	–	–	–	
I_3	–	–	–	S_7	–	–	6	
I_4	–	–	S_8	–	r_6	–	–	CLR parsing table
I_5	–	–	–	–	r_2	–	–	
I_6	–	–	S_9	–	–	–	–	
I_7	S_{10}	–	r_6	–	–	–	–	
I_8	–	–	–	–	r_4	–	–	
I_9	–	–	–	–	r_3	–	–	
I_{10}	–	–	–	–	r_5	–	–	