13 1. With the help of neat block diagram explain Various phases of compiler, Also write down the output of each phase for compression

a := b + c * 50 .

2. Construct LR(1)

S → x | Ay

B → ∈ | z

A → Bx

PHASES OF COMPILER

Lexical Analysis:-

Lexical Analyser divides the program into tokens scanning.

e.g.

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 and 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:-

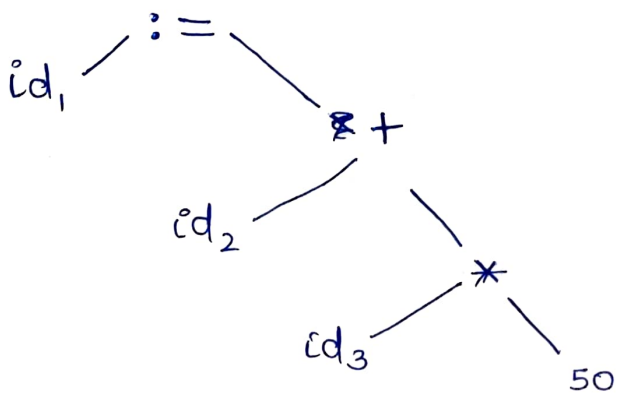
* Target program is generated in the machine language of the target architecture.
* Memory locations are selected for each variable
* Instructions are chosen for each operation
* Individual tree nodes are translated into sequence of machine language instructions

Price := amount + rate * 50

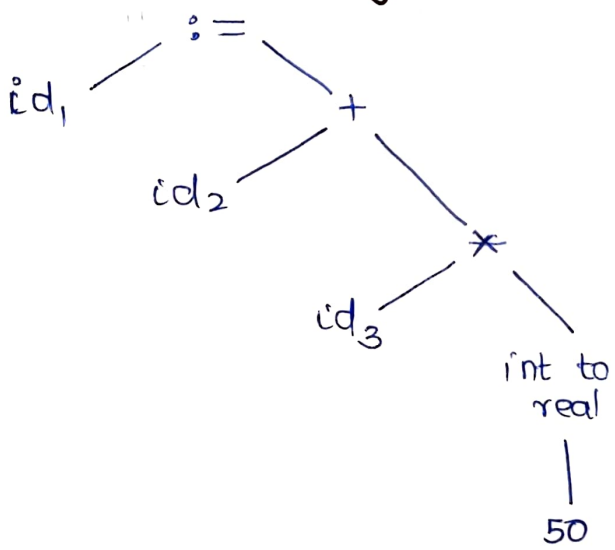
1. Lexical Analyser:

$id_1 := id_2 + id_3 * 50$

2. Syntax Analyser



3. Semantic Analyser



4) Intermediate Code Generator

temp 1 := int to real (50)
temp 2 := id_3 * temp 1
temp 3 := id_2 + temp 2
id_1 := temp 3

5) Code Optimizer

temp := id_3 * 50.0
id_1 = id_2 + temp 1

6) Code Generator

```
MOV F   R2 , id3
MULF    R2 , # 50.0
MOV F   R1 , id2
ADD F   R1 , R2
MOV F   id1 , R1
```

2) Construct LR(1)

$S \rightarrow x \mid Ay$

$B \rightarrow \epsilon \mid z$

$A \rightarrow Bx$

Augmented Grammar

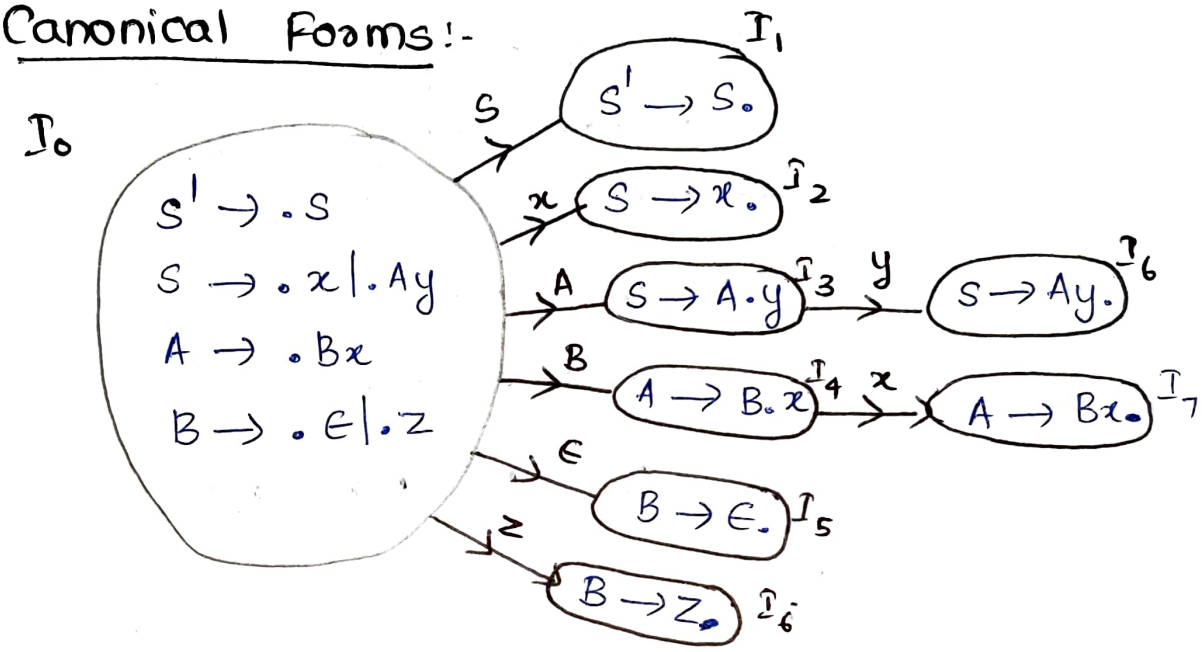
$S' \rightarrow S$

$S \rightarrow x \mid Ay$

$B \rightarrow \epsilon \mid x$

$A \rightarrow Bx$

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

$S \rightarrow (L) | a$

$L \rightarrow L, S | S$

f) what are the terminal, non-terminal and start Symbol

g) Find parse tree for the following sentences.

(IV) (a,a)

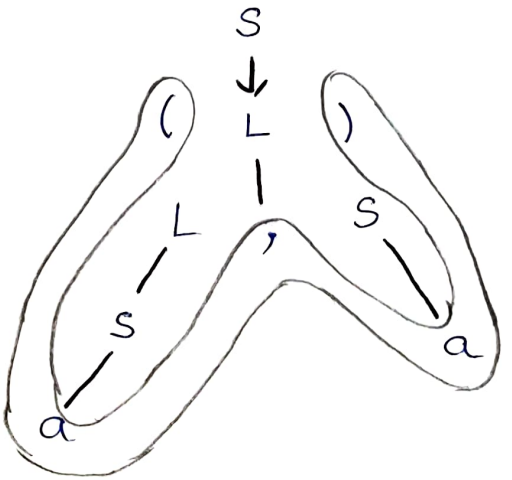
(V) (a, (a, a))

(VI) (a, ((a,a), (a,a)))

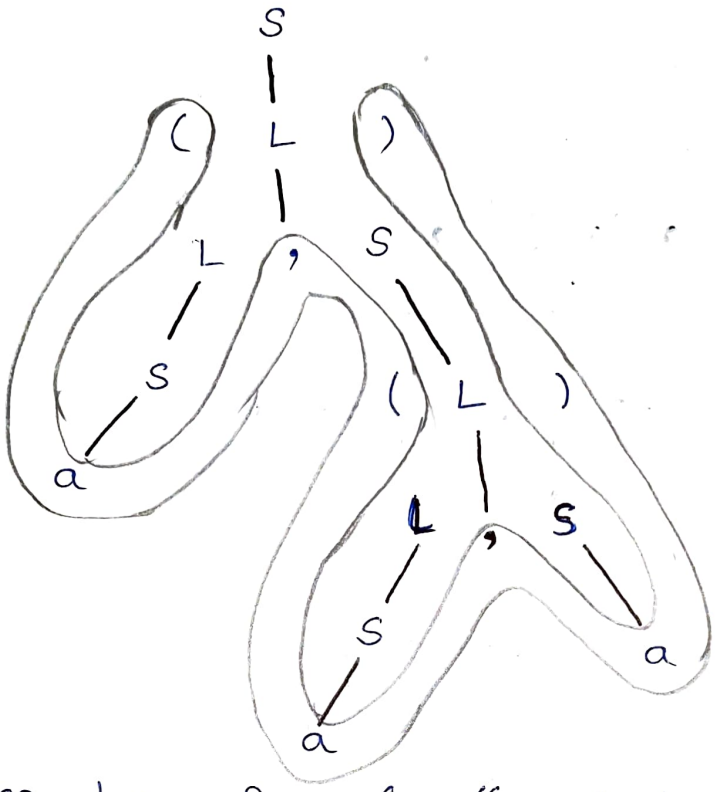
$S \rightarrow (L) | a$

$L \rightarrow L, S | S$

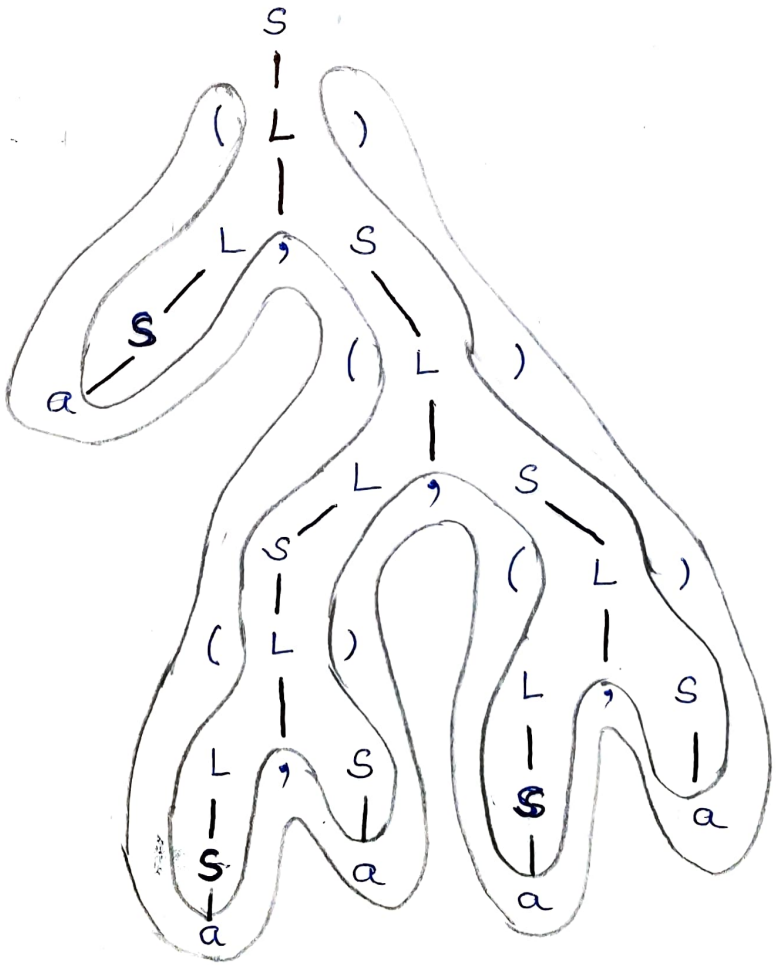
* terminals = { (,), , , a }
* Non-terminals = { S, L }
* Start Symbol = { S }
* Parse tree for (a, a)



Parse tree for (a, (a,a))



Parse tree for (a,((a,a),(a,a)))



1. What are the issues of the Lexical analyser?
2. Eliminate left recursion, perform left factoring and find FIRST & FOLLOW

$E \rightarrow E+T \mid T$

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

$X \rightarrow E, E \mid E$

3. Check the given grammar is LL(1) or NOT

$S \rightarrow (A) \mid o$

$A \rightarrow SB$

$B \rightarrow , SB \mid \epsilon$ and also parse the grammar (o, (o,o))

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 and white spaces has been removed.

* Improved Compiler Efficiency

Reading source code and classifying it in token is time consuming task. when we seperate

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 is 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 cannot tell whether "fi" is mis-spelling of "if" statement or a undeclared literal.

Probably the passer 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 irrelevant 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 characte of a large source program. Lexical analyzer may need to look atleast 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

$E \rightarrow TE'$

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

* Second production has left factoring

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

Removing left factoring

$T \rightarrow id\,T'$

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

Now the productions are

$E \rightarrow TE'$ —— ①

$E' \rightarrow \epsilon \mid TE'$ —— ②

$T \rightarrow id\,T'$ —— ③

$T' \rightarrow \epsilon \mid [] \mid [x]$ —— ④

$X \rightarrow E, E \mid E$ —— ⑤

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

FIRST(E') = {ϵ, id} FOLLOW(E') = {$, , ,]}

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

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

FIRST (x) = {id} FOLLOW(x) = {]}

40. 3. S → (A) | o

A → SB

B → , SB | ∈

No left recursion nor left factoring

FIRST (S) = {(, o} FOLLOW (S) = {, ,), o, $}

FIRST (A) = {(, o} FOLLOW (A) = {)}

FIRST (B) = {,} FOLLOW (B) = {)}

	S	A	B
(	S→ (A)	A→SB	
)			B→∈
,			B→ ,SB
o	S→o	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 → ∈

Now stack is empty with $, so accepted.

23 1. Define handle and handle pruning.

2. Construct LR parsing table.

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

id * id + id using stack implementation.

23)

1. **Handles:-**

* Formally, Handle of a right sentential form γ is $\langle A \to \beta$ location of β in $\gamma \rangle$
* ie., $A \to B$ 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-ambiguous grammar have one unique handle.

Handle Pruning:-

* The process of discovering a handle and 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. $E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$P \to (E)$

$F \to id$

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 $	γ_6
0 F 3	* id + id $	γ_4
0 T 2	* id + id $	S7
0 T 2 * 7	id + id $	S5
0 T 2 * 7 id 5	+ id $	γ_6
0 T 2 * 7 F 10	+ id $	γ_3
0 T 2	+ id $	γ_2
0 E 1	+ id $	S6
0 E 1 + 6	id $	S5
0 E 1 + 6 id 5	$	γ_6
0 E 1 + 6 F 3	$	γ_4
0 E 1 + 6 T 9	$	γ_1
0 E 1	$	accept

24 1) Differentiate between final states in a NFA and a DFA

2) Table :-

Remove left recursion	Remove left Factoring
$A \rightarrow A\alpha \mid \beta$	$S \rightarrow iEtS \mid iEtSeS \mid a$ $E \rightarrow b$
$S \rightarrow Aa \mid b$ $A \rightarrow Ac \mid Sd \mid \epsilon$	Stmt → if expr then stmt else stmt | if expr then stmt
$S \rightarrow aBDh$ $S \rightarrow Bb \mid C$ $D \rightarrow EF$ $E \rightarrow g \mid \epsilon$ $F \rightarrow f \mid \epsilon$	$S \rightarrow aSb \mid aTc$ $T \rightarrow dTU \mid \epsilon$ $U \rightarrow f$
$S \rightarrow SA \mid SB \mid a \mid b \mid c$	

24 1) **Final State of DFA**

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

Q - set of states

Σ - alphabets

δ - transition function

q_0 - Initial state

F - 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 - set of states

Σ - alphabets

δ - transition function

q_0 - initial state

F - Final state.

Final state ⇒ A non-empty set of final states and member of Q.

24

2. REMOVE LEFT RECURSION

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

$A \rightarrow \beta A'$

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

2) $S \rightarrow Aa \mid b \Rightarrow S \rightarrow Sda \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon \Rightarrow A \rightarrow Ac \mid Aad \mid \epsilon \mid bd$

$S \rightarrow Sda \mid b \Rightarrow$ $S \rightarrow bS'$, $S' \rightarrow daS' \mid \epsilon$

$A \rightarrow Ac \mid Aad \mid \epsilon \mid bd \Rightarrow A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

$\Rightarrow$ $A \rightarrow bdA' \mid \epsilon A'$, $A' \rightarrow cA' \mid adA' \mid \epsilon$

$\rightarrow$ ⇒ $S \rightarrow bS'$, $S' \rightarrow daS' \mid \epsilon$, $A \rightarrow bdA' \mid \epsilon A'$, $A' \rightarrow cA' \mid adA' \mid \epsilon$

(III) $S \rightarrow aBDh$
$S \rightarrow Bb|c$
$D \rightarrow EF$
$E \rightarrow g|\epsilon$
$F \rightarrow f|\epsilon$

No Left Recursion in this example.

(IV) $S \rightarrow SA|SB|a|b|c$
$S \rightarrow aS'|bS'|c$
$S' \rightarrow AS'|BS'|\epsilon$

REMOVE LEFT FACTORING

(I) $S \rightarrow iEts|iEtseS|a$
$E \rightarrow b$
$S \rightarrow iEtSS'|a$
$S' \rightarrow \epsilon|eS$
$E \rightarrow b$

(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) $S \rightarrow aSb|aTc$
$T \rightarrow dTU|\epsilon$
$U \rightarrow f$
$S \rightarrow aS'$
$S' \rightarrow \epsilon|Sb|Tc$
$T \rightarrow dTU|\epsilon$
$U \rightarrow f$

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 grammar is LL(1) or NOT

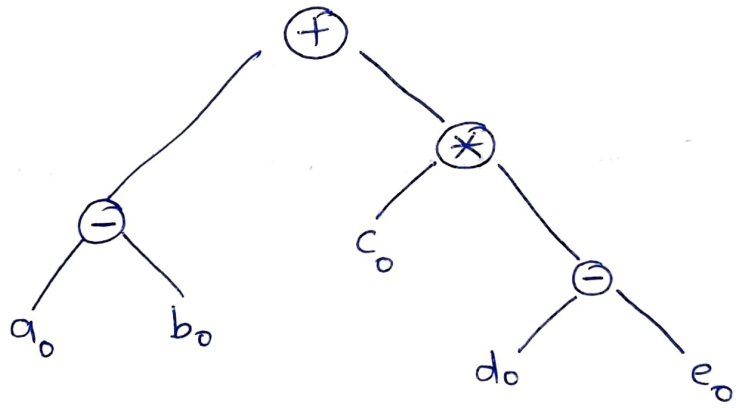
$S \rightarrow (A) \mid o$

$A \rightarrow SB$

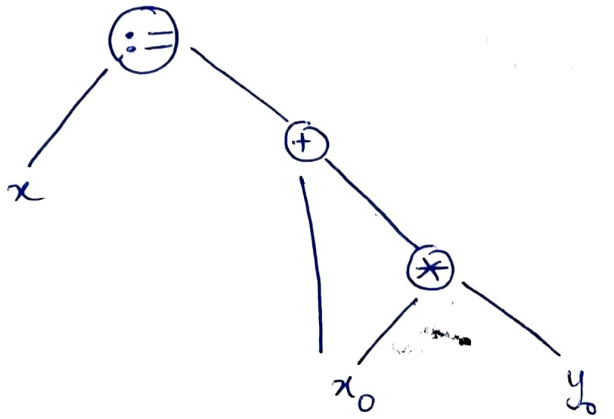
$B \rightarrow , SB \mid \epsilon$

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

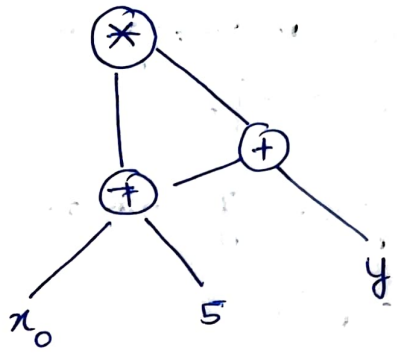
3 1) (a)(a-b) + c * (d/e)



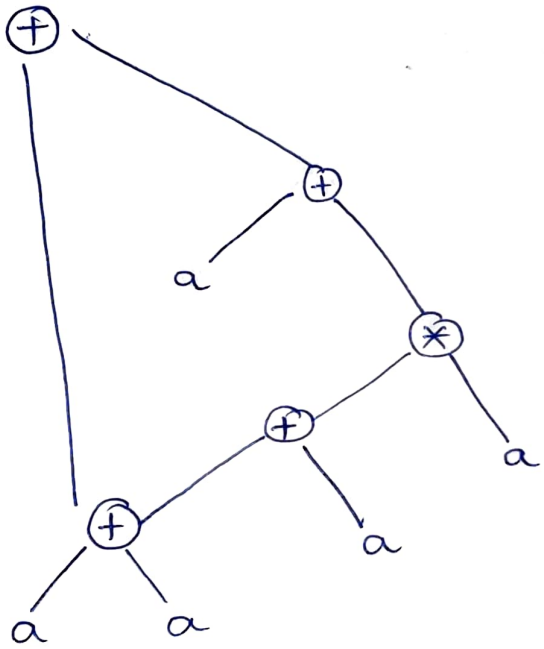
b) $x = x + x * y$



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



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



23)

2. $E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

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
0 B) B) $	0))	S → 0
) B) $	))	B → ∈
) $	)	B → ∈

Now stack is empty with $, so accepted.

1. 1. Check the given Grammar is ambiguous | unambiguous

$S \rightarrow S(S)S \mid \epsilon$

2. Prove the given grammar is LR(1), LALR(1) NOT SLR(1)

$S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d$

1) (i) $S \rightarrow S(S)S \mid \epsilon$

Let us consider the string $\Rightarrow$ ( )

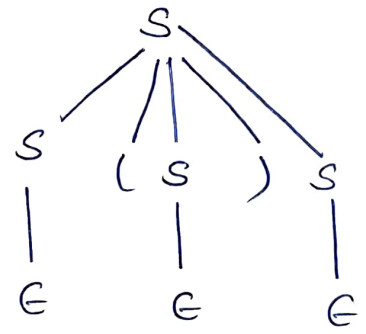
LMD

$S \rightarrow S(S)S \rightarrow \epsilon(S)S \rightarrow (S)S$

$\rightarrow (\epsilon)S \rightarrow ()S$

$\rightarrow ()\epsilon \rightarrow ()$

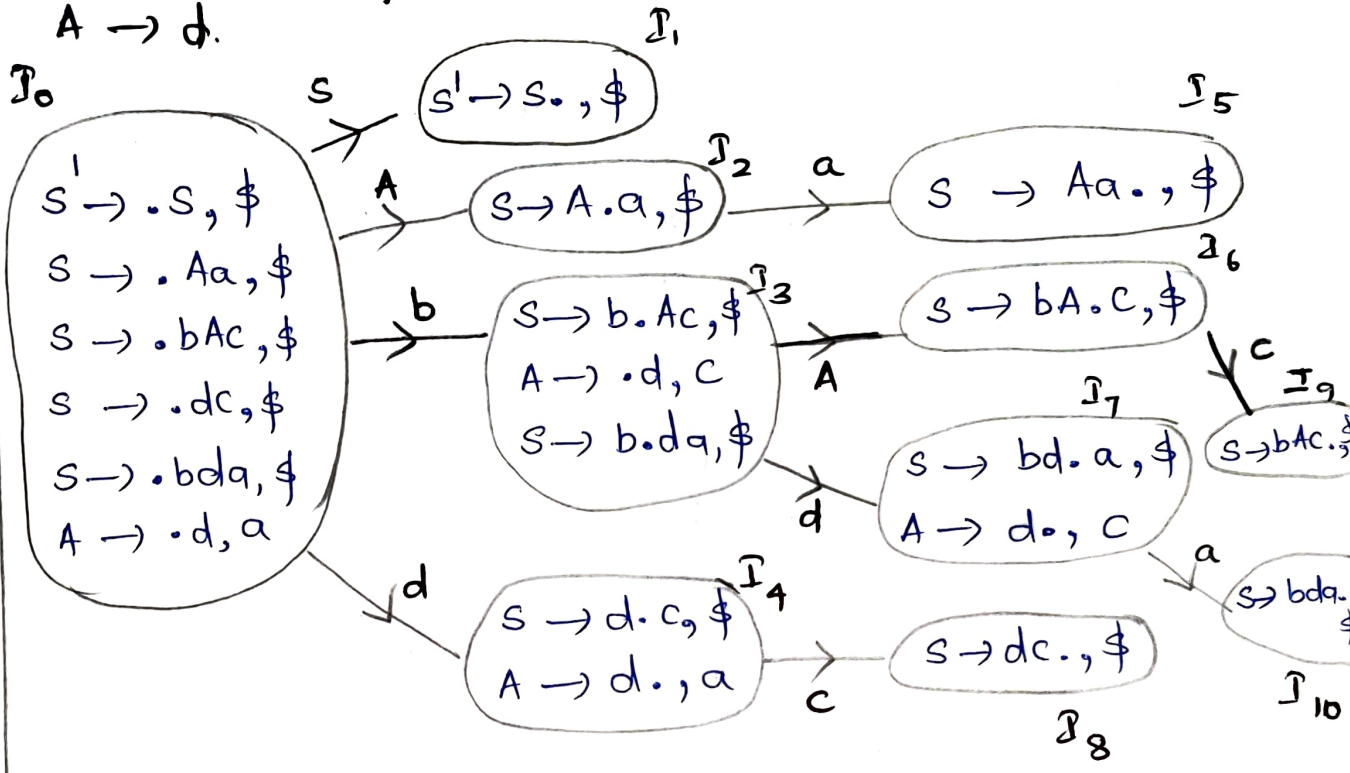
Parse Tree



We are able to derive atmost one left most derivation for the string ( ). Also only one way of Parse tree is available. Hence the given grammar is unambiguous.

(11) $S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d.$



State	Action					Goto	
	a	b	c	d	$	S	A
0		S_3		S_4		1	2
1					accept		
2	S_5						
3				S_7			6
4			S_8				
5							
6			S_9				
7	S_{10}		r_5				
8				r_3			
9				r_2			
10				r_4			

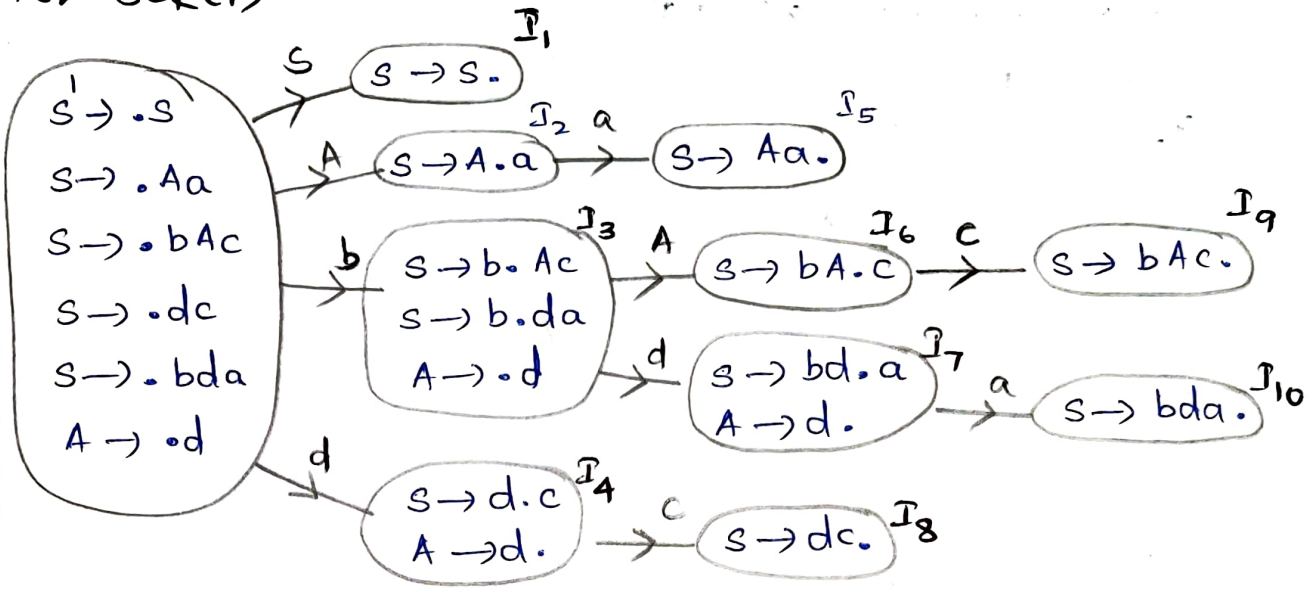
LALR(1) Table

There is no SR conflict and RR conflict

∴ The grammar is LALR(1) Parsable

If it is LALR(1) then for sure, it is LR(1)

For SLR(1)



Consider 7th state, symbol 'a', follow(A) = {a, c}, it moves and makes shift reduce conflict, so it is not SLR(1) parser.

2. 1. Find the following grammar is LL(1), LR(1)

$S \rightarrow AaAb \mid BbBa$

2. Check whether the following grammar is LR(0), SLR(1), LALR and LR(1)

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

2. FIRST (S) = {a, b} FOLLOW (S) = {$}

FIRST (A) = {ε} FOLLOW (A) = {a, b}

FIRST (B) = {ε} FOLLOW (B) = {b, a}

Parsing table:-

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

No Repitions
∴ LL(1)

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

1. Augmented Grammar

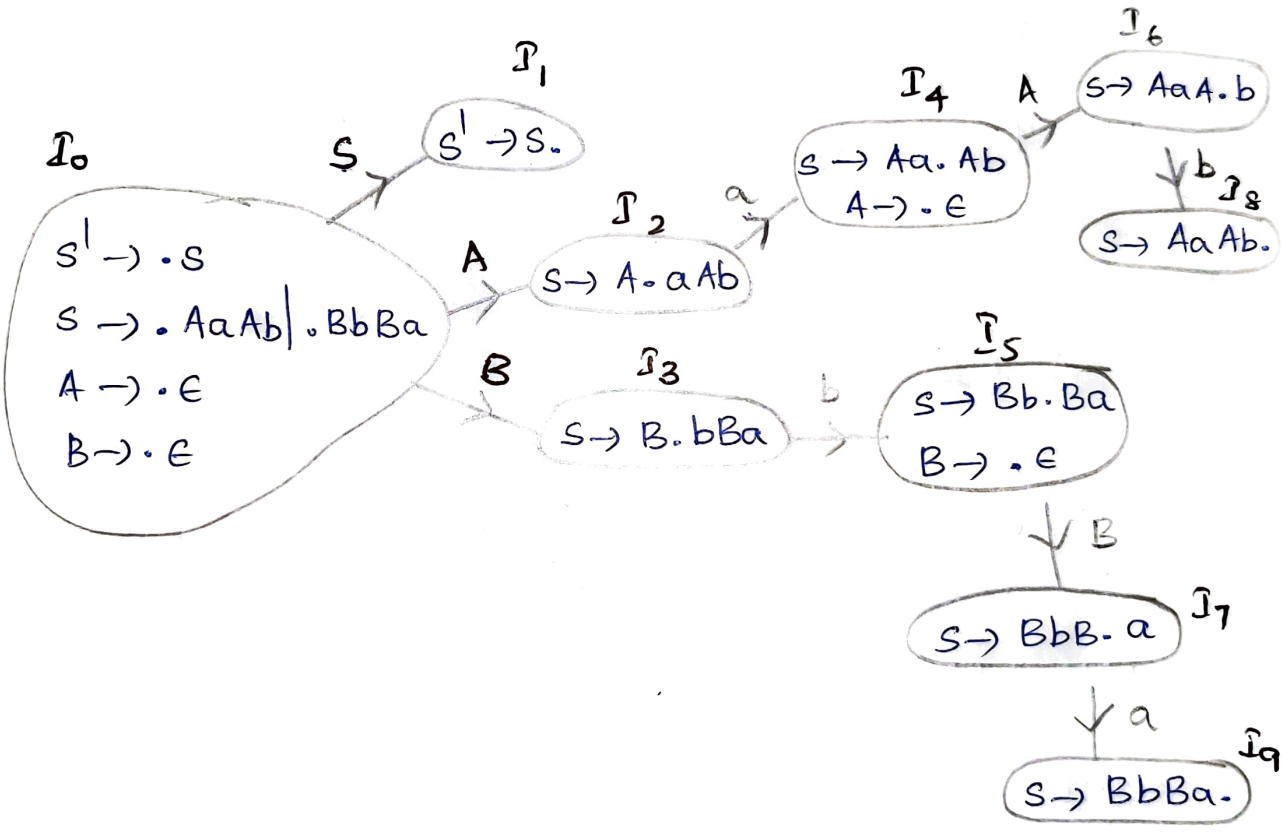
$S' \rightarrow S$

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

2. Canonical Forms:-



3) Parsing Table :-

	ACTION			GOTO		
	a	b	$	A	B	S
0				2	3	1
1			accept			
2	S_4					
3		S_5				
4				6		
5					7	
6		S_8				
7	S_9					
8	r_1	r_1	r_1			
9	r_2	r_2	r_2			

* There is no SR and RR conflict

Therefore it is LR(0)

* If it is LR(0) definitely it is SLR(1)