13)	1)	With the help of neat block diagram explain various phases of compiler ,Also write down the output of each phase for expression a:=b+c*50
	2)	Construct LR(1). S→x|Ay B→ε|z A→Bx

PHASES OF COMPILER:

Lexical Analysis:

Lexical analyser divides the program into tokens scanning

eg $a = b + 5 - (c * d)$

Token Type	value
Identifier	a, b, c, d
operator	+, −, *
constant	5
Delimeter	(,)

Syntax Analysis:

* It takes list of tokens produced by lexical Analysis.
* Then, these tokens are arranged in a tree like structure (syntax tree), which reflects program structure
* Also known as parsing

Semantic Analysis:

* It validates the syntax tree by applying rules & regulations of the target language.
* It does type checking, scope resolution, variable declaration, etc.
* It decorates the syntax tree by putting data types, values, etc.

Intermediate Code Generation:

* The program is translated to a simple machine independent intermediate language
* Register allocation of variables is done in this phase.

Code optimization:

* It aims to reduce process timings of any program
* It produces efficient programming code.
* It is an optional phase.
* Removing unreachable code
* Getting rid of unused variables
* Eliminating multiplication by 1
  addition by 0
* Removing statements that are not modified from the loop.
* Common sub-expression elimination

Code Generation

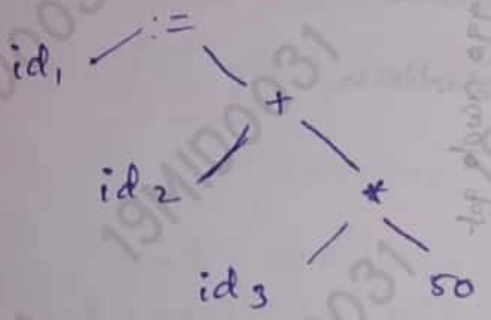
* Target program is generated in the machine language of the target architecture.
* Memory locations are selected for each variable.
* Instructions are chosen for each operation
* Individual tree nodes are translated into sequence of machine language instructions

all others → float Price := amount + rate * 50 → int

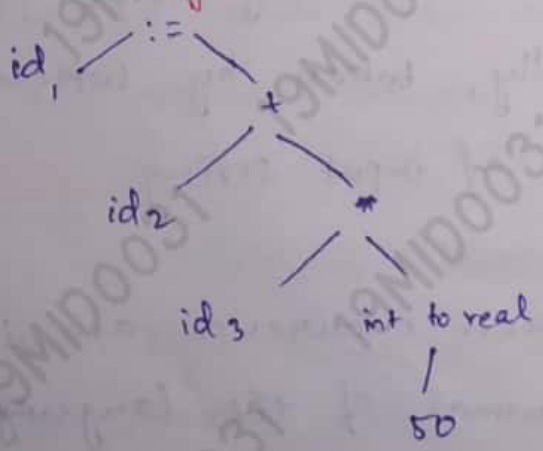
1) Lexical Analysis: removes spaces, comments

$id_1 := id_2 + id_3 * 50$

2) Syntax Analyser:



3) Semantic Analyser:



4) Intermediate code Generator:

```
temp1 := int to real (50)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

5) Code optimizer:

```
temp1 := id3 * 50.0
id1 := id2 + temp1
```

6) Code Generator:

```
MOVF  R2, id3
MULF  R2, #50.0
MOVF  R1, id2
ADDF  R1, R2
MOVF  id1, R1
```

2) Construct LR(1)

$S \rightarrow x \mid Ay$

$B \rightarrow \varepsilon \mid z$

$A \rightarrow Bx$

Augmented Grammar

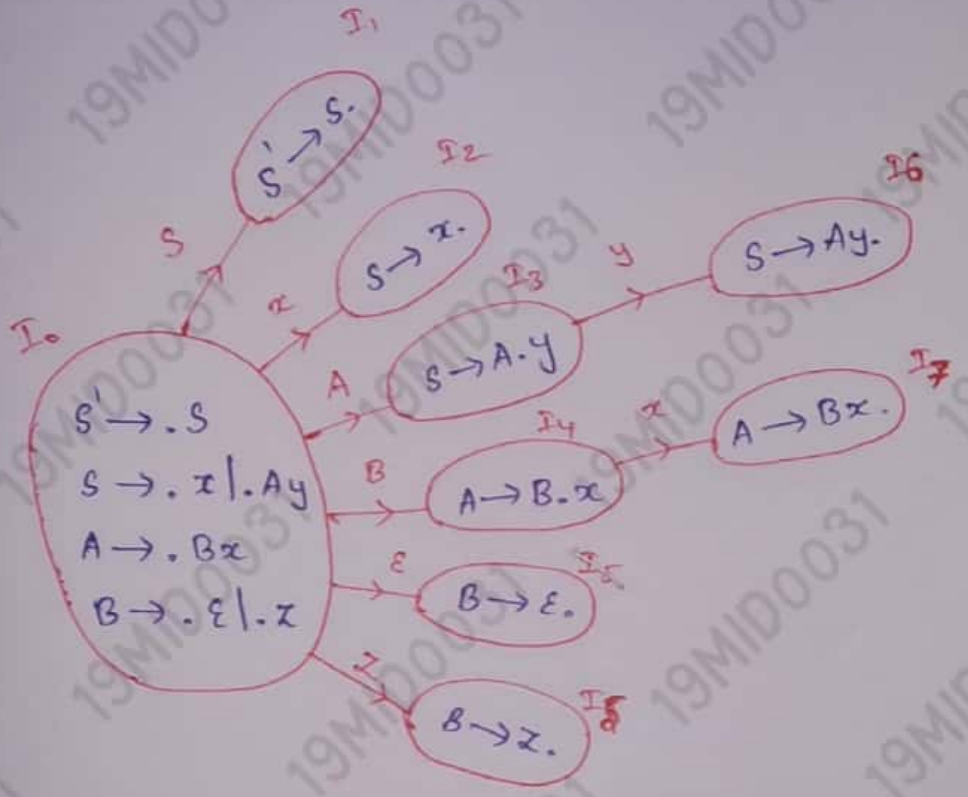
$S' \rightarrow S$

$S \rightarrow x \mid Ay$: r_1 and r_2

$B \rightarrow \varepsilon \mid z$: r_3

$A \rightarrow Bx$: r_4

canonical forms



	ACTION				GOTO		
	x	y	z	$	A	B	S
0	S_2		S_5		3	4	1
1				accept			
2				r_1			
3		S_6					
4	S_7						
5				r_3			
6				r_2			
7				r_4			

35) 1) Consider the grammar

S→(L)|a

L→L,S|S

f) What are the terminal, non-terminal and start symbol?

g) Find parse tree for the following sentences

(iv) (a,a)

(v) (a,(a,a))

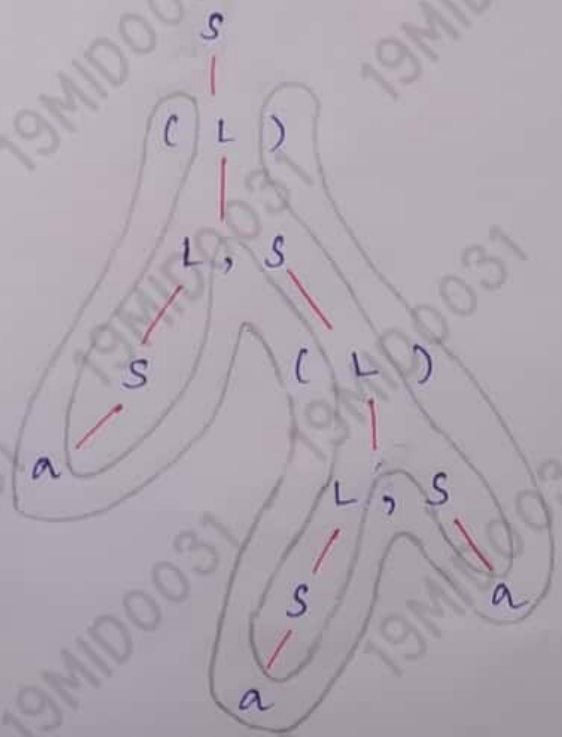
(vi) (a,((a,a),(a,a)))

35) $S \rightarrow (L) \mid a$

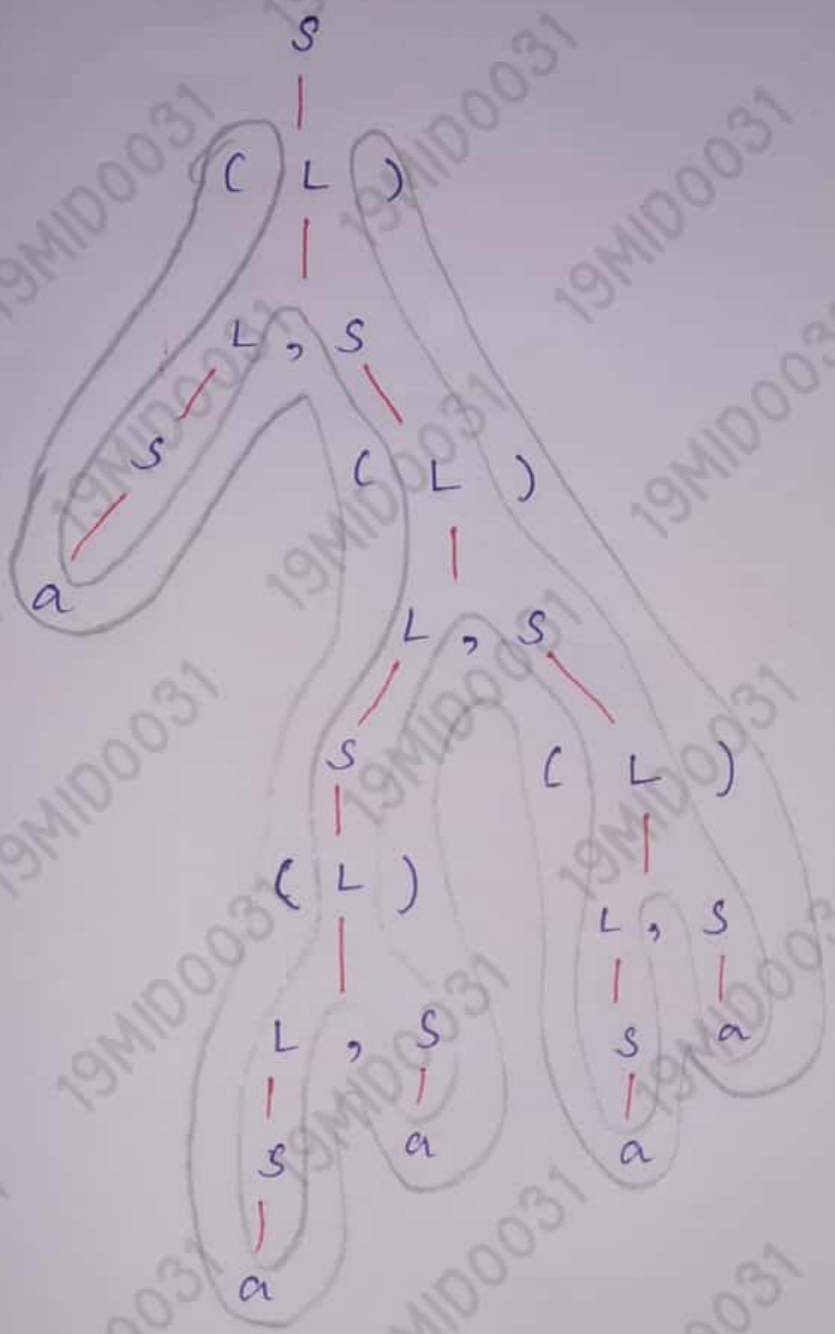
$L \rightarrow L, S \mid S$

* teaminals = { (,), , , a }
* Non-teaminals = { S, L }
* Start symbol = {S}
* parse tree for (a, a)

* parse tree for (a, (a, a))



* parse tree for (a, ((a, a), (a, a)))



40)

1) What are the issues of the Lexical analyser?
2) Eliminate left recursion, perform left factoring and find:
   FIRST & FOLLOW
   E→E+T|T
   T→id|id [ ] |id [X]
   X→E,E|E
3) Check the given grammar is LL(1) or NOT?
   S→(A) | 0
   A→SB
   B→,SB|ε and also parse the grammar (0,(0,0))

ISSUES IN LEXICAL ANALYSIS

We do seperate the work of Lexical Analysis and syntax Analysis for the following reasons.

* Simplicity of design

A parser containing the rules for comments and white space is more complex to make than a parser that can assume that comments & white spaces has been removed

* Improved compiler Efficiency:

Reading source code & classifying it in token is time consuming task. when we separate from parser, it allows us to use specialized technique for lexer, which can speed up scanning.

* Higher probability:

Input device specific peculiarities are restricted to lexer

Lexical Errors:

A character sequence which is not possible to scan into. any valid token is a lexical error.

It's hard for lexical analyzer without the aid of other components, that there is a source code error.

Eg: If the statement "if" is encountered for the first time in a C program, it can not tell whether fi is misspelling of "if" statement or a undeclared literal.

Probably the parser in this case will be able to handle this

* Also Error Handling is very localized with respect to input source.

Eg: whil (x=0) do generates no lexical error is PASCAL.

Handling Lexical Errors:

* Panic Mode Recovery

Delete successive characters from the remaining input until the analyzer can find a well formed token.

→ May confuse parser by creating syntactical errors.

* possible error Recovery Actions:

→ Deleting extra irrelavent character
→ Inserting missing input character
→ Replacing an incorrect character by a correct character
→ Transposing two adjacent characters

Input Buffering:

The amount of time taken to process characters of a large source program.

Lexical analyzer may need to look at least a character ahead to make a token decision

Sentinels:

During buffering for each character.

→ check the end of buffer

→ determine what character is read

40) 2) $E \rightarrow E+T \mid T$ — (1)

$T \rightarrow id \mid id[] \mid id[x]$ — (2)

$X \rightarrow E, E \mid E$ — (3)

* First production has left Recursion.

$E \rightarrow E+T \mid T$

Removing left Recursion.

$E \rightarrow TE'$

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

* Second production has Left factoring

$T \rightarrow id \mid id[] \mid id[x]$

Removing left factoring

$T \rightarrow id\,T'$

$T' \rightarrow \varepsilon \mid [] \mid [x]$

Now the productions are

$E \rightarrow TE'$ —— (1)

$E' \rightarrow \varepsilon \mid TE'$ —— (2)

$T \rightarrow idT'$ —— (3)

$T' \rightarrow \varepsilon \mid [\,] \mid [X]$ —— (4)

$X \rightarrow E,E \mid E$ —— (5)

FIRST(E) = { id }

FIRST(E') = { ε, id }

FIRST(T) = { id }

FIRST(T') = { ε, [}

FIRST(X) = { id }

FOLLOW(E) = { $, , ,] }

FOLLOW(E') = { $, , ,] }

FOLLOW(T) = { id, $, , ,] }

FOLLOW(T') = { id, $, , ,] }

FOLLOW(X) = {] }

A0) 3) $S \rightarrow (A) \mid 0$

$A \rightarrow SB$

$B \rightarrow ,SB \mid \varepsilon$

No Left recursion nor Left factoring

FIRST(S) = {(, 0}

FIRST(A) = {(, 0}

FIRST(B) = {, , ε}

FOLLOW(S) = {, ,), 0, $}

FOLLOW(A) = {)}

FOLLOW(B) = {)}

	S	A	B
(	S→(A)	A→SB	
)			B→ε
,			B→,SB
0	S→0	A→SB	

Stack	Input	production
S$	(0,(0,0))	
(A)$	(0,(0,0))	S→(A)
SB)$	0,(0,0))	A→SB
0B)$	0,(0,0))	S→0
,SB)$	,(0,0))	B→,SB
(A)B)$	(0,0))	S→(A)
SB)B)$	0,0))	A→SB
0B)B)$	0,0))	S→0
,SB)B)$	,0))	B→,SB
0B)B)$	0))	S→0
)B)$	))	B→ε
)$	)	B→ε

Now stack is empty with $, so accepted

23) 1) Define handle and handle pruning?

2) Construct LR parsing table

E→E+T

E→T

T→T*F

T→F

F→(E)

F→id *id*id+id* using stack implementation.

23) Handles
1)

* Formally, Handle of a right sentential form γ is $\langle A \to \beta$ location of β in $\gamma \rangle$

* i.e) $A \to B$ is a handle of $\alpha\beta\gamma$ at the location immediately after the end of α, if $S \Rightarrow \alpha A\gamma \Rightarrow \alpha\beta\gamma$

* A certain sentential form may have many different handles.

* Right sentential forms of a non-ambiguos grammar have one unique handle.

Handle pruning

* the process of discovering a handle & reducing it to the appropriate left hand side is called handle pruning

* Handle pruning forms the basis for a bottom-up parsing method.

23) 2) $E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id$ A.

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				accept			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Stack	Input	production
0	id * id + id $	s5
0 id 5	* id + id $	r6
0 F 3	* id + id $	r4
0 T 2	* id + id $	s7
0 T 2 * 7	id + id $	s5
0 T 2 * 7 id 5	+ id $	r6
0 T 2 * 7 F 10	+ id $	r3
0 T 2	+ id $	r2
0 E 1	+ id $	s6
0 E 1 + 6	id $	s5
0 E 1 + 6 id 5	$	r6
0 E 1 + 6 F 3	$	r4
0 E 1 + 6 T 9	$	r1
0 E 1	$	accept

24)

1) Differentiate between final states in a NFA and a DFA
2) Table:

Remove left recursion	Remove left Factoring
A→Aα |β	S→iEtS|iEtSeS|a E→b
S→Aa|b A→Ac|Sd|ε	Stmt→if expr then Stmt else Stmt|ifexpr then Stmt
S→aBDh S→Bb|C D→EF E→g|ε F→f|ε	S→aSb|aTc T→dTU|ε U→f
S→SA|SB|a|b|c	

24)

1) FINAL STATE OF DFA

$M = \{Q, \Sigma, \delta, q_0, F\}$

$Q \rightarrow$ set of states
$\Sigma \rightarrow$ alphabets
$\delta \rightarrow$ transition function
$q_0 \rightarrow$ Initial state.
$F \rightarrow$ Final state

Final state => It is non empty set of final states/ accepting states from the set belonging to Q

FINAL STATE OF NFA

$M = \{Q, \Sigma, \delta, q_0, F\}$

$Q \rightarrow$ set of states
$\Sigma \rightarrow$ alphabets
$\delta \rightarrow$ transition function
$q_0 \rightarrow$ Initial state
$F \rightarrow$ Final state

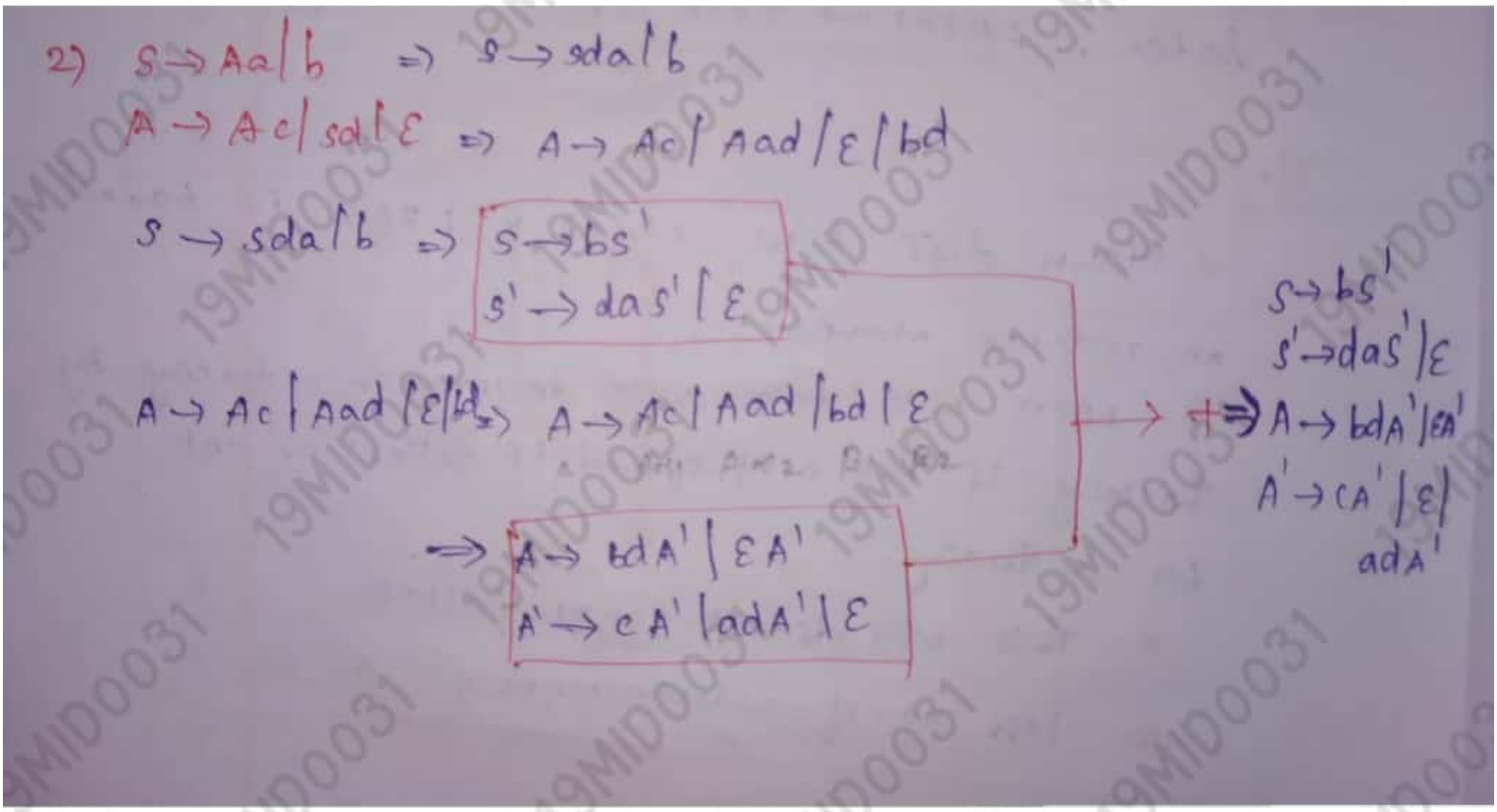
Final state => A non empty set of final states and member of Q.

24)

2) REMOVE LEFT RECURSION

(i) $A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$



(iii) $S \to aBDh$
$S \to Bb \mid C$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

No Left Recursion in this example

(iv) $S \to SA \mid SB \mid a \mid b \mid c$
$S \to aS' \mid bS' \mid c$
$S' \to AS' \mid BS' \mid \varepsilon$

REMOVE LEFT FACTORING

(i) $S \to iEtS \mid iEtSeS \mid a$
$E \to b$
$S \to iEtSS' \mid a$
$S' \to \varepsilon \mid eS$
$E \to b$

(ii) stmt → if expr then stmt else stmt | if expr then stmt
(Same as above)

Stmt → if expr then stmt Stmt'
Stmt' → ε | else stmt

(iii) $S \to aSb \mid aTc$
$T \to dTU \mid \varepsilon$
$U \to f$
$S \to aS'$
$S' \to \varepsilon \mid Sb \mid Tc$
$T \to dTU \mid \varepsilon$
$U \to f$

3) 1) Construct DAG for

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

b) x=x+x*y

c) (x+5)*(x+5+y)

d) a=(a+a)+a(a+a+a)+a

2) Check the given grammer is LL(1) or NOT?

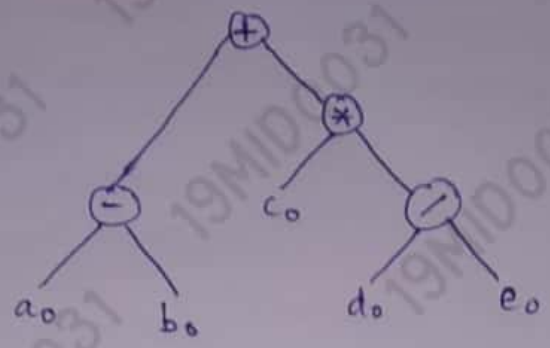
S→(A) | 0

A→SB

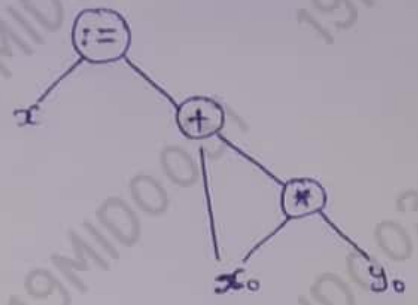
B→,SB|ε

and also parse the grammar (0,(0,0))

3) i)(a) (a-b)+c*(d/e)

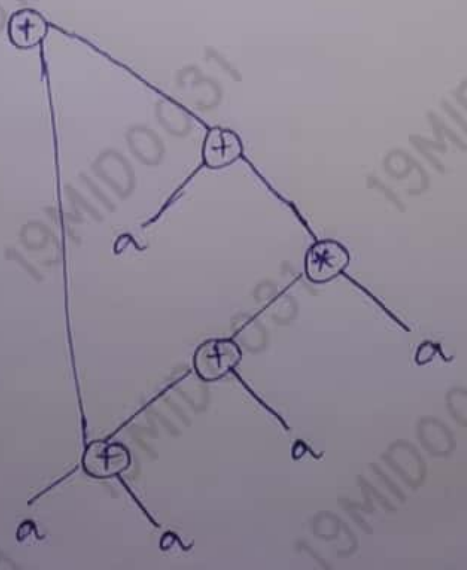


(b) x = x + x * y



(c) (x+5) * (x+5+y)

(d) a = (a+a) + a(a+a+a) + a



3) $S \rightarrow (A) \mid 0$

2) $A \rightarrow SB$

$B \rightarrow ,SB \mid \varepsilon$

No Left recursion nor Left factoring

FIRST(S) = {(, 0}

FIRST(A) = {(, 0}

FIRST(B) = {, , ε}

FOLLOW(S) = {, ,), 0, $}

FOLLOW(A) = {)}

FOLLOW(B) = {)}

	S	A	B
(	S→(A)	A→SB	
)			B→ε
,			B→,SB
0	S→0	A→SB	

Stack	Input	production
S$	(0,(0,0))	
(A)$	(0,(0,0))	S→(A)
SB)$	0,(0,0))	A→SB
0B)$	0,(0,0))	S→0
,SB)$	,(0,0))	B→,SB
(A)B)$	(0,0))	S→(A)
SB)B)$	0,0))	A→SB
0B)B)$	0,0))	S→0
,SB)B)$	,0))	B→,SB
0B)B)$	0))	S→0
B)B)$	))	B→ε
B)$	)	B→ε
$		

Now stack is empty with $, so accepted