PRINCIPLES OF COMPILER DESIGN

ASSESSMENT-1

SRUTHI S(19MID0053)

9)	1) Explain the various issues in the design of code generation. 2) Explain code generation phase with simple code generation algorithm 3) Find Recursive desent parser E→iE' E'→+iE' | ε
11)	1) Discuss the role of finite automata in compiler. 2) State what strategy LEX should adopt if keywords are not reserved words. 3) Write short note on input buffer with lexical analyser 4) Find FIRST and FOLLOW sets for the given grammer S→PQR P→a |Rb|ε Q→c|dp|ε R→e|f
15)	1) Construct SLR parsing table for the grammar S→AB|gDa A→ab|c B→dC C→gC|g D→fD|g 2) Discuss the role of finite automata in compiler. 3) State what strategy LEX should adopt if keywords are not reserved words.
37)	1) Define Activation tree 2) Explain about input buffering technique 3) Construct the NFA from the (a/b)*a(a/b) using Thompson's construction algorithm. 4) Find FIRST and FOLLOW sets for the given grammar S→PQR P→a |Rb|ε Q→c|dp|ε R→e|f

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→E+T|T T→id|id [ ] |id [X] X→E,E|E

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.

18)	1) Give the criteria for achieving machine dependent optimization 2) Construct a canonical parsing table S→CC C→cC|d

17)	1) Construct CLR parsing table from S→AA A→Aa|b 2) Explain code motion, copy propagation, dead code elimination.

PRINCIPLES OF COMPILER DESIGN

SRUTHI.S
19MID0053

DIGITAL ASSESSMENT - 1

(9.) → (i) Explain the various issues in the design of code generation.

The various issues are

(i) Input to code generators → It is the intermediate code generated by the front end, along with information in the symbol table that determines the run times address of the data object.

(ii) Target program → It is the output of the code generator. Absolute machine language as output that it can be fixed in memory location and can be immediately executed. Reallocation machine language as an output allows sub program & subroutine to the compiled separately. Assembly language as output makes the code generation easier.

(iii) Memory management → Mapping the names in the source program to the address of data object is done by the front end and the code generator.

(iv) Instruction selection → selecting the best instructor will improve the efficiency of program. It includes the instruction that should be complete & uniform & its play major role when efficiency is considered.

(v) Register allocation → use of register make the computer faster in the 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.

(vi) Evaluation order → The code generator decide the order in which the instruction will be executed. The order of computation affects the efficiency of the code.

(viii) Approaches to code generation → It always generate the code of correct. It is essential because of the number of special case that a code generator might fo

SRUTHI.S
19MID0053

(9.) → 2.b Explain code generation phase with simple code generation algorithm.

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

→ Consult the address descriptor 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 & register.

→ Generate the instruction op z', L where z' is a currently location of z. prefer a register to a memory location if z is in both.

→ If the current values of y or z have no next uses, are not live on exit from the block, and are in registers. after execution of X := y op z, it will no longer contain y or z

SRUTHI.S
19MID0053

(9) → B.Y Find recursive descent parser

E → iE'
E' → +iE' | ε

```
E()
{
    l = getchar()
    if (l == 'i')
    {
        match('i');
        E'();
    }
}
E'()
{
    l = getchar();
    if (l == '+')
    {
        match('+');
        match('i');
        E'();
    }
    else
    return;
}
match(char t)
{
    if (l == t)
    l = getchar(),
    else
    Printf("error")
}
main()
{
    E();
    if (l == '$')
    Printf("parsing success");
}
```

11. → i) Discuss the role of finite automata in compiler

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

The machine consists

Q → set of state Σ → input symbol q → starting state

F → final state σ → transition function

State → represented by circles. Start state → The state where the automata start. Final state → The state where the automata ends.

Transition → It happen from 1 state to another when a desire symbol is found.

SRUTHI.S
19MID0053

(1)→2) State what strategy LEX should adopt if keywords are not reserved words

Lex pattern are standard unix regular expression using symbol.

Standard regular expression	Matches	Example
c	single character not operator	x
\c	Any character following	*
"s"	string s literally	"**"
.	Any single character except (n)	a.*b.
^	Beginning of line	^abc
$	End of line	abc$
[s]	Any character in s	[abs]
[^s]	Any except character not s	[^abc]
r*	zero	a*
r+	one	a+
r?	zero	a?
r{m,n}	m to n occurence of r	a{1,5}
$r_1 r_2$	r_1 then r_2	ab
(r)	r	(a:b)
r_1/r_2	r_1 when followed by r_2	Abc/123
$r_1 \mid r_2$	r_1 or r_2	a:b

(2)→3) Write a short note on input buffer with LA.

→ The lexical analyzers scans the input from left to right one character at a time. It uses two pointers begins ptr (bp) & forward to keep track of the pointer of the

SRUTHI.S
19MID0053

input scanned. The forward ptr moves ahead to search for end of lexeme. As soon as blank space is encountered, it indicates end of lexeme.

(4.) ⇒ 4.Y Find FIRST & FOLLOW:

$S \to PQR$
$P \to a \mid Rb \mid \varepsilon$
$Q \to c \mid dp \mid \varepsilon$
$R \to e \mid f$.

State	FIRST	FOLLOW
S	{a, e, c, d, f}	{$}
P	{a, e, f, ε}	{c, d, e, f}
Q	{c, d, ε}	{e, f}
R	{e, f}	{b, $}

(5.) ⇒ 1.Y construct SLR parsing table for the grammer.

$S \to AB \mid gDa$
$A \to ab \mid C$
$B \to dc$
$C \to gc \mid g$
$D \to fD \mid g$

Augmented Grammer:

$S' \to .S$
$S \to AB$
$S \to g.Da$
$A \to ab$
$A \to c$
$B \to dc$
$C \to gc$
$C \to g$
$D \to fD$
$D \to g$

I0

$S' \to .S$
$S \to .AB$
$S \to .gDa$
$A \to .ab$
$A \to .c$
$B \to .dc$
$C \to .gc$
$C \to .g$
$D \to .fD$
$D \to .g$.

SRUTHI.S
19MIDC053

goto (I0, S)
S1 → S. (I1)

goto (I0, A)
S → A.B (I2)
B → .dC

goto (I0, g)
S → g.Da (I3)
D → .fD
C → g.C
C → .g ⇒ g.
D → g.

goto (I0, a)
A → a.b (I4)

goto (I0, C)
A → C. (I5)

goto (I0, d)
B → d.C (I6)
C → .gC | .g

goto (I0, f)
D → f.D (I7)
D → .g

goto (I2, B)
S → AB. (I8)

goto (I2, d)
B → d.C (I6)
C → .gC | .g

goto (I3, D)
S → gD.a (I8)

goto (I3, f)
D → f.D (I7)
D → .g

goto (I3, C)
C → gC. (I9)

goto (I4, b)
A → ab. (I10)

goto (I6, C)
B → dC. (I11)

goto (I6, g)
C → g.C (I12)
C → .g

goto (I7, D)
D → fD. (I13)

goto (I7, g)
D → g. (I14)

goto (I8, a)
S → gDa. (I15)

goto (I12, C)
C → gC. (I9)

goto (I12, g)
C → g. (I10)

FOLLOW {A} = {d}
FOLLOW {S} = {$}
FOLLOW {C} = {$}
FOLLOW {B} = {$}
FOLLOW {D} = {a}

Parsing Table

SRUTHI.S
19MID0053

	ACTION							GOTO				
state ↓	g	a	b	c	f	d	$	S	A	B	C	D
0	S_3	S_4		S_5	S_7	S_6		1	2			
1							All					
2						S_6				8.		
3					S_7						9	8
4				S_{10}								
5						R_4						
6	S_{12}										11	
7	S_{14}											13
8		S_{15}					R_1					
9							R_6					
10						R_3						
11							R_5					
12	S_{13}										16	
13		R_8										
14		R_9										
15							R_2					
16							R_7					

11. → i) Discuss the role of finite automata in compiler

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

The machine consists

Q → set of state Σ → input symbol q_0 → starting state

F → final state δ → transition function

State → represented by circles. Start state → The state where the automata start. Final state → The state where the automata ends.

Transition → It happen from 1 state to another when a desire symbol is found.

SRUTHI.S
19MID0053

(36.) ⇒ State what strategy LEX should adopt if keywords are not reserved words.

Lex pattern are standard unix regular expression using symbol.

Standard regular expression	Matches	Example
c	single character not operator	x
\c	Any character following	*
"s"	string s literally	"x**"
.	Any single character except (n)	a.*b.
^	Beginning of line	^abc
$	End of line	abc$
[s]	Any character in s	[abs]
[^s]	Any except character not s	[^abc]
r*	zero	a*
r+	one	a+
r?	zero	a?
r{m,n}	m to n occurence of r	a{1,5}
r1 r2	r1 then r2	ab
(r)	r	(a:b)
r1/r2	r1 when followed by r2	Abc/123
r1|r2	r1 or r2	a:b

(37.) ⇒ 1.) Define activation tree:

→ Whenever a procedure is executed its activation record is stored on the stack, also known as control stack when a called procedure is executed, it returns the control back to the caller.

→ This type of control flow makes it easier to represent a series of activation in the form of a tree known as Activation tree.

SRUTHI.S
19MID0053

(37) ⇒ 2.5 Explain input buffering technique:

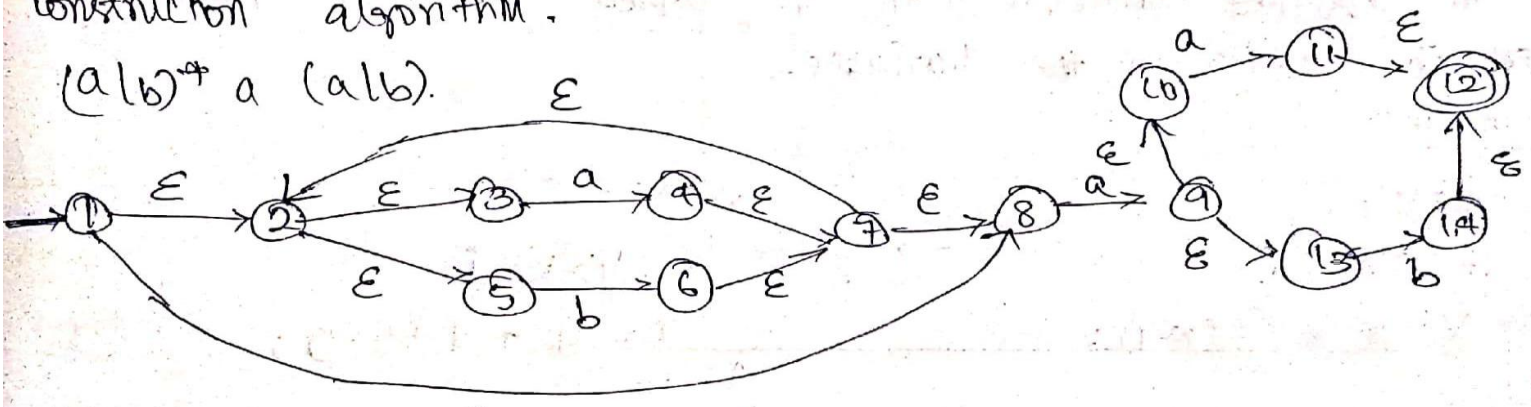
→ To ensure that a right lexeme is found, one (or) more character have to be looked up beyond the next lexeme. Hence a 2 buffer scheme is introduced to handle large lookheads safely.

Three approaches:

* By using a lexical generator in this generator provide routine for reading & buffering input.
* By writing the lexical analyser in a conventional system programming language using I/O facilities of that language to read the input.
* By assembly language & explicit managing reading of input.

(37) ⇒ 3* construct NFA from (a|b)*a(a|b) using thompson construction algorithm.

(a|b)* a (a|b).



(37) ⇒ 4* Find FIRST & FOLLOW:

S → PQR | Q → c | dp | ε
P → a | Rb | ε | R → elf.

States	FIRST	FOLLOW
S	{a, e, c, d, f}	{$}
P	{a, e, f, ε}	{c, d, e, f}
Q	{c, d, ε}	{e, f}
R	{e, f}	{b, $}

④ ⇒ (a) Explain & example following code optimization

a) common sub expression elimination :-

Idea → replace an expression with previously stored evaluation of that expression.

Example :-

$[a + i * 4] = [a + i * 4] + 7$.

common subexpression elimination removes the redundant add & multiply.

$t = a + i * a$; $[t] = [t] + 1$.

Always improves performance.

b) copy propagation :-

Idea ⇒ If one variable is assigned to another, replace uses of the assigned variable with the copied variable. Need to know where the copies of the variable.

Example :-

```
x=y;                        x=Y:
if (x>y) {          →       ie (y>1) {
  x= x + f(x-1);              x= y + f(y-1);
}                           }
```

can make the first assignment x dead code.

c) Dead code elimination :-

Idea → If a side free statement can never be observed, it is safe to eliminate the statement. A variable dead if it is never used after it is defined.

```
x = Y + Y  // x is dead ?   →   ...
...        // x never used       x = z * z
x = z * z
```

Dead variables are created by optimization.

SRUTHI.S
191910053

d.) code motion ::

idea → the code motion just means that the code is moved out of the loop as it went have any difference if it is performed inside the loop repeatedly. The compiler by taking the code that doesn't need to be in the loop & making it outside of it. for the optimization process.

(4) ⇒ 2.) Eliminating left recursion, left factoring, finding first & follows.

$E \to E+T \mid T$
$T \to id \mid id[\,] \mid id[X]$
$X \to \varepsilon, E \mid E$

Eliminating left recursion →

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T' \to [\,] \mid [X] \mid \varepsilon$
$X \to \varepsilon, E \mid E$

NONES	FIRST	FOLLOW
E	{id}	{$,)}
E'	{+, ε}	{+, $}
T	{id}	{+, $, ,}
T'	{[, ε}	{]}
X	{id}	{$}.

(3) ⇒ 1.) Define handle & handle pruning ?

handle' → Handle of a right sentential form γ is $\langle A \to \beta$ location of β in $\gamma \rangle$.

$A \to \beta$ is a handle of $\alpha\beta\gamma$ at the location immediately after the end of α.

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

$S = \gamma_0 \Rightarrow \gamma_1 = \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$.

(2) ⇒ 2.) construct LR parsing table ::

$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to (E)$
$F \to id$

id * id + id using Stack implementation.

SRUTHI S
19MIDOO53

First & Follow:

State	FIRST	FOLLOW
E	{ c, id }	{ +,), $ }
E'	{ +, ε }	
T	{ c, id }	{ +,), $, * }
T'	{ *, ε }	
F	{ c, id }	{ +,), $, * }

Parsing table:

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S_5						1	2	3
1		S_6				accept			
2		r_2			r_2	r_2			
3		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 implementation:

Stack	Inputs	Action
0	id* id +id$	shift
0 id 5	*id +id $	reduced by F → id.
0F3	* id +id $	reduced by T → F.
0T2	* id +id $	shift.
0T2 * 7	id +id $	shift.
0T2 * 7 id 5	+id $	reduced by F → id.

$0T_2 * T F 10$	$+id \$$	Reduced by $T \to T * F$
$0T_2$	$+id \$$	Reduced by $E \to T$
$0E1$	$+id \$$	shift
$0E1+6$	$id \$$	shift
$0E1+6id5$	$\$$	Reduced by $F \to id$
$0E1+6F3$	$\$$	Reduced by $T \to F$
$0E1+6T9$	$\$$	Reduced by $E \to E+T$
$0E1$	$\$$	Accept

(18) ⇒ 1.b Give the criteria for achieving machine dependent optimization

→ It involves transformation that take into consideration, the properties of the target machine (like register & special machine instruction sequence.

the measure are

* Allocation of adequate number of resources to enhance the execution efficiency of the program.
* Use of immediate instruction wherever required.
* The use of idiomic instruction.

(18) ⇒ 2.b construct canonical parsing table:

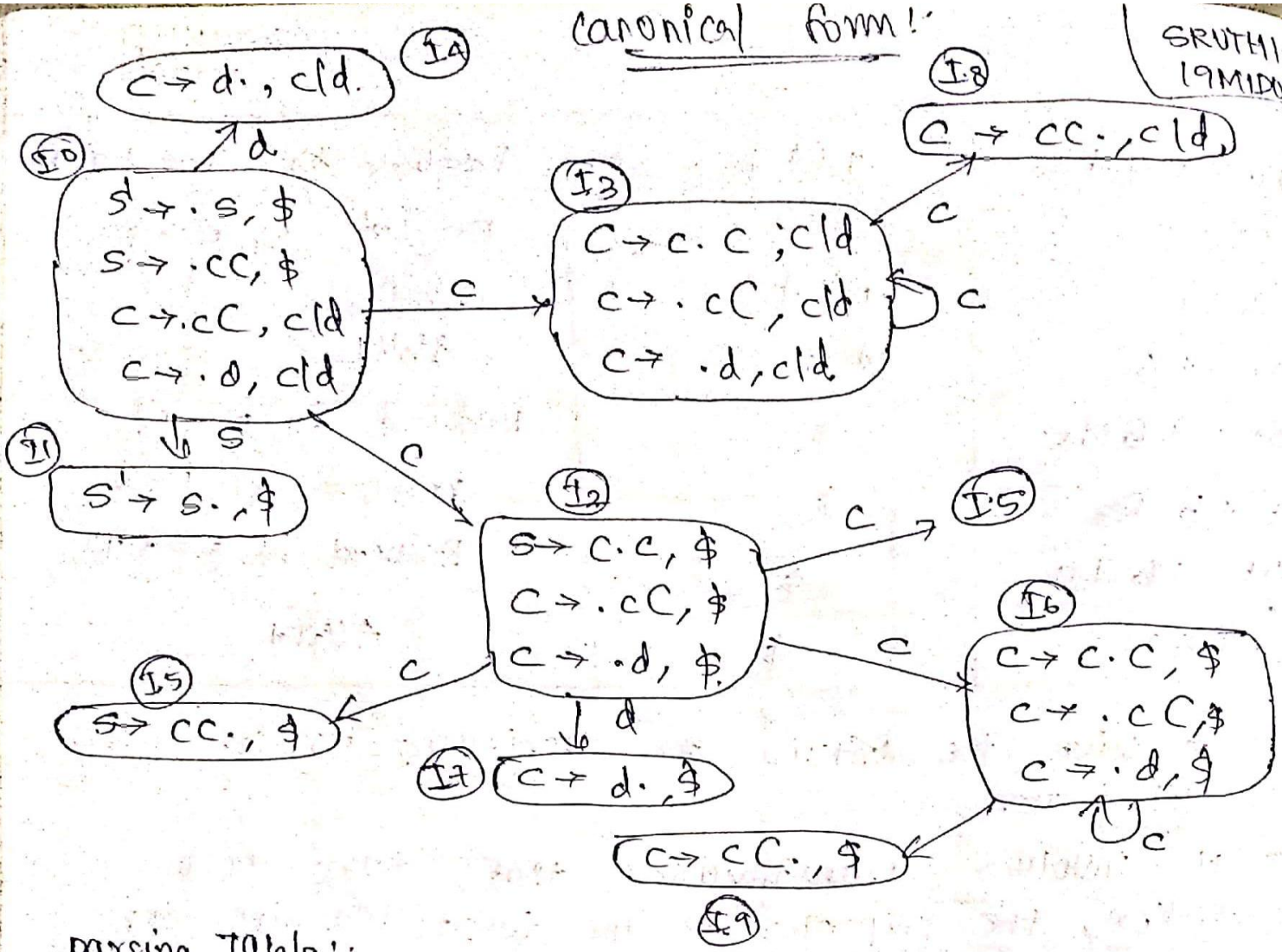
$S \to Cc$
$C \to cC | d.$

Augmented grammer

$S' \to S$
$S \to CC$
$C \to cC | d.$

SRUTHI.S
19MID0052

Canonical form:-



Parsing Table:-

State	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5				R_1	
6	S_6	S_7			9
7					
8	R_2	R_2			
9			R_2		

SRUTHI.S
19MID0053

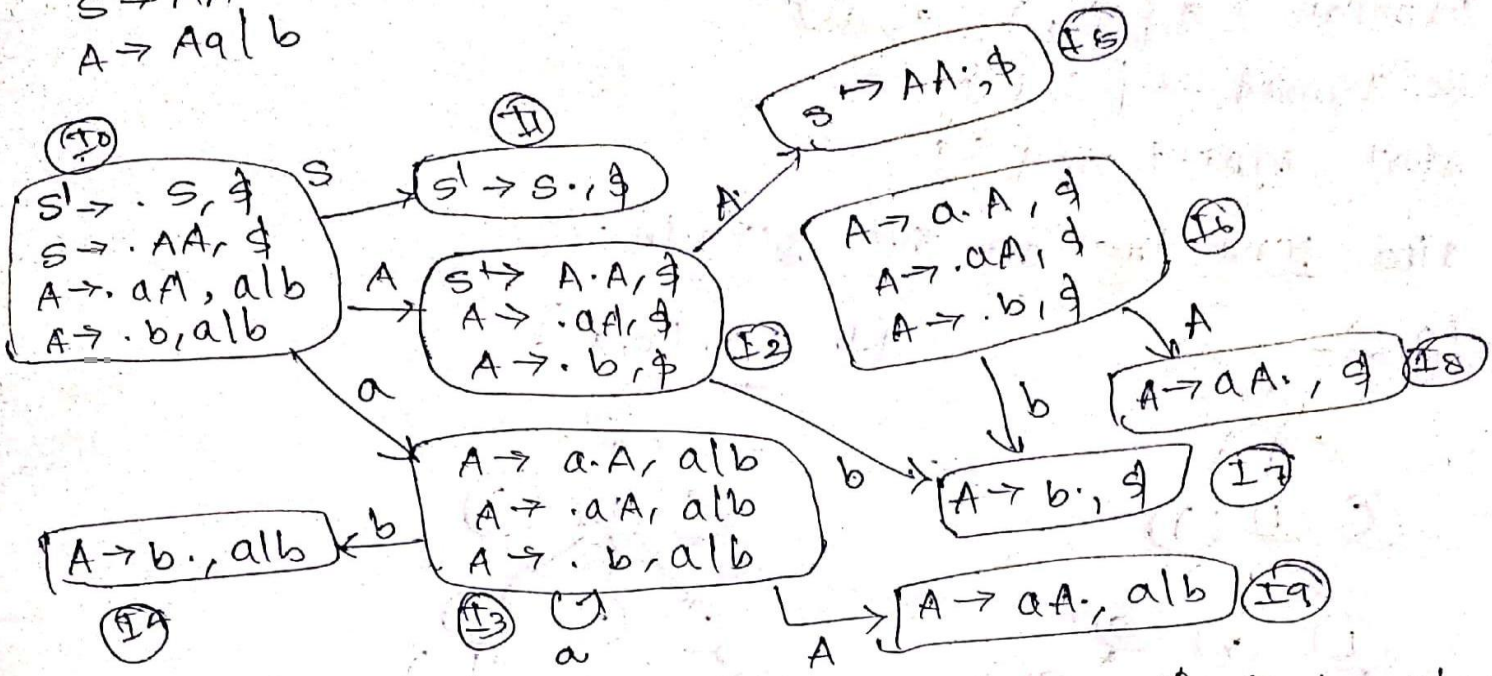
(7) Construct a CLR parsing table

$S \to AA \mid A \to Aa \mid b$

Augmented grammar:

$S' \to S$
$S \to AA$
$A \to Aa \mid b$

Closure $(S' \to \cdot S, \$)$



look ahead symbol gives where reduction is to be placed.

Parsing table:

States	Action			Goto	
	a	b	$	S	A
0				1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			9
4	r_3	r_3			
5					
6	S_6	S_7			8
7					
8					
9	r_2	r_2			

b.) copy propagation:

Idea ⇒ If one variable is assigned to another, replace uses of the assigned variable with the copied variable. Need to know where the copies of the variable.

Example:

```
x=y;                          x=y;
if (x>y) {          →         if (y>1) {
x= x * f(x-1);                x= y * f(y-1);
}                             }
```

can make the first assignment x dead code.

c.) Dead code elimination:

Idea⇒ If a side free statement can never be observed, it is safe to eliminate the statement. A variable dead if it is never used after it is defined.

```
x = y + y   // x is dead?   →   ...
...         // x never used      x = z * z
x = z * z
```

Dead variables are created by optimization

d.) code motion:

SRUTHI.S
191911D0053

Idea → The code motion just means that the code is moved out of the loop as it went have any difference if it is performed inside the loop repeatedly. The compiler by taking the code that doesn't need to be in the loop & making it outside of it. For the optimization process.