1. Construct CLR parsing table from

   $S \rightarrow AA$

   $S \rightarrow Aa/b$

2. Explain Code motion, copy propagation, dead code elimination.

Augmented Grammar

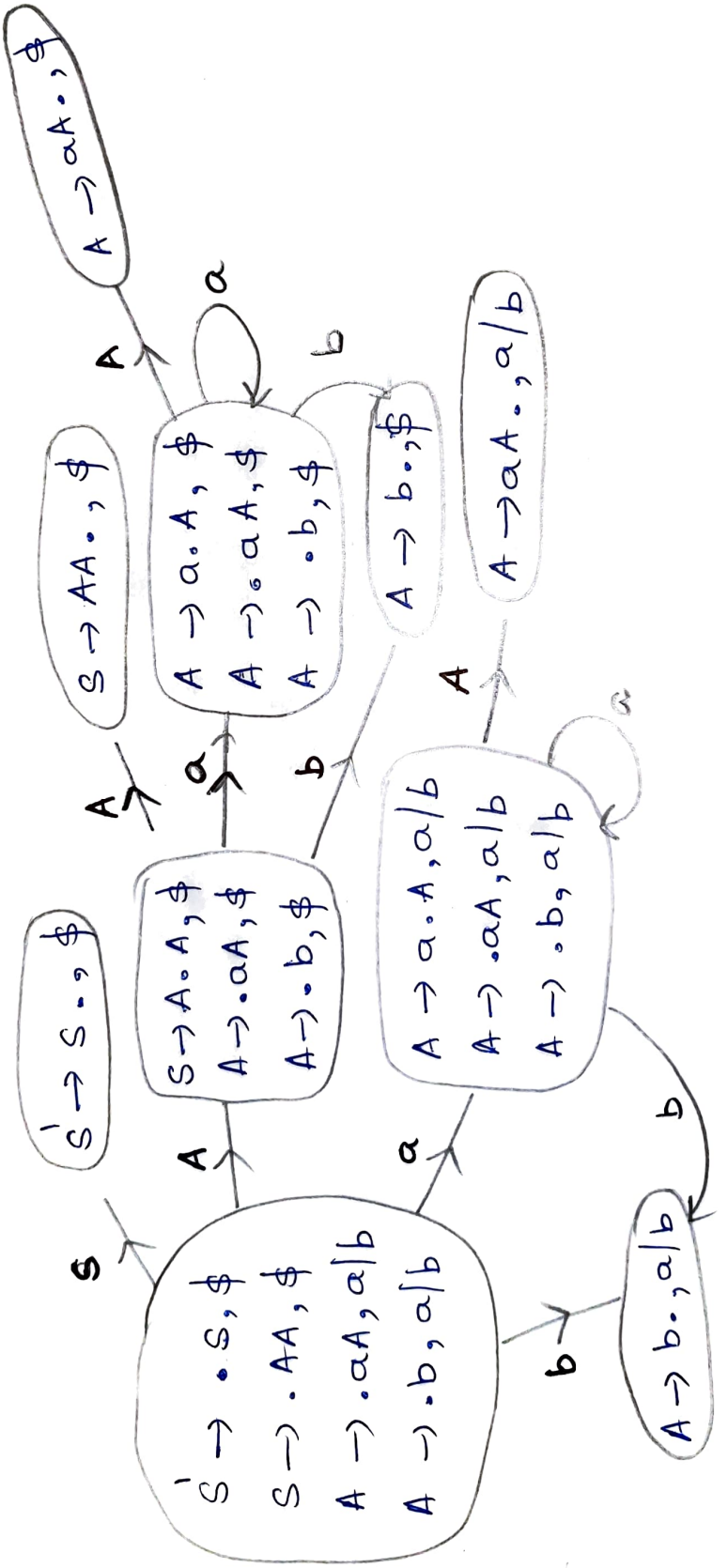
$S \rightarrow AA$

$A \rightarrow aA \mid b$

$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA \mid b$



CLR

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

LALR

	a	b	$	S	A
I_0	S_{36}	S_{47}			2
I_1			accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3			
I_5		R_1			
I_6					
I_{89}	R_2	R_2	R_2		

Code Motion:-

* It reduces the evaluation frequency of expression
* It brings loop invariant statements out of the loop.

eg.

```
a = 200
while (a>0)
{
  b = x+y;
  if (a%b==0)
     print (a)
}
```

↓

```
a = 200
b = x+y
while (a>0)
{
  if (a%b==0)
  } print (a)
```

Here in this e.g the expression $b = x+y$ is the loop invariant. 'b' is going to gain the same value throughout the loop. So we place it out of loop.

Now the expression $b=x+y$ is evaluated only once before entering the loop.
The frequency of evaluation of that expression is optimized.

* It is used to decrease the amount of code in loop.
* This transformation takes a statement of expression which can be moved outside the loop body without affecting the semantics of the program.

Variable Propagation:-

Before optimization

```
c = a * b
x = a
till
d = x * b + 4
```

After Optimization

```
c = a * b
x = a
till
d = a * b + 4
```

* Instead of taking reference from the variable x, we directly use the variable 'a'.
* Instead of substitution, direct value is taken.
* Here 'x' and 'a' are identified as common sub-expressions.

Dead code Elimination:-

Before elimination

```
c = a * b
x = b
till
d = a * b + 4
```

After elimination

```
c = a * b
till
d = a * b + 4
```

* The useless expressions are removed to reduce the number of lines of code and space.
* Prevents compiler on evaluating the expressions which are no use to the process or to the final result
* Before elimination, the expression x=b is called Dead state.

9. 1) Explain the various issues in the design of code generation.

2) Explain code generation phase with simple code generation algorithm.

9(1) **Issues in Code Generation:-**

* Input to code generators
* Target Program
* Memory management
* Instruction selection
* Register Allocation
* Evaluation order.
* Approaches to code generation.

Input to code Generators:-

It is the immediate code generated by the front end, along with information in the symbol table that determines the run times address of data object.

Target Program:-

It is the output of the code generation. Absolute machine language as output that it can be fixed in memory location and can be immediately executed. Reallocation to the machining language as an output allows subprogram and subroutine to the

compiled seperately. Assembly language as output makes the code generation easier.

Memory Management :-

Mapping the names in the source program to the address of data objects is done by the front end and the code generator.

Instruction Selection :-

Selecting the best instructor will improve the efficiency of program. It includes the instructio that should be complete and uniform and its play major role when efficiency is considered.

Register Allocation :-

Use of register make the computer faster in comparison to that memory. During register allocation, we select only these set of variables that will reside in the register. During a subsequent register assignment, the specific register is picked to variables.

Evaluation Order :-

The code generator decide the order in which the instruction will be executed. The order of computation affects the efficiency.

Approaches to code generation:-

It always generates the code of correct. It is essential because of the number of special case that a generator might face.

2. Code Generation:-

* The following actions are $x = y$ OP z
* Invoke a function get reg to determine the location L, where the result of the computation y OP z should be stored.
* Construct the address destructer for y to determine y', the current location of y prefer the register for y' if the value of y is currently both in memory and register.
* Generate the instruction OP z' where z' is a current location of z. After execution of $x = y$ OP z, it will no longer contain y or z.

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)

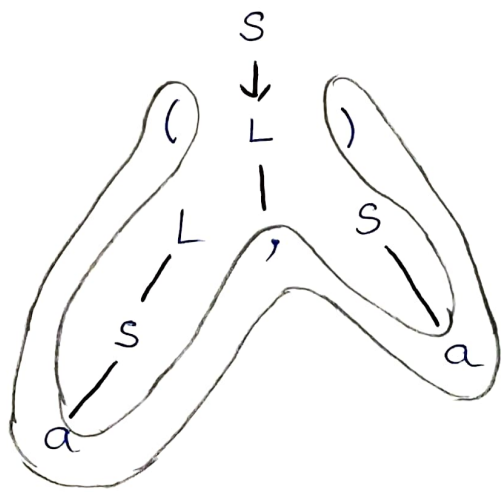
(V) (a, (a,a))

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

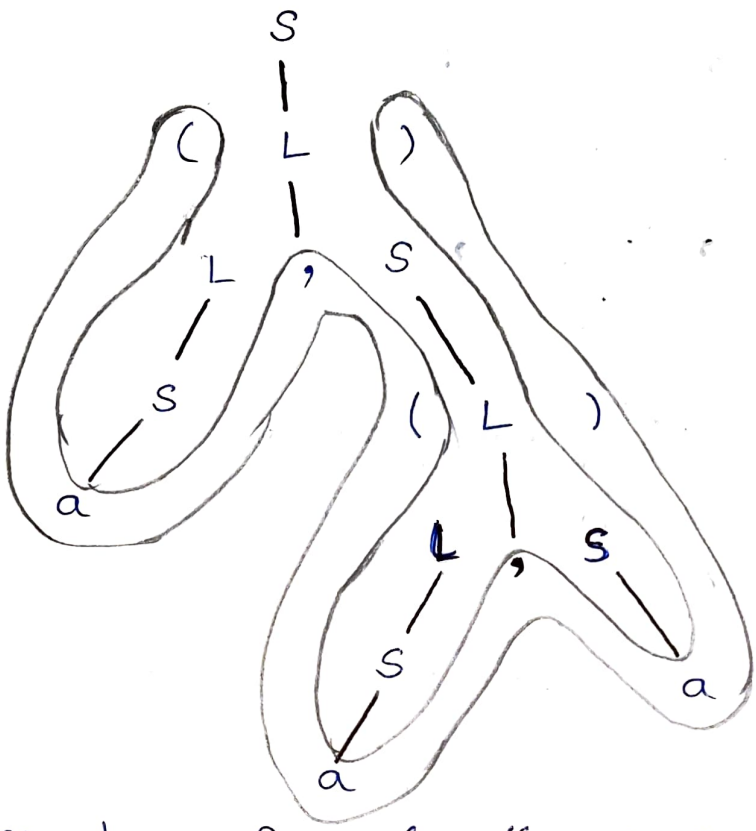
$S \rightarrow (L) | a$

$L \rightarrow L, S | S$

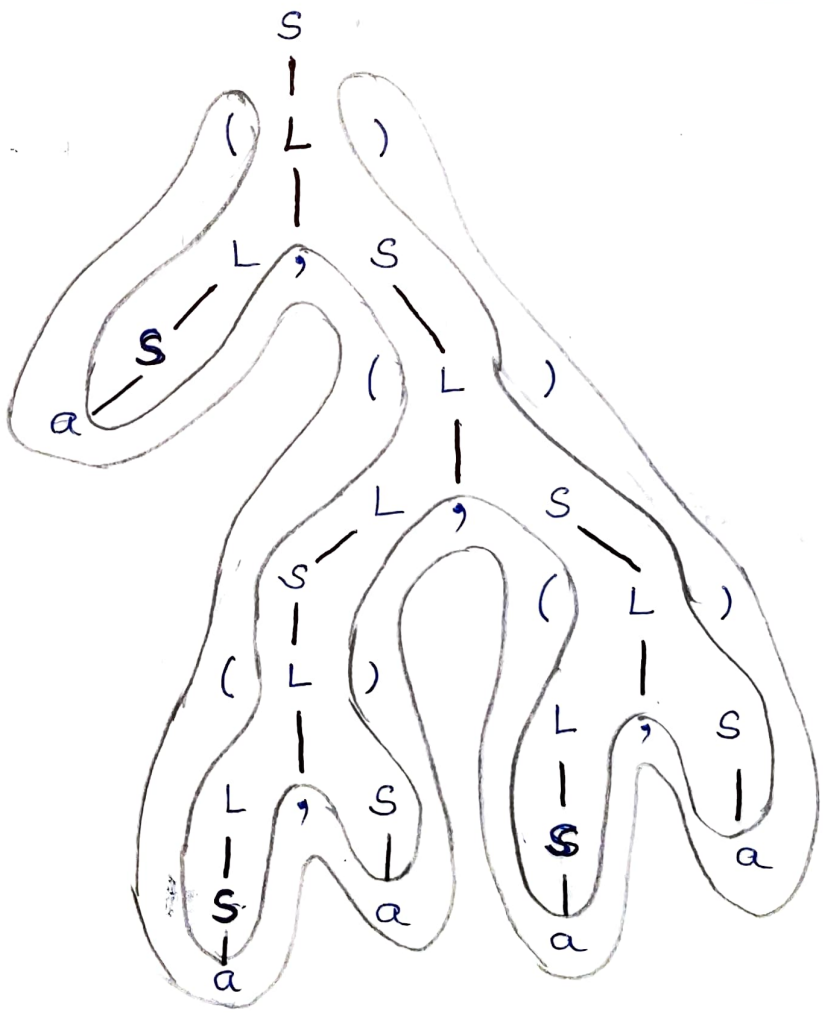
* terminals = { (,), , , a }
* Non-terminals = { S, L }
* Start Symbol = { S }
* Parse tree for (a,a)



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



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



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

4)

1. Explain& example following code optimization
   a) Common sub expression elimination
   b) Copy propagation
   c) Dead code elimination
   d) Code motion
2. Eliminate left recursion, perform left factoring and find: FIRST & FOLLOW

   $E \rightarrow E+T \mid T$

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

   $X \rightarrow E,E \mid E$

Variable Propagation:-

Before optimization

```
c = a * b
x = a
till
d = x * b + 4
```

After Optimization

```
c = a * b
x = a
till
d = a * b + 4
```

* Instead of taking reference from the variable x, we directly use the variable 'a'.
* Instead of substitution, direct value is taken.
* Here 'x' and 'a' are identified as common sub-expressions.

Dead code Elimination:-

Before elimination

```
c = a * b
x = b
till
d = a * b + 4
```

After elimination

```
c = a * b
till
d = a * b + 4
```

* The useless expressions are removed to reduce number of lines of code and space.
* Prevents compiler on evaluating the expressions which are no use to the process or to the final
* Before elimination, the expression x=b is called Dead state.

Code Motion:-

* It reduces the evaluation frequency of expression
* It brings loop invariant statements out of the loop.

eg.

```
a = 200
while (a>0)
{
  b = x+y ;
  if (a%b==0)
     print (a)
}
```

```
a = 200
b = x+y
while (a>0)
{
  if (a%b==0)
  { print (a)
```

Here in this e.g the expression $b = x+y$ is the loop invariant. 'b' is going to gain the same value throughout the loop. So we place it out of loop.

Now the expression $b=x+y$ is evaluated only once before entering the loop.
The frequency of evaluation of that expression is

* It is used to decrease the amount of code in loop.
* This transformation takes a statement of expression which can be moved outside the loop body without affecting the semantics of the program.

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) $S \rightarrow AaAb \mid BbBa$ $A \rightarrow \varepsilon$ $B \rightarrow \varepsilon$

$S \rightarrow AaAb \mid BbBa$ (e)

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

1. Augmented Grammar

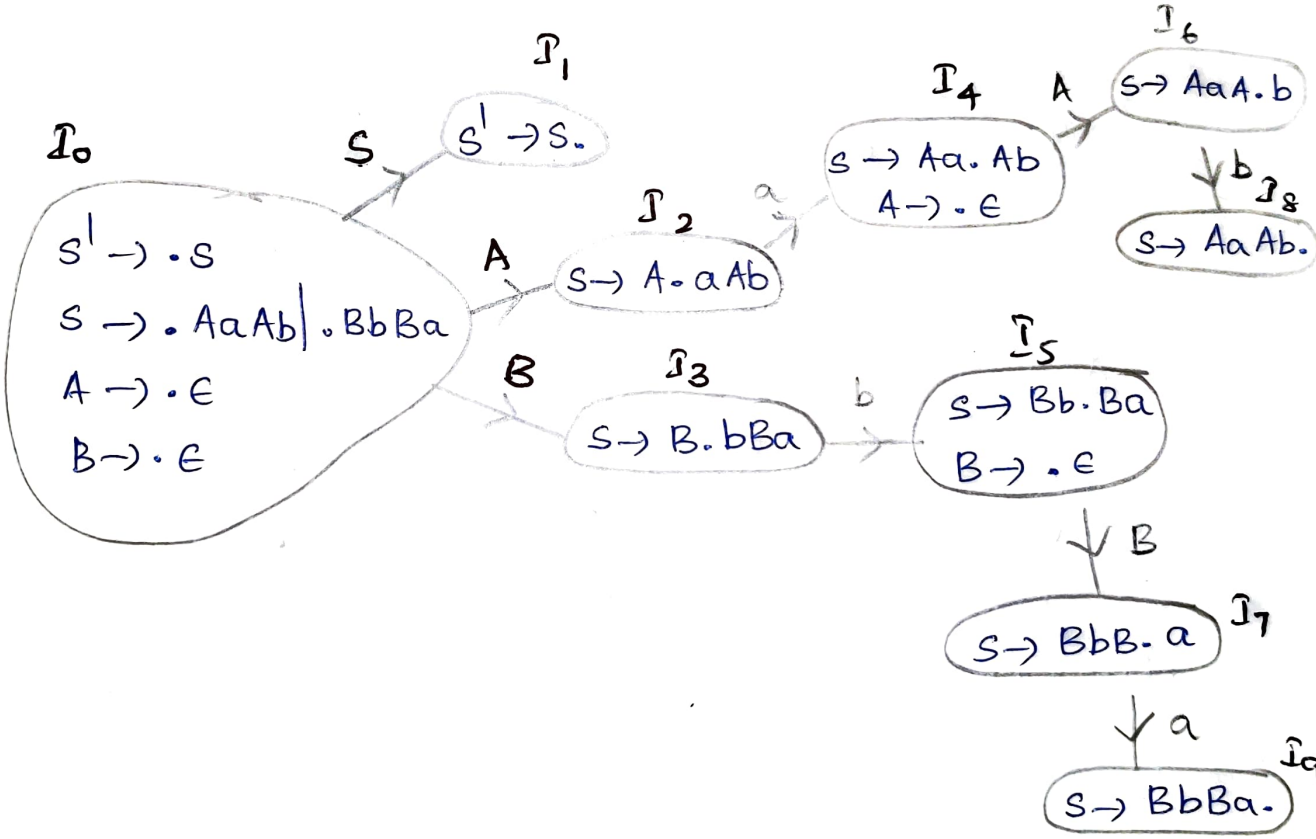
$S' \rightarrow S$

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

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)

2. $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

FIRST(S) = {a, b}
FIRST(A) = {ϵ}
FIRST(B) = {ϵ}

FOLLOW(S) = {$}
FOLLOW(A) = {a, b}
FOLLOW(B) = {b, a}

Parsing Table:-

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

No Repetitions
∴ LL(1)

11. 1. Discuss the role of finite automata in f Compiler

2. State what strategy LEX should adopt if keywords are not reserved words.

(1) Role of Finite Automata :-

Finite automata is a state machine that take a string as a symbol as input and changes its state accordingly and it is a recognizer for regular expression when a regular expression string is fed into finite automata.

The machine consists

Q - set of states

Σ - input symbols

q - starting state

f - final state

σ - transition function.

2)

RE	Matches	Example.
c	Single character not operator	x
\c	Any character following	*
"s"	String s literally	"***"
$	End of line	abc$
[s]	any character in s	[abs]
^	Begining of line	^abc.
[^s]	Any except character not s	[^abc]
r^*	Zero	a^*
$r+$	one	a^+
$r?$	Zero	a?
$r\{m,n\}$	m to n occurance of r	a[1,5]
r_1, r_2	r_1 then r_2	ab
(r)	r	(a.b)
$r_1 \mid r_2$	r_1 when followed by r_2	Abc|123
r_1 / r_2	r_1 or r_2	a:b.